Panda3D Manual: Main Page

Search

Table of Contents

I. Introduction to Panda
II. Installing Panda
III. Panda Bootstrap
IV. Introductory Tutorials
    A. A Panda "Hello World"
        1. Starting Panda3D
        2. Loading the Grassy Scenery
        3. Controlling the Camera
        4. Loading and Animating the Panda Model
        5. Using Intervals to move the Panda
V. Programming with Panda
    A. The Scene Graph
        1. Scene Graph Manipulations
        2. Common State Changes
        3. Manipulating a Piece of a Model
        4. Searching the Scene Graph
        5. Render Attributes
        6. Instancing
    B. Panda Filename Syntax
    C. The Configuration File
        1. Accessing Config Vars in a Program
    D. Actors and Characters
        1. Loading Actors and Animations
        2. Actor Animations
        3. Attaching an Object to a Joint
        4. Controlling a Joint Procedurally
    E. Camera Control
        1. The Default Camera Driver
        2. Lenses and Field of View
        3. Orthographic Lenses
    F. Sound
        1. Loading and Playing Sounds and Music
        2. Manipulating Sounds
        3. DSP Effects
        4. 3DAudio
        5. Multi-Channel
    G. Intervals
        1. Lerp Intervals
        2. Function Intervals
        3. Actor Intervals

4. Sound Intervals
5. Motion Path and Particle Intervals
6. Sequences and Parallels
7. Projectile Intervals

H. Tasks and Event Handling
1. Tasks
2. Event Handlers
3. Main Loop

I. Fog and Lighting
1. Fog
2. Lighting
3. Example

J. Text Rendering
1. Text Fonts
2. Text Node
3. OnscreenText
4. Embedded Text Properties

K. DirectGUI
1. DirectButton
2. DirectCheckButton
3. DirectDialog
4. DirectEntry
5. DirectFrame
6. DirectLabel
7. DirectOptionMenu
8. DirectScrolledList
9. DirectWaitBar
10. DirectSlider
11. DirectScrollBar
12. DirectScrolledFrame

L. Render Effects
1. Compass Effects
2. Billboard Effects

M. Texturing
1. Simple Texturing
2. Choosing a Texture Size
3. Texture Wrap Modes
4. Texture Filter Types
5. Simple Texture Replacement
6. Multitexture Introduction
7. Texture Blend Modes
8. Texture Order
9. Texture Combine Modes
10. Texture Transforms
11. Multiple Texture Coordinate Sets
12. Automatic Texture Coordinates
13. Projected Textures
14. Simple Environment Mapping

15. 3-D Textures
16. Cube Maps
17. Environment Mapping with Cube Maps
18. Dynamic Cube Maps
19. Automatic Texture Animation
20. Playing MPG and AVI files
21. Transparency and Blending

N. Pixel and Vertex Shaders
1. Shader Basics
2. List of Possible Shader Inputs
3. Shaders and Coordinate Spaces
4. Known Shader Bugs and Limitations

O. Finite State Machines
1. FSM Introduction
2. Simple FSM Usage
3. FSM with input
4. Advanced FSM Tidbits

P. Advanced operations with Panda's internal structures
1. How Panda3D Stores Vertices and Geometry
a. GeomVertexData
b. GeomVertexFormat
c. GeomPrimitive
d. Geom
e. GeomNode
2. Procedurally Generating 3D Models
a. Defining your own GeomVertexFormat
b. Pre-defined vertex formats
c. Creating and filling a GeomVertexData
d. Creating the GeomPrimitive objects
e. Putting your new geometry in the scene graph
3. Other Vertex and Model Manipulation
a. Reading existing geometry data
b. Modifying existing geometry data
c. More about GeomVertexReader, GeomVertexWriter, and GeomVertexRewriter
d. Creating New Textures from Scratch
e. Writing 3D Models out to Disk
f. Generating Heightfield Terrain

Q. Panda Rendering Process
1. The Graphics Pipe
2. The Graphics Engine
3. The GraphicsOutput class
4. Graphics Buffers and Windows
5. Multi-Pass Rendering
6. Render to Texture
7. How to Control Render Order

R. Panda Utility Functions

S. Particle Effects

1. Using the Particle Panel
2. Particle Effect Basic Parameters
3. Particle Factories
4. Particle Emitters
5. Particle Renderers

T. Collision Detection
1. Collision Solids
2. Collision Handlers
3. Collision Entries
4. Collision Traversers
5. Collision Bitmasks
6. Rapidly-Moving Objects
7. Pusher Example
8. Event Example
9. Bitmask Example
10. Clicking on 3D Objects
11. Example for Clicking on 3D Objects

U. Hardware support
1. Keyboard Support
2. Mouse Support
3. Joystick Support
4. VR Helmets and Trackers
5. Jam-O-Drum

V. Math Engine
1. Matrix Representation

W. Physics Engine
1. Enabling physics on a node
2. Applying physics to a node
3. Types of forces
4. Notes and caveats

X. Motion Paths

Y. Timing
1. The Global Clock

Z. Networking
1. Datagram Protocol
    a. Client-Server Connection
    b. Transmitting Data

VI. Debugging and Performance Tuning
A. The Python Debugger
B. Running Panda under the CXX Debugger
C. Log Messages
D. Measuring Performance with PStats
E. Graphics Card Performance

VII. Panda Tools
A. The Scene Graph Browser
1. Enhanced Mouse Navigation
B. The Scene Editor
C. Python Editors

1. SPE

D. Pipeline Tips

E. Model Export

1. Converting from 3D Studio Max
2. Converting from Maya
3. Converting from Blender
4. Converting from SoftImage
5. Converting from Milkshape 3D
6. Converting from GMax
7. Converting from other Formats
8. Converting Egg to Bam
9. Parsing and Generating Egg Files

F. Previewing 3D Models in Pview

G. Building a Self-Extracting EXE using packpanda

VIII. Building Panda from Source

A. Troubleshooting ppremake on Windows

B. Troubleshooting ppremake on Linux

C. Troubleshooting makepanda on Windows

D. Troubleshooting makepanda on Linux

IX. Video Lectures

A. Disney Video Lectures

B. Scene Editor Lectures

C. Panda 3D Video Tutorial Series

X. API Reference Materials

XI. List of Panda Executables

XII. More Panda Resources

XIII. FAQ

XIV. Examples Contributed by the Community

XV. Start Guide For The Absolute Beginner

Search

Panda3D Manual: Introduction to Panda

Panda3D is a *3D engine*: a library of subroutines for 3D rendering and game development. The library is C++ with a set of Python bindings. Game development with Panda3D usually consists of writing a Python program that controls the Panda3D library.

Panda3D is unusual in that its design emphasis is on supporting a short learning curve and rapid development. It is ideal whenever deadlines are tight and turnaround time is of the essence.

For example, in a class called Building Virtual Worlds at the Entertainment Technology Center, interdisciplinary groups of four students are asked to create virtual worlds in two weeks each. Screenshots of their projects are visible throughout this site. Panda3D is what makes this rapid turnaround possible.

Panda3D was developed by Disney for their massively multiplayer online game, Toontown. It was released as free software in 2002. Panda3D is now developed jointly by Disney and Carnegie Mellon University's Entertainment Technology Center.

Panda3D's Free Software License

Panda3D has a very simple License, which classifies as a free software license. That means that with few restrictions, anyone is free to download and use Panda3D at will: for commercial purposes, for teaching, or most any other use. Also importantly, anyone may view, use, and alter the source code. This allows for a strong community to work together to improve the engine.

Who is Working on Panda3D

There are a number of developers in the commercial and open-source community. Currently, the two most active members of the development community are Disney and the Entertainment Technology Center at Carnegie Mellon. Because both organizations have specific goals, Panda3D must necessarily serve both:

- Disney's primary interest in Panda3D is commercial. Panda3D is being used in the development of a number of Disney games and amusement-park exhibits. To serve Disney's needs, Panda3D must be a fully-featured engine, capable of all the performance and quality one expects in any 'A-grade' commercial title.
- The Entertainment Technology Center's primary goal is education. To serve the Entertainment Technology Center's needs, Panda3D must be well-suited for use in student projects. Since students have a unique talent for causing crashes, bulletproof reliability is needed. Since projects only last one semester, the learning curve must be very short, and prototyping must be very rapid.

As it turns out, the two sets of goals are complementary. The rapid development and high reliability needed by the Entertainment Technology Center are also highly advantageous in a game-development studio, since they lower development time and costs. The good visual

quality and full feature set needed by Disney to make a professional-quality game also turn out to be useful in a university setting: with a broad range of features at their disposal, students can explore their creativity more fully than they could with a more limited engine.

<<prev top next>> Search

Panda3D Manual: Installing Panda

Panda3D comes as one file that includes everything that is needed to create and run Panda3D applications. The tutorial, examples, models, and animations can be found in subdirectories after the installation of Panda3D.

For panda programming practice, many Creative Commons Licensed egg files of models, animations, and materials, all in many .zip files are available in the 3D Model Archive: 3D Models

The Installation Process - Windows

If you have already installed panda previously, you should uninstall it before installing a new version.

Your next step is to download the "Windows Installer" from the download page. Run the installer, and follow the prompts. Next, proceed to the "Testing the Installation" section below.

The Installation Process - Linux

The easiest way to install panda is to use the RPM or DEB packages. This is only possible if your version of Linux is one of the provided versions of Linux. If not, you will need to compile from source. If there is an installer available, download and install the RPM or DEB appropriate to your version of Linux.

It has been discovered that some of the DEB and RPM files work on versions other than the one for which they were intended. For example, the package for Debian Sarge has also been found to work with Debian Sid. If you have a slight variant of one of the supplied operating systems, it may be easier to try the package before you bother with compiling panda yourself.

RPM files may be installed with the following command:

```
rpm -i filename.
rpm
```

DEB packages may be installed with this command:

```
dpkg -i filename.
deb
```

Where filename is the name of the file you downloaded. You need to be root to do either installation.

Testing the Installation

Panda comes with a program called "Greeting Card" that can be used to verify that panda is working correctly.

If you are using Windows, the installer will run the greeting card for you. If you wish, you can run it again from the start menu.

If you are using Unix, you need to do it manually. Change directory to `samples/GreetingCard` and run `ppython GreetingCard.py`. The manual procedure works under Windows as well.

Troubleshooting

If the test programs don't run, then usually, you need to update your video drivers.

If you are using Linux, your one useful step should be to install a small game called "Tux Racer" - this game is included with most versions of Linux. This is a very simple OpenGL game, not written in Panda. It is useful because it will tell you if OpenGL is working correctly. If Tux Racer works (and runs fast), then Panda should work.

If you are using Windows, you have a choice between OpenGL and DirectX. Panda3D, by default, is configured to use OpenGL. If you have a video card that doesn't support OpenGL properly, you can try DirectX instead, by editing the panda configuration file.

If neither of these works, please report the problem to the Panda3D maintainers, using the Panda3D forums.

Troubleshooting with "Path Error Msg" for Windows Users

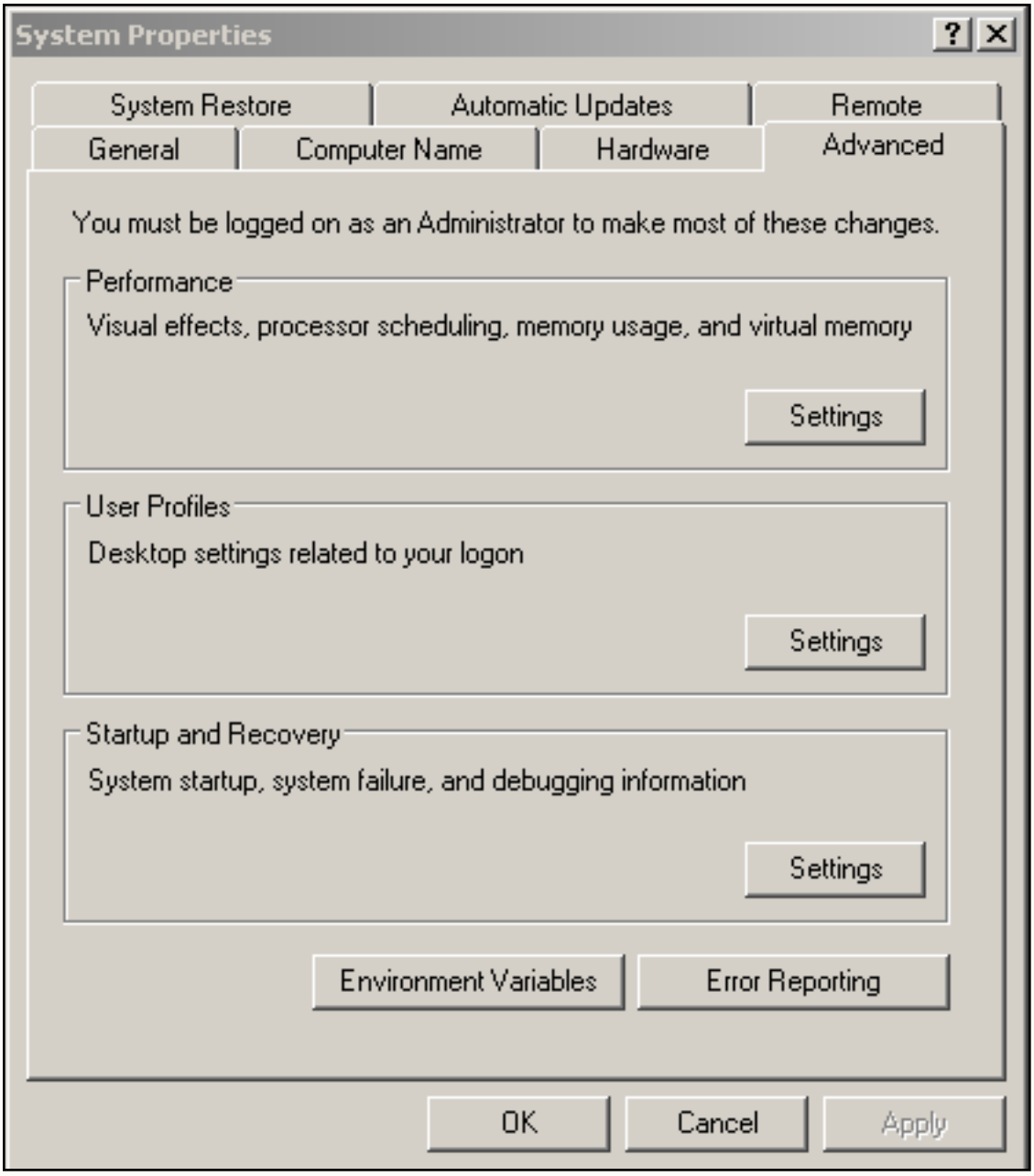
Sometimes, even after running the Panda installer, you will be unable to use the Panda3d python interpreter PPython.exe from the command line. You would receive message like "Ppython is not a recognised internal command" when typing "ppython yourPanda3Dscript.py"

Ok, on Window there is what is called the PATH. this is variable that hold predefined path for some exe you want to be able to use from any folder without having to type the whole path.

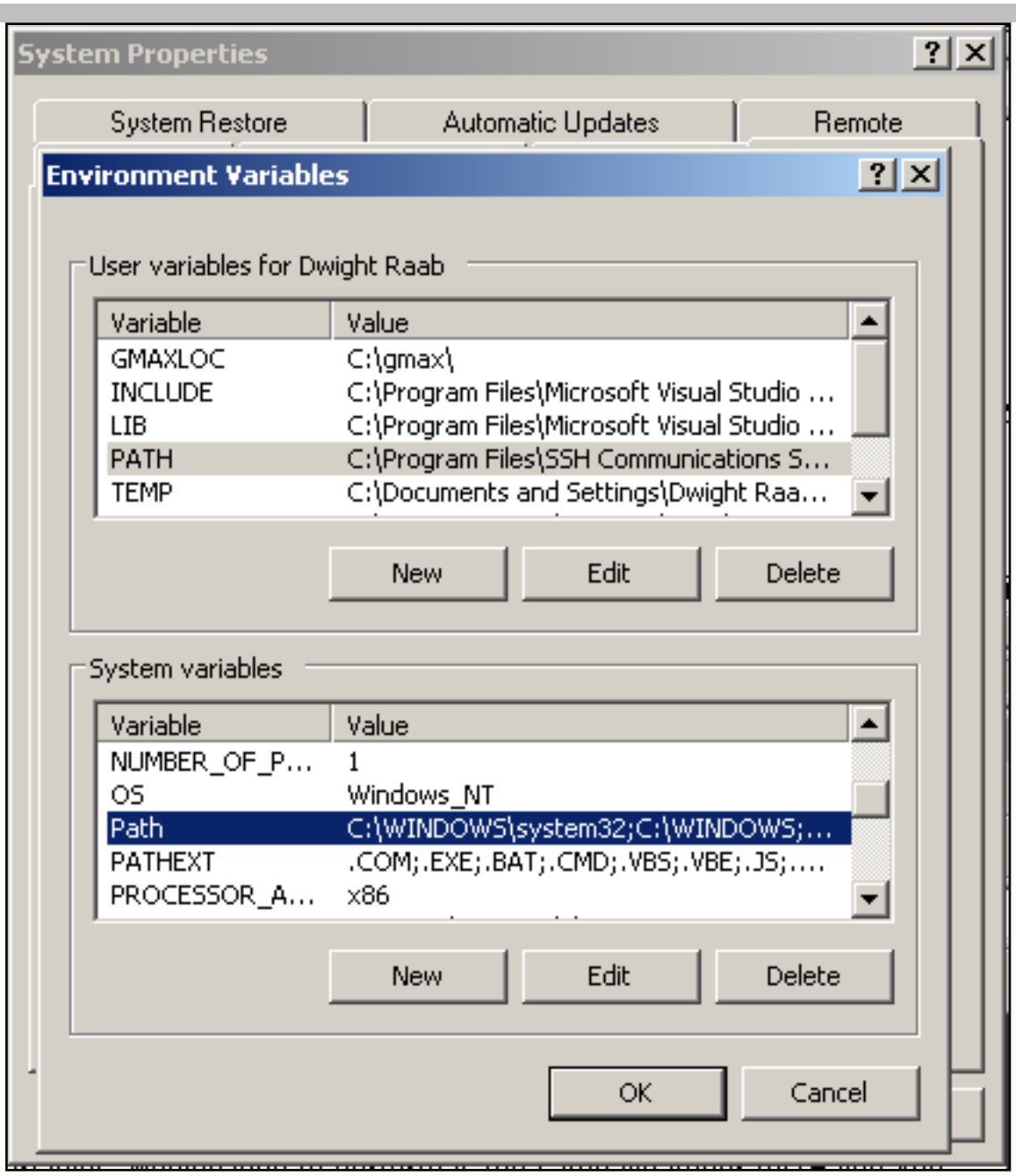
For example if you installed Panda3D in C:\Panda3D then , since ppython (the panda3D executable) is located in C:\Panda3D\bin, putting C:\Panda3d\bin in the path will allow you to use" ppython myPanda3dScript.py from any folder of your PC without having to type the full path ie "C:\Panda3d\bin\ppython.exe " myPanda3dScript.py.

How to put the path?

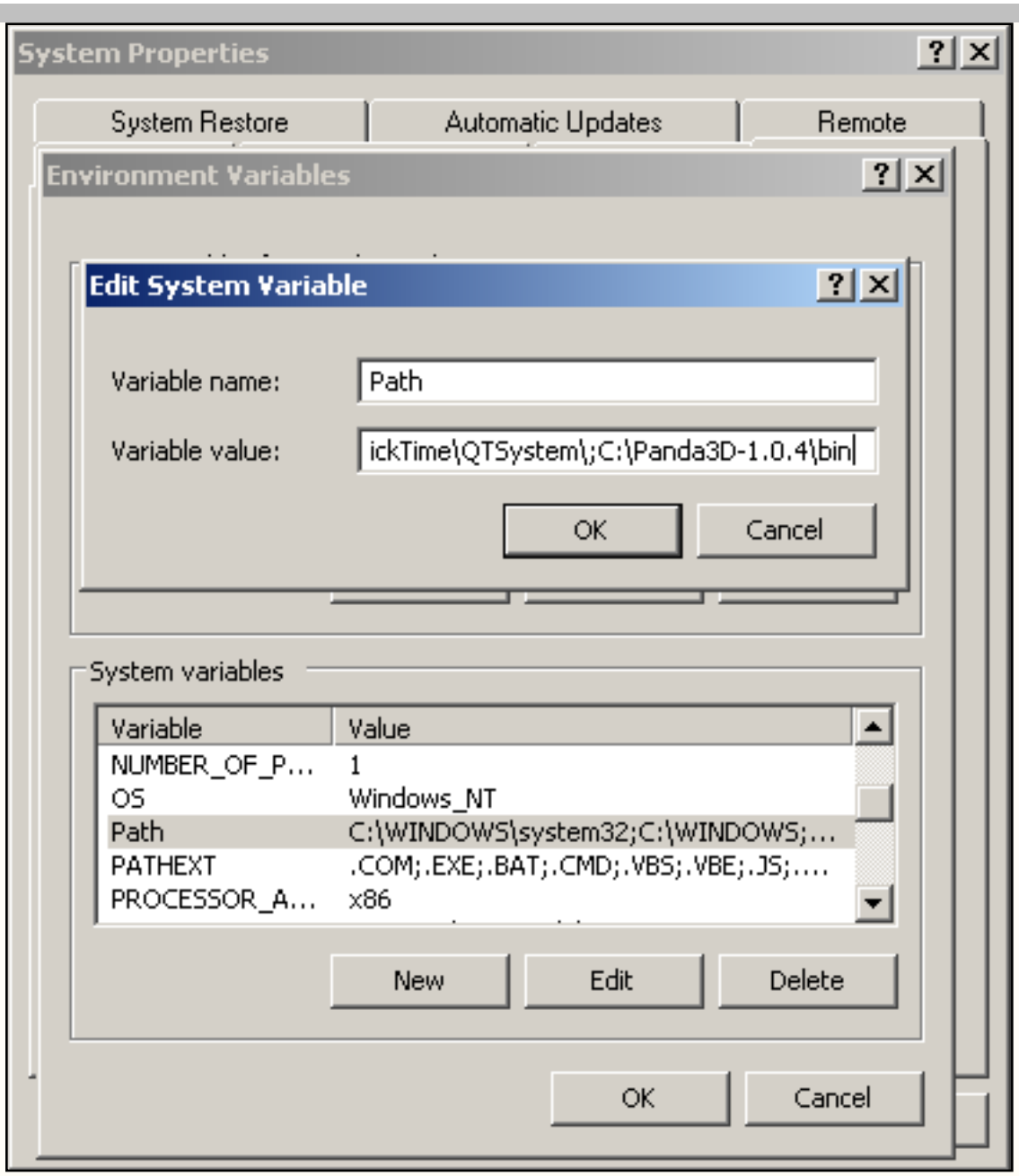
1)You have to make a right click on Workspace icon on your desktop and select Properties. Then you choose the tab "Advanced". Then you click the Button "Environnment Settings". (second button in the pane on Win2K).



2)Then if you are "Admin" on your PC , select the PATH line in the list under System Variables , else select the PATH line in the list under "MyUserName" Variables. Modification to perform is the same whatever PATH line you selected.



3)Now double click the PATH (or Path) line. a small window opens with Variable Name=Path and Variable Value = some folder paths separated by a ";". In there , just append "C:\panda3D \bin;" at the end of the line. NB: this assume you installed Panda3D in C:\Panda3D else just take the real path to your Panda3D\bin Folder.



4) Validate by OK to close the small window then OK to close the Properties Panel.

5) Log off and Log on from your session (or reboot the PC), that's it!!!

Writing your First Program

Finally, it will be time to try writing your own program using the Panda3D library. The Introductory Tutorials section will guide you through the process.

Panda3D Manual: Panda Bootstrap

Though Panda3D is easy enough, it needs some getting used to before you can understand the manuals and examples.

Here are a few links, which should make understanding Panda3D easier:

- http://www.python.org/pycon/dc2004/papers/29/ This link gives you a very broad overview of panda, which is helpful to more clearly understand the Introductory examples.
- http://www.etc.cmu.edu/bvw/scripting/index.html (Note, currently not functional with Panda3D-1.1.0)
  Read this example once you are done with the introduction example following this section. I found referring to the manual helpful and informative while working on this example.
- The samples that come with the installation are also very informative and well commented. They should be really helpful for people new to Panda3D.
- Have the Panda3D and BVW quick references handy while you are coding.
- http://www.gamasutra.com/features/20040128/goslin_pfv.htm A postmortem of ToonTown built with Panda3D.

Having the API reference handy while reading the manual is also advisable, as the manual is lacking explanations at times.

Panda3D Manual: Introductory Tutorials

This section of the manual includes some basic tutorials that will teach you the fundamentals of using the Panda3D library.

Since all of these tutorials require you to use the Python scripting language, it would be a good idea to familiarize yourself with Python before continuing.

Python is an interpreted, interactive, object-oriented language comparable to Java or Perl. It is available on several platforms, including UNIX, Windows, OS/2, and Mac. Python also has a large number of modules outside of the standard Python installation, and additional modules can be created in C or C++. Because it is late-binding and requires minimal memory management, it is an ideal language for rapid prototyping.

Panda 3D comes with a version of python that has all of the panda libraries hooked up to it.

To use Panda's python in windows:

1. Make sure the \bin directory inside your panda installation is on your PATH environment variable.

2. Go into a command prompt and type: *ppython <your program>*

There are a lot of other resources available for programming in Python. Here is a list of some of the best:

Links from the official python website:

- Official Website - http://www.python.org
- Current Python Documentation
- Python Documentation This is the version still being used in Panda3D.
- Python Tutorial. Written by Guido Van Rossum, the author of Python.

Here are some other good links for learning python:

- Programming in Python
  - Byte of Python
  - Python for Non-Programmers
  - Dive Into Python
  - Python 101
- Miscellaneous python documentation
  - Python Examples and Sample Code
  - The Standard Python Library
  - Python Book List

Panda3D Manual: A Panda "Hello World"

This tutorial is called "A Panda Hello World." It is a typical example of a simple Panda3D program. Walking through this tutorial will enable you to obtain some limited familiarity with the Panda3D API, without having to learn the entire thing.

The program that we are going to create will load up a small scene containing some grass and a panda. The panda will be animated to walk back and forth over the grass.

Panda3D Manual: Starting Panda3D

To start Panda3D, create a text file and save it with the .py extension. PYPE and IDLE are Python-specific text-editors, but any text editor will work. Enter the following text into your Python file:

```
import direct.directbase.
DirectStart
run()
```

DirectStart loads most of the other Panda3D modules, and causes the 3D window to appear. The *run* subroutine contains the Panda3D main loop. It renders a frame, handles the background tasks, and then repeats. It does not normally return, so it only needs to be called once and must be the last line in your script. In this particular example, there will be nothing to render, so you should expect a window containing an empty grey area.

To run your program, type this at the command prompt:

```
ppython filename.
py
```

If Panda3D has been installed properly, a gray window titled Panda appear. There is nothing we can do with this window, but that will change shortly.

Panda3D Manual: Loading the Grassy Scenery

With Panda3D running properly, it is now possible to load some grassy scenery. Update your code as follows:

```
import direct.directbase.DirectStart

#Load the first environment model
environ = loader.loadModel("models/
environment")
environ.reparentTo(render)
environ.setScale(0.25,0.25,0.25)
environ.setPos(-8,42,0)
#Run the tutorial
run()
```

The command `loader.loadModel()` loads the specified file, in this case the environment.egg file in the models folder. The return value is a 'NodePath', effectively a pointer to the model. Note that Panda Filename Syntax uses the forward-slash, even under Windows.

Panda3D contains a data structure called the *scene graph*. The scene graph is a tree containing all objects that need to be rendered. At the root of the tree is an object named `render`. Nothing is rendered until it is first installed into the scene graph.

To install the grassy scenery model into the scene graph, we use the method `reparentTo`. This sets the parent of the model, thereby giving it a place in the scene graph. Doing so makes the model visible.

Finally, we adjust the position and scale of the model. In this particular case, the environment model is a little too large and somewhat offset for our purposes. The `setScale` and `setPosition` rescale and recenter the model.

Go ahead and run the program. You should see this:

The rock and tree appear to be hovering. The camera is slightly below ground, and backface culling is making the ground invisible to us. If we reposition the camera, the terrain will look better.

<<prev top next>> Search

Panda3D Manual: Controlling the Camera

By default, Panda3D runs a task that enables you to move the camera using the mouse. The keys to do this are:

- Left Button: pan left and right
- Right Button: move forwards and backwards
- Middle Button: rotate around the origin of the application
- Right and Middle Buttons: roll the point of view around the view axis

Go ahead and try this camera control system. The problem with this camera control system is that it is sometimes awkward, it is not always easy to get the camera pointed in the direction we want.

Instead, we are going to write a *task* that controls the camera's position explicitly. A *task* is nothing but a subroutine that gets called every frame. Update your code as follows:

```
import direct.directbase.DirectStart
from direct.task import Task
from direct.actor import Actor
import math

#Load the first environment model
environ = loader.loadModel("models/environment")
environ.reparentTo(render)
environ.setScale(0.25,0.25,0.25)
environ.setPos(-8,42,0)

#Task to move the camera
def SpinCameraTask(task):
  angledegrees = task.time * 6.0
  angleradians = angledegrees * (math.pi / 180.0)
  base.camera.setPos(20*math.sin(angleradians),-20.0*math.cos
(angleradians),3)
  base.camera.setHpr(angledegrees, 0, 0)
  return Task.cont

taskMgr.add(SpinCameraTask, "SpinCameraTask")

run()
```

The function `taskMgr.add` tells the panda *task manager* that it should call the subroutine `SpinCameraTask` every frame. This is a subroutine that we have written to control the camera. As long as the subroutine `SpinCameraTask` returns the constant `Task.cont`, the task manager will continue to call it every frame.

The subroutine calculates the desired position of the camera based on how much time has

elapsed. The camera rotates 6 degrees every second. The first two lines compute the desired orientation of the camera, first in degrees, then in radians. The `setPos` call actually sets the position of the camera. The `setHpr` call actually sets the orientation.

The camera should no longer be underground, and furthermore, the camera should now be rotating about the clearing:

<<prev top next>> Search

Panda3D Manual: Loading and Animating the Panda Model

Now that the scenery is in place, we will now load an actor. Update your code to look like this:

```
import direct.directbase.DirectStart
from direct.task import Task
from direct.actor import Actor
import math

#Load the first environment model
environ = loader.loadModel("models/environment")
environ.reparentTo(render)
environ.setScale(0.25,0.25,0.25)
environ.setPos(-8,42,0)

#Task to move the camera
def SpinCameraTask(task):
  angledegrees = task.time * 6.0
  angleradians = angledegrees * (math.pi / 180.0)
  base.camera.setPos(20*math.sin(angleradians),-20.0*math.cos
(angleradians),3)
  base.camera.setHpr(angledegrees, 0, 0)
  return Task.cont

taskMgr.add(SpinCameraTask, "SpinCameraTask")

#Load the panda actor, and loop its animation
pandaActor = Actor.Actor("models/panda-model",{"walk":"models/panda-walk4"})
pandaActor.setScale(0.005,0.005,0.005)
pandaActor.reparentTo(render)
pandaActor.loop("walk")

run()
```

The `Actor` class is for animated models. Note that we use `loadModel` for static models, and `Actor` only when they are animated. The two constructor arguments for the `Actor` class are the name of the file containing the model, and a Python dictionary containing the names of the files containing the animations.

The command `loop("walk")` causes the walk animation to begin looping. The result is a panda walking in place, as if on a treadmill:

<<prev top next>>

Search

Panda3D Manual: Using Intervals to move the Panda

The next step is to cause the panda to actually move back and forth. Add the following lines of code:

```
import direct.directbase.DirectStart
from pandac.PandaModules import *

from direct.task import Task
from direct.actor import Actor
from direct.interval.IntervalGlobal import *
import math

#Load the first environment model
environ = loader.loadModel("models/environment")
environ.reparentTo(render)
environ.setScale(0.25,0.25,0.25)
environ.setPos(-8,42,0)

#Task to move the camera
def SpinCameraTask(task):
  angledegrees = task.time * 6.0
  angleradians = angledegrees * (math.pi / 180.0)
  base.camera.setPos(20*math.sin(angleradians),-20.0*math.cos(angleradians),3)
  base.camera.setHpr(angledegrees, 0, 0)
  return Task.cont

taskMgr.add(SpinCameraTask, "SpinCameraTask")

#Load the panda actor, and loop its animation
pandaActor = Actor.Actor("models/panda-model",{"walk":"models/panda-walk4"})
pandaActor.setScale(0.005,0.005,0.005)
pandaActor.reparentTo(render)
pandaActor.loop("walk")

#Create the four lerp intervals needed to walk back and forth
pandaPosInterval1= pandaActor.posInterval(13,Point3(0,-10,0), startPos=Point3
(0,10,0))
pandaPosInterval2= pandaActor.posInterval(13,Point3(0,10,0), startPos=Point3
(0,-10,0))
pandaHprInterval1= pandaActor.hprInterval(3,Point3(180,0,0), startHpr=Point3
(0,0,0))
pandaHprInterval2= pandaActor.hprInterval(3,Point3(0,0,0), startHpr=Point3
(180,0,0))

#Create and play the sequence that coordinates the intervals
pandaPace = Sequence(pandaPosInterval1, pandaHprInterval1,
  pandaPosInterval2, pandaHprInterval2, name = "pandaPace")
pandaPace.loop()

run()
```

Intervals are tasks that change a property from one value to another over a specified period of time. Starting an interval effectively starts a background process that modifies the property over the specified period of time. For example, consider the `pandaPosInterval1` above. When that interval is started, it will gradually adjust the position of the panda from (0,10,0) to (0,-10,0) over a period of 13 seconds. Similarly, when the `pandaHprInterval1` is started, the orientation of the panda will rotate 180 degrees over a period of 3 seconds.

Sequences are tasks that execute one interval after another. The `pandaPace` sequence above causes the panda to move in a straight line, then turn, then in the opposite straight line, then to turn again. The code `pandaPace.loop()` causes the Sequence to be started in looping mode.

The result of all this is to cause the panda to pace back and forth from one tree to the other.

<<prev top next>> Search

Panda3D Manual: Programming with Panda

The *programming with panda* section of the manual is designed to teach the basic concepts associated with using panda.

Panda3D Manual: The Scene Graph

The Scene Graph: a Tree of Nodes

Many simple 3D engines maintain a list of 3D models to render every frame. In these simple engines, one must allocate a 3D model (or load it from disk), and then insert it into the list of models to render. The model is not "visible" to the renderer until it is inserted into the list.

Panda3D is slightly more sophisticated. Instead of maintaining a list of objects to render, it maintains a tree of objects to render. An object is not visible to the renderer until it is inserted into the tree.

The tree consists of objects of class `PandaNode`. This is actually a superclass for a number of other classes: `ModelNode`, `GeomNode`, `LightNode`, and so forth. Throughout this manual, it is common for us to refer to objects of these classes as simply *nodes*. The root of the tree is a node called `render`. (Note: there may be additional roots for specialized purposes.)

Panda3D's "tree of things to render" is named the *scene graph.*

What you Need to Know about the Hierarchical Scene Graph

Here are the most important things you need to know about the hierarchical arrangement of the scene graph:

1. You control where objects go in the tree. When you insert an object into the tree, you specify where to insert it. You can move branches of the tree around. You can make the tree as deep or as shallow as you like.
2. Positions of objects are specified relative to their parent in the tree. For example, if you have a 3D model of a hat, you might want to specify that it always stays five units above a 3D model of a certain person's head. Insert the hat as a child of the head, and set the position of the hat to (0,0,5).
3. When models are arranged in a tree, any *rendering attributes* you assign to a node will propagate to its children. For example, if you specify that a given node should be rendered with depth fog, then its children will also be rendered with depth fog, unless you explicitly override at the child level.
4. Panda3D generates bounding boxes for each node in the tree. A good organizational hierarchy can speed frustum and occlusion culling. If the bounding box of an entire branch is outside the frustum, there is no need to examine the children.

Beginners usually choose to make their tree completely flat--everything is inserted immediately beneath the root. This is actually a very good initial design. Eventually, you will find a reason to want to add a little more depth to the hierarchy. But it is wise not to get complicated until you have a clear, specific reason to do so.

NodePaths

There is a helper class called NodePath which is a very small object containing a pointer to a node, plus some administrative information. For now, you can ignore the administrative information; it will be explained in a later section of the manual. It is the intent of the panda designers that you should think of a NodePath as a handle to a node. Any function that creates a node returns a NodePath that refers to the newly-created node.

A NodePath isn't exactly a pointer to a node; it's a "handle" to a node. Conceptually, this is almost a distinction without a difference. However, there are certain API functions that expect you to pass in a NodePath, and there are other API functions that expect you to pass in a node pointer. Because of this, although there is little conceptual difference between them, you still need to know that both exist.

You can convert a NodePath into a "regular" pointer at any time by calling nodePath.node(). However, there is no unambiguous way to convert back. That's important: sometimes you *need* a NodePath, sometimes you *need* a node pointer. Because of this, it is recommended that you store NodePaths, not node pointers. When you pass parameters, you should probably pass NodePaths, not node pointers. The callee can always convert the NodePath to a node pointer if it needs to.

NodePath-Methods and Node-Methods

There are many methods that you can invoke on nodepaths, which are appropriate for nodes of any type. Specialized node types, like LODNodes and Cameras (for instance), provide additional methods that are available only for nodes of that type, which you must invoke on the node itself. Here are some assorted examples:

```
# NODEPATH METHODS:
myNodePath.setPos(x,y,z)
myNodePath.setColor(banana)

# LODNODE METHODS:
myNodePath.node().addSwitch(1000, 100)
myNodePath.node().setCenter(Point(0, 5, 0))

# CAMERA NODE METHODS:
myNodePath.node().setLens(PerspectiveLens
())
myNodePath.node().getCameraMask()
```

Always remember: when you invoke a method of NodePath, you are actually performing an operation on the node to which it points.

In the example above, we call node-methods by first converting the nodepath into a node, and then immediately calling the node-method. This is the recommended style.

Panda3D Manual: Scene Graph Manipulations

The default scene graphs

By default, there are two different scene graphs created automatically when you start up Panda3D. These graphs are referred to by their top nodes: **render** and **render2d**.

You use render most often; this is the top of the ordinary 3-D scene. In order to put an object in the world, you will need to parent it to render (or to some node that is in turn parented to render).

You will use render2d to render 2-D GUI elements, such as text or buttons, that you want to display onscreen; for instance, a heads-up display. Anything parented to render2d will be rendered on top of the 3-D scene, as if it were painted on the screen glass.

The coordinate system of render2d is set up to match that of the mouse inputs: the lower-left corner of the screen is (-1, 0, -1), and the upper-right corner is (1, 0, 1). Since this is a square coordinate system, but the screen is usually non-square, objects parented directly to render2d may appear squashed. For this reason, Panda3D also defines a child of render2d, called **aspect2d**, which has a scale applied to it to correct the non-square aspect ratio of render2d. Most often, you will parent GUI elements to aspect2d rather than render2d.

Finally, you may see references to one other top-level node called **hidden**. This is simply an ordinary node that has no rendering properties set up for it, so that things parented to hidden will not be rendered. Older Panda3D code needed to use hidden to remove a node from the render scene graph. However, this is no longer necessary, and its use is not recommended for new programs; the best way to remove a node from render is to call nodePath.detachNode().

Loading models

You can load up a model with a filename path, in the Panda Filename Syntax, to the model's egg or bam file. In many examples, the filename extension is omitted; in this case, Panda will look for a file with either the .egg or .bam extension.

```
myNodePath = loader.loadModel('my/path/to/models/myModel.
egg')
```

If you want to load multiple copies of a particular model, you can avoid re-reading the disk each time:

```
teapot1 = loader.loadModelCopy('teapot.
egg')
teapot2 = loader.loadModelCopy('teapot.
egg')
teapot3 = loader.loadModelCopy('teapot.
egg')
```

The first time you call loadModelCopy() for a particular model, that model is read and saved in a table in memory; on each subsequent call, the model is simply copied from the table, instead of reading the file.

The above calls are appropriate for loading static models; for animated models, see Loading Actors and Animations.

Reparenting nodes and models

One of the most fundamental scene graph manipulations is changing a node's parent. You need to do this at least once after you load a model, to put it under render for viewing:

```
myModel.reparentTo
(render)
```

And to remove it again:

```
myModel.detachNode
()
```

As you become more comfortable with scene graph operations, you may find yourself taking more and more advantage of a deeply nested scene graph, and you may start to parent your models to other nodes than just render. Sometimes it is convenient to create an empty node for this purpose, for instance, to group several models together:

```
dummyNode = render.attachNewNode("Dummy Node
Name")
myModel.reparentTo(dummyNode)
myOtherModel.reparentTo(dummyNode)
```

Since a node inherits its position information from its parent node, when you reparent a node in the scene graph you might inadvertently change its position in the world. If you need to avoid this, you can use a special variant on reparentTo():

```
myModel.wrtReparentTo
(newParent)
```

The "wrt" prefix stands for "with respect to". This special method works like reparentTo(), except that it automatically recomputes the local transform on myModel to compensate for the change in transform under the new parent, so that the node ends up in the same position relative to the world.

Note that the computation required to perform wrtReparentTo() is a floating-point matrix computation and is therefore inherently imprecise. This means that if you use wrtReparentTo() repeatedly, thousands of times on the same node, it may eventually accumulate enough numerical inaccuracies to introduce a slight scale on the object (for instance, a scale of 1, 1, 0.99999); if left unchecked, this scale could eventually become noticeable.

Beginners tend to overuse this method; you should not use wrtReparentTo() unless there is a real reason to use it.

<<prev top next>> Search

Panda3D Manual: Common State Changes

Two of the most common changes are position and orientation.

```
myNodePath.setPos(X,Y,
Z)
myNodePath.setHpr(H,P,
R)
```

By default in Panda3D, the X axis points to the right, the Y axis is forward, and Z is up. An object's rotation is usually described using Euler angles called Heading, Pitch, and Roll (sometimes called Yaw, Pitch, and Roll in other packages)--these specify angle rotations in degrees. (If you are more comfortable using quaternions, the `setQuat()` method can be used to specify the rotation as a quaternion.)

You can change an object's size, either uniformly, or with a different value of x, y, and z.

```
myNodePath.setScale(uniform)
myNodePath.setScale(SX,SY,
SZ)
```

Sometimes it is convenient to adjust a single component individually:

```
myNodePath.setX(X)
myNodePath.setY(Y)
myNodePath.setZ(Z)
myNodePath.setH(H)
myNodePath.setP(P)
myNodePath.setR(R)
myNodePath.setSx
(SX)
myNodePath.setSy
(SY)
myNodePath.setSz
(SZ)
```

Or all at the same time:

```
myNodePath.setPosHprScale(X,Y,Z,H,P,R,SX,SY,
SZ)
```

You can also query the current transform information for any of the above:

```
myNodePath.getPos
()
myNodePath.getX()
myNodePath.getY()
myNodePath.getZ()
```

Also, by using the functions `setTag()` and `getTag()` you can store your own information in key value pairs. For example:

```
myNodePath.setTag('Key', 'Value')
object=myNodePath.getTag('Key') #returns
'Value'
```

As a more advanced feature, you may also set or query the position (or any of the above transform properties) of a particular NodePath with respect to another one. To do this, specify the relative NodePath as the first parameter:

```
myNodePath.setPos(otherNodePath, X, Y,
Z)
myNodePath.getPos(otherNodePath)
```

Putting a NodePath as the first parameter to any of the transform setters or getters makes it a relative operation. The above `setPos()` means to set myNodePath to the position (X, Y, Z), relative to otherNodePath--that is, the position myNodePath would be in if it were a child of otherNodePath and its position were set to (X, Y, Z). The `getPos()` call returns the position myNodePath would have if it were a child of otherNodePath.

It also important to note that you can use the NodePath in its own relative sets and gets. This maybe helpful in situations where you are concerned with distances. For example:

```
# if you want to move myNodePath 3 units foward in the x
myNodePath.setPos(myNodePath, 3, 0, 0)
```

These relative sets and gets are a very powerful feature of Panda's scene graph, but they can also be confusing; don't worry if it doesn't make sense right now.

The `lookAt()` method rotates an model to face another object; that is, it rotates the first object so that its +Y axis points toward the second object. Note that a particular model might

or might not have been generated with the +Y axis forward, so this doesn't necessarily make a model "look at" the given object.

```
myNodePath.lookAt
(otherObject)
```

Color changes are another common alteration. Values for color are floating point numbers from 0 to 1, 0 being black, 1 being white.

```
myNodePath.setColor(R,G,B,
A)
```

If models have textures, they may not be distinguishable or even visible at certain color settings. Setting the color to white may restore the visibility of the texture, but it is better to simply clear the current color settings.

```
myNodePath.clearColor
()
```

Note the fourth component of color is alpha. This is usually used to indicate transparency, and it is usually 1.0 to indicate the object is not transparent. If you set the alpha to a value between 0 and 1, you can fade the object to invisible. However, in order for the alpha value to be respected, you must first enable transparency:

```
myNodePath.setTransparency(TransparencyAttrib.
MAlpha)
```

The parameter to setTransparency() is usually `TransparencyAttrib.MAlpha`, which is ordinary transparency. You can also explicitly turn transparency off with `TransparencyAttrib.MNone`. (Other transparency modes are possible, but that is a more advanced topic. Some older code may pass just 0 or 1 for this parameter, but it is better to name the mode.) If you don't explicitly enable transparency first, the alpha component of color may be ignored. Be sure you don't enable transparency unnecessarily, since it does enable a more expensive rendering mode.

Setting an object's color completely replaces any color on the vertices. However, if you have created a model with per-vertex color, you might prefer to modulate the object's color without losing the per-vertex color. For this there is the `setColorScale()` variant, which multiples the indicated color values by the object's existing color:

```
myNodePath.setColorScale(R,G,B,
A)
```

One use of setColorScale() is to apply it at the top of the scene graph (e.g. render) to darken the entire scene uniformly, for instance to implement a fade-to-black effect.

Since alpha is so important, there is also a method for scaling it without affecting the other color components:

```
myNodePath.setAlphaScale
(SA)
```

To temporarily prevent an object from being drawn, use `hide()` and `show()`:

```
myNodePath.hide
()
myNodePath.show
()
```

Any object that is parented to the object that is hidden will also be hidden.

<<prev top next>> Search

Panda3D Manual: Manipulating a Piece of a Model

Every model, when loaded, becomes a `ModelNode` in the scene graph. Beneath the `ModelNode` are one or more `GeomNodes` containing the actual polygons. If you want to manipulate a piece of a model, for instance, if you want to change the texture of just part of a model, you need a pointer to the relevant geomnode.

In order to obtain such a pointer, you must first ensure that the relevant geometry is in a `GeomNode` of its own (and not merged with all the other geometry). In other words, you must ensure that panda's optimization mechanisms do not cause the geometry to be merged with the geometry of the rest of the model. While normally this optimization is a good thing, if you want to change textures on a specific part of the model (for example, just a character's face) you will need this geometry to be separate.

There are two different ways that you should do this, according to the type of model it is.

Animated (skeleton animation) models

If your model is animated via keyframe animation in a package such as 3DSMax or Maya--that is, the sort of model you expect to load in via the Actor interface--then Panda will be aggressive in combining all of the geometry into as few nodes as possible. In order to mark particular geometry to be kept separate, you should use the `egg-optchar` program.

The name "optchar" is short for "optimize character", since the egg-optchar program is designed to optimize an animated character for runtime performance by removing unused and unneeded joints. However, in addition to this optimization, it also allows you to label a section of a model for later manipulation. Once you have labeled a piece of geometry, Panda's optimization mechanisms will not fold it in to the rest of the model.

Your first step is to note the name of the object in your modeling program. For example, suppose you want to control the texture of a model's head, and suppose (hypothetically) the head is labeled "Sphere01" in your modeling program. Use egg-optchar to tell panda that "Sphere01" deserves to be kept separate and labeled:

```
egg-optchar -d outputDir -flag Sphere01=theHead modelFile.egg anim1.egg anim2.
egg
```

Note that you must always supply the model file(s) and all of its animation files to egg-optchar at the same time. This is so it can examine all of the joints and determine which joints are actually animated; and it can remove joints by operating on all the files at once. The output of egg-optchar is written into the directory named by the "-d" parameter.

The "-flag" switch will ensure that panda does not rearrange the geometry for the named polyset, folding it into the model as a whole. It also assigns the polyset a meaningful name. Once you have labeled the relevant piece of geometry, you can obtain a pointer to it using the

`find` method:

```
myModelsHead = myModel.find("**/
theHead")
```

With this nodepath, you can manipulate the head separately from the rest of the model. For example, you can move the piece using `setPos`, or change its texture using `setTexture`, or for that matter, do anything that you would do to any other scene graph node.

Unanimated (environment) models

Other kinds of models, those that do not contain any skeleton or animations, are not optimized as aggressively by the Panda loader, on the assumption that the model's hierarchy was structured the way it is intentionally, to maximize culling (see Pipeline Tips). Thus, only certain nodes are combined with others, so it's quite likely that an object that you modeled as a separate node in your modeling package will still be available under the same name when you load it in Panda. But Panda doesn't promise that it will never collapse together nodes that it thinks need to be combined for optimization purposes, unless you tell it not to.

In the case of an unanimated model, the way to protect a particular node is to insert the `<Model>` flag into the egg file within the particular group. The way to do this depends on your modeling package (and this documentation still needs to be written).

Panda3D Manual: Searching the Scene Graph

It is often useful to get a handle to a particular node deep within the scene graph, especially to get a sub-part of a model that was loaded from a single file. There are a number of methods dedicated to finding entrenched nodes and returning the NodePaths.

First, and most useful, is the ls() command:

```
nodePath.ls
()
```

This simply lists all of the children of the indicated NodePath, along with all of their children, and so on until the entire subgraph is printed out. It also lists the transforms and Render Attributes that are on each node. This is an especially useful command for when you're running interactively with Python; it's a good way to check that the scene graph is what you think it should be.

The two methods `find()` and `findAllMatches()` will return will return a `NodePath` and a `NodePathCollection` respectively. These methods require a path string as an argument. Searches can based on name or type. In its simplest form this path consists of a series of node names separated by slashes, like a directory pathname. When creating the string each component may optionally consist of one of the following special names, instead of a node name.

*	Matches exactly one node of any name
**	Matches any sequence of zero or more nodes
+typename	Matches any node that is or derives from the given type
-typename	Matches any node that is the given type exactly
=tag	Matches any node that has the indicated tag
=tag=value	Matches any node whose tag matches the indicated value

Standard filename globbing characters, such as *, ?, and [a-z] are also usable. Also the @@ special character before a node name indicates that this particular node is a stashed node. Normally, stashed nodes are not returned. @@*, by extension, means any stashed node.

The argument may also be followed with control flags. To use a control flag, add a semicolon after the argument, followed by at least one of the special flags with no extra spaces or punctuation.

-h	Do not return hidden nodes
+h	Return hidden nodes
-s	Do not return stashed nodes unless explicitly referenced with @@

+s	Return stashed nodes even without any explicit @@ characters
-i	Node name comparisons are not case insensitive: case must match exactly
+i	Node name comparisons are case insensitive: case is not important. This affects matches against the node name only; node type and tag strings are always case sensitive

The default flags are +h-s-i.

The `find()` method searches for a single node that matches the path string given. If there are multiple matches, the method returns the shortest match. If it finds no match, it will return an empty NodePath. On the other hand, `findAllMatches()` will return all NodePaths found, shortest first.

```
nodePath.find("<Path>")
nodePath.findAllMatches
("<Path>")
```

Some examples:

```
nodePath.find("house/
door")
```

This will look for a node named "door", which is a child of a node named "house", which is a child of the starting path.

```
nodePath.find("**/
red*")
```

This will look for any node anywhere in the tree (below the starting path) with a name that begins with "red".

In addition there are also the methods `getParent()` and `getChildren()`. `getParent()` returns the NodePath of the parent node. `getChildren()` returns the children of the current node as a NodePathCollection(use `getChildrenAsList()` if you want them as a List).

`getChildrenAsList` Example

```
for child in nodePath.getChildrenAsList():
   print child
```

some examples:

```
#if you wanted to search up the Scene Graph until you found a certain
node
while nodePath.getParent()!=someAncestor:
   nodePath=nodePath.getParent()
nodePath=nodePath.getParent()
```

For more information and a complete list of NodePath functions please see the API reference.

<<prev top next>> Search

Panda3D Manual: Render Attributes

Panda uses a set of *attributes* to define the way geometry is rendered. The complete set of attributes in effect on a given node is called the **RenderState**; this state determines all the render properties such as color, texture, lighting, and so on.

These individual attributes can be stored on any node of the scene graph; setting an attribute on a node automatically applies it to that node as well as to all of the children of the node (unless an override is in effect, but that's a more advanced topic).

It is possible to create these attributes and assign them to a node directly:

```
nodePath.node().setAttrib
(attributeObject)
```

But in many cases, especially with the most commonly-modified attributes, you don't need to create the attributes directly as there is a convenience function on NodePath (e.g. `nodePath.setFog()`) that manages the creation of the attributes for you; there will also be a corresponding clear function on NodePath to remove the attribute (`nodePath.clearFog()`).

The following is a list of the attributes available in Panda3D as of the time of this writing, along with the primary NodePath method to set them, and/or a reference to the manual page that describes the attribute in more detail:

AlphaTestAttrib	-
ClipPlaneAttrib	nodePath.setClipPlane(planeNode)
ColorAttrib	nodePath.setColor(r, g, b, a)
ColorBlendAttrib	-
ColorScaleAttrib	nodePath.setColorScale(r, g, b, a)
ColorWriteAttrib	-
CullBinAttrib	nodePath.setBin('binName', order)
CullFaceAttrib	nodePath.setTwoSided(flag)
DepthOffsetAttrib	-
DepthTestAttrib	nodePath.setDepthTest(flag)
DepthWriteAttrib	nodePath.setDepthWrite(flag)
FogAttrib	nodePath.setFog(fog); See Also: Fog
LightAttrib	nodePath.setLight(light); See Also: Lighting
MaterialAttrib	nodePath.setMaterial(material)
RenderModeAttrib	nodePath.setRenderMode(RenderModeAttrib.Mode)
ShaderAttrib	nodePath.setShader(shader); See Also: Using Cg Shaders

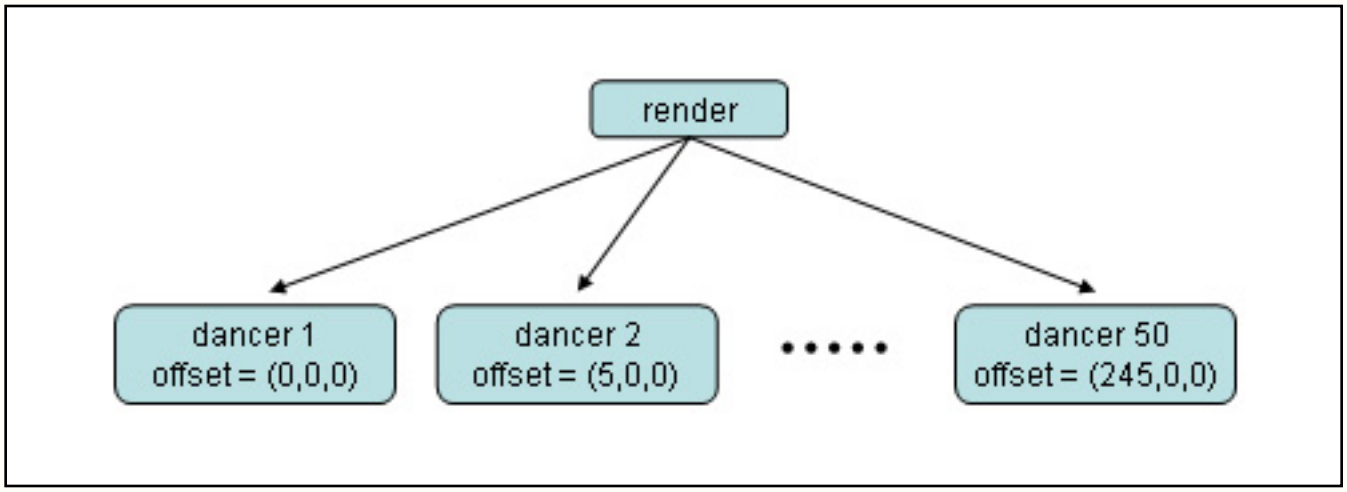
TexGenAttrib	nodePath.setTexGen(stage, TexGenAttrib.Mode); See Also: Automatic Texture Coordinates
TexMatrixAttrib	nodePath.setTexTransform(TransformState.make(mat)); See Also: Texture Transforms
TextureAttrib	nodePath.setTexture(tex); See Also: Simple Texturing and Multitexture Introduction
TransparencyAttrib	nodePath.setTransparency(TransparencyAttrib.Mode)

Panda3D Manual: Instancing

In the musical "A Chorus Line," the most well-known scene is when about 50 identical-looking young women line up left-to-right across the stage, and they all kick-left-kick-right in unison. To implement this in Panda3D, you might do this:

```
for i in range(50):
  dancer = Actor.Actor("chorus-line-dancer.egg", {"kick":"kick.
egg"})
  dancer.loop("kick")
  dancer.setPos(i*5,0,0)
  dancer.reparentTo(render)
```

Here is the scene graph that we just created:

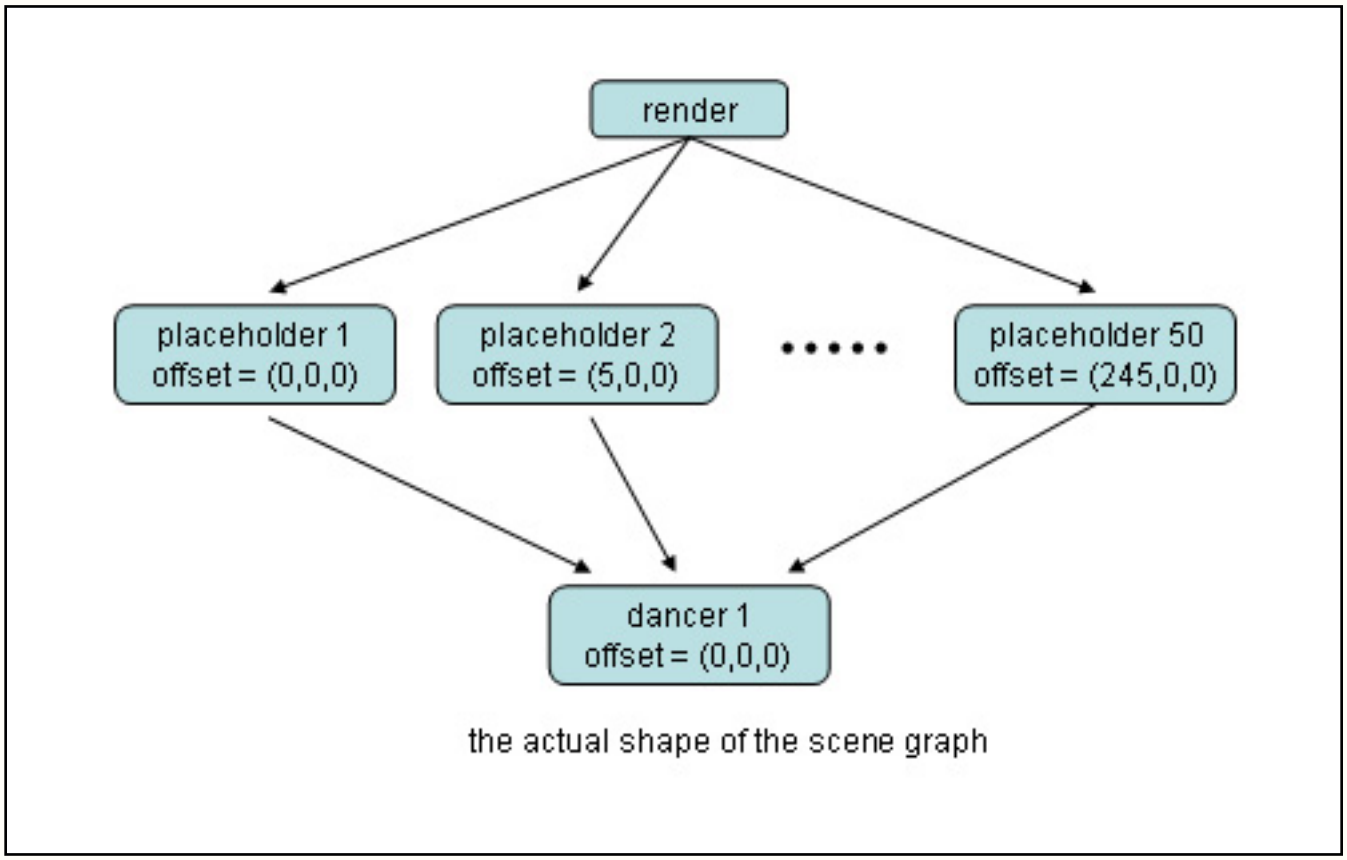


This works fine, but it is a little expensive. Animating a model involves a lot of per-vertex matrix calculations. In this case, we're animating 50 copies of the exact same model using 50 copies of the exact same animation. That's a lot of redundant calculation. It would seem that there *must* be some way avoid calculating the exact same values 50 times. There is: the technique is called *instancing*.

The idea is this: instead of creating 50 separate dancers, create only one dancer, so that the engine only has to update her animation once. Cause the engine to render her 50 times, by inserting her into the scene graph in 50 different places. Here is how it is done:

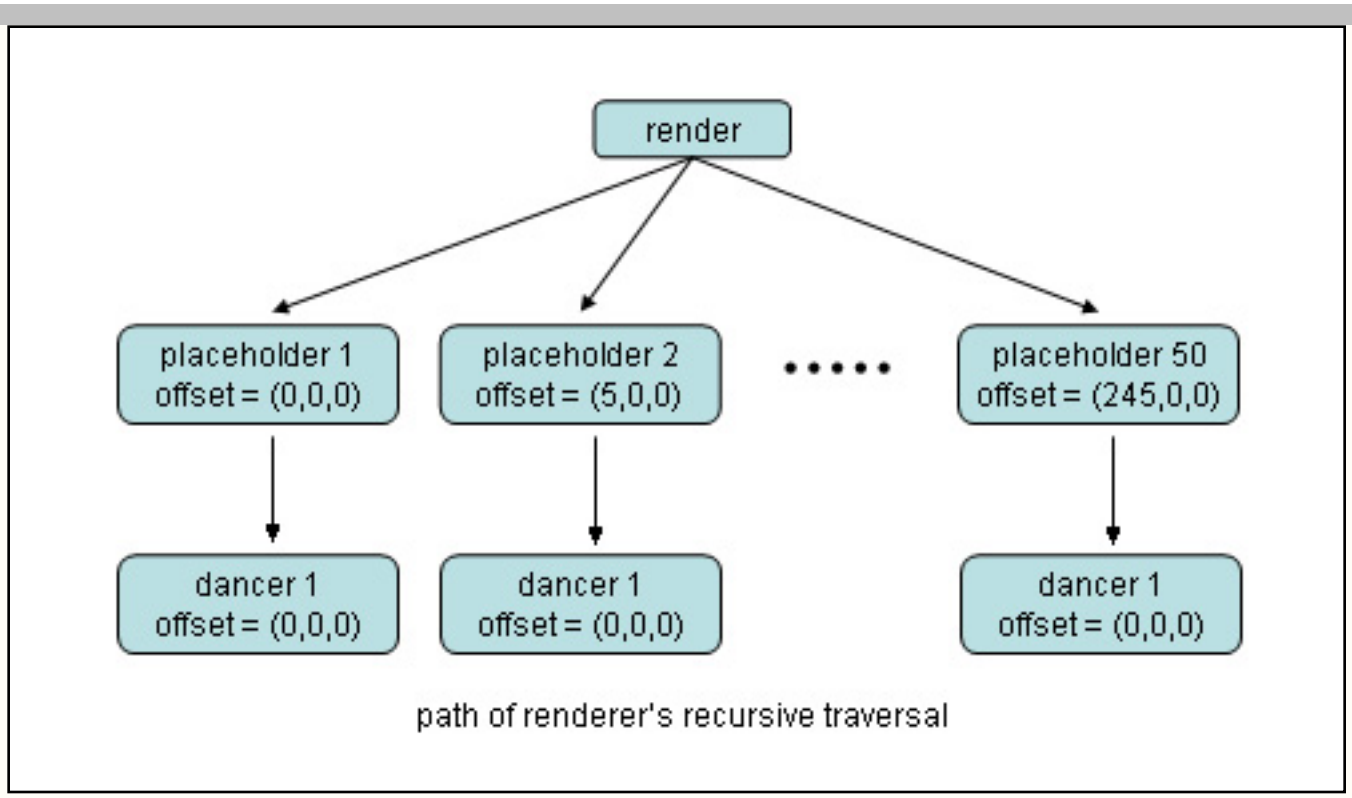
```
dancer = Actor.Actor("chorus-line-dancer.egg", {"kick":"kick.
egg"})
dancer.loop("kick")
dancer.setPos(0,0,0)
for i in range(50):
  placeholder = render.attachNewNode("Dancer-Placeholder")
  placeholder.setPos(i*5,0,0)
  dancer.instanceTo(placeholder)
```

Here is a diagram of the scene graph we just created:



the actual shape of the scene graph

It's not a tree any more, it is a directed acyclic graph. But the renderer still traverses the graph using a recursive tree-traversal algorithm. As a result, it ends up traversing the dancer node 50 times. Here is a diagram of the depth-first traversal that the renderer takes through the graph. Note that this is *not* a diagram of the scene graph - it's a diagram of the renderer's *path* through the scene graph:



In other words, the renderer visits the dancer actor 50 times. It doesn't even notice that it's visiting the same actor 50 times, rather than visiting 50 different actors. It's all the same to the renderer.

There are 50 placeholder nodes, lined up across the stage. These are called *dummy nodes*. They don't contain any polygons, they're little tiny objects used mainly for organization. In this case, I'm using each placeholder as a platform on which a dancer can stand.

The position of the dancer is (0,0,0). But that's relative to the position of the parent. When the renderer is traversing placeholder 1's subtree, the dancer's position is treated as relative to placeholder 1. When the renderer is traversing placeholder 2's subtree, the dancer's position is treated as relative to placeholder 2. So although the position of the dancer is fixed at (0,0,0), it appears in multiple locations in the scene (on top of each placeholder).

In this way, it is possible to render a model multiple times without storing and animating it multiple times.

Advanced Instancing

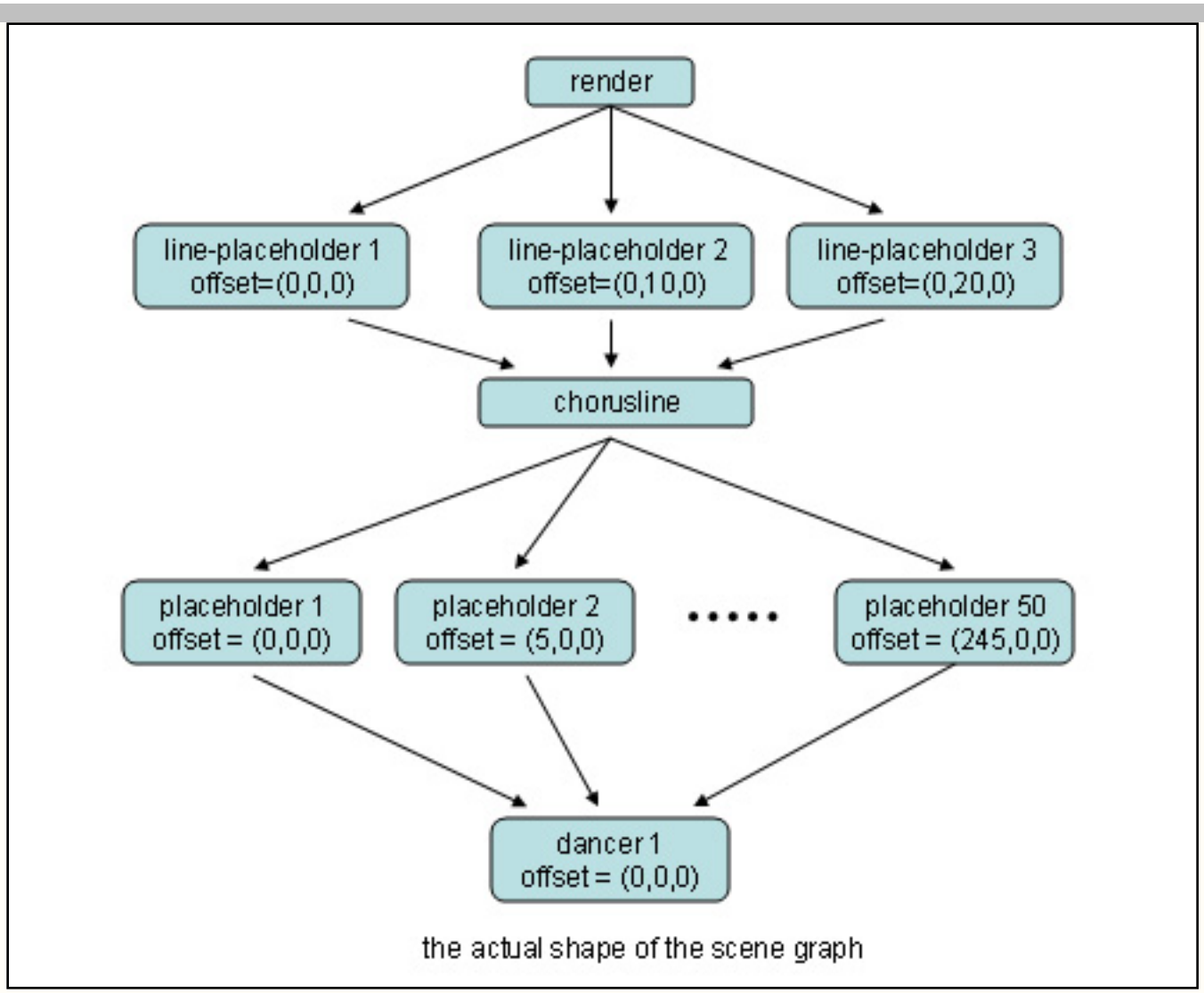
Now, let's go a step further:

```
dancer = Actor.Actor("chorus-line-dancer.egg", {"kick":"kick.
egg"})
dancer.loop("kick")
dancer.setPos(0,0,0)
chorusline = NodePath()
for i in range(50):
  placeholder = chorusline.attachNewNode("Dancer-Placeholder")
  placeholder.setPos(i*5,0,0)
  dancer.instanceTo(placeholder)
```

This is the exact same code as before, except that instead of putting the 50 placeholders beneath `render`, I put them beneath a dummy node called `chorusline`. So my line of dancers is not part of the scene graph yet. Now, I can do this:

```
for i in range(3):
  placeholder = render.attachNewNode("Line-
Placeholder")
  placeholder.setPos(0,i*10,0)
  chorusline.instanceTo(placeholder)
```

Here is the scene graph I just created:



the actual shape of the scene graph

But when the renderer traverses it using a recursive tree-traversal algorithm, it will see 3 major subtrees (rooted at a line-placeholder), and each subtree will contain 50 placeholders and 50 dancers, for a grand total of 150 apparent dancers.

Instancing: an Important Caveat

Instancing saves panda quite a bit of CPU time when animating the model. But that doesn't change the fact that the renderer still needs to render the model 150 times. If the dancer is a 1000 polygon model, that's still 150,000 polygons.

Note that each instance has its own bounding box, each is occlusion-culled and frustum-culled separately.

The NodePath: a Pointer to a Node plus a Unique Instance ID

If I had a pointer to the chorus-line dancer model, and I tried to ask the question "where is the dancer," there would be no well-defined answer. The dancer is not in one place, she is in 150 places. Because of this, the data type *pointer to node* does not have a method that retrieves the net transform.

This is very inconvenient. Being able to ask "where is this object located" is fundamental. There are other incredibly useful queries that you cannot perform because of instancing. For example, you cannot fetch the parent of a node. You cannot determine its global color, or any other global attribute. All of these queries are ill-defined, because a single node can have many positions, many colors, many parents. Yet these queries are essential. It was therefore necessary for the panda3d designers to come up with some way to perform these queries, even though a node can be in multiple locations at the same time.

The solution is based on the following observation: if I had a pointer to the chorus line-dancer model, and I *also* had a unique identifier that distinguishes one of the 150 instances from all the others, then I could meaningfully ask for the net transform of that particular instance of the node.

Earlier, it was noted that a NodePath contains a pointer to a node, plus some administrative information. The purpose of that administrative information is to uniquely identify one of the instances. There is no method `Node::getNetTransform`, but there *is* a method `NodePath::getNetTransform`. Now you know why.

To understand how NodePath got its name, think about what is necessary to uniquely identify an instance. Each of the 150 dancers in the graph above corresponds to a single path through the scene graph. For every possible path from root to dancer, there exists one dancer-instance in the scene. In other words, to uniquely identify an instance, you need a *list of nodes* that starts at the leaf and goes up to the root.

The administrative information in a nodepath is a list of nodes. You can fetch any node in the list, using the `NodePath::node(i)` method. The first one, `node(0)`, is the node to which the NodePath points.

Panda3D Manual: Panda Filename Syntax

For easier portability, Panda3D uses Unix-style pathnames, even on Microsoft Windows. This means that the directory separator character is always a forward slash, not the Windows backslash character, and there is no leading drive letter prefix. (Instead of a leading drive letter, Panda uses an initial one-letter directory name to represent the drive.)

There is a fairly straightforward conversion from Windows filenames to panda filenames. Always be sure to use Panda filename syntax when using a Panda3D library function, or one of the panda utility programs:

```
# WRONG:
loader.loadModel("c:\\Program Files\\My Game\\Models\\Model1.
egg")

# CORRECT:
loader.loadModel("/c/Program Files/My Game/Models/Model1.egg")
```

Panda uses the `Filename` class to store Panda-style filenames; many Panda functions expect a Filename object as a parameter. The Filename class also contains several useful methods for path manipulation and file access, as well as for converting between Windows-style filenames and Panda-style filenames; see the API reference for a more complete list.

To convert a Windows filename to a Panda pathname, use code similar to the following:

```
from pandac.PandaModules import Filename
winfile = "c:\\MyGame\\Model1.egg"
pandafile = Filename.fromOsSpecific
(winfile)
print pandafile
```

To convert a Panda filename into a Windows filename, use code not unlike this:

```
from pandac.PandaModules import Filename
pandafile = Filename("/c/MyGame/Model1.
egg")
winfile = pandafile.toOsSpecific()
print winfile
```

The Filename class can also be used in combination with python's built-in path manipulation mechanisms. Let's say, for instance, that you want to load a model, and the model is in the

"model" directory that is in the same directory as the main program's "py" file. Here is how you would load the model:

```
import sys,os
from pandac.PandaModules import Filename

# get the location of the 'py' file I'm running:
mydir = os.path.abspath(sys.path[0])

# convert that to panda's unix-style notation
mydir = Filename.fromOsSpecific(mydir).getFullpath()

# now load the model
model = loader.loadModel(mydir + "/models/mymodel.
egg")
```

Panda3D Manual: The Configuration File

In the *etc* subdirectory, you will find a configuration file *Config.prc*. This controls several of Panda's configuration options - does it use OpenGL or DirectX, how much debugging output does it print, and so forth. The following table lists several of the most commonly-used variables.

Variable	Values	Default	Details
load-display	pandagl pandadx8	pandagl	Specifies which graphics GSG to use for rendering (OpenGL or DirectX 8)
win-size	Pixels x y	800 600	Specifies the size of the Panda3D window
win-origin	Pixels x y	100 0	Specifies the onscreen placement of the upper left corner of Panda3d window
fullscreen	#t #f	#f	Enables full-screen mode (true or false)
undecorated	#t #f	#f	Removes border from window (true or false)
cursor-hidden	#t #f	#f	Hides mouse cursor (true or false)
show-frame-rate-meter	#t #f	#f	Shows the fps in the upper right corner of the screen (true or false)
audio-cache-limit	number	32	limits the number of sounds you can load
notify-level-[package]	fatal error warning info debug spam	info	Sets notification levels for various Panda3D packages to control the amount of information printed during execution (fatal being least, spam being most)
model-path	Directory name		Adds specified directory to the list of directories searched when loading a model or texture
sound-path	Directory name		Adds specified directory to the list of directories searched when loading a sound
load-file-type	ptloader		Allows the loading of file types for which converters have been written for in pandatool

audio-library-name	fmod_audio miles_audio null	fmod_audio	Loads the appropriate audio drivers. Miles is a propriertary audio, so only select that option if you currently have it.
want-directtools	#t #f	#t line commented out	Enables directtools, a suite of interactive object/camera manipulation tools
want-tk	#t #f	#t line commented out	Enables support for using Tkinter/PMW (Pythonâ€™s wrappers around Tk)

You can get a more complete list of available config variables at runtime (once you have imported DirectStart), with the Python command:

```
cvMgr.listVariables
()
```

<<prev top next>> Search

Panda3D Manual: Accessing Config Vars in a Program

Panda3D uses a configuration file named Config.prc. Panda supplies functions to easily read values out of Config.prc, and to alter their values in memory (the modified values are not written back out to disk). The ability to read an alter configuration settings programmatically has two major uses:

1. Storing your own configuration data.
2. Tweaking Panda's behavior.

By "storing your own configuration data," I mean that your game might have its own settings that need to be stored. Rather than writing your own configuration file parser, you might consider adding your configuration data to the panda configuration file instead.

Suppose hypothetically that you are writing an online game, and your online game connects to a server. You need a configuration file to tell you the name of the server. Open up the "Config.prc" file and add the following line at the end of the file.

```
my-game-server pandagame.
com
```

Note that I invented the variable name "my-game-server" out of thin air, this variable is not recognized by panda in any way. Therefore, this line has no effect on panda whatsoever.

To manipulate this variable programmatically, use code not unlike the following, which creates an object of class `ConfigVariableString` and then manipulates it using the methods `setValue` and `getValue`:

```
from pandac.PandaModules import ConfigVariableString

mygameserver = ConfigVariableString("my-game-server","127.0.0.1")
print "Server specified in config file: ", mygameserver.getValue
()

# allow the user to change servers on the command-line:
if (sys.argv[1]=="--server"): mygameserver.setValue(sys.argv[2])

print "Server that we will use: ", mygameserver.getValue()
```

The second parameter to the ConfigVariableString constructor is the default value that should be returned, in case the line "my-game-server" does not appear in any Config.prc file. There is also an optional third parameter, which is a description of the purpose of the variable; this string will be displayed when the user executes the command `print cvMgr`.

The types of configuration variable are:

```
ConfigVariableString
ConfigVariableInt
ConfigVariableBool
ConfigVariableDouble
ConfigVariableFilename
ConfigVariableList
ConfigVariableSearchPath
```

Most of these follow the same form as ConfigVariableString, above, except that the default value (and the parameter from setValue() and getValue()) is of the indicated type, rather than a string. The two exceptions are ConfigVariableList and ConfigVariableSearchPath; these types of variables do not accept a default value to the constructor, since the default value in both cases is always the empty list or search path.

To display the current value of a particular variable interactively (in this example, for a string-type variable), type the following:

```
print ConfigVariableString('my-game-
server')
```

Panda3D will automatically load any *.prc files it finds in its standard config directory at startup. You can view a list of the files it has actually loaded with the following command:

```
print
cpMgr
```

It is helpful to do this to ensure that you are editing the correct Config.prc file.

Sometimes, it is desirable to load an additional configuration file from disk, by giving an explicit filename. To do so, use "loadPrcFile". Note that Panda Filename Syntax uses a forward slash, even under Windows:

```
from pandac.PandaModules import
loadPrcFile
loadPrcFile("config/Config.prc")
```

The filename you specify is searched for along the model-path, in the same way that an egg or bam file is searched for when you use `loader.loadModel()`.

You can also use "loadPrcFileData" to load a string that you define in your Python code, as if it

were the contents read from a disk file. The `loadPrcFileData()` call requires two parameters; the first parameter is an arbitrary string name to assign to this "file" (and it can be the empty string if you don't care), while the second parameter is the contents of the file itself. This second parameter should contain newlines between variable definitions if you want to set the value of more than one variable.

For example, let's say that panda's configuration file contains this line:

```
fullscreen
0
```

By default, panda programs will run in a window, not fullscreen. However, if you do this:

```
from pandac.PandaModules import
loadPrcFileData
loadPrcFileData("", "fullscreen 1")
import direct.directbase.DirectStart
```

Then by the time you load `direct.directbase.DirectStart`, you will have changed the fullscreen-flag to true, and your program will run full-screen. There are other ways to go to fullscreen, this is not necessarily the most straightforward, but it illustrates the point.

Panda3D Manual: Actors and Characters

Panda3D supports both skeletal animation and morph animations. Panda's egg file format can contain an animatable model, a recorded sequence of animations, or both.

The python class `Actor` is designed to hold an animatable model and a set of animations. Since the Actor class inherits from the NodePath class, all NodePath functions are applicable to actors.

Note that `Actor` is actually a high-level interface to a set of lower-level classes. It is not recommended to use these lower-level classes directly, with one exception: the Model class is easy to use directly as long as you don't plan to animate the model. You can therefore eliminate a bit of overhead by using a Model instead of an Actor for your static objects.

This section assumes you have a valid egg file which has an animatable model, and some additional egg files containing animations. To learn how to convert a model into an egg file see the Model Export section.

Panda3D Manual: Loading Actors and Animations

The Actor class must be imported before any loading or manipulation of actors.

```
from direct.actor import
Actor
```

Once the module is loaded, the actor object must be constructed, and the model and animations must be loaded:

```
nodePath = Actor.Actor()
nodePath.loadModel(â€˜Model Pathâ€™>
nodePath.loadAnims({â€˜Arbitrary Name1â€™:â€™Animation Path 1â
€™})
nodePath.loadAnims({â€˜Arbitrary Name2â€™:â€™Animation Path 2â
€™})
```

Loading each animation requires a tuple: the name one is giving the animation and the path to the animation. This entire process can be shortened to a single command:

```
nodePath = Actor.Actor('Model Path', {
  'Animation Name 1':'Animation Path
1',
  'Animation Name 2':'Animation Path
2',
})
```

Animations may also be unloaded using the same tuple used in creating them.

```
nodePath.unloadAnims({'Animation Name':'Animation
Path'})
```

Although this is a rarely-used technique, it is possible to assemble a character model out of several separate pieces (separate models). If this is the case, then the pieces must contain bones that can be attached to each other. For example, if you have a robot consisting of a set of legs and a swappable torso, and if you want to glue them together at the waist, then the legs model should contain a bone "waist", and the torso model should also contain a bone "waist". You can then attach them together:

```
nodePath = Actor.Actor({
    'legs':'RobotLegs.egg',
    'torso':'RobotTorso1.egg',
  },{'dance':'RobotDance.egg'})
nodePath.attach
('legs','torso','waist')
```

Panda3D Manual: Actor Animations

Since the Actor class inherits from NodePath, everything that can be done to a NodePath, such as `reparentTo()` and `setPos()`, etc., may also be done to an Actor. In addition to the basic NodePath functionality, Actors have several additional methods to control animation. In order for Actors to animate, their pointer (variable) must be retained in memory. The following is only a brief introduction; see the API reference for a complete list.

Basic animation playing

Animations may either be played or looped. When an animation is played, the actor goes through the animation once. When an animation is looped, the animation will play continuously. There is no tweening done between the last and the first frame, so if an animation is going to be looped, it needs to be constructed with that thought in mind. Finally, animations may be stopped at any point. When an animation is stopped, the actor will stay in the position it stopped on.

```
actor.play('Animation Name')
actor.loop('Animation Name')
actor.stop()
```

You may use the `pose()` method to tell an actor to hold a particular frame of the animation. Frames are numbered beginning at 0.

```
actor.pose('Animation Name',
FrameNumber)
```

Posing an actor to a frame doesn't automatically specify the start frame of the next starting animation. Instead, if you don't want to start at the first frame, you can specify these using the optional parameters `fromFrame` and `toFrame` to the methods `play()` and `loop()`:

```
actor.play('Animation Name', fromFrame = 10)
actor.loop('Animation Name', fromFrame = 24, toFrame = 36)
```

However, the loop method does have another optional parameter called `restart`, which is 1 by default, meaning the animation will restart from the beginning. If you pass it 0 instead, then the animation will begin looping from the current frame:

```
actor.pose('Animation Name', 30)
actor.loop('Animation Name', restart = 0, fromFrame = 24, toFrame = 36)
```

Play rate

The animation play rate may be set to any floating point value, which can be used to speed up or slow down the animation. This is a scale factor on the base animation rate; 1.0 means to play the animation at its normal speed, while 2.0 plays it twice as fast, and 0.5 plays it at half speed. It is also possible to play an animation backwards by specifying a negative play rate, for instance -1.0.

```
actor.setPlayRate(newPlayRate, 'Animation Name')
```

Blending

Multiple different animations for an actor may be played at the same time, and the animations blended together at runtime. The net result is that, each frame, the actor ends up somewhere between the different poses it would be in for each contributing animation, if each animation were playing independently.

Note that in blend mode each contributing animation still affects the actor's entire body. If you want to play one animation on, say, the left arm, while a different animation is playing on the legs, then you need to use half-body animation, which is different from blending.

To use blending, you must first call `enableBlend()` to activate the blending mode and indicate your intention to play multiple animations at once. While the actor is in blend mode, playing a new animation does not automatically stop the previously playing animation. Also, while in blend mode, you must explicitly specify how much each animation contributes to the overall effect, with the `setControlEffect()` method (the default for each animation is 0.0, or no contribution). For example:

```
actor.enableBlend()
actor.setControlEffect('animation1', 0.2)
actor.setControlEffect('animation2', 0.8)
actor.loop('animation1')
actor.loop('animation2')
```

The above specifies that 20% of animation1 and 80% of animation2 will be visible on the character at the same time. Note that you still have to start both animations playing (and they can be playing from different frames or at different play rates). Starting or stopping an animation in blend mode does not change its control effect; you must set an animation's control effect to 0.0 if you don't want it to have any more affect on the actor.

When you call `stop()` in blend mode, you can stop a particular animation by name, if you

want; or you can stop all of the animations by calling `stop()` with no parameters:

```
actor.stop('animation1')
```

Note that specifying an animation name to stop() is only meaningful when you are in blend mode. When not in blend mode, actor.stop() will always stop whatever animation is currently playing, regardless of the animation name you specify.

When you are done using blending and want to return to the normal mode of only playing one animation at a time, call `disableBlend()`:

```
actor.disableBlend()
```

Actor Intervals

Another way to play an animation on an actor is to use an ActorInterval, which gives you a lot more frame-by-frame control over the animation, and is particularly useful when building a complex script using Intervals. However, the ActorInterval interface is a little bit slower than the above interfaces at runtime, so you should prefer the more fundamental interfaces unless there is a good reason to use ActorInterval.

The Task manager

On a more complex program, you may find that Animations can not be loaded from any point in your program. In any application there needs to be exactly one call to run(), and it should be the last thing you do after starting up. This starts the task manager. Think of this as the main loop of the application: your startup procedure is to set up your loading screen, start any initial tasks or intervals, hang any initial messenger hooks, and then go get lost in run(). Thereafter everything must run in a task, in an interval, or is a response to a message. This is true for both animations and sound.

Panda3D Manual: Attaching an Object to a Joint

If an actor has a skeleton, then it is possible to locate one of the joints, and attach an object to that joint:

```
myNodePath = actorNodePath.exposeJoint(None,"modelRoot","Joint
Name")
```

This function returns a nodepath which is attached to the joint. By reparenting any object to this nodepath, you can cause it to follow the movement of the joint.

The string "modelRoot" represents the name of the model node - the string "modelRoot" is usually the correct value.

The string "Joint Name" represents the name of the joint. Typically it would be something like "Femur", or "Neck", or "L Finger1". This is usually set inside the modeling package. For example, in MAX, each object in the scene has a name, including the bones. If necessary, you can determine the joint names by scanning the egg file for strings like `<Joint> Femur`.

Panda3D Manual: Controlling a Joint Procedurally

Sometimes one wishes to procedurally take control of a model's joint. For example, if you wish to force a character model's eyes to follow the mouse, you will need to procedurally take control of the neck and head. To achieve this, use *controlJoint*. **Caution**: the behavior of *controlJoint* is not entirely straightforward, so be sure to read this entire section.

```
myNodePath = actor.controlJoint(None,"modelRoot","Joint
Name")
```

This creates a dummy node. Every frame, the transform is copied from the dummy node into the joint. By setting the transform of the dummy node, you can control the joint. Normally, one would want to use `setHpr` to rotate the joint without moving it. The dummy node is initialized in such a way that the joint is in its default location, the one specified in the model's egg file.

You must store a local (not global) transform in the dummy node. In other words, the transform is relative to the joint's parent bone. If you are controlling the forearm of a model, for instance, the transform will be relative to the upperarm.

The string "modelRoot" represents the name of the model node - the string "modelRoot" is usually the correct value.

The string "Joint Name" represents the name of the joint. Typically it would be something like "Femur", or "Neck", or "L Finger1". This is usually set inside the modeling package. For example, in MAX, each object in the scene has a name, including the bones. If necessary, you can determine the joint names by scanning the egg file for strings like `<Joint> Femur`. Beginning in Panda3D version 1.2, you can also use the call `actor.listJoints()` to show the complete hierarchy of joints.

Cautions and limitations

- controlJoint only works when an animation is playing on the joint. The animation of the controlled joint is completely overridden. However, the animated movement of other joints continues normally. In other words, you have to create an animation for the model, and the animation must manipulate the joint you wish to control. If the animation ends, the control stops as well. If the animation is restarted, you regain control.
- controlJoint works by setting up some internal structures that must be in place before a given animation has been started the first time. Thus, it is important to make all of your controlJoint() calls for a particular model before you make the first call to play(), loop(), or pose().
- controlJoint cannot be undone. Once you call controlJoint, the joint in question is forever under application control; the animation channels no longer affect the joint. If you need to restore animation control to a joint in your application, one strategy would be to load two copies of the Actor, and only call controlJoint on one of them. When you

want to restore animation control, swap in the other Actor. A different strategy might be to create a new, dummy joint in your character that doesn't have any animation on it anyway, and only call controlJoint on that dummy joint.

These limitations are due to the implementation of the animation subsystem. Eventually, we hope to remove the these quirks in controlJoint.

<<prev top next>> Search

Panda3D Manual: Camera Control

Panda3D's camera is considered a PandaNode. It can therefore be manipulated as any other node.

The actual camera is defined in ShowBase as a NodePath named `base.cam`. There is also a plain node above the camera, which is a NodePath called `base.camera`. Generally you want to control the `base.camera` NodePath with your code.

Panda3D Manual: The Default Camera Driver

By default, panda runs a task that enables you to move the camera using the mouse. This task will conflict with any code you write to move the camera. The task controls the camera by updating the camera's position every frame to where the mouse thinks it should be. This means that any other code that directly controls the camera will not seem to work, because it will be fighting the mouse for control.

If you want to move the camera directly under show control, you must first disable the camera control task and then the camera will move as expected.

```
base.disableMouse
()
```

The ShowBase class contains some handy methods to allow the user control over the camera. The `useDrive()` command enables keyboard and mouse control. Both control systems move only on the x and y axes, so moving up and down along the z axis is impossible with these systems.

The keyboard system uses the arrow keys. Up moves the camera forward, and down move it back. The left and right arrows turn the camera.

The mouse system responds whenever any button is held. If the pointer is towards the top of the sceen, the camera moves forward. If it is towards the bottom, the camera moves backwards. If it is on either side, the camera rotates to that direction. The speed the camera moves is determined by how far from the center the mouse pointer is. Additionally, there is another command that allows control based on trackball mice.

```
base.useDrive()
base.useTrackball
()
```

ShowBase also provides the method `oobe()` to give you to control of the basic camera node (base.cam) with the mouse/trackball while the code continues to move the camera node (base.camera). This can be useful for debugging purposes. The word stands for "out-of-body experience" and is handy for giving you a God's-eye view of your application at any point in development. The method is a toggle; call it once to enable OOBE mode, and then again to disable it.

```
base.oobe
()
```

`oobeCull()` is a variant on `oobe()`, and it works similarly, except that it still culls the scene as if the camera were still in its original position, while drawing the scene from the point of view of your camera's new position. So now you can view the scene from your "out of body" placement, and walk around, and you can see things popping in and out of view as your view frustum moves around the world.

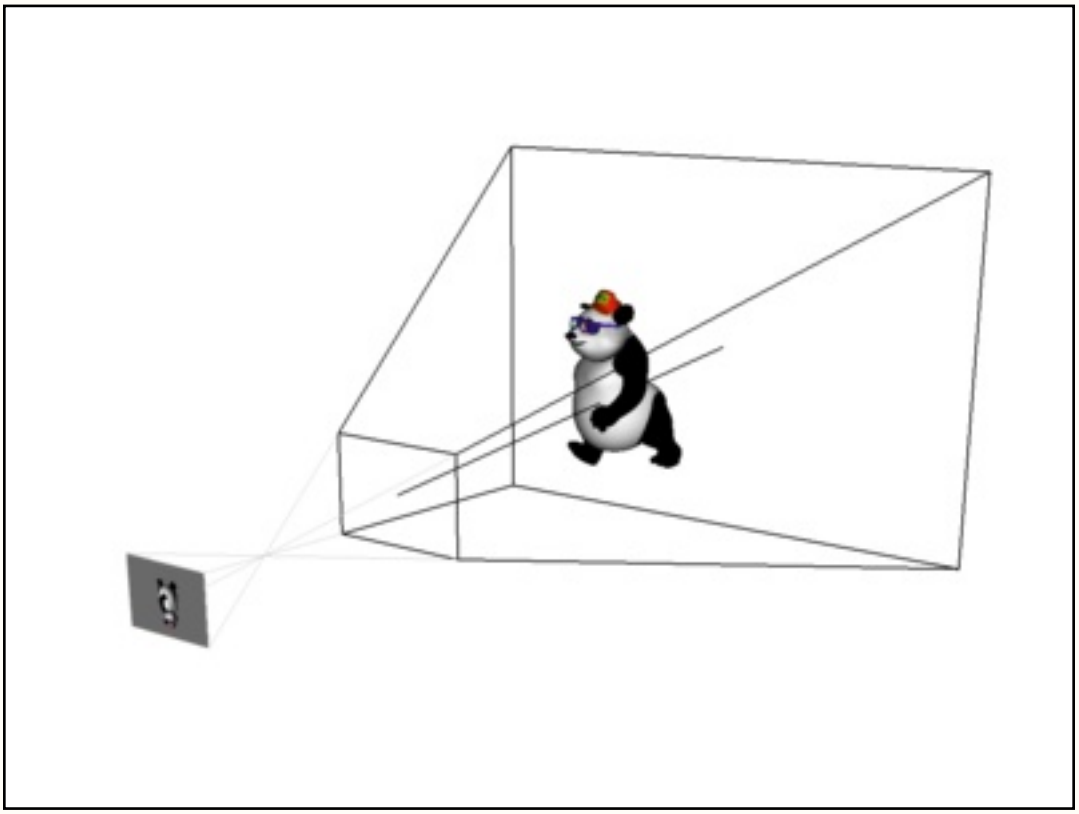
<<prev top next>> Search

Panda3D Manual: Lenses and Field of View

Every Camera has a Lens object that defines the properties of its view. For simple applications, you do not need to think about the lens; you will probably be happy with the default lens properties. However, you will occasionally want to adjust some properties of the lens, such as its field of view, and there are several interfaces to do this, depending on how you want to think about the lens.

When you start Panda3D, a default camera and lens are created for you automatically. The default camera object is stored in `base.cam` (although by convention, if you want to move the default camera you should manipulate `base.camera` instead), and the default lens is `base.camLens`.

This default lens will almost always be a perspective lens--that is, an instance of the class `PerspectiveLens`--unless you have done something to change it to another kind of lens. A perspective lens is by far the most common kind of lens used, and it behaves the same way the physical lens in a camera works, or for that matter the same way the lenses in our eyes work:

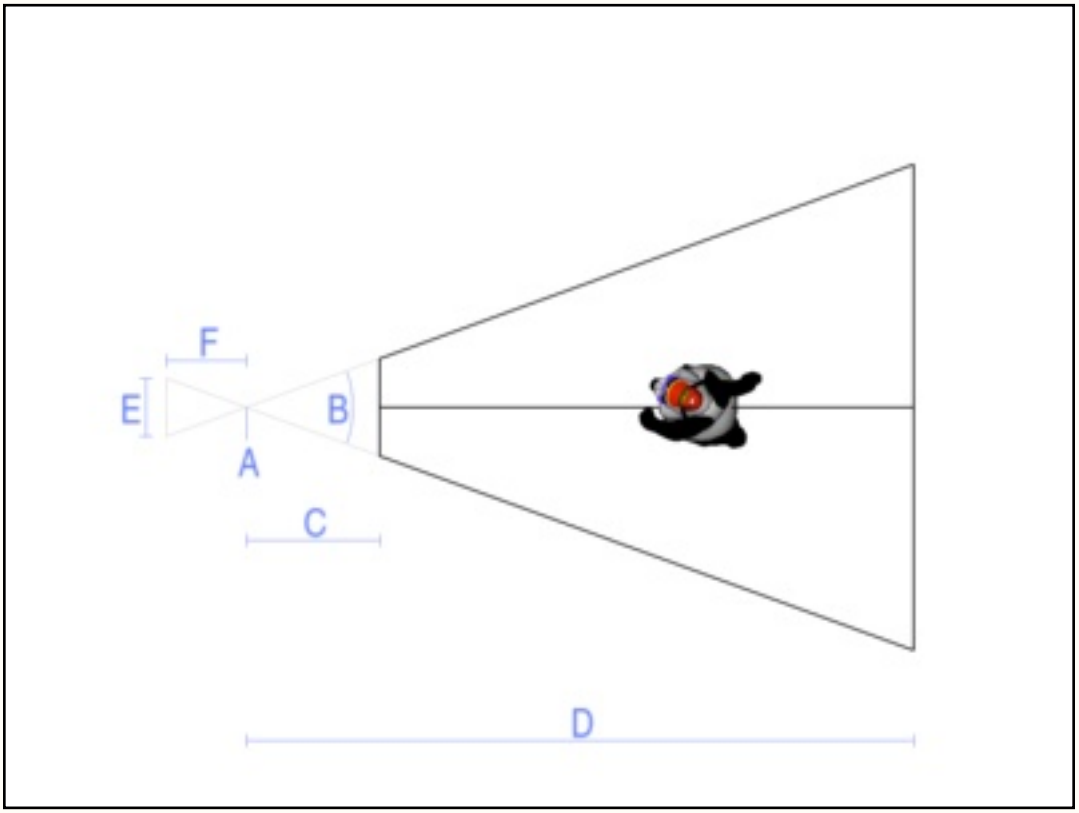


The illustration above shows a camera with an ordinary perspective lens observing a model in the world. The camera can only see the part of the world that falls within the black lines; this area is called the lens **frustum**.

In the picture, you can also see the image that the lens is capturing (and that image is shown

upside-down, just as it would be in a real, physical camera). This image is just for the purposes of illustration; it isn't really part of a Panda3D camera. It is included to help show the relationship between a Panda3D lens and a real, physical lens.

There are several different properties that can be set on a PerspectiveLens. Not all of them are independent; setting some properties will change the values of other properties. Here is an illustration:



A. This point is the **nodal point** or eyepoint of the lens. It is also (usually) the origin, that is, the (0, 0, 0) point of the camera that holds the lens. Normally (in a default Z-up coordinate system), the lens will be looking down the +Y axis, so in the above illustration the +Y axis extends to the right from point A. The plane containing the nodal point, perpendicular the viewing direction (that is, the plane corresponding to the vertical line through point A), is called the **camera plane**.

Although it is possible to change the nodal point or view direction of a lens to some point other than (0, 0, 0) or some direction other than down the +Y axis, it is usually simplest and best just to move the entire camera using the basic NodePath operations like setPos() and setHpr ().

B. This angle is the **field of view**, or fov, of the lens. You can easily change this by setting a new value in degrees with `lens.setFov(angle)`. Making the field of view smaller will bring things in closer, like a telephoto lens; it will also diminish the visible effects of perspective. Making the field of view larger will open up the view to more objects, like a wide-angle lens; it will also increase the visible distortion of perspective. The field of view must be greater than 0 degrees and less than 180, but values greater than 90 will seem extremely distorted. (In the real world, perspective lenses rarely go wider than 80 degrees, and that's pretty wide.) The default field of view is 40 degrees, which is usually a pretty comfortable viewing angle.

There is actually a separate horizontal field of view and vertical field of view, both of which may be independently controlled with the two-parameter form of setFov: `lens.setFov(horizontalAngle, verticalAngle)`. Using the two-parameter form will change the **aspect ratio** of the lens (see below). Normally, you would set the field of view using only the one-parameter form, which sets the horizontal field of view directly, and automatically recomputes the vertical field of view to preserve the same aspect ratio.

C. This distance is called the **near distance** or **near plane** of the lens. Objects that are closer than this to the camera plane will not be rendered. You may set the near distance as small as you like, but it must be greater than 0; and the smaller you set it, the greater the likelihood that you will observe an artifact called **Z-fighting**, a shimmering of objects that are off in the distance. The default near distance is 1.0, which for many scenes is a reasonable compromise. Of course, the most appropriate value for your scene depends on the nature of the scene (as well as the measurement units in which your scene is modeled).

You may change the near distance at any time with the call `lens.setNear(distance)`.

D. This is the **far distance** or **far plane** of the lens. Similar to the near distance, objects that are farther than this from the camera plane will not be rendered. You may set this as large as you like, but like the near distance, setting it too large may result in Z-fighting. (However, the near distance value has a much greater impact on Z-fighting than the far distance value, because of the nature of the math involved.) The default far distance is 1000.0, which is appropriate for small scenes; you may need to set it larger if you have a large scene.

You may change the far distance with the call `lens.setFar(distance)`. Since the near distance and far distance are often changed at the same time, there is a convenience function to set then both: `lens.setNearFar(nearDistance, farDistance)`.

E. This size is the **film size** of the lens. This is only an abstract concept in Panda3D; it is designed to simulate the actual film size of a physical lens. In a real, physical camera, the lens casts light onto a piece of film behind the lens, and the size of the film impacts the effective field of view of the lens via a mathematical formula that every photographer knows (and which I won't repeat here). In Panda3D, you will probably ignore the film size, unless you are a photographer, or you want to set up a virtual lens that exactly matches the properties of some real, physical lens.

You can specify the film size with `lens.setFilmSize(width)` or `lens.setFilmSize(width, height)`. Like field of view, the film size has two components, a horizontal film size and a vertical film size. Also like field of view, if you specify both components at once it will change the **aspect ratio** of the lens, but if you set only the width, Panda will automatically compute the height to keep the aspect ratio the same.

Setting the film size defines the units to be used for some of the other advanced lens properties, such as the **focal length** (below) and the **lens offset**. For instance, a 35mm camera exposes a rectangle on the film about 24mm x 36mm, so if you wanted to simulate a 35mm camera, you would use `lens.setFilmSize(24, 36)`. This establishes that your film units are in millimeters, so you could then specify a lens with a focal length of 50mm using `lens.setFocalLength(50)`. (Setting both the film size and the focal length like this would automatically calculate the field of view; see below.)

F. This distance is the **focal length** of the lens. Like film size, this is only an abstract concept in Panda3D, but it is a very important concept in a real, physical camera. Technically, it is the distance between a lens's nodal point or camera plane and its focal plane or film plane, and it affects the field of view of the lens. In real photography, lenses are typically described by their focal length, rather than by their field of view. You can set the focal length via `lens.setFocalLength(distance)`.

G (not pictured). The final important property of a lens is its **aspect ratio**. This is the ratio of the width to the height of the image produced by the lens. It is almost, but not quite, the same as the ratio of the horizontal field of view to the vertical field of view. (It is not quite this, because a perspective lens is not linear in proportion to the angle.) Normally, you will want the aspect ratio of the lens to match the aspect ratio of your window; if it is something different, the image may seem stretched or squashed.

You can set the aspect ratio explicitly via `lens.setAspectRatio(ratio)`. For instance, if you open a window that is 800 pixels wide and 300 pixels tall, you might want to call `lens.setAspectRatio(800.0 / 300.0)`.

Interplay of lens properties

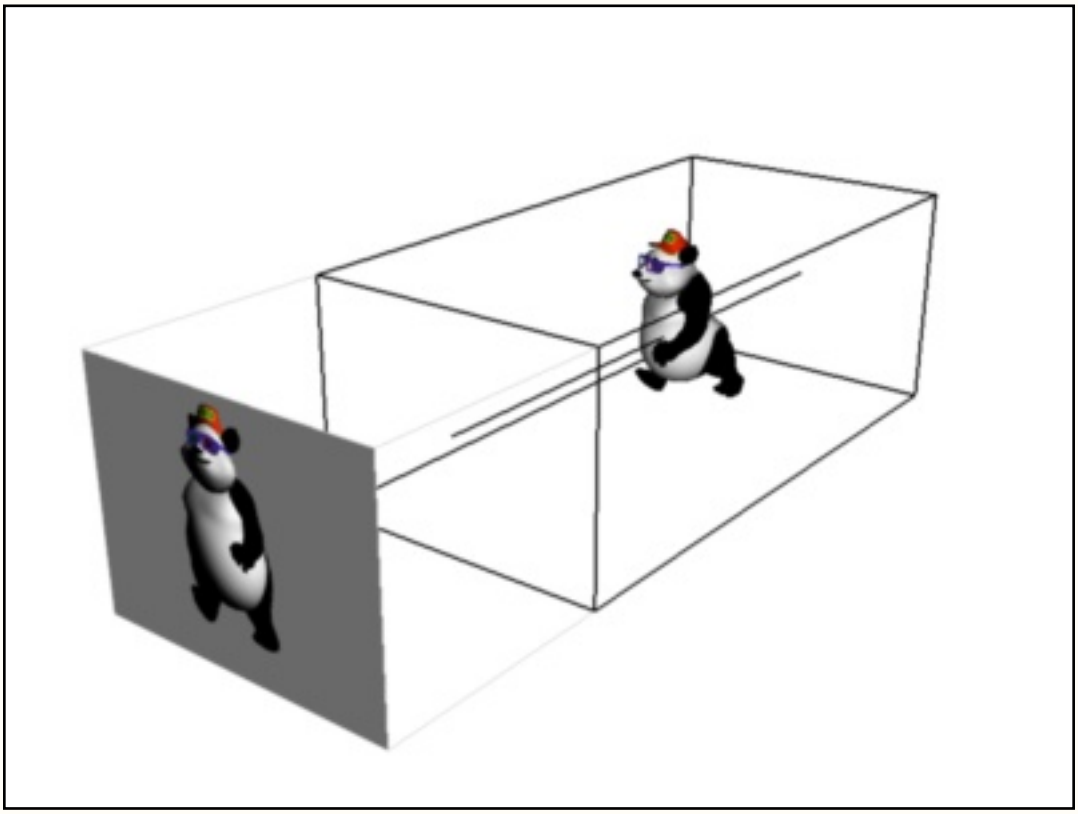
Note that, as mentioned above, several of these properties are interrelated. In particular, the field of view, focal length, and film size are closely tied together. Setting any two of these three properties will implicitly define the third one.

Panda will let you set all three of these properties as often as you like, but only the last two properties you set will be important. That is, if you set field of view and film size, Panda will calculate the focal length. If you set film size and focal length, Panda will calculate the field of view. If you set focal length and field of view, Panda will calculate the film size.

Also, the aspect ratio can be set either implicitly, by using the two-parameter `setFov()` or `setFilmSize()` methods, or explicitly, by directly specifying it with `setAspectRatio()`. If you set the aspect ratio explicitly, Panda will recompute your vertical field of view and vertical film size to match.

Panda3D Manual: Orthographic Lenses

The previous page described the PerspectiveLens class, and the various properties of a perspective lens, especially its field of view. There is another kind of lens that is frequently used in 3-D rendering, but it doesn't have a field of view in the same sense at all. This is an **orthographic lens**.



In an orthographic lens, there is no perspective--parallel lines viewed by the lens don't converge; instead, they remain absolutely parallel in the image. While a `PerspectiveLens` closely imitates the behavior of a real, physical camera lens, there is no real lens that does what an `OrthographicLens` does. An `OrthographicLens`, therefore, is most useful for special effects, where you want that unnatural look, or to emulate the so-called 2Â½-D look of several popular real-time strategy games, or strictly to render 2-d objects that shouldn't have any perspective anyway. In fact, the default camera created for the render2d scene graph, which is used to draw all of the onscreen GUI elements in Panda, uses an OrthographicLens.

Since an orthographic lens doesn't have a field of view angle, the `lens.setFov()` method does nothing. To adjust the amount that the orthographic lens sees, you must adjust its film size. And unlike a PerspectiveLens, the film size units are not arbitrary--for an OrthographicLens, the film size should be specified in spatial units, the same units you used to model your scene. For instance, the film size of the OrthographicLens in the above illustration was set with the call `lens.setFilmSize(20, 15)`, which sets the film size to 20 feet by 15 feet--because the scene is modeled in feet, and the panda is about 12 feet tall.

Another nice property of an orthographic lens is that the near distance does not have to be

greater than zero. In fact, it can be negative--you can put the near plane behind the camera plane, which means the camera will see objects behind itself. The OrthographicLens for render2d is set up with `setNearFar(-1000, 1000)`, so it will render any objects with a Z value between -1000 and 1000. (Of course, in render2d almost all objects have a Z value of 0, so it doesn't matter much.)

If you like, you can change the default camera to use an orthographic lens with something like this:

```
lens = OrthographicLens()
lens.setFilmSize(20, 15) # or whatever is appropriate for your
scene
base.cam.node().setLens(lens)
```

Note that using an orthographic lens can be nonintuitive at times--for instance, objects don't get larger as you come closer to them, and they don't get smaller as you get farther away--so it may be impossible to tell your camera is even moving!

<<prev top next>> Search

Panda3D Manual: Sound

Open source Panda3D uses a commercial sound library called FMOD to play sound. If your program is not intended to make any money, and is not charged for in any way, then you may use FMOD in it for free.

FMOD is a very capable multi-platform sound engine. FMOD supports various types of sound files - MP3, WAV, AIFF, MIDI, MOD, WMA, OGG Vorbis. More detailed descriptions of FMOD, and the FMOD licenses are available at their website http://www.fmod.org.

If you do not wish to use FMOD in your progam, remove the fmod.dll and libfmod_audio.dll files from the panda \bin folder. Note that if you do remove FMOD, Panda3D will not be able to play or manipulate sounds.

Also note that even without Panda3D's built-in sound support, Python can still be used play sounds through other libraries, such as Pygame, or PyOpenAL.

Panda3D Manual: Loading and Playing Sounds and Music

Architecture

The implementation of the sound system in Panda3d allows for a division of audio into two categories - Sound Effects and Music. This division is only a convenience for programmers as Panda3d allows these two audio groups to be treated individually. These differences are explained on the next page.

Basics

Loading a Sound

Loading sound is done through the Loader class. (Loading sounds through the 'base' builtin is deprecated and is only present for backwards compatibility.)

In a normal Panda3D environment, loader is a builtin when you import DirectStart like this:

```
</td>
import direct.directbase.
DirectStart
```

Load the sound, by supplying the path to the sound. Here's an example:

```
mySound1 = loader.loadSfx("SoundFile.
wav")
```

These will return an object of the type AudioSound. It is necessary to put the extension in the sound filename.

Playing a Sound

To play sounds you can do the following:

```
mySound.play
()
```

Stopping a Sound

To stop a sound:

```
mySound.stop
()
```

Quering Sound Status

To check the status of a sound:

```
mySound.status
()
```

status() returns 1 if it isn't currently playing and 2 if it is playing.

Setting Volume

The volume can be set between 0 and 1 and will linearly scale between these.

```
mySound.setVolume
(0.5)
```

Panning a Sound

You can change the balance of a sound. The range is between -1.0 to 1.0. Hard left is -1.0 and hard right is 1.0.

```
mySound.setBalance(-
0.5)
```

NOTE !!!

If running Panda from interactive prompt you must call the Update() command, after you play a sound.

```
base.sfxManagerList[n].update
()
```

This is because the update() command is called every frame to reset a sound's channel.

In interactive mode Panda's frame update is suppended, and does not run automatically.

Panda3D Manual: Manipulating Sounds

Basics - Part Duex

Looping a Sound

To cause a sound to loop [IE Cause it repeat once it is finished playing] do the following:

```
mySound.setLoop
(True)
mySound.play()
```

To stop a sound from looping pass False in the setLoop() function.

```
mySound.setLoop
(False)
```

Sounds can also be looped for a certain number of times:

```
mySound.setLoopCount
(n)
```

Where 'n' can be any positive integer. 0 will cause a sound to loop forever. 1 will cause a sound to play only once. >1 will cause a sound to loop that many times.

NOTE Setting a sound's loop count will automatically set a sound's loop flag to 0 or >1 will automatically setLoop() to TRUE.

Notes on Looping Sounds Seamlessly

Looping a sound seamlessly should be as simple as loading the sound, then calling `setLoop` and `play`. However, occassionally Panda users have had difficulty getting sounds to loop seamlessly. The problems have been traced to three(!) different causes:

1. Some MP3 encoders contain a bug where they add blank space at the end of the sound. This causes a skip during looping. Try using a wav instead.
2. Some have tried using Sound Intervals to create a loop. Unfortunately, sound intervals depend on Panda's Thread to restart the sound, and if the CPU is busy, there's a skip. This is not a seamless method, in general. Use `setLoop` instead.

3. There is a bug in Miles sound system, which requires a workaround in Panda3D. At one time, the workaround was causing problems with fmod, until we devised a new workaround. This bug no longer exists, you can ignore it.

So the easiest way to get a reliable looping sound is to use wav files, and to use `setLoop`, not sound intervals. Of course, when it comes time to ship your game, you can convert your sounds to mp3, but before you do, test your mp3 encoder to see if it contains the blank-space bug.

Cueing Time

There are getTime(), setTime() and length() functions for sounds. These will respectively, report the current time position, set the current time position and report the length. All these are in seconds.

```
mySound.length
()
```

Will return the length of a sound file in seconds.

```
mySound.getTime
()
```

Will get the current time the 'playback head' of a sound is at in seconds.

```
mySound.setTime
(n)
```

Will set the 'playhead head' of a sound to n (where is seconds).

NOTE Sounds will start playing IMMEDIATELY after the command is issued. & Calling play() will cause the sound to start over from the beginning.

Changing Playback speed

You can change the rate at which a sound plays back.

To change a sounds playback speed use....

```
mySound.setPlayRate
(N)
```

Where N is any float.

NOTE!!! Negative numbers will play a sound backwards. 0 is 'Pause' a sound.

You can also get a sound's play rate with..

```
mySound.getPlayRate
()
```

<<prev top next>> Search

Panda3D Manual: DSP Effects

Panda3D 1.3.0 can now make use of FMOD-EX's Audio Effects.

Currently the effects available are...

Panda3D ID	Effect
DSPChorus	Chorus
DSPCompressor	Compression
DSPDistortion	Distortion
DSPEcho	Echo [Delay]
DSPFlange	Flange
DSPHighpass	Highpass Filter
DSPItecho	Echo [For Mod/Tracker Files]
DSPItlowpass	Lowpass Filter [For Mod/Tracker Files]
DSPLowpass	Lowpass Filter
DSPNormalize	Noramlize
DSPParameq	Parametric EQ
DSPPitchshift	Pitchshifter
DSPReverb	Reverb

To use an effect do the following.

First create and effect object [we will create an echo for example].

</td>

```
echo = base.sfxManagerList[0].createDsp(base.sfxManagerList[0].
DSPEcho)
```

The effects are constants in the sound manager classes which is why you need to specify "base.sfxManagerList[n].<effect>".
Where N is the respective audio manager [0 for Sound, 1 for Music] and
<effect> is the respective effect as listed in the table above.

Once you have your DSP object you can attach it to a sound.

</td>

```
mySound.addDsp
(echo)
```

and you are ready to go.

Some Useful DSP commands.

At the interactive prompt to get the parameters of a DSP type...

```
</td>
dspEffect.listParameterInfo
()
```

This will list all the DSP parameters and their possible values you can edit.

You edit parameters in your code with the following.

```
</td>
dspEffect.setParameter("nameOfParameter",
value)
```

The name of Parameter must be in Quotes.

You can retrieve the current parameter with...

```
</td>
dspEffect.getParameter
("nameOfParameter")
```

You can turn a paramter on or off with...

```
</td>
dspEffect.setBypass
(n)
```

Where n is a BOOL

You can reset an effect completely to its default values with...

</td>

```
dspEffect.reset
(n)
```

And if you need to detach an effect from a sound use...

</td>

```
mySound.removeDsp
(effect)
```

You can also attach an effect to all the sounds under a Audio Manager with the following...

</td>

```
base.sfxManagerList[n].addDsp
(effect)
```

NOTE!!! An effect attached to a Manager will affect ALL THE SOUNDS under that manager.

Otherwise the same commands for DSP apply.

Panda3D Manual: 3DAudio

As we said Sound Effects and Music are handled separately in Pand3D. Each can handle 16 different sounds. This value is actually set as the audio-cache-limit in the panda config.prc (found in your install directory) and can be changed. There are times where either sound effects, music, or both should be disabled and later enabled. These commands affect entire categories of sounds. Passing True or False in the last 2 functions will disable or enable the respective groups.

```
base.disableAllAudio()
base.enableAllAudio()
base.enableMusic(bEnableMusic)
base.enableSoundEffects
(bEnableSoundEffects)
```

Explained below is some advanced audio processing routines. Sound Effects and Music in code are implemented as AudioManager objects. You can access these audio managers with the following code.

```
soundEffectsMgr = base.sfxManagerList
[0]
musicMgr = base.sfxManagerList[1]
```

Notice that sound effects are the first item in the sfxManagerList and the music is next.

A few things can be controlled with AudioManager objects. Setting the doppler factor, dropoff factor can be set through these objects. Positional audio is implemented through these objects. A wrapper Audio3DManager class has been implemented to help do positional audio. Audio3DManager takes as input an audioManager and a listener for the sound. A listener is the point of reference from where the sound should be heard. For a player in a Panda3D session, this could be the camera. Sounds further away from the camera will not be loud. Objects nearer to the camera will be loud.

```
from direct.showbase import Audio3DManager
audio3d = Audio3DManager.Audio3DManager(base.sfxManagerList[0],
camera)
```

To create a sound that is positional, you need to use the loadSfx() function on the Audio3DManager rather than the normal loader.loadSfx() which is for non-positional sounds. e. g.

```
mySound = audio3d.loadSfx('blue.
wav')
```

Sounds can be attached to objects such that when they move, the sound source will move along with them.

```
audio3d.attachSoundToObject( mySound,
teapot )
```

You can use the Audio3DManager's setSoundVelocity() and setListenerVelocity() to set the velocity of sounds or the listener to get the doppler pitch shifting of moving objects. If you would like the Audio3DManager to help you adjust the velocity of moving objects automatically like it does with their position, you can call setSoundVelocityAuto() or setListenerVelocityAuto () like this:

```
audio3d.setSoundVelocity(sound,velocityVector)
audio3d.setListenerVelocity(velocityVector)

base.cTrav = CollisionTraverser()
audio3d.setSoundVelocityAuto(sound)
audio3d.setListenerVelocityAuto()
```

Currently, for the latter to work, a CollisionTraverser must be attached to base.cTrav as you see in the example. If you already have one assigned to do collision detection that will be sufficient.

The attenuation of moving sounds by distance is based the way sound works in the real world. By default it assumes a scale of 1 panda unit equal to 1 foot. If you use another scale you'll need to use setDistanceFactor to adjust the scale. If you want to position the sounds but don't want the volume to be effected by their distance, you can set the distance factor to 0.

```
audio3d.setDistanceFactor
(scale)
```

Panda3D Manual: Multi-Channel

Write Me

Panda3D Manual: Intervals

Panda3D's Interval system is a sophisticated mechanism for playback of scripted actions. With the use of Intervals, you can build up a complex interplay of animations, sound effects, or any other actions, and play the script on demand.

The core of system is the `Interval` class. There are several different kinds of Intervals, which will be discussed in detail in the following pages, but all of them have in common the following property: each Interval represents an action (or a series of actions) that occur over a specific, finite interval of time (hence the name).

The real power of the Interval system comes from Sequences and Parallels, which are a special kind of Interval that can contain nested Intervals of any kind (including additional Sequences and/or Parallels). By using these grouping Intervals, you can easily assemble complex scripts from the basic atoms.

Using Intervals

In any Panda3D module that uses Intervals, you should first import the interval module:

```
from direct.interval.IntervalGlobal import
*
```

There are a handful of methods that all Intervals have in common.

To start an Interval playing, use one of the following:

```
interval.start()
interval.start(startT, endT,
playRate)
interval.loop()
interval.loop(startT, endT, playRate)
```

The three parameters are optional. The startTime and endTime parameters define the subset of the interval to play; these should be given as times in seconds, measured from the start of the interval. The playRate, if specified, allows you play the interval slower or faster than real time; the default is 1.0, to play at real time.

Normally, an Interval will play to the end and stop by itself, but you can stop a playing Interval prematurely:

```
interval.finish
()
```

This will stop the interval and move its state to its final state, as if it had played to the end. This is a very important point, and it allows you to define critical cleanup actions within the interval itself, which are guaranteed to have been performed by the time the interval is finished.

You can also temporarily pause and resume an interval:

```
interval.pause()
interval.resume
()
```

If you pause an interval and never resume or finish it, the remaining actions in the interval will not be performed.

And you can jump around in time within an interval:

```
interval.setT
(time)
```

This causes the interval to move to the given time, in seconds since the beginning of the interval. The interval will perform all of the actions between its current time and the new time; there is no way to skip in time without performing the intervening actions.

It is legal to set the time to an earlier time; the interval will do its best to reset its state to the previous state. In some cases this may not be possible (particularly if a Function Interval is involved).

Finally, there are a handful of handy query methods:

```
interval.getDuration
()
```

Returns the length of the interval in seconds.

```
interval.getT
()
```

Returns the current elapsed time within the interval, since the beginning of the interval.

```
interval.isPlaying
()
```

Returns true if the interval is currently playing, or false if it was not started, has already finished, or has been explicitly paused or finished.

```
interval.isStopped
()
```

Returns true if the interval has not been started, has already played to its completion, or has been explicitly stopped via finish(). This is not quite the same this as `(not interval.isPlaying())`, since it does not return true for a paused interval.

<<prev top next>> Search

Panda3D Manual: Lerp Intervals

The LerpIntervals are the main workhorse of the Interval system. The word "lerp" is short for "linearly interpolate", and means to smoothly adjust properties, such as position, from one value to another over a period of time. You can use LerpIntervals to move and rotate objects around in your world.

The LerpIntervals are also the most complex of all of the intervals, since there are many different parameters that you might want to specify to control the lerp.

An overview of the NodePath-based LerpIntervals

Most LerpIntervals adjust the various transform properties of a NodePath, such as pos, hpr, and scale, and they all have a similar form. Consider the LerpPosInterval, which will smoothly move a model from one point in space to another:

```
i = LerpPosInterval(model, duration, pos, startPos = None,
                    other = None, blendType = 'noBlend',
                    bakeInStart = 1, fluid = 0, name =
None)
```

The only required parameters are the model whose position is being changed, the length of time to apply the move, and the model's new position. The remaining parameters are all optional, and are often omitted.

Here is a breakdown of what each parameter means:

model	The model whose position is being changed. This should be a NodePath.
duration	The duration of the lerp, in seconds.
pos	The model's target position (the new position it will move to). Usually, this is a Point3(x, y, z), but as a special advanced feature, it might be a Python function that, when called, returns a Point3. If it is a function, it will be called at the time the lerp actually begins to play.
startPos	The starting position of the model at the beginning of the lerp. If this is omitted, the model will start from its current position. As with pos, above, this might be a Python function, which will be called at the time the lerp actually begins. Note that if you intend to move an object from its current position, it is better to omit this parameter altogether rather than try to specify it explicitly with something like `startPos=object.getPos()`, since the latter will be evaluated at the time the interval is created, not when it is played. This is especially true if you plan to embed a series of consecutive LerpIntervals within a Sequence.

other	Normally, this is set to None, to indicate a normal lerp. If a NodePath is passed in, however, it indicates that this is a relative lerp, and the pos and startPos will be computed as a relative transform from that NodePath. The relative transform is recomputed each frame, so if the other NodePath is animating during the lerp, the animation will be reflected here. For this reason, you should not attempt to lerp a model relative to itself.
blendType	This specifies how smoothly the lerp starts and stops. It may be any of the following values:
	'easeIn' The lerp begins slowly, ramps up to full speed, and stops abruptly.
	'easeOut' The lerp begins at full speed, and then slows to a gentle stop at the end.
	'easeInOut' The lerp begins slowly, ramps up to full speed, and then slows to a gentle stop.
	'noBlend' The lerp begins and ends abruptly.
bakeInStart	This is an advanced feature. Normally, this is 1, which means the original, starting position of the model is determined when the interval starts to play, and saved for the duration of the interval. You almost always want to keep it that way. If you pass this as 0, however, the starting position is cleverly re-inferred at each frame, based on the model's current position and the elapsed time in the lerp; this allows your application to move the model even while it is being lerped, and the lerp will adapt. This has nothing to do with controlling when the startPos parameter is evaluated.
fluid	If this is 1, then the lerp uses `setFluidPos()` rather than `setPos()` to animate the model. See Rapidly-Moving Objects. This is meaningful only when the collision system is currently active on the model. Since usually there is no reason to have the collision system active while a model is under direct application control, this parameter is rarely used.
name	This specifies the name of the lerp, and may be useful for debugging. Also, by convention, there may only be one lerp with a given name playing at any given time, so if you put a name here, any other interval with the same name will automatically stop when this one is started. The default is to assign a unique name for each interval.

The rest of the NodePath-based LerpIntervals

Many NodePath properties other than position may be controlled via a lerp. Here is the list of the various LerpIntervals that control NodePath properties:

```
LerpPosInterval(model, duration, pos, startPos)
LerpHprInterval(model, duration, hpr, startHpr)
LerpQuatInterval(model, duration, quat, startQuat)
LerpScaleInterval(model, duration, scale, startScale)
LerpShearInterval(model, duration, shear, startShear)
LerpColorInterval(model, duration, color, startColor)
LerpColorScaleInterval(model, duration, colorScale,
startColorScale)
```

Each of the above has a similar set of parameters as those of LerpPosInterval; they also have

a similar shortcut (e.g. `model.hprInterval()`, etc.)

Finally, there are also a handful of combination LerpIntervals, that perform multiple lerps at the same time. (You can also achieve the same effect by combining several LerpIntervals within a Parallel, but these combination intervals are often simpler to use, and they execute just a bit faster.)

```
LerpPosHprInterval(model, duration, pos, hpr, startPos, startHpr)
LerpPosQuatInterval(model, duration, pos, quat, startPos, startQuat)
LerpHprScaleInterval(model, duration, hpr, scale, startHpr, startScale)
LerpQuatScaleInterval(model, duration, quat, scale, startQuat, startScale)
LerpPosHprScaleInterval(model, duration, pos, hpr, scale, startPos, startHpr,
startScale)
LerpPosQuatScaleInterval(model, duration, pos, quat, scale, startPos,
startQuat, startScale)
LerpPosHprScaleShearInterval(model, duration, pos, hpr, scale, shear,
startPos, startHpr, startScale, startShear)
LerpPosQuatScaleShearInterval(model, duration, pos, quat, scale, shear,
startPos, startQuat, startScale, startShear)
```

Other types of LerpInterval

Beyond animating NodePaths, you can create a LerpInterval that blends any parameter of any object over time. This can be done with a LerpFunctionInterval:

```
def myFunction(t):
  ... do something based on t ...

i = LerpFunc(myFunction, fromData = 0, toData = 1, duration =
0.0,
  blendType = 'noBlend', extraArgs = [], name = None)
```

This advanced interval has many things in common with all of the above LerpIntervals, but instead of directly animating a value, it instead calls the function you specify, passing a single floating-point parameter, t, that ranges from fromData to toData over the duration of the interval.

It is then up to your function to set whatever property of whatever object you like according to the current value of t.

Panda3D Manual: Function Intervals

Function intervals are different from function lerp intervals. While the function lerp interval passes data to a function over a period of time, a function interval will simply execute a function when called. As such, a function intervalâ€™s use really appears when combined with sequences and parallels. The function intervalâ€™s format is simple.

```
intervalName = Func
(myFunction)
```

You pass the function without parentheses (i.e. you pass Func a function pointer). If `myFunction` takes arguments than pass them as arguments to Func as follows:

```
def myFunction(arg1,arg2):
   blah

intervalName = Func(myFunction, arg1,
arg2)
```

Functions cannot be called on their own in sequences and parallels, so it is necessary to wrap them in an interval in order to call them. Since function intervals have no duration, they complete the moment they are called.

Panda3D Manual: Actor Intervals

Actor intervals allow actor animations to be played as an interval, which allows them to be combined with other intervals through sequences and parallels.

The subrange of the animation to be played may be specified via frames (startFrame up to and including endFrame) or seconds (startTime up to and including endTime). It may also be specified with a startFrame or startTime in conjunction with the duration, in seconds. If none of these is specified, then the default is to play the entire range of the animation.

If endFrame is before startFrame, or if the play rate is negative, then the animation will be played backwards.

You may specify a subrange that is longer than the actual animation, but if you do so, you probably also want to specify either loop = 1 or constrainedLoop = 1; see below.

The loop parameter is a boolean value. When it is true, it means that the animation restarts and plays again if the interval extends beyond the animation's last frame. When it is false, it means that the animation stops and holds its final pose when the interval extends beyond the animation's last frame. Note that, in neither case, will the ActorInterval loop indefinitely: all intervals always have a specific, finite duration, and the duration of an ActorInterval is controlled by either the duration parameter, the startTime/endTime parameters, or the startFrame/endFrame parameters. Setting loop=1 has no effect on the duration of the ActorInterval, it only controls what the actor does if you try to play past the end of the animation.

The parameter constrainedLoop works similarly to loop, but while loop = 1 implies a loop within the entire range of animation, constrainedLoop = 1 implies a loop within startFrame and endFrame only. That is, if you specify loop = 1 and the animation plays past endFrame, in the next frame it will play beginning at frame 0; while if you specify constrainedLoop = 1 instead, then the next frame after endFrame will be startFrame again.

All parameters other than the animation name are optional.

```
myInterval = myactor.actorInterval
(
    "Animation Name",
    loop= <0 or 1>,
    contrainedLoop= <0 or 1>,
    duration= D,
    startTime= T1,
    endTime= T2,
    startFrame= N1,
    endFrame= N2,
    playRate = R
    partName = PN,
    lodName = LN,
)
```

Panda3D Manual: Sound Intervals

See Loading and Playing Sounds and Music for basic information on how to load and play sounds.

Sound intervals play sounds from inside an interval. Like actor intervals, sound intervals have a loop parameter and the ability to be paused. Sound intervals also have volume and start time parameters.

```
mySound=loader.loadSfx("mySound.
wav")

myInterval = SoundInterval(
    mySound,
    loop = 0 or 1,
    duration = myDuration,
    volume = myVolume,
    startTime = myStartTime
)
```

The looping provided by the sound interval is not clean. There will be a pause between loops of roughly a tenth of a second. See Loading and Playing Sounds and Music for a better way to loop sounds.

Panda3D Manual: Motion Path and Particle Intervals

Motion paths are an advanced feature of Panda3D, and they are discussed later. Still, motion paths have their own intervals. A motion path interval is much like a function interval in that there are no additional parameters other than the motion path and the NodePath it is affecting.

```
intervalName = MopathInterval(<Motion Path Name>,NodePath,â€˜<Name>â€™)
```

Particle effects can be run from inside intervals as well:

```
intervalName = ParticleInterval
(
    <Particle Effect Name>,
    <Parent>,
    worldRelative = 1,
    loop = 0 or 1,
    duration = myDuration
)
```

Panda3D Manual: Sequences and Parallels

You will need to have this include statement to use Sequences and Parallels.

```
from direct.interval.IntervalGlobal import
*
```

Sequences and Parallels can control when intervals are played. Sequences play intervals one after the other, effectively a â€œdo in orderâ€• command. Parallels are a â€œdo together,â€• playing all intervals at the same time. Both have simple formats, and every kind of interval may be used.

```
mySequence = Sequence(<Interval>,â€¦,<Interval>, name = "Sequence
Name")
myParallel = Parallel(<Interval>,â€¦,<Interval>, name = "Parallel
Name")
```

Sequences and Parallels may also be combined for even greater control. Also, there is a wait interval that can add a delay to Sequences. While it can be defined beforehand, it does not have to be.

```
delay = Wait(2.5)
pandaWalkSeq = Sequence(Parallel(pandaWalk,pandaWalkAnim),
delay,
    Parallel(pandaWalkBack,pandaWalkAnim), Wait(1.0))
```

In the above example, a wait interval is generated. After that, a Sequence is made that uses a Parallel, the defined wait interval, another Parallel, and a wait interval generated in the Sequence. Such Sequences can get very long very quick, so it may be prudent to define the internal Parallels and Sequences before creating the master Sequence.

Panda3D Manual: Projectile Intervals

Projectile intervals are used to move a nodepath through the trajectory of a projectile under the influence of gravity.

```
myInterval = ProjectileInterval(<Node Path>, startPos = Point3(X,Y,Z), endPos
= Point3(X,Y,Z),
  duration = <Time in seconds>, startVel = Point3(X,Y,Z), endZ = Point3(X,Y,
Z),
  gravityMult = <multiplier>, name = <Name>)
```

All parameters don't have to be specified. Here are a combination of parameters that will allow you to create a projectile interval. (If startPos is not provided, it will be obtained from the node's position at the time that the interval is first started. Note that in this case you must provide a duration.)

- startPos, endPos, duration - go from startPos to endPos in duration seconds
- startPos, startVel, duration - given a starting velocity, go for a specific time period
- startPos, startVel, endZ - given a starting velocity, go until you hit a given Z plane

In addition you may alter gravity by providing a multiplier in 'gravityMult'. '2' will make gravity twice as strong, '.5' half as strong.'-1' will reverse gravity.

Here's a little snippet of code that will demonstrate projectile intervals:

```
camera.setPos(0,-45,0)

# load the ball model
self.ball = loader.loadModel("smiley")
self.ball.reparentTo(render)
self.ball.setPos(-15,0,0)

# setup the projectile interval

self.trajectory = ProjectileInterval(self.ball, startPos = Point3(-
15,0,0),
  endPos = Point3(15,0, 0), duration = 1)
self.trajectory.loop()
```

Panda3D Manual: Tasks and Event Handling

Tasks are subroutines that you write that get called by Panda every frame. Event handlers are subroutines that you write that get called by Panda when certain special events occur. Together, these two mechanisms enable you to update your panda world between rendering steps.

Panda3D Manual: Tasks

Tasks are special functions that are called once, each frame, while your application executes. They are similar in concept to threads, but in Panda, tasks are not separate threads; instead, all tasks are run cooperatively, one at a time, within the main thread. This design simplifies game programming considerably by removing the requirement to protect critical sections of code from mutual access.

When you start Panda3D by importing DirectStart, a handful of tasks are created by default, but you are free to add as many additional tasks as you like.

The task function

A task is defined with a function or class method; this function is the main entry point for the task and will be called once per frame while the task is running. By default, the function receives one parameter, which is the task object; the task object carries information about the task itself, such as the amount of time that the task has been running.

Your task function should return when it has finished processing for the frame. Because all tasks are run in the same thread, you must not spend too much time processing any one task function; the entire application will be locked up until the function returns.

The task function may return either `Task.cont` to indicate that the task should be called again next frame, or `Task.done` to indicate that it should not be called again. If it returns None (which is to say, it does not return anything), then the default behavior is to call the task function again.

The below example imports the Task module and shows a function used as a task.

```
from direct.task import Task

#This task runs for two seconds, then prints
done
def exampleTask(task):
  if task.time < 2.0:
    return Task.cont
  print 'Done'
  return Task.done
```

The Task Manager

All tasks are handled through the global Task Manager object, called `taskMgr` in Panda3D. The Task Manager keeps a list of all currently-running tasks. To add your task function to the task list, call `taskMgr.add()` with your function and an arbitrary name for the task.

```
taskMgr.add(exampleTask,
'MyTaskName')
```

To remove the task and stop it from executing, call taskMgr.remove(). You can pass in either the name of the task, or the task object (which was returned by `taskMgr.add()`, above).

```
taskMgr.remove
('MyTaskName')
```

To print the list of tasks currently running, simply print out `taskMgr`. Among your own tasks, you may see the following system tasks listed:

dataloop	Processes the keyboard and mouse inputs
tkloop	Processes Tk GUI events
eventManager	Processes events generated by C++ code, such as collision events
igloop	Draws the scene

Task timing

To see the specific timing information for each task when you print taskMgr, add the following line to your Config.prc file

```
task-timer-verbose
#t
```

(see The Configuration File for config syntax)

The do-later task

A useful special kind of task is the do-later: this is similar to a task, but rather than being called every frame it will be called only once, after a certain amount of time (in seconds) has elapsed. You can, of course, implement a do-later task with a regular task that simply does nothing until a certain amount of time has elapsed (as in the above example), but using a do-later is a much more efficient way to achieve the same thing, especially if you will have many such tasks waiting around.

```
taskMgr.doMethodLater(delayTime, myFunction, 'Task
Name')
```

In this case myFunction must accept a task variable. If you wish to use a function that does not accept a task variable:

```
taskMgr.doMethodLater(delayTime, myFunction, 'Task Name', extraArgs =
[variables])
```

Note: if you wish to call a function which takes no variables simply pass extraArgs = []

<<prev top next>> Search

Panda3D Manual: Event Handlers

Events occur either when the user does something (such as clicking a mouse or pressing a key) or when sent by the script using `messenger.send()`. When an event occurs, Panda's "messenger" will check to see if you have written an "event handler" routine. If so, your event handler will be called. The messenger system is object-oriented, to create an event handler, you have to first create a class that inherits from DirectObject. Your event handler will be a method of your class.

Defining a class that can Handle Events

The first step is to import class DirectObject:

```
from direct.showbase import
DirectObject
```

With DirectObject loaded, it is possible to create a subclass of DirectObject. This allows the class to inherit the messaging API and thus listen for events.

```
class myClassName(DirectObject.
DirectObject):
```

The sample below creates a class that can listen for events. The "accept" function notifies panda that the printHello method is an event handler for the mouse1 event. The "accept" function and the various event names will be explained in detail later.

```
class Hello(DirectObject.DirectObject):
  def __init__(self):
    self.accept('mouse1',self.
printHello)
  def printHello(self):
    print 'Hello!'
h = Hello()
```

Event Handling Functions

Events first go to a mechanism built into panda called the "Messenger." The messenger may accept or ignore events that it receives. If it accepts an event, then an event handler will be called. If ignored, then no handler will be called.

An object may accept an event an infinite number of times or accept it only once. If checking

for an accept within the object listening to it, it should be prefixed with self. If the accept command occurs outside the class, then the variable the class is associated with should be used.

```
myDirectObject.accept('Event Name',myDirectObjectMethod)
myDirectObject.acceptOnce('Event Name',
myDirectObjectMethod)
```

Specific events may be ignored, so that no message is sent. Also, all events coming from an object may be ignored.

```
myDirectObject.ignore('Event
Name')
myDirectObject.ignoreAll()
```

Finally, there are some useful utility functions for debugging. The messenger typically does not print out when every event occurs. Toggling verbose mode will make the messenger print every event it receives. Toggling it again will revert it to the default. A number of methods exist for checking to see what object is checking for what event, but the print method will show who is accepting each event. Also, if accepts keep changing to the point where it is too confusing, the clear method will start the messenger over with a clear dictionary.

```
messenger.toggleVerbose
()
print messenger
messenger.clear()
```

Sending Custom Events

Custom events can be sent by the script using the code

```
messenger.send('Event
Name')
```

A list of parameters can optionally be sent to the event handler. Parameters defined in `accept()` are passed first, and then the parameters defined in `send()`. for example this would print out "eggs sausage foo bar":

```
class Test(DirectObject):
    def __init__(self):
        self.accept('spam',self.OnSpam,['eggs','sausage'])
    def OnSpam(self,a,b,c,d):
        print a,b,c,d
Test()
messenger.send('spam',['foo','bar'])
run()
```

A Note on Object Management

When a DirectObject accepts an event, the messenger retains a reference to that DirectObject. To ensure that objects that are no longer needed are properly disposed of, they must ignore any messages they are accepting.

For example, the following code may not do what you expect:

```
import direct.directbase.DirectStart
from direct.showbase import DirectObject
from pandac.PandaModules import *

class Test(DirectObject.DirectObject):

   def __init__(self):
       self.accept("FireZeMissiles",self._fireMissiles)

   def _fireMissiles(self):
       print "Missiles fired! Oh noes!"


foo=Test() # create our test object

del foo # get rid of our test object

messenger.send("FireZeMissiles") # oops! Why did those missiles fire? run
()
```

Try the example above, and you'll find that the missiles fire even though the object that would handle the event had been deleted.

One solution (patterned after other parts of the Panda3d architecture) is to define a "destroy" method for any custom classes you create, which calls "ignoreAll" to unregister from the event-handler system.

```
import direct.directbase.DirectStart
from direct.showbase import DirectObject
from pandac.PandaModules import *


class Test(DirectObject.DirectObject):


   def __init__(self):
       self.accept("FireZeMissiles",self._fireMissiles)


   def _fireMissiles(self):
       print "Missiles fired! Oh noes!"

   # function to get rid of me
   def destroy(self):
       self.ignoreAll()


foo=Test() # create our test object


foo.destroy() # get rid of our test object del foo


messenger.send("FireZeMissiles") # No missiles fire run
()
```

<<prev top next>> Search

Panda3D Manual: Main Loop

A typical form of a Panda program may my the following:

```
from direct.showbase.DirectObject import DirectObject # To listen for
Events

class World(DirectObject):
   __init__(self):
      #initialize instance self. variables here
   method1():
      # Panda source goes here

w = World()
run() # main loop
```

run is a function that never returns. It is the main loop.

For an alternative, run() could not be called at all. Panda doesn't really need to own the main loop.

Instead, taskMgr.step() can be called intermittently, which will run through one iteration of Panda's loop. In fact, run() is basically just an infinite loop that calls Task.step() repeatedly.

taskMgr.step() must be called quickly enough after the previous call to taskMgr.step(). This must be done quick enough to be faster than the frame rate.

This may useful when an imported third party python module that also has its own event loop wants and wants to be in control of program flow. A third party example may be Twisted, the event-driven networking framework.

The solution to this problem is to create two program threads, one for Panda and one for Twisted. Two queue structures, an input queue and an output queue are created to exist, to pass messages between the two threads.

In the Panda3D area of the code, create a task which runs once per frame, which checks the incoming Queue for messages from the server and updates world objects in Panda as needed. Player inputs call event handlers which place messages on the outgoing queue. In the Twisted area of the code, create a loop which periodically checks the outgoing queue for messages, and processes them as needed. Processed messages are then placed on the incoming queue and sent to the Panda task as they happen. See http://twistedmatrix.com/trac/

Another third party example is wxPython GUI, that is a blending of the wxWidgets C++ class library with the Python programming language. Panda's run() function, and wx's app.MainLoop () method, both are designed to handle all events and never return. They are each supposed to serve as the one main loop of the application. Two main loops can not effectively run an

application.

wxPython also supplies a method that can be called occasionally, instead of a function that never returns. In wx's case, it's app.Dispatch().

A choice can be made whether or not to make wx handle the main loop, and call taskMgr.step () intermittently, or whether or not to make Panda handle the main loop, and call app.Dispatch () intermittently. The better performance choice is to have panda handle the main loop.

In the case that Panda handles the main loop, a task needs to be started to call app.Dispatch() every frame, if needed. Instead of calling wxPython's app.MainLoop(), do something like the following:

```
app = wx.App(0)

def handleWxEvents(task):
   while app.Pending():
      app.Dispatch()
   return Task.cont

taskMgr.add(handleWxEvents,
'handleWxEvents')
run() # panda handles the main loop
```

In the case that wxPython handles the main loop using app.MainLoop(), to keep the framerate quick and reduce the CPU, add sleep(0.001) in the body of the program. This will yield to panda. After the sleep is over, control will return to wxPython. wxPython can then check for user events. wxPython's user generated callback events are generally generated only at infrequent intervals (based on when the user is interacting with the window). This is appropriate for a 2-D application that is completely response-driven, but not very useful for a 3-D application that continues to be active even when a user is not interacting with it.

Panda3D Manual: Fog and Lighting

Fog and lighting are two techniques to add dimension to a virtual space. Panda3D contains a variety of lights that work by vertex lighting. Vertex lighting shades an entire polygon, so the more polygons the world uses, the better the lighting becomes. In areas where lighting is critical, it is best to tessellate the area as much as can be done without killing the frame rate.

Panda3D Manual: Fog

Creating fog is quite simple:

```
myFog = Fog("Fog Name")
myFog.setColor(R,G,B)
myFog.setExpDensity(Float 0 to
1)
render.setFog(myFog)
```

However, there is more here than meets the eye. We have created a *fog node*, which goes into the scene graph. Therefore, the fog has a position, a place where the fog is (conceptually) thickest.

If the fog object is not parented into the scene graph (in the example above, for instance), then the fog's position is ignored, and the fog is camera-relative. Likewise, if the fog is exponential, the fog's position is ignored, and the fog is camera-relative.

The `setFog` directive creates a fog attribute object. Like any Render Attribute, the fog attribute affects the node that it is attached to, and any nodes below it in the scene graph. So you can easily cause only a subset of the objects (or just a single model) to be affected by the fog, by calling `setFog` on the root of the subgraph you want to be affected. To remove the fog attribute later, use the `clearFog` directive:

```
render.clearFog
()
```

While you have fog in effect, it is often desirable to set the background color to match the fog:

```
base.setBackgroundColor
( myFogColor )
```

Panda3D Manual: Lighting

Panda3D defines four different kinds of light objects: point, directional, ambient, and spotlight. Each kind of light has a slightly different effect when it is enabled; the differences between the lights are discussed below.

Each light is a node that should be attached somewhere within the scene graph. All lights have a color, which is specified by `light.setColor(VBase4(r, g, b, a))`. The default color is full white: `setColor(VBase4(1, 1, 1, 1))`. The alpha component is largely irrelevant.

Most lights also have a position and/or orientation, which is determined by the basic scene graph operations like `setPos()`, `setHpr()`, etc. The `lookAt()` method is particularly useful for pointing spotlights and directional lights at a particular object.

Note that, unlike a real, physical light bulb, the light objects are not themselves directly visible. Although you can't see a Panda light itself, you *can* see the effect it has on the geometry around it. If you want to make a light visible, one simple trick is to load a simple model (like a sphere) and parent it directly to the light itself.

In general, you can create a light and add it into the scene graph like any other node. However, because of a problem with multiple inheritance, for the moment you have to call `upcastToPandaNode()` to put a light in the scene graph, like this:

```
dlight = DirectionalLight('dlight')
dlnp = render.attachNewNode(dlight.upcastToPandaNode
())
```

If you forget to use `upcastToPandaNode()`, Panda will almost certainly crash. Note that this will no longer be true in Panda3D 1.1. In version 1.1, you can treat the light as an ordinary node without having to upcast it first.

Simply creating the light and putting it in the scene graph doesn't, by itself, have any visible effect. In order to turn the light on, you have to first decide which object or objects will be illuminated by the light. To do this, use the `nodePath.setLight()` method, which turns on the light for the indicated NodePath and everything below it in the scene graph.

In the simplest case, you want all of your lights to illuminate everything they can, so you turn them on at render, the top of the scene graph:

```
render.setLight
(dlnp)
```

To turn the light off again, you can remove the light setting from render:

```
render.clearLight
(dlnp)
```

You could also apply the `setLight()` call to a sub-node in the scene graph, so that a given light only affects a particular object or group of objects.

Note that there are two (or more) different NodePaths involved here: the NodePath of the light itself, which defines the position and/or orientation of the light, and the NodePath(s) on which you call `setLight()`, which determines what subset of the scene graph the light illuminates. There's no requirement for these two NodePaths to be related in any way.

Point Lights

Point lights are the easiest kind of light to understand: a point light simulates a light originating from a single point in space and shining in all directions, like a very tiny light bulb. A point light's position is important, but its orientation doesn't matter.

```
plight = PointLight('plight')
plight.setColor(VBase4(0.2, 0.2, 0.2, 1))
plnp = render.attachNewNode(plight.upcastToPandaNode
())
plnp.setPos(10, 20, 0)
render.setLight(plnp)
```

Directional Lights

A directional light is an infinite wave of light, always in the same direction, like sunlight. A directional light's position doesn't matter, but its orientation is important. The default directional light is shining down the forward (+Y) axis; you can use `nodePath.setHpr()` or `nodePath.lookAt()` to rotate it to face in a different direction.

```
dlight = DirectionalLight('dlight')
dlight.setColor(VBase4(0.8, 0.8, 0.5, 1))
dlnp = render.attachNewNode(dlight.upcastToPandaNode
())
dlnp.setHpr(0, -60, 0)
render.setLight(dlnp)
```

Ambient Lights

An ambient light is used to fill in the shadows on the dark side of an object, so it doesn't look

completely black. The light from an ambient light is uniformly distributed everywhere in the world, so the ambient light's position and orientation are irrelevant.

Usually you don't want to create an ambient light without also creating one of the other kinds of lights, since an object illuminated solely by ambient light will be completely flat shaded and you won't be able to see any of its details. Typically, ambient lights are given a fairly dark gray color, so they don't overpower the other lights in the scene.

```
alight = AmbientLight('alight')
alight.setColor(VBase4(0.2, 0.2, 0.2, 1))
alnp = render.attachNewNode(alight.upcastToPandaNode
())
render.setLight(alnp)
```

Spotlights

Spotlights represent the most sophisticated kind of light. A spotlight has both a point and a direction, and a field-of-view. In fact, a spotlight contains a lens, just like a camera does; the lens should be a PerspectiveLens and is used to define the area of effect of the light (the light illuminates everything within the field of view of the lens).

Note that the English word "spotlight" is one word, as opposed to the other kinds of lights, which are two words. Thus, the class name is correctly spelled "Spotlight", not "SpotLight".

Also, because a spotlight has a different inheritance than the other kinds of lights, you need to use `upcastToLensNode()` instead of `upcastToPandaNode()`.

```
slight = Spotlight('slight')
slight.setColor(VBase4(1, 1, 1, 1))
lens = PerspectiveLens()
slight.setLens(lens)
slnp = render.attachNewNode(slight.upcastToLensNode
())
slnp.setPos(10, 20, 0)
slnp.lookAt(myObject)
render.setLight(slnp)
```

Panda3D Manual: Example

Here is an example of lighting. There are an ambient light and two directional lights lighting the scene, and a green ambient light that only affects one of the pandas.

```
import direct.directbase.DirectStart
from pandac.PandaModules import *

# Put two pandas in the scene, panda x and panda y.
x= loader.loadModel("panda")
x.reparentTo(render)
x.setPos(10,0,-6)

y= loader.loadModel("panda")
y.reparentTo(render)
y.setPos(-10,0,-6)

# Position the camera to view the two pandas.
base.trackball.node().setPos(0, 60, 0)

# Now create some lights to apply to everything in the scene.

# Create Ambient Light
ambientLight = AmbientLight( 'ambientLight' )
ambientLight.setColor( Vec4( 0.1, 0.1, 0.1, 1 ) )
ambientLightNP = render.attachNewNode( ambientLight.upcastToPandaNode() )
render.setLight(ambientLightNP)

# Directional light 01
directionalLight = DirectionalLight( "directionalLight" )
directionalLight.setColor( Vec4( 0.8, 0.2, 0.2, 1 ) )
directionalLightNP = render.attachNewNode( directionalLight.upcastToPandaNode() )
# This light is facing backwards, towards the camera.
directionalLightNP.setHpr(180, -20, 0)
render.setLight(directionalLightNP)

# Directional light 02
directionalLight = DirectionalLight( "directionalLight" )
directionalLight.setColor( Vec4( 0.2, 0.2, 0.8, 1 ) )
directionalLightNP = render.attachNewNode( directionalLight.upcastToPandaNode() )
# This light is facing forwards, away from the camera.
directionalLightNP.setHpr(0, -20, 0)
render.setLight(directionalLightNP)

# Now attach a green light only to object x.
ambient = AmbientLight('ambient')
ambient.setColor(Vec4(.5,1,.5,1))
ambientNP = x.attachNewNode(ambient.upcastToPandaNode())

# If we did not call setLightOff() first, the green light would add to
# the total set of lights on this object.  Since we do call
```

```
# setLightOff(), we are turning off all the other lights on this
# object first, and then turning on only the green light.
x.setLightOff()
x.setLight(ambientNP)

#run the example
run()
```

Panda3D Manual: Text Rendering

Panda includes support for easily rendering dynamic text onscreen or in the 3-d world. It supports full use of the Unicode character set, so it can easily render international languages (including Asian languages, when used with an appropriate font).

There are three interfaces for creating text, depending on your requirements: the TextNode, which is the fundamental text-rendering class and serves as the implementation for the other two, OnscreenText, a simple high-level wrapper around TextNode, and DirectLabel, which integrates with the rest of the DirectGUI system.

International character sets

By default, Panda assumes the text strings you give it are formatted in the **iso8859** character set, also called **latin-1** on Linux. This is a standard character set that uses one byte per character and supports most Western European languages, and is likely to be the character set you're using anyway. If you have need for more characters than are supported by **iso8859**, you must use a more advanced encoding system like **utf-8** (which may use one, two, or three bytes per character). To use utf-8, put the following in your Config.prc file:

```
text-encoding
utf8
```

And then make sure the strings you pass as text strings are encoded using the "utf-8" encoding method, for instance by saving your Python source files in "utf-8" (this can actually be tricky to do properly in Windows; we recommend you use a non-Microsoft editor such as Emacs or Eclipse to do this).

Panda3D Manual: Text Fonts

Panda3D can render text using a variety of fonts. If your version of Panda3D has been compiled with support for the FreeType library (the default distribution of Panda3D has been), then you can load any TTF file, or any other font file type that is supported by FreeType, directly:

```
font = loader.loadFont('arial.
ttf')
```

The named file is searched for along the model-path, just like a regular egg file. You can also give the full path to the font file if you prefer (but remember to observe the Panda Filename Syntax).

It is also possible to pre-generate a font with the egg-mkfont command-line utility:

```
egg-mkfont -o arial.egg arial.
ttf
```

This will generate an egg file (arial.egg in the above example) and an associated texture file that can then be loaded as if it were a font:

```
font = loader.loadFont('arial.
egg')
```

There are several options you can specify to the egg-mkfont utility; use "egg-mkfont -h" to give a list.

The advantages to pre-generating a font are (a) the resulting egg file can be used by a version of Panda that does not include support for FreeType, and (b) you can apply some painterly effects to the generated texture image using Photoshop or a similar program. On the other hand, you have to decide ahead of time which characters you will want to use from the font; the default is the set of ASCII characters.

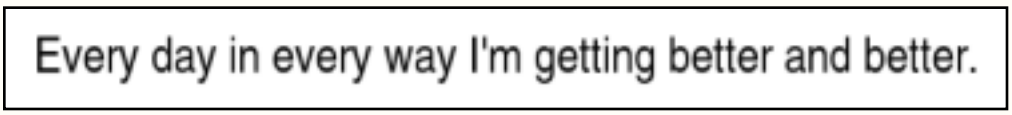
There are three default font files supplied with the default distribution of Panda3D in the models subdirectory; these are "cmr12.egg", a Roman font, "cmss12.egg", a Sans-Serif font, and "cmtt12.egg", a Teletypewriter-style fixed-width font. These three fonts were generated from free fonts provided with the Metafont utility (which is not a part of Panda3D). There is also a default font image which is compiled into Panda if you do not load any other font.

Panda3D Manual: Text Node

The most fundamental way to render text in Panda3D is via the TextNode interface. This may be a little more work than the OnscreenText or DirectLabel objects, but it gives you a lot more control over the appearance of the text.

To use a TextNode, simply create one and call setText() to set the actual text to display, and then parent the TextNode wherever you like (you can put it under aspect2d to make a 2-d onscreen text, or you can put it in the 3-d world for in-the-world text). Note that if you parent the text to render2d or aspect2d, you will probably need to give it a fairly small scale, since the coordinate space of the whole screen in render2d is in the range (-1, 1).

```
text = TextNode('node name')
text.setText("Every day in every way I'm getting better and
better.")
textNodePath = aspect2d.attachNewNode(text)
textNodePath.setScale(0.07)
```

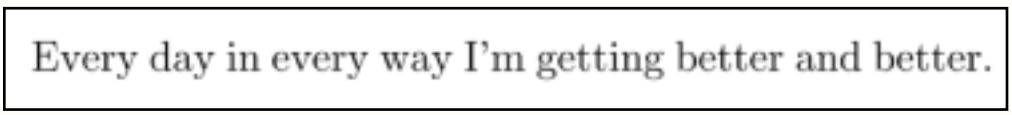


Note that the TextNode constructor takes a string name, which is not related to the text that is to be displayed. Also note that the default text color is white; we show it as black in these examples to make it easier to see on the white background.

There are a large number of properties that you can specify on the TextNode to control the appearance of your text.

Font

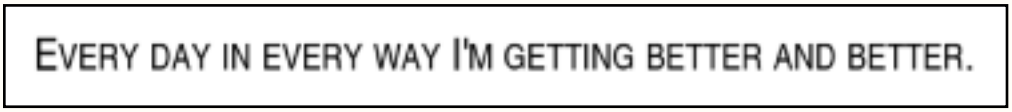
```
cmr12 = loader.loadFont('cmr12.
egg')
text.setFont(cmr12)
```



You may use any font you like, including a TTF file; see Text Fonts.

Small Caps

```
text.setSmallCaps
(1)
```



`setSmallCaps()` accepts a boolean true or false value; set it true to enable small caps mode. In this mode, instead of rendering lowercase letters, the TextNode renders capital letters that are a bit smaller than the true capital letters. This is an especially useful feature if your font of choice doesn't happen to include lowercase letters.

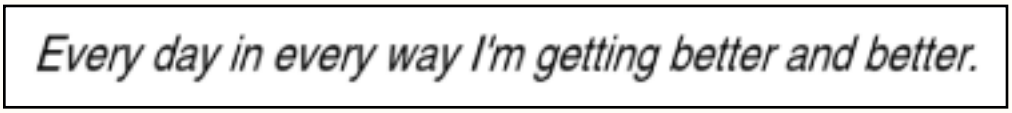
You can also specify the relative scale of the "lowercase" letters:

```
text.setSmallCapsScale
(0.4)
```

Where 1.0 is exactly the same size as the capital letters, and 0.5 is half the size. The default is 0.8.

Slant

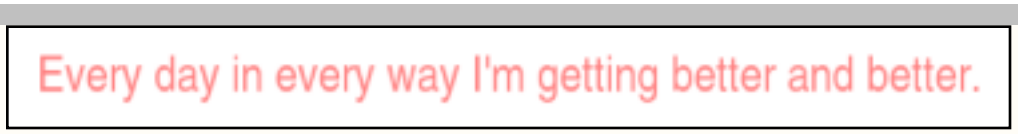
```
text.setSlant
(0.3)
```



Slant can be used to give an effect similar to italicizing. The parameter value is 0.0 for no slant, or 1.0 for a 45-degree rightward slant. Usually values in the range 0.2 to 0.3 give a pleasing effect. You can also use a negative number to give a reverse slant.

Color

```
text.setTextColor(1, 0.5, 0.5,
1)
```



The color is specified with its r, g, b, a components. Note that if a is not 1, the text will be slightly transparent.

Shadow

```
text.setShadow(0.05, 0.05)
text.setShadowColor(0, 0, 0,
1)
```

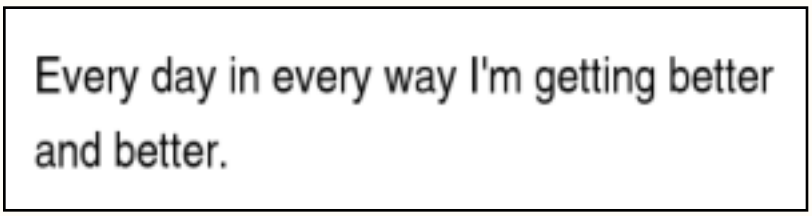
A shadow is another copy of the text, drawn behind the original text and offset slightly to the right and down. It can help make the text stand out from its background, especially when there is not a high contrast between the text color and the background color. (The text color in this example is exactly the same pink color used in the example above, but note how much clearer it is with the shadow.) The downside of a shadow is that it doubles the number of polygons required to render the text.

Setting a shadow requires two calls: `setShadow()` accepts a pair of numbers indicating the distance to shift the shadow right and down, respectively, in screen units; these are usually very small numbers like 0.05. `setShadowColor()` accepts the r, g, b, a color of the shadow; the default is white.

Wordwrap

By default, text will be formatted on one line, unless it includes newline characters. Enabling wordwrap will automatically break the text into multiple lines if it doesn't fit within the specified width.

```
text.setWordwrap
(15.0)
```

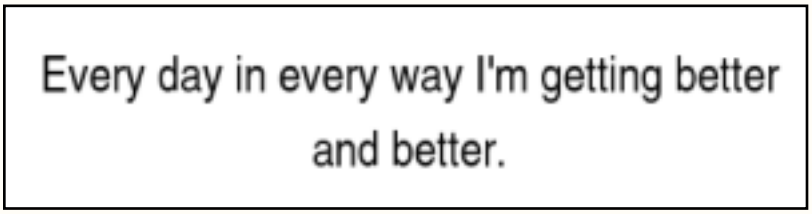


The parameter to `setWordwrap()` should be the maximum width of each line, in screen units.

Alignment

Text is left-aligned by default; that is, it starts at the position you specify with textNodePath.setPos() and goes out to the right from there. If you have multiple lines of text, you may prefer to center the text or right-align it instead:

```
text.setAlign(TextNode.
ACenter)
```

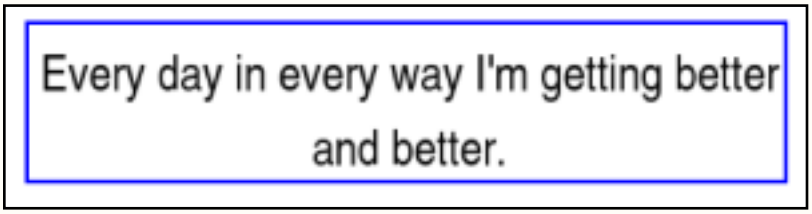


The parameter to `setAlign()` should be one of `TextNode.ALeft`, `TextNode.ACenter`, or `TextNode.ARight`. Note that changing the alignment of the text will shift its position relative to the starting point.

Frame

You can specify that a thin frame should be drawn around the entire text rectangle:

```
text.setFrameColor(0, 0, 1, 1)
text.setFrameAsMargin(0.2, 0.2, 0.1,
0.1)
```

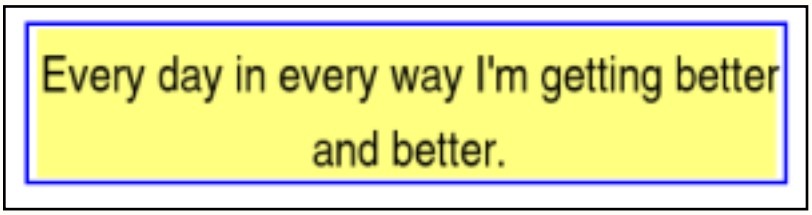


As with the shadow, specifying a frame requires two calls; one to specify the color, and another to specify the dimensions of the frame. The call `setFrameAsMargin()` specifies four parameters, which represent the amount of space to insert between the edge of the text and the frame on the left, right, bottom, and top edges, respectively. All four parameters can be 0.0 to tightly enclose the text (although some fonts render a little bit outside their reported boundaries).

Card

Finally, you can draw a solid card behind the text rectangle:

```
text.setCardColor(1, 1, 0.5, 1)
text.setCardAsMargin(0, 0, 0,
0)
```



This can also help to make the text easier to read when it is against a similar-colored background. Often, you will want the card to be semitransparent, which you can achieve by specifying an alpha value of 0.2 or 0.3 to the `setCardColor()` method.

The parameters to `setCardAsMargin()` are the same as those for `setFrameAsMargin()`, above: the distance to extend the card beyond the left, right, bottom, and top edges, respectively. (In this example, we have both the card and the frame on at the same time, and you can see that the card exactly fits the text, while the frame extends a little bit beyond--showing the effects of the slightly different parameters passed to `setFrameAsMargin()` and `setCardAsMargin()` in this example.)

Picking a Text Node

Strictly speaking, a TextNode has no geometry, so you can't pick it.

There are two possible workarounds.

(1) Create your own card to go behind the TextNode, using e.g. CardMaker. You should be able to say cardMaker.setFrame(textNode.getFrameActual()) to set the card to be the same dimensions as the text's frame. Then you will need to either offset the text a few inches in front of the card to prevent Z-fighting, or explicitly decal the text onto the card, with something like this:

```
card = NodePath(cardMaker.generate()) tnp = card.attachNewNode(textNode) card.
setEffect(DecalEffect.make())
```

(2) Instead of parenting the TextNode directly to the scene, parent the node returned by textNode.generate() instead. This will be a static node that contains the polygons that render the text. If the text changes in the future, it won't automatically update the geometry in this node; you will have to replace this node with the new result of textNode.generate(). But this node will be 100% pickable. In particular, if you have specified textNode.setCardDecal(1), then the first child of the node should be the card geometry.

Panda3D Manual: OnscreenText

The OnscreenText object is a convenience wrapper around TextNode. You can use it as a quick way to put text onscreen without having to go through the trouble of creating a TextNode and setting properties on it. However, it doesn't have the full range of rendering options that you can get with TextNode directly; and it doesn't support the DirectGUI features of a DirectLabel. Use an OnscreenText whenever you want a quick way to display some ordinary text without a lot of fancy requirements.

```
from direct.gui.OnscreenText import OnscreenText
textObject = OnscreenText(text = 'my text string', pos = (-0.5, 0.02), scale =
0.07)
```

The OnscreenText object inherits from NodePath, so all of the standard NodePath operations can be used on the text object. When you are ready to take the text away, use:

```
textObject.destroy
()
```

The following keyword parameters may be specified to the constructor:

text	the actual text to display. This may be omitted and specified later via setText() if you don't have it available, but it is better to specify it up front.
style	one of the pre-canned style parameters defined at the head of OnscreenText.py. This sets up the default values for many of the remaining parameters if they are unspecified; however, a parameter may still be specified to explicitly set it, overriding the pre-canned style.
pos	the x, y position of the text on the screen.
scale	the size of the text. This may either be a single float (and it will usually be a small number like 0.07) or it may be a 2-tuple of floats, specifying a different x, y scale.
fg	the (r, g, b, a) foreground color of the text. This is normally a 4-tuple of floats or ints.
bg	the (r, g, b, a) background color of the text. If the fourth value, a, is nonzero, a card is created to place behind the text and set to the given color.
shadow	the (r, g, b, a) color of the shadow behind the text. If the fourth value, a, is nonzero, a little drop shadow is created and placed behind the text.
frame	the (r, g, b, a) color of the frame drawn around the text. If the fourth value, a, is nonzero, a frame is created around the text.
align	one of TextNode.ALeft, TextNode.ARight, or TextNode.ACenter.
wordwrap	either the width to wordwrap the text at, or None to specify no automatic word wrapping.

font	the font to use for the text.
parent	the NodePath to parent the text to initially; the default is aspect2d.
mayChange	pass true if the text or its properties may need to be changed at runtime, false if it is static once created (which leads to better memory optimization). The default is false.

Panda3D Manual: Embedded Text Properties

It's possible to change text properties in the middle of a paragraph. To do this, you must first define the different kinds of text properties you might want to change to, and give each one a name; then you can embed special characters in your text string to switch these pre-defined text properties in and out.

Defining your text properties

You can create any number of `TextProperties` objects. Each of these can store a different set of text properties, any of the text properties that you can set directly on a TextNode. These include the per-character attributes such as font, color, shadow, and slant, as well as per-line formatting properties such as alignment and wordwrap.

```
tpRed = TextProperties()
tpRed.setTextColor(1, 0, 0, 1)
tpSlant = TextProperties()
tpSlant.setSlant(0.3)
tpRoman = TextProperties()
tpRoman.setFont(cmr12)
```

You can set as many or as few different attributes on any one TextProperties object as you like. Only the attributes you specify will be applied to the text string; any attributes you don't mention will remain unchanged when you apply the TextProperties. In the above example, applying the tpRed structure to a particular text string will only change the text color to red; other properties, such as slant, shadow, and font, will remain whatever they were previously. Similarly for tpSlant, which only changes the slant, and tpRoman, which only changes the font.

Registering the new TextProperties objects

You will need a pointer to the global `TextPropertiesManager` object:

```
tpMgr = TextPropertiesManager.getGlobalPtr()
```

After you have created your TextProperties objects, you must register each one with the TextPropertiesManager, under a unique name:

```
tpMgr.setProperties("red", tpRed)
tpMgr.setProperties("slant", tpSlant)
tpMgr.setProperties("roman", tpRoman)
```

Referencing the TextProperties in text strings

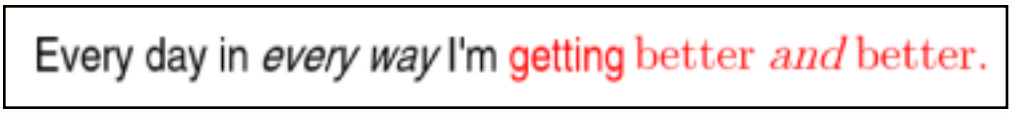
Now you're ready to put the special characters in your text string to activate these mode changes. To do this, you will use the special character '\1', or the ASCII 0x01 character. You use the \1 character twice, as a kind of quotation mark before and after the name you have used above to register your TextProperties object, e.g. '\1red\1' to activate tpRed, or '\1slant\1' to activate tpSlant.

The sequence '\1red\1' acts as a *push* operation. It applies tpRed to the current text properties, but also remembers the previous properties. To go back to the previous properties, use the character '\2' by itself. You can nest property changes like this; each '\2' will undo the most recent '\1name\1' that is still in effect.

The following text string:

```
text.setText("Every day in \1slant\1every way\2 I'm \1red\1getting \1roman
\1better \1slant\1and\2 better.\2\2")
```

Looks like this:



You can use these special characters in any Panda construct that generates text, including TextNode, OnscreenText, and any DirectGui object.

Panda3D Manual: DirectGUI

Panda3D comes with a set of tools for the creation of a graphical interface for any program. The DirectGui system is used to create buttons, labels, text entries, and frames within the program. All of these items can be decorated with text, images, and 3D graphics. Commands may be associated with these items as well. Since these objects inherit from the NodePath class, anything done to a NodePath may be done to them, such as show()/hide(), setPos(), posInterval(), and so on. Also, since DirectGui objects are by default parented to the node aspect2d, they will stay on the screen no matter how the user navigates through the world.

The direct-gui-edit option in the Config.prc file allows the user to use the middle mouse button to move around widgets, and resize them while holding the control key; this is very useful to lay a screen out during development. If you need to turn this ability off for an individual object, set its `enableEdit` keyword parameter to False.

All of the DirectGui objects are constructed in a similar way:

```
from direct.gui.DirectGui import *
myObject = Directxxxxxx(keyword=value, keyword=value, ...)
```

Each DirectGui object may contain any of four fundamental pieces that determine its appearance. There may be an optional **text**, an optional **geom**, an optional **image**, and an optional **frame**.

A DirectGui's **text** label may be any arbitrary text string, and whatever text string you supply is automatically created using the OnscreenText interface and centered on the object. You can specify the text string using the `text` keyword. You can also specify further parameters to control the appearance or placement of the text using the form `text_parameter`, where `parameter` is any valid keyword to the OnscreenText constructor.

A DirectGui's **geom** can be any NodePath that you design, to represent the appearance of the gui object. Typically, this will be a model that you created via the command `egg-texture-cards`, based on a texture that you painted by hand. Using this interface, you can completely customize the look of the DirectGui object to suit your needs. You can specify the geom object using the `geom` keyword, and like the text parameter, you can also control the geom's placement using keywords like `geom_parameter`.

The **image** is less often used. It is the filename of a texture image (or an already-loaded Texture object). It is intended for displaying a simple texture image for which you don't already have a model created via `egg-texture-cards`. A default card will be created to display this texture, with a bounding box of (-1, 0, -1) to (1, 0, 1); that is, a square with sides of length 2 units, centered on the origin. You can position and scale this card with the keywords `image_pos` and `image_scale`.

Finally, the DirectGui may have a **frame** created for it. This is typically a gray rectangular background with an optional bevel. There are a handful of different frame styles; you can use the `relief` keyword to select from one of the available styles; your choices are SUNKEN, RAISED, GROOVE, or RIDGE. You can also specify `relief = None` to avoid creating a frame polygon altogether (this is commonly done when you have specified your own geom object with the `geom` keyword).

The overall size of the DirectGui object is controlled with the `frameSize` keyword. This is a four-tuple of floating-point numbers of the form (left, right, bottom, top), which specifies the bounding box region of the DirectGui object. That is, the lower-left corner will be at position (left, 0, bottom), and the upper-right will be at (right, 0, top). Note that these values represent coordinates from the origin of the frame. Setting the frameSize to (-0.1, 0.1, -0.1, 0.1), for instance, will create a box, 0.2 units wide and 0.2 units in height, with 0,0 being the center of the frame located at `pos` on the screen.

The `frameSize` keyword is optional. If you omit it, the default frameSize is computed based on the bounding box of the **text**, **geom**, and/or **image** that you have specified.

The following is a list of keywords that are typically available to DirectGui objects of all kinds. Individual kinds of DirectGui objects may add more options to this list, but these keywords are not repeated on each of the following pages, for brevity:

Keyword	Definition	Value
text	Text to be displayed on the object	String
text_bg	Background color of the text on the object	(R,G,B,A)
text_fg	Color of the text	(R,G,B,A)
text_pos	Position of the displayed text	(x,z)
text_roll	Rotation of the displayed text	Number
text_scale	Scale of the displayed text	(sx,sz)
text_*	Parameters to control the appearance of the text	Any keyword parameter appropriate to OnscreenText.
frameSize	Size of the object	(Left,Right,Bottom,Top)
frameColor	Color of the object's **frame**	(R,G,B,A)
relief	Relief appearance of the **frame**	SUNKEN, RAISED, GROOVE, RIDGE, FLAT, or None
invertedFrames	If true, switches the meaning of SUNKEN and RAISED	0 or 1
borderWidth	If relief is SUNKEN, RAISED, GROOVE, or RIDGE, changes the size of the bevel	(Width,Height)
image	An **image** to be displayed on the object	image filename or Texture object
image_pos	Position of the displayed image	(x,y,z)
image_hpr	Rotation of the displayed image	(h,p,r)
image_scale	Scale of the displayed image	(sx,sy,sz)

geom	A **geom** to represent the object's appearance	NodePath
geom_pos	Position of the displayed geom	(x,y,z)
geom_hpr	Rotation of the displayed geom	(h,p,r)
geom_scale	Scale of the displayed geom	(sx,sy,sz)
pos	Position of the object	(X,Y,Z)
hpr	Orientation of the object	(H,P,R)
scale	Scale of the object	Number
pad	When frameSize is omitted, this determines the extra space around the **geom** or **text**'s bounding box by which to expand the default **frame**	(Width,Height)
state	The initial state of the object	NORMAL or DISABLED
frameTexture	Texture applied directly to the **frame** generated when relief is FLAT	image filename or Texture object
enableEdit	Affects direct-gui-edit functionality	0 or 1
suppressKeys	If 1, suppresses triggers of global keyboard-related Panda events (not part of the GUI system)	0 or 1
suppressMouse	If 1, suppresses triggers of global mouse-related Panda events (e.g. camera controls)	0 or 1
sortOrder	Specifies render order for overlapping objects. Higher numbers are drawn in front of lower numbers.	Number
textMayChange	Whether the text of an object can be changed after creation	0 or 1

Remember that the axes for Panda3D use x for left and right, y for in and out of the screen, and z for up and down. An object's **frame** is always in the background of the object. The **geom**, if any, is shown in front of the frame, and **text** is shown in front of the geom.

It is possible to change most of these values after object creation, using:

```
myDirectObject['keyword'] =
value
```

Most properties can be updated in this way, although position and other transform-related values cannot be updated via the keyword parameters--attempts to update them will silently fail. Instead, use the NodePath methods to change the object's transform.

Some types of updates, such as changing the text or the geom, may also change the size of the object. If you change any of these properties after the object has been created, it is

necessary to tell the object to re-determine its size:

```
myDirectObject.resetFrameSize
()
```

If you don't do this, you may find, for example, that a button isn't clickable because it believes it has a zero-width frame.

To permanently remove a DirectGUI object, you should use the method:

```
myDirectObject.destroy
()
```

It is not sufficient to simply call removeNode(), since the DirectGUI system adds a number of messenger hooks that need to be cleaned up. However, if you have a hierarchy of DirectGUI objects, for instance a number of buttons parented to a frame, it is sufficient to call destroy() only on the topmost object; it will propagate downwards.

Panda3D Manual: DirectButton

DirectButton is a DirectGui object that will respond to the mouse and can execute an arbitrary function when the user clicks on the object. This is actually implemented by taking advantage of the "state" system supported by every DirectGui object.

Each DirectGui object has a predefined number of available "states", and a current state. This concept of "state" is completely unrelated to Panda's FSM object. For a DirectGui object, the current state is simply as an integer number, which is used to select one of a list of different NodePaths that represent the way the DirectGui object appears in each state. Each DirectGui object can therefore have a completely different appearance in each of its states.

Most types of DirectGui objects do not use this state system, and only have one state, which is state 0. The DirectButton is presently the only predefined object that has more than one state defined by default. In fact, DirectButton defines four states, numbered 0 through 3, which are called *ready, press, rollover,* and *disabled,* in that order. Furthermore, the DirectButton automatically manages its current state into one of these states, according to the user's interaction with the mouse.

With a DirectButton, then, you have the flexibility to define four completely different NodePaths, each of which represents the way the button appears in a different state. Usually, you want to define these such that the *ready* state is the way the button looks most of the time, the *press* state looks like the button has been depressed, the *rollover* state is lit up, and the *disabled* state is grayed out. In fact, the DirectButton interfaces will set these NodePaths up for you, if you use the simple forms of the constructor (for instance, if you specify just a single text string to the `text` parameter).

Sometimes you want to have explicit control over the various states, for instance to display a different text string in each state. To do this, you can pass a 4-tuple to the text parameter (or to many of the other parameters, such as relief or geom), where each element of the tuple is the parameter value for the corresponding state, like this:

```
b = DirectButton(text = ("OK", "click!", "rolling over",
"disabled"))
```

The above example would create a DirectButton whose label reads "OK" when it is not being touched, but it will change to a completely different label as the mouse rolls over it and clicks it.

Another common example is a button you have completely customized by painting four different texture maps to represent the button in each state. Normally, you would convert these texture maps into an egg file using `egg-texture-cards` like this:

```
egg-texture-cards -o button_maps.egg -p 240,240 button_ready.png button_click.png
button_rollover.png button_disabled.png
```

And then you would load up the that egg file in Panda and apply it to the four different states like this:

```
maps = loader.loadModel('button_maps.egg')
b = DirectButton(geom = (maps.find('**/button_ready'),
                         maps.find('**/button_click'),
                         maps.find('**/button_rollover'),
                         maps.find('**/button_disabled')))
```

You can also access one of the state-specific NodePaths after the button has been created with the interface `myButton.stateNodePath[stateNumber]`. Normally, however, you should not need to access these NodePaths directly.

The following are the DirectGui keywords that are specific to a DirectButton. (These are in addition to the generic DirectGui keywords described on the previous page.)

Keyword	Definition	Value
command	Command the button performs when clicked	Function
extraArgs	Extra arguments to the function specified in command	[Extra Arguments]
commandButtons	Which mouse button must be clicked to do the command	LMB, MMB, or RMB
rolloverSound	The sound made when the cursor rolls over the button	AudioSound instance
clickSound	The sound made when the cursor clicks on the button	AudioSound instance
pressEffect	Whether or not the button sinks in when clicked	<0 or 1>

Example

```
import direct.directbase.DirectStart
from direct.gui.OnscreenText import OnscreenText
from direct.gui.DirectGui import *

#add some text
bk_text = "This is my Demo"
textObject = OnscreenText(text = bk_text, pos = (0.95,-0.95),
scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#callback function to set  text
def setText():
        bk_text = "Button Clicked"
        textObject.setText(bk_text)

#add button
b = DirectButton(text = ("OK", "click!", "rolling over", "disabled"),scale=.05,command=setText)

#run the tutorial
run()
```

Note that you will not be able to set the text unless the mayChange flag is 1. This is an optimisation, which newbies might miss.

Panda3D Manual: DirectCheckButton

DirectCheckButtons are similar to buttons, except they represent a binary state that is toggled when it is clicked. Their usage is almost identical to regular buttons, except that the text area and box area can be modified separately.

Keyword	Definition	Value
text_scale	Scale of the displayed text	(sx,sz)
indicatorValue	The initial boolean state of the checkbox	0 or 1
boxImage	Image on the checkbox	Image Path
boxImageColor	Color of the image on the box	(R,G,B,A)
boxImageScale	Scale of the displayed image	Number
boxPlacement	Position of the box relative to the text area	'left','right'
boxRelief	Relief appearance of the checkbox	SUNKEN or RAISED
boxBorder	Size of the border around the box	Number
command	Command the button performs when clicked (0 or 1 is passed, depending on the state)	Function
extraArgs	Extra arguments to the function specified in command	[Extra Arguments]
commandButtons	Which mouse button must be clicked to do the command	LMB, MMB, or RMB
rolloverSound	The sound made when the cursor rolls over the button	Sound File Path
clickSound	The sound made when the cursor clicks on the button	Sound File Path
pressEffect	Whether or not the button sinks in when clicked	<0 or 1>

Example

```
import direct.directbase.DirectStart
from direct.gui.OnscreenText import OnscreenText
from direct.gui.DirectGui import *

#add some text
bk_text = "This is my Demo"
textObject = OnscreenText(text = bk_text, pos = (0.95,-0.95),
scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#callback function to set  text
def setText(status):
        if(status):
                bk_text = "Checkbox Selected"
        else:
                bk_text = "Checkbox Not Selected"
        textObject.setText(bk_text)

#add button
b = DirectCheckButton(text = "CheckButton" ,scale=.05,command=setText)
```

```
#run the tutorial
run()
```

A note on boxImage and other box* keywords

Just as DirectButton may be passed a 4-tuple of values to be used in the four button states, the box* keyword arguments may be supplied with multiple entries to denote the unchecked and checked state. To supply arguments to be used in the two states of the checkbox, construct a 3-tuple of values with a 'None' in the final entry, i.e. (unchecked, checked, None). For example, to set two different images for the unchecked and checked states:

```
boxImage = ("pathToDisabledImage.jpg","pathToEnabled.jpg",None)
```

<<prev top next>> Search

Panda3D Manual: DirectDialog

DirectDialog objects are popup windows to alert or interact with the user. It is invoked just like the other DirectGUI objects, but it also has some unique keywords. Integral to DirectDialog are dialogName, buttonTextList, buttonImageList, and buttonValueList. The dialogName should ideally be the name of the NodePath created to hold the object. The button lists contain the various properties of the buttons within the dialog box. No maximum number of buttons needs to be declared.

Panda3D contains a number of shortcuts for common dialog options. For example, rather than specifying the rather common text list ("Yes","No"), there is a YesNoDialog that functions exactly like a normal dialog but has buttonTextList already defined. The other similar dialogs are OkCancelDialog, OkDialog, RetryCancelDialog, and YesNoCancelDialog.

Keyword	Definition	Value
dialogName	Name of the dialog	String
buttonTextList	List of text to show on each button	[Strings]
buttonGeomList	List of geometry to show on each button	[NodePaths]
buttonImageList	List of images to show on each button	[Image Paths]
buttonValueList	List of values sent to dialog command for each button. If value is [] then the ordinal rank of the button is used as its value	[Numbers]
buttonHotKeyList	Shortcut key for each button (the button must have focus)	[Characters]
buttonSize	4-tuple used to specify custom size for each button (to make bigger then geom/text for example)	(Left,Right,Bottom,Top)
topPad	Extra space added above text/geom/image	Number
midPad	Extra space added between text/buttons	Number
sidePad	Extra space added to either side of text/buttons	Number
buttonPadSF	Scale factor used to expand/contract button horizontal spacing	Number
command	Callback command used when a button is pressed. Value supplied to command depends on values in buttonValueList	Function
extraArgs	Extra arguments to the function specified in command	[Extra Arguments]
fadeScreen	If 1, fades screen to black when the dialog appears	0 or 1

YesNo Dialog Example

```
import direct.directbase.DirectStart
from direct.gui.OnscreenText import OnscreenText
from direct.gui.DirectGui import *
from direct.task import Task
from direct.actor import Actor
from direct.interval.IntervalGlobal import *

#add some text
bk_text = "DirectDialog- YesNoDialog Demo"
textObject = OnscreenText(text = bk_text, pos = (0.85,0.85),
scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#add some text
output = ""
textObject = OnscreenText(text = output, pos = (0.95,-0.95),
 scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#callback function to set  text
def itemSel(arg):
        if(arg):
                output = "Button Selected is: Yes"
        else:
                output = "Button Selected is: No"
        textObject.setText(output)

#create a frame
dialog = YesNoDialog(dialogName="YesNoCancelDialog", command=itemSel)

base.camera.setPos(0,-20,0)
#run the tutorial
run()
```

Panda3D Manual: DirectEntry

The DirectEntry creates a field that accepts text entered by the user. It provides a blinking cursor and support for backspace and the arrow keys. It can accept either a single line of text, with a fixed width limit (it doesn't scroll), or it can accept multiple word-wrapped lines.

Keyword	Definition	Value
initialText	Initial text to load in the field	String
entryFont	Font to use for text entry	Font object
width	Width of field in screen units	Number
numLines	Number of lines in the field	Integer
cursorKeys	True to enable the use of cursor keys (arrow keys)	0 or 1
obscured	True to hide passwords, etc.	0 or 1
command	Function to call when enter is pressed (the text in the field is passed to the function)	Function
extraArgs	Extra arguments to the function specified in command	[Extra Arguments]
rolloverSound	The sound made when the cursor rolls over the field	Sound File Path
clickSound	The sound made when the cursor inside the field	Sound File Path
focus	Whether or not the field begins with focus (focusInCommand is called if true)	0 or 1
backgroundFocus	If true, field begins with focus but with hidden cursor, and focusInCommand is not called	0 or 1
focusInCommand	Function called when the field gains focus	Function
focusInExtraArgs	Extra arguments to the function specified in focusInCommand	[Extra Arguments]
focusOutCommand	Function called when the field loses focus	Function
focusOutExtraArgs	Extra arguments to the function specified in focusOutCommand	[Extra Arguments]

Example

```
import direct.directbase.DirectStart
from direct.gui.OnscreenText import OnscreenText
from direct.gui.DirectGui import *

#add some text
bk_text = "This is my Demo"
textObject = OnscreenText(text = bk_text, pos = (0.95,-0.95),
scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#callback function to set  text
def setText(textEntered):
        textObject.setText(textEntered)

#clear the text
def clearText():
        b.enterText('')

#add button
b = DirectEntry(text = "" ,scale=.05,command=setText,
initialText="Type Something", numLines = 2,focus=1,focusInCommand=clearText)

#run the tutorial
run()
```

This example implements a text entry widget typically seen in web pages.

Panda3D Manual: DirectFrame

A frame is a container object for multiple DirectGUI objects. This allows for the control over several objects that are reparented to the same frame. When DirectGUI objects are parented to a frame, they will be positioned relative to the frame.

DirectFrame has no unique keywords, since it is simply used to arrange other objects.

Panda3D Manual: DirectLabel

Labels are like buttons, but they do not respond to mouse-clicks. This means a DirectLabel is basically just a text string, and in that respect is similar to OnscreenText, except that the DirectLabel integrates better with the rest of the DirectGUI system (and the constructor accepts more DirectGUI-like options).

If you are making a text label to appear on a DirectFrame or in conjunction with DirectGUI somehow, you should probably use a DirectLabel. For all other uses of text, you would probably be better off using OnscreenText or a making a plain Text Node instead.

DirectLabel's only unique keyword can be used if you want to create a label with multiple states. If you set the value of activeState to a nonexistent state, the label will disappear, since the default state is undefined.

Keyword	Definition	Value
activeState	The "active" or normal state of the label	Number

Panda3D Manual: DirectOptionMenu

The DirectOptionMenu class models a popup menu with an arbitrary number of items. It is composed of the menu bar, the popup marker, and the popup menu itself. The popup menu appears when the menu is clicked on and disappears when the user clicks again; if the click was inside the popup, the selection changes. By default, the text on the menu changes to whatever item is currently selected. The attributes that affect the appearance of the menu bar don't apply to the popup. Make sure to specify the items option or it may crash.

Keyword	Definition	Value
textMayChange	Whether the text on the menu changes with the selection	0 or 1
initialitem	The index of the item that appears next to the cursor when the popup appears	Number
items	List of items in the popup menu	[Strings]
command	Function called when an item is selected (the item is passed in as a parameter)	Function
commandButtons	Which mouse button must be clicked to open the popup	LMB, MMB, or RMB
extraArgs	Extra arguments to the function specified in command	[Extra Arguments]
highlightColor	Color of highlighted text	(R,G,B,A)
highlightScale	Scale of highlighted text	(Width,Height)
rolloverSound	The sound made when the cursor rolls over the button	Sound File Path
clickSound	The sound made when the cursor clicks on the button	Sound File Path
popupMarkerBorder	Use width to change the size of the border around the popup marker	(Width,Height)

Example

```
import direct.directbase.DirectStart
from direct.gui.OnscreenText import OnscreenText
from direct.gui.DirectGui import *

#add some text
bk_text = "DirectOptionMenu Demo"
textObject = OnscreenText(text = bk_text, pos = (0.85,0.85),
scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#add some text
output = ""
textObject = OnscreenText(text = output, pos = (0.95,-0.95),
 scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#callback function to set  text
def itemSel(arg):
        output = "Item Selected is: "+arg
        textObject.setText(output)

#create a frame
menu = DirectOptionMenu(text="options", scale=0.1,items=["item1","item2","item3"],initialitem=2,
highlightColor=(0.65,0.65,0.65,1),command=itemSel)

#run the tutorial
run()
```

This is a simple demonstration of the DirectOptionMenu.

Dynamic Updating of a Menu

```
import direct.directbase.DirectStart
from direct.gui.OnscreenText import OnscreenText
from direct.gui.DirectGui import *

#add some text
bk_text = "DirectOptionMenu Demo"
textObject = OnscreenText(text = bk_text, pos = (0.85,0.85),
scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#add some text
output = ""
textObject = OnscreenText(text = output, pos = (0.95,-0.95),
scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#callback function to set  text
def itemSel(arg):
        if(arg != "Add"): #no need to add an element
                output = "Item Selected is: "+arg
                textObject.setText(output)
        else: #add an element
                tmp_menu = menu['items']
                new_item = "item"+str(len(tmp_menu))
                tmp_menu.insert(-1,new_item) #add the element before add
                menu['items'] = tmp_menu
                #set the status message
                output = "Item Added is: "+new_item
                textObject.setText(output)

#create a frame
menu = DirectOptionMenu(text="options", scale=0.1,items=["item1","item2","item3","Add"],
initialitem=2,highlightColor=(0.65,0.65,0.65,1),command=itemSel,textMayChange=1)

#run the tutorial
run()
```

In this example we add an item to the menu whenever the Add item is selected.

Panda3D Manual: DirectScrolledList

DirectScrolledLists create a list of DirectGuiWidgets. Each object is created individually and can then be added to the list. Some useful methods are:

```
addItem(item, refresh)
getItemIndexForItemID(self, itemID)
getSelectedIndex(self)
getSelectedText(self)
removeItem(self, item, refresh)
scrollBy(self, delta)
scrollTo(self, index, centered)
scrollToItemID(self, itemID,
centered)
selectListItem(self, item)
```

In the above methods, item is a new item, either a string or a DirectGUI element, and itemID is an arbitrary identification number for each item (but not necessarily a zero-based index number). The itemID for a new item is the return value of addItem(). The centered parameter is a boolean; if true, the list scrolls so that the given index is centered, otherwise it scrolls so that the index is on top of the list.

The items option should either be a list of DirectGUI items or of strings. If strings are used, the itemMakeFunction (and possibly itemMakeExtraArgs) option should be defined to point to a function that will take the supplied string, the index, and the extra args as parameters and return a DirectGUI object to insert into the list. If items is a list of strings and itemMakeFunction is not specified, it will create a list of DirectLabels. itemMakeFunction is redundant if a list of DirectGUI objects is passed into items to begin with.

DirectScrolledLists come with two scroll buttons for navigating through the list. By default, they both start at (0,0,0) relative to the list with size 0, and their positions and size need to be set explicitly. You can set any of the values except relief appearance as you initialize the list:

```
myScrolledList = DirectScrolledList(incButton_propertyName = value,
decButton_propertyName = value)
```

incButton scrolls forward through the list; decButton backward. Note that this only works for initialization. To change a property of the scroll buttons later in the program, you must use:

```
myScrolledList.incButton['propertyName'] = value
myScrolledList.decButton['propertyName'] =
value
```

Unlike the first method, this does not work with NodePath options like position; use setPos(...) for that.

For example, the following creates a scrolled list and resizes and moves the buttons appropriately.

```
myScrolledList = DirectScrolledList(incButton_pos= (.5,0,0), incButton_text =
"Inc", decButton_pos= (-.5,0,0), decButton_test = "Dec")
myScrolledList.incButton['frameSize'] = (0, 0.2, 0, 0.2)
myScrolledList.decButton['framesize'] = (0, 0.2, 0, 0.2)
myScrolledList.incButton['text_scale'] = .2
myScrolledList.decButton['text_scale'] = .2
```

Keyword	Definition	Value
command	Function called when the list is scrolled	Function
extraArgs	Extra arguments to the function specified in command	[Extra Arguments]
text_scale	Scale of the displayed text	(sx,sz)
items	List of the objects to appear in the ScrolledList	[DirectGUI items] or [Strings]
numItemsVisible	Number of items visible at a time	Number
forceHeight	Forces the height of the list to be a given number	Number
itemMakeFunction	Function that makes DirectGUI items out of strings	Function
itemMakeExtraArgs	Extra arguments to the function in itemMakeFunction	[Extra Arguments]

Panda3D Manual: DirectWaitBar

DirectWaitBars are similar to status bars; use them to indicate a slow process gradually completing (e.g. a loading screen). It has various options for both the background bar and the loading bar that fills up as the process progresses. You can call finish() to automatically fill up the bar, or use:

```
myWaitBar['value'] = number
```

to set the value (it ranges from 0 to 100 by default).

Keyword	Definition	Value
value	Initial value of the loading bar (from 0 to 100)	Number
range	The maximum value of the loading bar	Number
barColor	The color of the loading bar	(R,G,B,A)
barRelief	The relief appearance of the loading bar	SUNKEN or RAISED
barBorderWidth	If barRelief is SUNKEN, RAISED, GROOVE, or RIDGE, changes the size of the loading bar's bevel	(Width,Height)
relief	The relief appearance of the background bar	SUNKEN or RAISED

Example

```
import direct.directbase.DirectStart
from direct.gui.OnscreenText import OnscreenText
from direct.gui.DirectGui import *

#add some text
bk_text = "This is my Demo"
textObject = OnscreenText(text = bk_text, pos = (0.95,-0.95),
scale = 0.07,fg=(1,0.5,0.5,1),align=TextNode.ACenter,mayChange=1)

#callback function to set  text
def incBar(arg):
        bar['value'] += arg
        text = "Progress is:"+str(bar['value'])+'%'
        textObject.setText(text)

#create a frame
frame = DirectFrame(text="main",scale=0.001)
#add button
bar = DirectWaitBar(text = "" , value=50,pos=(0,.4,.4))

#create 4 buttons
button_1 = DirectButton(text="+1",scale=0.05,pos=(-.3,.6,0), command=incBar,extraArgs = [1])
button_10 = DirectButton(text="+10",scale=0.05,pos=(0,.6,0), command=incBar,extraArgs = [10])
button_m1 = DirectButton(text="-1",scale=0.05,pos=(0.3,.6,0), command=incBar,extraArgs = [-1])
button_m10 = DirectButton(text="-10",scale=0.05,pos=(0.6,.6,0), command=incBar,extraArgs = [-10])

#run the tutorial
run()
```

Panda3D Manual: DirectSlider

Use a DirectSlider to make a slider, a widget that allows the user to select a value between a bounded interval. DirectSlider is available beginning in Panda3D 1.1.

A DirectSlider consists of a long bar, by default horizontal, along with a "thumb", which is a special button that the user may move left or right along the bar. The normal DirectGui parameters such as frameSize, geom, and relief control the look of the bar; to control the look of the thumb, prefix each of these parameters with the prefix "thumb_", e.g. `thumb_frameSize`.

If you want to get (or modify) the current value of the slider (by default, the range is between 0 and 1), use `mySlider['value']`.

Keyword	Definition	Value
value	Initial value of the slider	Default is 0
range	The (min, max) range of the slider	Default is (0, 1)
pageSize	The amount to jump the slider when the user clicks left or right of the thumb	Default is 0.1
orientation	The orientation of the slider	HORIZONTAL or VERTICAL
command	Function called when the value of the slider changes (takes no arguments)	Function
extraArgs	Extra arguments to the function specified in command	[Extra Arguments]
thumb_geom, thumb_relief, thumb_text, thumb_frameSize, etc.	Parameters to control the look of the thumb	Any parameters appropriate to DirectButton

Panda3D Manual: DirectScrollBar

A DirectScrollBar is similar to the "scroll bar" widget commonly used by the user to page through a large document. DirectScrolledBar is available beginning in Panda3D 1.1. It consists of a long trough, a thumb that slides along the trough, and a pair of buttons on either side of the trough to scroll one line at a time. A DirectScrollBar can be oriented either vertically or horizontally.

The DirectScrollBar is similar in function to DirectSlider, but it is specifically designed for scrolling through a large window. In fact, a pair of DirectScrollBars is used to implement the DirectScrolledFrame, which manages this scrolling functionality automatically. (Because DirectScrolledFrame exists, you will probably not need to create a DirectScrollBar directly, unless you have some custom purpose that requires a scroll bar.)

DirectScrollBar has many things in common with DirectSlider. Like DirectSlider, the normal DirectGui parameters such as frameSize, geom, and relief control the look of the trough. You can control the look of the thumb by prefixing each of these parameters with the prefix "thumb_", e.g. `thumb_frameSize`; similarly, you can control the look of the two scroll buttons by prefixing these with "incButton_" and "decButton_". You can retrieve or set the current position of the thumb with `myScrollBar['value']`.

Keyword	Definition	Value
value	Initial position of the thumb	Default is 0
range	The (min, max) range of the thumb	Default is (0, 1)
pageSize	The amount to jump the thumb when the user clicks left or right of the thumb; this also controls the width of the thumb when resizeThumb is True	Default is 0.1
scrollSize	The amount to move the thumb when the user clicks once on either scroll button	Default is 0.01
orientation	The orientation of the scroll bar	HORIZONTAL or VERTICAL
manageButtons	Whether to automatically adjust the buttons when the scroll bar's frame is changed	True or False
resizeThumb	Whether to adjust the width of the thumb to reflect the ratio of pageSize to the overall range; requires manageButtons to be True as well	True or False

command	Function called when the position of the thumb changes (takes no arguments)	Function
extraArgs	Extra arguments to the function specified in command	[Extra Arguments]
thumb_geom, thumb_relief, thumb_text, thumb_frameSize, etc.	Parameters to control the look of the thumb	Any parameters appropriate to DirectButton
incButton_geom, incButton_relief, incButton_text, incButton_frameSize, etc.	Parameters to control the look of the lower or right scroll button	Any parameters appropriate to DirectButton
decButton_geom, decButton_relief, decButton_text, decButton_frameSize, etc.	Parameters to control the look of the upper or left scroll button	Any parameters appropriate to DirectButton

Panda3D Manual: DirectScrolledFrame

The DirectScrolledFrame is a special variant of DirectFrame that allows the user to page through a larger frame than would otherwise fit onscreen. The DirectScrolledFrame consists of a small onscreen frame which is actually a window onto a potentially much larger virtual canvas; the user can scroll through this canvas through the use of one or two DirectScrollBars on the right and bottom of the frame. DirectScrolledFrame is available beginning with Panda3D version 1.1.

The `frameSize` parameter controls the size and placement of the visible, onscreen frame; use the `canvasSize` parameter to control the size of the larger virtual canvas.

You can then parent any widgets you like to the NodePath returned by `myFrame.getCanvas()`. The DirectGui items you attach to this canvas NodePath will be visible through the small window; you should position them within the virtual canvas using values within the coordinate range you established via the `canvasSize` parameter.

By default, the scroll bars are automatically created with the DirectScrolledFrame and will be hidden automatically when they are not needed (that is, if the virtual frame size is equal to or smaller than the onscreen frame size). You can adjust either frame size at runtime and the scroll bars will automatically adjust as needed. If you would prefer to manage the scroll bars yourself, you can set one or both of `manageScrollBars` and `autoHideScrollBars` to False.

Keyword	Definition	Value
canvasSize	Extents of the virtual canvas	(Top, left, bottom, right)
manageScrollBars	Whether to automatically position and scale the scroll bars to fit along the right and bottom of the frame	True or False
autoHideScrollBars	Whether to automatically hide one or both scroll bars when not needed	True or False
scrollBarWidth	Specifies the width of both scroll bars at construction time	Default is 0.08
verticalScroll_relief, verticalScroll_frameSize, etc.	Parameters to control the look of the vertical scroll bar	Any parameters appropriate to DirectScrollBar
horizontalScroll_relief, horizontalScroll_frameSize, etc.	Parameters to control the look of the horizontal scroll bar	Any parameters appropriate to DirectScrollBar

Panda3D Manual: Render Effects

There are a number of special render effects that may be set on scene graph nodes to change the way they render. This includes BillboardEffect, Compass Effect, DecalEffect, PolylightEffect, and ShowBoundsEffect.

RenderEffect represents render properties that must be applied as soon as they are encountered in the scene graph, rather than propagating down to the leaves. This is different from RenderAttrib, which represents properties like color and texture that don't do anything until they propagate down to a GeomNode.

You should not attempt to create or modify a RenderEffect directly; instead, use the make() method of the appropriate kind of effect you want. This will allocate and return a new RenderEffect of the appropriate type, and it may share pointers if possible. Do not modify the new RenderEffect if you wish to change its properties; instead, create a new one.

Once you have created a render Effect, you need to decide what it should affect. If you have an effect that should affect evereything in the scene the nodepath in the next line of code is "render". If you only want it to affect specific objects, choose the appropriate place in the scene graph.

```
NodePath.node().setEffect(<Render Effect>)
```

Panda3D Manual: Compass Effects

A CompassEffect causes a node to inherit its rotation (or pos or scale, if specified) from some other reference node in the graph, or more often from the root.

In its purest form, a CompassEffect is used to keep the node's rotation fixed relative to the top of the scene graph, despite other transforms that may exist above the node. Hence the name: the node behaves like a magnetic compass, always pointing in the same direction.

As an couple of generalizing extensions, the CompassEffect may also be set up to always orient its node according to some other reference node than the root of the scene graph. Furthermore, it may optionally adjust any of pos, rotation, or scale, instead of necessarily rotation; and it may adjust individual pos and scale components. (Rotation may not be adjusted on an individual component basis, that's just asking for trouble.)

Be careful when using the pos and scale modes. In these modes, it's possible for the CompassEffect to move its node far from its normal bounding volume, causing culling to fail. If this is an issue, you may need to explicitly set a large (or infinite) bounding volume on the effect node.

```
Effect=CompassEffect.make
(NodePath)
```

Panda3D Manual: Billboard Effects

A billboard is a special effect that causes a node to rotate automatically to face the camera, regardless of the direction from which the camera is looking. It is usually applied to a single textured polygon representing a complex object such as a tree. Judicious use of billboards can be an effective way to create a rich background environment using very few polygons.

Panda indicates that a node should be billboarded to the camera by storing a BillboardEffect on that node. Normally, you do not need to create a BillboardEffect explicitly, since there are a handful of high-level methods on NodePath that will create one for you:

```
myNodePath.setBillboardAxis()
myNodePath.setBillboardPointWorld
()
myNodePath.setBillboardPointEye()
```

Each of the above calls is mutually exclusive; there can be only one kind of billboard effect on a node at any given time. To undo a billboard effect, use:

```
myNodePath.clearBillboard
()
```

The most common billboard type is an axial billboard, created by the `setBillboardAxis()` method. This kind of billboard is constrained to rotate around its vertical axis, so is usually used to represent objects that are radially symmetric about the vertical axis (like trees).

Less often, you may need to use a point billboard, which is free to rotate about any axis. There are two varieties of point billboard. The world-relative point billboard always keeps its up vector facing up, i.e. along the Z axis, and is appropriate for objects that are generally spherical and have no particular axis of symmetry, like clouds. The eye-relative point billboard, on the other hand, always keeps its up vector towards the top of the screen, no matter which way the camera tilts, and is usually used for text labels that float over objects in the world.

There are several more options available on a BillboardEffect, but these are rarely used. If you need to take advantage of any of these more esoteric options, you must create a BillboardEffect and apply it to the node yourself:

```
myEffect=BillboardEffect.make(
  upVector= vec3,
  eyeRelative= bool,
  axialRotate= bool,
  offset= float,
  lookAt= nodepath,
  lookAtPoint= point3
)
myNodePath.node().setEffect
(myEffect)
```

Panda3D Manual: Texturing

At its simplest, texturing merely consists of applying a texture in your modeling program. When you export the model, pay attention to the relative path between the egg file you create, and the image files. That relative path is encoded into the egg file. When panda attempts to load the egg file, it will look in the same position relative to the egg file. Panda can load JPG, PNG, TIF, and a number of other file formats.

More advanced texturing methods are described in the sections that follow.

Panda3D Manual: Simple Texturing

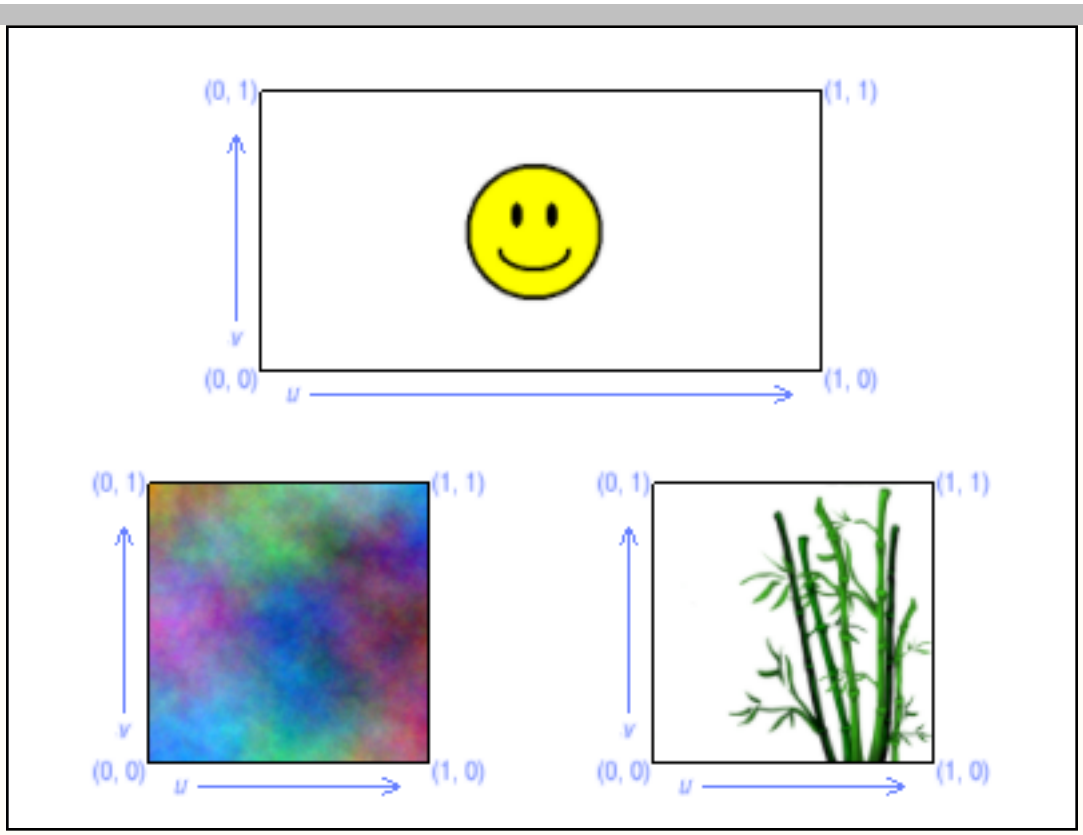
A **texture map** or **texture image** is a two-dimensional image file, like a JPEG or a Windows BMP file, that is used to apply color to a 3-D model. It is called a "texture" because one of the earliest uses of this technique was to apply an interesting texture to walls and floors that would otherwise be one flat, plastic-looking color. Nowadays texturing is so common in 3-D applications that it is often the only thing used to apply color to models--without texture maps, many models would simply be white.

There are a vast array of rendering effects that can be achieved with different variants on texturing. Before you can learn about them, it is important to understand the basics of texturing first.

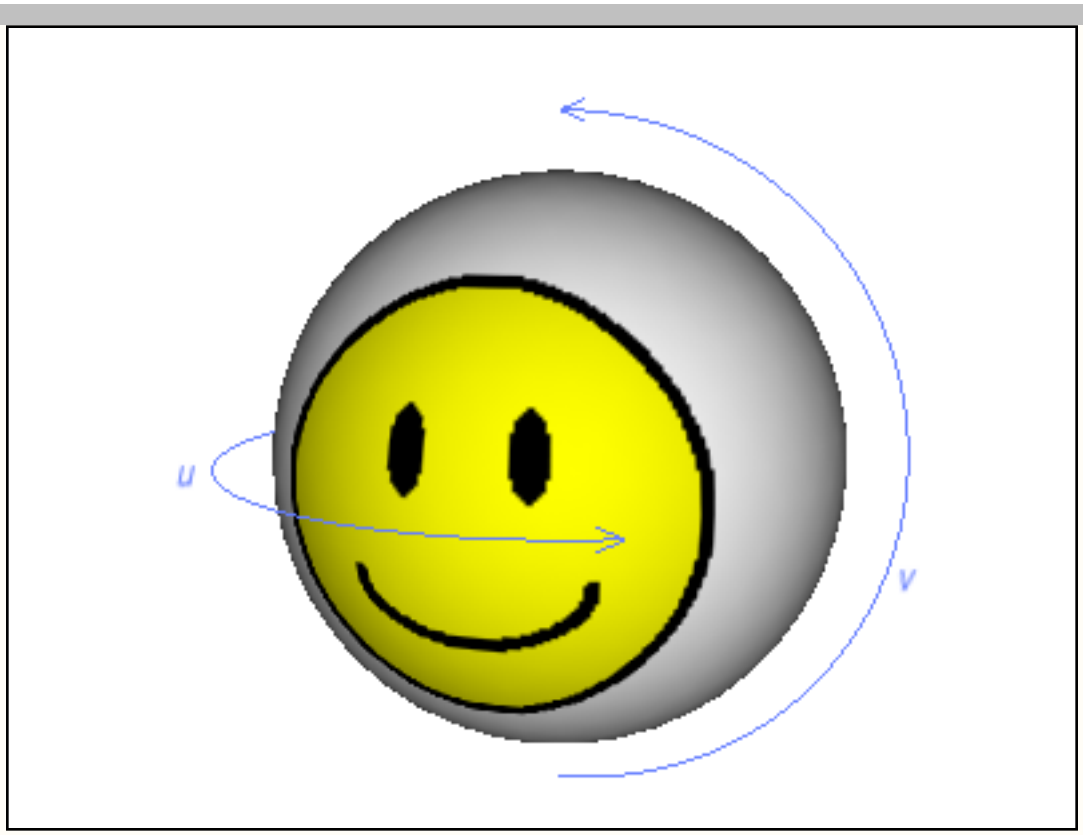
In simple texturing--by far the most common form--you can think of the texture map as a layer of paint that is applied to the model. In order for the graphics hardware to know in what direction the paint should be applied, the model must have been created with **texture coordinates**--a special (u, v) coordinate pair that is associated with each vertex of your model. Each vertex's (u, v) texture coordinates place the vertex at a particular point within the texture map, in the same way that the vertex's (x, y, z) coordinates place the vertex at a particular point in 3-D space.

These texture coordinates are sometimes called **uv's** because of the (u, v) name of the coordinate pair. Almost any modeling package that you might use to create a model can create texture coordinates at the same time, and many do it without even asking.

By convention, every texture map is assigned a (u, v) coordinate range such that the u coordinate ranges from 0 to 1 from right to left, and the v coordinate ranges from 0 to 1 from bottom to top. This means that the bottom-left corner of the texture is at coordinate (0, 0), and the top-right corner is at (1, 1). For instance, take a look at some typical texture maps:



It is the (u, v) texture coordinates that you assign to the vertices that determine how the texture map will be applied to your model. When each triangle of your model is drawn, it is drawn with the colors from your texture map that fall within the same triangle of vertices in (u, v) texture map space. For instance, the sample smiley.egg model that ships with Panda has its vertices defined such that the *u* coordinate increases from 0 to 1 around its diameter, and the *v* coordinate increases from 0 at the bottom to 1 at the top. This causes the texture image to be wrapped horizontally around the sphere:



Note that the (u, v) range for a texture image is always the same, 0 to 1, regardless of the size of the texture.

<<prev top next>> Search

Panda3D Manual: Choosing a Texture Size

Most graphics hardware requires that your texture images always be a size that is a power of two in each dimension. That means you can use any of the following choices for a texture size: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, or so on (but unless you have a *really* high-end card, you'll probably need to stop there).

The textures don't usually have to be *square:* they don't have to have the same size in both dimensions. But each dimension does usually have to be a power of two. So 64 × 128 is all right, for instance, or 512 × 32, or 256 × 256. But you can't make a texture image that is 200 × 200 pixels, since 200 isn't a power of two.

By default, Panda3D will automatically rescale any texture image down to the nearest smaller power of two when you read it from disk, so you usually don't have to think about this--but your application will load faster if you scale your textures properly in the first place.

Some newer graphics cards can render textures that are not a power of two. If you have one of these cards and you don't want Panda3D to scale your textures, you can disable this automatic scaling by putting the following line in your Config.prc:

```
textures-power-2 none
```

Note that some cards *appear* to be able to render non-power-of-two textures, but the driver is really just scaling the textures at load time. With cards like these, you're better off letting Panda do the scaling, or dynamic textures may render *very* slowly.

Other choices for `textures-power-2` are `down` (to scale down to the nearest smaller power of two, the default) or `up` (to scale up to the next larger power of two).

Finally, note that the size you choose for the texture image has nothing to do with the size or shape of the texture image onscreen--that's controlled by the size and shape of the polygon you apply it to. Making a texture image larger won't make it make it appear larger onscreen, but it will tend to make it crisper and more detailed. Similarly, making a texture smaller will tend to make it fuzzier.

Panda3D Manual: Texture Wrap Modes

As described earlier, the *(u, v)* texture coordinates that you assign to your vertices are what determines how the texture fits on your geometry. Often, you will use texture coordinates that always fall within the range [0, 1], which is the complete range of the pixels of your texture image. However, it is also legal to use texture coordinates that go outside this range; you can have negative values, for instance, or numbers higher than 1.

So if the texture image is only defined over the range [0, 1], what does the texture look like outside this range? You can specify this with the **texture wrap mode**.

```
texture.setWrapU(wrapMode)
texture.setWrapV(wrapMode)
```

The wrapMode parameter is specified separately for the *u* and *v* directions (there is also a `setWrapW()` for 3-D textures, but that's an advanced topic). The wrapMode may be any of the following values:

`Texture.WMRepeat`	The texture image repeats to infinity.
`Texture.WMClamp`	The last pixel of the texture image stretches out to infinity.
`Texture.WMBorderColor`	The color specified by `texture.setBorderColor()` is used to fill the space.
`Texture.WMMirror`	The texture image flips back-and-forth to infinity.
`Texture.WMMirrorOnce`	The texture image flips backwards, once, and then the "border color" is used.

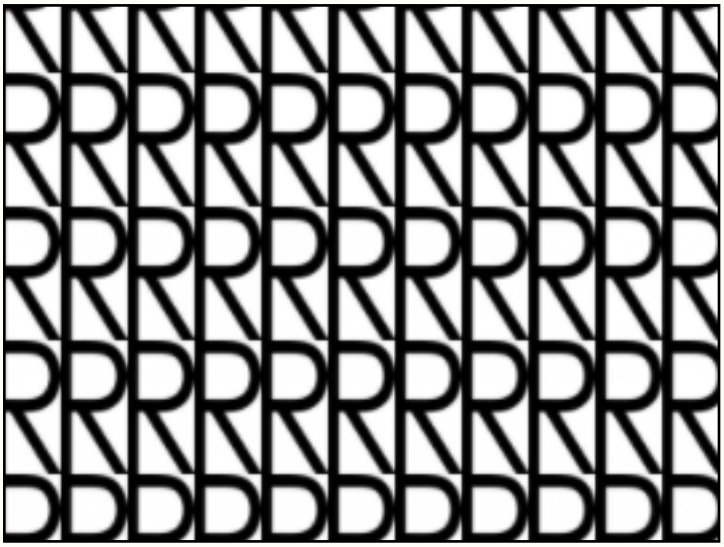
The default wrap mode is `WMRepeat`.

Consider the following simple texture image:

We will apply this texture in the center of a large polygon whose texture coordinates range considerably farther than [0, 1] in both directions.

WMRepeat

```
texture.setWrapU(Texture.WMRepeat)
texture.setWrapV(Texture.WMRepeat)
```



`WMRepeat` mode is often used to tile a relatively small texture over a large surface.

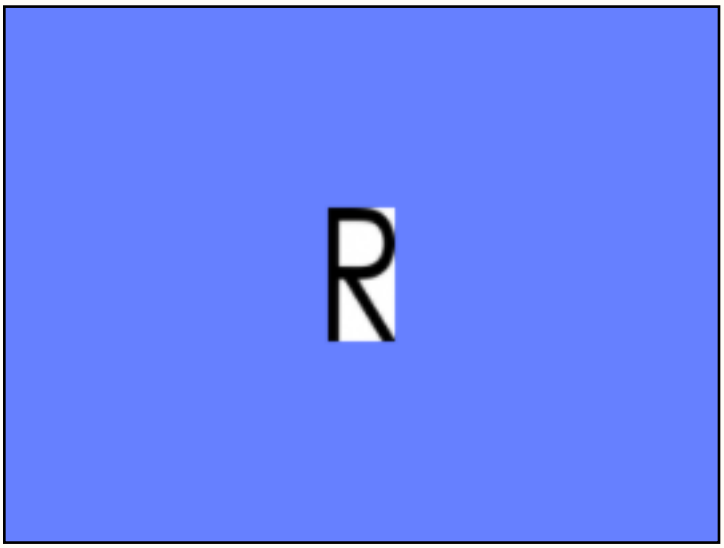
WMClamp

```
texture.setWrapU(Texture.WMClamp)
texture.setWrapV(Texture.WMClamp)
```

`WMClamp` mode is rarely used on large polygons because, frankly, it looks terrible when the pixels stretch out to infinity like this; but this mode is usually the right choice when the texture exactly fills its polygon (see *One caution about a common wrap error,* below).

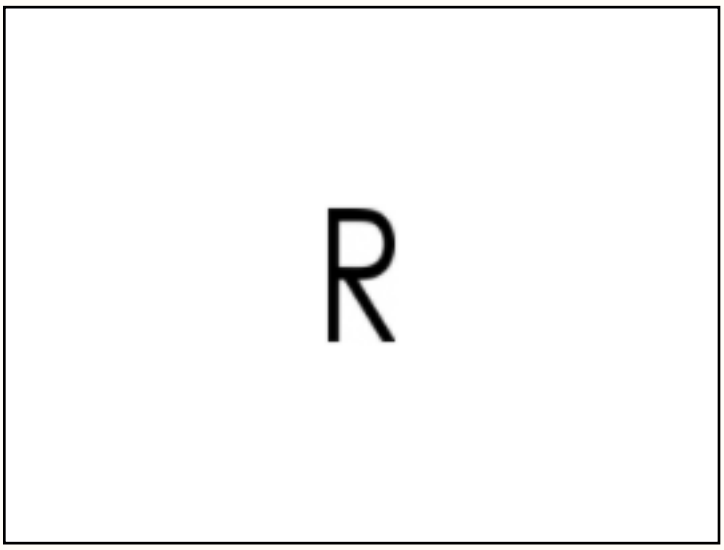
WMBorderColor

```
texture.setWrapU(Texture.WMBorderColor)
texture.setWrapV(Texture.WMBorderColor)
texture.setBorderColor(VBase4(0.4, 0.5, 1, 1))
```



The above blue color was chosen for illustration purposes; you can use any color you like for the border color. Normally, you would use the background color of the texture as the border color, like this:

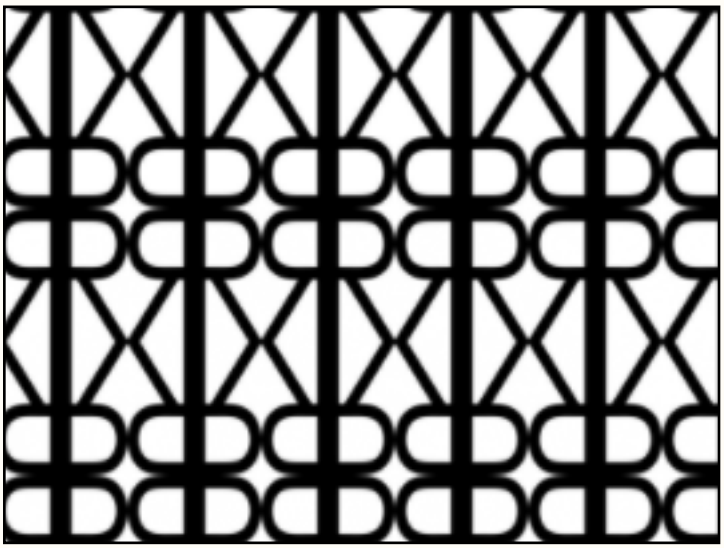
```
texture.setWrapU(Texture.WMBorderColor)
texture.setWrapV(Texture.WMBorderColor)
texture.setBorderColor(VBase4(1, 1, 1, 1))
```



Some very old graphics drivers don't support `WMBorderColor`. In this case, Panda3D will fall back to `WMClamp`, which will look similar as long as there is a sufficient margin of background color around the edge of your texture (unlike our sample texture, which goes all the way out the edge).

WMMirror

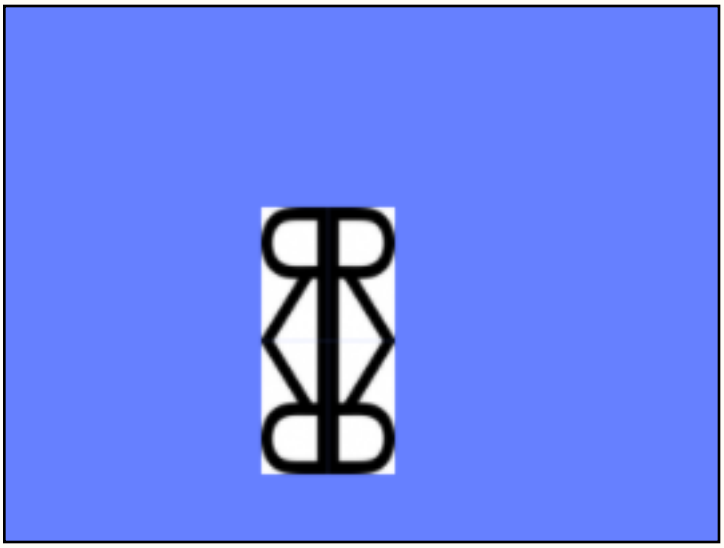
```
texture.setWrapU(Texture.WMMirror)
texture.setWrapV(Texture.WMMirror)
```



Many older graphics drivers do not support `WMMirror`. In this case, Panda3D will fall back to `WMRepeat`.

WMMirrorOnce

```
texture.setWrapU(Texture.WMMirrorOnce)
texture.setWrapV(Texture.WMMirrorOnce)
texture.setBorderColor(VBase4(0.4, 0.5, 1, 1))
```



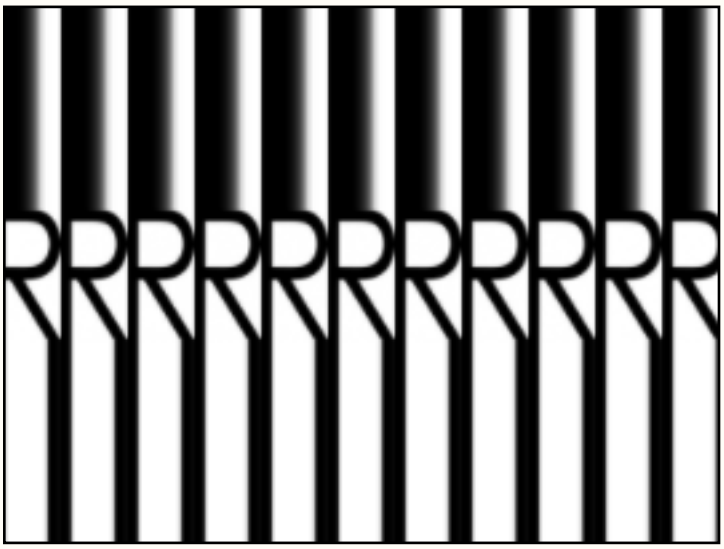
Few graphics drivers support `WMMirrorOnce`. In this case, Panda3D will fall back to

`WMBorderColor`.

Setting different wrap modes

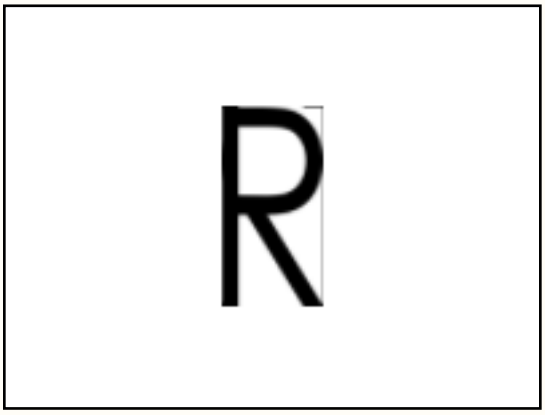
It is possible to set different wrap modes in the *u* and *v* directions:

```
texture.setWrapU(Texture.WMRepeat)
texture.setWrapV(Texture.WMClamp)
```



One caution about a common wrap mode error

When you apply a texture that is intended to exactly fill a polygon--that is, the texture coordinates range from 0 to 1, but no further--you should usually set its wrap mode to *clamp*. This is because if you let it keep the default value of *repeat*, the color may bleed in from the opposite edge, producing a thin line along the edge of your polygon, like this:



This is a particularly common error with a texture that is painted as an alpha cutout, where there is an image with a fully transparent background: you will often see an thin, barely-visible edge floating along the top (for instance) of the polygon. This edge is actually the bottom edge of the texture bleeding onto the top, because the designer specified `WMRepeat` instead of the

correct mode, `WMClamp`.

Panda3D Manual: Texture Filter Types

It's rare that the pixels of a texture image match one-to-one with actual screen pixels when a texture is visible onscreen. Usually, it is the case that either a single pixel of the texture is stretched over multiple screen pixels (**texture magnification**--the texture image is stretched bigger), or the opposite, that multiple pixels of a texture contribute to the color of a single screen pixel (**texture minification**--the texture image is squished smaller). Often, a single polygon will have some texture pixels that need to be magnified, and some pixels that need to be minified (the graphics card can handle both cases on a single polygon).

You can control how the texture looks when it is magnified or minified by setting its **filter type**.

```
texture.setMagfilter(filterType)
texture.setMinfilter(filterType)
```

There is a separate filterType setting for magnification and for minification. For both magnification and minification, the filterType may be one of:

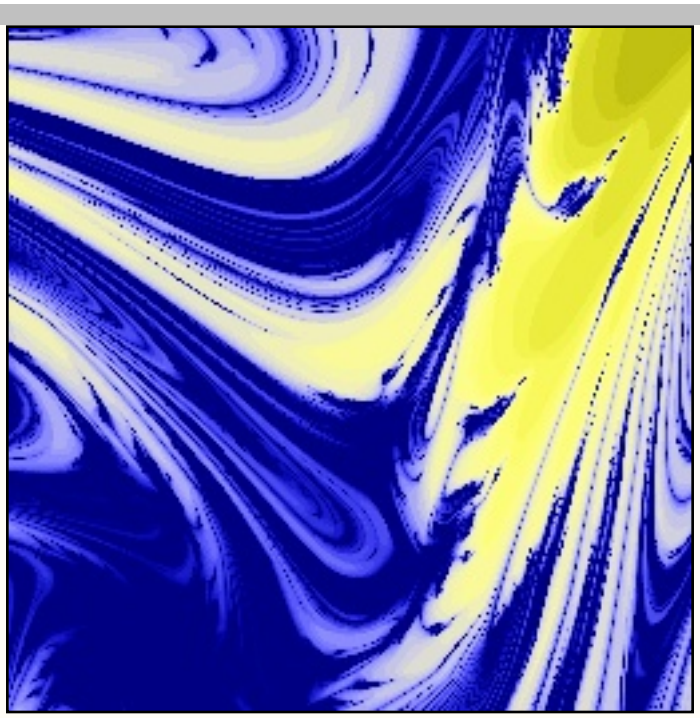
`Texture.FTNearest`	Sample the nearest pixel.
`Texture.FTLinear`	Sample the four nearest pixels, and linearly interpolate them.

For minification only, in addition to the above two choices, you can also choose from:

`Texture.FTNearestMipmapNearest`	Point sample the pixel from the nearest mipmap level.
`Texture.FTLinearMipmapNearest`	Bilinear filter the pixel from the nearest mipmap level.
`Texture.FTNearestMipmapLinear`	Point sample the pixel from two mipmap levels, and linearly blend.
`Texture.FTLinearMipmapLinear`	Bilinearly filter the pixel from two mipmap levels, and linearly blend the results. This is also called **trilinear filtering**.

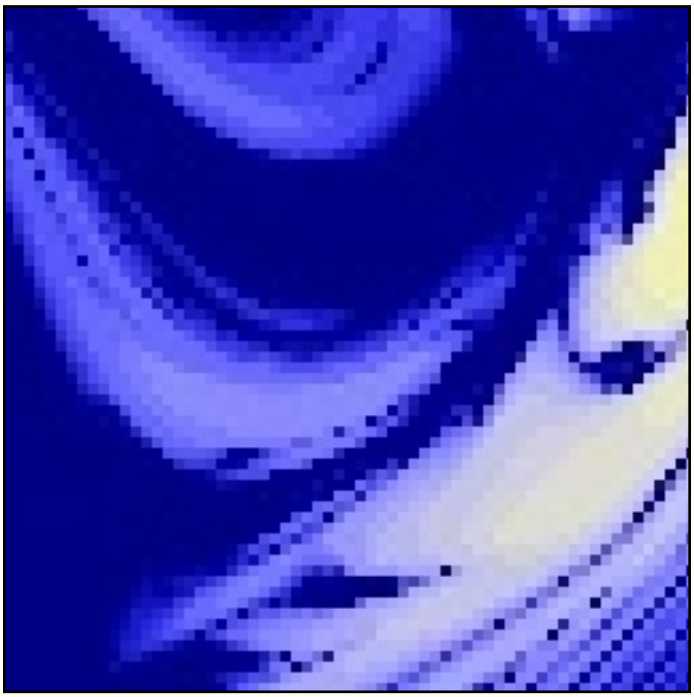
The default filter type for both magnification and minification is `FTLinear`.

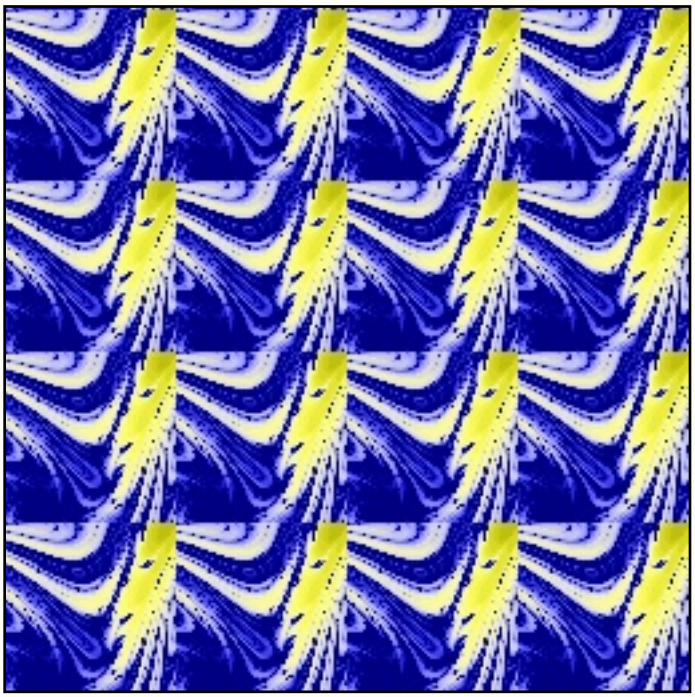
Consider the visual effects of the various filter types on magnification and minification of the following texture:



FTNearest

```
texture.setMagfilter(Texture.FTNearest)
texture.setMinfilter(Texture.FTNearest)
```

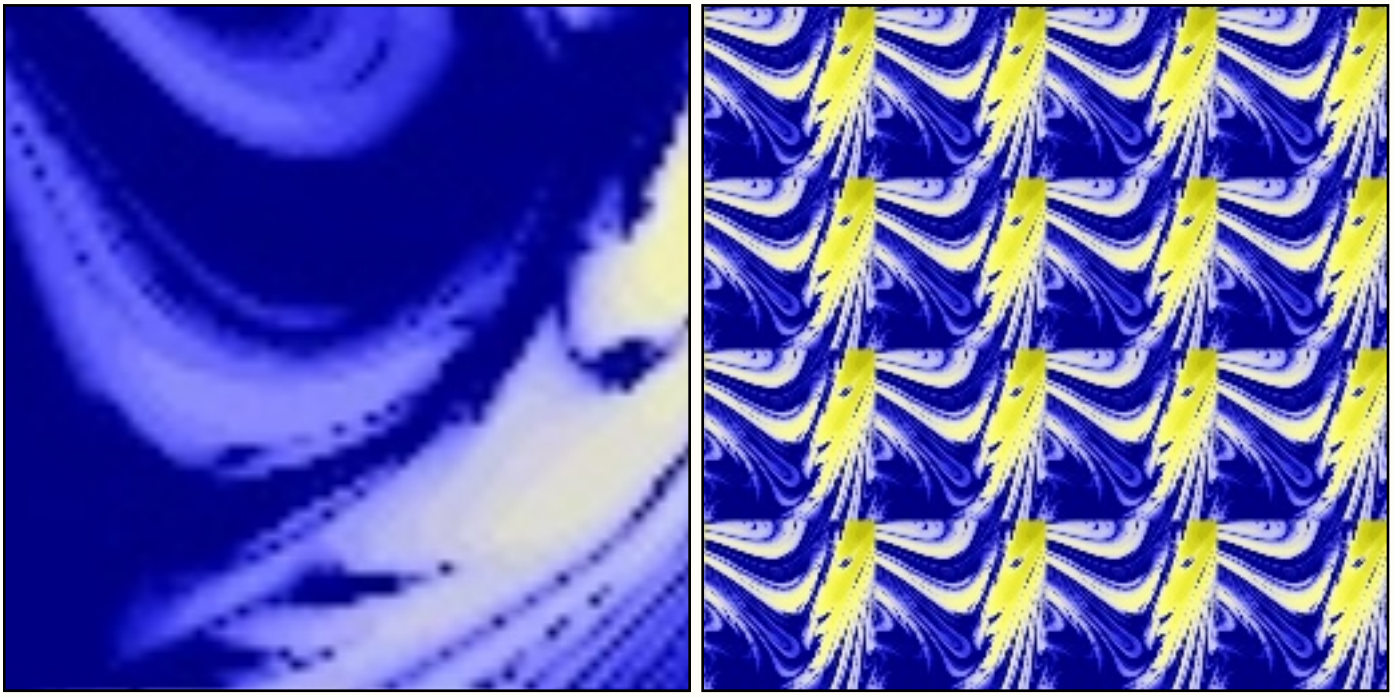




Usually, `FTNearest` is used only to achieve a special pixelly effect.

FTLinear

```
texture.setMagfilter(Texture.FTLinear)
texture.setMinfilter(Texture.FTLinear)
```



FTLinear is a good general-purpose choice, though it isn't perfect.

Mipmaps

Many graphics tutorials will go on for pages and pages about exactly what mipmapping means and how it all works inside. We'll spare you those details here; but you should understand the following things about mipmapping:

(1) It requires 33% more texture memory (per mipmapped texture), but it renders quickly.

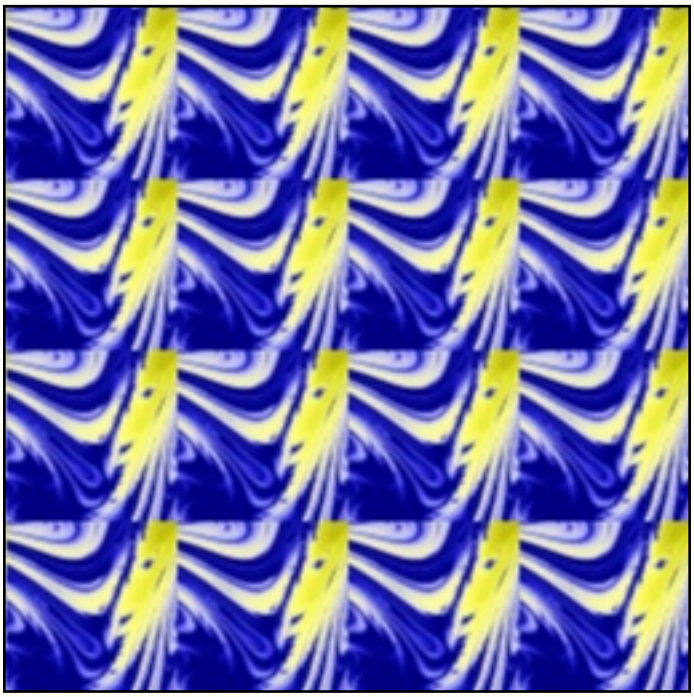
(2) It helps the texture look much smoother than filtering alone when it is minified.

(3) Mipmapping doesn't have anything at all to do with magnification.

(4) It has a tendency to blur minified textures out a little too much, especially when the texture is applied to a polygon that is very nearly edge-on to the camera.

There are four different filter types that involve mipmapping, but you almost always want to use just the last one, FTLinearMipmapLinear. The other modes are for advanced uses, and sometimes can be used to tweak the mipmap artifacts a bit (especially to reduce point 4, above). If you don't understand the description in the table above, it's not worth worrying about.

```
texture.setMinfilter(Texture.FTLinearMipmapLinear)
```



Anisotropic Filtering

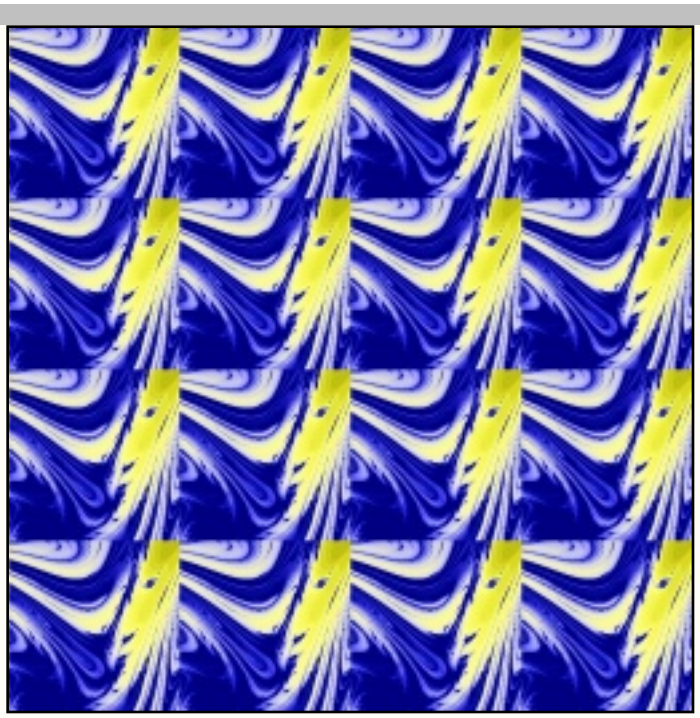
There is one final addition to the texture filtering equation: you can enable anisotropic filtering on top of any of the above filter modes, which enables a more expensive, slightly slower rendering mode that generally produces superior effects. In particular, anisotropic filtering is usually better at handling texture minification than mipmapping, and doesn't tend to blur out the texture so much.

To enable anisotropic filtering, you specify the degree:

```
texture.setAnisotropicDegree(degree)
```

The degree should be an integer number. The default value is 1, which indicates no anisotropic filtering; set it to a higher number to indicate the amount of filtering you require. Larger numbers are more expensive but produce a better result, up to the capability of your graphics card. Many graphics cards don't support any degree other than 2, which is usually sufficient anyway.

```
texture.setAnisotropicDegree(2)
```



At the present, anisotropic filtering is only supported by the DirectX interfaces. Some older graphics cards cannot perform anisotropic filtering at all.

<<prev top next>> Search

Panda3D Manual: Simple Texture Replacement

Although usually you will load and display models that are already textured, you can also apply or replace a texture image on a model at runtime. To do this, you must first get a handle to the texture, for instance by loading it directly:

```
myTexture = loader.loadTexture("myTexture.png")
```

The above loadTexture() call will search along the current model-path for the named image file (in this example, a file named "myTexture.png"). If the texture is not found or cannot be read for some reason, None is returned.

Once you have a texture, you can apply it to a model with the `setTexture()` call. For instance, suppose you used the CardMaker class to generate a plain white card:

```
cm = CardMaker('card')
card = render.attachNewNode(cm.generate())
```

Then you can load up a texture and apply it to the card like this:

```
tex = loader.loadTexture('maps/noise.rgb')
card.setTexture(tex)
```

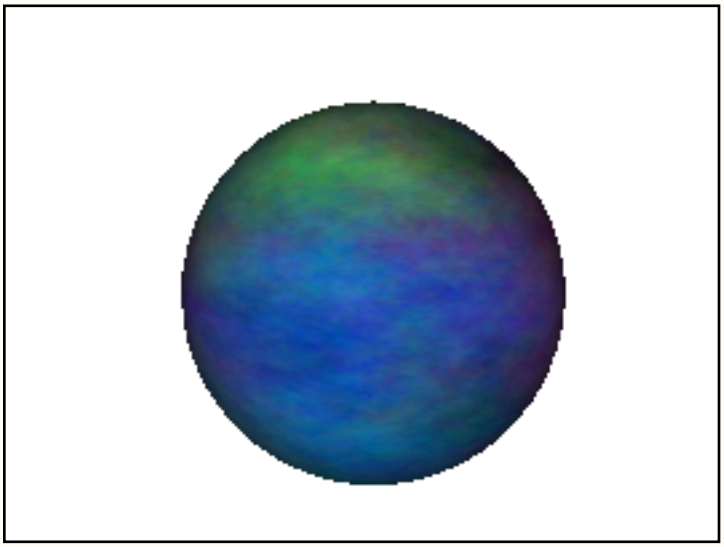
(Note that it is not necessary to use the override parameter to the setTexture() call--that is, you do not need to do card.setTexture(tex, 1)--because in this case, the card does not already have any other texture applied to it, so your texture will be visible even without the override.)

In order for this to work, the model you apply it to must already have **texture coordinates** defined (see Simple Texturing). As it happens, the CardMaker generates texture coordinates by default when it generates a card, so no problem there.

You can also use `setTexture()` to replace the texture on an already-textured model. In this case, you must specify a second parameter to setTexture, which is the same optional Panda override parameter you can specify on any kind of Panda state change. Normally, you simply pass 1 as the second parameter to setTexture(). Without this override, the texture that is assigned directly at the Geom level will have precedence over the state change you make at the model node, and the texture change won't be made.

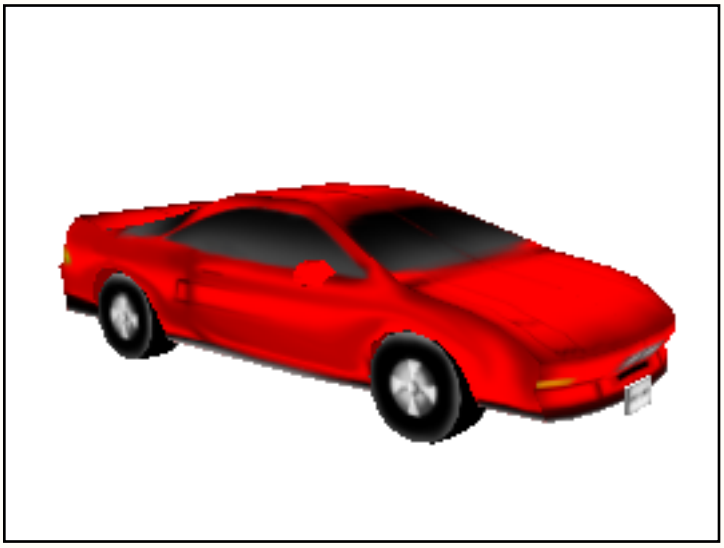
For instance, to change the appearance of smiley:

```
smiley = loader.loadModel('smiley.egg')
smiley.reparentTo(render)
tex = loader.loadTexture('maps/noise.rgb')
smiley.setTexture(tex, 1)
```



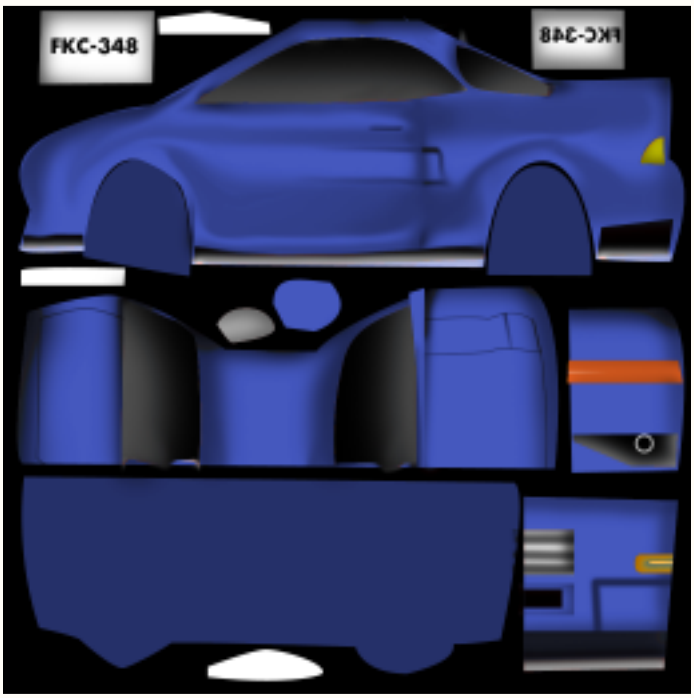
Often, you want to replace the texture on just one piece of a model, rather than setting the texture on every element. To do this, you simply get a NodePath handle to the piece or pieces of the model that you want to change, as described in the section Manipulating a Piece of a Model, and make the `setTexture()` call on those NodePaths.

For instance, this car model has multiple textures available in different colors:



For the most part, this car was painted with one big texture image, which looks like this:

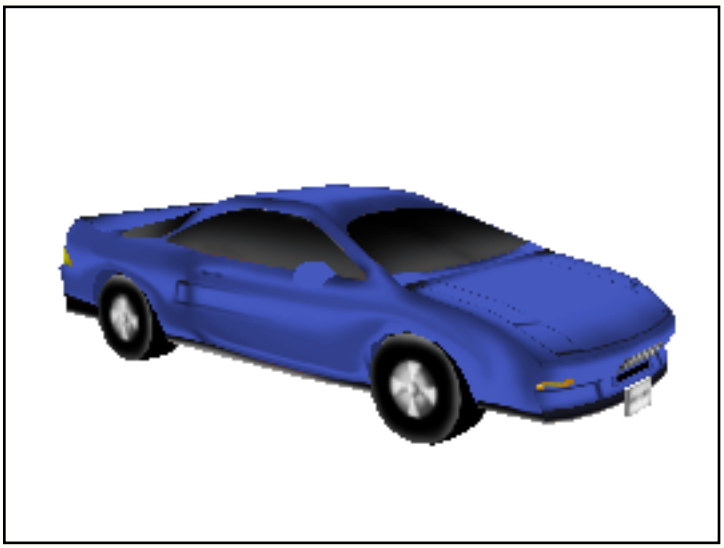
But we also have a blue version of the same texture image:



Although it is tempting to use setTexture() to assign the blue texture to the whole car, that would also assign the blue texture to the car's tires, which need to use a different texture map. So instead, we apply the blue texture just to the pieces that we want to change:

```
car = loader.loadModel('bvw-f2004--carnsx/carnsx.egg')
blue = loader.loadTexture('bvw-f2004--carnsx/carnsx-blue.png')
car.find('**/body/body').setTexture(blue, 1)
car.find('**/body/polySurface1').setTexture(blue, 1)
car.find('**/body/polySurface2').setTexture(blue, 1)
```

And the result is this:



<<prev top next>> Search

Panda3D Manual: Multitexture Introduction

Panda3D provides the ability to apply more than one texture image at a time to the polygons of a model. The textures are applied on top of each other, like coats of paint; very much like the "layers" in a popular photo-paint program.

To layer a second texture on a model, you will have to understand Panda's concept of a **TextureStage**. Think of a TextureStage as a slot to hold a single texture image. You can have as many different TextureStages as you want in your scene, and each TextureStage might be used on one, several, or all models.

When you apply a texture to a model, for instance with the `setTexture()` call, you are actually binding the texture to a particular TextureStage. If you do not specify a TextureStage to use, Panda assumes you mean the "default" TextureStage object, which is a global pointer which you can access as `TextureStage.getDefault()`.

Each TextureStage can hold one texture image for a particular model. If you assign a texture to a particular TextureStage, and then later (or at a lower node) assign a different texture to the same TextureStage, the new texture completely replaces the old one. (Within the overall scene, a given TextureStage can be used to hold any number of different textures for different nodes; but it only holds one texture for any one particular node.)

However, you can have as many different TextureStages as you want. If you create a new TextureStage and use it to assign a second texture to a node, then the node now has both textures assigned to it.

Although there is no limit to the number of TextureStages you assign this way, your graphics card will impose some limit on the number it can render on any one node. Modern graphics cards will typically have a limit of 4 or 8 textures at once; some older cards can only do 2, and some very old cards have a limit of 1 (only one texture at a time). You can find out the multitexture limit on your particular card with the call `base.win.getGsg().getMaxTextureStages()`.

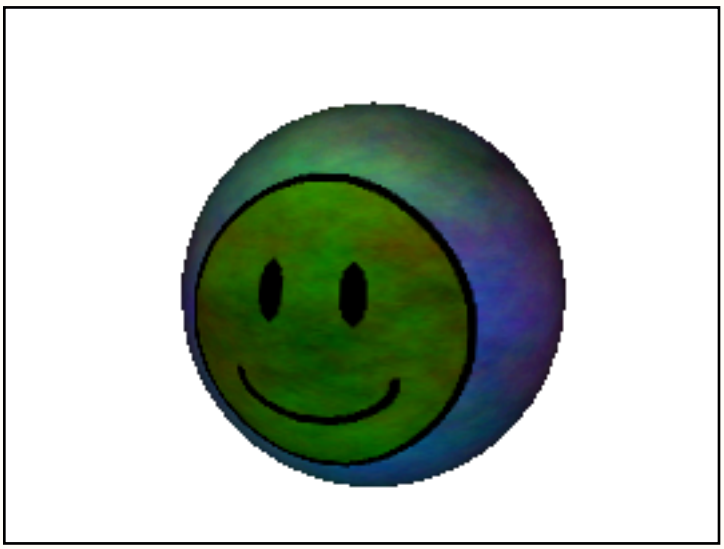
Remember, however, that this limit only restricts the number of different TextureStages you can have on any one particular node; you can still have as many different TextureStages as you like as long as they are all on different nodes.

Let's revisit the example from Simple Texture Replacement, where we replaced the normal texture on smiley.egg with a new texture image that contains a random color pattern. This time, instead of assigning the new texture to the default TextureStage, we'll create a new TextureStage for it, so that both textures will still be in effect:

```
smiley = loader.loadModel('smiley.egg')
smiley.reparentTo(render)
tex = loader.loadTexture('maps/noise.rgb')
ts = TextureStage('ts')
smiley.setTexture(ts, tex)
```

Note that we can create a new TextureStage object on the fly; the only parameter required to the TextureStage parameter is a name, which is significant only to us. When we pass the TextureStage as the first parameter to `setTexture()`, it means to assign the indicated texture to that TextureStage. Also note that we no longer need to specify an override to the setTexture () call, since we are not overriding the texture specified at the Geom level, but rather we are adding to it.

And the result is this:



To undo a previous call to add a texture, use:

```
smiley.clearTexture(ts)
```

passing in the same TextureStage that you used before. Or, alternatively, you may simply use:

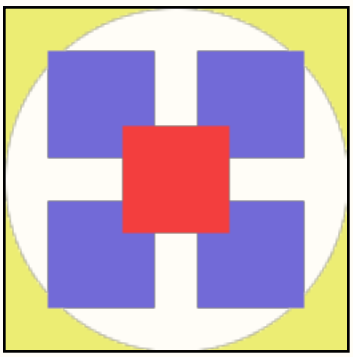
```
smiley.clearTexture()
```

to remove *all* texture specifications that you previously added to the node smiley. This does not remove the original textures that were on the model when you loaded it; those textures are assigned at a different node level, on the Geom objects that make up the model.

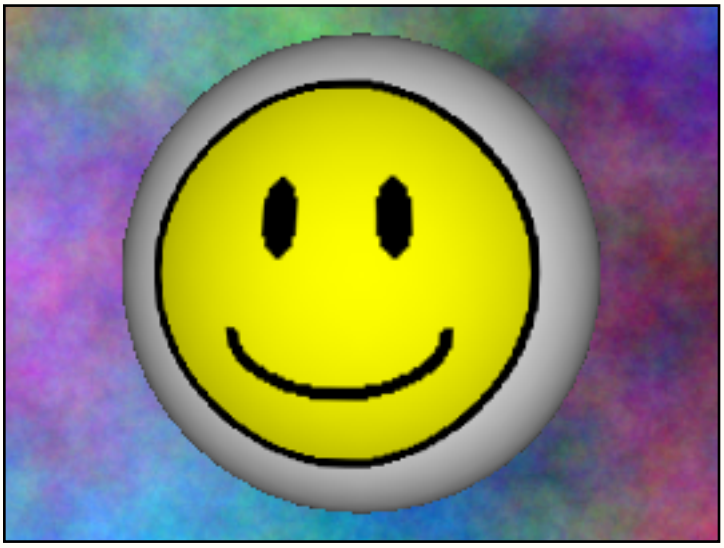
Panda3D Manual: Texture Blend Modes

When you start applying more than one texture at a time, it becomes important to control how the textures combine together. There are several options, and they are controlled through the TextureStage object, specifically through the `TextureStage.setMode()` call.

Let's go back to the example of applying a texture to the smiley model. In this case, we'll create a new TextureStage to apply the following texture image:



To this scene:



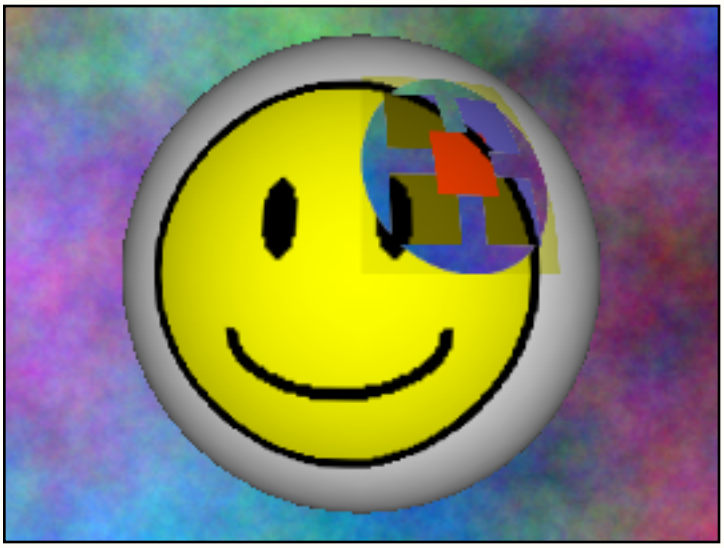
Note that the circular white part of the sample image is actually not white at all, but an alpha cutout (you are seeing through the image to the white page background). We have rendered smiley.egg against a colored background so you can see the effects of alpha in the various modes below; in some of them, the alpha is propagated through to the final color, so smiley is transparent in those parts of the image, but in other modes, the alpha is used for a different purpose, and smiley is not transparent there.

Note also that, for the purposes of illustration, we have only applied the sample texture image to a portion of the smiley model, rather than to the whole model. (This was done by transforming the texture coordinates of this texture stage, which is covered in a later topic.)

Modulate mode

This is the default blend mode. In this mode, the top texture color is multiplied by the bottom texture color to produce the result. This means the resulting texture color will be darker (or at least, no brighter) than both of the original texture colors.

```
ts = TextureStage('ts')
ts.setMode(TextureStage.MModulate)
smiley.setTexture(ts, tex)
```



Note that in this mode, an alpha cutout in the top texture produces an alpha cutout in the resulting image.

Add mode

In this mode, the top texture color is added to the bottom texture color, and clamped to 1 (white). This means the resulting texture color will be brighter (or at least, no darker) than both of the original texture colors.

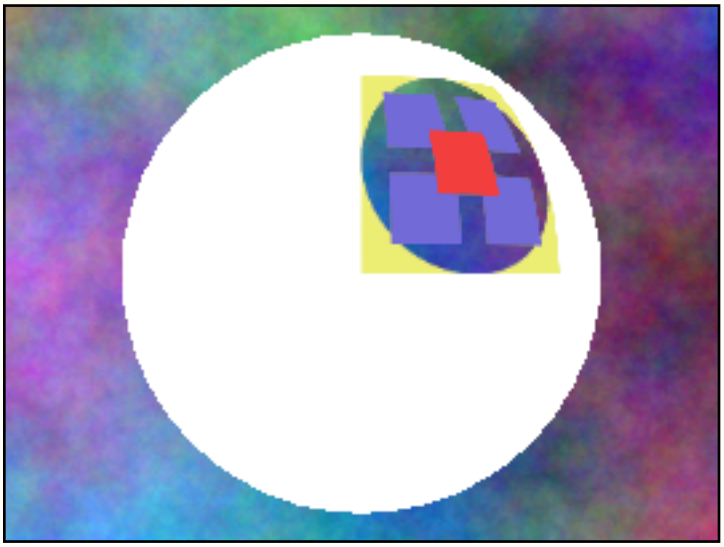
```
ts = TextureStage('ts')
ts.setMode(TextureStage.MAdd)
smiley.setTexture(ts, tex)
```

Note that in this mode, as in modulate mode, an alpha cutout in the top texture produces an alpha cutout in the resulting image. Also note that, unless one or both of your source textures was rather dark, there is a tendency for the colors to get washed out at white where everything clamps to 1.

Replace mode

In this mode the top texture completely replaces the bottom texture. This mode is not often used.

```
ts = TextureStage('ts')
ts.setMode(TextureStage.MReplace)
smiley.setTexture(ts, tex)
```

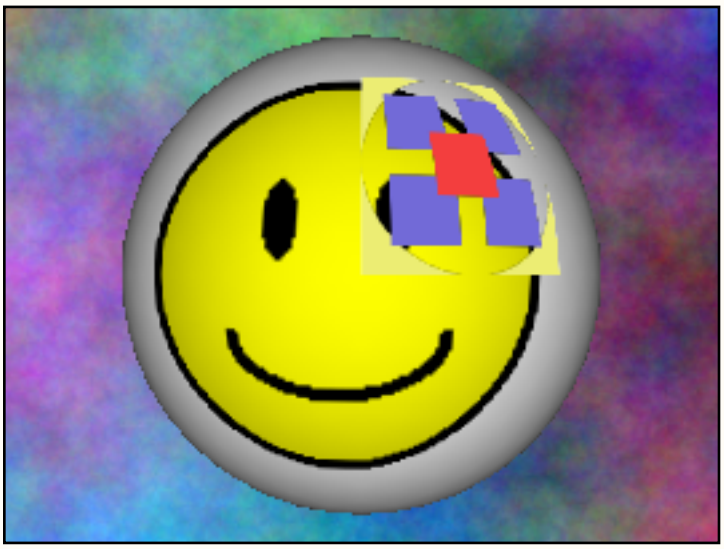


Note that the alpha cutout is preserved, but the effects of lighting (which are considered part of the underlying texture) have been lost.

Decal mode

In this mode the top texture completely replaces the bottom texture, but only where alpha = 1 in the top texture. When alpha = 0, the bottom texture shows through, and there is a smooth blending for alpha values between 0 and 1.

```
ts = TextureStage('ts')
ts.setMode(TextureStage.MDecal)
smiley.setTexture(ts, tex)
```



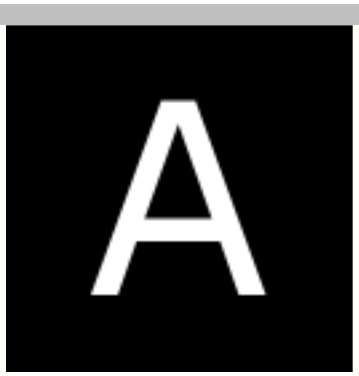
Note that the alpha cutout is no longer preserved in this mode, because alpha is used to determine which texture should be visible. Also note that the effects of lighting are lost for the decalled part of the texture.

Panda3D also provides a built-in decal capability, for rendering a small polygon coplanar with and embedded within a larger polygon, which is not related to the decal texture blend mode.

Blend mode

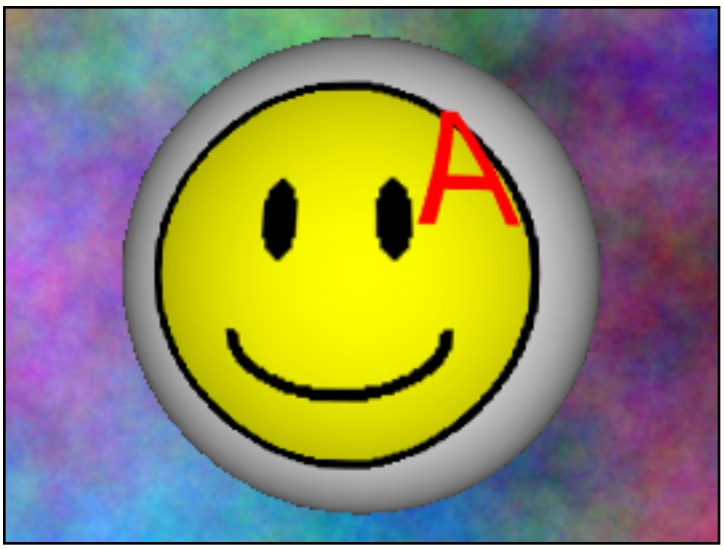
Blend mode is similar to decal mode, except you can specify the color of the decal as a parameter at runtime. You can vary the color and you don't have to have a different texture image prepared for each possible color. However, the decal will always be monochromatic (it will be drawn in different shades of whatever color you specify).

Blend mode can only be used with a grayscale texture, and it does not use alpha. Since the sample texture above is not a grayscale texture, we will use a different texture for this example:



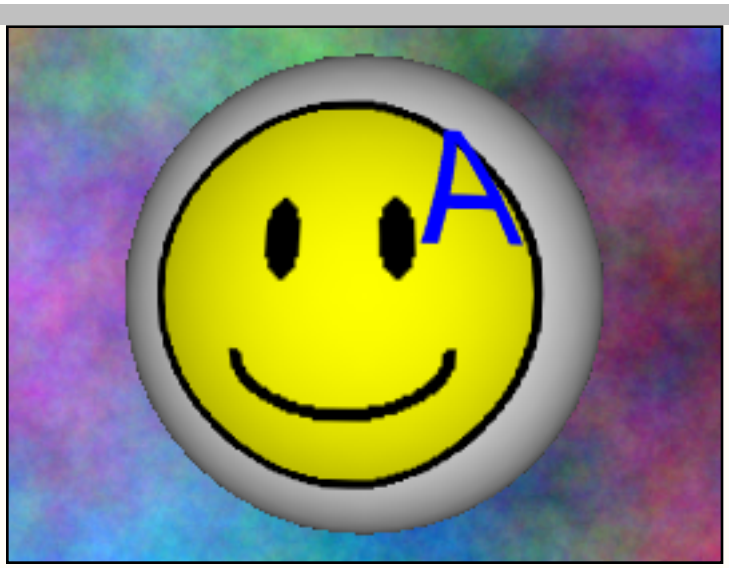
This texture does not have an alpha channel; it is simply a grayscale image with a large white "A" on a field of black. Blend mode will produce the original color where the image is black, and the color we specify with `TextureStage.setColor()` where the image is white. Where the image is shades of gray, there will be a smooth blending between the colors.

```
ts = TextureStage('ts')
ts.setMode(TextureStage.MBlend)
ts.setColor(1, 0, 0, 1)
smiley.setTexture(ts, tex)
```



And we can change the color of the decal at will, simply with:

```
ts.setColor(0, 0, 1, 1)
```



Note that, as with the decal example above, the lighting information is lost where the decal is applied.

<<prev top next>> Search

Panda3D Manual: Texture Order

When there are multiple textures in effect, depending on the Texture Blend Mode in use, it may be important to control the order in which the textures apply. For instance, although Modulate mode and Add mode are order-independent, texture order makes a big difference to Decal mode, Replace mode, and Blend mode.

To specify the texture order, use `TextureStage.setSort()` on one or more of your TextureStages. If you do not specify a sort value, the default sort value is 0. When the geometry is rendered, all of the textures are rendered in increasing order of sort value, such that the largest sort value is rendered on top. Thus, if you want to use Decal mode, for instance, to apply a texture on top of a lower texture, it would be a good idea to use setSort() to give a higher sort value to your decal texture.

Also, since some hardware might not be able to render all of the TextureStages that you have defined on a particular node, Panda provides a way for you to specify which texture(s) are the most important. Use `TextureStage.setPriority()` for this.

The priority value is only consulted when you have applied more TextureStages to a particular node than your current hardware can render. In this case, Panda will select the n textures with the highest priority value (and then sort them in order by the setSort() value). Between two textures with the same priority, Panda will prefer the one with the lower sort value. The default priority is 0.

Panda3D Manual: Texture Combine Modes

In addition to the several Texture Blend Modes described previously, there is a more advanced interface on TextureStage that allows for a larger vocabulary of texture blending options.

Although several of the following options (CMReplace, CMModulate, CMAdd) have obvious parallels with the simpler blend modes described previously, they are in fact more powerful, because with each of the following you may specify the particular source or sources to be used for the operation; you are not limited to simply applying the operation to the top texture and the texture below.

RGB modes

The following specify the effect of the RGB (color) channels. A separate set of methods, below, specifies the effect of the alpha channel.

```
ts.setCombineRgb(TextureStage.CMReplace, source, operand)
```

This mode is similar to "replace mode". Whatever color is specified by source and operand becomes the new color.

```
ts.setCombineRgb(TextureStage.CMModulate, source0, operand0, source1, operand1)
```

This mode is similar to "modulate mode". The color from source0/operand0 is multiplied by the color from source1/operand1.

```
ts.setCombineRgb(TextureStage.CMAdd, source0, operand0, source1, operand1)
```

This mode is similar to "add mode". The color from source0/operand0 is added to the color from source1/operand1, and the result is clamped to 1 (white).

```
ts.setCombineRgb(TextureStage.CMAddSigned, source0, operand0, source1, operand1)
```

In this mode, the colors are added as signed numbers, and the result wraps.

```
ts.setCombineRgb(TextureStage.CMSubtract, source0, operand0, source1, operand1)
```

In this mode, source1/operand1 is subtracted from source0/operand0.

```
ts.setCombineRgb(TextureStage.CMInterpolate, source0, operand0, source1, operand1,
                 source2, operand2)
```

This is the only mode that uses three sources. The value of source2/operand2 is used to select between source0/operand0 and source1/operand1. When source2 is 0, source0 is selected, and when source2 is 1, source1 is selected. When source2 is between 0 and 1, the color is smoothly blended between source0 and source1.

Alpha modes

The following methods more-or-less duplicate the functionality of the above, but they control what happens to the alpha channel. Thus, you have explicit control over whether an alpha cutout in the top texture should produce an alpha cutout in the resulting object.

```
ts.setCombineAlpha(TextureStage.CMReplace, source, operand)
ts.setCombineAlpha(TextureStage.CMModulate, source0, operand0, source1, operand1)
ts.setCombineAlpha(TextureStage.CMAdd, source0, operand0, source1, operand1)
ts.setCombineAlpha(TextureStage.CMAddSigned, source0, operand0, source1, operand1)
ts.setCombineAlpha(TextureStage.CMSubtract, source0, operand0, source1, operand1)
ts.setCombineAlpha(TextureStage.CMInterpolate, source0, operand0, source1, operand1,
                   source2, operand2)
```

Source values

This table lists the legal values for any of source, source0, source1, or source2, in the above calls. This broadly gives you control over which two (or three) textures are used as inputs to the above combine modes.

</tr></tr>
</tr>
</tr>

TextureStage.CSTexture	The current, or "top" texture image.
TextureStage.CSConstant	A constant color, specified via TextureStage. setColor().
TextureStage. CSConstantColorScale	The same as CSConstant, but the color will be modified by NodePath. setColorScale().

TextureStage.CSPrimaryColor	The "primary" color of the object, before the first texture stage was applied, and including any lighting effects.		
TextureStage.CSPrevious	The result of the previous texture stage; i.e. the texture below.	TextureStage.CSLastSavedResult	The result of any of the previous texture stages; specifically, the last stage for which TextureStage.setSavedResult(True) was called.

Operands

This table lists the legal values for any of operand, operand0, operand1, or operand2, in the above calls. This fine-tunes the channel data that is used from each texture input.

TextureStage.COSrcColor	Use the RGB color. When used in a setCombineAlpha() call, RGB is automatically aggregated into grayscale.
TextureStage.COOneMinusSrcColor	The complement of the RGB color.
TextureStage.COSrcAlpha	Use the alpha value. When used in a setCombineRgb() call, alpha is automatically expanded into uniform RGB.
TextureStage.COOneMinusSrcAlpha	The complement of the alpha value.

<<prev top next>> Search

Panda3D Manual: Texture Transforms

It is possible to apply a matrix to transform the (u, v) texture coordinates of a model before rendering. In this way, you can adjust the position, rotation, or scale of a texture, sliding the texture around to suit your particular needs.

Use the following NodePath methods to do this:

```
nodePath.setTexOffset(TextureStage, uOffset, vOffset);
nodePath.setTexScale(TextureStage, uScale, vScale);
nodePath.setTexRotate(TextureStage, degrees);
```

If you don't have a particular TextureStage, use `TextureStage.getDefault()` as the first parameter.

Note that the operation in each case is applied to the (u, v) texture coordinates, not to the texture; so it will have the opposite effect on the texture. For instance, the call `nodePath.setTexScale(ts, 2, 2)` will effectively double the values of the texture coordinates on the model, which doubles the space over which the texture is applied, and thus makes the texture appear half as large.

The above methods apply a 2-d transform to your texture coordinates, which is appropriate, since texture coordinates are usually two-dimensional. However, sometimes you are working with 3-d texture coordinates, and you really do want to apply a 3-d transform. For those cases, there are the following methods:

```
nodePath.setTexPos(TextureStage, uOffset, vOffset, wOffset);
nodePath.setTexScale(TextureStage, uScale, vScale, wScale);
nodePath.setTexHpr(TextureStage, h, p, r);
```

And there is also one generic form:

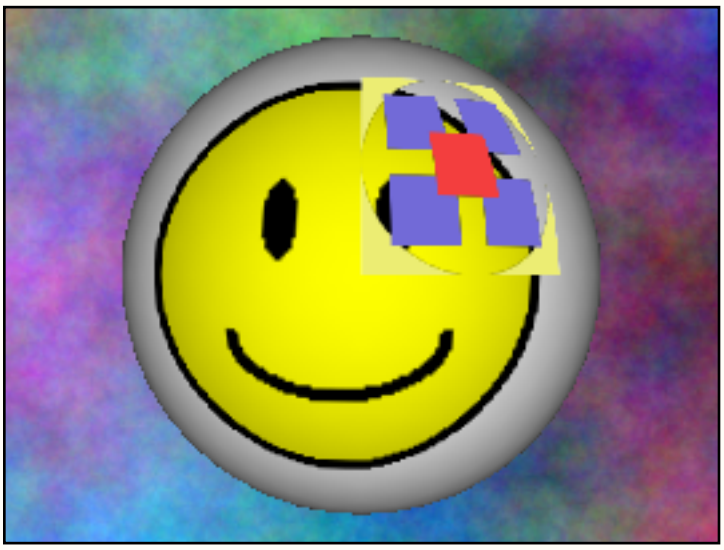
```
nodePath.setTexTransform(TextureStage, transform);
```

This last method sets a generic TransformState object. This is the same kind of 4x4 transform matrix object that you can get from a NodePath via e.g., `NodePath.getTransform()`. You can also construct a new TransformState via a number of methods like `TransformState.makePos(VBase3(0, 1, 0))`. If you intend to apply a 2-d transform only, you should restrict yourself to methods like `TransformState.makePos2d(VBase2(0, 1))`; using only 2-d operations may allow the graphics backend to use a slightly simpler calculation.

Note that the texture transform is associated with a particular TextureStage; it is not a fixed property of the model or its texture coordinates. You can therefore apply a different texture transform to each different TextureStage, so that if you have multiple textures in effect on a particular node, they need not all be in the same place, even if they all use the same texture coordinates. For instance, this technique was used to generate the sample images in the Texture Blend Modes section. In fact, the following code was used to place this sample texture (excerpted):

```
smiley = loader.loadModel('smiley.egg')
ts = TextureStage('ts')
pattern = loader.loadTexture('color_pattern.png')
smiley.setTexture(ts, pattern)
smiley.setTexScale(ts, 8, 4)
smiley.setTexOffset(ts, -4, -2)
```

and the resulting texture:



In the above example, we have applied a scale of (8, 4) to reduce the size of the decal image substantially, and then we specified an offset of (-4, -2) to slide it around in the positive (u, v) direction to smiley's face (since the (0, 0) coordinate happens to be on smiley's backside). However, these operations affect only the decal image; the original smiley texture is unchanged from its normal position, even though both textures are using the same texture coordinates.

Panda3D Manual: Multiple Texture Coordinate Sets

In addition to simple texture transforms, it is also possible to have more than one set of texture coordinates on a model. Panda allows you to define as many different sets of texture coordinates as you like, and each set can be completely unrelated to all of the others.

When you have **multiple texture coordinate sets** (sometimes called **multiple UV sets**) on a model, each set will have its own name, which is any arbitrary string. The default texture coordinate set has no name (its name is the empty string).

Normally, you create multiple texture coordinate sets in the same modeling package that you use to create the model. Not all modeling packages, and not all Panda converters, support multiple texture coordinates. In fact, as of the time of this writing, only the Panda3D 1.1 version (or newer) of the maya2egg converter is known to convert multiple texture coordinates into Panda.

If you happen to have a model with multiple texture coordinate sets, you can specify which set a particular texture should use by calling `TextureStage.setTexcoordName("name")`.

Remember, a TextureStage is used to apply a texture to a model, and so every texture will have an associated TextureStage (though most textures just use the default TextureStage). If you do not call this method for a particular TextureStage, the default behavior is to use the default, unnamed texture coordinate set.

The different TextureStages on a model might share the same texture coordinate sets, or they might each use a different texture coordinate set, or any combination.

Panda3D Manual: Automatic Texture Coordinates

In addition to using texture coordinates that are built into the model, it is also possible to generate texture coordinates at runtime. Usually you would use this technique to achieve some particular effect, such as projective texturing or environment mapping, but sometimes you may simply want to apply a texture to a model that does not already have texture coordinates, and this is the only way to do that.

The texture coordinates generated by this technique are generated on-the-fly, and are not stored within the model. When you turn off the generation mode, the texture coordinates cease to exist.

Use the following NodePath method to enable automatic generation of texture coordinates:

```
nodePath.setTexGen(TextureStage, texGenMode)
```

The texGenMode parameter specifies how the texture coordinates are to be computed, and may be any of the following options. In the list below, "eye" means the coordinate space of the observing camera, and "world" means world coordinates, e.g. the coordinate space of render, the root of the scene graph.

TexGenAttrib.MWorldPosition	Copies the (x, y, z) position of each vertex, in world space, to the (u, v, w) texture coordinates.
TexGenAttrib.MEyePosition	Copies the (x, y, z) position of each vertex, in camera space, to the (u, v, w) texture coordinates.
TexGenAttrib.MWorldNormal	Copies the (x, y, z) lighting normal of each vertex, in world space, to the (u, v, w) texture coordinates.
TexGenAttrib.MEyeNormal	Copies the (x, y, z) lighting normal of each vertex, in camera space, to the (u, v, w) texture coordinates.
TexGenAttrib.MEyeSphereMap	Generates (u, v) texture coordinates based on the lighting normal and the view vector to apply a standard reflection sphere map.
TexGenAttrib.MEyeCubeMap	Generates (u, v, w) texture coordinates based on the lighting normal and the view vector to apply a standard reflection cube map.
TexGenAttrib.MWorldCubeMap	Generates (u, v, w) texture coordinates based on the lighting normal and the view vector to apply a standard reflection cube map.
TexGenAttrib.MPointSprite	Generates (u, v) texture coordinates in the range (0, 0) to (1, 1) for large points so that the full texture covers the square. This is a special mode that should only be applied when you are rendering *sprites,* special point geometry that are rendered as squares. It doesn't make sense to apply this mode to any other kind of geometry. Normally you wouldn't set this mode directly; let the SpriteParticleRenderer do it for you.
TexGenAttrib.MLightVector	Generates special (u, v, w) texture coordinates that represent the vector from each vertex to a particular Light in the scene graph, in each vertex's tangent space. This is used to implement normal maps. This mode requires that each vertex have a *tangent* and a *binormal* computed for it ahead of time; you also must specify the NodePath that represents the direction of the light. Normally, you wouldn't set this mode directly either; use NodePath.setNormalMap(), or implement normal maps using programmable shaders.

Note that several of the above options generate 3-D texture coordinates: (u, v, w) instead of just (u, v). The third coordinate may be important if you have a 3-D texture or a cube map (described later), but if you

just have an ordinary 2-D texture the extra coordinate is ignored. (However, even with a 2-D texture, you might apply a 3-D transform to the texture coordinates, which would bring the third coordinate back into the equation.)

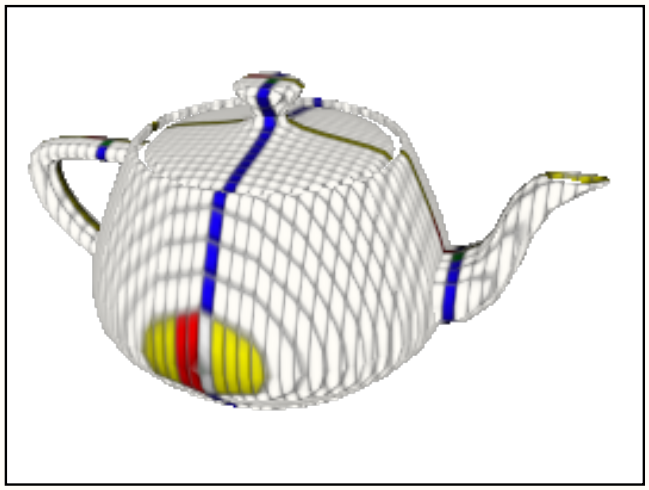
Also, note that almost all of these options have a very narrow purpose; you would generally use most of these only to perform the particular effect that they were designed for. This manual will discuss these special-purpose TexGen modes in later sections, as each effect is discussed; for now, you only need to understand that they exist, and not worry about exactly what they do.

The mode that is most likely to have general utility is the first one: `MWorldPosition`. This mode converts each vertex's (x, y, z) position into world space, and then copies those three numeric values to the (u, v, w) texture coordinates. This means, for instance, that if you apply a normal 2-D texture to the object, the object's (x, y) position will be used to look up colors in the texture.

For instance, the teapot.egg sample model that ships with Panda has no texture coordinates built in the model, so you cannot normally apply a texture to it. But you can enable automatic generation of texture coordinates and then apply a texture:

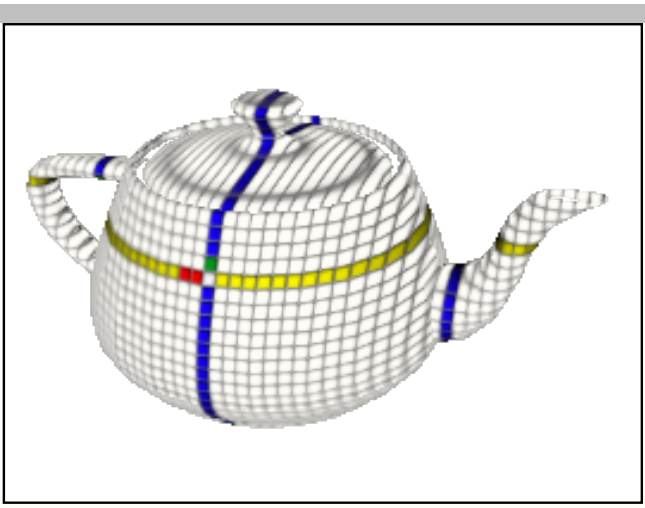
```
teapot = loader.loadModel('teapot.egg')
tex = loader.loadTexture('maps/color-grid.rgb')
teapot.setTexGen(TextureStage.getDefault(), TexGenAttrib.MWorldPosition)
teapot.setTexture(tex)
```

And you end up with something like this:



You can use this in conjunction with a texture transform to further manipulate the texture coordinates. For instance, to rotate the texture 90 degrees, you could do something like this:

```
teapot.setTexTransform(TextureStage.getDefault(), TransformState.makeHpr(VBase3(0, 90, 0)))
```



Finally, consider that the only two choices for the coordinate frame of the texture coordinate generation are "world" and "eye", for the root NodePath and the camera NodePath, respectively. But what if you want to generate the texture coordinates relative to some other node, say the teapot itself? The above images are all well and good for a teapot that happens to be situated at the origin, but suppose we want the teapot to remain the same when we move it somewhere else in the world?

If you use only `MWorldPosition`, then when you change the teapot's position, for instance by parenting it to a moving node, the teapot will seem to move while its texture pattern stays in place--maybe not the effect you had in mind. What you probably intended was for the teapot to take its texture pattern along with it as it moves around. To do this, you will need to compute the texture coordinates in the space of the teapot node, rather than in world space.

Panda3D provides the capability to generate texture coordinates in the coordinate space of any arbitrary node you like. To do this, use `MWorldPosition` in conjunction with Panda's "texture projector", which applies the relative transform between any two arbitrary NodePaths to the texture transform; you can use it to compute the relative transform from world space to teapot space, like this:

```
teapot.setTexGen(TextureStage.getDefault(), TexGenAttrib.MWorldPosition)
teapot.setTexProjector(TextureStage.getDefault(), render, teapot);
```

It may seem a little circuitous to convert the teapot vertices to world space to generate the texture coordinates, and then convert the texture coordinates back to teapot space again--after all, didn't they start out in teapot space? It would have saved a lot of effort just to keep them there! Why doesn't Panda just provide an `MObjectPosition` mode that would convert texture coordinates from the object's native position?

That's a fair question, and `MObjectPosition` would be a fine idea for a model as simple as the teapot, which is after all just one node. But for more sophisticated models, which can contain multiple sub-nodes each with their own coordinate space, the idea of `MObjectPosition` is less useful, unless you truly wanted each sub-node to be re-textured within its own coordinate space. Rather than provide this feature of questionable value, Panda3D prefers to give you the ability to specify the particular coordinate space you had in mind, unambiguously.

Note that you *only* want to call `setTexProjector()` when you are using mode `MWorldPosition`. The other modes are generally computed from vectors (for instance, normals), not positions, and it usually doesn't makes sense to apply a relative transform to a vector.

Panda3D Manual: Projected Textures

In a previous section, we introduced ways to apply an explicit transformation to a model's texture coordinates, with methods like `setTexOffset()` and `setTexScale()`. In addition to this explicit control, Panda3D offers a simple mechanism to apply an automatic texture transform each frame, as computed from the relative transform between any two nodes.

```
nodePath.setTexProjector(textureStage, fromNodePath, toNodePath)
```

When you have enabled this mode, the relative scene-graph transform from `fromNodePath` to `toNodePath`--that is, the result of `fromNodePath.getTransform(toNodePath)`--is automatically applied as a texture-coordinate transform to the indicated textureStage. The result is more-or-less as if you executed the following command every frame:

```
nodePath.setTexTransform(textureStage, fromNodePath.getTransform(toNodePath))
```

There is no need for either `fromNodePath` or `toNodePath` to have any relation to the nodePath that is receiving the `setTexProjector()` call; they can be any two arbitrary NodePaths. If either of them is just `NodePath()`, it stands for the top of the graph.

This has several useful applications. We have already introduced one application, in conjunction with `MWorldPosition`, to move the generated texture coordinates from the root of the graph to the model itself.

Interval-animated texture transforms

Another handy application for a TexProjector is to enable the use of the various LerpIntervals to animate a texture transform. Although there are no LerpIntervals that directly animate texture transforms, you can make a LerpInterval animate a NodePath--and then set up a TexProjector effect to follow that NodePath. For example:

```
smiley = loader.loadModel('smiley.egg')
lerper = NodePath('lerper')
smiley.setTexProjector(TextureStage.getDefault(), NodePath(), lerper)
i = lerper.posInterval(5, VBase3(0, 1, 0))
i.loop()
```

Note that you don't even have to parent the animated NodePath into the scene graph. In the above example, we have set up the interval `i` to repeatedly move the standalone NodePath `lerper` from position (0, 0, 0) to (0, 1, 0) over 5 seconds. Since `smiley` is assigned a

TexProjector that copies the relative transform from `NodePath()` to `lerper`--that is, the net transform of `lerper`--it means we are really animating the texture coordinates on `smiley` from (0, 0) to (0, 1) (the Z coordinate is ignored for an ordinary 2-D texture).

Projected Textures

Another useful application of the TexProjector is to implement **projected textures**--that is, a texture applied to geometry as if it has been projected from a lens somewhere in the world, something like a slide projector. You can use this to implement a flashlight effect, for instance, or simple projected shadows.

This works because the TexProjector effect does one additional trick: if the second NodePath in the `setTexProjector()` call happens to be a LensNode, then the TexProjector automatically applies the lens's projection matrix to the texture coordinates (in addition to applying the relative transform between the nodes).

To implement projected textures, you need to do three steps:

1. Apply the texture you want to the model you want to project it onto, usually on its own TextureStage, so that it is multitextured.

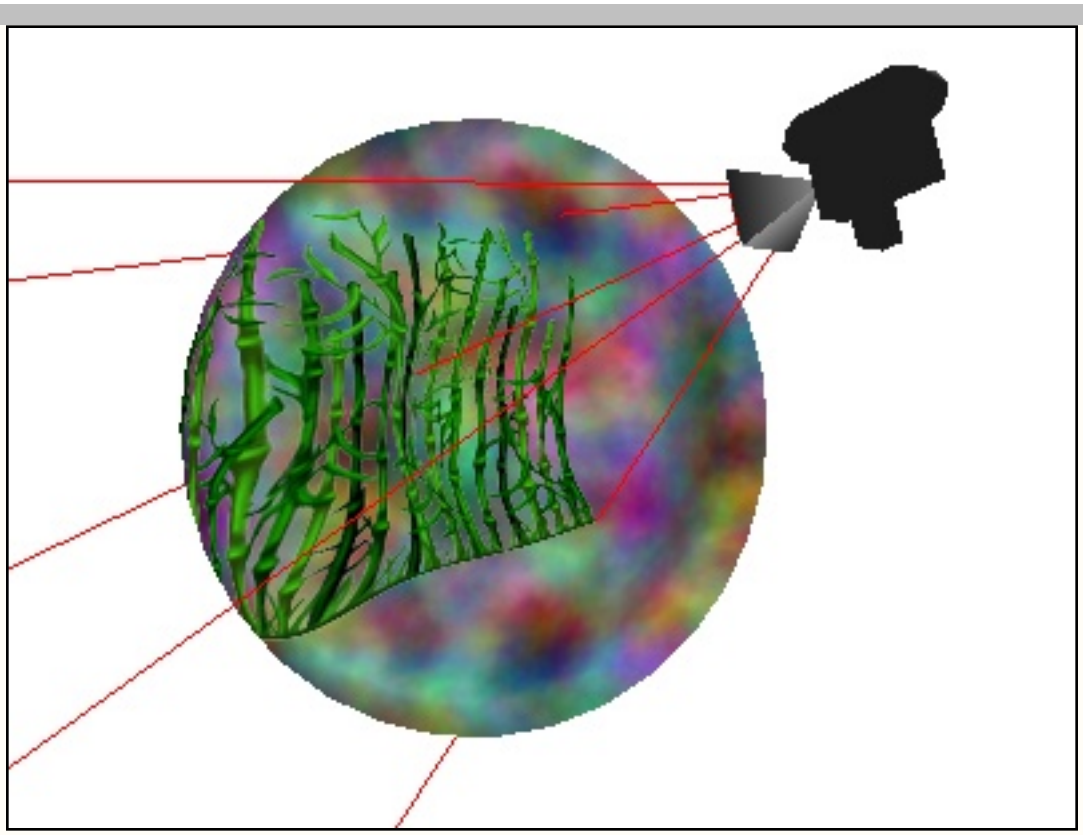
2. Put the `MWorldPosition` TexGen mode on the model. This copies the model's vertex positions into its texture coordinates, for your texture's TextureStage.

3. Call `model.setTexProjector(textureStage, NodePath(), projector)`, where `projector` is the NodePath to the LensNode you want to project from.

For your convenience, the NodePath class defines the following method that performs these three steps at once:

```
nodePath.projectTexture(textureStage, texture, lensNodePath)
```

For instance, we could use it to project the bamboo texture ("envir-reeds.png") onto the ripple.egg model, like this:



You could move around the projector in the world, or even change the lens field of view, and the bamboo image would follow it. (In the above image, the camera model and the projection lines are made visible only for illustration purposes; normally you wouldn't see them.)

This image was generated with the following code (excerpted; click on the image for the complete program):

```
ripple = Actor.Actor('ripple.egg')
ripple.reparentTo(render)

proj = render.attachNewNode(LensNode('proj'))
lens = PerspectiveLens()
proj.node().setLens(lens)
proj.reparentTo(render)
proj.setPos(1.5, -7.3, 2.9)
proj.setHpr(22, -15, 0)

tex = loader.loadTexture('maps/envir-reeds.png')
ts = TextureStage('ts')
ripple.projectTexture(ts, tex, proj)
```

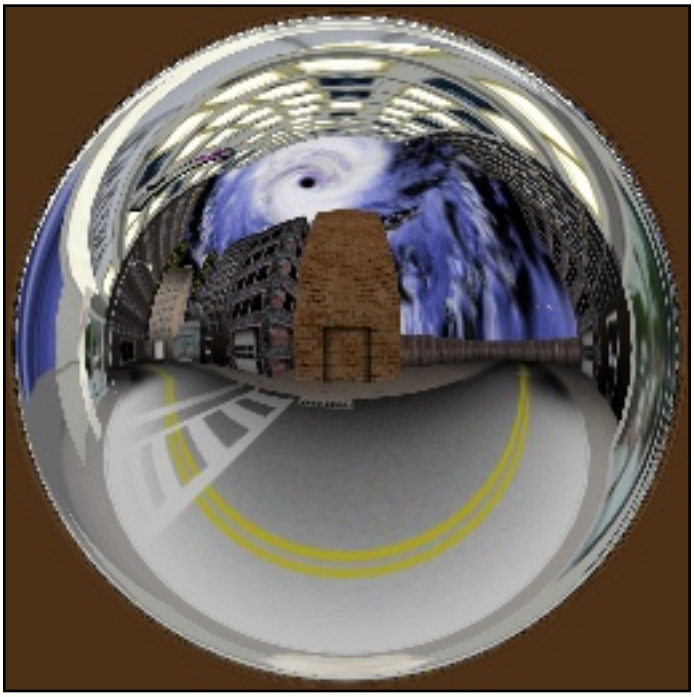
Panda3D Manual: Simple Environment Mapping

There is a classic technique in real-time computer graphics for making objects appear shiny or reflective. It's called **environment mapping** or sometimes **reflection mapping** or, in this case, **sphere mapping**.

Environment mapping is not ray tracing. But it's a cheesy way to get a similar effect. The idea for both of them is that, mathematically, it's easy to calculate the direction from which a ray of light must have been coming before it bounced off a particular point of a shiny object and entered your eye. If the renderer were using ray tracing, it would follow this ray, for each point on your shiny object, backwards from your eye, and determine what object in the environment the ray came from; and that's what you'd see in the reflection.

Ray tracing is still too computation-intensive to be done in real time. But a reflection vector is easy to calculate per-vertex, and if we could turn a reflection vector into a (u, v) texture coordinate pair, the graphics hardware is particularly good at looking up the color in a texture image that corresponds to that (u, v) pair. So all we need is an image that shows the objects in our environment.

In **sphere mapping**, the 3-D reflection vector is turned into a 2-D texture coordinate pair by mathematically applying a spherical distortion. This means the environment map should be a view of the world as seen through a 360-degree fisheye lens, or as reflected in a shiny ball like a holiday ornament. You can see why it is called sphere mapping.



Panda3D can generate sphere maps for you. The above sphere map was generated with the following code:

```
scene = loader.loadModel('bvw-f2004--streetscene/street-scene.egg')
scene.reparentTo(render)
scene.setZ(-2)
base.saveSphereMap('streetscene_env.jpg', size = 256)
```

The idea is simply to put the camera in the middle of your environment, approximately where your shiny object would be. Then just call `base.saveSphereMap()`, and a suitable sphere map image will be generated and written to disk for you. Note that this feature is new as of Panda3D 1.1.

Now you can apply the environment map to just about any object you like. For instance, the teapot:

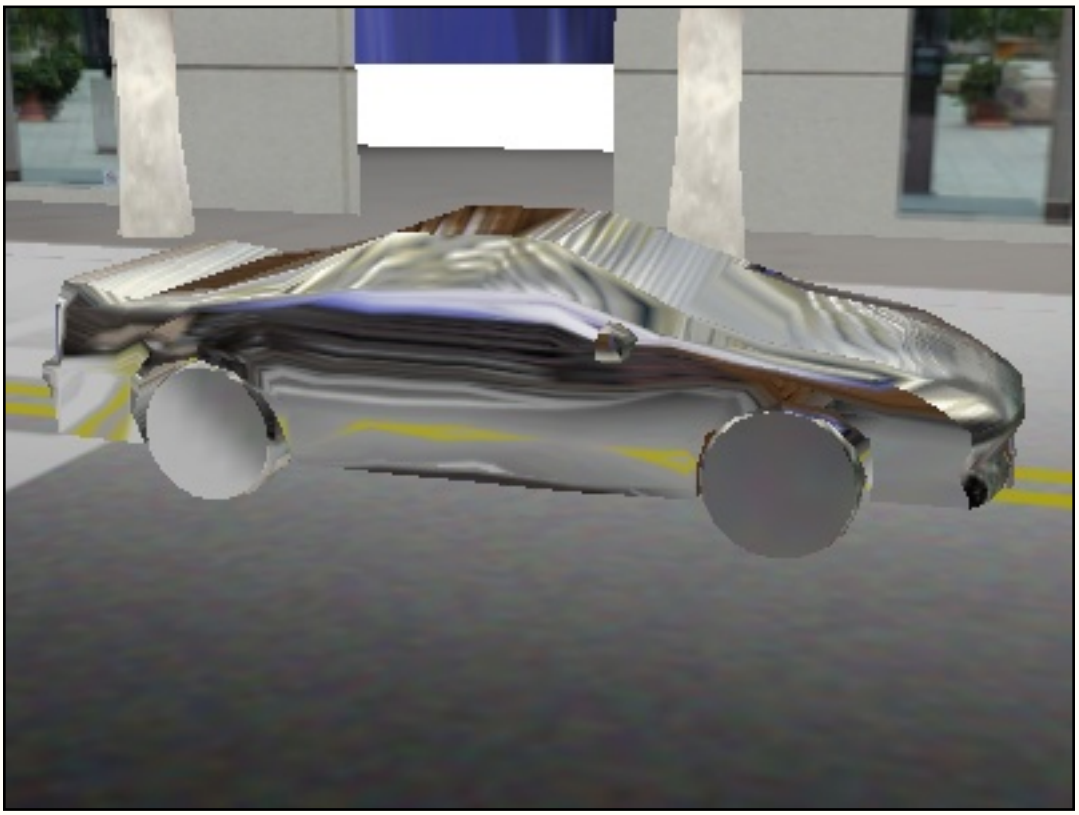
```
tex = loader.loadTexture('streetscene_env.jpg')
teapot.setTexGen(TextureStage.getDefault(), TexGenAttrib.MEyeSphereMap)
teapot.setTexture(tex)
```

In this example, you can see that the key to sphere mapping in Panda is to set the TexGen mode to MEyeSphereMap. This mode computes a spherical (u, v) texture coordinate pair based on the reflection vector for each vertex of the teapot. In order for this to work, your model must have normals defined for all its vertices (the teapot has good normals).

Shiny teapots are one thing, but it would be nice to make something like, say, a car look shiny. We could just do exactly the same thing as above, but our car has a texture map

already. If we just replace the texture map with the environment map we'll end up with a chrome car:

```
car = loader.loadModel('bvw-f2004--carnsx/carnsx.egg')
tex = loader.loadTexture('streetscene_env.jpg')
car.setTexGen(TextureStage.getDefault(), TexGenAttrib.MEyeSphereMap)
car.setTexture(tex, 1)
```

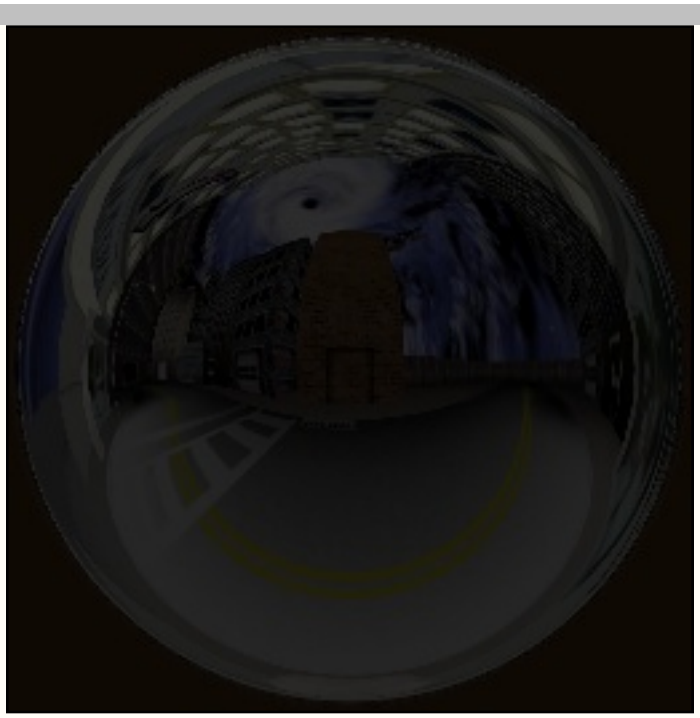


That looks pretty silly. So we'd really prefer to use multitexture to apply both the car's regular texture, and layer a little bit of shine on top of that. We'll use Add mode to add the environment map to the existing color, which is appropriate for a shiny highlight on an object.

In order to use Add mode without oversaturating the colors, we need to darken the environment map substantially. We could use any image processing program to do this; for this example, we'll use Panda3D's `image-trans` utility:

```
image-trans -cscale 0.2 -o streetscene_env_dark.jpg streetscene_env.jpg
```

So the new map looks like this:



While we're fixing things up, let's move the wheels to a different node, so we can assign the shine just to the metal and glass body of the car:

```
car = loader.loadModel('bvw-f2004--carnsx/carnsx.egg')
body = car.find('**/body')
body.findAllMatches('**/FL_wheel*').reparentTo(car)
```

And now the shine is applied like this:

```
tex = loader.loadTexture('streetscene_env_dark.jpg')
ts = TextureStage('env')
ts.setMode(TextureStage.MAdd)
body.setTexGen(ts, TexGenAttrib.MEyeSphereMap)
body.setTexture(ts, tex)
```

Note that the shiny highlights are now quite subtle, but still compelling, especially when you see the car move.

The sphere map technique isn't perfect. The biggest problem with it is that you have to prepare it ahead of time, which means you have to know exactly what will be reflected in your shiny objects--it's impossible for an object to reflect a dynamic object (for instance, an adjacent car).

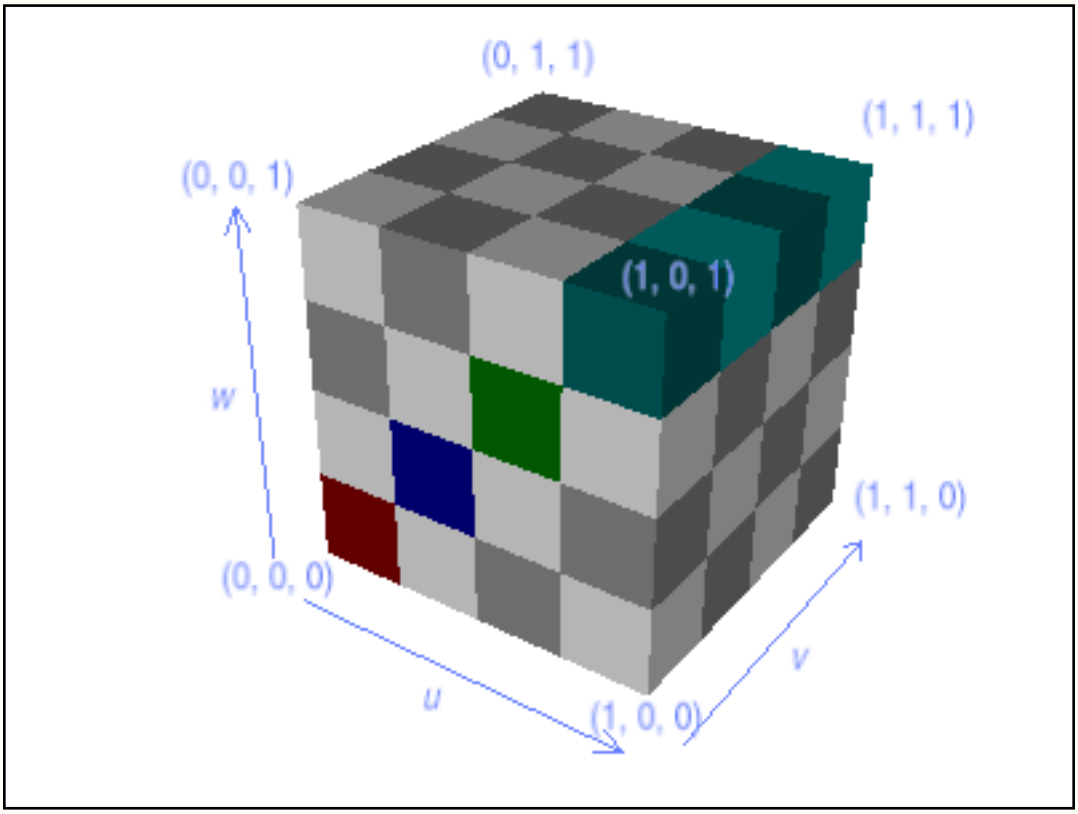
Another problem is that the point-of-view is baked into the sphere map, so that if the camera were to swing around to view the car from the other side, the things you could see in the reflection would still be the objects behind the camera on this side.

Both of these problems can be solved by cube mapping, which is a more advanced technique for, among other things, applying environment maps. However, cube maps aren't always ideal; very often, the venerable sphere map really is the best choice.

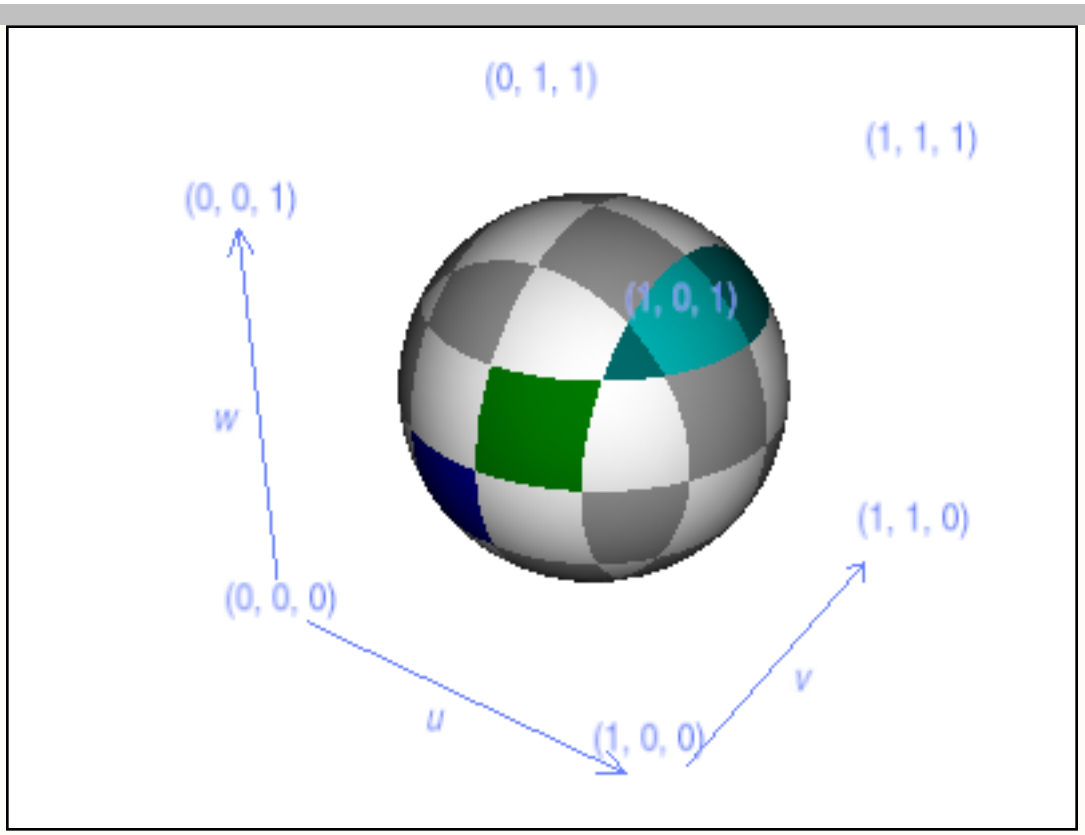
It is rare that an application presents a closeup view of a smooth, round mirrored object in which you can see reflections clearly, like the teapot example above; usually, reflections are just a subtle glinting on the surface, like the car. In these cases the sphere map is ideal, since it is not so important exactly *what* the reflections are, but simply that there *are* reflections. And the sphere map is the easiest and fastest way to render reflections.

Panda3D Manual: 3-D Textures

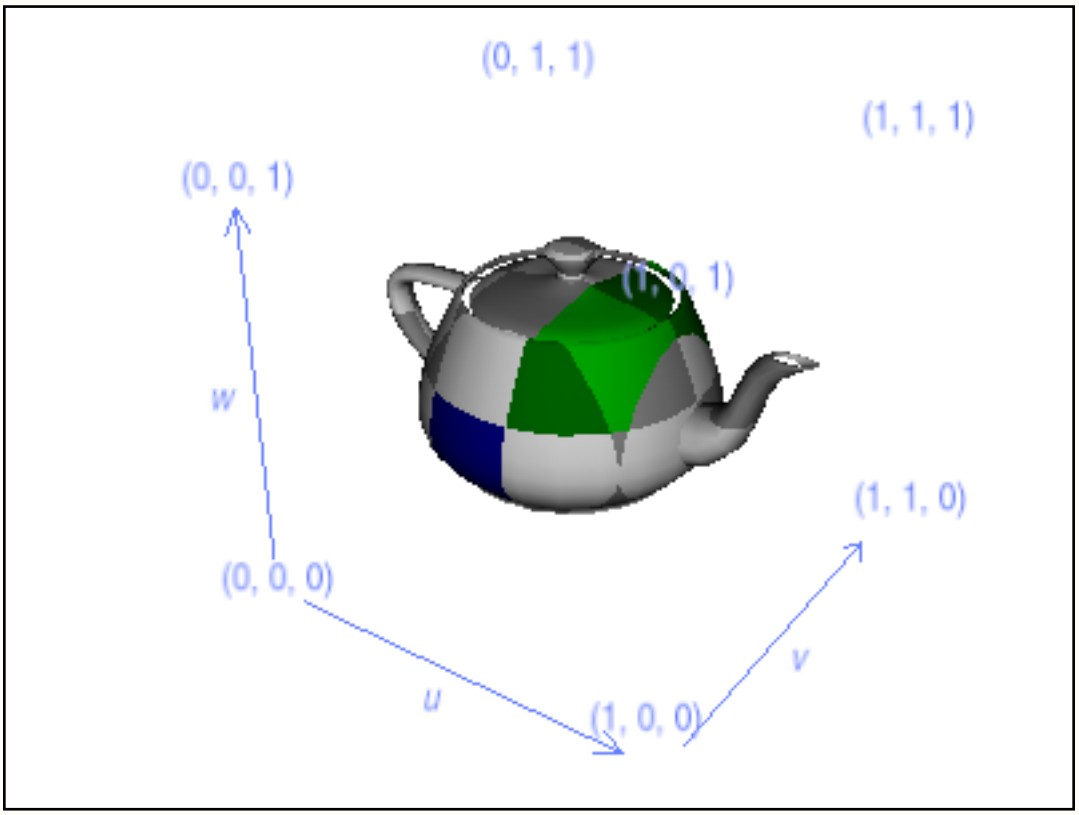
So far, we have only talked about ordinary 2-D texture maps. Beginning with version 1.1, Panda3D also supports the concept of a 3-D texture map. This is a volumetric texture: in addition to a height and a width, it also has a depth:



The 3-D texture image is solid all the way through; if we were to cut away part of the cube we would discover that the checkerboard pattern continues within:



This is true no matter what shape we carve out of the cube:



In addition to the usual *u* and *v* texture dimensions, a 3-D texture also has *w*. In order to apply a 3-D texture to geometry, you will therefore need to have 3-D texture coordinates *(u, v, w)* on your geometry, instead of just the ordinary *(u, v)*.

There are several ways to get 3-D texture coordinates on a model. One way is to assign appropriate 3-D texture coordinates to each vertex when you create the model, the same way you might assign 2-D texture coordinates. This requires that your modeling package (and its Panda converter) support 3-D texture coordinates; however, at the time of this writing, none of the existing Panda converters currently do support 3-D texture coordinates.

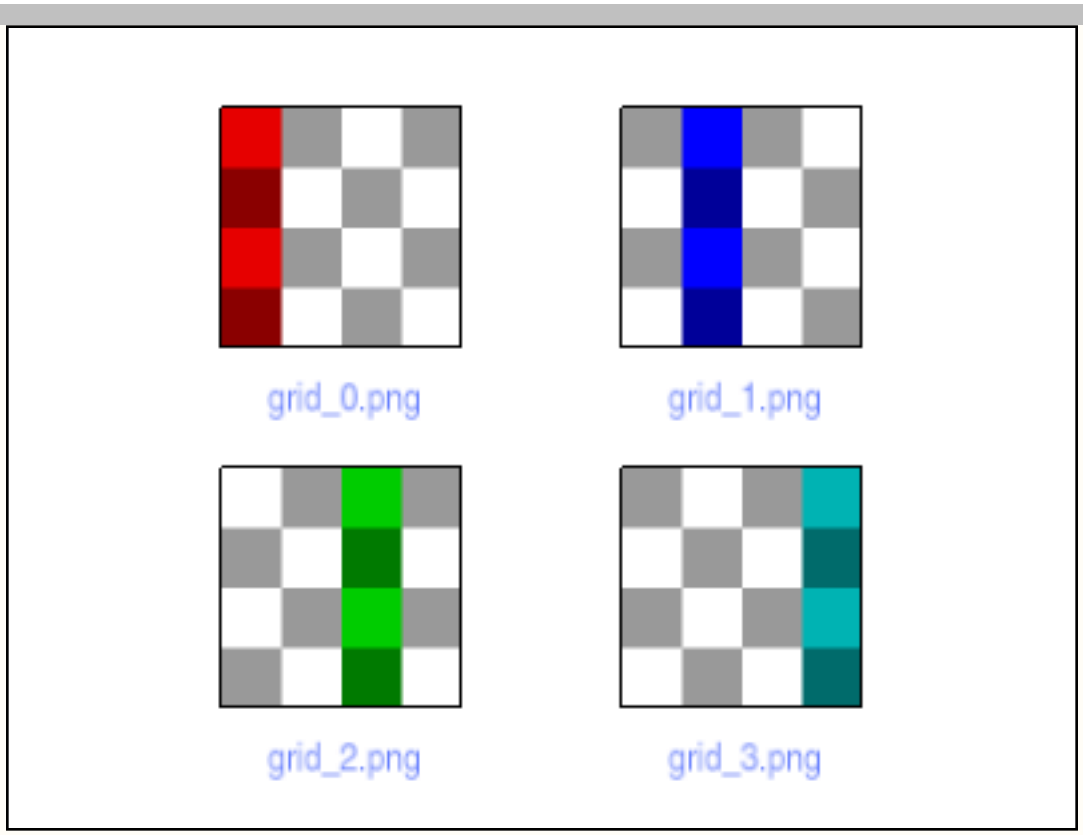
More commonly, 3-D texture coordinates are assigned to a model automatically with one of the TexGen modes, especially MWorldPosition. For example, to assign 3-D texture coordinates to the teapot, you might do something like this:

```
teapot = loader.loadModel('teapot.egg')
teapot.setTexGen(TextureStage.getDefault(), TexGenAttrib.MWorldPosition)
teapot.setTexProjector(TextureStage.getDefault(), render, teapot)
teapot.setTexPos(TextureStage.getDefault(), 0.44, 0.5, 0.2)
teapot.setTexScale(TextureStage.getDefault(), 0.2)
```

The above assigns 3-D texture coordinates to the teapot based on the (x, y, z) positions of its vertices, which is a common way to assign 3-D texture coordinates. The setTexPos() and setTexScale() calls in the above are particular to the teapot model; these numbers are chosen to scale the texture so that its unit cube covers the teapot.

Storing 3-D texture maps on disk is a bit of a problem, since most image formats only support 2-D images. By convention, then, Panda3D will store a 3-D texture image by slicing it into horizontal cross-sections and writing each slice as a separate 2-D image. When you load a 3-D texture, you specify a series of 2-D images which Panda3D will load and stack up like pancakes to make the full 3-D image.

The above 3-D texture image, for instance, is stored as four separate image files:



Note that, although the image is stored as four separate images on disk, internally Panda3D stores it as a single, three-dimensional image, with height, width, and depth.

The Panda3D convention for naming the slices of a 3-D texture is fairly rigid. Each slice must be numbered, and all of the filenames must be the same except for the number; and the first (bottom) slice must be numbered 0. If you have followed this convention, then you can load a 3-D texture with a call like this:

```
tex = loader.load3DTexture("grid_#.png")
```

The hash sign ("#") in the filename passed to `loader.load3DTexture()` will be filled in with the sequence number of each slice, so the above loads files named "grid_0.png", "grid_1.png", "grid_2.png", and so on. If you prefer to pad the slice number with zeros to a certain number of digits, repeat the hash sign; for instance, loading "grid_###.png" would look for files named "grid_000.png", "grid_001.png", and so on. Note that you don't have to use multiple hash marks to count higher than 9. You can count as high as you like even with only one hash mark; it just won't pad the numbers with zeros.

Remember that you must usually choose a power of two for the size of your texture images. This extends to the *w* size, too: for most graphics cards, the number of slices of your texture should be a power of two. Unlike the ordinary *(u, v)* dimensions, Panda3D won't automatically rescale your 3-D texture if it has a non-power-of-two size in the *w* dimension, so it is important that you choose the size correctly yourself.

Applications for 3-D textures

3-D textures are often used in scientific and medical imagery applications, but they are used only rarely in 3-D game programs. One reason for this is the amount of memory they require; since a 3-D texture requires storing *(u × v × w)* texels, a large 3-D texture can easily consume a substantial fraction of your available texture memory.

But probably the bigger reason that 3-D textures are rarely used in games is that the texture images in games are typically hand-painted, and it is difficult for an artist to paint a 3-D texture. It is usually much easier just to paint the surface of an object.

So when 3-D textures are used at all, they are often generated procedurally. One classic example of a procedural 3-D texture is wood grain; it is fairly easy to define a convincing woodgrain texture procedurally. For instance, click here to view a Panda3D program that generates a woodgrain texture and stores it as a series of files named woodgrain_0.png, woodgrain_1.png, and so on. The following code applies this woodgrain texture to the teapot, to make a teapot that looks like it was carved from a single block of wood:

```
teapot = loader.loadModel('teapot.egg')
teapot.setTexGen(TextureStage.getDefault(), TexGenAttrib.MWorldPosition)
teapot.setTexProjector(TextureStage.getDefault(), render, teapot)
teapot.setTexPos(TextureStage.getDefault(), 0.44, 0.5, 0.2)
teapot.setTexScale(TextureStage.getDefault(), 0.2)

tex = loader.load3DTexture('woodgrain_#.png')
teapot.setTexture(tex)
```

However, even procedurally-generated 3-D textures like this are used only occasionally. If the algorithm to generate your texture is not too complex, it may make more sense to program a

pixel shader to generate the texture implicitly, as your models are rendered.

Still, even if it is used only occasionally, the 3-D texture remains a powerful rendering technique to keep in your back pocket.

<<prev top next>> Search

Panda3D Manual: Cube Maps

There is one more special kind of texture map: the **cube map**, which is introduced in Panda3D version 1.1. A cube map is similar to a 3-D texture, in that it requires 3-D texture coordinates *(u, v, w)*; also, a cube map is stored on disk as a sequence of ordinary 2-D images.

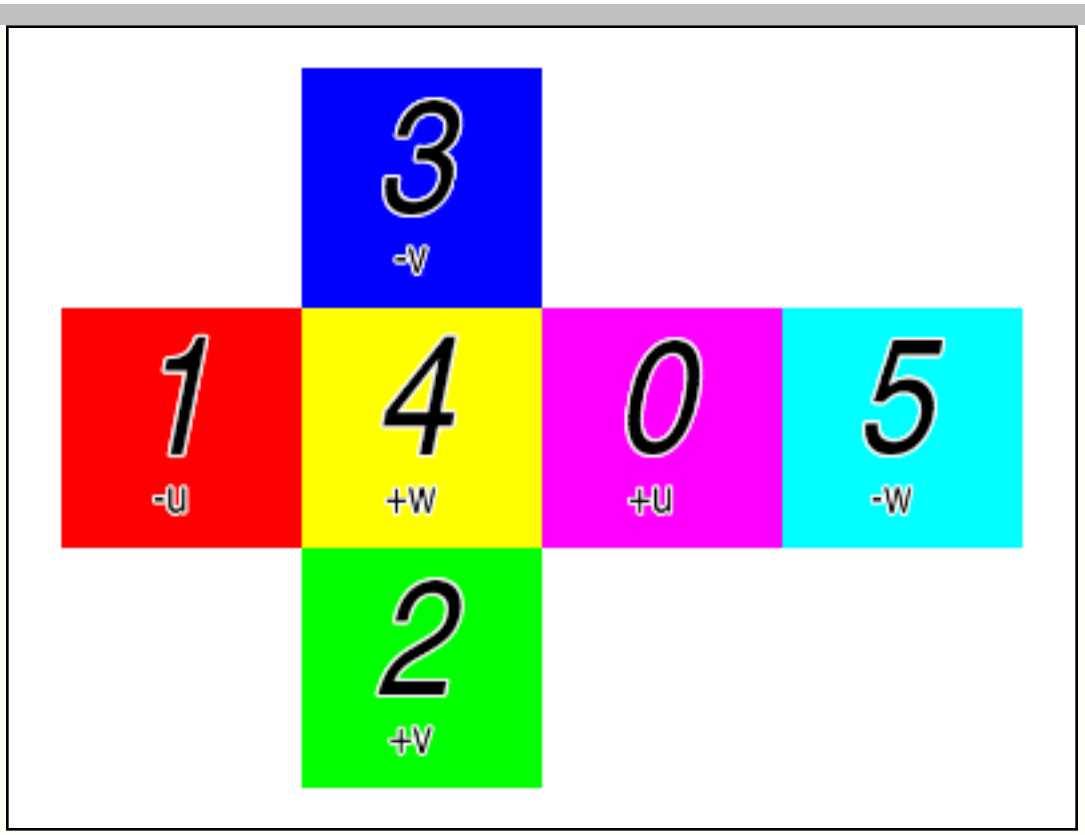
But unlike a 3-D texture, which is defined by stacking up an arbitrary number 2-D images like pancakes to fill up a volume, a cube map is always defined with exactly six 2-D images, which are folded together to make a cube.

The six images of a cube map are numbered from 0 to 5, and each image corresponds to one particular face of the cube:

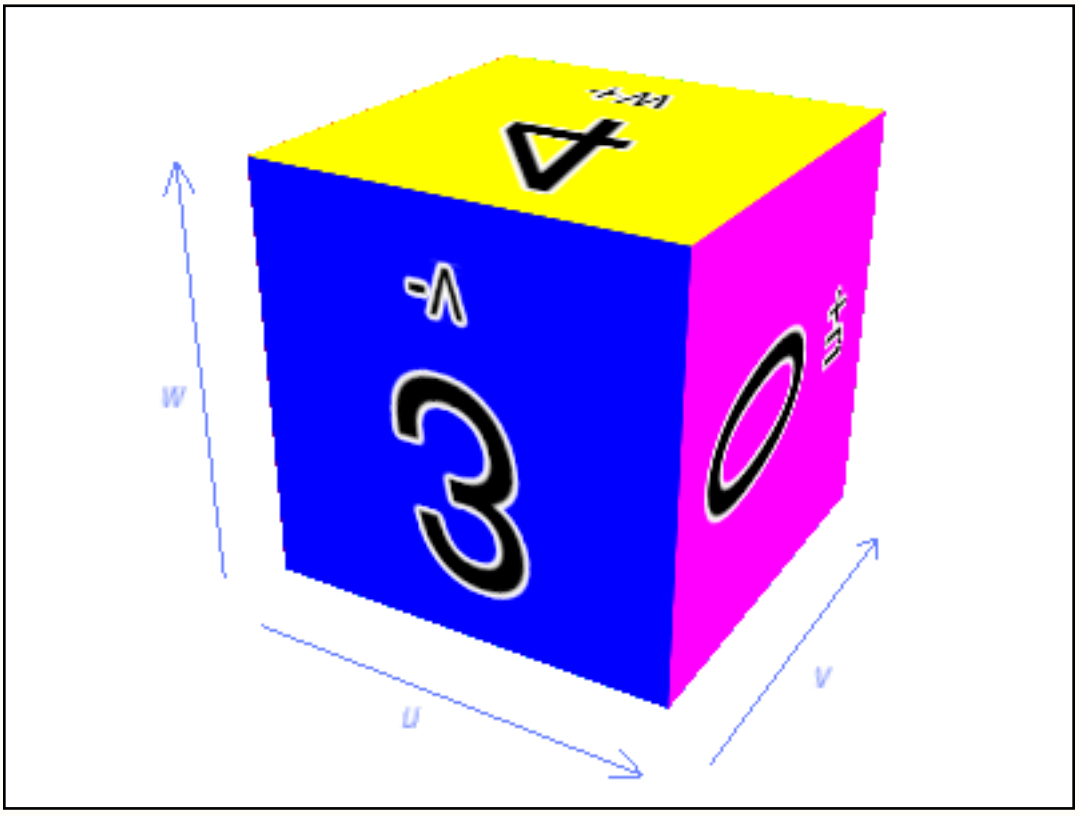
image 0 The +*u* (or +*x*) face (right)
image 1 The -*u* (or -*x*) face (left)
image 2 The +*v* (or +*y*) face (forward)
image 3 The -*v* (or -*y*) face (back)
image 4 The +*w* (or +*z*) face (up)
image 5 The -*w* (or -*z*) face (down)

By +*x* face, we mean the face of the cube farthest along the positive X axis. In Panda3D's default Z-up coordinate system, this is the *right* face. Similarly, the -*x* face is the face farthest along the negative X axis, or the *left* face, and so on for the Y and Z faces. Since the coordinates of a texture map are called *(u, v, w)* instead of *(x, y, z)*, it is technically more correct to call these the +*u* and -*u* faces, though it is often easier to think of them as +*x* and -*x*.

The faces are laid out according to the following diagram:



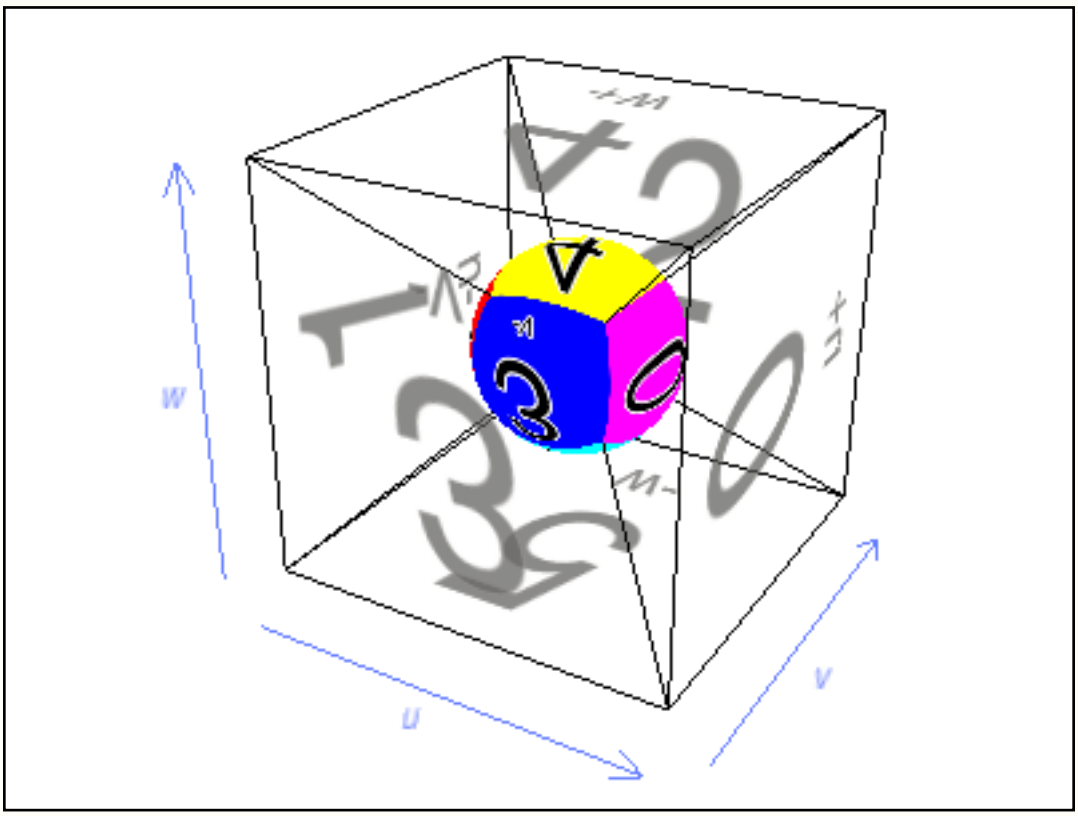
Imagine that you cut out the above diagram and folded it into a cube. You'd end up with something like this:



Note that, when you hold the cube so that the axis indications for each face are in the appropriate direction (as in the picture above), several of the faces are upside-down or sideways. That's because of the way the graphics card manufacturers decided to lay out the

cube map faces (and also because of Panda3D's default coordinate system). But in fact, it doesn't matter which way the faces are oriented, as long as you always generate your cube map images the same way.

In some sense, a cube map is a kind of surface texture, like an ordinary 2-D texture. But in other sense, it is also volumetric like a 3-D texture: every point within the 3-D texture coordinate space is colored according to the face of the cube it comes closest to. A sphere model with the cube map applied to it would pick up the same six faces:



Note that, while a 3-D texture assigns a different pixel in the texture to every *point* within a volume, a cube map assigns a different pixel in the texture to every *direction* from the center.

You can load a cube map from a series of six image files, very similar to the way you load a 3-D texture:

```
tex = loader.loadCubeMap('cubemap_#.png')
```

As with a 3-D texture, the hash mark ("#") in the filename will be filled in with the image sequence number, which in the case of a cube map will be a digit from 0 to 5. The above example, then, will load the six images "cubemap_0.png", "cubemap_1.png", "cubemap_2.png", "cubemap_3.png", "cubemap_4.png", and "cubemap_5.png", and assemble them into one cube map.

Panda3D Manual: Environment Mapping with Cube Maps

Although there are other applications for cube maps, one very common use of cube maps is as an **environment map**, similar to sphere mapping. In fact, it works very much the same as sphere mapping.

Just as with a sphere map, you can have Panda3D generate a cube map for you:

```
scene = loader.loadModel('bvw-f2004--streetscene/street-scene.egg')
scene.reparentTo(render)
scene.setZ(-2)
base.saveCubeMap('streetscene_cube_#.jpg', size = 256)
```

Click here to see the six images generated by the above sample code.

With the cube map saved out as above, you could apply it as an environment map to the teapot like this:

```
tex = loader.loadCubeMap('streetscene_cube_#.jpg')
teapot.setTexGen(TextureStage.getDefault(), TexGenAttrib.MEyeCubeMap)
teapot.setTexture(tex)
```

And the result looks very similar to the sphere map:

In fact, it looks so similar that one might wonder why we bothered. So far, a cube map looks pretty similar to a sphere map, except that it consumes six times the texture memory. Hardly impressive.

But as we mentioned earlier, there are two problems with sphere maps that cube maps can solve. One of these problems is that the point-of-view is permanently baked into the sphere map. Cube maps don't necessarily have the same problem. In fact, we can solve it with one simple variation:

```
tex = loader.loadCubeMap('streetscene_cube_#.jpg')
teapot.setTexGen(TextureStage.getDefault(), TexGenAttrib.MWorldCubeMap)
teapot.setTexture(tex)
```

By changing `MEyeCubeMap` to `MWorldCubeMap`, we have indicated that we would like this cube map to vary its point-of-view as the camera moves. Now the reflected environment will vary according to the direction we are looking at it, so that it shows what is behind the camera at runtime, instead of always showing the area behind the camera when the cube map was generated, as a sphere map must do. In order for this to work properly, you should ensure that your camera is unrotated (that is, `setHpr(0, 0, 0)`) when you generate the cube map initially.

Even with MWorldCubeMap, though, the image is still generated ahead of time, so the reflection doesn't *actually* show what is behind the camera at runtime. It just uses the current camera direction to figure out what part of the reflection image to show.

However, you can make a cube map that truly does reflect dynamic objects in the scene, by

rendering a **dynamic cube map**. This will be discussed in the next section.

<<prev top next>> Search

Panda3D Manual: Dynamic Cube Maps

Since the six faces of a cube map are really just six different views of a scene from the same point, it's possible to generate a cube map automatically by rendering these six different views at runtime.

This is really just a form of offscreen rendering to a texture. Instead of rendering just one 2-D texture image, though, rendering a dynamic cube map means rendering six different 2-D images, one for each face of a cube map texture.

Panda3D makes this easy for you. To start rendering a dynamic cube map, simply call:

```
rig = NodePath('rig')
buffer = base.win.makeCubeMap(name, size, rig)
```

This will return an offscreen `GraphicsBuffer` that will be used to render the cube map. The three required parameters to makeCubeMap() are:

`name`: An arbitrary name to assign to the cube map and its associated GraphicsBuffer. This can be any string.

`size`: The size in pixels of one side of the cube. Many graphics cards require this size to be a power of two. Some cards don't *require* a power of two, but will perform very slowly if you give anything else.

`rig`: The camera rig node. This should be a new NodePath; it will be filled in with six cameras. See below.

There are also additional, optional parameters to makeCubeMap():

`cameraMask`: This specifies the DrawMask that is associated with the cube map's cameras. This is an advanced Panda3D feature that can be used to hide or show certain objects specifically for the cube map cameras.

`toRam`: This is a boolean flag that, when True, indicates the texture image will be made available in system RAM, instead of leaving it only in texture memory. The default is False. Setting it True is slower, but may be necessary if you want to write out the generated cube map image to disk.

Note that we passed a new NodePath, called `rig` in the above example, to the makeCubeMap() call. This NodePath serves as the "camera rig"; the makeCubeMap() method will create six cameras facing in six different directions, and attach them all to the camera rig. Thus, you can parent this rig into your scene and move it around as if it were a six-eyed camera. Normally,

for environment maps, you would parent the rig somewhere within your shiny object, so it can look out of the shiny object and see the things that should be reflected in it.

The actual cube map itself be retrieved with the call:

```
tex = buffer.getTexture()
```

You can apply the texture to geometry as in the previous example. You should use the MWorldCubeMap mode to generate texture coordinates for your geometry, since the camera rig will have a CompassEffect on it to keep it unrotated with respect to render.

When you are done with the cube map, you should remove its buffer (and stop the cube map from continuing to render) by calling:

```
base.graphicsEngine.removeWindow(buffer)
```

As a complete example, here is how we might load up a dynamic cube map environment on our teapot, and move the teapot down the street to show off the dynamic reflections:

```
scene = loader.loadModel('bvw-f2004--streetscene/street-scene.egg')
scene.reparentTo(render)
scene.setZ(-2)

teapot = loader.loadModel('teapot.egg')
teapot.reparentTo(render)

rig = NodePath('rig')
buffer = base.win.makeCubeMap('env', 64, rig)
rig.reparentTo(teapot)

teapot.setTexGen(TextureStage.getDefault(), TexGenAttrib.MWorldCubeMap)
teapot.setTexture(buffer.getTexture())

zoom = teapot.posInterval(5, VBase3(20, 0, 0), startPos = VBase3(-20, 0, 0))
zoom.loop()
```

A word of caution

When you render a dynamic cube map, don't forget that you are re-rendering your scene *six times* every frame, in addition to the main frame render. If you are not careful, and you have a complex scene, you could easily end up reducing your frame rate by a factor of seven.

It is a good idea to limit the amount of geometry that you render in the cube map; one simple way to do this is to ensure that the far plane on the cube map cameras is set relatively close

in. Since all of the cube map cameras share the same lens, you can adjust the near and far plane of all of the cameras at once like this:

```
lens = rig.find('**/+Camera').node().getLens()
lens.setNearFar(1, 100)
```

It is especially important, when you are using cube maps, that you structure your scene graph hierarchically and divide it up spatially, so that Panda3D's view-frustum culling can do an effective job of eliminating the parts of the scene that are behind each of the six cameras. (Unfortunately, the streetscene model used in the above example is not at all well-structured, so the example performs very poorly on all but the highest-end hardware.)

It's also usually a good idea to keep the cube map size (the `size` parameter to makeCubeMap) no larger than it absolutely has to be to get the look you want.

You can also take advantage of the DrawMask to hide things from the cube cameras that are not likely to be important in the reflections. The documentation for this advanced feature of Panda3D will be found in another section of the manual (which, as of the time of this writing, has yet to be written).

Finally, you can temporarily disable the cube map rendering from time to time, if you know the environment won't be changing for a little while; the cube map will retain its last-rendered image. You can do this with `buffer.setActive(0)`. Use `buffer.setActive(1)` to re-activate it.

Panda3D Manual: Automatic Texture Animation

It's possible to generate a model that automatically rotates through a sequence of textures when it is in the scene graph, without having to run a special task to handle this.

To do this, use the `egg-texture-cards` command-line utility. This program will accept a number of texture filenames on the command line, and output an egg file that rotates through each texture at the specified frame rate:

```
egg-texture-cards -o flip.egg -fps 30 explosion*.
jpg
```

This actually creates a model with a different polygon for each frame of the texture animation. Each polygon is put in a separate node, and all the nodes are made a child of a special node called a `SequenceNode`.

The SequenceNode is a special node that only draws one of its children at a time, and it rotates through the list of children at a particular frame rate. You can parent the model under render and it will automatically start animating through its textures. If you need it to start at a particular frame, use something like this:

```
flip = loader.loadModel('flip.egg')
flip.find('**/+SequenceNode').node().setVisibleChild
(startFrame)
flip.reparentTo(render)
```

By default, all of the polygons created by `egg-texture-cards` will have the same size. This means that all of your textures must be the same size as well. While this is a simple configuration, it may not be ideal for certain effects. For instance, to animate an explosion, which starts small and grows larger, it would be better to use a small texture image on a small polygon when the image is small, and have a larger image on a larger polygon when it grows larger. You can achieve this effect, with the -p parameter; specifying -p scales each frame's polygon in relation to the size of the corresponding texture.

```
egg-texture-cards -o flip.egg -fps 30 -p 240,240 explosion*.
jpg
```

There are several other parameters as well; `use egg-texture-cards -h` for a complete list.

Panda3D Manual: Playing MPG and AVI files

Panda now supports AVI format for textures in Panda.

Usage

```
myMovieTexture=loader.loadTexture("myMovie.avi")
myObject.setTexture(myMovieTexture)
```

there are also a bunch of utility functions (by default the texture loops)

```
myMovieTexture.play()
myMovieTexture.play(<first frame>, <end frame>)
myMovieTexture.loop()
myMovieTexture.loop(<first frame>, <end frame>)
myMovieTexture.stop()
myMovieTexture.pose(<frame to jump to>)
```

Issues

The video texure works by decoding on a frame by frame basis and copying into the texture buffer. As such, it is inadvisable to use more than a few high res video textures at the same time.

Certain encoding formats do not work. So far, DV format has been determined incompatible with Panda.

Panda3D Manual: Transparency and Blending

HOW TO FIX TRANSPARENCY ISSUES

Note: this page is cut-and-pasted from a howto we found. We'll polish it later.

Usually transparency works as expected in Panda automatically, but sometimes it just seems to go awry, where a semitransparent object in the background seems to partially obscure a semitransparent object in front of it. This is especially likely to happen with large flat polygon cutouts, or when a transparent object is contained within another transparent object, or when parts of a transparent object can be seen behind other parts of the same object.

The fundamental problem is that correct transparency, in the absence of special hardware support involving extra framebuffer bits, requires drawing everything in order from farthest away to nearest. This means sorting each polygon--actually, each pixel, for true correctness--into back-to-front order before drawing the scene.

It is, of course, impossible to split up every transparent object into individual pixels or polygons for sorting individually, so Panda sorts objects at the Geom level, according to the center of the bounding volume. This works well 95% of the time.

You run into problems with large flat polygons, though, since these tend to have parts that are far away from the center of their bounding volume. The bounding-volume sorting is especially likely to go awry when you have two or more large flats close behind the other, and you view them from slightly off-axis. (Try drawing a picture, of the two flats as seen from the top, and imagine yourself viewing them from different directions. Also imagine where the center of the bounding volumes is.)

Now, there are a number of solutions to this sort of problem. No one solution is right for every situation.

First, the easiest thing to do is to use M_dual transparency. This is a special transparency mode in which the completely invisible parts of the object aren't drawn into the Z-buffer at all, so that they don't have any chance of obscuring things behind them. This only works well if the flats are typical cutouts, where there is a big solid part (alpha == 1.0) and a big transparent part (alpha == 0.0), and not a lot of semitransparent parts (0.0 < alpha < 1.0). It is also a slightly more expensive rendering mode than the default of M_alpha, so it's not enabled by default in Panda. But egg-palettize will turn it on automatically for a particular model if it detects textures that appear to be cutouts of the appropriate nature, which is another reason to use egg-palettize if you are not using it already.

If you don't use egg-palettize (you really should, you know), you can just hand-edit the egg files to put the line:

```
 <Scalar> alpha { dual }
```

within the <Texture> reference for the textures in question.

A second easy option is to use M_multisample transparency, which doesn't have any ordering issues at all, but it only looks good on very high-end cards that have special multisample bits to support full-screen antialiasing. Also, at the present it only looks good on these high-end cards in OpenGL mode (since our pandadx drivers don't support M_multisample explicitly right now). But if M_multisample is not supported by a particular hardware or panda driver, it automatically falls back to M_binary, which also doesn't have any ordering issues, but it always has jaggy edges along the cutout edge. This only works well on texture images that represent cutouts, like M_dual, above.

If you use egg-palettize, you can engage M_multisample mode by putting the keyword "ms" on the line with the texture(s). Without egg-palettize, hand-edit the egg files to put the line:

```
<Scalar> alpha { ms }
```

within the <Texture> reference for the textures in question.

A third easy option is to chop up one or both competing models into smaller pieces, each of which can be sorted independently by Panda. For instance, you can split one big polygon into a grid of little polygons, and the sorting is more likely to be accurate for each piece (because the center of the bounding volume is closer to the pixels). You can draw a picture to see how this works. In order to do this properly, you can't just make it one big mesh of small polygons, since Panda will make a mesh into a single Geom of tristrips; instead, it needs to be separate meshes, so that each one will become its own Geom. Obviously, this is slightly more expensive too, since you are introducing additional vertices and adding more objects to the sort list; so you don't want to go too crazy with the smallness of your polygons.

A fourth option is simply to disable the depth write on your transparent objects. This is most effective when you are trying to represent something that is barely visible, like glass or a soap bubble. Doing this doesn't improve the likelihood of correct sorting, but it will tend to make the artifacts of an incorrect sorting less obvious. You can achieve this by using the transparency option "blend_no_occlude" in an egg file, or by explicitly disabling the depth write on a loaded model with node_path.set_depth_write(false). You should be careful only to disable depth write on the transparent pieces, and not on the opaque parts.

A final option is to make explicit sorting requests to Panda. This is often the last resort because it is more difficult, and doesn't generalize well, but it does have the advantage of not adding additional performance penalties to your scene. It only works well when the transparent objects can be sorted reliably with respect to everything else behind them. For instance, clouds in the sky can reliably be drawn before almost everything else in the scene, except the sky itself. Similarly, a big flat that is up against an opaque wall can reliably be drawn after all of the opaque objects, but before any other transparent object, regardless of where the camera happens to be placed in the scene. See howto.control_render_order.txt for more information about explicitly controlling the rendering order.

Panda3D Manual: Pixel and Vertex Shaders

Currently, Pixel and Vertex shaders are supported through Cg. The following section tells how to bring Cg shaders into Panda3D.

Writing Panda3D Shaders

Currently, Panda3D only supports the Cg shading language. This section assumes that you have a working knowledge of the Cg shader language. If not, it would be wise to read about Cg before trying to understand how Cg fits into Panda3D.

To write a shader, you must create a shader program that looks much like this:

```
//Cg

void vshader(float3 vtx_position : POSITION,
  float2 vtx_texcoord0 : TEXCOORD0,
  out float4 out_position : POSITION,
  out float2 out_texcoord0 : TEXCOORD0,
  uniform float4x4 mat_modelproj)
{
  out_position=mul(mat_modelproj,
vtx_position);
  out_texcoord0=vtx_texcoord0;
}

void fshader(float2 vtx_texcoord0 : TEXCOORD0,
  sampler2D arg_tex : TEXUNIT0,
  out float4 out_color : COLOR)
{
  out_color=tex2D(arg_tex, vtx_texcoord0);
}
```

The first line of a Cg shader needs to be *//Cg*. Do not put a space between the two slashes and the word "Cg". In the future, we may support other shader languages, in which case, those shader languages will have their own header identifiers.

The shader must contain the two subroutines named `vshader` and `fshader`, the vertex shader and fragment shader. In addition, it may contain additional routines named `vshader1`, `fshader1`, `vshader2`, `fshader2`, and so forth. These latter pairs of subroutines represent fallback codepaths, to be used when the video card doesn't support the first pair. If none of the pairs is supported, the shader is disabled and has no effect (ie, rendering proceeds normally using the standard pipeline).

In the following code sample, a shader is loaded and applied to a model:

```
myShader = Shader.load("myshader.
sha")
myModel.setShader(myShader)
```

In the first line, the shader is loaded. The object returned is of class `Shader`. The call to `setShader` causes myModel to be rendered with that shader. Shaders propagate down the scene graph: the node and everything beneath it will use the shader.

The Shader can Fetch Data from the Panda Runtime

Each shader program contains a parameter list. Panda3D scans the parameter list and interprets each parameter name as a request to extract data from the panda runtime. For example, if the shader contains a parameter declaration `float3 vtx_position : POSITION`, Panda3D will interpret that as a request for the vertex position, and it will satisfy the request. Panda3D will only allow parameter declarations that it recognizes and understands.

Panda3D will generate an error if the parameter qualifiers do not match what Panda3D is expecting. For example, if you declare the parameter `float3 vtx_position`, then Panda3D will be happy. If, on the other hand, you were to declare `uniform shader2d vtx_position`, then Panda3D would generate two separate errors: Panda3D knows that vtx_position is supposed to be a float-vector, not a texture, that it is supposed to be varying, not uniform.

Again, all parameter names must be recognized. There is a list of possible shader inputs that shows all the valid parameter names, and the data that Panda3D will supply.

Supplying data to the Shader Manually

Most of the data that the shader could want can be fetched from the panda runtime system by using the appropriate parameter names. However, it is sometimes necessary to supply some user-provided data to the shader. For this, you need `setShaderInput`. Here is an example:

```
myModel.setShaderInput("tint", Vec4(1.0, 0.5, 0.5,
1.0))
```

The method `setShaderInput` stores data that can be accessed by the shader. It is possible to store data of type Texture, NodePath, and Vec4. The `setShaderInput` method also accepts separate floating point numbers, which it combines into a Vec4.

The data that you store using `setShaderInput` isn't necessarily used by the shader. Instead, the values are stored in the node, but unless the shader explicitly asks for them, they will sit unused. So the `setShaderInput("tint", Vec4(1.0, 0.5, 0.5, 1.0))` above simply stores the vector, it is up to the shader whether or not it is interested in a data item labeled "tint."

To fetch data that was supplied using `setShaderInput`, the shader must use the appropriate

parameter name. See the list of possible shader inputs, many of which refer to the data that was stored using `setShaderInput`.

Shader Inputs propagate down the scene graph, and accumulate as they go. For example, if you store `setShaderInput("x",1)` on a node, and `setShaderInput("y",2)` on its child, then the child will contain both values. If you store `setShaderInput("z",1)` on a node, and `setShaderInput("z",2)` on its child, then the latter will override the former. The method `setShaderInput` accepts a third parameter, priority, which defaults to zero. If you store `setShaderInput("w",1,1000)` on a node, and `setShaderInput("w",2,500)` on the child, then the child will contain ("w"==1), because the priority 1000 overrides the priority 500.

Shader Render Attributes

The functions `nodePath.setShader` and `nodePath.setShaderInput` are used to apply a shader to a node in the scene graph. Internally, these functions manipulate a render attribute of class `ShaderAttrib` on the node.

In rare occasions, it is necessary to manipulate ShaderAttrib objects explicitly. The code below shows how to create a ShaderAttrib and apply it to a camera, as an example.

```
myShaderAttrib = ShaderAttrib.make()
myShaderAttrib = myShaderAttrib.setShader(Shader.load("myshader.sha"))
myShaderAttrib = myShaderAttrib.setShaderInput("tint", Vec4
(1.0,0.5,0.5,1.0))
base.cam.node().setInitialState(render.getState().addAttrib(myShaderAttrib))
```

Be careful: attribs are immutable objects. So when you apply a function like `setShader` or `setShaderInput` to a ShaderAttrib, you aren't modifying the attrib. Instead, these functions work by returning a new attrib (which contains the modified data).

Deferred Shader Compilation

When you create an object of class shader, you are just storing the shader's body. You are not (yet) compiling the shader. The actual act of compilation takes place during the rendering process.

Therefore, if the shader contains a syntax error, or if the shader is not supported by your video card, then you will not see any error messages until you try to render something with the shader.

In the unusual event that your computer contains multiple video cards, the shader may be compiled more than once. It is possible that the compilation could succeed for one video card, and fail for the other.

Panda3D Manual: List of Possible Shader Inputs

Shader parameters must have names that are recognized by panda. Here is a list of the allowable parameter names:

uniform sampler2D tex_0	The model's first texture. This requires that the model be textured in the normal manner. You may also use tex_1, tex_2, and so forth, if the model is multitextured. If the model uses a 3D texture or a cubemap, you may also specify sampler3D or samplerCUBE.
uniform sampler2D tex_0_suffix	Obtain a texture by concatenating a hyphen and the suffix to the filename of the model's first texture. For example, if tex_0 is "woman.jpg", then tex_0_normalmap is "woman-normalmap.jpg", and tex_0_specular is "woman-specular.jpg". You may also use tex_1_suffix, tex_2_suffix, and so forth, if the model is multitextured. If the model uses a 3D texture or a cubemap, you may also specify sampler3D or samplerCUBE.
float3 vtx_position: POSITION	Vertex Position. Vertex shader only. You may also use float4, in which case (w==1).
float3 vtx_normal: NORMAL	Vertex Normal. Vertex shader only.
float2 vtx_texcoord0: TEXCOORD0	Texture coordinate associated with the model's first texture. This requires that the model be textured in the normal manner. You may also use vtx_texcoord1, vtx_texcoord2, and so forth if the model is multitextured. Vertex shader only.
float3 vtx_tangent0	Tangent vector associated with the model's first texture. This can only be used if the model has been textured in the normal manner, and if binormals have been precomputed. You may also use vtx_tangent1, vtx_tangent2, and so forth if the model is multitextured. Vertex shader only.
float3 vtx_binormal0	Binormal vector associated with vtx_texcoord0. This can only be used if the model has been textured in the normal manner, and if binormals have been precomputed. You can also use vtx_binormal1, vtx_binormal2, and so forth if the model has been multitextured. Vertex shader only.

floatX vtx_anything	Panda makes it possible to store arbitrary columns of user-defined data in the vertex table; see GeomVertexData. You can access this data using this syntax. For example, vtx_chicken will look for a column named "chicken" in the vertex array. Vertex shader only.
uniform float4x4 trans_x_to_y	A matrix that transforms from coordinate system X to coordinate system Y. See the section on Shaders and Coordinate Spaces for more information.
uniform float4x4 tpose_x_to_y	Transpose of trans_x_to_y
uniform float4 row0_x_to_y	Row 0 of trans_x_to_y.
uniform float4 row1_x_to_y	Row 1 of trans_x_to_y.
uniform float4 row2_x_to_y	Row 2 of trans_x_to_y.
uniform float4 row3_x_to_y	Row 3 of trans_x_to_y.
uniform float4 col0_x_to_y	Col 0 of trans_x_to_y.
uniform float4 col1_x_to_y	Col 1 of trans_x_to_y.
uniform float4 col2_x_to_y	Col 2 of trans_x_to_y.
uniform float4 col3_x_to_y	Col 3 of trans_x_to_y.
uniform float4x4 mstrans_x	Model-Space Transform of X, aka trans_x_to_model
uniform float4x4 cstrans_x	Camera-Space Transform of X, aka trans_x_to_camera
uniform float4x4 wstrans_x	World-Space Transform of X, aka trans_x_to_world
uniform float4 mspos_x	Model-Space Position of X, aka row3_x_to_model
uniform float4 cspos_x	Camera-Space Position of X, aka row3_x_to_camera
uniform float4 wspos_x	World-Space Position of X, aka row3_x_to_world
uniform float4x4 mat_modelview	Modelview Matrix
uniform float4x4 inv_modelview	Inverse Modelview Matrix
uniform float4x4 tps_modelview	Transposed Modelview Matrix
uniform float4x4 itp_modelview	Inverse Transposed Modelview Matrix
uniform float4x4 mat_projection	Projection Matrix
uniform float4x4 inv_projection	Inverse Projection Matrix
uniform float4x4 tps_projection	Transposed Projection Matrix
uniform float4x4 itp_projection	Inverse Transposed Projection Matrix
uniform float4x4 mat_modelproj	Composed Modelview/Projection Matrix
uniform float4x4 inv_modelproj	Inverse ModelProj Matrix
uniform float4x4 tps_modelproj	Transposed ModelProj Matrix

uniform float4x4 itp_modelproj	Inverse Transposed ModelProj Matrix
uniform float4 k_anything	A constant vector that was stored using setShaderInput. Parameter *k_anything* would match data supplied by the call *setShaderInput ("anything", Vec4(x,y,z,w))*
uniform sampler2d k_anything	A constant texture that was stored using setShaderInput. Parameter *k_anything* would match data supplied by the call *setShaderInput ("anything", myTex)*
uniform float4x4 k_anything	A constant matrix that was stored using setShaderInput. Parameter *k_anything* would match data supplied by the call *setShaderInput ("anything", myNodePath)*. The matrix supplied is the nodepath's *local* transform.
uniform float2 sys_cardcenter	Texture coordinates of center of this window's texture card. To generate texture coords for this window's texture card, use (clipx,clipy) * cardcenter + cardcenter.
floatX l_position: POSITION	Linearly interpolated Position, as supplied by the vertex shader to the fragment shader. Declare "out" in the vertex shader, "in" in the fragment shader.
floatX l_color0: COLOR0	Linearly interpolated Primary color, as supplied by the vertex shader to the fragment shader. Declare "out" in the vertex shader, "in" in the fragment shader.
floatX l_color1: COLOR1	Linearly interpolated Secondary color, as supplied by the vertex shader to the fragment shader. Declare "out" in the vertex shader, "in" in the fragment shader.
floatX l_texcoord0: TEXCOORD0	Linearly interpolated Texture Coordinate 0, as supplied by the vertex shader to the fragment shader. You may also use l_texcoord1, l_texcoord2, and so forth. Declare "out" in the vertex shader, "in" in the fragment shader.
out floatX o_color: COLOR	Output Color, as supplied by the fragment shader to the blending units. Fragment shader only.

Panda3D Manual: Shaders and Coordinate Spaces

The Major Coordinate Spaces

When writing complex shaders, it is often necessary to do a lot of coordinate system conversion. In order to get this right, it is important to be aware of all the different coordinate spaces that panda uses. You must know what "space" the coordinate is in. Here is a list of the major coordinate spaces:

Model Space: If a coordinate is in model space, then it is relative to the center of the model currently being rendered. The vertex arrays are in model space, therefore, if you access the vertex position using vtx_position, you have a coordinate in model space. Model space is z-up right-handed.

World Space: If a coordinate is in world space, then it is relative to the scene's origin. World space is z-up right-handed.

View Space: If a coordinate is in view space, then it is relative to the camera. View space is z-up right-handed.

API View Space: This coordinate space is identical to view space, except that the axes may be flipped to match the natural orientation of the rendering API. In the case of opengl, API view space is y-up right-handed. In the case of directx, API view space is y-up left-handed.

Clip Space: Panda's clip space is a coordinate system in which (X/W, Y/W) maps to a screen pixel, and (Z/W) maps to a depth-buffer value. All values in this space range over [-1,1].

API Clip Space: This coordinate space is identical to clip space, except that the axes may be flipped to match the natural orientation of the rendering API, and the numeric ranges may be rescaled to match the needs of the rendering API. In the case of opengl, the (Z/W) values range from [-1, 1]. In the case of directx, the (Z/W) values range from [0,1].

Supplying Translation Matrices to a Shader

You can use a shader parameter named "trans_x_to_y" to automatically obtain a matrix that converts any coordinate system to any other. The words x and y can be "model," "world," "view," "apiview," "clip," or "apiclip." Using this notation, you can build up almost any transform matrix that you might need. Here is a short list of popular matrices that can be recreated using this syntax. Of course, this isn't even close to exhaustive: there are seven keywords, so there are 7x7 possible matrices, of which 7 are the identity matrix.

Desired Matrix	Syntax
The Modelview Matrix	trans_model_to_apiview
The Projection Matrix	trans_apiview_to_apiclip
the DirectX world matrix	trans_model_to_world

the DirectX view matrix	trans_world_to_apiview
gsg.getCameraTransform()	trans_view_to_world
gsg.getWorldTransform()	trans_world_to_view
gsg.getExternalTransform()	trans_model_to_view
gsg.getInternalTransform()	trans_model_to_apiview
gsg.getCsTransform()	trans_view_to_apiview
gsg.getInvCsTransform()	trans_apiview_to_view

Recommendation: Don't use API View Space or API Clip Space

The coordinate systems "API View Space" and "API Clip Space" are not very useful. The fact that their behavior changes from one rendering API to the next makes them extremely hard to work with. Of course, you have to use the composed modelview/projection matrix to transform your vertices, and in doing so, you are implicitly using these spaces. But aside from that, it is strongly recommended that you not use these spaces for anything else.

Model_of_x, View_of_x, Clip_of_x

When you use the word "model" in a trans directive, you implicitly mean "the model currently being rendered." But you can make any nodepath accessible to the shader subsystem using `setShaderInput`:

```
myhouse = loader.loadModel("myhouse")
render.setShaderInput('myhouse',
myhouse)
```

Then, in the shader, you can convert coordinates to or from the model-space of this particular nodepath:

```
uniform float4x4
trans_world_to_model_of_myhouse
```

or, use the syntactic shorthand:

```
uniform float4x4
trans_world_to_myhouse
```

Likewise, you can create a camera and pass it into the shader subsystem. This is particularly useful when doing shadow mapping:

```
render.setShaderInput('shadowcam', self.
shadowcam)
```

Now you can transform vertices into the clip-space of the given camera using this notation:

```
uniform float4x4
trans_model_to_clip_of_shadowcam
```

If you transform your model's vertices from model space into the clip space of a shadow camera, the resulting (X/W,Y/W) values can be used as texture coordinates to projectively texture the shadow map onto the scene (after rescaling them), and the (Z/W) value can be compared to the value stored in the depth map (again, after rescaling it).

Panda does support the notation "trans_x_to_apiclip_of_y", but again, our recommendation is not to use it.

You can transform a vertex to the view space of an alternate camera, using "view of x." In fact, this is exactly identical to "model of x," but it's probably good form to use "view of x" when x is a camera.

Panda3D Manual: Known Shader Bugs and Limitations

Known Bugs in the Shader Subsystem

Here is a list of known bugs, with workarounds:

Problem: Register Allocation.

Problem: nVidia's Cg compiler tries to assign registers to parameters. Under a variety of circumstances, the Cg compiler will assign the same register to two parameters, or to a parameter and to a constant in the program.

Workaround: We have found that if you manually allocate registers by supplying a semantic string for each parameter, this problem is bypassed.

Problem: Bad Target Languages.

Problem: nVidia's Cg compiler will choose one of several different "target" languages to translate the Cg program into. When the Cg compiler tries to translate the program into the VP40/FP40 language, it often produces incorrect output.

Workaround: We have discovered that translation into ARBVP1/ARBFP1 seems to work reliably. Since that language is supported on essentially every video card, it is usually safe to translate into that language. We have provided a directive where you can recommend a target language to the Cg compiler:

//Cg profile arbvp1 arbfp1

If you supply this directive, it will use the recommended language. If you supply more than one directive, it will try the languages in the order provided.

Problem: Untested/Unfinished DirectX Support.

Problem: Shader development is currently being done in OpenGL. The DirectX support typically lags behind, and is often less fully-tested.

Workaround: The default setting for Panda is to use OpenGL, not DirectX. For now, when using shaders, do not change this setting.

Panda3D Manual: Finite State Machines

A "Finite State Machine" is a concept from computer science. Strictly speaking, it means any system that involves a finite number of different states, and a mechanism to transition from one state to another.

In Panda3D, a Finite State Machine, or FSM, is implemented as a Python class. To define a new FSM, you should define a Python class that inherits from the FSM class. You define the available states by writing appropriate method names within the class, which define the actions the FSM takes when it enters or leaves certain states. Then you can request your FSM to transition from state to state as you need it to.

You may come across some early Panda3D code that creates an instance of the ClassicFSM class. ClassicFSM is an earlier implementation of the FSM class, and is now considered deprecated. It is no longer documented here. We recommend that new code use the FSM class instead, which is documented on the following pages.

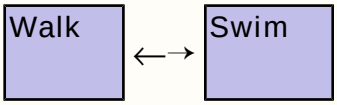
Panda3D Manual: FSM Introduction

In Panda3D, FSM's are frequently used in game code to automatically handle the cleanup logic in game state changes. For instance, suppose you are writing a game in which the avatar spends most of his time walking around, but should go into swim mode when he enters the water. While he is walking around, you want certain animations and sound effects to be playing, and certain game features to be active; but while he is swimming, there should be a different set of animations, sound effects, and game features (this is just an example, of course):

Walk state	**Swim state**
• Should be playing "walk" animation • Should hear footsteps sound effect • Collision detection with doors should be active	• Should be playing "swim" animation • Should hear underwater sound effect • Should have fog on camera • Should have an air timer running

So, when your avatar switches from walking to swimming, you would need to stop the footsteps sound effect, disable the door collisions, start playing the "swim" animation, start the underwater sound effect, enable the fog on the camera, and start the air timer.

You could do all this by hand, of course. But using an FSM can make it easier. In this simple model, you could define an FSM with two states, "Walk" and "Swim". This might be represented graphically like this:



To implement this as a Panda3D FSM, you would declare an new class that inherits from FSM. FSM, and within this class you would define four methods: enterWalk(), exitWalk(), enterSwim (), and exitSwim(). This might look something like this:

```
from direct.fsm import FSM

class AvatarFSM(FSM.FSM):
    def __init__(self):#optional because FSM already defines __init__
        #if you do write your own, you *must* call the base __init__ :
        FSM.FSM.__init__(self,'avatarFSM')
        ##do your init code here

    def enterWalk(self):
        avatar.loop('walk')
        footstepsSound.play()
        enableDoorCollisions()

    def exitWalk(self):
        avatar.stop()
        footstepsSound.stop()
        disableDoorCollisions()

    def enterSwim(self):
        avatar.loop('swim')
        underwaterSound.play()
        render.setFog(underwaterFog)
        startAirTimer()

    def exitSwim(self):
        avatar.stop()
        underwaterSound.stop()
        render.clearFog()
        stopAirTimer()

myfsm = AvatarFSM()
```

Keep in mind this is just an imaginary example, of course; but it should give you an idea of what an FSM class looks like.

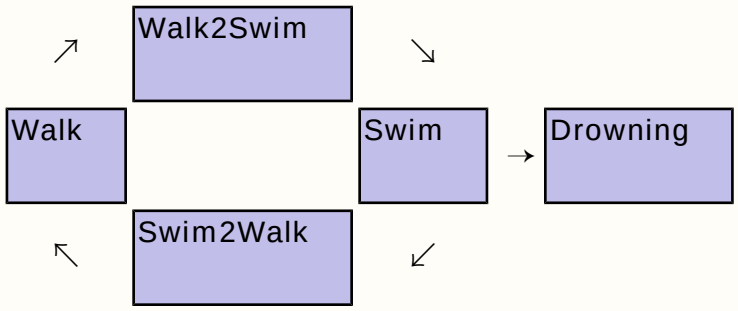
Note that each enter method activates everything that is important for its particular state, and--this is the important part--the corresponding exit method turns off or undoes *everything* that was turned on by the enter method. This means that whenever the FSM leaves a particular state, you can be confident that it will completely disable anything it started when it entered that state.

Now to switch from Walk state to Swim state, you would just need to request a transition, like this:

```
myfsm.request('Swim')
```

This FSM is a very simple example. Soon you will find the need for more than two states. For instance, you might want to play a transition animation while the avatar is moving from Walk state to Swim state and back again, and these can be encoded as separate states. There might

be a "drowning" animation if the avatar stays too long underwater, which again might be another state. Graphically, this now looks like this:



In a real-world example, you might easily find you have a need for dozens of states. This is when using the FSM class to manage all of these transitions for you can really make things a lot simpler; if you had to keep all of that cleanup code in your head, it can very quickly get out of hand.

Panda3D Manual: Simple FSM Usage

A Panda3D FSM is implemented by defining a new Python class which inherits from the class direct.fsm.FSM.FSM (normally imported as FSM.FSM or simply FSM), and defining the appropriate enter and exit methods on the class.

FSM states are represented by name strings, which should not contain spaces or punctuation marks; by Panda3D convention, state names should begin with a capital letter. An FSM is always in exactly one state a time; the name of the current state in stored in `fsm.state`. When it transitions from one state to another, it first calls `exitOldState()`, and then it calls `enterNewState()`, where OldState is the name of the previous state, and NewState is the name of the state it is entering. While it is making this transition, the FSM is not technically in either state, and `fsm.state` will be None--but you can find both old and new state names in `fsm.oldState` and `fsm.newState`, respectively.

To define a possible state for an FSM, you only need to define an `enterStateName()` and/or `exitStateName()` method on your class, where StateName is the name of the state you would like to define. The `enterStateName()` method should perform all the necessary action for entering your new state, and the corresponding `exitStateName()` method should generally undo everything that was done in `enterStateName()`, so that the world is returned to a neutral state.

An FSM starts and finishes in the state named "Off". When the FSM is created, it is already in "Off"; and when you destroy it (by calling `fsm.cleanup()`), it automatically transitions back to "Off".

To request an FSM to transition explicitly to a new state, use the call `fsm.request('StateName')`, where StateName is the state you would like it to transition to.

Arguments to enterStateName methods

Normally, both `enterStateName()` and `exitStateName()` take no arguments (other than self). However, if your FSM requires some information before it can transition to a particular state, you can define any arguments you like to the enterStateName method for that state; these arguments should be passed in to the `request()` call, following the state name.

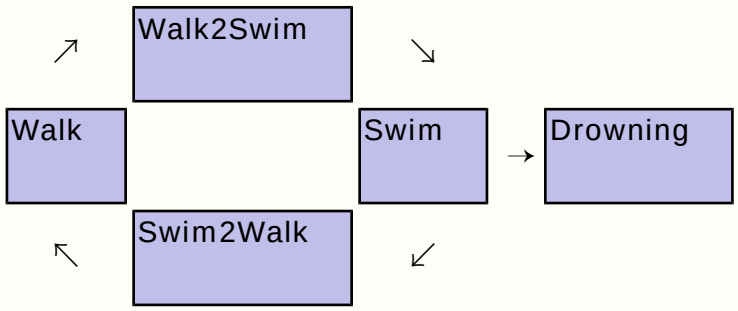
```
from direct.fsm import FSM

class AvatarFSM(FSM.FSM):
    def enterWalk(self, speed, doorMask):
        avatar.setPlayRate(speed, 'walk')
        avatar.loop('walk')
        footstepsSound.play()
        enableDoorCollisions(doorMask)

    def exitWalk(self):
        avatar.stop()
        footstepsSound.stop()
        disableDoorCollisions()

myfsm = AvatarFSM()
myfsm.request('Walk', 1.0, BitMask32.bit(2))
```

Note that the exitStateName method must always take no arguments.

Allowed and disallowed state transitions

By default, every state transition request is allowed: the call `fsm.request('StateName')` will always succeed, and the the FSM will be left in the new state. You may wish to make your FSM more robust by disallowing certain transitions that you don't want to happen.

For instance, consider the example FSM described previously, which had the following state diagram:



In this diagram, the arrows represent legal transitions. It is legal to transition from 'Walk' to 'Walk2Swim', but not from 'Walk' to 'Swim2Walk'. If you were to request the FSM to enter state 'Swim2Walk' while it is currently in state 'Walk', that's a bug; you might prefer to have the FSM throw an exception, so you can find this bug.

To enforce this, you can store `self.defaultTransitions` in the FSM's `__init__()` method. This should be a map of allowed transitions from each state. That is, each key of the map is a state name; for that key, the value is a list of allowed transitions from the indicated state. Any transition not listed in defaultTransitions is considered invalid. For example:

```
class AvatarFSM(FSM.FSM):
    def __init__(self):
        FSM.FSM.__init__(self)
        self.defaultTransitions = {
            'Walk' : [ 'Walk2Swim' ],
            'Walk2Swim' : [ 'Swim' ],
            'Swim' : [ 'Swim2Walk', 'Drowning' ],
            'Swim2Walk' : [ 'Walk' ],
            'Drowning' : [ ],
            }
```

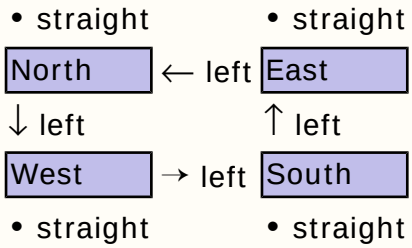
If you do not assign anything to self.defaultTransitions, then all transitions are legal. However, if you *do* assign a map like the above, then requesting a transition that is not listed in the map will raise the exception `FSM.RequestDenied`.

<<prev top next>> Search

Panda3D Manual: FSM with input

Another common use for FSM's is to provide an abstraction for AI state. For this purpose, you would like to supply an "input" string to the FSM, and let the FSM decide which state it should transition to, rather than explicitly specifying the target state name.

Consider the following FSM state diagram:



Here the text next to an arrow represents the "input" string given to the FSM, and the direction of the arrow represents the state transition that should be made for that particular input string, from the indicated starting state.

In this example, we have encoded a simple FSM that determines which compass direction a character will be facing after either turning left or continuing straight. The input will be either "left" or "straight", and the result is a transition to a new state that represents the new compass direction, based on the previous compass direction. If we request "left" from state North, the FSM transitions to state West. On the other hand, if we request "left" from state South, the FSM transitions to state East. If we request "straight" from any state, the FSM should remain in its current state.

To implement this in Panda3D, we define a number of **filter functions**, one for each state. The purpose of this function is to decide what state to transition to next, if any, on receipt of a particular input.

A filter function is created by defining a python method named `filterStateName()`, where StateName is the name of the FSM state to which this filter function applies. The filterStateName method receives two parameters, a string and a tuple of arguments (the arguments contain the optional additional arguments that might have been passed to the `fsm.request()` call; it's usually an empty tuple). The filter function should return the name of the state to transition to. If the transition should be disallowed, the filter function can either return None to quietly ignore it, or it can raise an exception. For example:

```
class CompassDir(FSM.FSM):
    def filterNorth(self, request, args):
        if request == 'straight':
            return 'North'
        elif request == 'left':
            return 'West'
        else:
            return None

    def filterWest(self, request, args):
        if request == 'straight':
            return 'West'
        elif request == 'left':
            return 'South'
        else:
            return None

    def filterSouth(self, request, args):
        if request == 'straight':
            return 'South'
        elif request == 'left':
            return 'East'
        else:
            return None

    def filterEast(self, request, args):
        if request == 'straight':
            return 'East'
        elif request == 'left':
            return 'North'
        else:
            return None
```

Note that input strings, by convention, should begin with a lowercase letter, as opposed to state names, which should begin with an uppercase letter. This allows you to make the distinction between requesting a state directly, and feeding a particular input string to an FSM. To feed input to this FSM, you would use the request() call, just as before:

```
myfsm.request('left')
myfsm.request('left')
myfsm.request('straight)
myfsm.request('left')
```

If the FSM had been in state North originally, after the above sequence of operations it would now be in state East.

The defaultFilter method

Although defining a series of individual filter methods gives you the most flexibility, for many

FSM's you may not need this much explicit control. For these cases, you can simply define a defaultFilter method that does everything you need. If a particular `filterStateName()` method does not exist, then the FSM will call the method named `defaultFilter()` instead; you can put any logic here that handles the general case.

For instance, we could have defined the above FSM using just the defaultFilter method, and a lookup table:

```
class CompassDir(FSM.FSM):
    nextState = {
        ('North', 'straight') : 'North',
        ('North', 'left') : 'West',
        ('West', 'straight') : 'West',
        ('West', 'left') : 'South',
        ('South', 'straight') : 'South',
        ('South', 'left') : 'East',
        ('East', 'straight') : 'East',
        ('East', 'left') : 'North',
        }

    def defaultFilter(self, request, args):
        key = (self.state, request)
        return self.nextState.get(key)
```

The base FSM class defines a `defaultFilter()` method that implements the default FSM transition rules (that is, allow all direct-to-state (uppercase) transition requests unless self. defaultTransitions is defined; in either case, quietly ignore input (lowercase) requests).

In practice, you can mix-and-match the use of the defaultFilter method and your own custom methods. The defaultFilter method will be called only if a particular state's custom filter method does not exist. If a particular state's filterStateName method *is* defined, that method will be called upon a new request; it can do any custom logic you require (and it can call up to the defaultFilter method if you like).

Panda3D Manual: Advanced FSM Tidbits

request vs. demand

As stated previously, you normally request an FSM to change its state by calling either `fsm.request('NewState', arg1, arg2, ...)`, or `fsm.request('inputString', arg1, arg2, ...)`, where arg1, arg2, ... represent optional arguments to the destination state's enter function (or to the filter function). The call to `request()` will either succeed or fail, according to what the filter function for the current state does. If it succeeds, it will return the tuple `('NewState', arg1, arg2)`, indicating the new state it has transitioned to. If it fails, it will simply return None (unless the filter function was written to throw an exception on failure).

If you request an FSM to make a transition, and the request fails, you might consider this an error condition, and you might prefer to have your code to stop right away rather than continuing. In this case, you should call `fsm.demand()` instead. The syntax is the same as that for `request()`, but instead of returning None on failure, it will always raise an exception if the state transition is denied. There is no return value from `demand()`; if it returns, the transition was accepted.

FSM.AlreadyInTransition

An FSM is always in exactly one state, except while it is in the process of transitioning between states (that is, while it is calling the exitStateName method for the previous state, followed by the enterStateName method for the new state). During this time, the FSM is not considered in either state, and if you query `fsm.state` it will contain None.

During this transition time, it is not legal to call `fsm.request()` to request a new state. If you try to do this, the FSM will raise the exception `FSM.AlreadyInTransition`. This is a particularly common error if some cleanup code that is called from the exitStateName method has a side-effect that triggers a transition to a new state.

However, there's a simple solution to this problem: call `fsm.demand()` instead. Unlike `request()`, `demand()` can be called while the FSM is currently in transition. When this happens, the FSM will queue up the demand, and will carry it out as soon as it has fully transitioned into its new state.

forceTransition()

There is also a method `fsm.forceTransition()`. This is similar to `demand()` in that it never fails and does not have a return value, but it's different in that it completely bypasses the filter function. You should therefore only pass an uppercase state name (along with any optional arguments) to forceTransition, never a lowercase input string. The FSM will always transition to the named state, even if it wouldn't otherwise be allowed. Thus, `forceTransition()` can be useful in special cases to skip to another state that's not necessarily connected to the current

state (for instance, to handle emergency cleanup when an exception occurs). Be careful that you don't overuse `forceTransition()`, though; consider whether `demand()` would be a better choice. If you find yourself making lots of calls to `forceTransition()`, it may be that your filter functions (or your defaultTransitions) are poorly written and are disallowing what should be legitimate state transitions.

Filtering the optional arguments

The filterStateName method receives two parameters: the string request, and a tuple, which contains the additional arguments passed to the request (or demand) call. It then normally returns the state name the FSM should transition to, or it returns None to indicate the transition is denied.

However, the filter function can also return a tuple. If it returns a tuple, it should be of the form ('StateName', arg1, arg2, ...), where arg1, arg2, ... represent the optional arguments that should be passed to the enterStateName method. Usually, these are the same arguments that were passed to the filterStateName method (in this case, you can generate the return value tuple with the python syntax `('StateName',) + args`).

The returned arguments are not necessarily the same as the ones passed in, however. The filter function is free to check, modify, or rearrange any of them; or it might even make up a completely new set of arguments. In this way, the filter function can filter not only the state transitions themselves, but also the set of data passed along with the request.

Panda3D Manual: Advanced operations with Panda's internal structures

The following pages provide descriptions of Panda's internal representation of vertices and the renderable geometry that uses them, as well as instructions for directly reading or manipulating this data.

This is an advanced topic of Panda3D and is not necessary for ordinary model rendering and animation, but the advanced user may find this information useful.

Panda3D Manual: How Panda3D Stores Vertices and Geometry

This section describes the structure and interconnections of Panda's internal vertex and geometry data objects, in general terms.

You should read through this section carefully, so that you have a good understanding of Panda's data structures, before attempting to read the section about generating procedural data.

Panda3D Manual: GeomVertexData

The fundamental object used to store vertex information in Panda is the **GeomVertexData**. This stores a list of vertices, organized conceptually as a table, where each row of the table represents a different vertex, and the columns of the table represent the different kinds of per-vertex data that may be associated with each vertex. For instance, the following table defines four vertices, each with its own vertex position, normal vector, color, and texture coordinate pair:

	vertex	normal	color	texcoord
0	(1, 0, 0)	(0, 0, 1)	(0, 0, 1, 1)	(1, 0)
1	(1, 1, 0)	(0, 0, 1)	(0, 0, 1, 1)	(1, 1)
2	(0, 1, 0)	(0, 0, 1)	(0, 0, 1, 1)	(0, 1)
3	(0, 0, 0)	(0, 0, 1)	(0, 0, 1, 1)	(0, 0)

Vertices are always numbered beginning at 0, and continue to the number of rows in the table (minus 1).

Not all GeomVertexData objects will use these same four columns; some will have fewer columns, and some will have more. In fact, all columns, except for "vertex", which stores the vertex position, are optional.

The order of the columns is not meaningful, but the column names are. There are certain column names that are reserved for Panda, and instruct Panda what the meaning of each column is. For instance, the vertex position column is always named "vertex", and the lighting normal column, if it is present, must be named "normal". See GeomVertexFormat for the complete list of reserved column names.

You can define your own custom columns. If there are any columns that have a name that Panda does not recognize, Panda will not do anything special with the column, but it can still send it to the graphics card. Of course, it is then up to you to write a vertex shader that understands what to do with the data in the column.

It is possible to break up a GeomVertexData into more than one array. A **GeomVertexArray** is table of vertex data that is stored in one contiguous block of memory. Typically, each GeomVertexData consists of just one array; but it is also possible to distribute the data so that some columns are stored in one array, while other columns are stored in another array:

	vertex	texcoord	normal	color
0	(1, 0, 0)	(1, 0)	(0, 0, 1)	(0, 0, 1, 1)
1	(1, 1, 0)	(1, 1)	(0, 0, 1)	(0, 0, 1, 1)
2	(0, 1, 0)	(0, 1)	(0, 0, 1)	(0, 0, 1, 1)
3	(0, 0, 0)	(0, 0)	(0, 0, 1)	(0, 0, 1, 1)

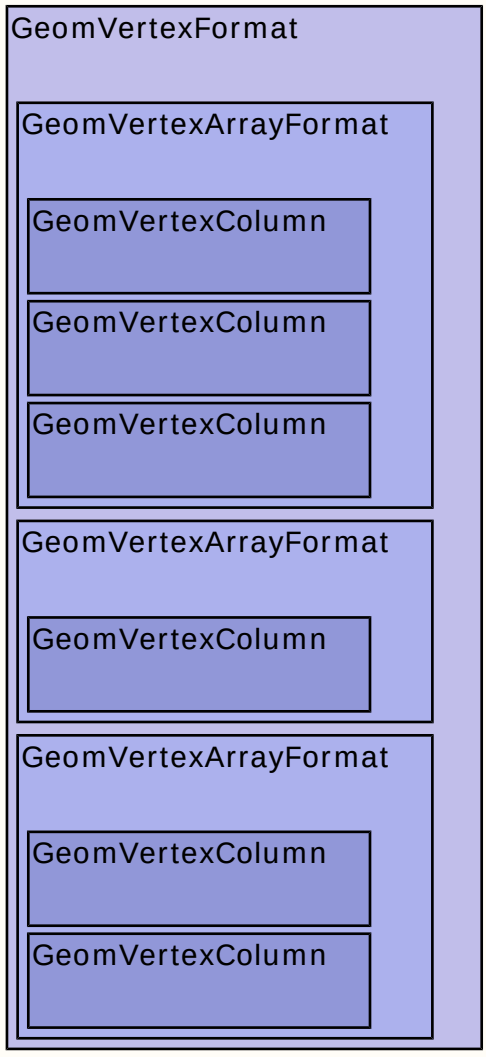
You might want to do this, for instance, if you have certain columns of data that are always the same between different blocks of vertices; you can put those columns in a separate array, and then use the same array within multiple different GeomVertexData objects. There is no limit to the number of different arrays you can have within one GeomVertexData; you can make each column a separate array if you like. (There may be performance implications to consider. Some graphics drivers may work better with one block of contiguous data--one array--while others may prefer many different arrays. This performance difference is likely to be small, however.)

<<prev top next>> Search

Panda3D Manual: GeomVertexFormat

The **GeomVertexFormat** object describes how the columns of a GeomVertexData are ordered and named, and exactly what kind of numeric data is stored in each column. Every GeomVertexData has an associated GeomVertexFormat, which describes how the data in that object is stored.

Just as a GeomVertexData object is really a list of one or more GeomVertexArrayData objects, a GeomVertexFormat object is a list of one or more **GeomVertexArrayFormat** objects, each of which defines the structure of the corresponding array. There will be one GeomVertexArrayFormat object for each array in the format. Each GeomVertexArrayFormat, in turn, consists of a list of **GeomVertexColumn** objects, one for each column in the array. For instance, the format for a GeomVertexData with six columns, distributed over three different arrays, might look like this:



Each GeomVertexColumn has a number of properties:

getNumComponents()	This defines the number of numeric components of the data in the column. For instance, the vertex position, which is typically an (X, Y, Z) triple, has three components: X, Y, and Z. A texture coordinate usually has two components (U, V), but sometimes has three components (U, V, W).
getNumericType()	This defines the kind of numeric data that is stored in each component. It must be one of the following symbols:

Geom.NTFloat32	Each component is a 32-bit floating-point number. This is by far the most common type.
Geom.NTUint8	Each component is a single 8-bit integer, in the range 0 - 255. OpenGL encodes an RGBA color value as a four-component array of 8-bit integers of this type, in R, G, B, A order.
Geom.NTUint16	Each component is a single 16-bit integer, in the range 0 - 65535.
Geom.NTUint32	Each component is a single 32-bit integer, in the range 0 - 4294967295.
Geom.NTPackedDcba	Each component is a 32-bit word, with four 8-bit integer index values packed into it in little-endian order (D, C, B, A), DirectX-style. This is usually used with a 1-component column (since each component already has four values). DirectX uses this format to store up to four indexes into a transform table for encoding vertex animation. (The GeomVertexReader and GeomVertexWriter classes will automatically reorder the A, B, C, D parameters you supply into DirectX's D, C, B, A order.)

Geom.NTPackedDabc	Each component is a 32-bit word, with four 8-bit integer index values packed into it in ARGB order (D, A, B, C). As above, this is normally used with a 1-component column. DirectX uses this format to represent an RGBA color value. (The GeomVertexReader and GeomVertexWriter classes will automatically reorder the R, G, B, A parameters you supply into DirectX's A, R, G, B order.)

getContents()

This defines, in a general way, the semantic meaning of the data in the column. It is used by Panda to decide how the data should be modified when a transform matrix or texture matrix is applied; it also controls the default value for the column data, as well as the way data is stored and fetched from the column.

The contents specification must be one of the following symbols:

Geom.CPoint	The data represents a point in object coordinates, either in 3-D space (if it is a 3-component value) or in 4-D homogenous space (if it is a 4-component value). When a transform matrix is applied to the vertex data, the data in this column is transformed as a point. If a 4-component value is stored into a 3-component column, the fourth component is understood to be a homogenous coordinate, and it implicitly scales the first three. Similarly, if a 4-component value is read from a 3-component column, the fourth value is implicitly 1.0.
Geom.CClipPoint	The data represents a point already transformed into clip coordinates; that is, these points have already been transformed for rendering directly. Panda will not transform the vertices again during rendering. Points in clip coordinates should be in 4-D homogeneous space, and thus usually have four components.

Geom.CVector	The data represents a 3-D vector, such as a normal, tangent, or binormal, in object coordinates. When a transform matrix is applied to the vertex data, the data in this column is transformed as a vector (that is, ignoring the matrix's translation component).
Geom.CTexcoord	The data represents a texture coordinate, either 2-D or 3-D. When a texture matrix (not a transform matrix) is applied to the vertex data, it transforms the data in this column, as a point.
Geom.CColor	The data represents an RGBA color value. If a floating-point value is used to read or write into an integer color component, it is automatically scaled from 0.0 .. 1.0 into the full integer range. Also, the default value of a color column is (1, 1, 1, 1), as opposed to any other column type, whose default value is 0.
Geom.CIndex	The data represents an integer index into some table.
Geom.CMorphDelta	The data represents an offset value that will be applied to some other column during animation.
Geom.COther	The data has some other, custom meaning; do not attempt to transform it.

getName()

The column name is the most important single piece of information to Panda. The column name tells Panda the specific meaning of the data in the column. The name is also a unique handle to the column; within a given GeomVertexFormat, there may not be two different columns with the same name.

There are a number of column names that have special meaning to Panda:

vertex	The position in space of each vertex, usually given as an (x, y, z) triple in 3-D coordinates. This is the only mandatory column for rendering geometry; all other columns are optional. The vertex is usually Geom. NTFloat32, Geom.CPoint, 3 components.
normal	The surface normal at each vertex. This is used to compute the visible effects of lighting; it is not related to the collision system, which has its own mechanism for determining the surface normal. You should have a normal column if you intend to enable lighting; if this column is not present, the object may look strange in the presence of lighting. The normal should always be Geom.NTFloat32, Geom.CVertex, 3 components.
texcoord	The U, V texture coordinate pair at each vertex, for the default coordinate set. This column is necessary in order to apply a texture to the geometry (unless you use a TexGenAttrib). It is usually a 2-D coordinate pair, but sometimes, when you are using 3-d textures or cube maps, you will need a 3-D U, V, W coordinate triple. The texcoord should be Geom.NTFloat32, Geom.CTexcoord, 2 or 3 components.

texcoord.*foo*	This is the U, V texture coordinate pair for the texture coordinate set with the name *"foo"* (where *foo* is any arbitrary name). It is only necessary if you need to have multiple different texture coordinate sets on a piece of geometry, in order to apply multitexturing. As with texcoord, above, it may be a 2-d or a 3-d value.
tangent **binormal**	These two columns work together, along with the normal column, to implement normal maps (bump maps). They define the normal map space at each vertex. Like a normal, these should be Geom.NTFloat32, Geom.CVertex, 3 components.
tangent.*foo* **binormal.*foo***	These column names define a tangent and binormal for the texture coordinate set with the name *"foo"*.
color	This defines an RGBA color value. If this column is not present, the default vertex color is white (unless it is overridden with a nodePath.setColor() call). Internally, OpenGL expects the color format to be Geom.NTUint8 (or Geom.NTFloat32), Geom.CColor, 4 components, while DirectX expects the color to be Geom.NTPackedDabc, Geom.CColor, 1 component. In fact, you may use either format regardless of your current rendering backend, and Panda will automatically convert the column as necessary.
rotate **size** **aspect_ratio**	These three columns are used when rendering sprites (that is, GeomPoints with nodePath. setRenderModeThickness() in effect). If present, they control the rotation counterclockwise in degrees, the per-vertex thickness, and the aspect ratio of the square, respectively. Each of these should be Geom.NTFloat32, Geom.COther, 1 component.

The remaining column names have meaning only to define vertex animation, for instance to implement Actors. Although these column names are documented below, vertex animation is an advanced feature of the Panda vertex representation; we recommend you let Panda take care of setting up the vertex animation tables, rather than attempting to create them yourself.

transform_blend	This is used to control vertex assignment to one or more animated transform spaces. The value in this column is an integer index into the TransformBlendTable that is associated with the GeomVertexData; each entry in the TransformBlendTable defines a different weighted combination of transform spaces, so by indexing into this table, you can associate each vertex with a different weighted combination of transform spaces.
transform_weight **transform_index**	These two columns work together, in a manner similar to transform_blend, but they index into the TransformTable associated with the GeomVertexData, instead of the TransformBlendTable. This is particularly suited for sending vertices to OpenGL or DirectX to do the animation, rather than performing the animation on the CPU.
column*.morph.*slider	Columns with names of this form define a floating-point morph offset that should be scaled by the value of the morph slider named *"slider"*, and then added to the column named *"column"* (where *slider* and *column* are arbitrary names). This is used during vertex animation on the CPU.

A column may have any name (though each name must be unique within a given GeomVertexFormat). If there are additional columns with names other than those in the above table, Panda will not do anything special with the columns, but it will send the vertex data to any vertex shader that requests that data by name, using the vtx_*columnname* parameter name. See List of Possible Shader Inputs. </tr> <td valign="top" style="border-top: 1px solid black; padding:

5pt"></td> <td style="border-top: 1px solid black; padding: 5pt"></td> </td></tr> </table>

There are also additional properties associated with each GeomVertexColumn that determine its exact offset and byte-alignment within each row of the array, but normally you do not need to worry about these, unless you are designing a GeomVertexFormat that matches some already-existing block of data. See the auto-generated API specification for more details.

<<prev top next>>

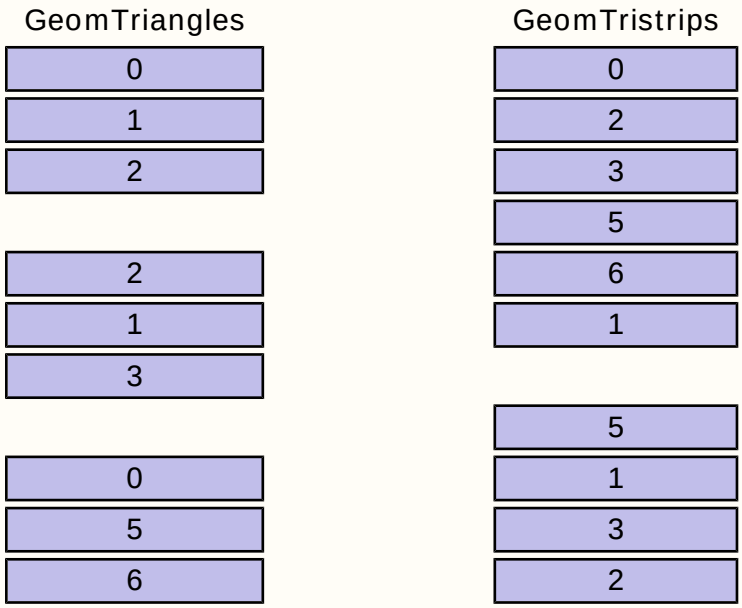
Search

Panda3D Manual: GeomPrimitive

In order to use the vertices in a GeomVertexData to render anything, Panda needs to have a **GeomPrimitive** of some kind, which indexes into the vertex table and tells Panda how to tie together the vertices to make lines, triangles, or individual points.

There are several different kinds of GeomPrimitive objects, one for each different kind of primitive. Each GeomPrimitive object actually stores several different individual primitives, each of which is represented simply as a list of vertex numbers, indexing into the vertices stored in the associated GeomVertexData. For some GeomPrimitive types, like GeomTriangles, all the primitives must have a fixed number of vertex numbers (3, in the case of GeomTriangles); for others, like GeomTristrips, each primitive can have a different number of vertex numbers.

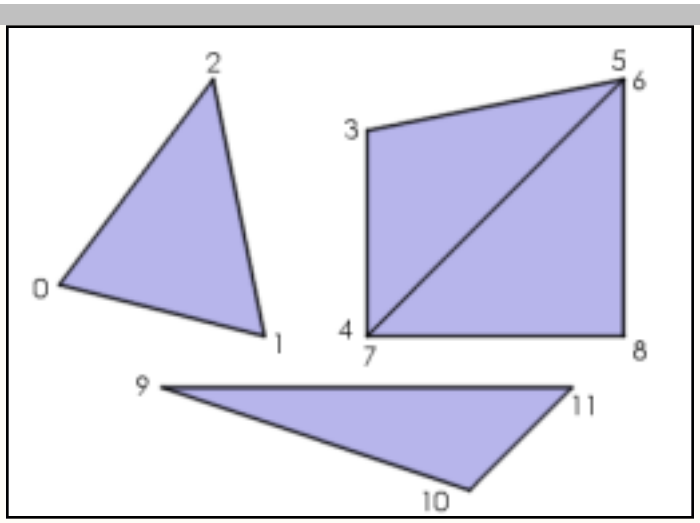
For instance, a GeomTriangles object containing three triangles, and a GeomTristrips containing two triangle strips, might look like this:



Note that the GeomPrimitive objects don't themselves contain any vertex data; they only contain a list of vertex index numbers, which is used to look up the actual vertex data in a GeomVertexData object, stored elsewhere.

GeomTriangles

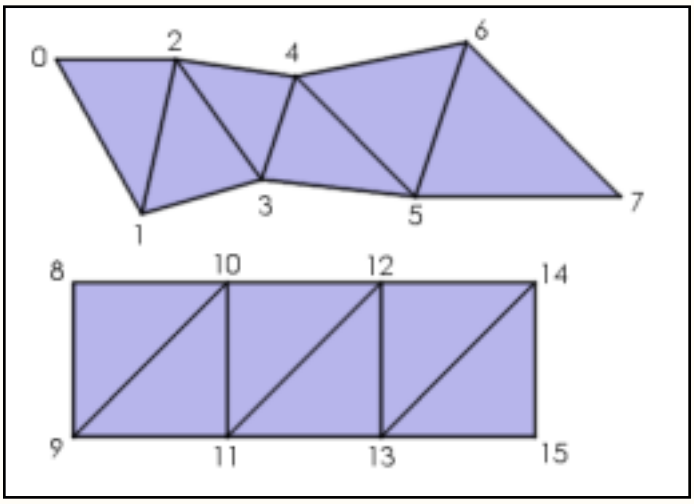
This is the most common kind of GeomPrimitive. This kind of primitive stores any number of connected or unconnected triangles. Each triangle must have exactly three vertices, of course. In each triangle, the vertices should be listed in counterclockwise order, as seen from the front of the triangle.



GeomTristrips

This kind of primitive stores lists of connected triangles, in a specific arrangement called a triangle strip. You can store any number of individual triangle strips in a single GeomTristrips object, and each triangle strip can have an arbitrary number of vertices (at least three).

The first three vertices of a triangle strip define one triangle, with the vertices listed in counterclockwise order. Thereafter, each additional vertex defines an additional triangle, based on the new vertex and the preceding two vertices. The vertices go back and forth, defining triangles in a zig-zag fashion.



Note that the second triangle in a triangle strip is defined in clockwise order, the third triangle is in counterclockwise order, the fourth triangle is in clockwise order again, and so on.

On certain hardware, particularly older SGI hardware and some console games, using triangle strips is an important optimization to reduce the number of vertices that are sent to the graphics pipe, since most triangles (except for the first one) can be defined with only a single vertex, rather than three vertices for each triangle.

Modern PC graphics cards prefer to receive a group of triangle strips connected together into one very long triangle strip, by the introduction of repeated vertices and degenerate triangles.

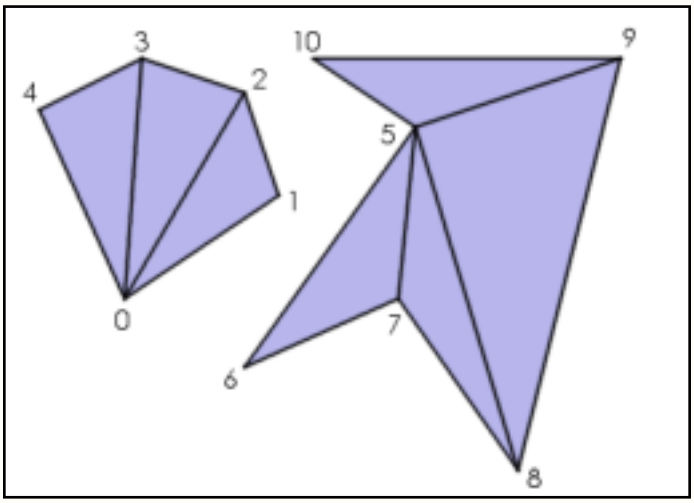
Panda will do this automatically, but in order for this to work you should ensure that every triangle strip has an even number of vertices in it.

Furthermore, since modern PC graphics cards incorporate a short vertex cache, they can generally render individual, indexed triangles as fast as triangle strips; so triangle strips are less important on PC hardware than they have been in the past. Unless you have a good reason to use a GeomTristrips, it may be easier just to use GeomTriangles.

When loading a model from an egg file, Panda will assemble the polygons into triangle strips if it can do so without making other compromises; otherwise, it will leave the polygons as individual triangles.

GeomTrifans

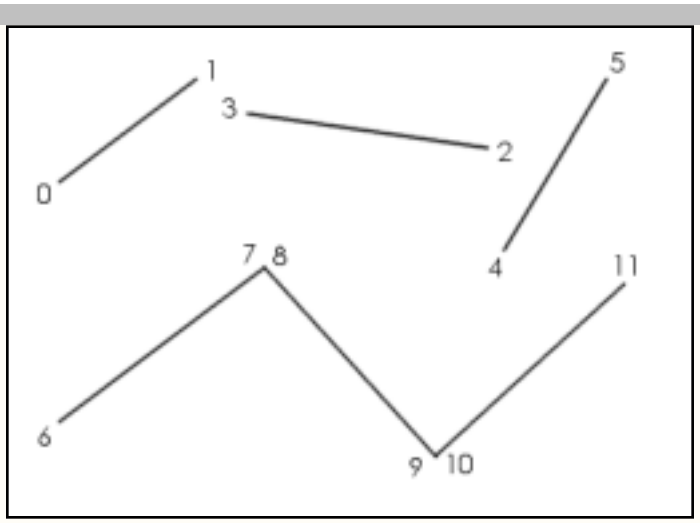
This is similar to a GeomTristrips, in that the primitive can contain any number of triangle fans, each of which has an arbitrary number of vertices. Within each triangle fan, the first three vertices (in counterclockwise order) define a triangle, and each additional vertex defines a new triangle. However, instead of using the preceding two vertices to define each new triangle, a triangle fan uses the previous vertex and the *first* vertex, which means that all of the resulting triangles fan out from a single point, like this:



Like the triangle strip, a triangle fan can be an important optimization on certain hardware. However, its use can actually incur a performance *penalty* on modern PC hardware, because it is impossible to send more than one triangle fan in one batch, so you probably shouldn't use triangle fans on a PC. Use GeomTriangles or GeomTristrips instead.

GeomLines

This kind of GeomPrimitive stores any number of connected or unconnected line segments. It is similar to a GeomTriangles, but it draws lines instead of triangles. Each line has exactly two vertices.

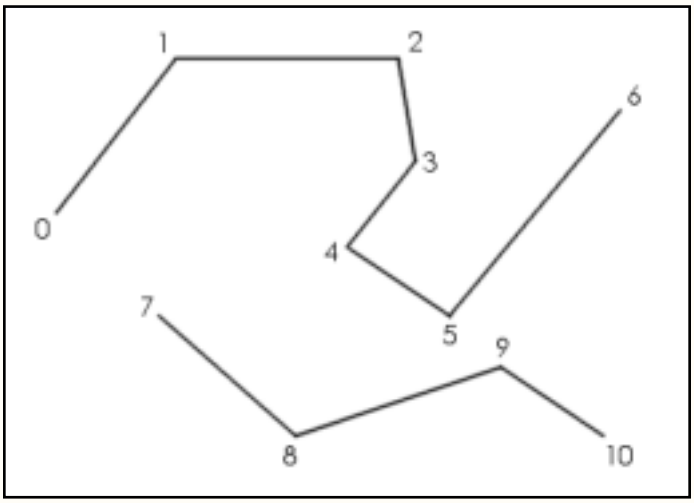


By default, line segments are one pixel wide, no matter how far away they are from the camera. You can use **nodePath.setRenderModeThickness()** to change this; if you specify a thickness greater than 1, this will make the lines render as thick lines, the specified number of pixels wide. However, the lines will always be the same width in pixels, regardless of how far away from the camera they are.

Thick lines are not supported by the DirectX renderer; in DirectX, the thickness parameter is ignored.

GeomLinestrips

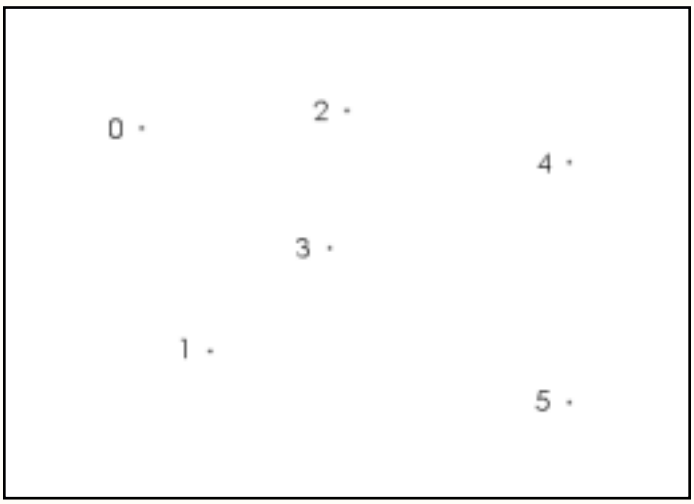
This is the analogue of a GeomTristrips object: the GeomLinestrips object can store any number of line strips, each of which can have any number of vertices, at least two. Within a particular line strip, the first two vertices define a line segment; and thereafter, each new vertex defines an additional line segment, connected end-to-end with the previous line segment. This primitive type can be used to draw a curve approximation with many bends fairly easily.



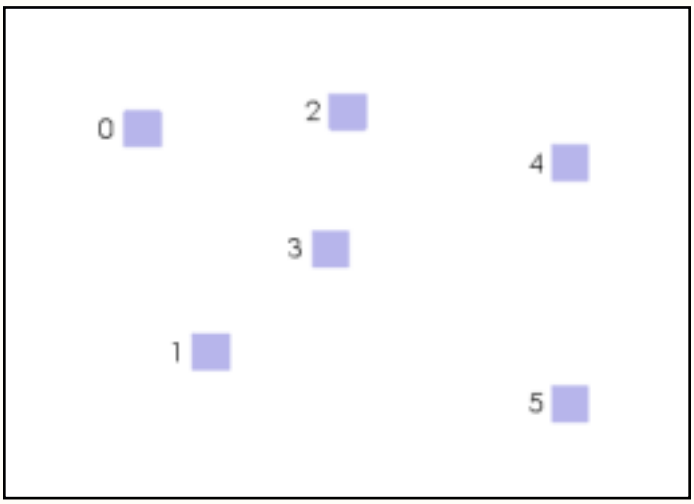
GeomPoints

This is the simplest kind of GeomPrimitive; it stores a number of individual points. Each point

has exactly one vertex.



By default, each point is rendered as one pixel. You can use **nodePath.setRenderModeThickness()** to change this; if you specify a thickness greater than 1, this will make the points render as squares (which always face the camera), where the vertex coordinate is the center point of the square, and the square has the specified number of pixels along each side. Each point will always be the same width in pixels, no matter how far it is from the camera. Unlike line segments, thick points *are* supported by DirectX.



In addition to ordinary thick points, which are always the same size no matter how far they are from the camera, you can also use **nodePath.setRenderModePerspective()** to enable a mode in which the points scale according to their distance from the camera. This makes the points appear more like real objects in the 3-D scene, and is particularly useful for rendering sprite polygons, for instance for particle effects. In fact, Panda's SpriteParticleRenderer takes advantage of this render mode. (This perspective mode works only for points; it does not affect line segments.)

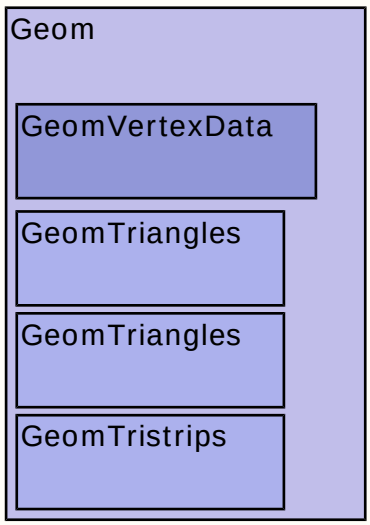
Even though the sprite polygons are rendered as squares, remember they are really defined with one vertex, and each vertex can only supply one UV coordinate. This means each sprite normally has only one UV coordinate pair across the whole polygon. If you want to apply a texture to the face of each sprite, use nodePath.setTexGen() with the mode **TexGenAttrib.MPointSprite**; this will generate texture coordinates on each polygon in the range (0, 0) to

(1, 1). You can then transform the texture coordinates, if you wish, using one of the methods like nodePath.setTexOffset(), setTexScale(), etc.

Panda3D Manual: Geom

The **Geom** object collects together a GeomVertexData and one or more GeomPrimitive objects, to make a single renderable piece of geometry. In fact, an individual Geom is the smallest piece into which Panda will subdivide the scene for rendering; in any given frame, either an entire Geom is rendered, or none of it is.

Fundamentally, a Geom is very simple; it contains a pointer to a single GeomVertexData, and a list of one or more GeomPrimitives, of various types, as needed. All the associated GeomPrimitives index into the same GeomVertexData.

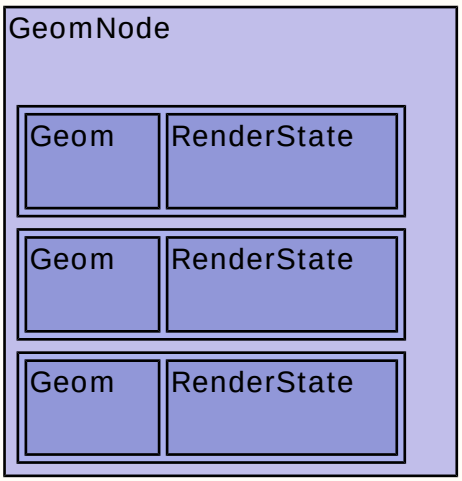


The GeomVertexData pointer may be unique to each Geom, or one GeomVertexData may be shared among many different Geoms (each of which might use a different subset of its vertices). Also, although the GeomPrimitive objects are usually unique to each Geom, they may also be shared between different Geoms.

Although a Geom can have any number of GeomPrimitives associated with it, all of the GeomPrimitives must be of the same fundamental primitive type: triangles, lines, or points. A particular Geom might have GeomTriangles, GeomTristrips, and GeomTrifans; or it might have GeomLines and GeomLinestrips; or it might have GeomPoints. But no one Geom can have primitives from two different fundamental types. You can call **geom.getPrimitiveType()** to determine the fundamental primitive type stored within a particular Geom.

Panda3D Manual: GeomNode

Finally, **GeomNode** is the glue that connects Geoms into the scene graph. A GeomNode contains a list of one or more Geoms.



The GeomNode class inherits from PandaNode, so a GeomNode can be attached directly to the scene graph like any other node; and like any node, it inherits a transform and a render state from its parents in the scene graph. This transform and state is then applied to each of the node's Geoms.

Furthermore, the GeomNode stores an additional render state definition for each Geom. This allows each Geom within a given GeomNode to have its own unique state; for instance, each Geom may have a different texture applied.

When a model is loaded from an egg file, normally all the state definitions required to render the geometry will be stored on these per-Geom state definitions, rather than at the GeomNode level. These per-Geom states will override any state that is inherited from the scene graph, unless that scene graph state has a priority higher than the default priority of zero. (This is why it is necessary to specify a second parameter of 1 to the nodePath.setTexture() call, if you want to replace a texture that was applied to a model in the egg file.)

Panda3D Manual: Procedurally Generating 3D Models

Building on the fundamental concepts introduced in the last section, the following section explains how to use Panda's basic geometry building blocks to generate your own custom geometry at runtime.

Panda3D Manual: Defining your own GeomVertexFormat

Before you can create any geometry in Panda3D, you must have a valid **GeomVertexFormat**. You can decide exactly which columns you want to have in your format, by building the format up one column at a time. (But you might be able to avoid this effort by taking advantage of one of the pre-defined formats listed on the next page.)

To build up your custom format, you need to first create an empty **GeomVertexArrayFormat**, and add columns one at a time by calling addColumn():

```
array = GeomVertexArrayFormat()
array.addColumn(InternalName.make('vertex'),
3,
                Geom.NTFloat32, Geom.CPoint)
```

The parameters to addColumn() are, in order, the column name, the number of components, the numeric type, and the contents specification. See GeomVertexFormat for a detailed description of each of these parameters and their appropriate values. You may also supply an optional fifth parameter, which specifies the byte offset within the row at which the column's data begins; but normally you should omit this to indicate that the column's data immediately follows the previous column's data.

Note that the column name should be an **InternalName** object, as returned by a call to InternalName.make(). This is Panda's mechanism for tokenizing a string name, to allow for fast name lookups during rendering. Other than this detail, the column name is really just an arbitrary string.

It is your responsibility to ensure that all of the parameters passed to addColumn() are appropriate for the column you are defining. The column data will be stored exactly as you specify. When rendering, Panda will attempt to convert the column data as it is stored to whatever format your graphics API (e.g. OpenGL or DirectX) expects to receive.

For instance, to define a vertex format that includes a vertex position and a (U, V) texture coordinate:

```
array = GeomVertexArrayFormat()
array.addColumn(InternalName.make('vertex'), 3,
                Geom.NTFloat32, Geom.CPoint)
array.addColumn(InternalName.make('texcoord'),
2,
                Geom.NTFloat32, Geom.CTexcoord)
```

Once you have defined the columns of your array, you should create a GeomVertexFormat to hold the array:

```
format = GeomVertexFormat()
format.addArray(array)
```

If you want your format to consist of multiple different arrays, you can create additional arrays and add them at this point as well.

Finally, before you can use your new format, you must *register* it. Registering a format builds up the internal tables necessary to use the vertex format for rendering. However, once you have registered a format, you can no longer add or remove columns, or modify it in any way; if you want to make changes to the format after this point, you'll have to start over with a new GeomVertexFormat object.

```
format = GeomVertexFormat.registerFormat(format)
```

You should always register a format with a syntax similar to the above: that is, you should use the return value of registerFormat as your new, registered format object, and discard the original format object. (The returned format object may be the same format object you started with, or it may be a different object with an equivalent meaning. Either way, the format object you started with should be discarded.)

Panda3D Manual: Pre-defined vertex formats

Panda3D pre-defines a handful of standard GeomVertexFormat objects that might be useful to you. If you don't have any special format needs, feel free to use any of these standard formats, which have already been defined and registered, and are ready to use for rendering.

Each of these formats includes one or more of the standard columns vertex, normal, color, and/or texcoord. For the formats that include a color column, there are two choices, since OpenGL and DirectX have competing internal formats for color (but you can use either form regardless of your current rendering API; Panda will automatically convert the format at render time if necessary).

Standard format	**vertex** (X, Y, Z)	**normal** (X, Y, Z)	**color**, 4-component RGBA (OpenGL style)	**color**, packed RGBA (DirectX style)	**texcoord** (U, V)
GeomVertexFormat.getV3()	✓				
GeomVertexFormat.getV3n3()	✓	✓			
GeomVertexFormat.getV3t2()	✓				✓
GeomVertexFormat.getV3n3t2()	✓	✓			✓
GeomVertexFormat.getV3c4()	✓		✓		
GeomVertexFormat.getV3c4n3()	✓	✓	✓		
GeomVertexFormat.getV3c4t2()	✓		✓		✓
GeomVertexFormat.getV3c4n3t2()	✓	✓	✓		✓
GeomVertexFormat.getV3cp()	✓			✓	
GeomVertexFormat.getV3cpn3()	✓	✓		✓	
GeomVertexFormat.getV3cpt2()	✓			✓	✓
GeomVertexFormat.getV3cpn3t2()	✓	✓		✓	✓

Panda3D Manual: Creating and filling a GeomVertexData

Once you have a GeomVertexFormat, registered and ready to use, you can use it to create a **GeomVertexData**.

```
vdata = GeomVertexData('name', format, Geom.UHStatic)
```

The first parameter to the GeomVertexData constructor is the name of the data, which is any arbitrary name you like. This name is mainly for documentation purposes; it may help you identify this vertex data later. You can leave it empty if you like.

The second parameter is the GeomVertexFormat to use for this GeomVertexData. The format specifies the number of arrays that will be created for the data, the names and formats of the columns in each array, and the number of bytes that need to be allocated for each row.

The third parameter is a usage hint, which tells Panda how often (if ever) you expect to be modifying these vertices, once you have filled them in the first time. If you will be filling in the vertices once (or only once in a while) and using them to render many frames without changing them, you should use Geom.UHStatic. The vast majority of vertex datas are of this form. Even GeomVertexDatas that include vertex animation tables should usually be declared Geom.UHStatic, since the vertex data itself will not be changing (even though the vertices might be animating).

However, occasionally you might create a GeomVertexData whose vertices you intend to adjust in-place every frame, or every few frames; in this case, you can specify Geom.UHDynamic, to tell Panda not to make too much effort to cache the vertex data. This is just a performance hint; you're not required to adhere to the usage you specify, though you may get better render performance if you do.

If you are unsure about this third parameter, you should probably use Geom.UHStatic.

Now that you have created a GeomVertexData, you should create a number of GeomVertexWriters, one for each column, to fill in the data.

```
vertex = GeomVertexWriter(vdata, 'vertex')
normal = GeomVertexWriter(vdata, 'normal')
color = GeomVertexWriter(vdata, 'color')
texcoord = GeomVertexWriter(vdata, 'texcoord')
```

It is your responsibility to know which columns exist in the GeomVertexFormat you have used. It is legal to create a GeomVertexWriter for a column that doesn't exist, but it will be an error if you later attempt to use it to add data.

To add data, you can now iterate through your vertices and call one of the addData methods on each GeomVertexWriter.

```
vertex.addData3f(1, 0, 0)
normal.addData3f(0, 0, 1)
color.addData4f(0, 0, 1, 1)
texcoord.addData2f(1, 0)

vertex.addData3f(1, 1, 0)
normal.addData3f(0, 0, 1)
color.addData4f(0, 0, 1, 1)
texcoord.addData2f(1, 1)

vertex.addData3f(0, 1, 0)
normal.addData3f(0, 0, 1)
color.addData4f(0, 0, 1, 1)
texcoord.addData2f(0, 1)

vertex.addData3f(0, 0, 0)
normal.addData3f(0, 0, 1)
color.addData4f(0, 0, 1, 1)
texcoord.addData2f(0, 0)
```

Each call to addData() adds a new row (vertex) to the vertex data, if there is not already one there. The above sample code creates the following data table:

	vertex	normal	color	texcoord
0	(1, 0, 0)	(0, 0, 1)	(0, 0, 1, 1)	(1, 0)
1	(1, 1, 0)	(0, 0, 1)	(0, 0, 1, 1)	(1, 1)
2	(0, 1, 0)	(0, 0, 1)	(0, 0, 1, 1)	(0, 1)
3	(0, 0, 0)	(0, 0, 1)	(0, 0, 1, 1)	(0, 0)

Note that there is no relationship between the different GeomVertexWriters, other than the fact that they are operating on the same table. Each GeomVertexWriter maintains its own counter of its current row. This means you must fill in the data for every row of each column, even if you don't care about writing the data for some particular column on certain rows. For instance, even if you want to allow the default color for vertex 1 and 2, you must still call color.addData4f() four times, in order to fill in the color value for vertex 3.

Panda3D Manual: Creating the GeomPrimitive objects

Now that you have a GeomVertexData with a set of vertices, you can create one or more **GeomPrimitive** objects that use the vertices in your GeomVertexData.

In general, you do this by first creating a GeomPrimitive of the appropriate type, and then calling addVertex() for each vertex in your primitive, followed by closePrimitive() after each primitive is complete.

Different GeomPrimitive types have different requirements for the number of vertices per primitive. For instance, for a GeomTriangles object, whose primitive type (triangle) has exactly three vertices each, you should call addVertex() three times, followed by closePrimitive() after every three vertices. For a GeomTristrips object, whose primitive type (triangle strip) may have three or more vertices, you should call addVertex() once for each vertex in your triangle strip, followed by closePrimitive() to mark the end of the triangle strip. Then you can begin adding vertices for the second triangle strip in the primitive, and so on.

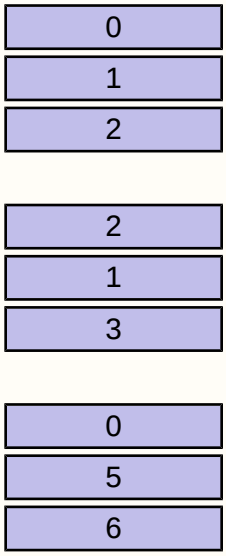
For example:

```
prim = GeomTriangles(Geom.UHStatic)

prim.addVertex(0)
prim.addVertex(1)
prim.addVertex(2)
prim.closePrimitive()

prim.addVertex(2)
prim.addVertex(1)
prim.addVertex(3)
prim.closePrimitive()

prim.addVertex(0)
prim.addVertex(5)
prim.addVertex(6)
prim.closePrimitive()
```

Note that the GeomPrimitive constructor requires one parameter, which is a usage hint, similar to the usage hint required for the GeomVertexData constructor. Like that usage hint, this tells Panda whether you will frequently adjust the vertex indices on this primitive after it has been created. Since it is very unusual to adjust the vertex indices on a primitive (usually, if you intend to animate the vertices, you would operate on the vertices, not these indices), this is almost always Geom.UHStatic, even if the primitive is associated with a dynamic GeomVertexData. However, there may be special rendering effects in which you actually do manipulate this vertex index table in-place every few frames, in which case you should use Geom.UHDynamic. As with the GeomVertexData, this is only a performance hint; you're not required to adhere to the usage you specify.

If you are unsure about this parameter, you should use Geom.UHStatic.

The above sample code defines a GeomTriangles object that looks like this:



The actual positions of the vertices depends on the values of the vertices numbered 0, 1, 2, 3, and 5 in the associated GeomVertexData (you will associate your GeomPrimitives with a GeomVertexData in the next step, when you attach the GeomPrimitives to a Geom).

Finally, there are a few handy shortcuts for adding multiple vertices at once:

prim.addVertices(v1, v2) prim.addVertices(v1, v2, v3) prim.addVertices(v1, v2, v3, v4)	Adds 2, 3, or 4 vertices in a single call.
prim.addConsecutiveVertices(start, numVertices)	Adds *numVertices* consecutive vertices, beginning at vertex *start*. For instance, addConsecutiveVertices(5, 3) adds vertices 5, 6, 7.
prim.addNextVertices(numVertices)	Adds *numVertices* consecutive vertices, beginning with the next vertex after the last vertex you added, or beginning at vertex 0 if these are the first vertices. For instance, prim.addVertex(10) adds vertex 10. If you immediately call prim.addNextVertices(4), it adds vertices 11, 12, 13, 14.

None of the above shortcut methods calls closePrimitive() for you; it is still your responsibility to call closePrimitive() each time you add the appropriate number of vertices.

Panda3D Manual: Putting your new geometry in the scene graph

Finally, now that you have a GeomVertexData and one or more GeomPrimitive objects, you can create a **Geom** object and a **GeomNode** to put the new geometry in the scene graph, so that it will be rendered.

```
geom = Geom(vdata)
geom.addPrimitive(prim)

node = GeomNode('gnode')
node.addGeom(geom)

nodePath = render.attachNewNode(node)
```

The Geom constructor requires a pointer to the GeomVertexData object you will be using. There is only one GeomVertexData associated with any particular Geom. You can reset the Geom to use a different GeomVertexData later, if you like, by calling geom.setVertexData().

The GeomNode constructor requires a name, which is the name of the node and will be visible in the scene graph. It can be any name you like that means something to you.

In the above example, we have created only one Geom, and added only one GeomPrimitive to the Geom. This is the most common case when you are creating geometry at runtime, although in fact a GeomNode may include multiple Geoms, and each Geom may include multiple GeomPrimitives. (However, all of the primitives added to a Geom must have the same fundamental primitive type: triangles, lines, or points. You can add GeomTriangles and GeomTristrips to the same Geom, or you can add GeomLines and GeomLinestrips, but if you have GeomTriangles and GeomLinestrips, you must use two different Geoms.)

It is important that the range of vertex index numbers used by your GeomPrimitives is consistent with the number of vertices in your GeomVertexData (for instance, if you have 100 vertices in your GeomVertexData, your GeomPrimitives must only reference vertices numbered 0 through 99). If this is not the case, you will get an exception when you call addPrimitive().

Panda3D Manual: Other Vertex and Model Manipulation

This section shows some other ways to manipulate low-level geometry and vertex information, including dynamically creating other kinds of Panda3D objects like Textures.

Panda3D Manual: Reading existing geometry data

You can fairly easily extract and examine or operate on the vertices for an existing model, although you should be aware that the *order* in which the vertices appear in a model is undefined. There is no correlation between the order in which vertices are listed in an egg file, and the order in which they will appear in the resulting loaded model. Panda may rearrange the vertices, or even add or remove vertices, as needed to optimize the model for rendering performance. Even from one session to the next, the vertices might come out in a different order.

This does make certain kinds of vertex operations difficult; if you plan to write code that expects to encounter the vertices of a model in a particular order, we recommend you build up those vertices yourself using a GeomVertexWriter (as described in Creating and filling a GeomVertexData), so that you have explicit control over the vertex order.

However, if you have no need to operate on the vertices in any particular order, or if you just want to casually browse the vertices in a model, feel free to use the following instructions to read the data.

When you load a model, you have a handle to the root node of the model, which is usually a **ModelRoot** node. The geometry itself will be stored in a series of **GeomNodes**, which will be children of the root node. In order to examine the vertex data, you must visit the GeomNodes in the model. One way to do this is to walk through all the GeomNodes like this:

```
geomNodeCollection = model.findAllMatches('**/+GeomNode')
for nodePath in geomNodeCollection.asList():
  geomNode = nodePath.node()
  processGeomNode(geomNode)
```

Once you have a particular GeomNode, you must walk through the list of **Geoms** stored on that node. Each Geom also has an associated RenderState, which controls the visible appearance of that Geom (e.g. texture, backfacing, etc.).

```
def processGeomNode(geomNode):
  for i in range(geomNode.getNumGeoms()):
    geom = geomNode.getGeom(i)
    state = geomNode.getGeomState(i)
    print geom
    print state
    processGeom(geom)
```

Note that geomNode.getGeom() is only appropriate if you will be reading, but not modifying, the data. If you intend to modify the geom data in any way (including any nested data like vertices or primitives), you should use geomNode.modifyGeom() instead.

Each Geom has an associated **GeomVertexData**, and one or more **GeomPrimitives**. Some GeomVertexData objects may be shared by more than one Geom, especially if you have used flattenStrong() to optimize a model.

```
def processGeom(geom):
  vdata = geom.getVertexData()
  print vdata
  processVertexData(vdata)
  for i in range(geom.getNumPrimitives()):
    prim = geom.getPrimitive(i)
    print prim
    processPrimitive(prim, vdata)
```

As above, getVertexData() is only appropriate if you will only be reading, but not modifying, the vertex data. Similarly, getPrimitive() is appropriate only if you will not be modifying the primitive index array. If you intend to modify either one, use modifyVertexData() or modifyPrimitive(), respectively.

You can use the **GeomVertexReader** class to examine the vertex data. You should create a GeomVertexReader for each column of the data you intend to read. It is up to you to ensure that a given column exists in the vertex data before you attempt to read it (you can use vdata. hasColumn() to test this).

```
def processVertexData(vdata):
  vertex = GeomVertexReader(vdata, 'vertex')
  texcoord = GeomVertexReader(vdata, 'texcoord')
  while not vertex.isAtEnd():
    v = vertex.getData3f()
    t = texcoord.getData2f()
    print "v = %s, t = %s" % (repr(v), repr(t))
```

Each GeomPrimitive may be any of a handful of different classes, according to the primitive type it is; but all GeomPrimitive classes have the same common interface to walk through the list of vertices referenced by the primitives stored within the class.

You can use the setRow() method of GeomVertexReader to set the reader to a particular vertex. This affects the next call to getData(). In this way, you can extract the vertex data for the vertices in the order that the primitive references them (instead of in order from the beginning to the end of the vertex table, as above).

```
def processPrimitive(prim, vdata):
  vertex = GeomVertexReader(vdata, 'vertex')

  prim = prim.decompose()

  for p in range(prim.getNumPrimitives()):
    s = prim.getPrimitiveStart(p)
    e = prim.getPrimitiveEnd(p)
    for i in range(s, e):
      vi = prim.getVertex(i)
      vertex.setRow(vi)
      v = vertex.getData3f()
      print "prim %s has vertex %s: %s" % (p, vi, repr(v))
    print
```

You may find the call to prim.decompose() useful (as shown in the above example). This call automatically decomposes higher-order primitive types, like GeomTristrips and GeomTrifans, into the equivalent component primitive types, like GeomTriangles; but when called on a GeomTriangles, it returns the GeomTriangles object unchanged. Similarly, GeomLinestrips will be decomposed into GeomLines. This way you can write code that doesn't have to know anything about GeomTristrips and GeomTrifans, which are fairly complex; it can assume it will only get the much simpler GeomTriangles (or, in the case of lines or points, GeomLines and GeomPoints, respectively).

<<prev top next>> Search

Panda3D Manual: Modifying existing geometry data

If you want to load a model and operate on its vertices, you can walk through the vertices as shown in the previous section, but you should substitute modifyGeom(), modifyVertexData(), and modifyPrimitive() for getGeom(), getVertexData(), and getPrimitive(), respectively. These calls ensure that, in case the data happens to be shared between multiple different GeomNodes, you will get your own unique copy to modify, without inadvertently affecting other nodes.

If you want to modify the vertex data, you have two choices. The simplest option is to create a new GeomVertexData and fill it up with your new vertex data (as described in Creating and filling a GeomVertexData), and then assigning this data to the geom with the call geom.setVertexData(). You must ensure that you add enough vertices to the new GeomVertexData to satisfy the GeomPrimitives that reference it.

Your second choice is to modify the vertex data in-place, by operating on the existing vertices. You can do this with a GeomVertexWriter. For instance, if you want to copy the (X, Y) position of each vertex to its (U, V) texture coordinate, you could do something like this:

```
texcoord = GeomVertexWriter(vdata, 'texcoord')
vertex = GeomVertexReader(vdata, 'vertex')

while not vertex.isAtEnd():
  v = vertex.getData3f()
  texcoord.setData2f(v[0], v[1])
```

> **Important!** When you are simultaneously reading from and writing to the same GeomVertexData object, you should create all of the GeomVertexWriters you need before you create any GeomVertexReader. This is because of Panda's internal referencing-counting mechanism; creating a GeomVertexWriter may automatically (and transparently) force a copy of the data in the GeomVertexData, which could invalidate any GeomVertexReaders you have already created.

Writing to a column with a GeomVertexWriter does require that the GeomVertexData's format already has the appropriate columns to handle the data you are writing (in the above example, for instance, the format must already have a 'texcoord' column, or the above code will fail). Furthermore, the columns must have the appropriate format. For instance, if you wanted to upgrade a model's texture coordinates from 2-D texture coordinates to 3-D texture coordinates, simply calling texcoord.setData3f(u, v, w) wouldn't change the fact that the existing texcoord column is a 2-component format; you would just be trying to stuff a 3-component value into a 2-component column.

If you want to add a new column to a GeomVertexData, or modify the format of an existing column, you will have to create a new GeomVertexFormat that includes the new column (see Defining your own GeomVertexFormat), and then change the format on the GeomVertexData via vdata.setFormat(format). This call will internally adjust all of the data to match the new

format. (Because of this internal adjustment, it is important to do this before you create the first GeomVertexWriter or GeomVertexReader.)

<<prev top next>> Search

Panda3D Manual: More about GeomVertexReader, GeomVertexWriter, and GeomVertexRewriter

The classes **GeomVertexReader** and **GeomVertexWriter** together represent the core interface for reading and writing the numeric data stored within a GeomVertexData object.

These two classes work similarly. Both are designed to get a temporary pointer to the data for a particular column when they are constructed, and they increment that pointer as you walk through the vertices. Initially, they start at row number 0 (the first vertex in the table), and after each setData/getData operation, they automatically increment to the next row (the next vertex).

You construct a GeomVertexReader or GeomVertexWriter with a pointer to the GeomVertexData object you are operating on, and the name of the column you wish to process, e.g.:

```
color = GeomVertexReader(vdata, 'color')
```

Because the GeomVertexReader and GeomVertexWriter classes only store a temporary pointer, which might become invalid between frames or even between different tasks within a frame, these objects should not be stored in any persistent object. Instead, they are designed to be temporary objects that are constructed locally, used immediately to iterate through a list of vertices, and then released. If you need to keep a persistent iterator for your vertex data, to be used over a long period of time (e.g. over several frames), then you should store just the GeomVertexData pointer (along with the current vertex index number if you require this), and construct a temporary GeomVertexReader/Writer each time you need to access it.

The following methods are available to read and write data in a column:

GeomVertexReader	GeomVertexWriter
x = getData1f()	setData1f(x) addData1f(x)
v2 = getData2f()	setData2f(x, y) setData2f(v2) addData2f(x, y) addData2f(v2)
v3 = getData3f()	setData3f(x, y, z) setData3f(v3) addData3f(x, y, z) addData3f(v3)
v4 = getData4f()	setData4f(x, y, z, w) setData4f(v4) addData4f(x, y, z, w) addData4f(v4)

x = getData1i()	setData1i(x) addData1i(x)
	setData2i(x, y) addData2i(x, y)
	setData3i(x, y, z) addData3i(x, y, z)
	setData4i(x, y, z, w) addData4i(x, y, z, w)

Each of the getData family of functions supported by GeomVertexReader returns the value of the data in the current column, converted to the requested type. The 'f' suffix indicates a floating-point value, while 'i' indicates an integer value; the digit indicates the number of components you expect to receive.

For instance, getData2f() always returns a VBase2, regardless of the type of data actually stored in the column. If the column contains a 2-component value such as a 2-D texture coordinate, then the returned value will represent the (U, V) value in that column. However, if the column type does not match the requested type, a conversion is quietly made; for instance, if you call getData2f() but the column actually contains a 3-D texture coordinate, the third component will be omitted from the return value, which will still be a VBase2.

Similarly, the setData and addData family of functions supported by GeomVertexWriter accept a value in the indicated format, and convert it to whatever format is required by the column. So if you call setData3f(), and the column has three components, you will set all three components with the x, y, z parameters of setData3f(); but if the column has only two components, only the x, y parameters will be used to set those two components, and the third parameter will be ignored.

Certain kinds of numeric conversions are performed automatically, according to the column's designated contents. For instance, if you store a floating-point value into an integer column, the fractional part of the value is usually truncated. However, if the column contents indicates that it represents a color value, then the floating-point value is automatically scaled from the range 0.0 .. 1.0 into the full numeric range of the column's integer value. This allows you to store color components in the range 0.0 .. 1.0, and get the expected result (that is, the value is scaled into the range 0 .. 255). A similar conversion happens when data is read.

There are no getData2i, 3i, or 4i methods available, simply because Panda does not currently define a multi-component integer value that can be returned to Python. Since most multi-component column types are floating-point, or can be expressed as floating-point, this is not generally a limitation.

Each GeomVertexReader keeps track of the current read row, which is initially 0; the current value can be retrieved by getReadRow(). Each call to a getData function returns the value of the column at the current read row, and then increments the current read row. It is an error to call getData when the read row has reached the end of the data, but you can call isAtEnd(), which returns true when the reader has reached the end. Thus, you can iterate through all the rows of a vertex table by repeatedly calling getData until isAtEnd() returns true.

Similarly, each GeomVertexWriter keeps track of the current write row, which is initially 0, and can be retrieved by getWriteRow(). Each call to setData or addData stores the given value in

the current write row, and then increments the current write row. It is an error to call setData when the write row has reached the end of the data; but as with the GeomVertexReader, you can call isAtEnd() to determine when you have reached the end of the data.

The addData family of functions work exactly like the setData functions, except that addData *can* be called when the GeomVertexWriter has reached its end. In this case, addData will add a new row to the table, and then fill in the specified data in that row (and then increment the current write row). If addData is called when the current write row already exists, it behaves exactly the same as setData.

With either GeomVertexReader or GeomVertexWriter, you can set the current read or write row at any time with the call setRow(). This sets the current read row (GeomVertexReader) or current write row (GeomVertexWriter) to the indicated value; the next call to getData or setData/addData will then operate on the specified row, and increment from there.

GeomVertexRewriter

The GeomVertexRewriter class exists as a convenience for code that needs to alternately read and write the data on a column. GeomVertexRewriter multiply inherits from GeomVertexReader and GeomVertexWriter, so it supports the getData family of functions, as well as the setData and addData family of functions. It also has both a current read row and a current write row, which might be different.

Normally, you would use a GeomVertexRewriter to walk through the list of vertices from the beginning to end, reading and writing as it goes. For instance, to set all of the Z components of a piece of geometry to 0.0, while preserving the X and Y components, you might write a loop such as:

```
vertex = GeomVertexRewriter(vdata, 'vertex')
while not vertex.isAtEnd():
  v = vertex.getData3f()
  vertex.setData3f(v[0], v[1], 0.0)
```

Note that this example code calls getData3f() and setData3f() exactly once through each iteration, which increments the current read row and current write row, respectively; so the current read row and current write row are kept in sync with each other.

> **Important!** When you are simultaneously reading from and writing to the same GeomVertexData object, you should create all of the GeomVertexWriters and GeomVertexRewriters you need before you create any GeomVertexReader. This is because of Panda's internal referencing-counting mechanism; creating a GeomVertexWriter may automatically (and transparently) force a copy of the data in the GeomVertexData, which could invalidate any GeomVertexReaders you have already created.

Panda3D Manual: Creating New Textures from Scratch

Creating Texture

The PNMImage Class

This class is how Panda3D handles regular images (.gif, .jpg, and the like). This class allows you to manipulate individual pixels of the image. You can load existing images using the function `read(fileName)` where `filename` is the path to the image file (in Panda Filename Syntax) wrapped in a `Filename` object. Or, you can create a brand new image from scratch, by passing the x, y size to the constructor.

```
myImage=PNMImage()
myImage.read(Filename("testImg.
gif"))
myEmptyImage = PNMImage(256, 256)
```

You can get the size of the image you have read using the `getXSize()` and `getYSize()` functions. Although you cannot change the size of an image directly, you can rescale an image by filtering it into a larger or smaller PNMImage:

```
fullSize=PNMImage(Filename("testImg.
gif"))
reduced=PNMImage(256, 256)
reduced.gaussianFilterFrom(1.0, fullSize)
```

You can get individual RGB values using the `getRed(x,y), getGreen(x,y), getBlue(x,y)` or `getRedVal(x,y), getGreenVal(x,y), getBlueVal(x,y)` where x and y tell what pixel to look at (lower-left is 0,0 upper right is `getXSize()-1, getYSize()-1` The difference between these functions is that the `get*Val` functions return a number between 0 and 1 while the `get*` functions retrun their value as an integer. For example, if your image uses 8-bit color calling `getGreenVal` on a green pixel will return 255 and calling `getGreen` will return 1. You can also get all the RGB information at the sametime using `getXel(x,y)` and `getXelVal(x,y)` which return a 3 component vector with red in the x, green in the y, and blue in the z.

```
 #the pixel at 0,0 is red and we're using 8-bit
color
myImage.getRedVel(0,0) #returns 255
myImage.getRed(0,0)#returns 1

colors=myImage.getXelVal(0,0) #returns (255,0,0)
colorVal=myImage.getXel(0,0) #returns (1,0,0)
```

The functions for setting pixel information are `setRed(x,y,value)`, `setGreen(x,y, value)`, `setBlue(x,y, value)` or `setRedVal(x,y,value)`, `setGreenVal(x,y, value)`, `setBlueVal(x,y, value)`. Theres still the same dichotomy as above when it comes to regular sets and using setvals. You can also use `setXel(x,y,colorVec)` and `setXelVal(x,y, colorVec)`. You can also fill an image with a color by using `fill(r,g,b)` and `fillVal(r,g,b)`.

```
myImage.setGreenVal(0,0, 255) # if pixel (0,0) was red before, now it is
yellow (255,255,0)
myImage.setBlue(0,0,1) #pixel (0,0) is now white

gray=Vec3(0.5,0.5,0.5)

#both of these set the origin to gray
myImage.setXelVal(0,0,gray*255)
myImage.setXel(0,0,gray)

#makes every pixel red
myImage.fillVal(255,0,0)
#makes every pixel green
myImage.fill(0,1,0)
```

There are also gets and sets for the alpha channel using the same interface as above. However, if you use them on an image that doesn't have an alpha channel you will cause a crash. To see if an image has an alpha channel use `hasAlpha()` which returns True if there is an alpha channel and False otherwise. You can add an alpha channel using `addAlpha()`. You can also remove it using `removeAlpha()`.

You can also make an image grayscale by using `makeGrayscale()`. You can now use sets and gets for Gray too. Using `getGray*` on a color image just returns the value in the blue channel. If you want to get the grayscale value of a pixel regardless of whether the image is a grayscale or a color image, you can use `getBright(x,y)`, which works equally well on color or on grayscale images. If you want to weight the colors use `getBright(x,y, r,g,b)` where r,g,b are the weights for their respective channels.

There are several other useful functions in the class this the API Reference for more information.

Getting the Image of a Texture

The Panda `Texture` class does not allow for pixel manipulation. However the `PNMImage` class below does. Therefore, if you want to change the image in a `Texture` object you must call its `store(myImage)` which saves the image of the texture into `myImage`.

```
myImage=PNMImage()
myTexture=loader.loadTexture("myTex.jpg")

#after this call, myImage now holds the same image as the
texture
myTexture.store(myImage)
```

Loading a PNMImage Into a Texture

Once you have changed all the data in the image you can now load it into a texture using the `Texture` objects `load(myImage)` function, where `myImage` is the PNMImage to make the texture from.

```
 #assume we already have myImage which is our modified
PNMImage
myTexture=Texture()

#This texture now represents myImage
myTexture.load(myImage)
```

Remember however, that most graphics cards require that the dimensions of texture have to be a power of two. `PNMImage` does not have this restriction and Panda will not automatically scale the image when you put it into a texture.

Panda3D Manual: Writing 3D Models out to Disk

Panda has two native file formats for models.

Egg files (with the extension `.egg`) are written in an ASCII human readable format. The egg format is designed to be easy to read and modify if necessary, and easy to write a convert into from another third-party format. Also, the egg format is intended to be backward-compatible from all future versions of Panda3D, so if you have an egg file that Panda can load now, it should always be able to load that file. (Well, we can't really make guarantees, but this is what we shoot for.) See Parsing and Generating Egg Files for more information about the egg format.

BAM Files

Because of the way the egg syntax is designed, an egg file might be very large, sometimes many times larger than the file it was converted from. It can also sometimes take several seconds for Panda to load a large egg file.

Bam files (with the extension `.bam`), on the other hand, are binary files that are closely tied to a particular version of Panda3D. The bam format is designed to be as similar as possible to the actual Panda data structures, so that a bam file is relatively small and can be loaded very quickly. However, you should not consider the bam file format to be a good long-term storage format for your models, since a future version of Panda3D may not be able to load bam files from older versions.

You can always convert egg files to bam files using the program `egg2bam`. For many simple models, it is also possible to convert back again with the program `bam2egg`, but you should not rely on this, since it does not convert advanced features like animation; and some structure of the original egg file may be lost in the conversion.

You can load files of these formats, as well as any other supported format, using the loader.loadModel interface. Any file types other than `.bam` or `.egg` will be automatically converted at runtime, exactly as if you had run the appropriate command-line conversion tool first.

The Bam Interface

The easiest way to save geometry is to use to call `writeBamFile(filename)` from the NodePath that contains your geometry.

```
myPanda=loader.loadModel("panda")
...
#do some fancy calculations on the normals, or texture coordinates that you dont
#want to do at runtime
...
#Save your new custom Panda
myPanda.writeBamFile("customPanda.bam")
```

The Egg Interface

One easy way to create `.egg` file for geometry that has already been made is to create a `.bam` file and use bam2egg. However, you will often want to use the egg interface to create geometry in the first place; this is usually the easiest way to create geometry in Panda3D.

The complete documentation for using the egg interfaces has yet to be written, but the egg library is really quite simple to use. The basic idea is that you create an EggData, and an EggVertexPool to hold your vertices; and then you can create a series of EggVertex and EggPolygon objects. If you want to create some structure in your egg file, you can create one or more EggGroups to separate the polygons into different groups. Here is an example:

```
def makeWedge(angleDegrees = 360, numSteps = 16):
    data = EggData()

    vp = EggVertexPool('fan')
    data.addChild(vp)

    poly = EggPolygon()
    data.addChild(poly)

    v = EggVertex()
    v.setPos(Point3D(0, 0, 0))
    poly.addVertex(vp.addVertex(v))

    angleRadians = deg2Rad(angleDegrees)

    for i in range(numSteps + 1):
        a = angleRadians * i / numSteps
        y = math.sin(a)
        x = math.cos(a)

        v = EggVertex()
        v.setPos(Point3D(x, 0, y))
        poly.addVertex(vp.addVertex(v))

    # To write the egg file to disk, use this:
    data.writeEgg(Filename("wedge.egg"))

    # To load the egg file and render it immediately, use this:
    node = loadEggData(data)
    return NodePath(node)
```

See the generated API documentation for more complete information about the egg library.

<<prev top next>> Search

Panda3D Manual: Generating Heightfield Terrain

WRITE ME: this section will tell how to implement a patch-based LOD heightfield terrain.

Panda3D Manual: Panda Rendering Process

The rendering process in Panda is comprised by four classes and their interactions: GraphicsPipe, GraphicsEngine, GraphicsStateGaurdian, and GraphicsOutput. The following sections will explain the purpose of each of these classes in detail.

Note that the following interfaces are for the advanced user only. If you are writing a simple application that only needs to open a window and perform basic 3-D rendering, there is no need to use any of these interfaces, as the appropriate calls to open a default window are made automatically when you `import direct.directbase.DirectStart` at the start of your application.

Panda3D Manual: The Graphics Pipe

The `GraphicsPipe` class is Panda3D's interface to the available 3-D API's, for instance OpenGL or DirectX. In order to create a window that renders using a particular API, you must have a GraphicsPipe for that API.

Normally, there is one default graphics pipe created for you automatically when you import DirectStart, accessible as `base.pipe`. For most applications, there is no need to create any additional graphics pipes.

There are two Config.prc variables that determine the graphics pipe or pipes that will be available to an application:

load-display: this variable specifies the first choice for the graphics pipe. It names the type of GraphicsPipe that should be attempted first, e.g. pandagl or pandadx8. If for some reason a GraphicsPipe of this type cannot be created, for instance because of lack of driver support, then Panda3D will fall back to the next variable:

aux-display: this variable can be repeated multiple times, and should list all of the available GraphicsPipe implementations. If Panda3D is unable to open a pipe of the type named by load-display, then it will walk through the list of pipes named by aux-display, in the order they appear in the Config.prc file, and try them one at a time until one is successfully opened.

Note that the name specified to each of the above variables, e.g. pandagl, actually names a Windows DLL or Unix shared-library file. Panda3D will put "lib" in front of the name and ".dll" or ".so" (according to the operating system) after the name, and then attempts to import that library. This means that "load-display pandagl" really means to try to import the file "libpandagl.dll". The various display DLL's are written so that when they are successfully imported, they will register support for the kind of GraphicsPipe they implement.

You can create additional graphics pipes, for instance to provide an in-game interface to switch between OpenGL and DirectX rendering. The easiest way to do this is to call `base.makeAllPipes()`. Then you can walk through the list of GraphicsPipes in `base.pipeList` to see all of the available GraphicsPipes available in particular environment.

When you walk through the GraphicsPipes in base.pipeList, you can call the following interface methods on each one:

`pipe.isValid()`	Returns True if the pipe is available for rendering, False if it can't be used.
`pipe.getDisplayWidth()`	Returns the width of the desktop, or the maximum width of any buffer for an offscreen-only GraphicsPipe.
`pipe.getDisplayHeight()`	Returns the height of the desktop, or the maximum height of any buffer for an offscreen-only GraphicsPipe.
`pipe.getInterfaceName()`	Returns the name of the API that this GraphicsPipe impements, e.g. "OpenGL" or "DirectX8".

`pipe.getType()`	Returns a unique TypeHandle object for each kind of pipe.

Panda3D Manual: The Graphics Engine

The graphics engine is the heart of the rendering process. The `GraphicsEngine` class is ultimately responsible for all of the drawing and culling operations per frame.

Normally, there is no need to create a GraphicsEngine, as Panda3D will create one for you at startup. This default GraphicsEngine is stored in `base.graphicsEngine`.

Note also that the following interfaces are strictly for the advanced user. Normally, if you want to create a new window or an offscreen buffer for rendering, you would just use the `base.openWindow()` or `window.makeTextureBuffer()` interfaces, which handle all of the details for you automatically.

However, please continue reading if you want to understand in detail how Panda manages windows and buffers, or if you have special needs that are not addressed by the above convenience methods.

Rendering a frame

There is one key interface to rendering each frame of the graphics simulation:

```
</td>
base.graphicsEngine.renderFrame
()
```

This method causes all open GraphicsWindows and GraphicsBuffers to render their contents for the current frame.

In order for Panda3D to render anything, this method must be called once per frame. Normally, this is done automatically by the task "igloop", which is created when you start Panda.

Using a GraphicsEngine to create windows and buffers

In order to render in Panda3D, you need a **GraphicsStateGuardian**, and either a **GraphicsWindow** (for rendering into a window) or a **GraphicsBuffer** (for rendering offscreen). You cannot create or destroy these objects directly; instead, you must use interfaces on the GraphicsEngine to create them.

Before you can create either of the above, you need to have a GraphicsPipe, which specifies the particular graphics API you want to use (e.g. OpenGL or DirectX). The default GraphicsPipe specified in your Config.prc file has already been created at startup, and can be accessed by `base.pipe`.

Now that you have a GraphicsPipe and a GraphicsEngine, you can create a GraphicsStateGuardian object. This object corresponds to a single graphics context on the graphics API, e.g. a single OpenGL context. (The context owns all of the OpenGL or DirectX objects like display lists, vertex buffers, and texture objects.) You need to have at least one GraphicsStateGuardian before you can create a GraphicsWindow:

```
</td>
myGsg=base.graphicsEngine.makeGsg(base.
pipe)
```

Now that you have a GraphicsStateGuardian, you can use it to create an onscreen GraphicsWindow or an offscreen GraphicsBuffer:

```
</td>
base.graphicsEngine.makeWindow(gsg, name, sort)
base.graphicsEngine.makeBuffer(gsg, name, sort, xSize, ySize,
wantTexture)
```

`gsg` is the GraphicsStateGuardian, `name` is an arbitrary name you want to assign to the window/buffer, and `sort` is an integer that determines the order in which the windows/buffers will be rendered.

The buffer specific arguments `xSize` and `ySize` decide the dimensions of the buffer, and `wantTexture` should be set to `True` if you want to retrieve a texture from this buffer later on.

You can also use `graphicsEngine.makeParasite(host,name,sort,xSize,ySize)`, where host is a GraphicsOutput object. It creates a buffer but it does not allocate room for itself. Instead it renders to the framebuffer of `host`. It effectively has `wantTexture` set to `True` so you can retrieve a texture from it later on.

See The GraphicsOutput class and Graphics Buffers and Windows for more information.

```
myWindow=base.graphicsEngine.makeWindow(myGsg, "Hello World", 0)
myBuffer=base.graphicsEngine.makeBuffer(myGsg, "Hi World", 0, 800,600, True)
myParasite=base.graphicsEngine.makeBuffer(myBuffer,"Im a leech", 0, 800,
600)
```

Note: if you want the buffers to be visible add `show-buffers true` to your configuration file. This causes the buffers to be opened as windows instead, which is useful while debugging.

Sharing graphics contexts

It is possible to share the same GraphicsStateGuardian among multiple different GraphicsWindows and/or GraphicsBuffers; if you do this, then the graphics context will be used to render into each window one at a time. This is particularly useful if the different windows will be rendering many of the same objects, since then the same texture objects and vertex buffers can be shared between different windows.

It is also possible to use a different GraphicsStateGuardian for each different window. This means that if a particular texture is to be rendered in each window, it will have to be loaded into graphics memory twice, once in each context, which may be wasteful. However, there are times when this may be what you want to do, for instance if you have multiple graphics cards and you want to to render to both of them simultaneously. (Note that the actual support for simultaneously rendering to multiple graphics cards is currently unfinished in Panda at the time of this writing, but the API has been designed with this future path in mind.)

Closing windows

To close a specific window or buffer you use `removeWindow(window)`. To close all windows `removeAllWindows()`

```
base.graphicsEngine.removeWindow
(myWindow)
base.graphicsEngine.removeAllWindows()
```

More about GraphicsEngine

Here is some other useful functionality of the `GraphicsEngine` class.

`getNumWindows()`	Returns the number of windows and buffers that this `GraphicsEngine` object is managing.
`isEmpty()`	Returns `True` if this `GraphicsEngine` is not managing any windows or buffers.

See API for advanced functionality of `GraphicsEngine` and `GraphicsStateGuardian` class.

Panda3D Manual: The GraphicsOutput class

Buffers and windows, encapsulated in the GraphicsBuffer and GraphicsWindow classes are almost interchangable in Panda. In fact most operations in the GraphicEngine class are defined on and return GraphicOutput objects, the class that both GraphicsBuffer and GraphicsWindow inherit from. Therefore, we will discuss the properties of GraphicOutput objects first.

The first very important note is that none of these classes are not meant to be constructed directly, i.e.:

```
myOutput=GraphicsOutput
()
myWindow=GraphicsWindow
()
myBuffer=GraphicsBuffer
()
```

will not work. Refer to The Graphics Engine for how to create these objects. Furthermore, since GraphicsOutput is an abstract class, GraphicsWindow objects will be used in code examples.

All GraphicsOutput objects have getGsg(), getPipe(), and getName() which return respectively their GraphicsStateGuardian, GraphicsPipe, and name. You can also get the width and length using getXSize() and getYSize().

```
from pandac.PandaModules import GraphicsWindow

#assume we already have a window setup and in
myWindow
myWindowGsg=myWindow.getGsg()
myWindowPipe=myWindow.getPipe()
myWindowName=myWindow.getName()

myWindowWidth=myWindow.getXSize()
myWindowLength=myWindow.getYSize()
```

You can also save a screenshot from any GraphicsOutput by using saveScreenShot(fileName), where fileName is the name of the picture(the format of the picture is specified by the extension of filename). Returns True upon succes and False otherwise. The picture is saved in the directory of the script you are running.

```
from pandac.PandaModules import Filename
myWindow.saveScreenShot(Filename('hello.
bmp'))
```

This naturally flows into rendering into a texture. We'll start with copying a scene. If you want to get a texture that simply copies whats in it GraphicsOutput object, you must first make a call to

setupCopyTexture(). You can then get the texture by using getTexture(). You can now apply the texture to a NodePath as you would a texture loaded from memory. Thanks to the magic of pointers, the texture automatically updates itself if the contents of its GraphicsOutput change. If you do not want this behaviour you should use detachTexture() when you no longer want the textute to be updated. However, since the first frame is always blank, the best way to use detachTexture() is in a do-later task or event.

</td>

```
myWindow.setupCopyTexture()
myTexture=myWindow.getTexture()

#assume myModel is already setup
myModel.setTexture(myTexture)

#and if you want to stop the texture from updating
itself
def stopUpdating():
   global myWindow
   myWindow.detachTexture()
taskMgr.doMethodLater(1,stopUpdating,'stops updating')
```

While this is helpful, you may want to render an entirely new scene into a GraphicsOutput and then place it on screen (i.e. you have a televsion in your main scene and want to generate the show on the spot). The first thing you do is create a GraphicsOutput to hold the scene. You do this by calling makeTextureBuffer. It makes a GraphicsOutput specifically for rendering a scene and then retrieving it by getTexture().

```
makeTextureBuffer(name, xSize,
ySize)
```

The arguments name, xSize, and ySize mean the same things they do for makeWindow and makeBuffer.

You then have to create a new camera for the new scene, using

</td>

```
base.makeCamera(win, sort=0, scene=None, displayRegion=(0,1,0,1), aspectRatio=None,
camName='cam')
```

Here's a break down of what the parameters mean:

win	The GraphicsOutput object that you want to make the camera for
sort	The sort value of the camera. Decides the order in which DisplayRegions in the same window are drawn. See API for more information.
scene	Due to deprecation of other functions this parameter does not affect anything.

displayRegion	The area of the new GraphicsOutput that you want to cover in the form (left start point, right end point, bottom start point, top end point). (0,0) represent the bottom left of the screen and (1,1) represents the top right. Therefore (0,1,0,1) represents the entire area. Arguments must be between 0 and 1.
aspectRatio	The aspectRatio of the GraphicsOutput. When this is left to None makeCamera uses the aspectRatio of the default window.
camName	The name of the node that represents this camera in the scene graph

Cameras render whatever is connected to their ancestors in the scene graph. Therefore if you want a truly independent scene you have to start a new scene graph. Create a dummy NodePath and now reparentTo the new camera to this node. Now you can treat the new scene and the new camera like you would render and your scene gets drawn to your GraphicsOuptut.

However, any state changes you make to the NodePath camera will no longer affect your new camera. Also, since the standard mouse controls work on the camera NodePath, these will not work either. You can alternatively use the Camera class method setScene(scenePath), where scenePath is the top of the scene graph you want to draw. This preserves the standard heirarchy stated in Camera Conrol.

```
#I use a GraphicsBuffer only because this is a process you probably want the user to see

myBuffer=myWindow.makeTextureBuffer("Another Scene", 800,600)

#You must pass a string to the NodePath constructor or attempts to set it as
#   a parent will remove the child from the graph
myNewScene=NodePath("myRender")
myNewCamera=base.makeCamera(myBuffer)
myNewCamera.reparentTo(myNewScene)
#or myNewCamera.node().setScene(myNewScene)

#You can now get a texture that represents anything you do in this new scene
#   (that is still automatically updated)
myTexture=myBuffer.getTexture()
```

Panda3D Manual: Graphics Buffers and Windows

We'll now describe in detail what functions are specfic to buffers and windows in Panda.

GraphicsBuffer and ParasiteBuffer

Use these if you want to do off-screen rendering. You must pass `True` when you create it if you want to get a texture from it. Otherwise, there is no difference in functionality than a GraphicsOutput.

The only difference between a `GraphicsBuffer` and a `ParasiteBuffer` is that a `ParasiteBuffer` does not create it own framebuffer space. To create a `ParasiteBuffer` you call `makeParasite()` from the graphics engine.

```
makeParasite(host, name,sort, xSize,
ySize)
```

The arguments `name`, `sort`, `xSize`, and `ySize` mean the same things they mean for makeWindow and makeBuffer. The new argument `host` is the GraphicsOutput object whose space in memory it will use. Any rendering done to the parasite is done to the same space in memory as its host. The function makeTextureBuffer sometimes returns a `ParasiteBuffer` for space saving reasons. It is also useful for API that dont support offscreen rendering.

`ParasiteBuffer` objects are automatically setup for calls to `getTexture()` since their contents get cleared when `host` draws itself.

GraphicsWindows

Unlike `GraphicsBuffer` objects `GraphicsWindow` objects have a lot more functionality than `GraphicsOutput` objects.

The most basic of these functions is `hasKeyboard()` and `hasPointer()` which returns whether or not this window has the focus of keyboard and pointer respectively. Any calls to keyboard or pointer functions when you do not have control of them generates an error.

You can get the number of input devices for this window by using `getNumInputDevices()`. In the abscence of a joystick, etc. there usually only one input device, the 'keyboard/mouse' device. If the API you are using supports it, you can move a mouse to a certain place in the window by using `movePointer(device, x,y)` where `device` is the name of the device that holds the mouse (most probabaly 'keyboard/mouse') and `x` and `y` is the screen position where you mwant to move the pointer. Returns `True` if it was succesful, `False` otherwise.

You can also ask a window if it `isFullscreen()` and if it `isClosed()`. It is important to note that a window is not automatically opened after a call to `makeWindow` and is not automatically closed after a call to `closeWindow`.

In order to get the full set of properties for a given window you use the function `getProperties()`. This returns a `WindowProperties` object that holds all the information for the given screen. See API for the full functionality of the `WindowProperties` class.

If you want to change these properties use `getRequestedProperties()` and apply the proper `WindowProperties` functions.

To run panda3d in full screen, rather than a window, do the following:

```
wp = WindowProperties()
wp.setfullcreen(true)
base.win.requestProperties
(wp)
```

An alternate exists. Modify the fullscreen configuration variable before importing direct. directbase.directstart.

```
from pandac.PandaModules import loadPrcFileData
loadPrcFileData("", """fullscreen 1
win-size 1024 768""")

from direct.showbase.DirectObject import DirectObject # for event
handling
import direct.directbase.DirectStart
import sys

class World(DirectObject):
   def __init__(self):
      self.accept("escape",sys.exit)

w= World()
run()
```

If a requested change is not possible or invalid you can call `getRejectedProperties`. It returns a `WindowProperties` object that holds all the properties that could not be changed.

Windows can also send Events when the user changes a property of the window. You can get the name of this event by calling `getWindowEvent()`. Initially, all windows send the same event when changed. If you want to setup events for a certain window use `setWindowEvent(name)` where `name` is the name of the event you want sent when this window gets changed externally.

For more advanced functionality see `GraphicsWindow` in the API.

Panda3D Manual: Multi-Pass Rendering

Multi-Pass Rendering

Sometimes you may need to draw the same scene more than once per frame, each view looking different. This is where multi-pass rendering comes into play.

The easiest way to do implement multi-pass rendering is to use the method mentioned near the bottom in The GraphicsOutput class. You:

1) setup a GraphicsBuffer object
2) create a camera for it and
3) place the camera in the scene.

However, this method assumes you have two independent scene graphs. If you use this method to render the same scene graph, it is only useful for showiing the scene from a different camera view. To actually make the scenes have different RenderStates (i.e. one without lighting, one with lighting) you must also change how each Camera renders the scene.

Each Camera node has a function called `setInitialState(state)`. It makes every object in the scene get drawn as if the top node in its scene graph has `state` as its RenderState. This still means that attributes can be changed/overriden after the Camera has been put on a scene.

```
 #this makes everything drawn by the default camera use myNodePath's
RenderState
base.cam.setInitialState(myNodePath.getState())
```

You may, however, want more control over what RenderState gets assigned to each node in the scene. You can do this using the Camera class methods `setTagStateKey(key)` and `setTagState(value, state)`. For any NodePaths that you want to recieve special treatment you call `setTag(key, value)`(See Common State Changes). Now, anytime the Camera sees a NodePath with a tag named `key` the Camera assigns it whatever RenderState is associated with `value`.

```
 #Assume we have CgShaderAttrib instances toonShadingAttrib and
blurShadingAttrib
#and we have a Camera whose NodePath is myCamera

base.cam.node().setTagStateKey("Toon Shading")
base.cam.node().setTagState("True", RenderState.make(toonShadingAttrib))

myCamera.node().setTagStateKey("Blur Shading")
myCamera.node().setTagState("True", RenderState.make(blurShadingAttrib))

#this makes myNodePath and its children get toonShaded when rendered by the
default camera
myNodePath.setTag("Toon Shading", "True")
....
#now if you want myNodePath to be blurred when seen by myCamera its as easy as
adding a tag
myNodePath.setTag("Blur Shading", "True")
```

<<prev top next>> Search

Panda3D Manual: Render to Texture

Render to Texture Basics

In Panda3D, rendering to a texture consists of three basic steps:

- Create a hidden window (class GraphicsBuffer).
- Render into the hidden window.
- Transfer the contents of the hidden window into a texture.

When I say "transfer" the contents of a window into a texture, I don't necessarily mean "copy." There are other ways to transfer the contents of a window into a texture that may be faster. For example, if the OpenGL implementation supports the ARB_pbuffers extension, then the transfer might be achieved using wglBindTexImageARB. The Panda user does not need to worry about how the transfer is done. It is only important that you know that Panda will use the fastest means available to transfer the contents of the window into the texture.

To generalize that a bit, although render-to-texture is usually done with a hidden window (class GraphicsBuffer), it can also be done with a visible window (class GraphicsWindow). You can transfer the contents of any window, hidden or not, into a texture. That's potentially useful - for example, you can transfer the contents of the main window into a texture, which you can then use when rendering the next frame. This can be used to create accumulation-buffer-like effects without an accumulation buffer.

The Simple API: makeTextureBuffer

Here is a short snippet of code that creates a hidden window, creates a camera that renders into that window, and creates a scene graph for that camera:

```
mybuffer = base.win.makeTextureBuffer("My Buffer", 512,
512)
mytexture = mybuffer.getTexture()</br>
mybuffer.setSort(-100)</br>
mycamera = base.makeCamera(mybuffer)
myscene = NodePath("My Scene")
mycamera.node().setScene(myscene)
```

The `makeTextureBuffer` is the simple interface to the render-to-texture functionality. It creates a new hidden window (usually a GraphicsBuffer), creates a texture to render into, and connects the texture to the hidden window. The (512, 512) in the function call specifies the size of the hidden window and texture. Of course, you need to use a power-of-two size. The `getTexture` method retrieves the texture, which will be rendered into every frame.

The `setSort` method sets a window's sort order. This controls the order in which panda renders the various windows. The main window's sort order is zero. By setting the sort order

of mybuffer to a negative number, we ensure that mybuffer will be rendered first. That, in turn, ensures that mytexture will be ready to use by the time that the main window is rendered.

The new hidden window is not automatically connected to the scene graph. In this example, we create a separate scene graph rooted at myscene, create a camera to view that scene graph, and connect the camera to mybuffer.

The function `makeTextureBuffer` usually creates a GraphicsBuffer (hidden window), but if the video card is not powerful enough to create an offscreen window, it may not be able to do so. In that case, `makeTextureBuffer` will create a `parasiteBuffer` instead. A parasite buffer is primarily a trick to emulate a GraphicsBuffer on video cards that are less powerful. The trick is this: instead of rendering to an offscreen window and then transferring the data into a texture, panda renders into the *main* window and then copies the data into the texture. The limitations of this trick are self-evident. First, it garbles the contents of the main window. This is usually no big deal, since the main window is usually cleared and rendered from scratch every frame anyway. The other problem with this trick is that it fails if the main window is smaller than the desired texture. Since neither of these problems is common in practice, `makeTextureBuffer` will use parasite buffers transparently if GraphicsBuffers are not available.

There is a debugging mode in which `makeTextureBuffer` will create a *visible* window (class GraphicsWindow) instead of a hidden one (class GraphicsBuffer). To enable this debugging mode, set the boolean variable "show-buffers #t" in your panda configuration file.

The Advanced API: addRenderTexture

The simple API is convenient, but there are a few things it can not do. For instance, it can not:

- Copy the main window into a texture.
- Copy the Z-buffer into a depth texture.
- Copy the window into a texture, but not every frame.
- Limit or force the use of Parasite buffers.

If you need this level of control, you need to use a lower-level API.

I'll finish this section later. -Josh

Panda3D Manual: How to Control Render Order

How to Control Render Order

In most simple scenes, you can naively attach geometry to the scene graph and let Panda decide the order in which objects should be rendered. Generally, it will do a good enough job, but there are occasions in which it is necessary to step in and take control of the process.

To do this well, you need to understand the implications of render order. In a typical OpenGL- or DirectX-style Z-buffered system, the order in which primitives are sent to the graphics hardware is theoretically unimportant, but in practice there are many important reasons for rendering one object before another.

Firstly, state sorting is one important optimization. This means choosing to render things that have similar state (texture, color, etc.) all at the same time, to minimize the number of times the graphics hardware has to be told to change state in a particular frame. This sort of optimization is particularly important for very high-end graphics hardware, which achieves its advertised theoretical polygon throughput only in the absence of any state changes; for many such advanced cards, each state change request will completely flush the register cache and force a restart of the pipeline.

Secondly, some hardware has a different optimization requirement, and may benefit from drawing nearer things before farther things, so that the Z-buffer algorithm can effectively short-circuit some of the advanced shading features in the graphics card for pixels that would be obscured anyway. This sort of hardware will draw things fastest when the scene is sorted in order from the nearest object to the farthest object, or "front-to-back" ordering.

Finally, regardless of the rendering optimizations described above, a particular sorting order is required to render transparency properly (in the absence of the specialized transparency support that only a few graphics cards provide). Transparent and semitransparent objects are normally rendered by blending their semitransparent parts with what has already been drawn to the framebuffer, which means that it is important that everything that will appear behind a semitransparent object must have already been drawn before the semitransparent parts of the occluding object is drawn. This implies that all semitransparent objects must be drawn in order from farthest away to nearest, or in "back-to-front" ordering, and furthermore that the opaque objects should all be drawn before any of the semitransparent objects.

Panda achieves these sometimes conflicting sorting requirements through the use of bins.

Cull Bins

The CullBinManager is a global object that maintains a list of all of the cull bins in the world, and their properties. Initially, there are five default bins, and they will be rendered in the following order:

```
   Bin Name        Sort  Type
   --------------  ----  ----------------
```

```
   "background"     10   BT_fixed
   "opaque"         20   BT_state_sorted
   "transparent"    30   BT_back_to_front
   "fixed"          40   BT_fixed
   "unsorted"       50   BT_unsorted
```

When Panda traverses the scene graph each frame for rendering, it assigns each Geom it encounters into one of the bins defined in the CullBinManager. (The above lists only the default bins. Additional bins may be created as needed, using either the CullBinManager::add_bin() method, or the Config.prc "cull-bin" variable.)

You may assign a node or nodes to an explicit bin using the NodePath::set_bin() interface. set_bin() requires two parameters, the bin name and an integer sort parameter; the sort parameter is only meaningful if the bin type is BT_fixed (more on this below), but it must always be specified regardless.

If a node is not explicitly assigned to a particular bin, then Panda will assign it into either the "opaque" or the "transparent" bin, according to whether it has transparency enabled or not. (Note that the reverse is not true: explicitly assigning an object into the "transparent" bin does not automatically enable transparency for the object.)

When the entire scene has been traversed and all objects have been assigned to bins, then the bins are rendered in order according to their sort parameter. Within each bin, the contents are sorted according to the bin type.

If you want geometry that's in back of something to render in front of something that it logically shouldn't, add the following code to the model that you want in front:

```
model.setBin("fixed", 40)
model.setDepthTest(False)
model.setDepthWrite
(False)
```

The following bin types may be specified:

```
 BT_fixed

   Render all of the objects in the bin in a fixed order specified by
   the user.  This is according to the second parameter of the
   NodePath::set_bin() method; objects with a lower value are drawn
   first.

 BT_state_sorted

   Collects together objects that share similar state and renders
   them together, in an attempt to minimize state transitions in the
   scene.
```

BT_back_to_front

Sorts each Geom according to the center of its bounding volume, in linear distance from the camera plane, so that farther objects are drawn first. That is, in Panda's default right-handed Z-up coordinate system, objects with large positive Y are drawn before objects with smaller positive Y.

BT_front_to_back

The reverse of back_to_front, this sorts so that nearer objects are drawn first.

BT_unsorted

Objects are drawn in the order in which they appear in the scene graph, in a depth-first traversal from top to bottom and then from left to right.

<<prev top next>> Search

Panda3D Manual: Panda Utility Functions

Panda3D has a set of utilities that may be used to learn more about various objects and methods within an application. To access these utilities you need to import the PythonUtil module as follows.

```
from direct.showbase.PythonUtil import
*
```

The * can be replaced by any of the utility functions in that module.

To get a detailed listing of a class or an object's attributes and methods, use the pdir() command. pdir() prints the information out to the command console. pdir() can take many arguments for formatting the output but the easiest way to use it is to provide it a NodePath.

pdir() will list all of the functions of the class of NodePath including those of its base classes

```
pdir(NodePath)
e.g. pdir
(camera)
```

There are many other useful functions in the PythonUtil module. All of these are not necessarily Panda specific, but utility functions for python. There are random number generators, random number generator in a gaussian distribution curve, quadratic equation solver, various list functions, useful angle functions etc. A full list can be found in the API.

An alternative command to `pdir` is `inspect()`. This command will create a window with methods and attributes on one side, and the details of a selected attribute on the other. `inspect()` also displays the current values of a classâ€™ attributes. If these attributes are changing, you may have to click on a value to refresh it. To use inspect() you have to do the following:

```
from direct.tkpanels.inspector import
inspect
inspect(NodePath)
```

```
E.g. inspect
(camera)
```

While the directtools suite calls upon a number of tools, if the suite is disabled, the user may

activate certain panels of the suite. The `place()` command opens the object placer console. The `explore()` opens the scene graph explorer, which allows you to inspect the hierarchy of a NodePath. Finally, in order to change the color of a NodePath, the `rgbPanel()` command opens color panel.

```
camera.place()
render.explore
()
panda.rgbPanel
()
```

Useful DirectTool panels are explained in the Panda Tools section.

<<prev top next>> Search

Panda3D Manual: Particle Effects

Particle effects involve the use of several small images acting on the same set of forces. These particles are created, they move, and they die out. These systems are dynamic, and may be used for such effects as fireworks, bubbling cauldrons, and even swarms of balloons.

In essence, any particle effect needs three key parts: the renderer, the emitter, and the factory. The renderer translates the particle object into a visible object on the screen. The emitter assigns initial locations and velocity vectors for the particles. The factory generates particles and assigns their attributes. There are many different types of each part, and they each have their own parameters.

Creating your own particle effects using code alone may be difficult. A particle effect panel is available to ease through this process. This section will discuss using the particle panel and the large number of variables associated with particle effects.

Panda3D Manual: Using the Particle Panel

The particle panel must be used from the python command prompt. Open a command prompt and enter the folder you wish to run this in.

```
ppython
import direct.directbase.DirectStart
from direct.tkpanels import ParticlePanel
pp = ParticlePanel.ParticlePanel()
run()
```

Once the desired effect is achieved, save the file out. This, after some alterations, may be inserted into your current project code. First, copy and paste the code into your existing project code. Above the first line, add this line:

```
from direct.particles.ParticleEffect import ParticleEffect
f = ParticleEffect.ParticleEffect()
```

After that, replace any â€œselfâ€• variable with the variable â€œf.â€• Finally, add these lines to the end of your particle effect code:

```
t = Sequence(Func(f.start, render, render),)
t.start()
```

Also, to use any particle effects, they must be enabled through a base command.

```
base.enableParticles()
```

Panda3D Manual: Particle Effect Basic Parameters

Every particle effect needs at least eleven parameters. These govern the overall properties, such as the number of particles on the screen, the birth and death rates, and the renderer, emitter, and factory that are used.

Variable	Definition	Values
poolSize	Maximum number of simultaneous particles	[0, infinity)
birthRate	Seconds between particle births	(0, infinity)
litterSize	Number of particles created at each birth	[1, infinity)
litterSpread	Variation of litter size	[0, infinity)
localVelocityFlag	Whether or not velocities are absolute	Boolean
systemGrowsOlder	Whether or not the system has a lifespan	Boolean
systemLifespan	Age of the system in seconds	[0, infinity)
BaseParticleRenderer* renderer	Pointer to particle renderer	Renderer type
BaseParticleRenderer* emitter	Pointer to particle emitter	Emitter type
BaseParticleRenderer* factory	Pointer to particle factory	Factory type

The renderer, emitter, and factory types will be discussed in the next three sections.

Panda3D Manual: Particle Factories

There are two types of particle factories, Point and ZSpin. The particle panel shows a third, Oriented, but this factory does not currently work. The differences between these factories lie in the orientation and rotational abilities.

First, there are some common variables to the factories.

Variable	Definition	Values
lifespanBase	Average lifespan in seconds	[0, infinity)
lifespanSpread	Variation in lifespan	[0, infinity)
massBase	Average particle mass	[0, infinity)
massSpread	Variation in particle mass	[0, infinity)
terminalVelocityBase	Average particle terminal velocity	[0, infinity)
terminalVelocitySpread	Variation in terminal velocity	[0, infinity)

Point particle factories generate simple particles. They have no additional parameters.

ZSpin particle factories generate particles that spin around the Z axis, the vertical axis in Panda3D. They have some additional parameters.

Variable	Definition	Values
initialAngle	Starting angle in degrees	[0, 360]
initialAngleSpread	Spread of initial angle	[0, 360]
finalAngle	Final angle in degrees	[0, 360]
fnalAngleSpread	Spread of final angle	[0, 360]

Panda3D Manual: Particle Emitters

There are a large number of particle emitters, each categorized by the volume of space they represent. Additionally, all emitters have three modes: explicit, radiate, and custom. Explicit mode emits the particles in parallel in the same direction. Radiate mode emits particles away from a specific point. Custom mode emits particles with a velocity determined by the particular emitter.

All emitters have a number of common parameters.

Variable	Definition	Values
emissionType	Emission mode	ET_EXPLICIT, ET_RADIATE, ET_CUSTOM
explicitLaunchVector	Initial velocity in explicit mode	(x, y, z)
radiateOrigin	Point particles launch away from in radiate mode	(x, y, z)
amplitude	Launch velocity multiplier	(-infinity, infinity)
amplitudeSpeed	Spread for launch velocity multiplier	[0, infinity)

The following list contains the different types of emitters, their unique parameters, and the effect of the custom mode.

BoxEmitter

Variable	Definition	Values
minBound	Minimum point for box volume	(x, y, z)
maxBound	Maximum point for box volume	(x, y, z)

Custom mode generates particles with no initial velocity.

DiscEmitter

Variable	Definition	Values
radius	Radius of disc	[0, infinity)
outerAngle	Particle launch angle at edge of disc	[0, 360]
innterAngle	Particle launch angle at center of disc	[0, 360]
outerMagnitude	Launch velocity multiplier at edge of disc	(-infinity, infinity)
innerMagnitude	Launch velocity multiplier at center of disc	(-infinity, infinity)
cubicLerping	Whether or not magnitude/angle interpolation is cubic	Boolean

Custom mode uses the last five parameters. Particles emitted from areas on the inside use interpolated magnitudes and angles, either liner or cubic.

PointEmitter

Variable	Definition	Values
location	Location of outer point	(x, y, z)

Custom mode generates particles with no initial velocity.

RectangleEmitter

Variable	Definition	Values
minBound	2D point defining the rectangle	(x, z)
maxBound	2D point defining the rectangle	(x, z)

Custom mode generates particles with no initial velocity.

RingEmitter

Variable	Definition	Values
radius	Radius of disc	[0, infinity)
angle	Particle launch angle	[0, 360]

Custom mode uses the second parameter to emit particles at an angle with respect to the vector from the ring center to the spawn point. 0 degrees emits particles away from the center, and 180 degrees emits particles into the center.

SphereSurfaceEmitter

Variable	Definition	Values
radius	Radius of sphere	[0, infinity)

Custom mode generates particles with no initial velocity.

SphereVolumeEmitter

Variable	Definition	Values
radius	Radius of sphere	[0, infinity)

Custom mode emits particles away from the sphere center. Their velocity is dependent on their spawn location within the sphere. It is 0 at the center, of magnitude 1 at the outer edge of the sphere, and linearly interpolated in between.

TangentRingEmitter

Variable	Definition	Values
radius	Radius of ring	[0, infinity)

Custom mode emits particles tangentially to the ring edge, with a velocity magnitude of 1.

Panda3D Manual: Particle Renderers

Particle renderers add particles to the visible scene graph according to the information stored in the particle objects and the type of renderer. All particle renderers have the following parameters:

Variable	Definition	Values
alphaMode	Alpha setting over particle lifetime	PR_ALPHA_NONE, PR_ALPHA_OUT, PR_ALPHA_IN, PR_ALPHA_USER
userAlpha	Alpha value for ALPHA_USER alpha mode	Boolean

The following list contains the different types of renderers and their unique parameters.

PointParticleRenderer

Renders particles as pixel points.

Variable	Definition	Values
pointSize	Width and height of points, in pixels	[0, infinity)
startColor	Starting color	(r, g, b, a)
endColor	Ending color	(r, g, b, a)
blendType	How the particles blend from the start color to the end color	ONE_COLOR, BLEND_LIFE, BLEND_VEL
blendMethod	Interpolation method between colors	LINEAR, CUBIC

ONE_COLOR: point is always the starting color.
BLEND_LIFE: color is interpolated from start to end according to the age of the point
BLEND_VEL: color is interpolated between start to end according to the velocity/terminal velocity.

LineParticleRenderer

Renders particles as lines between their current position and their last position.

Variable	Definition	Values
headColor	Color of leading end	(r, g, b, a)
tailColor	Color of trailing end	(r, g, b, a)

SparkleParticleRenderer

Renders particles star or sparkle objects, three equal-length perpendicular axial lines, much

like jacks. Sparkle particles appear to sparkle when viewed as being smaller than a pixel.

Variable	Definition	Values
centerColor	Color of center	(r, g, b, a)
edgeColor	Color of edge	(r, g, b, a)
birthRadius	Initial sparkle radius	[0, infinity)
deathRadius	Final sparkle radius	[0, infinity)
lifeScale	Whether or not sparkle is always of radius birthRadius	NO_SCALE, SCALE

SpriteParticleRenderer

Renders particles as an image, using a Panda3D texture object. The image is always facing the user.

Variable	Definition	Values
texture	Panda texture object to use as the sprite image	(r, g, b, a)
color	Color	(r, g, b, a)
xScaleFlag	If true, x scale is interpolated over particleâ€™s life	Boolean
yScaleFlag	If true, y scale is interpolated over particleâ€™s life	Boolean
animAngleFlag	If true, particles are set to spin on the Z axis	Boolean
initial_X_Scale	Initial x scaling factor	[0, infinity)
final_X_Scale	Final x scaling factor	[0, infinity)
initial_Y_Scale	Initial y scaling factor	[0, infinity)
final_Y_Scale	Final y scaling factor	[0, infinity)
nonAnimatedTheta	If false, sets the counterclockwise Z rotation of all sprites, in degrees	Boolean
alphaBlendMethod	Sets the interpolation blend method	LINEAR, CUBIC
alphaDisable	If true, alpha blending is disabled	Boolean

GeomParticleRenderer

Renders particles as full 3D objects. This requires a geometry node.

Variable	Definition	Values
geomNode	A geometry scene graph node	<Node>

Panda3D Manual: Collision Detection

Collision detection allows for two objects to bump into each other and react. This includes not only sending messages for events, but also to keep the objects from passing through each other. Collision detection is a very powerful tool for immersion, but it is somewhat complex.

There are two ways to go about collision detection. One is to create special collision geometry, such as spheres and polygons, to determine collisions. The other is to allow collisions against all geometry. While the first is somewhat more complex and takes more effort to implement, it is much faster to execute and is a better long-term solution. For quick-and-dirty applications, though, collision with geometry can be a fine solution.

This section of the manual will address both methods.

Panda3D Manual: Collision Solids

The CollisionSolid is the fundamental object of the collision system. CollisionSolids represent special invisible geometry that is created solely for the purpose of performing collision tests; these CollisionSolids are stored in the scene graph alongside the normal visible geometry.

The CollisionSolids are specifically optimized for performing collision tests quickly. Collisions can be performed against visible geometry as well, but this is more expensive since visible geometry is not optimized for this sort of thing.

You can create CollisionSolids interactively in program code, or you can construct them in your modeling package and load them up from an egg or bam file along with the rest of your scene.

When you create a CollisionSolid interactively, you must also create a CollisionNode to hold the solid. (When you load your CollisionSolids in from an egg file, the CollisionNodes are created for you.) Often, a CollisionNode will be used to hold only one solid, but in fact a CollisionNode can contain any number of solids, and this is sometimes a useful optimization, especially if you have several solids that always move together as a unit.

```
cs = CollisionSphere(0, 0, 0, 1)
cnodePath = avatar.attachNewNode(CollisionNode
('cnode'))
cnodePath.node().addSolid(cs)
```

CollisionNodes are hidden by default, but they may be shown for debugging purposes:

```
cnodePath.show
()
```

Note: Be aware that the collision algorithm has only limited awareness of scaling transforms applied to CollisionSolids. If unequal scaling is applied between a from collider and an into collider, unexpected results may occur. In general, strive to have as few scaling transforms applied to your collision solids as possible.

There are several kinds of CollisionSolids available.

CollisionSphere

The sphere is the workhorse of the collision system. Spheres are the fastest primitives for any collision calculation; and the sphere calculation is particularly robust. If your object is even vaguely spherical, consider wrapping a sphere around it.

Also, a sphere is a particularly good choice for use as a "from object", because a sphere can

reliably be tested for collision with most of the other solid types. The "from objects" are the objects that are considered the active objects in the world; see Collision Traversers. A sphere is usually the best choice to put around the player's avatar, for instance. The sphere also makes a good "into object"; it is the only object type that is a good choice for both "from" and "into" objects.

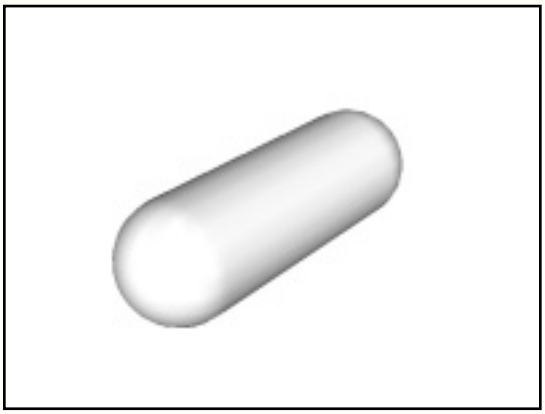
A sphere is defined in terms of a center and a radius. Note that, like any object, the sphere's coordinates are defined in the sphere's own coordinate space, so that often the center is (0, 0, 0).

```
sphere = CollisionSphere(cx, cy, cz,
radius)
```

CollisionTube

A "tube" is a cylinder with hemispherical endcaps. This shape is sometimes called a capsule in other collision systems.

The tube is good as an "into object", for objects that are largely cylindrical. It is not a very good choice for a "from object", because not many intersection tests have been written from tubes into other shapes.



A tube is defined with its two endpoints, and the cylindrical radius.

```
tube = CollisionTube(ax, ay, az, bx, by, bz,
radius)
```

CollisionInvSphere

The inverse sphere is a special-purpose solid that is rarely used, but occasionally it is very useful. It is an inside-out sphere: the solid part of the sphere is on the outside. Any object that is on the outside of the sphere is considered to be colliding with it; any object on the inside is not colliding.

Think of the inverse sphere as a solid mass that fills the whole universe in all directions, except for a bubble of space in the middle. It's useful for constraining an object within a particular space, since nothing can get out of an inverse sphere.

```
inv = CollisionInvSphere(cx, cy, cz,
radius)
```

CollisionPlane

The CollisionPlane is an infinite plane extending in all directions. It is not often used, but it can be useful in certain cases, for instance as a trigger placed below the ground to detect when an avatar has accidentally slipped through a crack in the world. You can also build a box out of six planes to keep objects perfectly constrained within a rectangular region, similar to an inverse sphere; such a box is much more reliable than one constructed of six polygons.

The plane actually divides the universe into two spaces: the space behind the plane, which is all considered solid, and the space in front of the plane, which is all empty. Thus, if an object is anywhere behind a plane, no matter how far, it is considered to be intersecting the plane.

A CollisionPlane is constructed using a Panda3D Plane object, which itself has a number of constructors, including the A, B, C, D plane equation, or a list of three points, or a point and a normal.

```
plane = CollisionPlane(Plane(Vec3(0, 0, 1), Point3(0, 0,
0)))
```

CollisionPolygon

A CollisionPolygon is the most general of the collision solids, since it is easy to model any shape with polygons (especially using a modeling package). However, it is also the most expensive solid, and the least robust--there may be numerical inaccuracies with polygons that allow collisions to slip through where they shouldn't.

Like a plane and a tube, a CollisionPolygon is only a good choice as an "into object". It doesn't support collision tests as a "from object".

In general, if you must use CollisionPolygons to model your shape, you should use as few polygons as possible. Use quads instead of triangles if possible, since two triangles take twice as much time to compute as a single quad. This does mean that you need to ensure that your quads are perfectly coplanar.

You can also make higher-order polygons like five-sided and six-sided polygons or more, but you cannot make concave polygons. If you create a concave or non-coplanar CollisionPolygon in your modeling package, Panda will automatically triangulate it for you (but this might result in a suboptimal representation, so it is usually better to subdivide a concave polygon by hand).

Unlike a plane, a CollisionPolygon is infinitely thin; an object is only considered to be colliding with the polygon while it is overlapping it.

When you create a CollisionPolygon interactively, you can only create triangles or quads (the higher-order polygons can only be loaded from an egg file). Simply specify the three or four points to the constructor, in counter-clockwise order.

```
quad = CollisionPolygon(Point3(0, 0, 0), Point3(0, 0,
1),
  Point3(0, 1, 1), Point3(0, 1, 0))
```

CollisionRay

The ray, line, and segment (below) are special collision solids that are useful only as a "from" object; since these objects have no volume, nothing will collide "into" a ray.

The CollisionRay represents an infinite ray that begins at a specific point, and stretches in one direction to infinity.

It is particularly useful for picking objects from the screen, since you can create a ray that starts at the camera's point of view and extends into the screen, and then determine which objects that ray is intersecting. (In fact, there is a method on CollisionRay called `setFromLens()` that automatically sets up the ray based on a 2-d onscreen coordinate; this is used by the "picker". See Clicking on 3D Objects.)

The CollisionRay is also useful in conjunction with the CollisionHandlerFloor; see Collision Handlers.

A CollisionRay is created by specifing an origin point, and a direction vector. The direction vector need not be normalized.

```
ray = CollisionRay(ox, oy, oz, dx, dy,
dz)
```

CollisionLine

This is essentially the same as a CollisionRay, except it extends to infinity in both directions. It is constructed with the same parameters, an origin point and a direction vector.

```
line = CollisionLine(ox, oy, oz, dx, dy,
dz)
```

CollisionSegment

Finally, a segment is another variant on the CollisionRay that does not extend to infinity, but only goes to a certain point and stops. It is useful when you want to put a limit on how far the CollisionRay would otherwise reach.

A CollisionSegment is constructed by specifying the two end points.

```
segment = CollisionSegment(ax, ay, az, bx, by,
bz)
```

<<prev top next>> Search

Panda3D Manual: Collision Handlers

You will need to create a CollisionHandler that specifies what to do when a collision event is detected. There are several possible kinds of CollisionHandler available.

CollisionHandlerQueue

The simplest kind of CollisionHandler, this object simply records the collisions that were detected during the most recent traversal. You can then iterate through the list using `queue.getNumEntries()` and `queue.getEntry()`:

```
queue = CollisionHandlerQueue()
traverser.addCollider(fromObject,
queue)
traverser.traverse(render)
for i in range(queue.getNumEntries()):
  entry = queue.getEntry(i)
  print entry
```

By default, the Collision Entries appear in the queue in no particular order. You can arrange them in order from nearest to furthest by calling `queue.sortEntries()` after the traversal.

CollisionHandlerEvent

This is another simple kind of CollisionHandler. Rather than saving up the collisions, it generates a Panda event when collision events are detected.

There are three kinds of events that may be generated: the "in" event, when a particular object collides with another object that it didn't in the previous pass, the "out" event, when an object is no longer colliding with an object it collided with in the previous pass, and the "again" event, when an object is still colliding with the same object that it did in the previous pass.

For each kind of event, the CollisionHandlerEvent will construct an event name out of the names of the from and into objects that were involved in the collision. The exact event name is controlled by a pattern string that you specify. For instance:

```
handler.addInPattern('%fn-into-%in')
handler.addAgainPattern('%fn-again-%
in')
handler.addOutPattern('%fn-out-%in')
```

In the pattern string, the following sequences have special meaning:

%fn	the name of the "from" object's node
%in	the name of the "into" object's node
%fs	't' if "from" is declared to be tangible, 'i' if intangible
%is	't' if "into" is declared to be tangible, 'i' if intangible
%ig	'c' if the collision is into a CollisionNode, 'g' if it is an ordinary visible GeomNode
%(tag)fh	generate event only if "from" node has the indicated tag
%(tag)fx	generate event only if "from" node does not have the indicated tag
%(tag)ih	generate event only if "into" node has the indicated tag
%(tag)ix	generate event only if "into" node does not have the indicated tag
%(tag)ft	the indicated tag value of the "from" node.
%(tag)it	the indicated tag value of the "into" node.

You may use as many of the above sequences as you like, or none, in the pattern string. In the tag-based sequences, the parentheses around (tag) are literal; the idea is to write the name of the tag you want to look up, surrounded by parentheses. The tag is consulted using the `nodePath.getNetTag()` interface.

In any case, the event handler function that you write to service the event should receive one parameter (in addition to self, if it is a method): the CollisionEntry. For example:

```
class MyObject(DirectObject.DirectObject):
  def __init__(self):
    self.accept('car-into-rail', handleRailCollision)

  def handleRailCollision(self, entry):
    print entry
```

Note that all of the following versions of CollisionHandler also inherit from CollisionHandlerEvent, so any of them can be set up to throw events in the same way.

CollisionHandlerPusher

This is the first of the more sophisticated handlers. The CollisionHandlerPusher, in addition to inheriting all of the event logic from CollisionHandlerEvent, will automatically push back on its from object to keep it out of walls. The visual effect is that your object will simply stop moving when it reaches a wall if it hits the wall head-on, or it will slide along the wall smoothly if it strikes the wall at an angle.

The CollisionHandlerPusher needs to have a handle to the NodePath that it will push back on, for each from object; you pass this information to `pusher.addCollider`. This should be the node that is actually moving. This is often, but not always, the same NodePath as the CollisionNode itself, but it might be different if the CollisionNode is set up as a child of the node that is actually moving.

```
smiley = loader.loadModel('smiley.egg')
fromObject = smiley.attachNewNode(CollisionNode
('colNode'))
fromObject.node().addSolid(CollisionSphere(0, 0, 0, 1))

pusher = CollisionHandlerPusher()
pusher.addCollider(fromObject, smiley)
```

Don't be confused by the call to `pusher.addCollider`; it looks a lot like the call to `traverser.addCollider`, but it's not the same thing, and you still need to add the collider and its handler to the traverser:

```
traverser.addCollider(fromObject,
pusher)
smiley.setPos(x, y, 0)
```

If you are using Panda's drive mode to move the camera around (or some other node), then you also need to tell the pusher about the drive node, by adding it into the `pusher.addCollider` call:

```
fromObject = base.camera.attachNewNode(CollisionNode
('colNode'))
fromObject.node().addSolid(CollisionSphere(0, 0, 0, 1))
pusher = CollisionHandlerPusher()
pusher.addCollider(fromObject, base.camera, base.drive.node())
```

PhysicsCollisionHandler

This kind of handler further specializes CollisionHandlerPusher to integrate with Panda's Physics Engine. It requires that the NodePath you pass as the second parameter to `pusher.addCollider` actually contains an ActorNode, the type of node that is moved by forces in the physics system.

```
anp = render.attachNewNode(ActorNode('actor'))
fromObject = anp.attachNewNode(CollisionNode
('colNode'))
fromObject.node().addSolid(CollisionSphere(0, 0, 0, 1))

pusher = PhysicsCollisionHandler()
pusher.addCollider(fromObject, anp)
```

Whenever you have an ActorNode that you want to respond to collisions, we recommend that you use a PhysicsCollisionHandler rather than an ordinary CollisionHandlerPusher. The

PhysicsCollisionHandler will keep the object out of walls, just like the CollisionHandlerPusher does, but it will also update the object's velocity within the physics engine, which helps to prevent the physics system from becoming unstable due to large accumulated velocities.

CollisionHandlerFloor

This collision handler is designed to serve one very specialized purpose: it keeps an object on the ground, or falling gently onto the ground, even if the floor is not level, without involving physics.

It is intended to be used with a `CollisionRay` or `CollisionSegment`. The idea is that you attach a ray to your object, pointing downward, such that the topmost intersection the ray detects will be the floor your object should be resting on. Each frame, the CollisionHandlerFloor simply sets your object's z value to the detected intersection point (or, if it is so configured, it slowly drops the object towards this point until it reaches it).

Using the CollisionHandlerFloor can be an easy way to simulate an avatar walking over uneven terrain, without having to set up a complicated physics simulation (or involve physics in any way). Of course, it does have its limitations.

```
smiley = loader.loadModel('smiley.egg')
fromObject = smiley.attachNewNode(CollisionNode
('colNode'))
fromObject.node().addSolid(CollisionRay(0, 0, 0, 0, 0, -
1))

lifter = CollisionHandlerFloor()
lifter.addCollider(fromObject, smiley)
```

Panda3D Manual: Collision Entries

For each collision detected, a new `CollisionEntry` object is created. This CollisionEntry stores all the information about the collision, including the two objects (nodes) involved in the collision, and the point of impact and the surface normal of the into object at that point.

The CollisionEntry object is passed to the event handler method by the `CollisionHandlerEvent` and its derivatives; it is also the object stored in the queue of collisions maintained by the `CollisionHandlerQueue`.

However you get a handle to CollisionEntry object, you can query it for information using the following methods:

`entry.getFromNodePath()`	Returns the NodePath of the "from" object. This NodePath will contain a CollisionNode.
`entry.getIntoNodePath()`	Returns the NodePath of the "into" object. This NodePath will contain a CollisionNode, or if the collision was made with visible geometry, a GeomNode.
`entry.getFrom()`	Returns the actual CollisionSolid of the "from" object. This is useful if there were more than one CollisionSolid in the "from" CollisionNode.
`entry.getInto()`	Returns the actual CollisionSolid of the "into" object. However, if the collision was made with visible geometry, there is no CollisionSolid, and this will be an invalid call.
`entry.hasInto()`	Returns true if the collision was made into a CollisionSolid as opposed to visible geometry, and thus the above call will be valid.
`entry.getSurfacePoint(nodePath)`	Returns the 3-D point of the collision, in the coordinate space of the supplied NodePath. This point will usually be on the surface of the "into" object.
`entry.getSurfaceNormal(nodePath)`	Returns the 3-D surface normal of the "into" object at the point of the collision, in the coordinate space of the supplied NodePath.
`entry.getInteriorPoint(nodePath)`	Returns the 3-D point, within the interior of the "into" object, that represents the depth to which the "from" object has penetrated.

The three methods that return points or vectors all accept a NodePath as a parameter. This can be any NodePath in the scene graph; it represents the coordinate space in which you expect to receive the answer. For instance, to receive the point of intersection in the coordinate space of the "into" object, use:

```
point = collisionEntry.getSurfacePoint(collisionEntry.getIntoNodePath())
```

If you wanted to put an axis at the point of the collision to visualize it, you might do something like this:

```
axis = loader.loadModel('zup-axis.egg')
axis.reparentTo(render)
point = collisionEntry.getSurfacePoint(render)
normal = collisionEntry.getSurfaceNormal(render)
axis.setPos(point)
axis.lookAt(point + normal)
```

<<prev top next>> Search

Panda3D Manual: Collision Traversers

A CollisionTraverser object performs the actual work of checking all solid objects for collisions. Normally, you will create a single CollisionTraverser object and assign it to `base.cTrav`; this traverser will automatically be run every frame. It is also possible to create additional CollisionTraversers if you have unusual needs; for instance, to run a second pass over a subset of the geometry. If you create additional CollisionTraversers, you must run them yourself.

The CollisionTraverser maintains a list of the active objects in the world, sometimes called the "colliders" or "from objects". The remaining collidable objects in the world that are not added to a CollisionTraverser are the "into objects". Each of the "from objects" is tested for collisions with all other objects in the world, including the other from objects as well as the into objects.

Note that it is up to you to decide how to divide the world into "from objects" and "into objects". Typically, the from objects are the moving objects in the scene, and the static objects like walls and floors are into objects, but the collision system does not require this; it is perfectly legitimate for a from object to remain completely still while an into object moves into it, and the collision will still be detected.

It is a good idea for performance reasons to minimize the number of from objects in a particular scene. For instance, the user's avatar is typically a from object; in many cases, it may be the only from object required--all of the other objects in the scene, including the walls, floors, and other avatars, might be simply into objects. If your game involves bullets that need to test for collisions with the other avatars, you will need to make either the bullets or the other avatars be from objects, but probably not both.

In order to add a from object to the CollisionTraverser, you must first create a CollisionHandler that defines the action to take when the collision is detected; then you pass the NodePath for your from object, and its CollisionHandler, to `addCollider`.

```
traverser = CollisionTraverser('traverser name')
base.cTrav = traverser
traverser.addCollider(fromObject, handler)
```

You only need to add the "from" objects to your traverser! Don't try to add the "into" objects to the CollisionTraverser. Adding an object to a CollisionTraverser makes it a "from" object. On the other hand, every object that you put in the scene graph, whether it is added to a CollisionTraverser or not, is automatically an "into" object.

Note that all of your "from" objects are typically "into" objects too (because they are in the scene graph), although you may choose to make them not behave as into objects by setting their CollideMask to zero.

Panda3D Manual: Collision Bitmasks

By default, every "from" object added to a CollisionTraverser will test for collisions with every other CollisionNode in the scene graph, and will not test for collisions with visible geometry. For simple applications, this is sufficient, but often you will need more control.

This control is provided with the collide masks. Every CollisionNode has two collide masks: a "from" mask, which is used when the CollisionNode is acting as a "from" object (i.e. it has been added to a CollisionTraverser), and an "into" mask, which is used when the node is acting as an "into" object (i.e. it is in the scene graph, and a from object is considering it for collisions).

In addition, visible geometry nodes--that is, GeomNodes--also have an "into" mask, so that visible geometry can serve as an "into" object also. (However, only a CollisionNode can serve as a "from" object.)

Before the solids in a "from" CollisionNode are tested for collisions with another CollisionNode or with a GeomNode, the collide masks are compared. Specifically, the "from" mask of the from object, and the "into" mask of the into object, are ANDed together. If the result is not zero--meaning the two masks have at least one bit in common--then the collision test is attempted; otherwise, the two objects are ignored.

The collide masks are represented using a BitMask32 object, which is really just a 32-bit integer with some additional methods for getting and setting particular bits.

You can only set the from collide mask on a collision node, and you must set it directly on the node itself, not on the NodePath:

```
nodePath.node().setFromCollideMask(BitMask32
(0x10))
```

However, the into collide mask may be set on the NodePath, for convenience; this recursively modifies the into collide mask for all the nodes at the given NodePath level and below.

```
nodePath.setCollideMask(newMask, bitsToChange,
nodeType)
```

The parameter newMask specifies the new mask to apply. The remaining parameters are optional; if they are omitted, then every node at nodePath level and below is assigned newMask as the new into collide mask. However, if bitsToChange is specified, it represents the set of bits that are to be changed from the original; bits that are 0 in bitsToChange will not be modified at each node level. If nodeType is specified, it should be a TypeHandle that represents the type of node that will be modified, e.g. `CollisionNode.getClassType()` to

affect only CollisionNodes.

Examples:

```
nodePath.setCollideMask(BitMask32
(0x10))
```

This sets the into collide mask of nodePath, and all children of nodePath, to the value 0x10, regardless of the value each node had before.

```
nodePath.setCollideMask(BitMask32(0x04), BitMask32
(0xff))
```

This replaces the lower 8 bits of nodePath and all of its children with the value 0x04, leaving the upper 24 bits of each node unchanged.

The default value for both from and into collide masks for a new CollisionNode can be retrieved by `CollisionNode.getDefaultCollideMask()`, and the default into collide mask for a new GeomNode is `GeomNode.getDefaultCollideMask()`. Note that you can create a CollisionNode that collides with visible geometry by doing something like this:

```
nodePath.node().setFromCollideMask(GeomNode.getDefaultCollideMask
())
```

Panda3D Manual: Rapidly-Moving Objects

Panda3D's collision system works by testing the current state of the world every frame for a possible intersection. If your objects are moving so quickly that they might pass completely through another object in the space of one frame, however, that collision might never be detected.

To avoid this problem, the Panda3D scene graph supports an advanced feature: it can record the *previous frame's position* of each moving object for the benefit of the CollisionTraverser. The CollisionTraverser can then take advantage of this information when it is testing for collisions. If it sees that a moving object was on one side of an object last frame, and on the opposite side this frame, it can trigger the collision detection even though the two objects might not currently be intersecting.

There are a few things you need to do to activate this mode.

1. First, you must tell the CollisionTraverser that you intend to use this mode; by default, it ignores the previous position information. To activate this mode, call:

```
base.cTrav.setRespectPrevTransform(True)
```

You only need to make this call once, at the beginning of your application (or whenever you create the CollisionTraverser). That switches the CollisionTraverser into the new mode. If you create any additional CollisionTraversers, you should make the call for them as well.

2. Ensure that `base.resetPrevTransform(render)` is called every frame. Actually, this is already done for you automatically by ShowBase.py, so normally you don't need to do anything for this step.

The `resetPrevTransform()` call should be made once per frame (at the very beginning of the frame) for every different scene graph in your application that involves collisions. It ensures that the current frame's position is copied to the previous frame's position, before beginning the processing for that frame. Note that if you have multiple CollisionTraversers handling the same scene graph, you only need to (and only should) call this function once, but if you have two or more disconnected scene graphs, you will need to call it for each scene graph.

If you don't understand the above paragraph, then you aren't using disconnected scene graphs, and you shouldn't worry about it.

3. Whenever you move an object from one point to another in your scene (except when you put it into your scene the first time), instead of using:

```
object.setPos(newPos)
```

You should use:

```
object.setFluidPos(newPos)
```

In general, `setPos()` means "put the object here, directly" and `setFluidPos()` means "slide the object here, testing for collisions along the way". It is important to make a clear distinction between these two calls, and make the appropriate call for each situation.

If you are moving an object with a LerpInterval, and you want collisions to be active (and fluid) during the lerp, you should pass the keyword parameter `fluid = 1` to the LerpInterval constructor. It is rare to expect collisions to be active while an object is moving under direct control of the application, however.

Visualizing the previous transform

When you are using the setFluidPos() call, and you have called `show()` on your CollisionNode to make it visible, you will see the CollisionNode itself each frame, plus a ghosted representation of where it was the previous frame. This can help you visually see that the previous-transform mechanism is working. (It does not guarantee that the `setRespectPrevTransform()` call has been made on your CollisionTraverser, however.)

Caveats

At the present, the CollisionTraverser only uses the previous transform information when it is testing a CollisionSphere into a CollisionPolygon--that is, when the "from" object is a CollisionSphere, and the "into" object is a CollisionPolygon (or a wall of CollisionPolygons). Other kinds of collision solids currently do not consider the previous transform. (However, the other collision solids are generally thicker than a CollisionPolygon, so it is less likely that a moving object will pass all the way through them in one frame--so it is not quite as bad as it seems.)

Enabling the previous transform mode helps reduce slipping through walls considerably. However, it's not perfect; no collision system is. If your object is moving tremendously fast, or just happens to get lucky and slip through a tiny crack between adjacent polygons, it may still get through without detecting a collision. Any good application will be engineered so that the occasional collision slip does not cause any real harm.

The CollisionHandlerFloor is especially bad about allowing objects to slip through floors, in spite of the previous transform state, especially when you avatar is walking up a sloping path. This is just because of the way the CollisionHandlerFloor works. If you are having problems with the CollisionHandlerFloor, consider reducing the slope of your floors, increasing the height of the ray above the ground, and/or reducing the speed of your avatar.

Panda3D Manual: Pusher Example

Here is a short example to show two small spheres using a Pusher.

```
import direct.directbase.DirectStart
from pandac.PandaModules import *
from direct.interval.IntervalGlobal import *

#initialize traverser
base.cTrav = CollisionTraverser()

#initialize pusher
pusher = CollisionHandlerPusher()

#########

#load a model. reparent it to the camera so we can move it.
smiley = loader.loadModel('smiley')
smiley.reparentTo(camera)
smiley.setPos(0, 25.5,0.5)

#create a collision solid for this model
cNode = CollisionNode('smiley')
cNode.addSolid(CollisionSphere(0,0,0,1.1))
smileyC = smiley.attachNewNode(cNode)
smileyC.show()

#########

#load a model
frowney = loader.loadModel('frowney')
frowney.reparentTo(render)
frowney.setPos(5, 25,0)

#create a collsion solid for this model
cNode = CollisionNode('frowney')
cNode.addSolid(CollisionSphere(0,0,0,1.1))
frowneyC = frowney.attachNewNode(cNode)
frowneyC.show()

#########
#add collision node to the traverser and the pusher

base.cTrav.addCollider(frowneyC,pusher)
pusher.addCollider(frowneyC,frowney, base.drive.node())

#########

#have the one ball moving to help show what is happening
frowney.posInterval(5,Point3(5,25,0),startPos=Point3(-5,25,0),fluid=1).loop
()

#run the world. move around with the mouse to see how the moving ball
```

```
changes
#course to avoid the one attached to the camera.

run()
```

Panda3D Manual: Event Example

Here is a short example of using the Collision Handler Events

```
import direct.directbase.DirectStart
from direct.interval.IntervalGlobal import * #to make things flash on collisions
from pandac.PandaModules import *
from direct.showbase import DirectObject #so that we can accept messages


class World( DirectObject.DirectObject ):
        def __init__( self ):
                #initialize traverser
                base.cTrav = CollisionTraverser()

                #initialize handler
                self.collHandEvent=CollisionHandlerEvent()
                self.collHandEvent.addInPattern('into-%in')
                self.collHandEvent.addOutPattern('outof-%in')

                #initialize collision count (for unique collision strings)
                self.collCount=0

                #load a model.  reparent it to the camera so we can move it.
                s = loader.loadModel('smiley')
                s.reparentTo(camera)
                s.setPos(0, 25,0)

                #setup a collision solid for this model
                sColl=self.initCollisionSphere(s, True)

                #add this object to the traverser
                base.cTrav .addCollider(sColl[0] , self.collHandEvent)

                #accept the events sent by the collisions
                self.accept( 'into-' + sColl[1], self.collide3)
                self.accept( 'outof-' + sColl[1], self.collide4)
                print sColl[1]

                #load a model.
                t = loader.loadModel('smiley')
                t.reparentTo(render)
                t.setPos(5, 25,0)

                #setup a collision solid for this model
                tColl=self.initCollisionSphere(t, True)

                #add this object to the traverser
                base.cTrav .addCollider(tColl[0], self.collHandEvent)

                #accept the events sent by the collisions
                self.accept( 'into-' + tColl[1], self.collide)
                self.accept( 'outof-' + tColl[1], self.collide2)
                print tColl[1]

                print "WERT"


        def collide(self, collEntry):
                print "WERT: object has collided into another object"
                Sequence(Func(collEntry.getFromNodePath().getParent().setColor, VBase4(1,0,0,1)),
                             Wait(.2),
                             Func(collEntry.getFromNodePath().getParent().setColor, VBase4(0,1,0,1)),
                             Wait(.2),
                             Func(collEntry.getFromNodePath().getParent().setColor, VBase4(1,1,1,1))).start
()
```

```
        def collide2(self, collEntry):
                #collEntry.getFromNodePath().getParent().remove()
                print "WERT.: object is no longer colliding with another object"

        def collide3(self, collEntry):
                #collEntry.getFromNodePath().getParent().remove()
                print "WERT2: object has collided into another object"

        def collide4(self, collEntry):
                #collEntry.getFromNodePath().getParent().remove()
                print "WERT2: object is no longer colliding with another object"


        def initCollisionSphere( self, obj, show=False):
                #get the size of the object for the collision sphere
                bounds = obj.getChild(0).getBounds()
                center = bounds.getCenter()
                radius = bounds.getRadius()*1.1

                #create a collision sphere and name it something understandable
                collSphereStr = 'CollisionHull' +str(self.collCount)+"_"+obj.getName()
                self.collCount+=1
                cNode=CollisionNode(collSphereStr)
                cNode.addSolid(CollisionSphere(center, radius ) )

                cNodepath=obj.attachNewNode(cNode)
                if show:
                        cNodepath.show()

                # return a tuple with the collision node and its corrsponding string
                # return the collison node so that the bitmask can be set
                return (cNodepath,collSphereStr )



#run the world.  move around with the mouse to create collisions
w= World()
run()
```

Panda3D Manual: Bitmask Example

Here is a short example of using bitmasks

```
import direct.directbase.DirectStart
from direct.showbase import DirectObject
from pandac.PandaModules import *


class World( DirectObject.DirectObject ):
        def __init__( self ):
                #initialize traverser
                base.cTrav = CollisionTraverser()

                #initialize handler
                self.collHandEvent=CollisionHandlerEvent()
                self.collHandEvent.addInPattern('into-%in')
                self.collHandEvent.addOutPattern('outof-%in')

                #initialize collision count (for unique collision strings)
                self.collCount=0


                self.loadObj1()
                self.loadObj2()


        def loadObj1(self):
                #load a model. reparent it to the camera so we can move it.
                s = loader.loadModel('smiley')
                s.reparentTo(camera)
                s.setPos(0, 25,0)

                #setup a collision solid for this model
                sColl=self.initCollisionSphere(s, True)

                #set up bitmasks
                #this object can only collide into things
                sColl[0].setIntoCollideMask( BitMask32.allOff() )
                #this object will only collide with objects with this bitmask
                sColl[0].setFromCollideMask( BitMask32.bit( 1 ) )

                #add this object to the traverser
                base.cTrav .addCollider(sColl[0] , self.collHandEvent)


        def loadObj2(self):
                #load a model.
                t = loader.loadModel('smiley')
                t.reparentTo(render)
                t.setPos(5, 25,0)

                #setup a collision solid for this model
                tColl=self.initCollisionSphere(t, True)

                #set up bitmasks
                #this object can only be collided into and will collide with the other object
                tColl[0].setIntoCollideMask( BitMask32.bit( 1 ))
                tColl[0].setFromCollideMask( BitMask32.allOff())
```

```
                #add this object to the traverser
                base.cTrav .addCollider(tColl[0], self.collHandEvent)

                #accept the events sent by the collisions
                self.accept( 'into-' + tColl[1], self.collide)
                self.accept( 'outof-' + tColl[1], self.collide2)


        def collide(self, collEntry):
                print "WERT: object has collided into another object"

        def collide2(self, collEntry):
                print "WERT: object is no longer colliding with another object"

        def initCollisionSphere( self, obj, show=False):
                #get the size of the object for the collision sphere
                bounds = obj.getChild(0).getBounds()
                center = bounds.getCenter()
                radius = bounds.getRadius()*1.1

                #create a collision sphere and name it something understandable
                collSphereStr = 'CollisionHull' +obj.getName()
                cNode=CollisionNode(collSphereStr)
                cNode.addSolid(CollisionSphere(center, radius ) )
                cNodepath=obj.attachNewNode(cNode)
                if show:
                        cNodepath.show()

                # return a tuple with the collision node and its corrsponding string
                # return the collison node so that the bitmask can be set
                return (cNode,collSphereStr )



#run the world. move around with the mouse to create collisions
w= World()
run()
```

Panda3D Manual: Clicking on 3D Objects

Thanks to Shao Zang and Phil Saltzman for their tutorial program included with Panda 3D 1.0.4.

The simplest way to click on 3D objects in Panda is to use very simplistic collision detection coupled with event processing.

First, after a CollisonTraverser and a CollisionHandler have been setup, attach a CollisionRay node to the camera. This node will have its `setFromCollideMask()` set to `GeomNode.getDefaultCollideMask()` in order to be as general as possible:

```
pickerNode=CollisonNode('mouseRay')
pickerNP=camera.attachNewNode(pickerNode)
pickerNode.setFromCollideMask(GeomNode.getDefaultCollideMask
())
pickerRay=CollisionRay()
pickerNode.addSolid(pickerRay)
myTraverser.addCollider(pickerNode, myHandler)
```

For any object that you want to be pickable you should add a flag to it. The easiest way is to use the `setTag()` function, e.g.:

```
object1.setTag('myObjectTag',
'1')
object2.setTag('myObjectTag',
'2')
```

The above example sets the tag 'myObjectTag' on two objects in your graph that you want to designate as pickable (we will check for the presence of this tag later, when we get the response back from the collision system).

Now assume that the function `myFunction()` is set up to be called for the 'mouse1' event. In `myFunction()` is where you call `pickerRay.setFromLens(origin, destX, destY)`. This makes the ray's origin `origin` and the ray's vector the direction from `origin` to the point (`destX`,`destY`).

```
def myFunction():
   #This gives up the screen coordinates of the mouse
   mpos=base.mouseWatcherNode.getMouse()

   #This makes the ray's origin the camera and makes the ray
point
   #to the screen coordinates of the mouse
   pickerRay.setFromLens(base.camNode, mpos.getX(), mpos.getY())
```

After this, you now call the traverser like any other collision, get the closest object, and "pick" it.

```
def myFunction():
   mpos=base.mouseWatcherNode.getMouse()
   pickerRay.setFromLens(base.camNode, mpos.getX(), mpos.getY())

   myTraverser.traverse(render)
   #assume for simplicity's sake that myHandler is a CollisionHandlerQueue
   if myHandler.getNumEntries > 0:
      myHandler.sortEntries() #this is so we get the closest object
      pickedObj=myHandler.getEntry(0).getIntoNodePath()
```

Now, the node `pickedObj` returned by the collision system may not be the object itself, but might be just a tiny piece of the object--a wheel, or a nose, or whatever. In particular, it will be one of the GeomNodes that make up the object (the `GeomNode` class contains the visible geometry primitives that are used to define renderable objects in Panda). Since your object might consist of more than one GeomNode, what you probably would prefer to get is the NodePath that represents the parent of all of these GeomNodes--that is, the NodePath that you set the 'myObjectTag' tag on, above. You can use `nodePath.findNetTag()` to return the parent NodePath that contains a specified tag.

(There are also other, similar methods on NodePath that can be used to query the tag specified on a parent node, such as `getNetTag()` and `hasNetTag()`. For simplicity we will restrict this example to `findNetTag()`.)

Now you can edit `myFunction` to look like:

```
def myFunction():
   mpos=base.mouseWatcherNode.getMouse()
   pickerRay.setFromLens(base.camNode, mpos.getX(), mpos.getY())

   myTraverser.traverse(render)
   #assume for simplicity's sake that myHandler is a CollisionHandlerQueue
   if myHandler.getNumEntries() > 0:
      myHandler.sortEntries() #this is so we get the closest object
      pickedObj=myHandler.getEntry(0).getIntoNodePath()
      pickedObj=pickedObj.findNetTag('myObjectTag')
      if not pickedObj.isEmpty():
         handlePickedObject(pickedObj)
```

<<prev top next>>

Panda3D Manual: Example for Clicking on 3D Objects

This is a small example program for clicking on 3D objects. A panda, a teapot, and a cube will be on screen. When you click on the screen the console will tell you the nodePath of what you have clicked on. Its basically a watered down version the tutorial included with Panda 3D 1.0.4. However, all the functionality for picking 3D objects is encapsulated into the Picker class which you may feel free to use in your own code.

```
import direct.directbase.DirectStart
#for the events
from direct.showbase import DirectObject
#for collision stuff
from pandac.PandaModules import *


class Picker(DirectObject.DirectObject):
   def __init__(self):
      #setup collision stuff

      self.picker= CollisionTraverser()
      self.queue=CollisionHandlerQueue()

      self.pickerNode=CollisionNode('mouseRay')
      self.pickerNP=camera.attachNewNode(self.pickerNode)

      self.pickerNode.setFromCollideMask(GeomNode.getDefaultCollideMask())

      self.pickerRay=CollisionRay()

      self.pickerNode.addSolid(self.pickerRay)

      self.picker.addCollider(self.pickerNode, self.queue)

      #this holds the object that has been picked
      self.pickedObj=None

      self.accept('mouse1', self.printMe)

   #this function is meant to flag an object as being somthing we can pick
   def makePickable(self,newObj):
      newObj.setTag('pickable','true')

   #this function finds the closest object to the camera that has been hit by our ray
   def getObjectHit(self, mpos): #mpos is the position of the mouse on the screen
      self.pickedObj=None #be sure to reset this
      self.pickerRay.setFromLens(base.camNode, mpos.getX(),mpos.getY())
      self.picker.traverse(render)
      if self.queue.getNumEntries() > 0:
         self.queue.sortEntries()
         self.pickedObj=self.queue.getEntry(0).getIntoNodePath()

         parent=self.pickedObj.getParent()
         self.pickedObj=None

         while parent != render:
            if parent.getTag('pickable')=='true':
               self.pickedObj=parent
               return parent
            else:
               parent=parent.getParent()
      return None

   def getPickedObj(self):
         return self.pickedObj
```

```
   def printMe(self):
         self.getObjectHit( base.mouseWatcherNode.getMouse())
         print self.pickedObj

mousePicker=Picker()

#load thest models
panda=loader.loadModel('panda')
teapot=loader.loadModel('teapot')
box=loader.loadModel('box')

#put them in the world
panda.reparentTo(render)
panda.setPos(camera, 0, 100,0)

teapot.reparentTo(render)
teapot.setPos(panda, -30, 0, 0)

box.reparentTo(render)
box.setPos(panda, 30,0,0)

mousePicker.makePickable(panda)
mousePicker.makePickable(teapot)
mousePicker.makePickable(box)

run()
```

If you are running this example on Panda 1.1, replace line 24 with

```
self.picker.addCollider(self.pickerNP, self.queue)
```

Note: If you switch to another window, the mouse picker may not work when you are back to the panda window. If that happens, just select the perform some window action.If it still doesn't work still, replace printMe method with this:

```
  def printMe(self):
        if base.mouseWatcherNode.hasMouse():
        self.getObjectHit( base.mouseWatcherNode.getMouse())
        print self.pickedObj
```

Panda3D Manual: Hardware support

Here is some advuice on interating with hardware using Panda3D's build in keyboard and mouse support, as well as some suggestions for getting joystick information into Panda3D.

Panda3D Manual: Keyboard Support

Panda3D has keyboard support built in. Keyboard presses send Events. Each keyboard key will send an event when it is first pressed down, when it is released, and one repeatedly while its pressed.

The events can be accepted with the following code:

```
self.accept( <event name> , <Function> )
self.accept( <event name> , <Function> , <parameters
list> )
```

<event name> is a string that labels the event.
<Function> is a python function to be called when the event is sent.
<parameters list> is a python list of parameters to use to call <Function>.

The <event name> that a key sends is fairly predictable base on these rules:

1. Keys that type a character are named that character. It is always lowercase even with shift or caps lock (Shift and other modifiers are explained below.)

e.g.

```
"a", "b", "c", "[", and "]"
```

not

```
"A", "B", "C", "{", and "}"
```

2. The key down event is named for the key.

3. As of 1.3.0 The keyboard autorepeat is named for the key + "-repeat" e.g.

```
"a-repeat", "b-repeat", "[-
repeat"
```

4. The key up event is named for the key + "-up" e.g.

```
"a-up", "b-up", "[-
up"
```

5. All key events (including "-up") have a corresponding time event labeled

```
"time-" + <key
name>
```

that send a time argument corresponding to the time that event was fired

6. Keys that don't type a character are labeled as follows:

```
"escape", "f"+"1-12" (e.g. "f1","f2",..."f12"), "print_screen-up" (no down.)
"scroll_lock"

"backspace", "insert", "home", "page_up", "num_lock"
"tab", "delete", "end", "page_down"
"caps_lock", "enter", "arrow_left", "arrow_up", "arrow_down", "arrow_right"
"shift", "lshift", "rshift",
"conrol", "alt", "lcontrol", "window-event"(no up?), "lalt", "space", "ralt",
"rcontrol"
```

7. Some physical keys are distinguishable from the events that they fire, and some are not. The modifier keys distinguish between left and right, but send a neutral event as well. (e.g. the left shift key sends both "lshift" and "shift" events when pressed) Save for "num_lock", "*", and "+" the numpad keys are indistinguishable from the main keyboard counterparts. (e. g. when Num Lock is on the both the numpad and keyboard 1 keys send "1")

8. Keys pressed in combination with modifiers send an additional event. The name is the modifier appended before the key and separated with a dash in the order shift conrol alt e.g.:

```
"shift-a" "shift-control-alt-a" "shift-alt-
a"
```

These compound events don't send a "time-" event. If you need one, use the "time-" event sent by one of the keys in the combination.

9. You can see these results for yourself using `messenger.toggleVerbose()`

Here are some examples in code:

```
self.accept('k' , self.__spam )#calls the function __spam() on the k key
event.
self.accept('k-up', self.__spam, [eggs, sausage, bacon,] )#calls __spam(eggs,
sausage,bacon)
self.accept('escape' , sys.exit )#exit on esc
self.accept('arrow_up' , self.spamAndEggs )#call spamAndEggs when up is
pressed
self.accept('arrow_up-repeat, self.spamAndEggs )#and at autorepeat if held
self.accept('arrow_up-up' self.spamAndEggs )#calls when the up arrow key is
released
```

Panda3D Manual: Mouse Support

Panda3D has mouse support built in. The default action of the mouse is to control the camera. If you want to disable this functionality you can use the command:

```
base.disableMouse
()
```

This function's name is slightly misleading. It only disables the task that drives the camera around, it doesn't disable the mouse itself. You can still get the position of the mouse, as well as the mouse clicks.

To get the position:

```
if base.mouseWatcherNode.hasMouse():
  x=base.mouseWatcherNode.getMouseX
()
  y=base.mouseWatcherNode.getMouseY
()
```

The mouse clicks generate "events." To understand what events are, and how to process them, you will need to read the Event Handling section. The names of the events generated are:

mouse1	Mouse Button 1 Pressed
mouse2	Mouse Button 2 Pressed
mouse3	Mouse Button 3 Pressed
mouse1-up	Mouse Button 1 Released
mouse2-up	Mouse Button 2 Released
mouse3-up	Mouse Button 3 Released

If you want to hide the mouse cursor, you want the line: "cursor hidden #t" in your Config.prc or this section of code:

```
props = WindowProperties()
props.setCursorHidden(1)
base.win.requestProperties
(props)
```

Panda3D Manual: Joystick Support

Note: I have been told that these instructions are inaccurate. - Josh

While Panda3D has mouse and keyboard support, it is best to look to the open source community for joystick support. Pygame is an open-source module that contains joystick support that may be easily included into a Panda3D application. Pygame may be found at http://www.pygame.org.

After downloading pygame, simply import the modules as you would any Panda3D module.

```
import pygame
```

Once pygame is imported, it needs to be initialized. Also, when the program is through with using pygame, it should be exited cleanly.

```
pygame.init()
pygame.quit
()
```

Also, the joystick should be initialized. It too has a quit function.

```
joystick.init()
joystick.quit
()
```

From here, it is possible to get the axis information of the joystick as well as the state of the buttons.

```
joystick.get_axis(<Axis>)
joystick.get_button
(<Button>)
```

These are the primary functions for the joystick, but there are a number of other functions available for joystick support. This can be found at the pygame website.

Panda3D Manual: VR Helmets and Trackers

This section is especially geared towards Carnegie Mellon University's virtual reality equipment. The tracking setup used consists of a magnetic tracker base that receives signals, a virtual reality headmount, and four small trackers, one of which is attached to the headmount. These magnetic trackers may be held in the hand or implanted in non-metallic objects, and they will relay their position and orientation to the tracker base, which then supplies it to the program.

First, make sure that the VRHandler is in the same folder as your python files. Then, import it as you would any Panda3D module.

```
from VRHandler import
*
```

Once imported, the VRHandler functions are now available. Make a constant in your code that controls whether or not the tracker is used.

```
USE_TRACKER =
True
```

or

```
USE_TRACKER =
False
```

Then, activate the tracker if required:

```
if USE_TRACKER:
  self.tracker = Tracker()
  camera.reparentTo(self.tracker.getHMDHelper())
else:
  print "Using Mouse and Keyboard Controls"
  self.controls = MouseAndKeyboardControls()
  self.controls.start()
```

If the tracker is not being used, the above code enables first-person-shooter style controls, using WASD to move, R and F for height, and the mouse to look around. You can add the following lines to your code if you want to customize the feel of the controls:

```
# Keyboard control constants
self.controls.acceleration = 15
self.controls.maxSpeed = 5
self.controls.friction = 6
```

Change those three numbers to whatever feels right for your world.

Typically, the tracker base is 6.5 feet off the ground, and the range of the trackers is from three or so inches from the tracker base to around the knees of an average person. The tracker base is treated as a NodePath, so it may be moved around. If you want the guest to move around the world while his real-life feet stay put, get the tracker base helper and move that around:

```
self.trackerBaseHelper = self.tracker.getTrackerBaseHelper()
self.trackerBaseHelper.setPos(0, 3, 0)
```

Finally, objects within the Panda3D application may be reparented to the trackers. The four trackers are HMDHelper, YellowHelper, GreenHelper, and BlueHelper. Also, remember that reparenting may create some strange inheritance issues.

```
NodePath.reparentTo(self.tracker.getHMDHelper() )
NodePath.reparentTo(self.tracker.getYellowHelper() )
NodePath.reparentTo(self.tracker.getGreenHelper() )
NodePath.reparentTo(self.tracker.getBlueHelper() )
```

Panda3D Manual: Math Engine

Panda3D has a number of vector, matrix, and quaternion operations built-in. The relevant classes are:

WRITE ME

Panda3D Manual: Matrix Representation

Periodically, the question arises: does Panda store matrices in column-major or row-major format? Unfortunately, people who ask that question often fail to realize that there are four ways to represent matrices, two of which are called "column major," and two of which are called "row major." So the answer to the question is not very useful. This section explains the four possible ways to represent matrices, and then explains which one panda uses.

The Problem

In graphics, matrices are mainly used to transform vertices. So one way to write a matrix is to write the four transform equations that it represents. Assuming that the purpose of a matrix is to transform an input-vector (Xi,Yi,Zi,Wi) into an output vector (Xo,Yo,Zo,Wo), the four equations are:

```
Xo = A*Xi + B*Yi + C*Zi +
D*Wi
Yo = E*Xi + F*Yi + G*Zi +
H*Wi
Zo = J*Xi + K*Yi + L*Zi +
M*Wi
Wo = N*Xi + O*Yi + P*Zi +
Q*Wi
```

There are two different orders that you can store these coefficients in RAM:

```
Storage Option 1: A,B,C,D,E,F,G,H,J,K,L,M,N,O,P,
Q
Storage Option 2: A,E,J,N,B,F,K,O,C,G,L,P,D,H,M,
Q
```

Also, when you're typesetting these coefficients in a manual (or printing them on the screen), there are two possible ways to typeset them:

```
A B C D          A E J N
E F G H          B F K O
J K L M          C G L P
N O P Q          D H M Q

Typesetting      Typesetting
Option 1         Option 2
```

These are *independent choices*! There is no reliable relationship between the order that people choose to store the numbers, and the order in which they choose to typeset them. That means that any given system could use one of four different notations.

So clearly, the two terms "row major" and "column major" are not enough to distinguish the four possibilities. Worse yet, to my knowledge, there is no established terminology to name the four possibilities. So the next part of this section is dedicated to coming up with a usable terminology.

The Coefficients are Derivatives

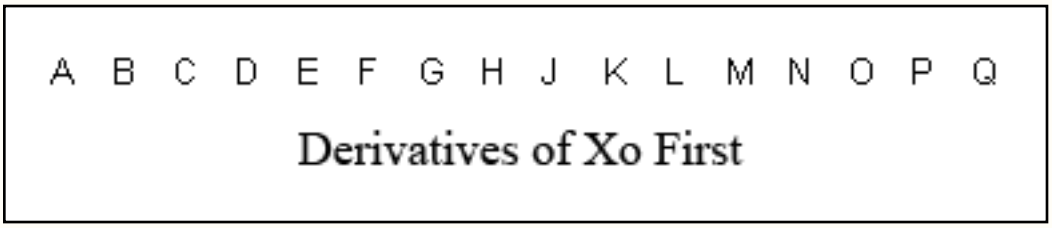
The equations above contain sixteen coefficients. Those coefficients are derivatives. For example, the coefficient "G" could also be called "the derivative of Yo with respect to Zi."

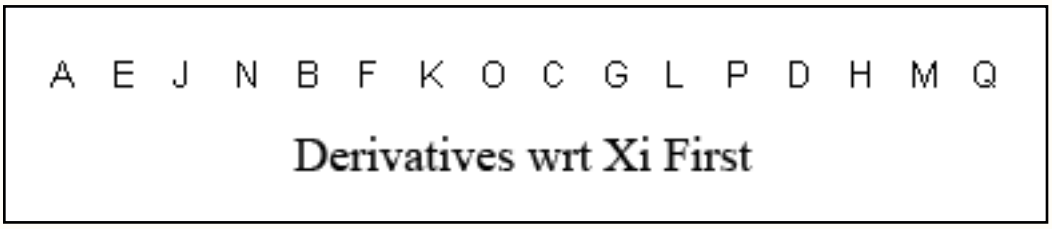
This gives us a handy way to refer to groups of coefficients. Collectively, the coefficients "A,B,C,D" could also be called "the derivatives of Xo with respect to Xi,Yi,Zi,Wi" or just "the derivatives of Xo" for short. The coefficients "A,E,J,N" could also be called "the derivatives of Xo,Yo,Zo,Wo with respect to Xi" or just "the derivatives with respect to Xi" for short.

This is a good way to refer to groups of four coefficients because it unambiguously names four of them without reference to which storage option or which typesetting option you choose.

What to Call the Two Ways of Storing a Matrix.

So here, again, are the two ways of storing a matrix. But using this newfound realization that the coefficients are derivatives, I have a meaningful way to name the two different ways of storing a matrix:





In the first storage scheme, the derivatives of Xo are stored first. In the second storage scheme, the derivatives with respect to Xi are stored first.

What to Call the Two Ways of Printing a Matrix.

One way to write the four equations above is to write them out using proper mathematical notation. There are two ways to do this, shown below:

$$\begin{bmatrix} Xo \\ Yo \\ Zo \\ Wo \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ J & K & L & M \\ N & O & P & Q \end{bmatrix} \times \begin{bmatrix} Xi \\ Yi \\ Zi \\ Wi \end{bmatrix}$$

Column Vectors

$$\begin{bmatrix} Xo & Yo & Zo & Wo \end{bmatrix} = \begin{bmatrix} Xi & Yi & Zi & Wi \end{bmatrix} \times \begin{bmatrix} A & E & J & N \\ B & F & K & O \\ C & G & L & P \\ D & H & M & Q \end{bmatrix}$$

Row Vectors

Notice that the two matrices shown above are laid out differently. The first layout is the appropriate layout for use with column vectors. The second layout is the appropriate layout for use with row vectors. So that gives me a possible terminology for the two different ways of typesetting a matrix: the "row-vector-compatible" notation, and the "column-vector-compatible" notation.

The Four Possibilities

So now, the four possible representations that an engine could use are:

1. Store derivatives of Xo first, typeset in row-vector-compatible notation.
2. Store derivatives of Xo first, typeset in column-vector-compatible notation.
3. Store derivatives wrt Xi first, typeset in row-vector-compatible notation.
4. Store derivatives wrt Xi first, typeset in column-vector-compatible notation.

The Terms "Column Major" and "Row Major"

The term "row-major" means "the first four coefficients that you store, are also the first row when you typeset." So the four possibilities break down like this:

1. Store derivatives of Xo first, typeset in row-vector-compatible notation (COLUMN MAJOR)
2. Store derivatives of Xo first, typeset in column-vector-compatible notation (ROW MAJOR)
3. Store derivatives wrt Xi first, typeset in row-vector-compatible notation (ROW MAJOR)
4. Store derivatives wrt Xi first, typeset in column-vector-compatible notation (COLUMN MAJOR)

That makes the terms "row major" and "column major" singularly useless, in my opinion. They tell you nothing about the actual storage or typesetting order.

Panda Notation

Now that I've established my terminology, I can tell you what panda uses. If you examine the panda source code, in the method "LMatrix4f::xform," you will find the four transform equations. I have simplified them somewhat (ie, removed some of the C++ quirks) in order to put them here:

```
define VECTOR4_MATRIX4_PRODUCT(output, input, M) \
output._0 = input._0*M._00 + input._1*M._10 + input._2*M._20 + input._3*M._30;
\
output._1 = input._0*M._01 + input._1*M._11 + input._2*M._21 + input._3*M._31;
\
output._2 = input._0*M._02 + input._1*M._12 + input._2*M._22 + input._3*M._32;
\
output._3 = input._0*M._03 + input._1*M._13 + input._2*M._23 + input._3*M._33;
```

Then, if you look in the corresponding header file for matrices, you will see the matrix class definition:

```
struct {
FLOATTYPE _00, _01, _02,
_03;
FLOATTYPE _10, _11, _12,
_13;
FLOATTYPE _20, _21, _22,
_23;
FLOATTYPE _30, _31, _32,
_33;
} m;
```

So this class definition shows not only how the coefficients of the four equations are stored, but also the layout in which they were intended to be typeset. So from this, you can see that panda stores derivatives wrt Xi first, and it typesets in row-vector-compatible notation.

Interoperability with OpenGL and DirectX

Panda is code-compatible with both OpenGL and DirectX. All three use the same storage format: derivatives wrt Xi first. You can pass a panda matrix directly to OpenGL's "glLoadMatrixf" or DirectX's "SetTransform".

However, remember that typesetting format and data storage format are *independent choices*. Even though two engines are interoperable at the code level (because their data storage formats match), their manuals might disagree with each other (because their typesetting

formats do not match).

The panda typesetting conventions and the OpenGL typesetting conventions are opposite from each other. The OpenGL manuals use a column-vector-compatible notation. The Panda manuals use a row-vector-compatible notation.

I do not know what typesetting conventions the DirectX manual uses.

<<prev top next>> Search

Panda3D Manual: Physics Engine

Panda3D has a very basic physics engine that may apply forces to classes. The physics engine can handle angular or linear forces, as well as viscosity.

To make use of the collision engine, first enable the particle system. The particle system relies upon the physics engine to move and update particles, so enabling the particle system adds the tasks in the engine that monitor and update the interactions of physics-enabled objects in the scene.

```
base.enableParticles()
```

The rest of this section will address how to prepare a model for physical interactions and apply forces to the model.

Panda3D Manual: Enabling physics on a node

The ActorNode is the component of the physics system that tracks interactions and applies them to a model. The calculations factor in the amount of time elapsed between frames, so the physics will be robust against changes in framerate.

To enable a node for physics, attach an ActorNode to it. The ActorNode keeps track of which NodePath it is attached to, and will change the position and orientation of that NodePath as physics is applied to the ActorNode.

When an ActorNode is created, it must also be attached to a PhysicsManager. The PhysicsManager will handle the physics calculations every frame and notify the ActorNode of any changes it needs to apply to its parent NodePath. Panda provides a default physics manager, base.physicsMgr, which will often be suitable for most applications.

```
jetpackGuy=loader.loadModel("models/jetpack_guy")
jetpackGuy.reparentTo(render)
an=ActorNode("jetpack-guy-physics")
anp=jetpackGuy.attachNewNode(an)
base.physicsMgr.attachPhysicalNode(an)
```

Now, the "jetpack guy" model will be updated every frame with the physics applied to it.

The ActorNode also serves as a repository for the PhysicsObject that describes the physical properties (i.e. mass) of the object. To modify these properties, use the getPhysicsObject call.

```
an.getPhysicsObject().setMass(136.077) #about 300
lbs
```

Panda3D Manual: Applying physics to a node

To apply forces to a physical object, collect them into a ForceNode and then apply them to the object. The ForceNode is a node that specifies the "context" of the force; i.e. the local coordinate transform that determines the direction of the force. Because ForceNodes are separate from ActorNodes, a ForceNode can be placed in a different portion of the model tree from the ActorNode to which the forces applies. This allows for forces to be applied indirectly to a model (such as wind sweeping across the scene, or a mechanical impulse from an appendage of the model) without having to do the calculations necessary to transform from the ActorNode's coordinates to the coordinates of the force's source.

To add a force to a physical object, add the force using either the addLinearForce method (for translational forces) or the addAngularForce method (for rotational forces):

```
actorNode.addLinearForce(pusherForce)
actorNode.addAngularForce
(spinnerForce)
```

Conversely, forces can be removed using the corresponding remove calls:

```
actorNode.removeLinearForce(pusherForce)
actorNode.removeAngularForce
(spinnerForce)
```

By default, linear forces don't factor in the mass of the object upon which they act (meaning they are more like accelerations). To factor in the mass of the object when applying the linear force, use the following call to enable mass-dependent calculations:

```
pusherForce.setMassDependent
(1)
```

Example 1: Gravity

To apply a gravitational pull to the "jetpack guy" from the previous example:

```
gravityFN=ForceNode('world-forces')
gravityFNP=render.attachNewNode(gravityFN)
gravityForce=LinearVectorForce(0,0,-9.8) #gravity acceleration
gravityFN.addForce(gravityForce)

base.physicsManager.addLinearForce(gravityForce)
```

Since the gravitational force is relative to the entire world (and shouldn't change if, for example, the jetpack guy tumbles head-over-heels), the gravityForce vector was added to a ForceNode attached to render. So regardless of the orienation of the NodePath controlled by an, the force will always pull towards the bottom of the scene.

Since all objects in the scene should be affected by gravity, the force was added to the set of forces managed by the PhysicsManager itself. Since forces ignore the mass of the objects they act upon by default, this force will pull all objects towards the ground at standard gravitational acceleration. The next example shows how to apply a force to a single object.

Example 2: Rotary Thruster

Here is another example of applying forces to objects and the way in which the ForceNode alters the effect:

```
thruster=NodePath("thruster") # make a thruster for the jetpack
thruster.reparentTo(jetpackGuy)
thruster.setPos(0,-2,3)

thrusterFN=ForceNode('jetpackGuy-thruster') # Attach a thruster force
thrusterFNP=thruster.attachNewNode(thrusterFN)
thrusterForce=LinearVectorForce(0,0,4000)
thrusterForce.setMassDependent(1)
thrusterFN.addForce(thrusterForce)

an.getPhysical(0).addLinearForce(thrusterForce)

thruster.setP(-45) # bend the thruster nozzle out at 45 degrees
```

When this force is applied to the jetpack guy, it will push upwards and forwards. If the thruster's pitch and roll were controlled (say, by a joystick), then the jetpack could be moved around merely by changing the pitch and roll values; the ForceNode would inherit the orientation of the thruster and automatcially change the direction it pushes.

The effect that this thruster force has upon the jetpack guy should be dependent upon the mass of the system, so the setMassDependent call is used to factor mass into the acceleration analysis.

Panda3D Manual: Types of forces

Panda3D provides several types of forces that you can apply to an object.

LinearVectorForce

A LinearVectorForce accelerates the center of mass of an object along a straight line. The direction of the line is determined by the relative orientations of the NodePath controlled by the ActorNodea and the ForceNode to which the LinearVectorForce is assigned.

```
lvf=LinearVectorForce(1,0,0) # push 1 newton in the positive-x direction
forceNode.addForce(lvf) # determine coordinate space of this force node
actorNode.getPhysical(0).addLinearForce(lvf)
```

AngularVectorForce

The AngularVectorForce changes the angular momentum of the object to which it is applied. Note that angular forces are applied to an object using the addAngularForce method, not the addLinearForce method used above.

```
avf=AngularVectorForce(1,0,0) # spin around the positive-x axis
forceNode.addForce(avf) #determine which positive-x axis we use for calculation
actorNode.getPhysical(0).addAngularForce(avf)
```

Editorial note: The linear vector force is the only one I have worked with as of right now. If anyone could flesh out this section with more detail or more forces, it would be greatly appreciated!

Panda3D Manual: Notes and caveats

Here are some caveats, quirks, and behaviors to be aware of when working with the physics engine:

1. You can add the same force to an object multiple times with repeated calls to addLinearForce or addAngularForce. The result will be that the total effect will be the effect of the force applied once times the number of times it is applied. Note, however, that to remove the force's effect on the object, you must call remove*Force the same number of times add*Force was called; each call to remove only removes one instance of the force. Of course, it is more efficient to use a single force with magnitude (n x # of copies) than to use the same force multiple times.

2. If a NodePath that is controlled by an ActorNode also needs collision calculations done upon it, be sure to use the PhysicsCollisionHandler instead of CollisionHandlerPusher. More info can be found in the section on Collision Handlers. If you intend to use a PhysicsCollisionHandler to prevent a model from falling through a floor (for example, if the scene has gravity applied), be sure to look at the friction coefficient options on the PhyicsCollisionHandler.

Panda3D Manual: Motion Paths

Motion paths in Panda3D are splines created by a modeler that are then exported to egg files. These egg files are then imported into a program, and various nodes can then use the motion path for complex movement. A viable egg file for a motion path has the â€œcurveâ€• tag.

First, the Mopath and MopathInterval modules must be loaded. While motion paths come with their own play functions, a motion path interval allows for more functionality.

```
from direct.directutil import Mopath
from direct.interval.MopathInterval import
*
```

With the modules loaded, the motion path is loaded much like an actor is loaded. A NodePath is created with the knowledge that it will be used for a motion path, and then the file is loaded.

```
myMotionPathName = Mopath.Mopath()
myMotionPathName.loadFile('File
Path')
```

Finally, the motion path interval may be created, and played like any interval can. The interval requires not only the name of the motion path, but also the NodePath that will be affected by it.

```
myInterval = MopathInterval(myMotionPathName,myNodePath,
name='Name')
```

Panda3D Manual: Timing

While the python time module can do a decent job of timing, panda has a built in timing system that allows for lag and cpu stutter.

Panda3D Manual: The Global Clock

The global clock is import into the global namespace when you start up panda

to get the time(in seconds) since the last frame was drawn:

```
globalClock.getDt()
```

another useful function is the time (in seconds since the program started

```
globalClock.getFrameTime()
```

Panda3D Manual: Networking

Panda3D contains support for networked games. This includes both a low-level stream based API, and a higher level distributed object API. The documentation in this section assumes some familiarity with the basic concepts of networking in geneal, and the IP protocol in particular.

The documentation on these features is still in development. Read the forums for the most up-to-date information.

Panda3D Manual: Datagram Protocol

Underpinning Panda's networking capabilities are the classes that compose the datagram protocol. These classes allow for developer-defined packets to be transmitted using either the UDP or TCP protocols. Panda's datagram layer can serve as a solid foundation for developing higher-level networking abstractions.

This section describes the classes used to establish a connection (QueuedConnectionManager, QueuedConnectionListener, QueuedConnectionReader, and ConnectionWriter), as well as the classes that transmit information (NetDatagram, PyDatagram, and PyDatagramIterator).

Panda3D Manual: Client-Server Connection

The first step in network communication is to establish the client-server connection. This entails two sets of operations: one for the server side (which listens for incoming connections), and one for the client side (which establishes a connection to the server). Both of these processes are described below.

Preparing the server for connection

An average Panda program acting as a server will need to create four classes:

1. A QueuedConnectionManager, which handles the low-level connection processes, establishes connections, and handles unexpected network termination
2. A QueuedConnectionListener, which waits for clients to request connection
3. A QueuedConnectionReader, which buffers incoming data from an active connection
4. A ConnectionWriter, which allows PyDatagrams to be transmitted out along an active connection

The first step is to instantiate these four classes.

```
cManager = QueuedConnectionManager()
cListener = QueuedConnectionListener(self.cManager, 0)
cReader = QueuedConnectionReader(self.cManager, 0)
cWriter = ConnectionWriter(self.cManager,0)

activeConnections=[] # We'll want to keep track of these
later
```

This method of instantiation prepares the classes in single-thread mode, which that realtime communication requires them to be polled periodically.

To accept client connections, the server opens a special "rendezvous" socket at a specific port address. This port address must be known by both the client and the server. Additionally, a backlog is speicified; this is the number of incoming connection requests that the connection will track before it starts rejecting connection attempts. The responsibility for managing the rendezvous socket is passed to the QueuedConnectionListener, and a task is spawned to periodically poll the listener.

```
port_address=9099 #No-other TCP/IP services are using this port
backlog=1000 #If we ignore 1,000 connection attempts, something is wrong!
tcpSocket = cManager.openTCPServerRendezvous(port_address,backlog)

cListener.addConnection(tcpSocket)
```

Sicne the network handlers we instantiated are polled, we'll create some tasks to do the polling.

```
taskMgr.add(tskListenerPolling,"Poll the connection listener",-39)
taskMgr.add(tskReaderPolling,"Poll the connection reader",-40)
```

When a connection comes in, the tskListenerPolling function below handles the incoming connection and hands it to the QueuedConnectionReader. The connection is now established.

```
def tskListenerPolling(taskdata):
  if cListener.newConnectionAvailable():

    rendezvous = PointerToConnection()
    netAddress = NetAddress()
    newConnection = PointerToConnection()

    if cListener.getNewConnection(rendezvous,netAddress,newConnection):
      newConnection = newConnection.p()
      activeConnections.append(newConnection) # Remember connection
      cReader.addConnection(newConnection) # Begin reading connection
  return Task.cont
```

Once a connection has been opened, the QueuedConnectionReader may begin processing incoming packets. This is similar to the flow of the listener's task, but it is up to the server code to handle the incoming data.

```
def tskReaderPolling(taskdata):
  if cReader.dataAvailable():
    datagram=NetDatagram() # catch the incoming data in this instance
    # Check the return value; if we were threaded, someone else could have
    # snagged this data before we did
    if cReader.getData(datagram):
      myProcessDataFunction(datagram)
  return Task.cont
```

Note that the QueuedConnectionReader retrieves data from all clients connected to the server. The NetDatagram can be queried using NetDatagram.getConnection to determine which client sent the message.

If the server wishes to send data to the client, it can use the ConnectionWriter to transmit back along the connection.

```
 # broadcast a message to all clients
myPyDatagram=myNewPyDatagram() # build a datagram to send
for aClient in activeConnections:
  cWriter.send(myPyDatagram,aClient)
```

Finally, the server may terminate a connection by removing it from the QueuedConnectionReader's responsibility. It may also deactivate its listener so that no more connections are received

```
 # terminate connection to all clients

for aClient in activeConnections:
  cReader.removeConnection(aClient)
activeConnections=[]

 # close down our listener
cManager.closeConnection(tcpSocket)
```

Connecting with a client

The process the client undertakes to connect to a server is extremely similar to the process the server undertakes to receive connections. Like the server, a client instantiates a QueuedConnectionManager, QueuedConnectionReader, and ConnectionWriter. However, there are some differences in the process. In general, a client has no need to open a rendezvous socket or create a QueuedConnectionListener, since it will be doing the connecting itself. Instead, the client connects to a specific server by specifying the server's IP address and the correct socket ID.

```
port_address=9099 # same for client and server

 # a valid server URL. You can also use a DNS name
 # if the server has one, such as "localhost" or "panda3d.org"
ip_address="192.168.0.50"

 # how long until we give up trying to reach the server?
timeout_in_miliseconds=3000 # 3 seconds

myConnection=cManager.openTCPClientConnection(ip_address,port_address,
timeout_in_miliseconds)
if myConnection:
  cReader.addConnection(myConnection) # receive messages from server
```

When the client has finished communicating with the server, it can close the connection.

```
cManager.closeConnection
(myConnection)
```

Panda3D Manual: Transmitting Data

Once a connection has been established, data can be transmitted from one Panda program to another using the classes described in this section. Communication can happen in both directions (i.e. client-to-server or server-to-client); once the connection has been established, either side may send messages along the connection to the other side.

This section describes message passing in detail, first transmission, then receipt of a message.

Sending a message

To send a message along an established connection, the sender must first construct a PyDatagram containing the message. This involves instantiating a PyDatagram object and then populating its contents with the desired data. The type of the data is determined by the functions used to pack it; see the full documentation of the PyDatagram class for more details.

```
 #Developer-defined constants, telling the server what to do.
 #Your style of how to store this information may differ; this
is
 #only one way to tackle the problem
PRINT_MESSAGE=1

def myNewPyDatagram(self):
  # send a test message
  myPyDatagram=PyDatagram()
  myPyDatagram.addUInt8(PRINT_MESSAGE)
  myPyDatagram.addString("Hello, world!")
  return myPyDatagram
```

As shown in the previous section, once the datagram is constructed you may then send it using a ConnectionWriter.

```
cWriter.send(myPyDatagram,
aConnection)
```

Receiving a message

As shown in the previous section, when a message is received via a QueuedConnectionReader, it can be retrieved into a NetDatagram:

```
datagram=NetDatagram
if cReader.getData(datagram):
  myProcessDataFunction
(datagram)
```

A NetDatagram contains the original information that was stored in the transmitted PyDatagram. It also contains knowledge of the connection over which it was received and the address of the connection. To retrieve the connection, use the getConnection method:

```
sourceOfMessage=datagram.getConnection
()
```

To retrieve the contents of the message, use the PyDatagramIterator. The iterator class acts as the complement of the PyDatagram class; its methods can be used to retrieve the content that was encoded using PyDatagram.

```
def myProcessDataFunction(netDatagram):
  myIterator=PyDatagramIterator(netDatagram)
  msgID=myIterator.getUInt8()
  if msgID==PRINT_MESSAGE:
    messageToPrint=myIterator.getString()
    print messageToPrint
```

Note: It is assumed that the message recipient will retrieve the same type of content in the same order that the message sender packed the content. No mechanism exists in the PyDatagramIterator to ensure that the data being unpacked matches the requested type. Unpacking the data using a different type function will probably result in unexpected behavior.

Panda3D Manual: The Python Debugger

Python is a very powerful interactive and interpreted language. Python's development cycle is very fast. Often the most effective way to to debug is to output relevant information. Having said that, there are many ways to enhance productivity with knowledge of good debugging techniques.

Using python -i mode

Python programs may be developed and tested with the help of the interactive mode of the Python interpreter, which, allows program components to be debugged, traced, profiled, and tested interactively. When invoking python with -i, this ensures that an interactive session of python is invoked. With regards to Panda3D this requires a little explanation. Panda3D programs typically have a command called run() to start rendering, so here's one way to start an interactive session. On the command prompt type:

```
ppython -i myPandaFile.
py
```

After Panda3D has loaded, make sure the command window has focus and type Ctrl-C. This will show a python prompt like so:

```
>>>
```

Now on the command prompt you can execute any python commands, related or unrelated to your Panda3D program. This is useful for looking at information at that specific point in time. You could even change that information for that running instance of the program.

pdb

Python offers hooks enabling interactive debugging. Module pdb supplies a simple text-mode interactive debugger. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger's prompt is "(Pdb) ". There are many ways to enter the debugger. Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import <mymodule>
>>> pdb.run('mymodule.test
()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
<string>(1)?()
(Pdb)
```

Detailed information about pdb can be found here. In addition to pdb, python also has two modules called `inspect` and `traceback`. `inspect` supplies functions to extract information from all kinds of objects, including the Python call stack and source files. The `traceback` module lets you extract, format and output information about tracebacks as normally produced by uncaught exceptions.

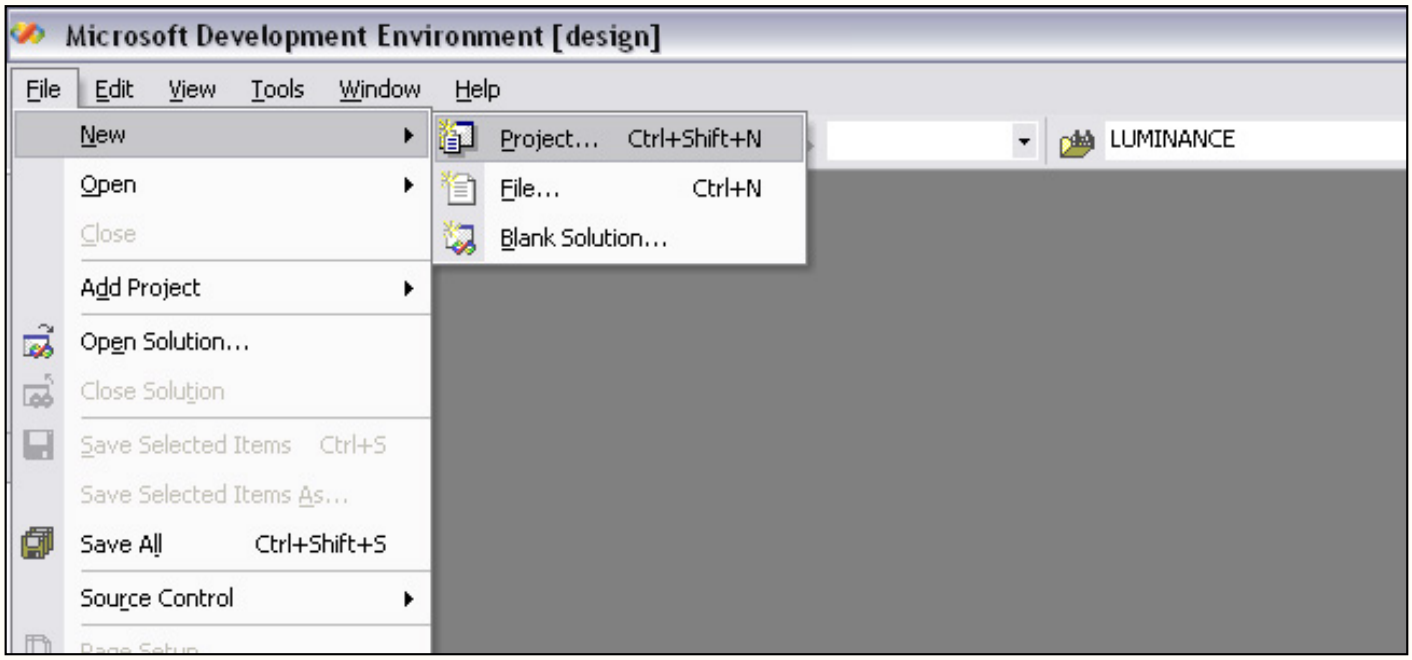
<<prev top next>> Search

Panda3D Manual: Running Panda under the CXX Debugger

These instructions show how to compile and run panda from inside the Microsoft Visual Studio debugger.

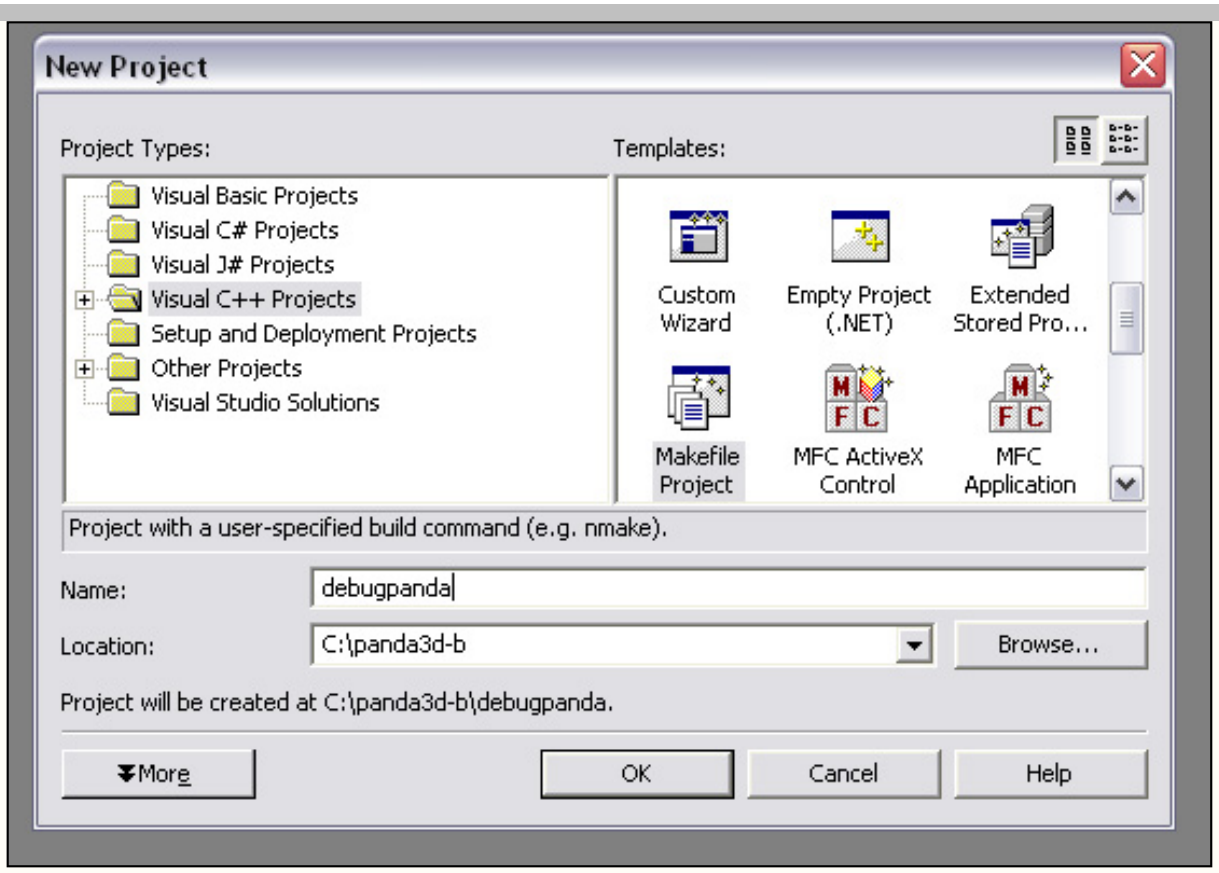
Before you Begin

The first step is to download the panda source code and compile it. The instructions can be found in the section Building Panda from Source. Be sure to compile with the Optimize setting of 1, otherwise, Visual Studio will not be able to debug properly. Once you have compiled panda, start up visual studio, and ask it to create a new project:

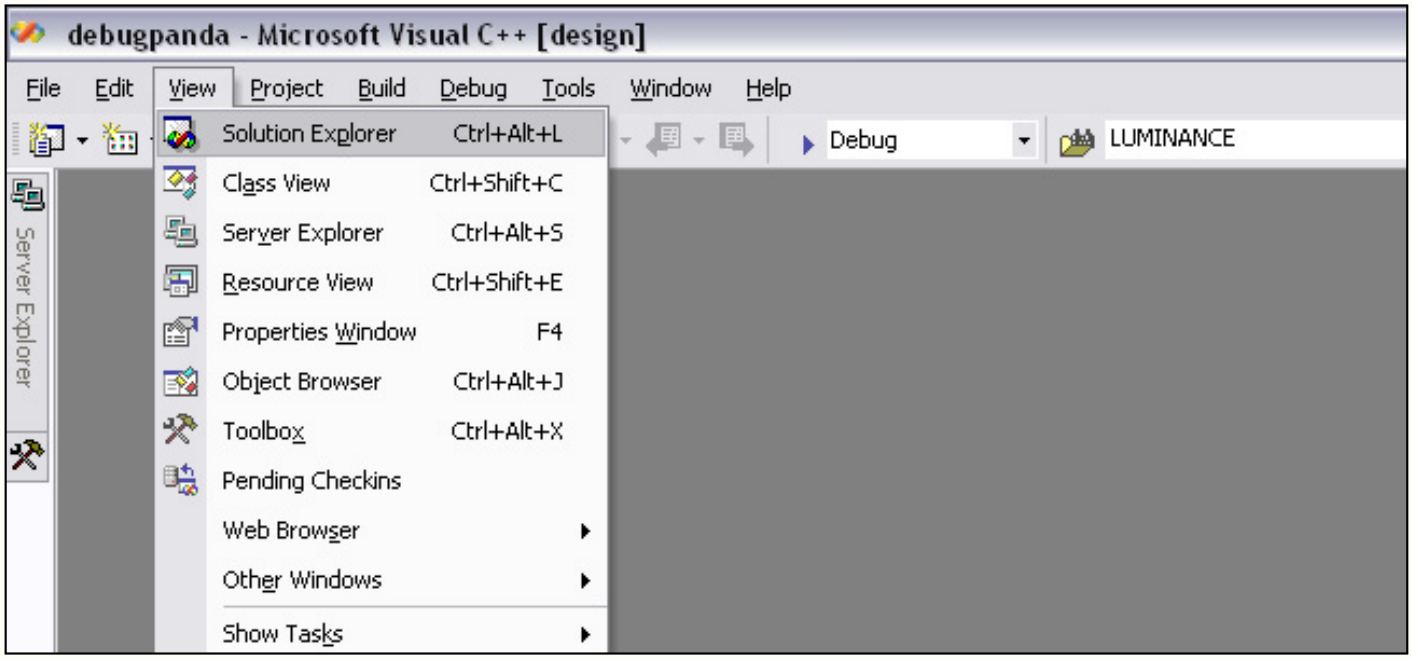


There are four pieces of information you need to enter into the new project dialog:

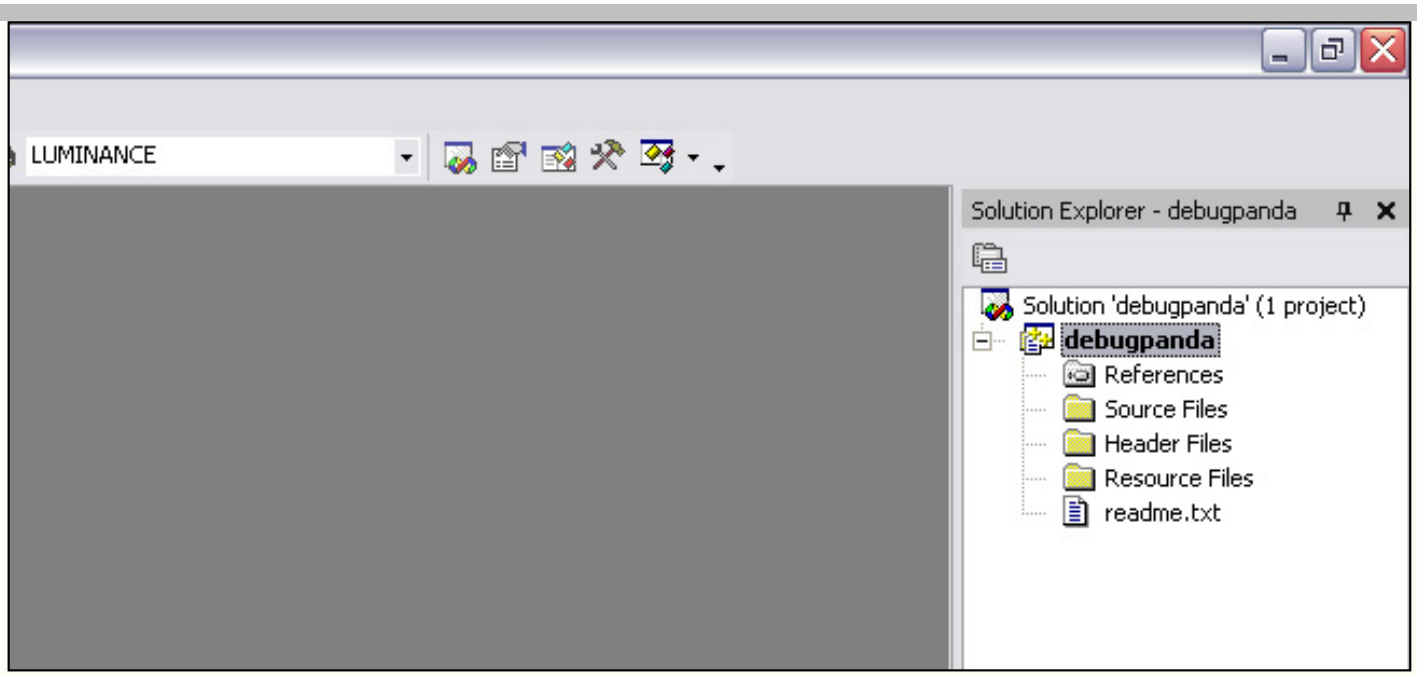
- Tell it to make a "C++ project"
- Tell it to use a "Makefile"
- Tell it the name of the project, "debugpanda"
- Tell it where the panda source tree is (ie, "c:\panda3d-b")



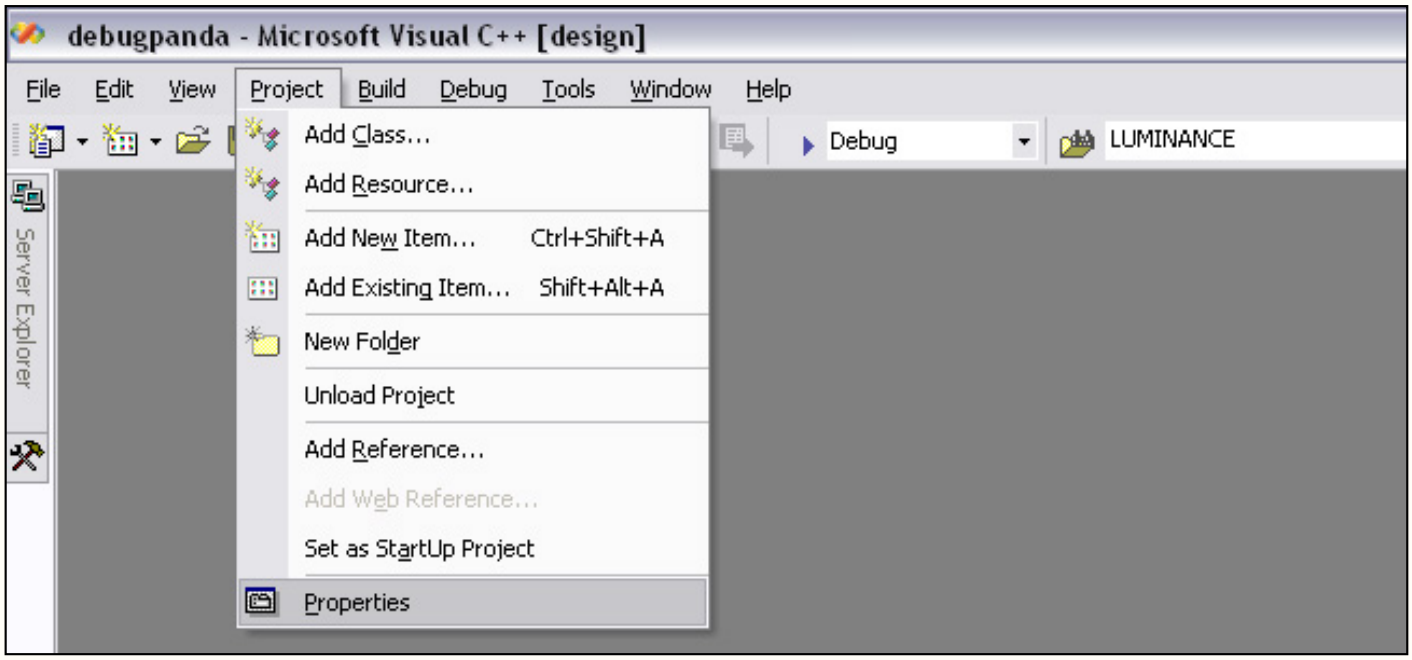
Click ok, and confirm. You have now created the project, and the project is open for editing. You now need to access the "solution explorer:"



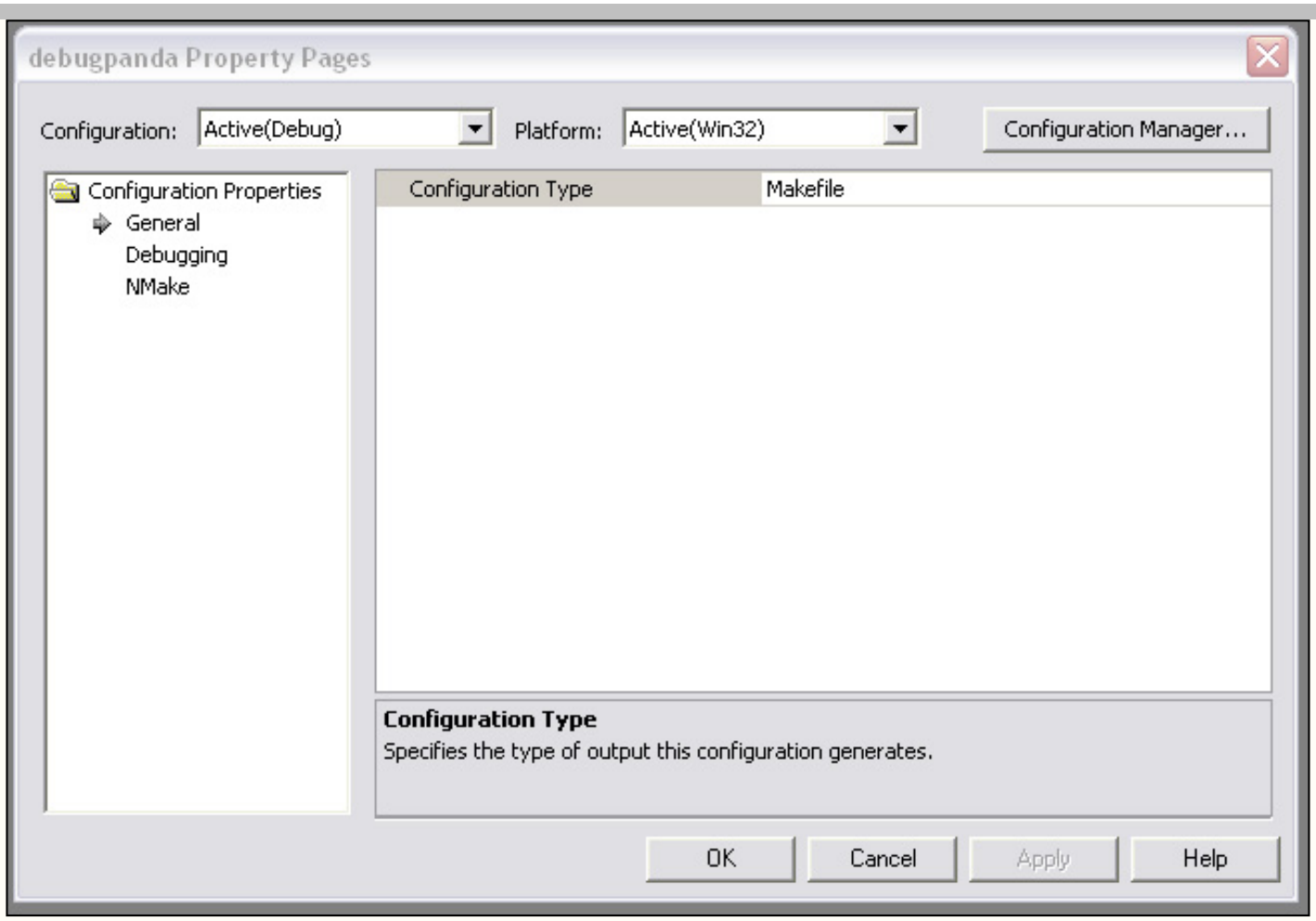
Normally, you can see the solution explorer in the upper-right corner of visual studio. It should show your project name (debugpanda). The word "debugpanda" needs to be highlighted - if it is not, click on it:



Now that your project is selected, you can edit its project properties:

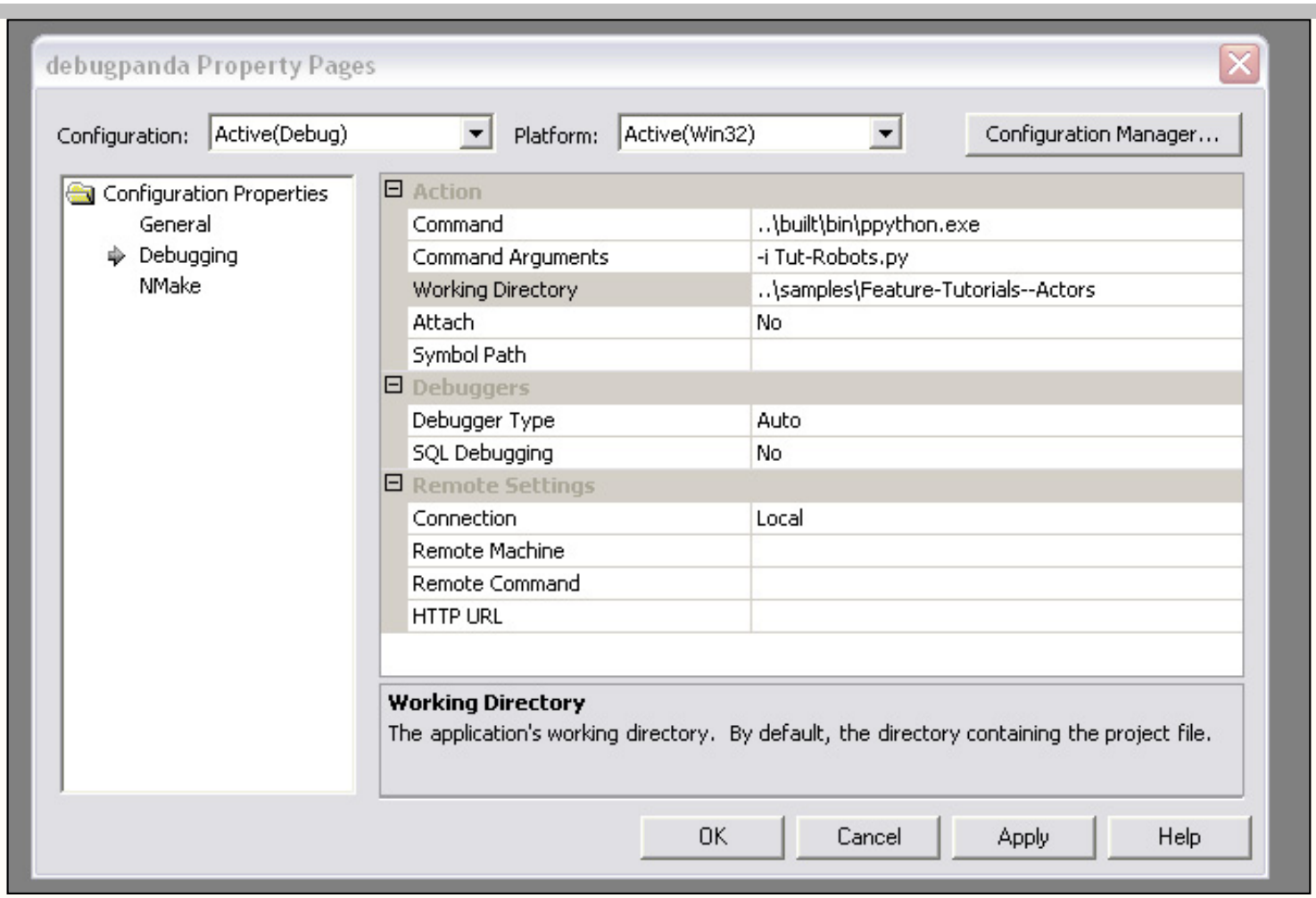


The project property dialog initially looks like this. It contains three subpanels, the "General" panel, the "Debugging" panel, and the "NMake" subpanel. You can see these three subheadings in the left pane:

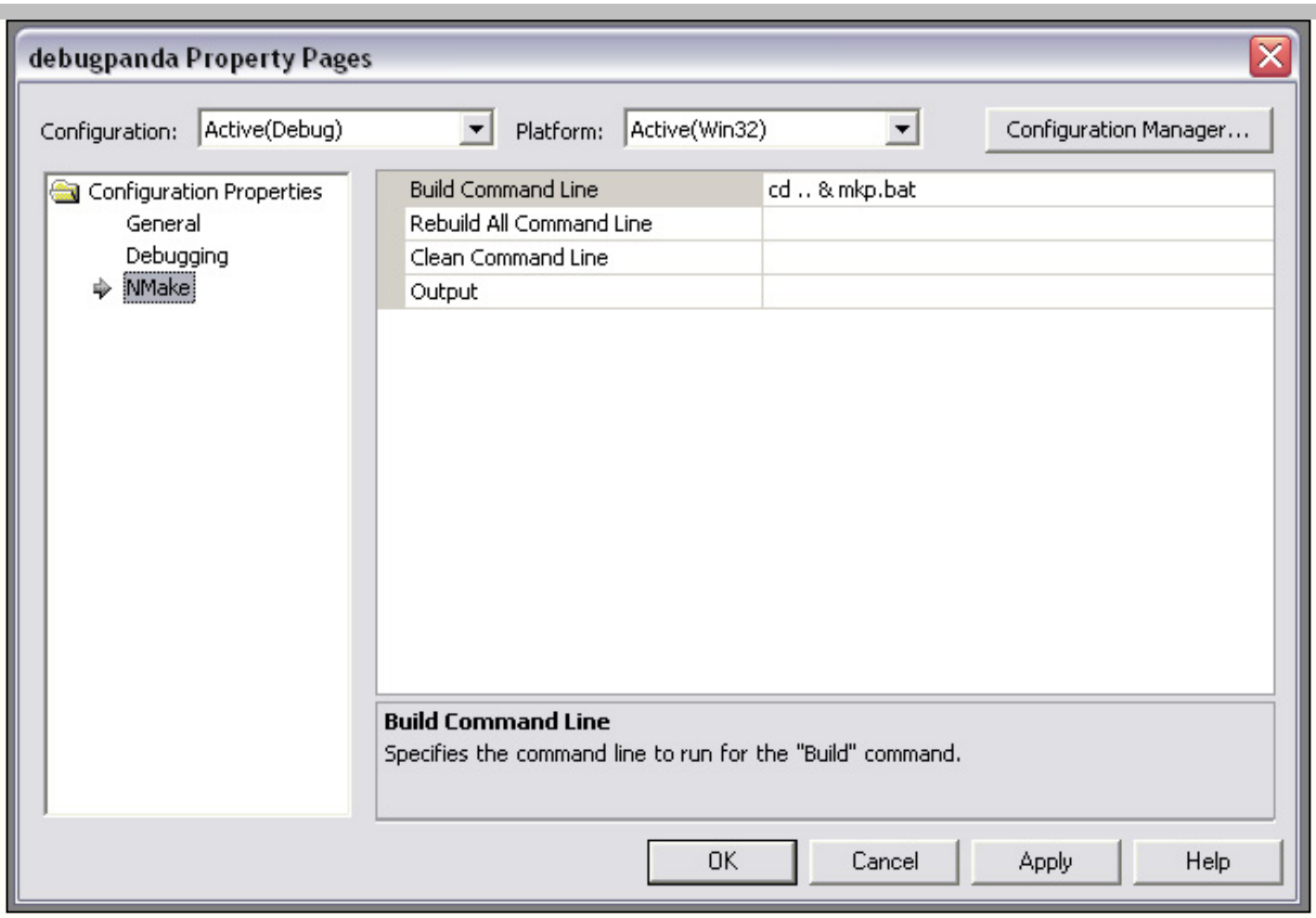


There is nothing to fill in on the general panel, so switch to the debugging panel. You need to fill in the command name, the command arguments, and the working directory. For now, we will ask it to debug the Actors/Robots sample program. Since visual studio puts the project file in a subdirectory, the paths need to be preceded by ".." to get to the root of the panda source tree:

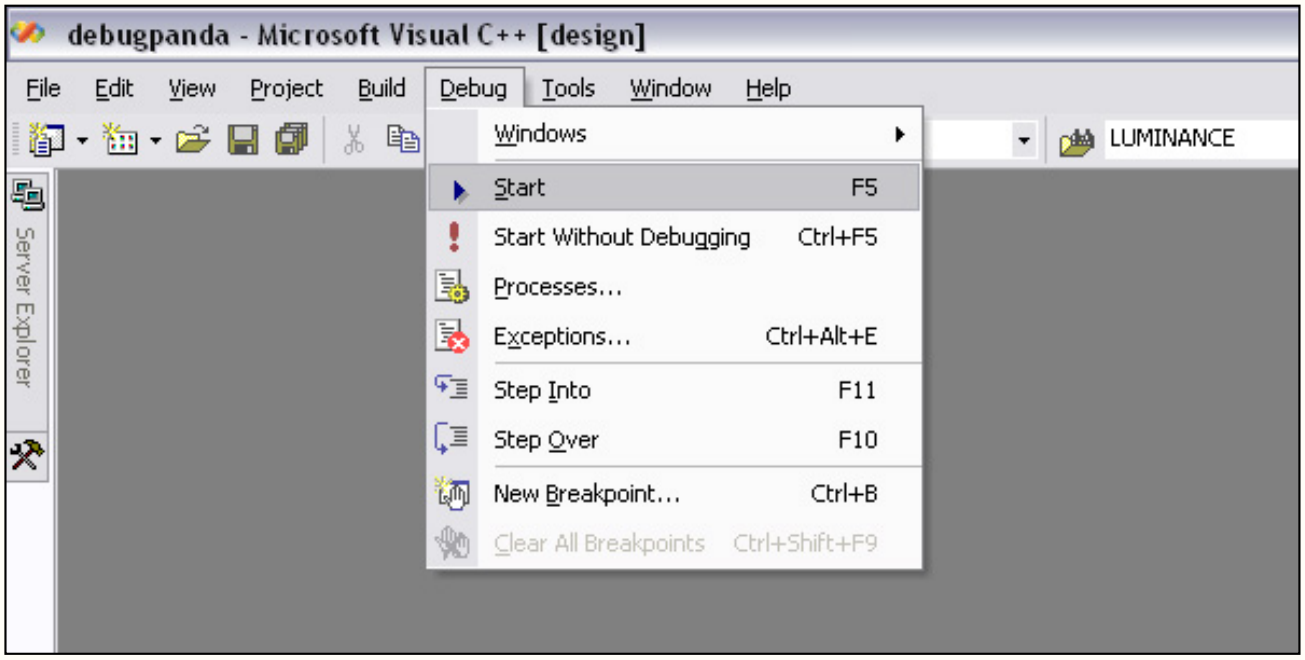
WARNING: these instructions are for panda 1.2.2 and beyond. In panda 1.2.1 and before, you need to debug "built \python\python.exe", NOT "built\bin\ppython.exe."



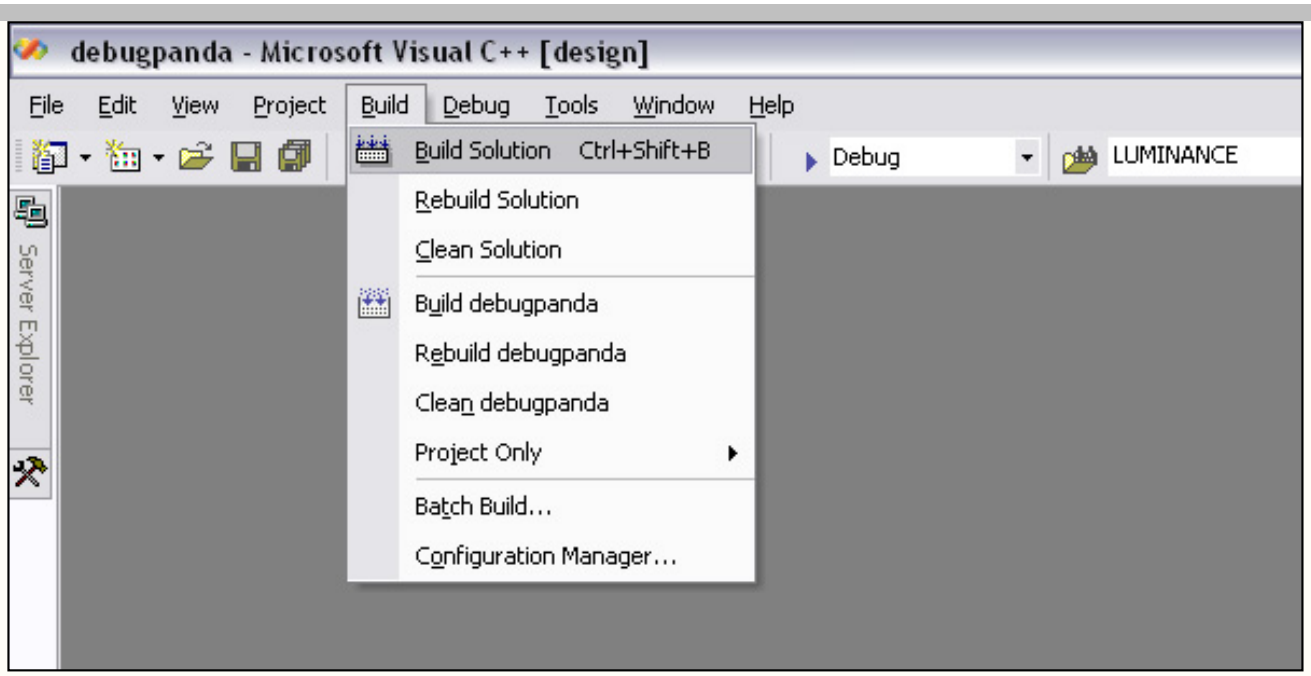
Finally, switch to the "NMake" panel. Here, you can tell it what the command is to recompile panda. I use a bat file "mkp.bat" to compile panda. Since the project file is in a subdirectory, the command needs to be preceded by "cd .." in order to get back to the root of the panda source tree:



Visual studio now knows how to run panda, and how to compile it. You can run your program (in this case, the "Actors/ Robots" tutorial) by clicking on the Debug menu:



You can rebuild panda at any time by clicking on the "Build" menu:



Now that you are running in the debugger, you can open any panda source file and set a breakpoint, or examine data. Of course, it may be advantageous to learn how to use the python debugger as well as the C++ debugger.

Panda3D Manual: Log Messages

Panda periodically outputs log messages for debugging purposes. A typical log message might look like this:

```
:util(warning): Adjusting global clock's real time by -3.3
seconds.
```

The first part of the message, `util`, is the name of the module that generated the message. The second part, `warning`, indicates the severity. The severity levels are, in decreasing order: fatal, error, warning, info, debug, and spam. The panda configuration file (Config.prc) contains these directives:

```
notify-level warning
default-directnotify-level
warning
```

Directives like these tell panda which messages to show, and which to suppress. In the default configuration (shown above), all messages whose severity is `warning` or above are shown, all messages whose severity is less are suppressed.

It is interesting and educational to change the configuration to this:

```
notify-level spam
default-directnotify-level
spam
```

If you do this, panda will print out vast amounts of information while it runs. These informational messages can be useful for debugging. However, there are so many print-statements that it slows panda down to a crawl. So it may be desirable to tell panda to narrow it down a little. The way to do that is to name a particular module in the panda config file. For example, you might do this:

```
notify-level warning
notify-level-glgsg spam
default-directnotify-level
warning
```

This tells panda that module "glgsg" should print out everything it can, but that every other

module should only print warnings and errors. By the way, module `glgsg` is a particularly interesting module to investigate. This is the module that invokes OpenGL. If you tell it to spam you, it will tell you what it's setting the MODELVIEW and PROJECTION matrices to, and lots of other interesting information.

Generating your own Log Messages

You can use the `Notify` class to output your own log messages.

FINISH THIS SECTION.

Redirecting Log Messages to a File

If you wish, you can redirect all of panda's log messages into a file. The following snippet will do the trick:

```
nout = MultiplexStream()
Notify.ptr().setOstreamPtr(nout,
0)
nout.addFile(Filename("out.txt"))
```

Panda3D Manual: Measuring Performance with PStats

QUICK INTRODUCTION

PStats is Panda's built-in performance analysis tool. It can graph frame rate over time, and can further graph the work spent within each frame into user-defined subdivisions of the frame (for instance, app, cull and draw), and thus can be an invaluable tool in identifying performance bottlenecks. It can also show frame-based data that reflects any arbitrary quantity other than time intervals, for instance, texture memory in use or number of vertices drawn.

The performance graphs may be drawn on the same computer that is running the Panda client, or they may be drawn on another computer on the same LAN, which is useful for analyzing fullscreen applications. The remote computer need not be running the same operating system as the client computer.

To use PStats, you first need to build the PStats server program, which is part of the Pandatool tree (it's called pstats.exe on Windows, and pstats on a Unix platform). Start by running the PStats server program (it runs in the background), and then start your Direct/Panda client with the following in your Config.prc file:

```
want-pstats 1
```

Or, at runtime, issue the Python command:

```
PStatClient.connect()
```

Or if you're running pview, press shift-S.

Any of the above will contact your running PStats server program, which will proceed to open a window and start a running graph of your client's performance.

If you have multiple computers available for development, it can be advantageous to run the pstats server on a separate computer so that the processing time needed to maintain and update the pstats user interface isn't taken from the program you are profiling. If you wish to run the server on a different machine than the client, start the server on the profiling machine and add the following variable to your client's Config.prc file, naming the hostname or IP address of the profiling machine:

```
pstats-host profiling-machine-ip-or-hostname
```

If you are developing Python code, you may be interested in reporting the relative time spent within each Python task (by subdividing the total time spent in Python, as reported under "Show Code"). To do this, add the following lines to your Config.prc file before you start ShowBase:

```
task-timer-verbose 1
pstats-tasks 1
```

THE PSTATS SERVER (The user interface)

The GUI for managing the graphs and drilling down to view more detail is entirely controlled by the PStats server program. At the time of this writing, there are two different versions of the PStats server, one for Unix and one for Windows, both called simply pstats. The interfaces are similar but not identical; the following paragraphs describe the Windows version.

When you run pstats.exe, it adds a program to the taskbar but does not immediately open a window. The program name is typically "PStats 5185", showing the default PStats TCP port number of 5185; see "HOW IT WORKS" below for more details about the TCP communication system. For the most part you don't need to worry about the port number, as long as server and client agree (and the port is not already being used by another application).

Each time a client connects to the PStats server, a new monitor window is created. This monitor window owns all of the graphs that you create to view the performance data from that particular connection. Initially, a strip chart showing the frame time of the main thread is created by default; you can create additional graphs by selecting from the Graphs pulldown menu.

Time-based Strip Charts

This is the graph type you will use most frequently to examine performance data. The horizontal axis represents the passage of time; each frame is represented as a vertical slice on the graph. The overall height of the colored bands represents the total amount of time spent on each frame; within the frame, the time is further divided into the primary subdivisions represented by different color bands (and labeled on the left). These subdivisions are called "collectors" in the PStats terminology, since they represent time collected by different tasks.

Normally, the three primary collectors are App, Cull, and Draw, the three stages of the graphics pipeline. Atop these three colored collectors is the label "Frame", which represents any remaining time spent in the frame that was not specifically allocated to one of the three child collectors (normally, there should not be significant time reported here).

The frame time in milliseconds, averaged over the past three seconds, is drawn above the upper right corner of the graph. The labels on the guide bars on the right are also shown in milliseconds; if you prefer to think about a target frame rate rather than an elapsed time in milliseconds, you may find it useful to select "Hz" from the Units pulldown menu, which changes the time units accordingly.

The running Panda client suggests its target frame rate, as well as the initial vertical scale of the graph (that is, the height of the colored bars). You can change the scale freely by clicking within the graph itself and dragging the mouse up or down as necessary. One of the horizontal guide bars is drawn in a lighter shade of gray; this one represents the actual target frame rate suggested by the client. The other, darker, guide bars are drawn automatically at harmonic subdvisions of the target frame rate. You can change the target frame rate with the Config.prc variable pstats-target-frame-rate on the client.

You can also create any number of user-defined guide bars by dragging them into the graph from the gray space immediately above or below the graph. These are drawn in a dashed blue line. It is sometimes useful to place one of these to mark a performance level so it may be compared to future values (or to alternate configurations).

The primary collectors labeled on the left might themselves be further subdivided, if the data is provided by the client. For instance, App is often divided into Show Code, Animation, and Collisions, where Show Code is the time spent executing any Python code, Animation is the time used to compute any animated characters, and Collisions is the time spent in the collision traverser(s).

To see any of these further breakdowns, double-click on the corresponding colored label (or on the colored band within the graph itself). This narrows the focus of the strip chart from the overall frame to just the selected collector, which has two advantages. Firstly, it may be easier to observe the behavior of one particular collector when it is drawn alone (as opposed to being stacked on top of some other color bars), and the time in the upper-right corner will now reflect just the total time spent within just this collector. Secondly, if there are further breakdowns to this collector, they will now be shown as further colored bars. As in the Frame chart, the topmost label is the name of the parent collector, and any time shown in this color represents time allocated to the parent collector that is not accounted for by any of the child collectors.

You can further drill down by double-clicking on any of the new labels; or double-click on the top label, or the white part of the graph, to return back up to the previous level.

Value-based Strip Charts

There are other strip charts you may create, which show arbitrary kinds of data per frame other than elapsed time. These can only be accessed from the Graphs pulldown menu, and include things such as texture memory in use and vertices drawn. They behave similarly to the time-based strip charts described above.

Piano Roll Charts

This graph is used less frequently, but when it is needed it is a valuable tool to reveal exactly how the time is spent within a frame. The PStats server automatically collects together all the time spent within each collector and shows it as a single total, but in reality it may not all have been spent in one continuous block of time.

For instance, when Panda draws each display region in single-threaded mode, it performs a cull traversal followed by a draw traversal for each display region. Thus, if your Panda client includes multiple display regions, it will alternate its time spent culling and drawing as it processes each of them. The strip chart, however, reports only the total cull time and draw time spent.

Sometimes you really need to know the sequence of events in the frame, not just the total time spent in each collector. The piano roll chart shows this kind of data. It is so named because it is similar to the paper music roll for an old-style player piano, with holes punched down the roll for each note that is to be played. The longer the hole, the longer the piano key is held down. (Think of the chart as rotated 90 degrees from an actual piano roll. A player piano roll plays from bottom to top; the piano roll chart reads from left to right.)

Unlike a strip chart, a piano roll chart does not show trends; the chart shows only the current frame's data. The horizontal axis shows time within the frame, and the individual collectors are stacked up in an arbitrary ordering along the vertical axis.

The time spent within the frame is drawn from left to right; at any given time, the collector(s) that are active will be drawn with a horizontal bar. You can observe the CPU behavior within a frame by reading the graph from left to right. You may find it useful to select "pause" from the Speed pulldown menu to freeze the graph on just one frame while you read it.

Note that the piano roll chart shows time spent within the frame on the horizontal axis, instead of the vertical axis, as it is on the strip charts. Thus, the guide bars on the piano roll chart are vertical lines instead of horizontal lines, and they may be dragged in from the left or the right sides (instead of from the top or bottom, as on the strip charts). Apart from this detail, these are the same guide bars that appear on the strip charts.

The piano roll chart may be created from the Graphs pulldown menu.

Additional threads

If the panda client has multiple threads that generate PStats data, the PStats server can open up graphs for these threads as well. Each separate thread is considered unrelated to the main thread, and may have the same or an independent frame rate. Each separate thread will be given its own pulldown menu to create graphs associated with that thread; these auxiliary thread menus will appear on the menu bar following the Graphs menu. At the time of this writing, support for multiple threads within the PStats graph is largely theoretical and untested.

HOW TO DEFINE YOUR OWN COLLECTORS

The PStats client code is designed to be generic enough to allow users to define their own collectors to time any arbitrary blocks of code (or record additional non-time-based data), from either the C++ or the Python level.

The general idea is to create a PStatCollector for each separate block of code you wish to time. The name which is passed to the PStatCollector constructor is a unique identifier: all collectors that share the same name are deemed to be the same collector.

Furthermore, the collector's name can be used to define the hierarchical relationship of each collector with other existing collectors. To do this, prefix the collector's name with the name of its parent(s), followed by a colon separator. For instance, PStatCollector("Draw:Flip") defines a collector named "Flip", which is a child of the "Draw" collector, defined elsewhere.

You can also define a collector as a child of another collector by giving the parent collector explicitly followed by the name of the child collector alone, which is handy for dynamically-defined collectors. For instance, PStatCollector(draw, "Flip") defines the same collector named above, assuming that draw is the result of the PStatCollector("Draw") constructor.

Once you have a collector, simply bracket the region of code you wish to time with collector.

start() and collector.stop(). It is important to ensure that each call to start() is matched by exactly one call to stop(). If you are programming in C++, it is highly recommended that you use the PStatTimer class to make these calls automatically, which guarantees the correct pairing; the PStatTimer's constructor calls start() and its destructor calls stop(), so you may simply define a PStatTimer object at the beginning of the block of code you wish to time. If you are programming in Python, you must call start() and stop() explicitly.

When you call start() and there was another collector already started, that previous collector is paused until you call the matching stop() (at which time the previous collector is resumed). That is, time is accumulated only towards the collector indicated by the innermost start() .. stop() pair.

Time accumulated towards any collector is also counted towards that collector's parent, as defined in the collector's constructor (described above).

It is important to understand the difference between collectors nested implicitly by runtime start/stop invocations, and the static hierarchy implicit in the collector definition. Time is accumulated in parent collectors according to the statically-defined parents of the innermost active collector only, without regard to the runtime stack of paused collectors.

For example, suppose you are in the middle of processing the "Draw" task and have therefore called start() on the "Draw" collector. While in the middle of processing this block of code, you call a function that has its own collector called "Cull:Sort". As soon as you start the new collector, you have paused the "Draw" collector and are now accumulating time in the "Cull:Sort" collector. Once this new collector stops, you will automatically return to accumulating time in the "Draw" collector. The time spent within the nested "Cull:Sort" collector will be counted towards the "Cull" total time, not the "Draw" total time.

Color and Other Optional Collector Properties

If you do not specify a color for a particular collector, it will be assigned a random color at runtime. At present, the only way to specify a color is to modify panda/src/pstatclient/pStatProperties.cxx, and add a line to the table for your new collector(s). You can also define additional properties here such as a suggested initial scale for the graph and, for non-time-based collectors, a unit name and/or scale factor. The order in which these collectors are listed in this table is also relevant; they will appear in the same order on the graphs. The first column should be set to 1 for your new collectors unless you wish them to be disabled by default. You must recompile the client (but not the server) to reflect changes to this table.

HOW IT WORKS (What's actually happening)

The PStats code is divided into two main parts: the client code and the server code.

The PStats Client

The client code is in panda/src/pstatclient, and is available to run in every Panda client unless it is compiled out. (It will be compiled out if OPTIMIZE is set to level 4, unless DO_PSTATS is also explicitly set to non-empty. It will also be compiled out if NSPR is not available, since both client and server depend on the NSPR library to exchange data, even when running the server on the same machine as the client.)

The client code is designed for minimal runtime overhead when it is compiled in but not enabled (that is, when the client is not in contact with a PStats server), as well as when it is enabled (when the client is in contact with a PStats server). It is also designed for zero runtime overhead when it is compiled out.

There is one global PStatClient class object, which manages all of the communications on the client side. Each PStatCollector is simply an index into an array stored within the PStatClient object, although the interface is intended to hide this detail from the programmer.

Initially, before the PStatClient has established a connection, calls to start() and stop() simply return immediately.

When you call PStatClient.connect(), the client attempts to contact the PStatServer via a TCP connection to the hostname and port named in the pstats-host and pstats-port Config.prc variables, respectively. (The default hostname and port are localhost and 5185.) You can also pass in a specific hostname and/or port to the connect() call. Upon successful connection and handshake with the server, the PStatClient sends a list of the available collectors, along with their names, colors, and hierarchical relationships, on the TCP channel.

Once connected, each call to start() and stop() adds a collector number and timestamp to an array maintained by the PStatClient. At the end of each frame, the PStatClient boils this array into a datagram for shipping to the server. Each start() and stop() event requires 6 bytes; if the resulting datagram will fit within a UDP packet (1K bytes, or about 84 start/stop pairs), it is sent via UDP; otherwise, it is sent on the TCP channel. (Some fraction of the packets that are eligible for UDP, from 0% to 100%, may be sent via TCP instead; you can specify this with the pstats-tcp-ratio Config.prc variable.)

Also, to prevent flooding the network and/or overwhelming the PStats server, only so many frames of data will be sent per second. This parameter is controlled by the pstats-max-rate Config.prc variable and is set to 30 by default. (If the packets are larger than 1K, the max transmission rate is also automatically reduced further in proportion.) If the frame rate is higher than this limit, some frames will simply not be transmitted. The server is designed to cope with missing frames and will assume missing frames are similar to their neighbors.

The server does all the work of analyzing the data after that. The client's next job is simply to clear its array and prepare itself for the next frame.

The PStats Server

The generic server code is in pandatool/src/pstatserver, and the GUI-specific server code is in pandatool/src/gtk-stats and pandatool/src/win-stats, for Unix and Windows, respectively. (There is also an OS-independent text-stats subdirectory, which builds a trivial PStats server that presents a scrolling-text interface. This is mainly useful as a proof of technology rather than as a usable tool.)

The GUI-specific code is the part that manages the interaction with the user via the creation of windows and the handling of mouse input, etc.; most of the real work of interpreting the data is done in the generic code in the pstatserver directory.

The PStatServer owns all of the connections, and interfaces with the NSPR library to communicate with the clients. It listens on the specified port for new connections, using the pstats-port Config.prc variable to determine the port number (this is the same variable that specifies the port to the client). Usually you can leave this at its default value of 5185, but there may be some cases in which that port is already in use on a particular machine (for instance, maybe someone else is running another PStats server on another display of the same machine).

Once a connection is received, it creates a PStatMonitor class (this class is specialized for each of the different GUI variants) that handles all the data for this particular connection. In the case of the windows pstats.exe program, each new monitor instance is represented by a new toplevel window. Multiple monitors can be active at once.

The work of digesting the data from the client is performed by the PStatView class, which analyzes the pattern of start and stop timestamps, along with the relationship data of the various collectors, and boils it down into a list of the amount of time spent in each collector per frame.

Finally, a PStatStripChart or PStatPianoRoll class object defines the actual graph output of colored lines and bars; the generic versions of these include virtual functions to do the actual drawing (the GUI specializations of these redefine these methods to make the appropriate calls).

Panda3D Manual: Graphics Card Performance

If a scene needs be rendered and has multiple nodes, Panda has to send each node to the graphics hardware as a separate batch of polygons (because the nodes might move independently, or have different state changes on them). Modern graphics hardware hasn't made any improvements recently in handling large numbers of batches, just in handling large numbers of polygons per batch. So if a scene is composed of a large number of nodes with a small number of polygons per node, the frame rate will suffer. This problem is not specific to Panda; any graphics engine will have the same problem. The problem due to the nature of the PC and the AGP bus.

For example, though your graphics card may claim it can easily handle 100,000 polygons, this may be true in practice only if all of those polygons are sent in one batch--that is, just a single Geom. If, however, your scene consists of 1,000 nodes with 100 polygons each, it may not have nearly as good a frame rate.

To inspect performance the nodePath.analyze() method is extremely useful. For example:

```
render.analyze
()
```

The response is printed to the command window. It may look something like this:

```
371 total nodes (including 43 instances).
21 transforms; 16% of nodes have some render attribute.
205 Geoms, with 94 GeomVertexDatas, appear on 133 GeomNodes.
21665 vertices, 21573 normals, 21557 texture coordinates.
35183 triangles:
   3316 of these are on 662 tristrips (5.00906 average tris per
strip).
   0 of these are on 0 trifans.
   31867 of these are independent triangles.
0 lines, 0 points.
99 textures, estimated minimum 326929K texture memory required.
```

For a scene with many static nodes there exists a workaround.

If a scene is composed of many static objects, for example boxes, and the intent of all of these boxes to just sit around and be part of the background, or to move as a single unit, they can flattened together into a handful of nodes (or even one node). To do this, parent them all to the same node, and use:

```
node.flattenStrong
()
```

One thing that flattenStrong() won't touch is geometry under a ModelRoot or ModelNode node. Since each egg or bam file loads itself up under a ModelRoot node, the proper way to handle this is to get rid of that node first to make the geometry from multiple different egg files to be flattened together. This can be done with the following:

```
modelRoot = loader.loadModel('myModel.egg')
newModel = NodePath('model')
modelRoot.getChildren().reparentTo
(newModel)
```

<<prev top next>> Search

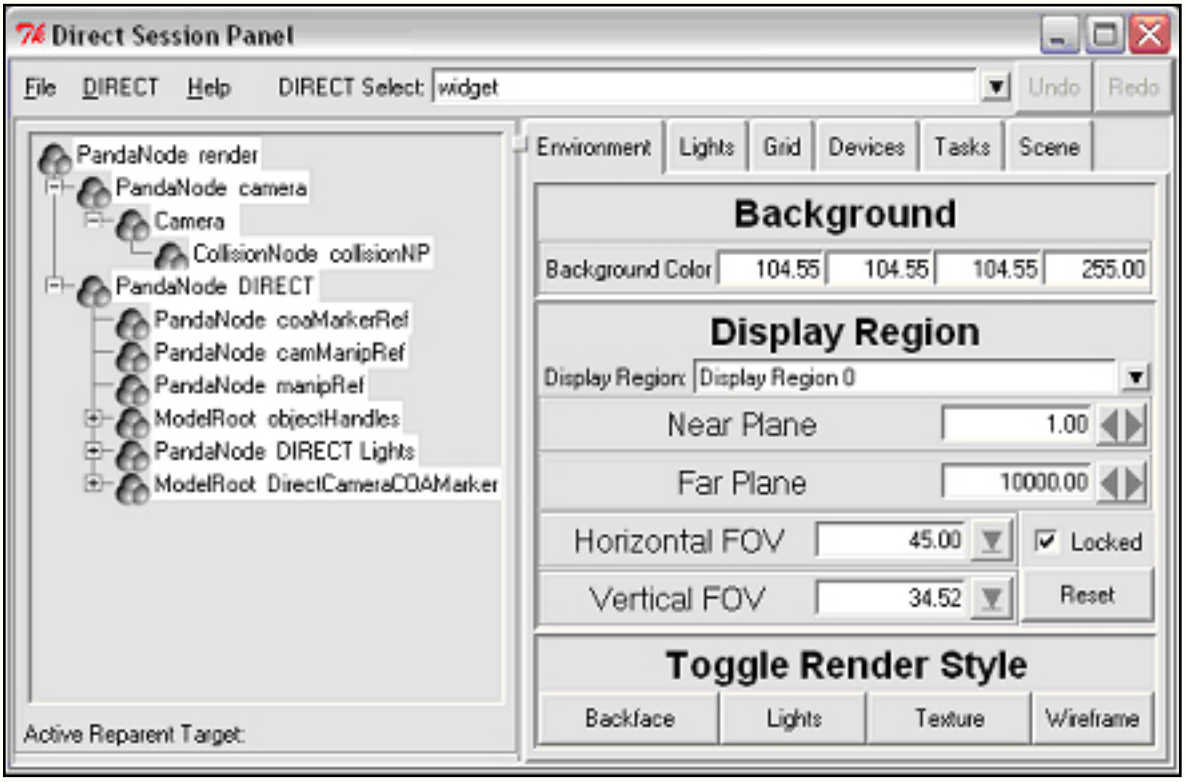
Panda3D Manual: Panda Tools

This section lists a number of tools to support programming in Panda.

Many of the tools provided are invoked by pressing a hot-key inside of the panda main window. These hot-key driven tools are called the "direct tools." The direct tools are initially disabled, because the hot-keys used to invoke them might interfere with the program's own keys and events.

To enable the hot-keys for the direct tools, you need to set the 'want-directtools' variable in config.prc, the main panda configuration file. More information about this configuration file is available in the Configuring Panda section.

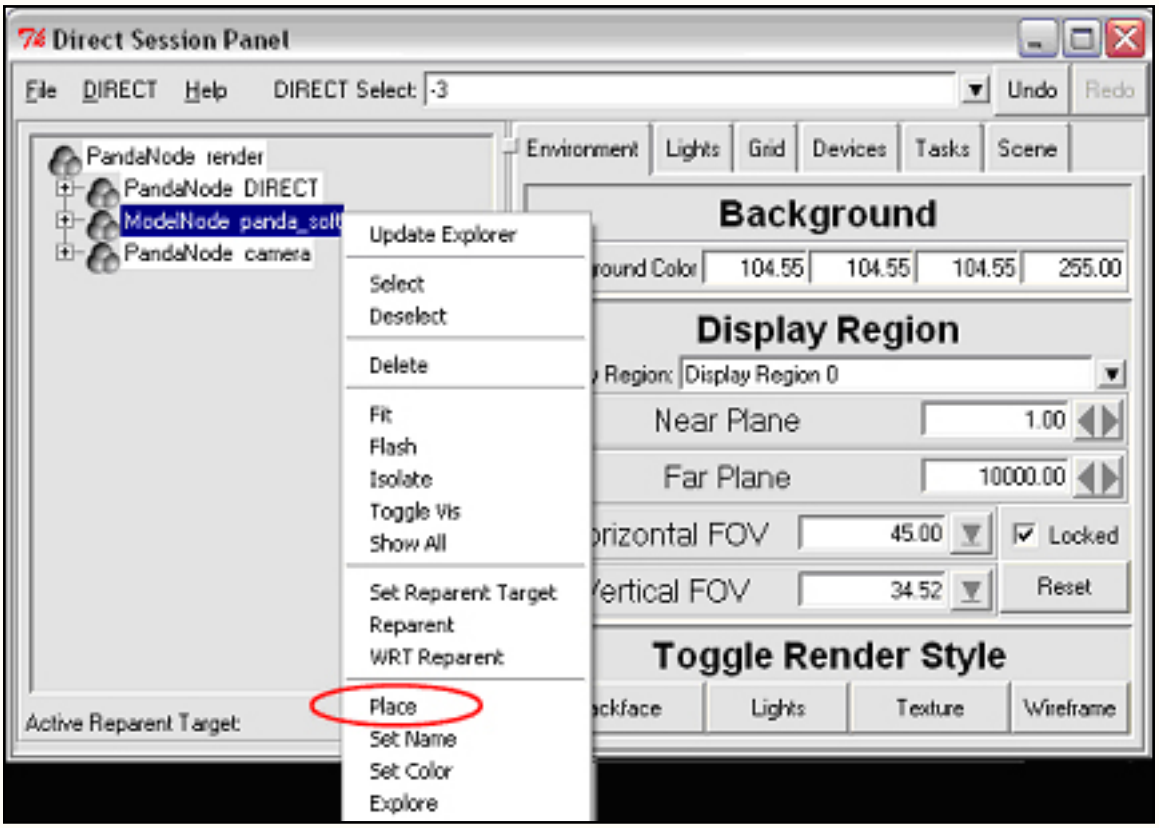
After enabling direct tools and starting panda, the Direct Session window should appear:



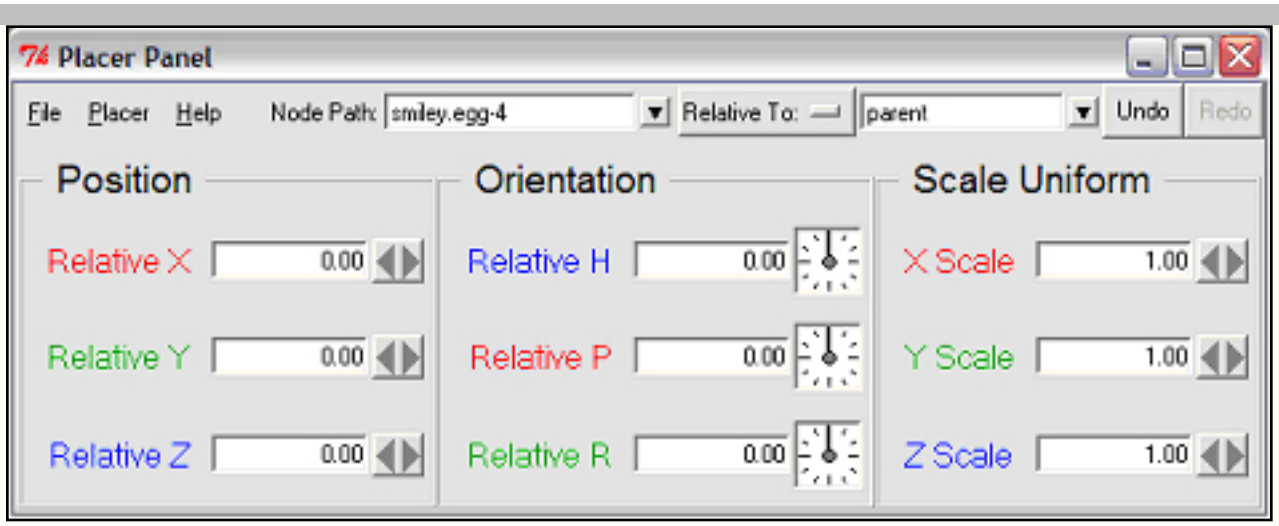
Panda3D Manual: The Scene Graph Browser

The Scene Graph Browser is only available when 'Direct Tools' are enabled. Information on enabling 'Direct Tools' is available in the Panda Tools section.

In the main Direct Session window, there are several scene placement tools available through the tabs at the right side of the screen. Clicking on these tabs will bring up a dialog box with the various properties for these aspects. On the left side of the panel is a collapsible scene graph for the render parent node. Right click any of the objects to bring up a list of possible commands.



Of particular interest is the placer panel, selected by the place command for an object. This brings up a separate window that may alter the position, orientation, and scale for the selected object. A dialog box at the top of the window allows these movements to be relative to another object in the program.



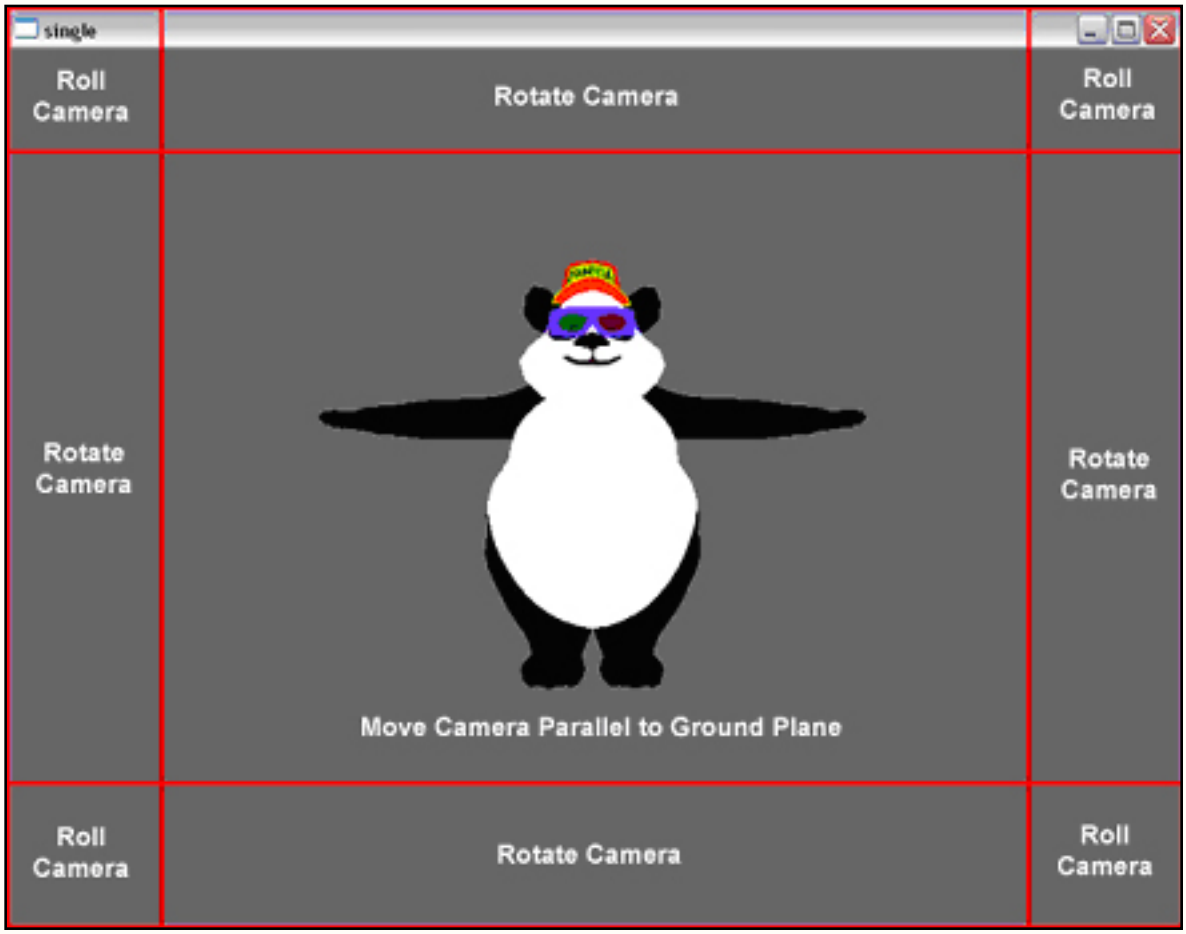
<<prev top next>>

Search

Panda3D Manual: Enhanced Mouse Navigation

Enhanced mouse navigation is only available when 'Direct Tools' are enabled. Information on enabling 'Direct Tools' is available in the Panda Tools section.

Direct Tools gives more functionality to the middle mouse button. First, clicking the middle mouse button changes the pivot point it uses to rotate around the environment. The middle mouse button will also move the camera depending on where the cursor is on the screen.



Middle Mouse Click	Sets pivot point for rotating around the world
Middle Mouse + Middle Region	Move camera parallel to ground
Shift + Middle Mouse + Middle Region	Move camera vertically
Middle Mouse + Edge Region	Rotate camera around pivot point
Middle Mouse + Corner Region	Roll camera around pivot point
Shift + Middle Mouse + Edge Region	Changes pitch of the camera

The left mouse button is now used to select and manipulate objects in the environment. Once an object is selected, it may be moved and rotated.

Left Mouse Click	Select an object
Left Mouse + Middle Region	Move object vertically

Shift + Left Mouse + Middle Region	Move object parallel to ground
Left Mouse + Edge Region	Rotate object around its pivot point
Left Mouse + Corner Region	Roll object around its pivot point
Control + Left Mouse	Rescale the model

Direct Tools uses a large number of hot keys for camera control, rendering styles, and object control. The full list is in the table below.

Camera Control

+	Zoom in
-	Zoom out
1	Front view (relative to render)
2	Back view (relative to render)
3	Right view (relative to render)
4	Left view (relative to render)
5	Top view (relative to render)
6	Bottom view (relative to render)
7	Â¾ view (relative to render)
8	Roll view about axis relative to cameraâ€™s axis
9	Rotate around hot point
0	Rotate around hot point
C	Center on hot point
F	Fit on hot point
H	Move to (0,0,0)
Shift + L	Toggle camera pivot point lock
N	Select next possible camera COA (along last intersection ray)
U	Orbit upright camera about hot point
Shift + U	Upright camera
`	Kill camera move task

Undo/Redo

Render Style

Shift + A	Show all
Control + F	Flash selected
B	Toggle backface
L	Toggle lights
T	Toggle texture
W	Toggle wireframe

Direct Controls

Delete	Delete selected object
Escape	Delete all
Page Down	Move down selected objectâ€™s hierarchy
Page Up	Move up selected objectâ€™s hierarchy
Tab	Toggle widget mode
Shift + F	Grow widget to fit current window
I	Plant selected object at cursor intersection point
M	Move widget in front of camera
P	Set active parent to selected object
R	WRT reparent selected to active parent
Shift + R	Reparent selected to active parent
S	Reselect last selected object
V	Toggle widget visibility
Shift + V	Toggle COA marker visibility

[	Undo	<	Shrink widget
]	Redo	>	Expand widget

Panda3D Manual: The Scene Editor

At the Entertainment Technology Center (Carnegie Mellon University) we have developed a Scene Editor to enable layout of objects in a 3D environment. This tool also allows you to do lighting, animation blending, creation of mopaths etc. This is just a first version of the tool and will be updated to encompass new features and improvements in the future.

To run the scene editor, run "sceneEditor.py" in the SceneEditor directory.

The scene editor's primary functions are:

- Object loading and manipulation
- Animation loading and blending
- Lighting
- Motion path generation
- Particle system generation
- Setting up collision detection

The scene editor will set up your scene for you and then output the layout at a normal python file. This file can then be used, edited as you desire. This tool is meant to give a visual interface for many functions that typically would have to be done manually in code. The best way to learn about the scene editor is to watch the Scene Editor Lectures:

- Introduction (48 mb)
- Camera Control and Object Manipulation (18mb)
- Animation Loading (8 mb)
- Lighting (11 mb)
- Animation Blending (26 mb)
- Motion Paths (24 mb)
- Particles (38 mb)
- Collision (34 mb)
- Miscellaneous Part I (20 mb)
- Miscellaneous Part II (16 mb)

Panda3D Manual: Python Editors

Using a well featured python editor or editor with python support will greatly ease the process of writing python scripts. Here are some of the features that you could look for when choosing an editor. Free editors don't usually have them all. There are quite a few capable editors out there though. In addition to all the features listed below, it should in general be a good and easy to use text editor.

- Syntax highlighting
- Smart indentation
- Auto-completion of text
- Call tips
- Tree view function/class selection
- Integrated shell windows
- Inbuilt debugger support
- Code folding support
- Customibility and extensibility
- Multi-platform support
- Unicode support

The python install itself comes with a very capable editor called IDLE, short for **I**ntegrated **D**eve**l**opment **E**nvironment. One of the features it has that the ones listed above do not is that it has an *enhanced* integrated shell. More information about IDLE can be found here.

Below is a list of a few free text editors. There is a comprehensive list of editors at http://www.python.org/moin/PythonEditors. Some of the information about the editors doesn't seem to be upto date at this listing. Try them yourself to find your favourite one.

PyPE	Syntax highlighting, auto-completion, smart indentation, calltips, tree view function/class selection, integrated shell, drag&drop, macros, customizable keyboard shortcuts, and more. Works on Windows and X. Written in python!
Emacs	Very powerful text editor, multiplatform, python support. Smart indentation, auto-completion, syntax highlighting, extensible.
Vim	Very powerful text editor, multiplatform, python support. Smart indentation, syntax highlighting, Scriptable in python.
ConText	Windows based free text editor with python syntax highlighting, and indentation support.
SPE	Syntax highlighting, smart indentation, auto-completion, call tips, class explorer, integrated shell, debugger, code folding, syntax coloring, UML viewer, syntax highlighting, source index, sticky notes, drag&drop, context help, and much more. Runs on Windows, Linux and Mac OS X. Also links to Blender. GUI is extensible with wxGlade. GPL'ed and written in Python.

Panda3D Manual: SPE

Installing SPE for use with Panda3D

Since Panda 1.3.0 the windows installer has the option to add panda's python to the registry. If you don't register or if you use an earlier version of panda, SPE cannot detect panda modules. Without them, features like autocomplete, the built-in console, debugger etc. do not work for Panda3D scripts. To correct this, SPE must first be integrated properly with panda's python (ppython-Panda3D's built-in python interpreter)

Integrate it manualy in this order:

1. download and install Panda3D

This installs ppython also. There should now be a directory "\Panda3D-x.x.x" where the x's are the version number.

2. download and install python

This installs the standard python interpreter required by the other installers. There should now be a directory "\Pythonxx" (again, x's are the version number)

This directory can be deleted after step 5 if you wish. (not before!)

3. download and install wxpython

SPE requires wxpython. This installs in "\Pythonxx\Lib\site-packages"

4. download and install SPE

This autodetects the standart python install, and also installs in "\Pythonxx\Lib\site-packages"

5. Manually copy the new files and folders in "\Pythonxx\Lib\site-packages" to "\Panda3D-x.x.x\bin\lib\site-packages"

When SPE is run from the new location using ppython it should have access to all the panda3D modules.

6. Write a script for convenience

e.g. in Windows create a new shortcut with target:

```
"C:\Panda3D-x.x.x\bin\ppython.exe" "C:\Panda3D-x.x.x\bin\lib\site-packages\_spe
\SPE.py"
```

This is just as easy in Linux using a shell script.

Importing DirectStart

SPE may need to or require you to import DirectStart for its PyDoc generation and calltips. By default this also pops up the main window. This can get annoying, but fortunately opening the window can be easily deferred by either adding the line

```
window-type
none
```

to your Config.prc file or by adding the line

```
loadPrcFileData("", "window-type
none")
```

to your script before you `import direct.directbase.DirectStart`. You may open the window later with the line

```
base.openMainWindow(type =
'onscreen')
```

Panda3D Manual: Pipeline Tips

This section isn't totally related to Panda3D. However, these are a few good pointers on how to keep the 'art to programmer' pipeline running smoothly.

How artists can help programmers with the pipeline:

"Keep renaming to a minimum, preferably with good names to start with, and especially for parts or joints that the programmer will have to manipulate."
Programmers generally deal with objects by their names, not with a graphical tool like artists. Every time a name changes, a programmer has to make the change in his or her code, and often has to hunt through the egg to try and figure out what the name has changed to. In a large model, this can be very time-consuming. If you do have to change a name, make sure to let the programmer know when you give him or her the new model.

"Check your model in pview before handing it off."
The biggest delays in the pipeline come from the back and forth iterations between artist and programmer. A quick check with pview will often find missing textures, backward facing polygons, incorrect normals, mis-tagged collision solids, and a host of other problems.

"Build models with good hierarchy, and don't change the hierarchy unnecessarily."
A well organized hierarchy can make a model much easier for a programmer to work with, and can also have a significant effect on rendering performance. For rendering purposes, good hierarchies group objects that are close to each other together, and don't have more than a few hundred to a few thousand triangles (depending on the target hardware) in each node. (Low-end hardware performs better with only a few hundred triangles per node; high-end hardware performs better with several thousand triangles per node.)

"Put groups of objects that the programmer will have to deal with in a special way under a single node."
For instance, if there's a section of an environment that will be hidden during some point in the game, put that entire section under a single node. The programmer might also like certain classes of collision solids to be under a single node.

"Don't use lossy compression (i.e. jpeg) for textures."
Although jpegs save space on disk, they also degrade your beautiful textures! If textures have to be manipulated later (i.e. downgraded), this degradation will only be compounded. Every time you edit and resave a jpeg, the image quality gets worse. Jpegs may need to be used in the finished product, but it's always best to make this conversion the last step in the process, not the first one. I recommend using the png format, which provides lossless compression and full support for all color depths as well as alpha.

"If there's any chance that an object will be broken apart and used as separate pieces, give each piece a separate texture map."
Nothing hurts worse than having to remap an object after its been painted, or wasting huge swaths of texture space.

"If parts of an object are semi-transparent, make sure those pieces are separate

parts in the hierarchy."
Rendering semi-transparent objects is a little tricky. Each object with transparency needs to be sorted back-to-front each frame by the rendering engine. If things aren't going quite as planned, a programmer might need to get a handle on a transparent part in order to manipulate its render order.

"If a semi-transparent object is very large, or you can see through multiple layers (like a glass sphere), break it up into separate pieces."
Objects with multiple layers of transparency won't render correctly depending on which angle they're being view from, because some of the polygons will be drawn before others, and if it's all one object, the rendering engine can't sort them back-to-front.

"Use quads, and higher-order polygons, for collision solids when possible, rather than triangles. But make sure your quads are planar."
In general, dividing a quad into two triangles doubles the time it takes to test a collision with it, so it is better to model collision polygons with quads when possible. The same goes for five-sided and higher-order polygons as well. However, there are two important requirements: (1) the collision polygons must be convex, not concave, and (2) they must be perfectly flat (all of the vertices must lie *exactly* in the same plane). If either of these is not met, Panda will triangulate the polygon anyway.

Things that don't matter as much, but will give programmers warm-fuzzies:

"Use a consistent naming scheme."
Programmers live in a world where names and naming conventions are incredibly important. Nothing makes them happier than when the names of art assets fit in well with their code. Common naming conventions are: mixedCaseNames, CapitalizedMixedCaseNames, names_with_underscores, names-with-hyphens. Pick one and stick with it.

"Don't misspell things."

"Add one little bit on the end... Think of `potatoe', how's it spelled? You're right phonetically, but what else...? There ya go... all right!"
-- Vice President Dan Quayle correcting a student's correct spelling of the word `potato' during a spelling bee at an elementary school in Trenton.

"I should have caught the mistake on that spelling bee card. But as Mark Twain once said, `You should never trust a man who has only one way to spell a word'."
-- Vice President Dan Quayle, actually quoting from President Andrew Jackson.

**"I should have remembered that was Andrew Jackson who said that, since he got his nickname `Stonewall' by vetoing bills passed by Congress."** -- Vice President Dan Quayle, confusing Andrew Jackson with Confederate General Thomas J. `Stonewall' Jackson, who actually got his nickname at the first Battle of Bull Run.

Panda3D Manual: Model Export

Panda3D uses a custom file format for its models, called egg. To create an egg file, you will need to use a modeling program (like 3D Studio Max or Maya) combined with either an export plugin or a file format converter. You can read more about this process in the following sections. Panda3D also provides a binary file format bam, which is quicker to load.

Both file formats contain:

- Vertices
- Triangles and larger polygons
- Joints (aka Bones)
- Vertex weights
- Texture pathnames (textures are not stored)
- Bone-based animation keyframes
- Morph targets (aka Blend targets)
- Morph animation keyframes
- Many control flags

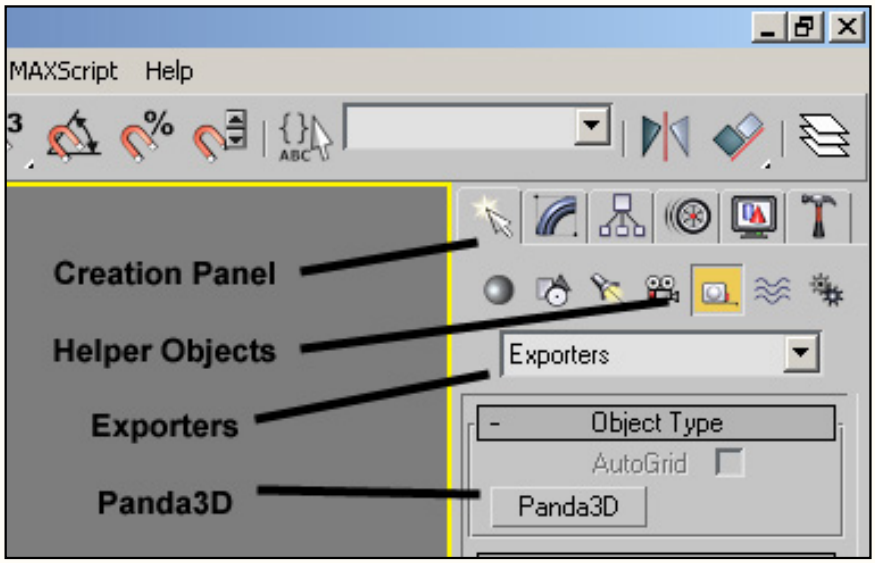
Texture pathnames in an egg file are first assumed to be relative to the egg file itself. If the texture is not found at that location, panda will search its model-path, which is specified in the panda config file. When doing this, panda concatenates the directory which is part of the model-path to the entire string in the egg-file. So if the model-path names the directory "/d/stuff", and the texture-path in the egg file is "mytextures/tex.png", then panda looks in "/d/stuff/mytextures/tex.png."

Panda3D Manual: Converting from 3D Studio Max

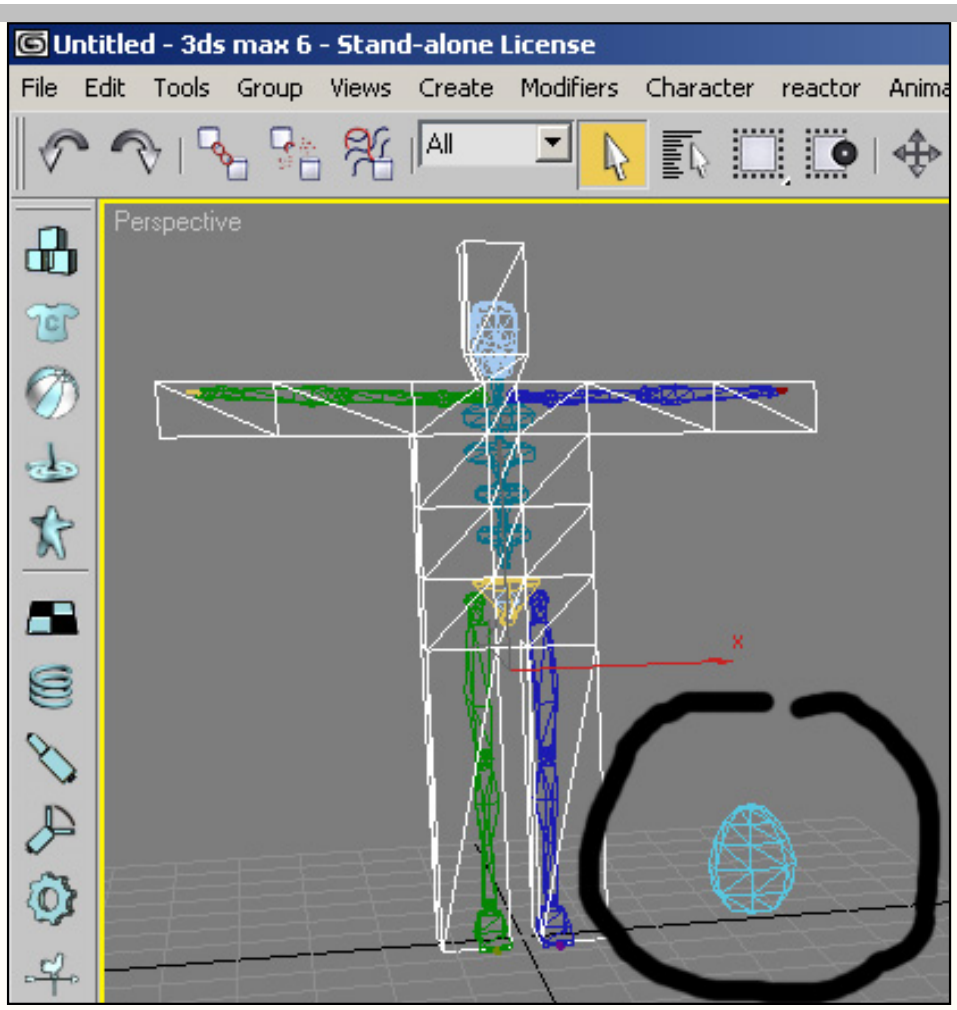
To install the exporter, look in the panda3d "plugins" directory. There, you will find the files `maxegg5.dlo`, `maxegg6.dlo`, and `maxegg7.dlo`. These are for 3D Studio Max versions 5, 6, and 7 respectively. Copy the correct file from the Panda3D plugins directory into the 3D Studio Max plugins directory, then, restart 3D Studio Max.

As of Panda3D 1.0.4, the old MAX exporter has been replaced with a brand new exporter. The new exporter is somewhat unconventional in its design. Max has a menu item "File/Export". Panda's egg format does *not* show up in this menu. Instead, Panda's exporter is a helper object. This enables the exporter to save your export settings from one session to the next.

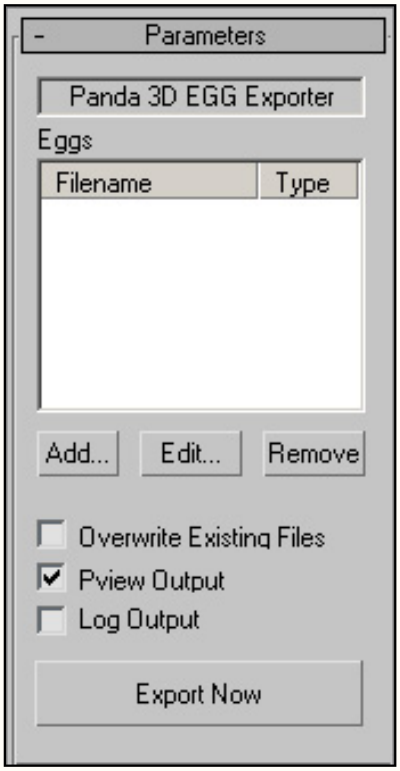
To export a scene, your first step is to create the necessary helper object. Go to the creation panel, select "helper objects," choose "exporters," and then click on the button to create a Panda3D export helper:



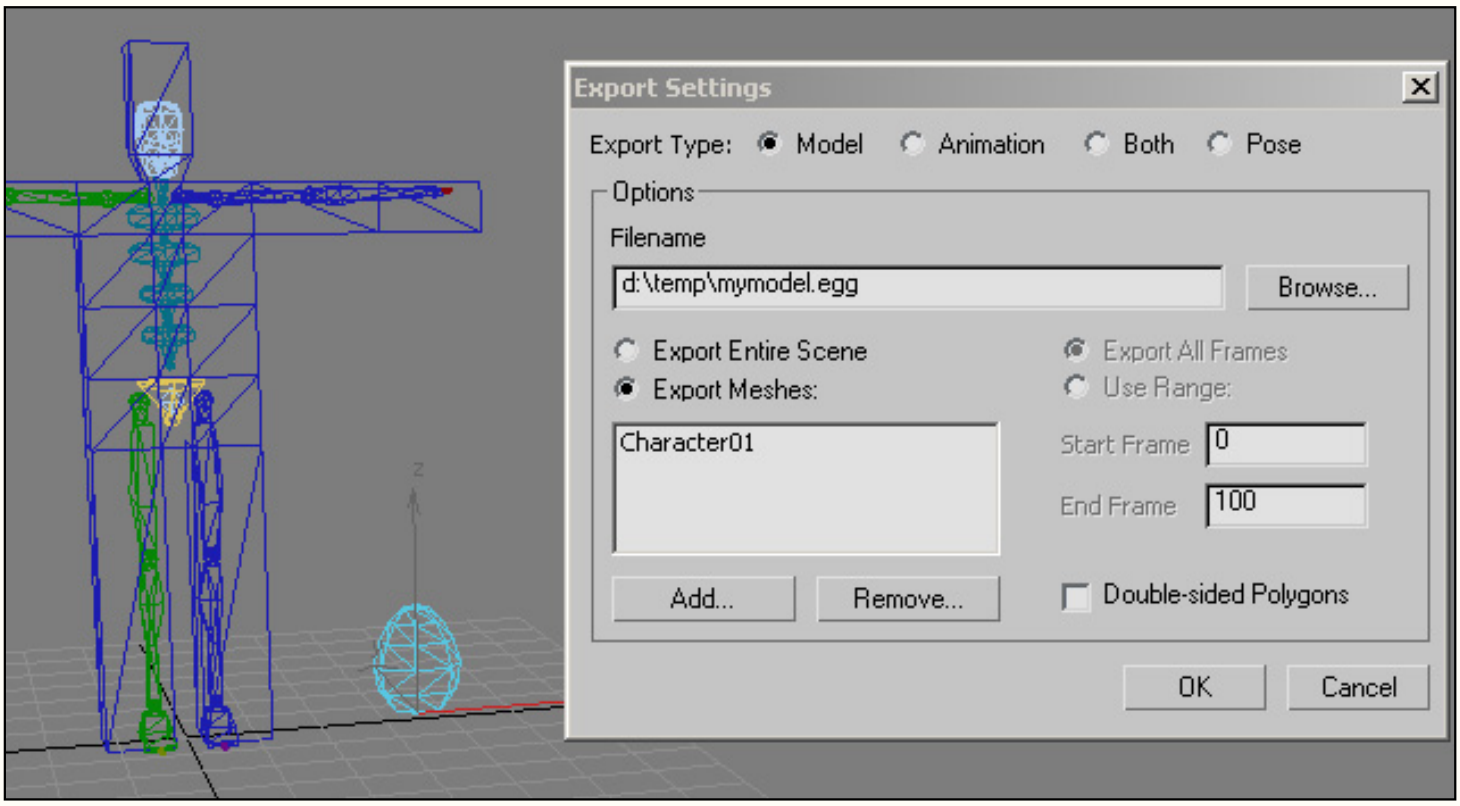
Drop an egg exporter into the scene:



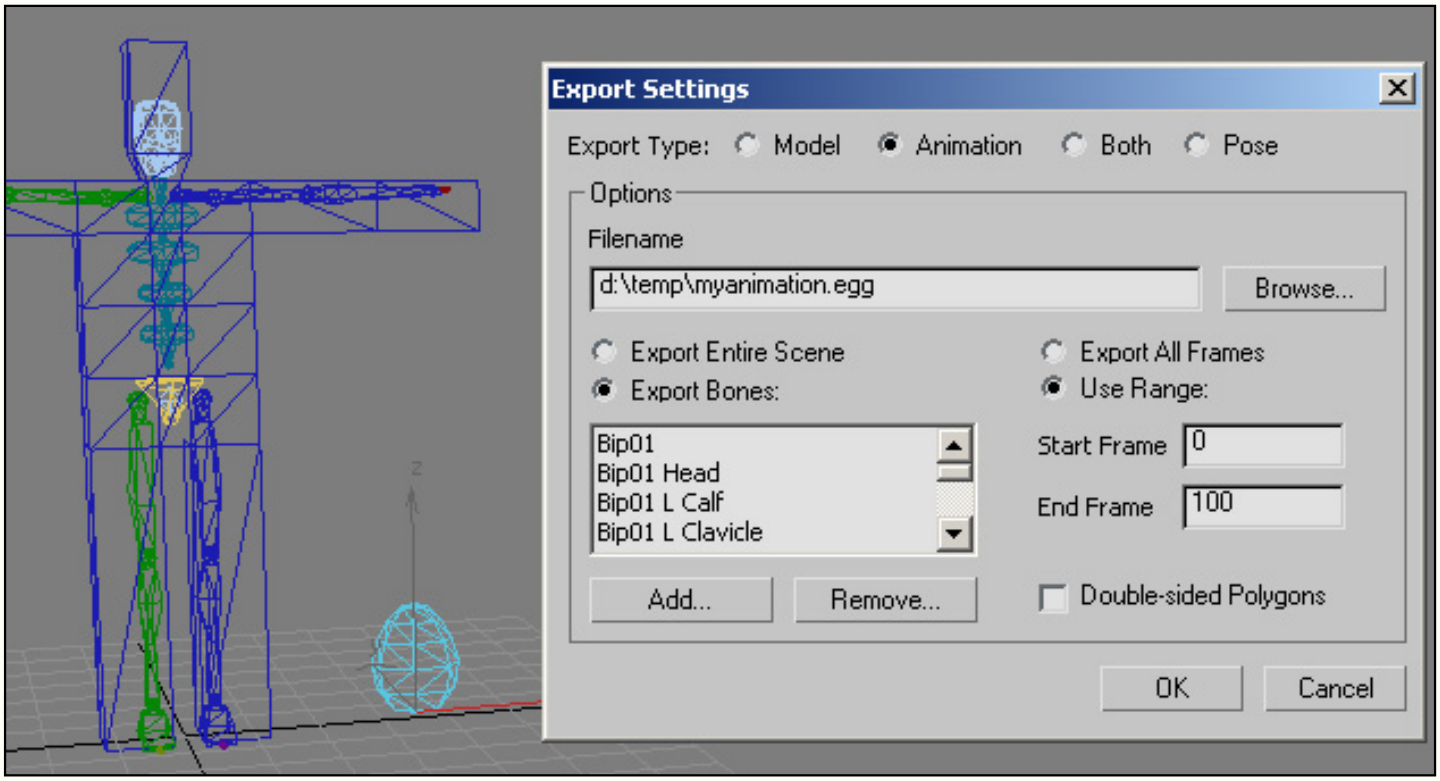
Switch to the modify panel. You will now be able to see the configuration settings that are stored in the export helper.



The exporter can generate several egg files from a single scene. The exporter therefore contains a list of egg files to generate. To export this particular scene (the one with the blocky humanoid and the biped skeleton), we will create an egg containing the model and one containing the animation. Click the "add..." button on the exporter's modify panel. You will be prompted:

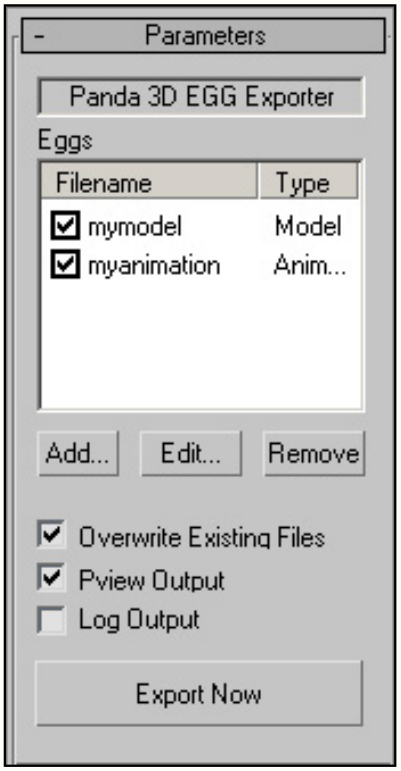


I have filled in the file name, and I have selected the mesh I want to export. When I click "OK," an egg file will be added to the list of eggs to generate. I then click the "add..." button again, and add another egg to the list:



This time, I'm generating an animation egg. I have listed the bones to export, and the range of animation frames.

Once I click OK, the modify panel for the egg exporter looks like this:



When I click the "export now" button, the two egg files are generated, and I am asked whether or not I would like to pview them.

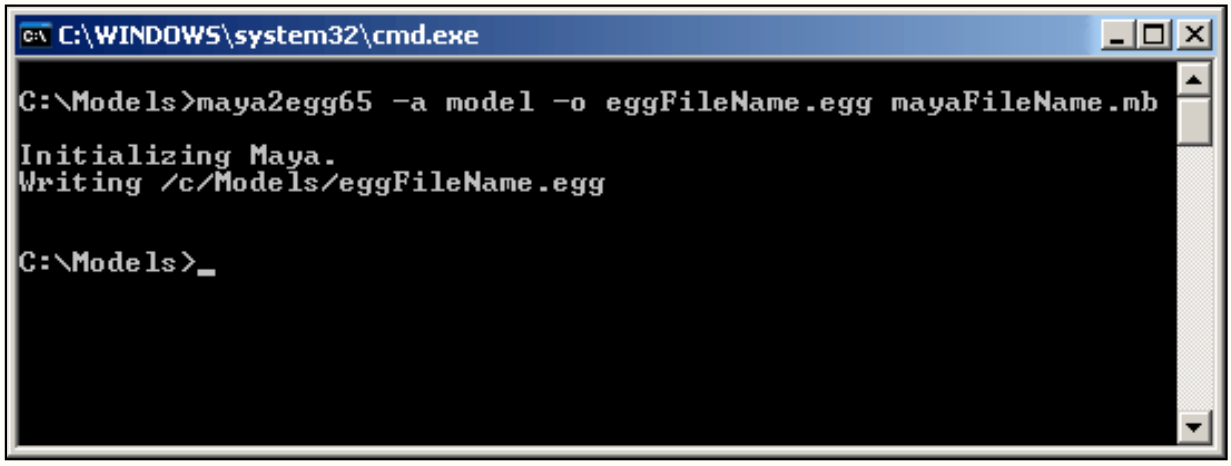
When you save your MAX file, the export helper will also be saved. The next time you load it up, it will still remember which meshes go in which egg files.

Panda3D Manual: Converting from Maya

Maya's exporter can be run from the command prompt or via a graphical panel. The command line utility is called *maya2egg*. There are multiple versions of it corresponding to different versions of maya. For instance, the version to use for Maya 5.0 is *maya2egg5*, for 6.0 its *maya2egg6* and for 6.5 its *maya2egg65*.

The following is an example of how to convert a file(maya binary .mb) if using Maya 6.0 through the command line.

```
maya2egg6 -o eggFileName.egg mayaFileName.
mb
```



The egg file will contain the filenames of the textures. These texture pathnames will be stored as *relative* paths, relative to the location of the egg file. For example, let's say that the files are laid out like this:

```
c:\My Models\Maya Files\Character.
mb
c:\My Models\Egg Files\Character.
egg
c:\My Models\Textures\Character.png
```

In that case, the command to export the model is:

```
c:\
cd c:\My Models\
maya2egg -o "Egg Files/Character.egg" "Maya Files/Character.
mb"
```

Note that Panda Filename Syntax uses forward slashes, even under Windows, and this applies to the exporter as well. After doing this export, the character egg will contain the following texture reference:

```
"../Textures/Character.
png"
```

Again, notice that this pathname is relative to the egg file itself. Many artists find it convenient to keep everything in

the same directory, to avoid having to think about this. This approach works fine.

The above conversion process will turn the character into a *static* model. Models which are rigged (they have bones to help them animate), skinned (polygons attached to the bones/skeleton), and are animated need to use one of the following options:

```
maya2egg -a model -o eggFileName.egg mayaFileName.
mb
maya2egg -a chan -o eggFileName.egg mayaFileName.mb
maya2egg -a pose -o eggFileName.egg mayaFileName.mb
maya2egg -a both -o eggFileName.egg mayaFileName.mb
```

The meanings of these options are:

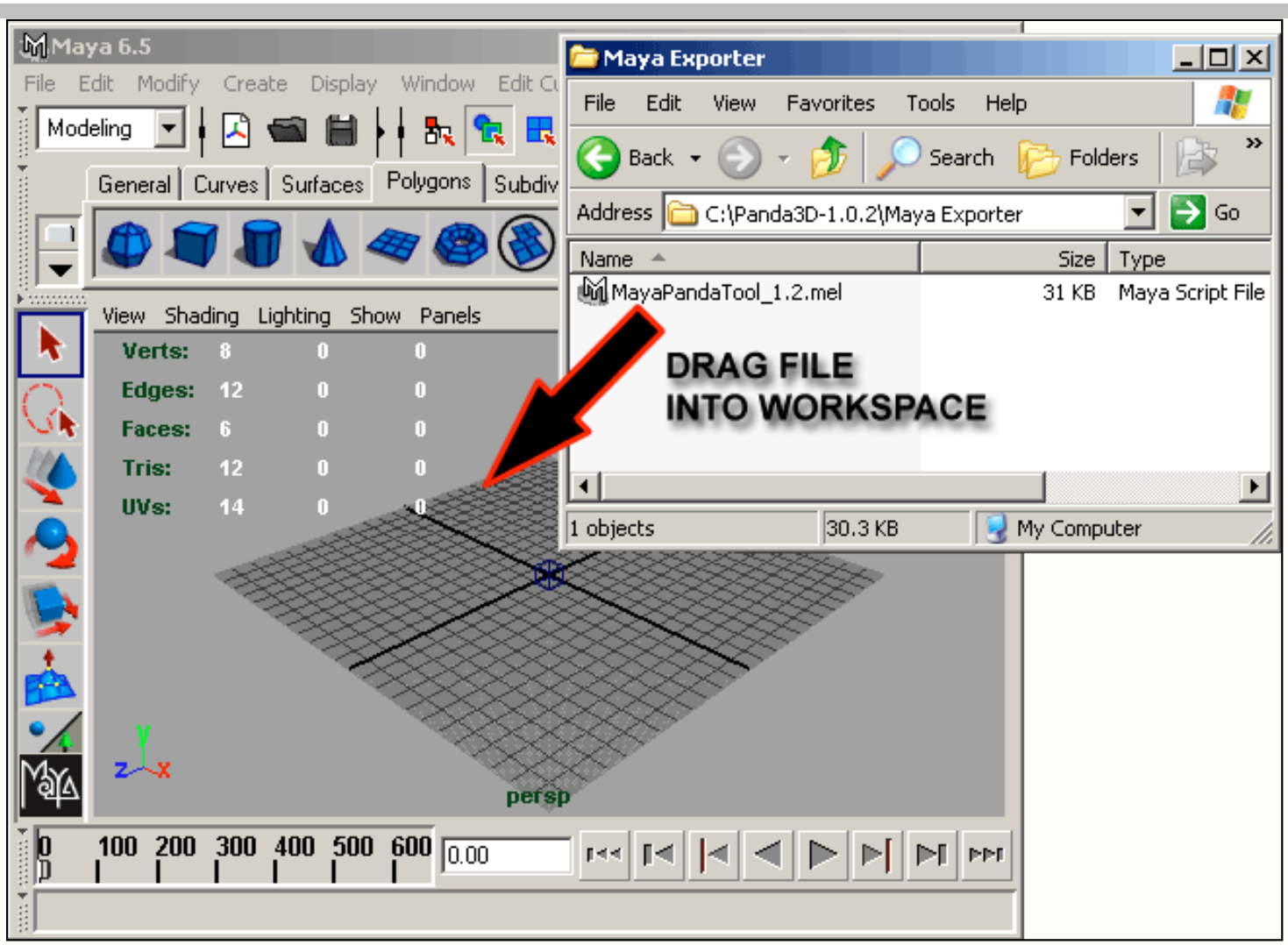
-a model	Save only the skinned and boned model, ready for animation but with no animation attached. This is ideal for models with several event- or interaction-based animations.
-a chan	Save only the animation data on the current model, but not the model itself. This is ideal applying animations to an already-existing model egg file that is ready to receive animation. A model may have several animation channels applied to it.
-a pose	Save the model in the current key position of the animation applied to it. This position must be selected before choosing to export the file. This format does not save the animation.
-a both	This will export the model and animation out as one file.
-uo ft	This converts the units of the model to feet, which is used by Panda3D.
-bface	Exports the model as double sided. Of course, this may very well cut the rendering speed in half.
â€“help	to see all the parameters that may be taken by the converter.

There are many options to maya2egg. For a complete list, run maya2egg with the *-h* argument.

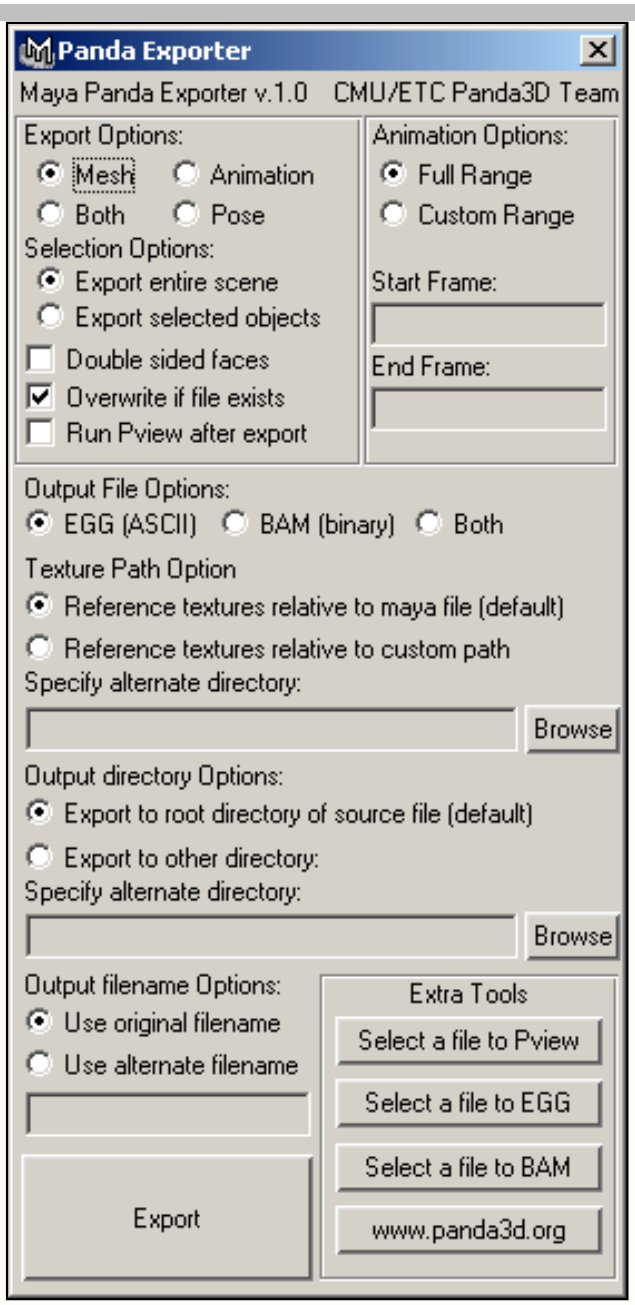
```
maya2egg -
h
```

Using the Graphical Front-End

As of Panda3D 1.0.4, there is a graphical front-end to *maya2egg*. To run the graphical Maya exporter, drag MayaPandaTool.mel from the panda plugins directory into the Maya workspace.

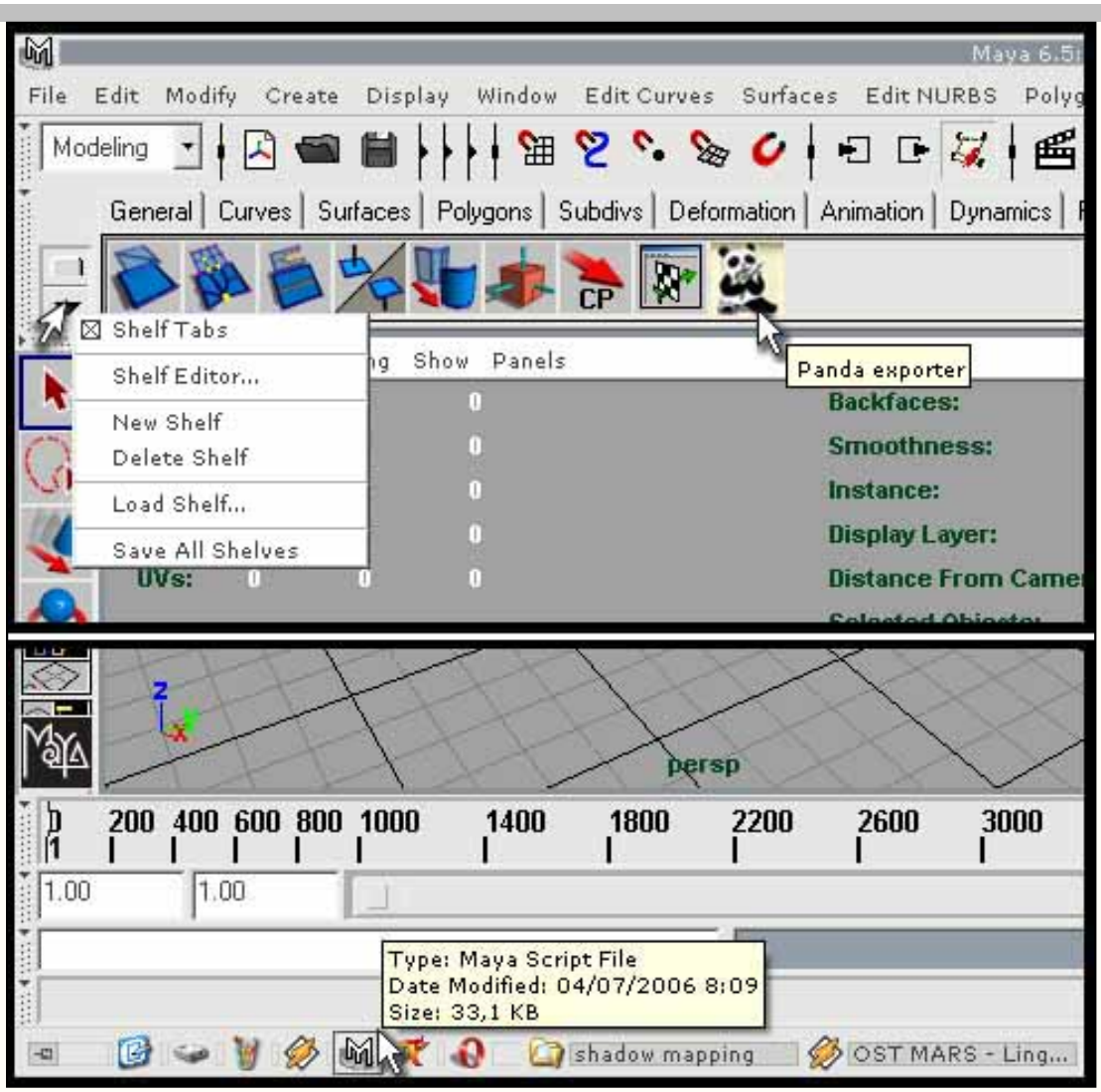


The UI window(below) will appear.



You can alternatively load the .mel file from the script editor. To save space, the graphical tool does not have access to all of the features of the exporter. It is designed for rapid verification of assests. The features you can execute with the graphical tool are identical to the respective ones of the command line exporter listed below.

Or, you can integrate the UI interface to Maya. There is a very convenient way to launch the MayaPandaTool, using Maya shelf to store the MEL script :
[1] open the Script Editor :
Window > General Editors > Script Editor
[2] In Script Editor, load the MayaPandaTool MEL script :
File > Open Script
[3] select the MEL text (press Ctrl+A)
[4] using middle mouse button, drag the selected text onto the shelf
[5] (optional) using Shelf Editor, you can change the image of Panda exporter icon to distinguish it from the others.



Anytime you need to open the Panda Exporter, just click the icon on the shelf.

For Windows users :
If you want a faster launch, you can put the MEL file on the QuickLaunch. Press Ctrl while dragging the MEL file onto QuickLaunch bar. If you don't press Ctrl, the actual file dropped to QuickLaunch is only the shortcut to the MEL file. Anytime you need to open the Panda exporter, just drag it from QuickLaunch bar to the 3D window of Maya.

<<prev top next>> Search

Panda3D Manual: Converting from Blender

Currently, there are two ways to get data from Blender into Panda3D.

Option 1: The "X" File format

There exists a free plugin for Blender that can export "X" (DirectX native) file format. Save the file from blender as an X file, then load it directly into Panda3D, which can read X file format. Alternately, if you're concerned about long load times (panda has to translate the file at load time), then pre-convert the model from X to Egg to Bam using the conversion programs supplied with Panda3D (x2egg, egg2bam).

Whenever you save a model in a non-native file format, you need to ask yourself: "does this file format support everything I need?" For example, when you save out a model in 3DS file format, you automatically lose all bone and animation data, because the 3DS file format doesn't contain bone and animation data. In the case of the X file format, you're in good shape: it's a fairly powerful file format, supporting vertices and triangles, bones and animation.

Note however, when an animated X file is converted to egg, the resulting egg file only plays the keyframes, but not whats supposed to be in between. For example, an animation could exist that should spawn 200 frames, gets sized down to about 40, and playback looks shakey. This shakeyness happens because the X file format supports the concept of keyframes, with implicit frames interpolated between them. The egg file format is explicit. An egg file must give all of the frames of an animation, even the frames that appear between "keyframes".

Therefore, a run of x2egg with an X file that omits a lot of frames between keyframes, will product a shakey animation. The only solution is to ensure your X files are generated with all frames. Testing of different X file exporters may be required.

Further, panda's native egg file format supports some esoteric things. For example, it supports blend targets (morph animations) and motion path curves, which are not supported by the X file format.

Caution: it has recently been discovered that there are two bugs in panda's X-file parser. One, it is case-sensitive and it should not be. Two, it does not handle hyphens in identifiers correctly. These two bugs will be fixed in an upcoming version of Panda3D. For information on temporary workarounds, search the forums.

Option 2: The Egg export Plugin for Blender

There are several Blender plugins contributed by Panda3D users.

Chicken is the most recently updated, documented and feature complete. It supports static meshes and armature animation, materials, vertex colors, alpha textures, tags, object types, etc. It also has advanced features such as automatic invocation of Panda tools (egg2bam, egg-optchar and pview) and Motion Extraction. You can find it at http://damthauer.byethost32.

com/panda/

EggX is another exporter that does materials, non-procedural textures, armature animation, static meshes. It can be found at http://www.wickwack.com/panda

Another exporter that only supports static meshes can be found at http://xoomer.virgilio.it/glabro1/panda.html

<<prev top next>> Search

Panda3D Manual: Converting from SoftImage

WRITE ME: Add information about X file format.

Panda3D Manual: Converting from Milkshape 3D

Currently, the best way to get data from Milkshape into Panda is to use X (DirectX native) file format. Save the model as X from Milkshape, then load it directly into Panda, which can read X file format. Alternately, if you're concerned about long load times (panda has to translate the file at load time), then pre-convert the model from X to Egg to Bam using the conversion programs supplied with Panda3D (x2egg, egg2bam).

Note that Milkshape 3D contains *two* X export plugins. I have heard that one of them does not work correctly. This may require some experimentation.

Whenever you save a model in a non-native file format, you need to ask yourself: "does this file format support everything I need?" For example, when you save out a model in 3DS file format, you automatically lose all bone and animation data, because the 3DS file format doesn't contain bone and animation data. In the case of the X file format, you're in good shape: it's a fairly powerful file format, supporting vertices and triangles, bones and animation.

However, egg file format supports some esoteric things. For example, it supports blend targets (morph animations), which are not supported by the X file format.

More Detailled explanations for MS3D users

You can use MS3D to create X Files (both static or animated) to be converted by Panda3d x2eggconverter. </br>

In MS3D there is two Direct X .X exporter: Direct 8.0 and DirectX(JT). So far i've managed to use only DirectX 8.0 File. (DirectX JT got a lot more parameters and only a few combination of it seemed to work but not on a predictable basis). I'll talk only for animation with bones (not tested other ones).But this exporter work also for static meshes.

A) Before Exporting to .X you must ensure: - no null material or null name in texture in your model (MS3D won't block you but will crashes during the export) - no hyphen in your bones names (underscore is ok) (No issue in MS3D but issue with panda converter). - animation mode is NOT enabled

B) To export , use Direct 8.0 file export. Select the required boxes. (Meshes, Materials , Animations) if you selected less than all checboxes (material animations...) you will have to edit manually the x files to remove the last 1 or 2 "}" of the file . before using X2egg to convert . It's ok to leave defaut settings (Lock Root Bone and 1 as Frame offset). Warning: Export can be very long in case of big models/animations.

C) Convert using X2egg converter

Warning:if you run X2egg without special args, you will need to have your textures also in the

same directory than the x files. Don't be surprised if .egg file size is 6 times your .X file size, it's pretty normal due to more explicit information in the .egg file format. In case size is an issue, bamming the .egg file will reduce the size and optimize loading time.

Also , before converting to .egg , you can load your .x in pview to check everything is fine.

D)TIPS: depending if you make your models fully in MS3D or import it from Poser, you may find an issue : all animations applied to root bone instead of correct bone. You can solve it in MS3D by regrouping all materials, export to HL SMD (1 or 2) then import again and export to . X.

NB: this have been written by a coder not an artist :-)

Bugs in the Process

Caution: at one time, it was discovered that there were two bugs in panda's X-file importer. One, it was case-sensitive and it should not be. Two, it did not handle hyphens in identifiers correctly. It is unknown whether or not these bugs have been fixed.

<<prev top next>> Search

Panda3D Manual: Converting from GMax

To convert models to Panda from GMax, you must first convert models to .X format, and then load them into Panda as .X files, or convert them further using x2egg and/or egg2bam.

There is a fair amount of work involved in setting up the GMax-to-X converter, but once it is set up the conversion process is reportedly very easy.

Installation

There are several plug-ins required to export .X files from gmax:

1. First download the â€˜Gmax gamepack SDKâ€™ found at this link: FlightSimulator exporter plugin. Size is about 15Mb. Although only 3 files are actually needed, they are not available as separate downloads, so unfortunately youâ€™ll need to download the whole thing.

2. Next, download programs 'MDLCommander' and 'Middleman'.

3. After download, youâ€™ll see that the 'fs2004_sdk-gmax-setup' is an exe. If you install it in the default gmax directory, youâ€™ll end up with a lot of extra stuff that you donâ€™t need. So create a new folder somewhere on your hard drive and install it there.

4. When done, open the folder, go to gamepacks > fs2004 > plugins. And copy all 3 files: 'FSModelExp.dle', 'makemdl.exe' and 'makemdl.parts.xml' to your main ../gmax/plugins folder.

5. You need to rename two of the files. Right click on 'makemdl.exe' and rename it to to 'mkmdl.exe'. Then right click on 'makemdl.parts.xml' and rename it to 'mkmdl.parts. xml' (without the quotes).

6. Next, unzip 'MDLCommander.zip'. Then copy 'mdlcommander.exe' to your main ../gmax/ plugins folder. This file also needs to be renamed. Right click on 'mdlcommander.exe' and rename it to 'makem.exe'.

7. Finally, unzip 'Middleman13beta3.zip'. Then copy 'makemdl.exe' to your main ../gmax/ plugins folder. Thatâ€™s it, youâ€™re done!

Using

To convert your gmax model to .X format. Go to â€˜File > Exportâ€™ and select â€˜Flightsim Aircraft Object (.MDL) from the file type dropdown. Type in a filename and click Save. The Middleman dialog window should now appear. Click the â€˜Optionsâ€™ tab and check â€˜SaveXFileâ€™ (this saves the x file) and â€˜nocompileâ€™ (this tells mdlcommander to only create an .X file not mdl/bgl). Then click the GO button.

After a few seconds the dialog will close and your newly exported .X model should be in the directory where you saved it to.

Bugs in the Process

The GMax converter writes slightly nonstandard .X files; it writes the name "TextureFileName" instead of "TextureFilename" for each texture reference. It may also generate hyphens in identifiers. Both of these can confuse the X file parser in Panda3D version 1.0.4 and earlier (this will be fixed in a future release of Panda). In the meantime, the temporary workaround is to edit the .X file by hand to rename these strings to the correct case and remove hyphens.

<<prev top next>> Search

Panda3D Manual: Converting from other Formats

There are several tools included with panda that can convert various file formats into egg file format:

```
lwo2egg
dxf2egg
flt2egg
vrml2egg
x2egg
```

Note that panda can load any of these file formats without conversion, doing so causes the conversion to occur at runtime.

Also, be aware: many of these file formats are limited. Most do not include bone or animation data. Some do not store normals. Currently the .x format is the only one of these that stores bones, joints and animations.

Panda3D Manual: Converting Egg to Bam

Panda's native egg file format is human-readable. This is convenient, but the files can get very large, and they can a little bit slow to load. To accelerate loading, Panda supports a second native format, bam. These files are smaller and are loaded very rapidly, but they cannot be viewed or edited in a text editor. Also, bam files are specific to the version of Panda they are created with, so they are not a good choice for long-term storage of your models.

Texture pathnames in an egg file are first assumed to be relative to the egg file itself. If the texture is not found at that location, panda will search its model-path, which is specified in the panda config file. When doing this, panda concatenates the directory which is part of the model-path to the entire string in the egg-file. So if the model-path names the directory "/d/stuff", and the texture-path in the egg file is "mytextures/tex.png", then panda looks in "/d/stuff/mytextures/tex.png."

Texture pathnames in a bam file may be stored relative to the bam file itself, relative to a directory on the model-path, or with a full pathname to the file, depending on the parameters given to the egg2bam program.

The program egg2bam is used to convert egg files to bam files. Egg2bam will complain if the textures aren't present. You must install the textures (into your model path) before you convert the bam file. You can run the egg2bam program as follows:

```
egg2bam -ps rel -o bamFileName.bam eggFileName.
egg
```

Here, "-ps rel" means to record the textures in the bam filename relative to the filename itself; if you use this option, you should ensure that you do not move the bam file later without also moving the textures. (The default option is to assume the textures have already been installed along the model path, and record them relative to the model path. If you use the default option, you should ensure the textures are already installed in their appropriate place, and the model-path is defined, before you run egg2bam.)

The egg2bam program accepts a number of other parameters that may be seen by running `egg2bam â€“h`.

Panda3D Manual: Parsing and Generating Egg Files

Transforms and Vertices

The egg syntax defines all transforms, including joint transforms, relative to the parent node only. When the animation is played, Panda accumulates the transforms for each joint.

Although joints are defined using a local transform, vertices are defined in an egg file using global coordinates, which is irrespective of transforms appearing within the egg file. This means when Panda loads the egg file is loaded, the vertex coordinates given in the egg file must be pre-transformed by the appropriate inverse matrix to compensate.

Custom .egg Readers/Writers

When writing an importer or exporter for panda, you have two choices.

One option is to use the panda runtime library, which includes code for reading, parsing, storing, and emitting Egg files. This approach can save you a great deal of effort. However, it does require that you link with the panda runtime system, which may be inconvenient if you wish to distribute a small, standalone file translator.

If you decide to use the panda runtime system, the classes you will need to use are the ones whose names start with "Egg," ie, `pandac.EggData`, `pandac.EggVertex`, `pandac.EggPolygon`, `pandac.EggGroup`, and so forth. Like all panda classes, these are documented in the API reference manual.

The other alternative is to parse/generate the Egg file entirely by yourself. In this case, you will need to read the syntax documentation for egg files. This documentation is part of the source code on sourceforge. The file format is human-readable, and fairly straightforward.

If you are writing a program to *generate* Egg files, either approach is equally good. However, if you are writing a program to *parse* Egg files, we do recommend using the panda runtime library, rather than writing your own parser, for the simple reason that it is difficult to write a parser that accepts all valid Egg files. Also, the Egg syntax might be extended from time to time, and relying on the runtime library to parse the Egg syntax will ensure that your program continues to parse future Egg files.

Panda3D Manual: Previewing 3D Models in Pview

Pview or Panda Viewer is a model and animation viewer for egg and bam files. This allows users to see if their files have converted correctly without having to create a Panda3D program. Pview is accessed through a command prompt.

To view a model that has been converted to an egg or bam file, type the following:

```
pview modelFile.
egg
```

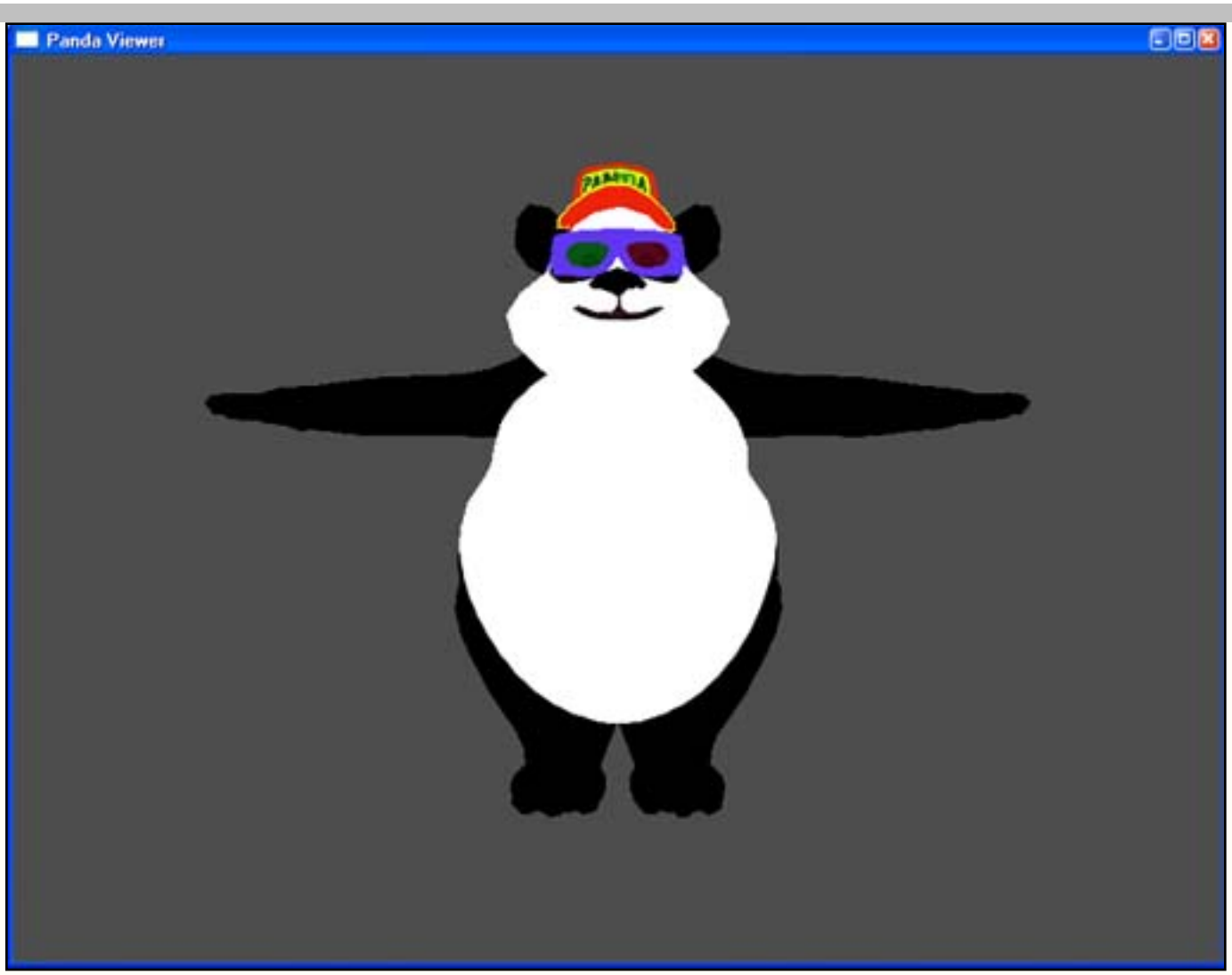
To view a character model with animations, simply add the name of the file with the animation.

```
pview modelFile.egg animationFile.
egg
```

Here's an example based on the panda model distributed with panda source.

```
pview
panda
```

A new window should pop-up and here's what you should see -



There are some controls and hotkeys available while using pview. To see the whole list press shift-question mark in the pview window. To turn this list off press shift-question mark again. For convenience here is the full list of as the time of writing:

Left-click and drag Mouse	Moves the model up, down, left, and right relative to the camera
Middle-click and drag Mouse	Rotates the model around its pivot
Right-click and drag Mouse	Moves the model away and towards the camera
f	Report framerate. The current framerate is output on the console window.
w	Toggle wireframe mode
t	Toggle texturing
b	Toggle back face (double-sided) rendering
i	Invert (reverse) single-sided faces
l	Toggle lighting
c	Recenter view on object
shift-c	Toggle collision surfaces
shift-b	Report bounding volume
shift-l	List hierarchy

h	Highight node
arrow-up	Move highlight to parent
arrown-down	Move highlight to child
arrow-left	Move highlight to sibling
arrow-right	Move highlight to sibling
shift-s	Activate PStats
f9	Take screenshot
,	Cycle through background colors
shift-w	Open new window
alt-enter	Toggle between full-screen and windowed mode
2	Split the window
W	Toggle wireframe
escape	Close Window
q	Close Window

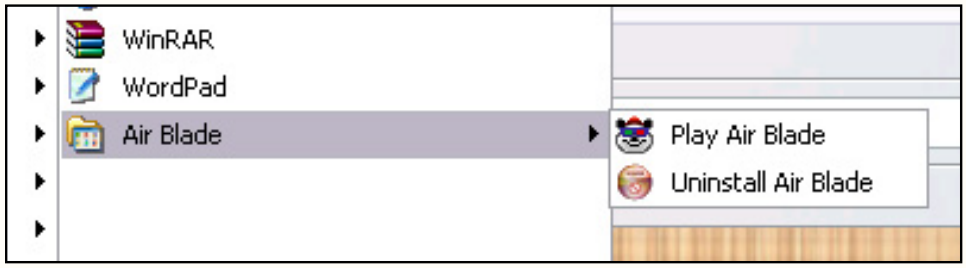
As of this writing there is a small caveat in loading textures for models into pview. Textures are searched for relative to the directory that pview is called from, not the directory that the model is in. Either make sure that the texture path is set correctly in the Config.prc file or ensure the bam or egg file look for the textures correctly (relative to where you are calling pview). The easiest way to get around this is to call pview from the directory that the model is placed.

Panda3D Manual: Building a Self-Extracting EXE using packpanda

Update: release 1.2.2 of panda contains a nonfunctioning copy of packpanda. To repair it, install panda, then move "packpanda.nsi" from the subdirectory "direct" to the subdirectory "direct\src\directscripts." This will be permanently fixed in the next release.

Packpanda is a utility that lets you create a windows installer for a panda game. The result looks like any other windows installer:

When the installation is done, the end-user will find your game in his start menu:



The end-user doesn't need to have a copy of panda. He doesn't even need to know that he is using panda. He just installs the game and plays it.

Files that your Game should Contain

Before you pack up your game, you need to put all of your game files into a single directory (which may have as many subdirectories as you desire). This directory will be packed up and shipped to the user, along with the panda runtime system. Your game directory needs to contain several files:

main.py. This is your main program. When the user clicks on the start-menu entry for the game, this is the file that will get executed.

installer.bmp. This image will appear on the installer screen. If present, it must be a 164x314 windows BMP file. This file is not required.

license.txt. This is your game's software license. The file, if present, must be plain ascii. The game's license will appear inside the installer, and will also be copied to the game's installation directory. Of course, your license only covers the code that you wrote, not panda itself, which is covered by the panda license. The license file is not required.

icon.ico. This is your game's icon, which will appear in the start menu. If you don't supply an icon, the panda icon will be used instead. This file is not required.

Packing up your Game

The command to pack up your game is "packpanda", and you must specify the "--dir" command line option to tell it the name of the directory containing your game. Packpanda will immediately analyze your game and print out a status report:

In this example, packpanda has inferred that the name of the game is "Airblade," based on the directory name. It plans to install the game into "C:\Airblade", and to add the start menu folder "Airblade". It plans to call the installer "Airblade.exe". Later, we will tell you how to override some of these defaults.

As you can see, packpanda is looking inside the game directory for the files mentioned above: main.py, installer.bmp, icon.ico, and license.txt. It notes that some of those files are "missing", which is not a problem. The only file that is required is main.py.

Packpanda can clean up your source tree before shipping it. When doing so, packpanda never modifies your original copy of the game. Instead, it copies the game to a temporary directory, as seen above.

EGG Verification and PY Verification

Packpanda will check all of your EGG and PY files to make sure that they compile correctly. It checks EGG files by running them through egg2bam. It checks PY files by running the python compiler on them. If any file fails, the game will not be packed.

Packpanda can optionally ship the BAM and PYC files it creates to the end-user. To ask it to do so, use the following command-line options;

```
packpanda --bam # Ship BAM
files
packpanda --pyc # Ship PYC
files
```

These command line options do *not* remove the corresponding EGG and PY files from the distribution. If you wish to remove EGG and PY files, you need to use the --rmext option, documented below.

When packpanda generates a BAM or PYC file, it puts it in the same directory as the corresponding EGG or PY file. If an EGG file contains a texture path, then the generated BAM will contain a *relative* texture path that is relative to the game's root directory. Packpanda makes sure that your game's root directory ends up on the model path.

If you do not supply the --bam or --pyc options, packpanda will still generate BAM and PYC files for verification purposes, but it will not ship the files it generates to the end-user.

Stripping Files from the Distribution

Often, your master copy of a game contains files that should not be shipped to the end-user. For situations like this, packpanda contains command-line options to strip out unnecessary files:

```
packpanda --rmdir dir # Strip all directories with given
name
packpanda --rmext ext # Strip all files with given extension
```

These options are particularly useful in several common situations:

To remove CVS directories: packpanda --rmdir CVS

To ship BAM instead of EGG: packpanda --bam --rmext egg

To ship PYC instead of PY: packpanda --pyc --rmext py

Changing the Game's Name

Normally, packpanda infers the game's name to be the same as the directory name. That isn't always convenient, especially when the game has a long name. The following command line option allows you to tell packpanda the name of the game:

```
packpanda --name "Evil Space Monkeys of The Planet
Zort"
```

This string will show up in a number of places: in particular, throughout the installation dialogs, and in the start menu.

Version Numbers

If you wish, you can assign your game a version number using this command line option:

```
packpanda --version X.Y.Z # Assign a version
number
```

The only thing this does is to add "X.Y.Z" to the install directory and to the start menu item. That, in turn, makes it possible for two versions of the same game to coexist on a machine without conflict.

Compression Speed

Normally, packpanda uses a very good compression algorithm, but it's excruciatingly slow to compress. You can specify this command line option to make it go faster, at the cost of compression effectiveness:

```
packpanda --fast # Quick but not so great
compression
```

Moving Beyond Packpanda

Packpanda has a lot of limitations. However, packpanda is actually a front end to NSIS, the "Nullsoft Scriptable Install System." NSIS is incredibly powerful, and very flexible, but unfortunately rather complicated to use. Packpanda hides all that complexity from you, but unfortunately, in so doing, it limits your options.

If you find yourself outgrowing packpanda, one sensible thing to do would be to learn how to use NSIS directly. This is an easy transition to make. The first step is to simply watch packpanda in action. It will show you all of the commands it is executing. You can then copy those commands into a batch file. If you run that batch file, you're executing NSIS directly.

Once you have direct control over NSIS, you can begin editing the NSIS command-line options and the NSIS configuration file (packpanda.nsi). Of course, to do so, you'll need to first read the NSIS manual (available on the web). From that point forward, you have unlimited flexibility.

Panda3D Manual: Building Panda from Source

Note that in the past, it was very difficult to build panda. Things have improved. It is now fairly straightforward to download and compile panda. In particular, this is a sensible thing to do if you wish to add some functionality to panda.

To avoid possible consistency problems, the documentation for building the source is included with the source, not in the manual. Download a source package from the panda website, then look in the directory "doc," where you will find two files: INSTALL-MK and INSTALL-PP. These contain the instructions on the two different build-systems, "makepanda" and "ppremake".

Having built from source, Python can be configured to let you import Panda3D modules as if they were installed in site-packages. Create a file called panda3d.pth inside your site-packages directory containing something similar to the following (this example is for Linux):

/opt/panda3d/built /opt/panda3d/built/lib

The two entries should be on separate lines. Note that this assumes the built version of Panda3D has been moved or copied to /opt/panda3d.

In addition to this path file, create a file __init__.py and save it into the "built" directory, or /opt/panda3d/built as in this example for Linux. This file can be empty. Without, Python will not be able to locate the Panda3D modules.

Note that I only tested this under Linux.

Panda3D Manual: Troubleshooting ppremake on Windows

This page describes potential build issues that may arise when building Panda3D using the PPremake build system on the Windows platform, and possible solutions.

ppremake doesnt run (access violation 0xc0000005)

The April/May 2005 version of Cygwin may have had an issue with getopt.

Simply open config.h in the ppremake directory, replace "#define HAVE_GETOPT 1" with "#undef HAVE_GETOPT", then rerun "make" and "make install".

invalid link option /DEBUG, when running make

This error (or something similar) is caused by the make running Cygwin's link rather than your msvc link.exe. Simply rename /usr/bin/link to /usr/bin/_link, and rerun make.

Building using Microsoft Visual C++ Toolkit 2003

Basically follow the official build instructions, using Cygwin as configuration platform and Microsoft Visual C++ Toolkit 2003 as compiler (you have to pretty much, this is very close to the primary configuration they support).

Pre-requisites:

- Microsoft Visual C++ Toolkit 2003
- msvcrt.lib and msvcprt.lib (see See Notes on Microsoft Visual CPP Toolkit 2003)
- DirectX (even though we're not using it, you need the headers
- Cygwin
- Panda source-code

The Panda source-code zip contains Python.

- First, you'll need to set up your variables for MSVC, within Cygwin. What you can do is to personalize the following file for yourself, and save it to /usr/local/panda/bin/setvars:

```
PATH=$PATH:"/cygdrive/f/Program Files/Microsoft Visual C++ Toolkit 2003/bin"
PATH=$PATH:/usr/local/panda/bin
PATH=$PATH:/usr/local/panda/lib
PATH=$PATH:/cygdrive/f/dev/panda3d-1.0.2-cyg/thirdparty/win-python

export CL=" /I\"F:\Program Files\Microsoft Visual C++ Toolkit 2003\include\" "
export CL="$CL /I\"F:\Program Files\Microsoft Visual C++ Toolkit 2003\include\" "
export CL="$CL /I\"F:\program Files\Microsoft SDK\include\" "
export CL="$CL /I\"F:\Program Files\Microsoft DirectX 9.0 SDK\include\" "
export CL="$CL /I\"F:\dev\panda3d-1.0.2-cyg\thirdparty\win-python\include\" "

export LINK=" /LIBPATH:\"F:\Program Files\Microsoft Visual C++ Toolkit 2003\lib\" "
export LINK="$LINK /LIBPATH:\"F:\Program Files\Microsoft Visual C++ Toolkit 2003\lib\" "
export LINK="$LINK /LIBPATH:\"F:\Program Files\Microsoft SDK\lib\" "
export LINK="$LINK /LIBPATH:\"F:\Program Files\Microsoft DirectX 9.0 SDK\lib\" "
export LINK="$LINK /LIBPATH:\"F:\dev\panda3d-1.0.2-cyg\thirdparty\win-python\libs\" "

export PANDA_ROOT='F:\Cygwin'
```

```
export PYTHONPATH="f:\dev\panda3d-1.0.2-cyg\thirdparty\win-python;f:\cygwin\usr\local\panda\lib"
```

- then run this file from Cygwin

```
. /usr/local/panda/bin/setvars
```

(note that there's a . at the start, and a space between the . and the rest of the command)

- also, you will need to rename /usr/bin/link to /usr/bin/_link , in order that cygwin finds msvc link, and not the gcc link
- Open cgywin/usr/local/panda/Config.pp (create the file, if you didnt already), and add the following lines:

```
#define HAVE_DX
```

- The HAVE_DX line means that you will not use DirectX (sic), which is a good thing, unless you happen to have an old DirectX 8 SDK lieing around (current version at microsoft.com is 9)
- Note that linking dynamically is the default; and this the configuration which builds easiest
- The Visual C++ Toolkit doesnt contain lib, only link, which can do the same thing, so open, in the panda source directory, dtool/pptempl/compilerSettings.pp, and replace

```
#define LIBBER lib
```

with

```
#define LIBBER link /lib
```

- Now, you're ready to run ppremake, make and so on in dtool, then panda, as per the instructions.

(These notes taken from: http://manageddreams.com/osmpwiki/index.php?title=Panda3D).

Panda3D Manual: Troubleshooting ppremake on Linux

This page describes potential build issues that may arise when building Panda3D using the PPremake build system on the Linux platform, and possible solutions.

(no known problems at this time)

Panda3D Manual: Troubleshooting makepanda on Windows

This page describes potential build issues that may arise when building Panda3D using the Makepanda build system on the Windows platform, and possible solutions.

(no known problems at this time)

Panda3D Manual: Troubleshooting makepanda on Linux

This page describes potential build issues that may arise when building Panda3D using the Makepanda build system on the Linux platform, and possible solutions.

(no known problems at this time)

Panda3D Manual: Video Lectures

The following sections contain links to video lectures about panda.

Panda3D Manual: Disney Video Lectures

David Rose, from Walt Disney Imagineering, periodically holds classes based on the fundamentals and working of Panda3D. Below are a few of these lectures. As of now each lecture is made to fit on a 700mb CD.

- Characters Part I (683 mb). Recorded October 8, 2003. This includes information about eggs, pandaNodes and actors.
- Characters Part II (669 mb). Recorded October 22, 2003. This is a recap of character information in Panda3D.

Panda3D Manual: Scene Editor Lectures

The Scene Editor was developed at the Entertainment Technology Center (Carnegie Mellon University). It is meant to be a level editing or layout tool.

Below are some video tutorials which explain how to use the Scene Editor. All tutorials were recorded by Shalin Shodhan in May of 2004.

- Introduction (48 mb)
- Camera Control and Object Manipulation (18mb)
- Animation Loading (8 mb)
- Lighting (11 mb)
- Animation Blending (26 mb)
- Motion Paths (24 mb)
- Particles (38 mb)
- Collision (34 mb)
- Miscellaneous Part I (20 mb)
- Miscellaneous Part II (16 mb)

Panda3D Manual: Panda 3D Video Tutorial Series

This is a video tutorial series available on FanFilmEngine.com. They were originally going to only be released to a certain game developement team, but the author has decided to share them with the public.

Part One: Downloading Panda 3D and Opening a Panda 3D Program
http://www.FanFilmEngine.com/panda_open.mov
Approximately 3.5 MB - Quicktime Format

Part Two: Downloading and Installing the Blender DirectX Exporter
http://www.FanFilmEngine.com/panda_to_x.mov
Approximately 2.5 MB - Quicktime Format

Panda3D Manual: API Reference Materials

Quick Reference: Functions and Classes

The following PDF reference sheets contain the most important panda functions in a single-page printable format.

Quick Reference Sheet

BVW Quick Reference Sheet

Comprehensive API Reference Manual

The following link will direct you to the API Reference manual. Note that while the reference is comprehensive, it is merely that: a reference. It does not make any attempt to explain concepts.

API Reference

As a general rule, it is best to search for the information you need in the Programming with Panda section, then, if you can't find what you need, dig down into the API reference manual.

Obtaining your own Copy of the Manual

It is now possible to download your own copy of the manual from our software downloads page.

However, the manual is updated daily, whereas the downloads page contains a snapshot taken on the day that the panda version was released. To obtain a more recent version of the manual, you will need a web-spider program called "wget." This tool is included with most versions of Linux, but you need to download it separately to use it under windows. To fetch the manual, use this command:

```
wget --restrict-file-names=windows -r -k -nd -E -Iwiki/index.php,wiki/images,
nimages,stylesheets http://panda3d.org/wiki/index.php/
```

Once you have a copy of the manual, you may wish to rename the following file:

```
copy Main_Page.1.html index.
html
```

Regenerating the API reference Manual

If you have compiled your own copy of the panda source code, you can regenerate the API reference manual. You can do so with the following commands:

```
makepanda\makedocs.
bat
```

Or, under Linux:

```
makepanda/makedocs.
py
```

The resulting manual will be found in the subdirectory "reference."

<<prev top next>> Search

Panda3D Manual: List of Panda Executables

This is meant to be a list of the executables in the /bin/ folder of Panda 3D. You can get a detailed synopsis of what the executables do by running them with -h as the arguement.

Filename	Description
bam-info.exe	Scans one or more .bam files and outputs their contents. See executable for more information.
bam2egg.exe	Converts models in the .bam format to the .egg format. For more information see Converting Egg to Bam.
bin2c.exe	Reads a file from disk and produces a table that when compiled by C compiler reproduces the same data. See executable for more informatiion.
cgc.exe	A compiler for NVidia's Cg language. For more information see Using Cg Shaders.
check_md5.exe	Outputs the MD5 hash for one or more files. See Executable for more information.
dxf-points.exe	Reads in an AutoCad .dxf file and prints out the points contained in it. See executable for more information.
dxf2egg.exe	Converts models from the Autocad format to the .egg format. For more information see Converting from other Formats
egg-crop.exe	Strips an .egg file of all parts that fall outside the given bounding volume. See executable for more information.
egg-make-tube.exe	Creats an .egg file representing a "tube" model. See executable for more information.
egg-mkfont.exe	Makes a .egg file from a FreeType (.ttf) font. For more information see Text Fonts.
egg-optchar.exe	Optimizes models by removing unused joints. Also allow you to label parts of the model. For more information see Manipulating a Piece of a Model.
egg-palettize.exe	Tries to combine textures in an egg file. Also performs some texture manipulation. See executable for more information.
egg-qtess.exe	Performs a tesselation on all of the NURBS surfaces in a .egg file. See executable for more information.
egg-texture-cards.exe	Creates an egg that automatically rotates through multiple textures. For more information see Automatic Texture Animation.
egg-topstrip.exe	Unnaplies the animations from one of the top joints in a model. Useful for character models that stack on top of each other. See executable for more information
egg-trans.exe	Produces out essentially the same .egg file. Useful for applying rotational and positional tronsformations. See executable for more information.
egg2bam.exe	Converts files in the .egg format to the .bam format. For more information see Converting Egg to Bam.

egg2c.exe	Reads a .egg file and produce C/C++ code that will almost compile. See executable for more information.
egg2dxf.exe	Converts files in the .egg format to the AutoCad format.For more information see Converting from other Formats.
egg2flt.exe	Converts files in the .egg format to the Open Flight format.For more information see Converting from other Formats.
egg2x.exe	Converts files in the .egg format to the DirectX format. Especially useful because it holds bone, joint and animation data. For more information see Converting from other Formats.
flt-info.exe	Reads an OpenFlight file and prints out information about its contents. See executable for more information.
flt-trans.exe	Produces esentially the same .flt file. Useful for postional and rotational transformations. See executable for more information.
flt2egg.exe	Converts files in the OpenFlight format to the .egg format. For more information see Converting from other Formats.
genpycode.exe	Genreates the Python wrappings necessary to inteface with the C++ libraries that are the backbone of Panda. Also generates the API Reference Manual in `<Panda Directory>/pandac/docs`. For more information see API Reference Materials
httpbackup.exe	Used to retreive a document from an HTTP server and save it on disk. See executable for more information.
image-info.exe	Reports the sizes of one or more images. See executable for more information.
image-resize.exe	Resizes an image. See executable for more information.
image-trans.exe	Produces an identical picture. Can also be used for file format conversion. See executable for more information.
indexify,exe	Takes image directories and creates HTML pages with thumbnails of these pictures. See executable for more information.
interrogate.exe	Parses C++ code and creates wrappers so that it can be called in a Scrtipting language. See executable for more information.
lwo-scan.exe	Prints the contents of a .lwo file. See executable for more information.
lwo2egg.exe	Converts files in the LightWave 3D format to the .egg format. For more information see Converting from other Formats.
make-prc-key.exe	Generates one or more new key to be used for signing a prc file. See executable for more information.
maya2egg5.exe	Converts files in the Maya 5 format to the .egg format. For more information see Converting from Maya.
maya2egg6.exe	Converts files in the Maya 6 format to the .egg format. For more information see Converting from Maya.
maya2egg65.exe	Converts files in the Maya 6.5 format to the .egg format. For more information see Converting from Maya.
multify.exe	Stores and extracts files from a Panda MultiFile. Can also extract file in program using the `VitrualFileSystem` (see API for usage). For more information see executable.
pdecrypt.exe	Decompress a file compressed by pencrypt. See executable for more inforamtion.

pencrypt.exe	Runs an encryption algorithm on the specified file. The original file can only be recovered by using pdecrypt. See executable for more information.
ppython.exe	Used to start Panda 3D. For more information see Starting Panda3D
pstats.exe	Panda's built in performance tool. For more information see Measuring Performance with PStats
pview.exe	Used to view models in the .egg or .bam format without having to create a Panda program. For more information see Previewing 3D Models in Pview.
stitch-command.exe	Checks the synatx and preproecees and sticth command file. See executable for more information.
vrml2egg.exe	Converts files in the Virtual Reality Modeling Language format to the .egg format. For more information see Converting from other Formats.
x2egg.exe	Converts files in the Direct X format to the .egg format. Especially useful because it holds bone, joint and animation data. For more information see Converting from other Formats.

<<prev top next>> Search

Panda3D Manual: More Panda Resources

Panda Specific Resources

The BVW Panda Tutorials. This site has the tutorials that are used in Carnegie Mellon's Building Virtual Worlds class.

Alice Gallery. This site holds many of the models created for use with Alice but exported into the .egg format. Be advised though, some of these models may not work properly.

Python Libraries

Pygame. GNU LGPL. Pygame is a set of Python modules designed for writing games. It includes Python bindings for SDL. Recommended for joystick support. Sound support is a free alternative to FMOD, though not a as capable (no 3D stereo sound for instance)

PyODE. GNU LGPL, *or* BSD-style license. Python bindings for The Open Dynamics Engine. ODE is a Real-time rigid body dynamics/collision detection physics engine. ODE is much more capable than Panda3D's built-in physics engine.

Psyco. MIT-style license. Python JIT optimizer. Can often make Python code run *faster* than code written in C (on Intel compatible chips.) Panda3D is written in C, but programed in Python. If Python function calls become a bottleneck, try using Psyco.

Twisted MIT-style license. An event-driven networking framework.

PyOpenAL GNU LGPL. PyOpenAL is a binding of OpenAL for Python. OpenAL is a cross-platform 3D audio API. This is another alternative to FMOD.

Useful Tools

Blender. 3D modeling and animation. Extensible in Python. Very fast user interface. Egg exporters are in the works.

The GIMP. A very capable free software raster image editor. Useful for converting image formats, creating and editing textures.

SWIG. A software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python.

Panda3D Manual: FAQ

Panda FAQ

Note: Many of these issues came up as a result of issues that popped up in the Building Virtual Worlds class. The class uses Maya 6.0 for modelling.

Q: I have a bunch of Maya Animations of one model in different mb files. I used maya2egg6 to port them into panda, but only one of the animations work.

A: The key is to use the -cn <character's name> flag in maya2egg6 for every file. This ensures that the files work together.

```
Lets say you are making an animated dog.
You have the following animations:
  dog-walk.mb
  dog-sit.mb
  dog-run.mb

To convert these into panda, you would call

maya2egg6 dog-walk.mb -a model -cn dog -o dog-model.egg

Note, we can grab the model from any of the animations, as long as they are all using the exact same rig


maya2egg6 dog-walk.mb -a chan -cn dog -o dog-walk.egg
maya2egg6 dog-sit.mb -a chan -cn dog -o dog-sit.egg
maya2egg6 dog-run.mb -a chan -cn dog -o dog-run.egg
```

Q: I'm using the lookAt function on a nodepath to point it at another object. It works fine until I point upwards, and then it starts to spin my object around randomly

A: lookAt works as long as you aren't telling it to look in the direction of its up vector. Luckily, you can specify the up vector as the second argument.

```
lookAt(object,Vec3(0,0,1))
```

Q: I'm building a 3d game, and I have a huge world. When my world starts up, the program hangs for a few seconds the first time I look around. Is there any way to avoid this?

A: The problem is that panda can a while to prepare objects to be rendered. Ideally, you don't want this to happen the first time you see an object. You can offload the wait time to the beginning by calling:

```
#self.myWorld is a nodepath that contains a ton of objects
self.myWorld.prepareScene(base.win.getGsg())
#This will walk through the scene graph, starting at
#self.myWorld, and prepare each object for rendering.
```

Q: Is there a way to hide the mouse pointer so that it doesn't show up on my screen?

A: You can change to properties of the panda window so that it doesn't show the cursor:

```
    props = WindowProperties()
    props.setCursorHidden(True)
    base.win.requestProperties(props)
```

Panda3D Manual: Examples Contributed by the Community

These are some of the examples contributed by the community:

IPKnightly:

- First Panda3D Tutorial
- Second Panda3D Tutorial
- Third Panda3D Tutorial
- Fourth Panda3D Tutorial
- Fifth Panda3D Example

TipToe::

- Edge Screen Tracking

Networking client server examples follow.

Yellow and bigfoot29:

- Panda3D network example

bigfoot29:

- small Chatserver/client

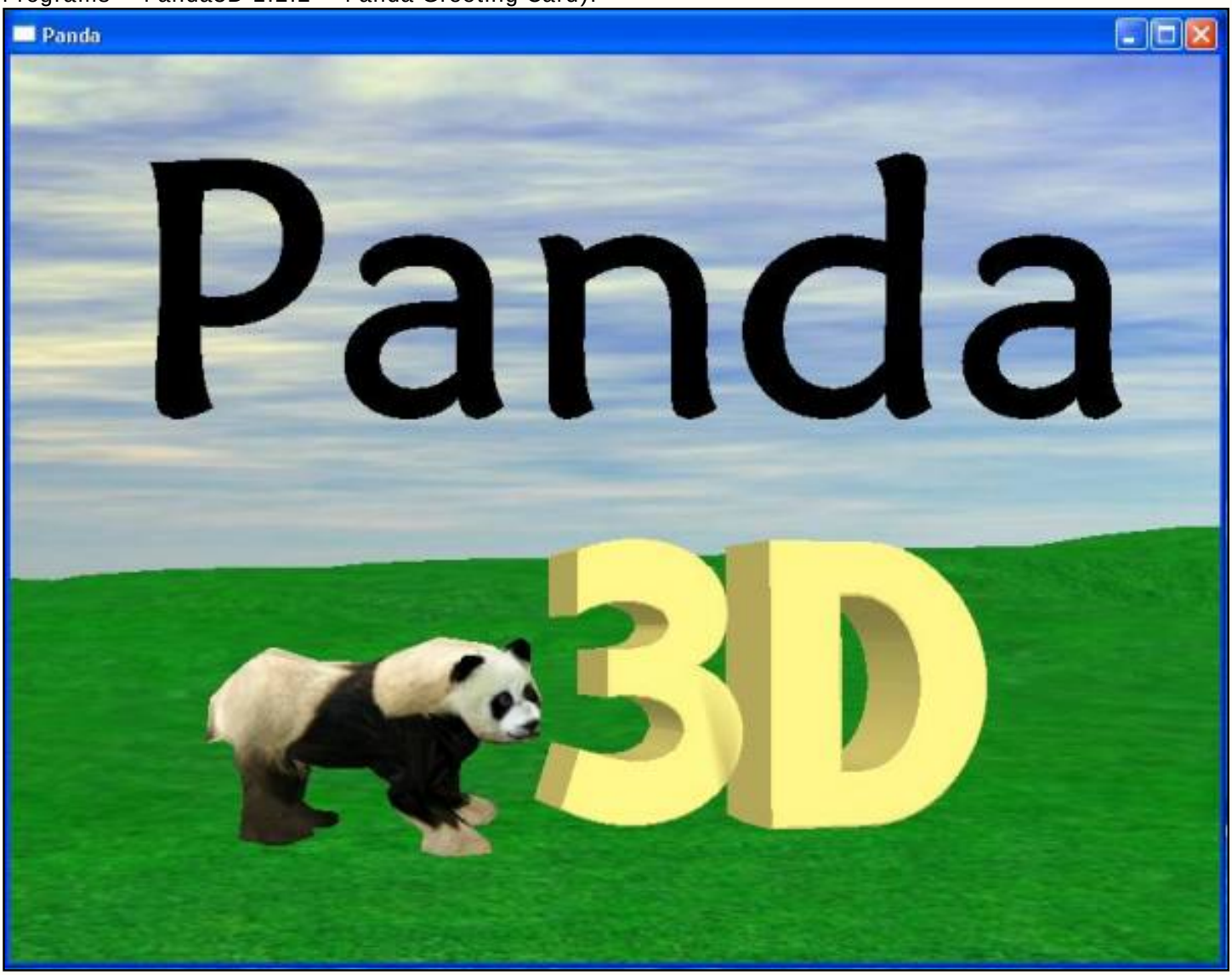
Panda3D Manual: Start Guide For The Absolute Beginner

Installing Panda3D and Creating A New Folder

1. First, go to the Downloads section and download the latest version of Panda3D (as of writing this is version 1.2.2). For Windows users download panda3d-1.2.2.exe.

2. When the download has finished, double-click the Panda3D icon to begin the installation process.

3. When finished, make sure that Panda3D has been successfully installed by running the â€˜Panda Greeting Cardâ€™ program (if it does not run automatically when the installation finishes, click Start > All Programs > Panda3D 1.2.2 > Panda Greeting Card).

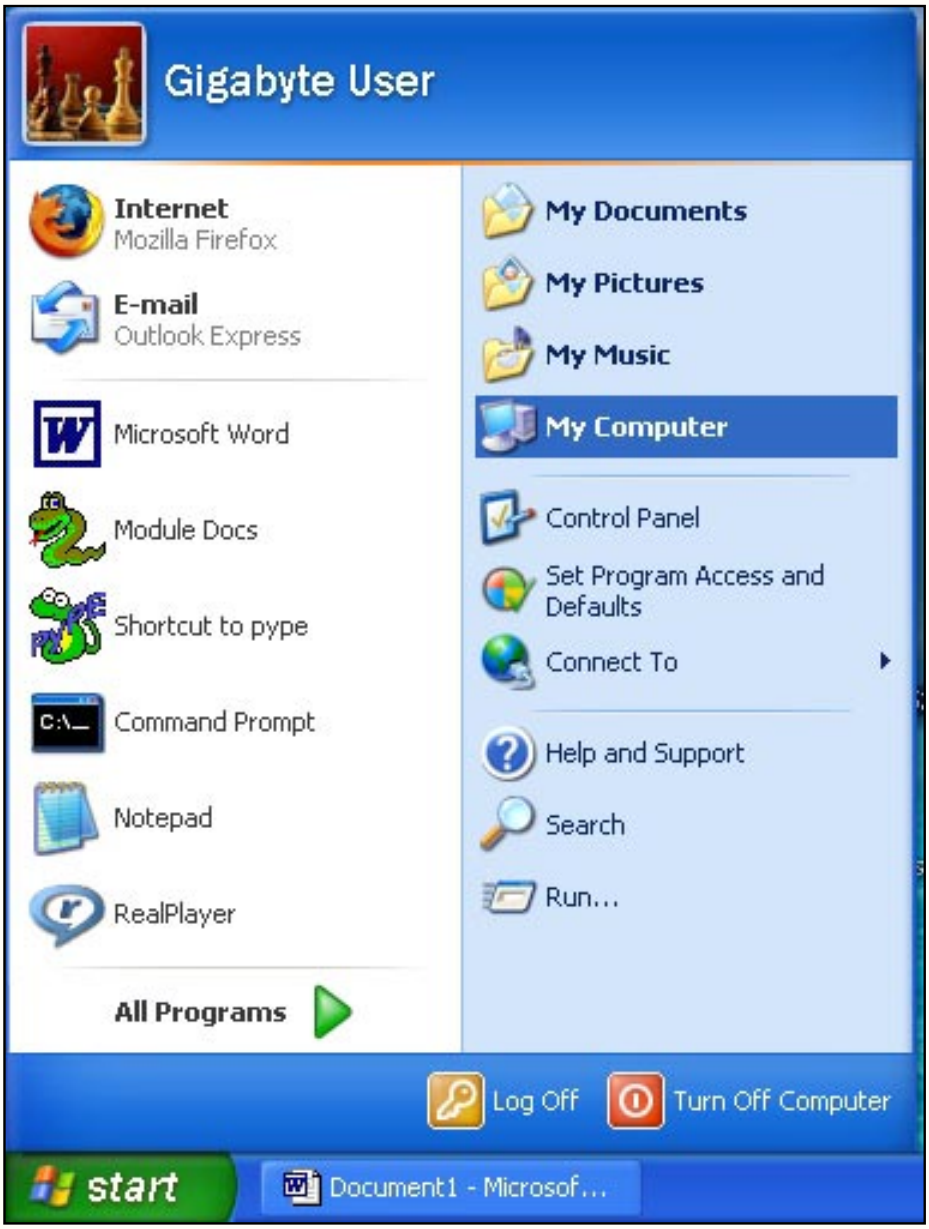


4. Now, a very important thing to note, Panda3D is a development tool; NOT an application. What does this mean? It means, that unlike all the other programs on your computer, you wonâ€™t find any program icons or shortcuts to start it. To run Panda3D you must write a python script which tells it what to do (this is a great way to learn programming, because youâ€™re basically starting from the ground up).

5. So, you now need to create a place where you can save all your scripts. Panda3D is very clever, it can run a script from anywhere on your computer, but it looks for models and other assets in the folder that it

is run from. If those assets arenâ€™t present, then youâ€™ll get an error message when you try to run it. So, I believe that the easiest thing for the beginner to do, is to make a new folder in the main Panda3D directory itself (which already contains all the models and other assets that youâ€™ll need).

6. To do this, click your computers â€˜Startâ€™ button, then â€˜My Computerâ€™.



Then double-click â€˜Local Disk (C)â€™ (if the contents of this drive are hidden, just click â€˜Show the contents of this folderâ€™). Now find the Panda3D-1.2.2 folder and double-click on it to open it.

7. On the top menu bar click â€˜File > New > Folderâ€™. This will create a new folder in the Panda3D directory, backspace out the name and type a new name for it (I called mine mystuff). Good! Youâ€™re almost ready to begin.

Missing image
Pic021ey.jpg
Image:pic021ey.jpg

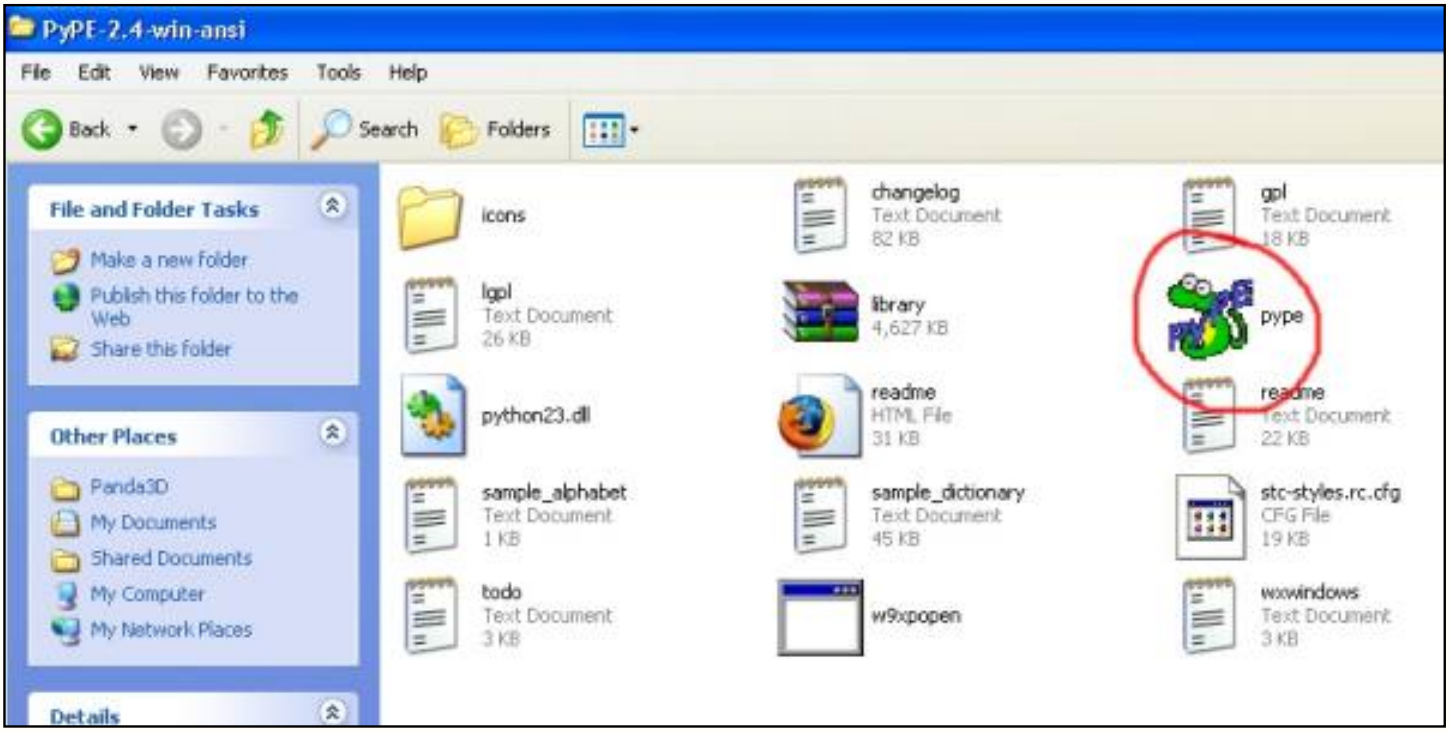
Writing Your First Script

Download a Python Editor

One of pythonâ€™s strong points, is that you donâ€™t need a fancy or expensive compiler to write your scripts. Python scripts can be written in a simple text editor such as â€˜Notepadâ€™ (which is already installed on all Windows computers). However, because Notepad isnâ€™t really designed for writing scripts, it will make learning to program much, much harder. So, I think the best thing you can do, is to use a proper python editor. I use PyPE (which is completely free and makes writing python scripts a whole lot easier) here's the link:

http://sourceforge.net/projects/pype/

If you're using Windows, download the PyPE 2.4-win-anzi.zip (as of writing this was the latest version). You don't need to install this program, you just download and unzip it somewhere, then open the unzipped folder and double-click on the PyPE icon to run it (or right-click on the icon and send it to the desktop as a shortcut, then simply run it by double-clicking the icon on your desktop).



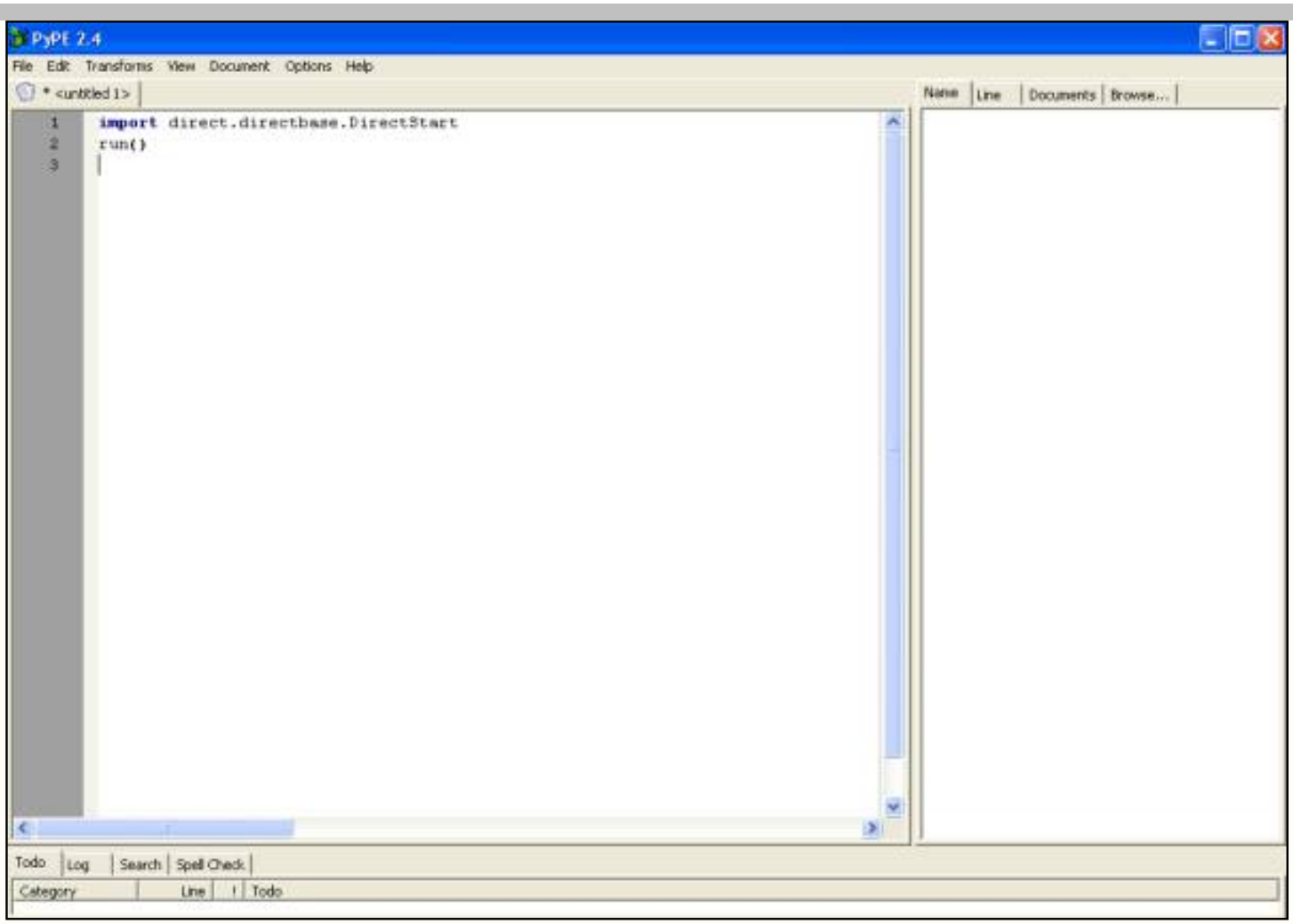
Write the script

1. Open PyPE, then click â€˜File > Newâ€™ on the top menu bar to open a new work environment.

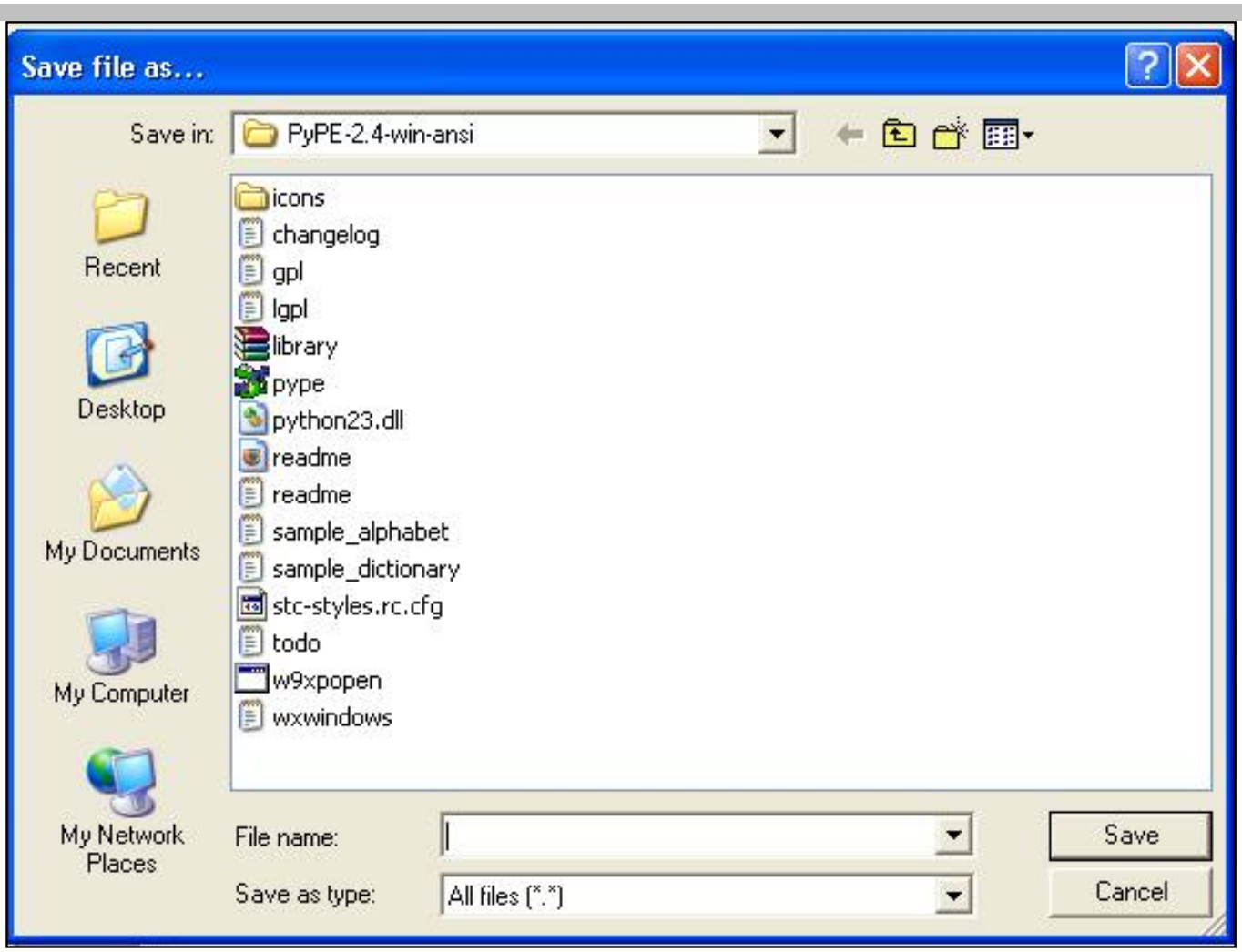
2. Now type the following code (or just copy and paste it):

```
import direct.directbase.DirectStart run
()
```

(Notice that PyPE automatically highlights certain words and numbers the lines for you. This is a very nice feature which makes finding errors much easier Very Happy).

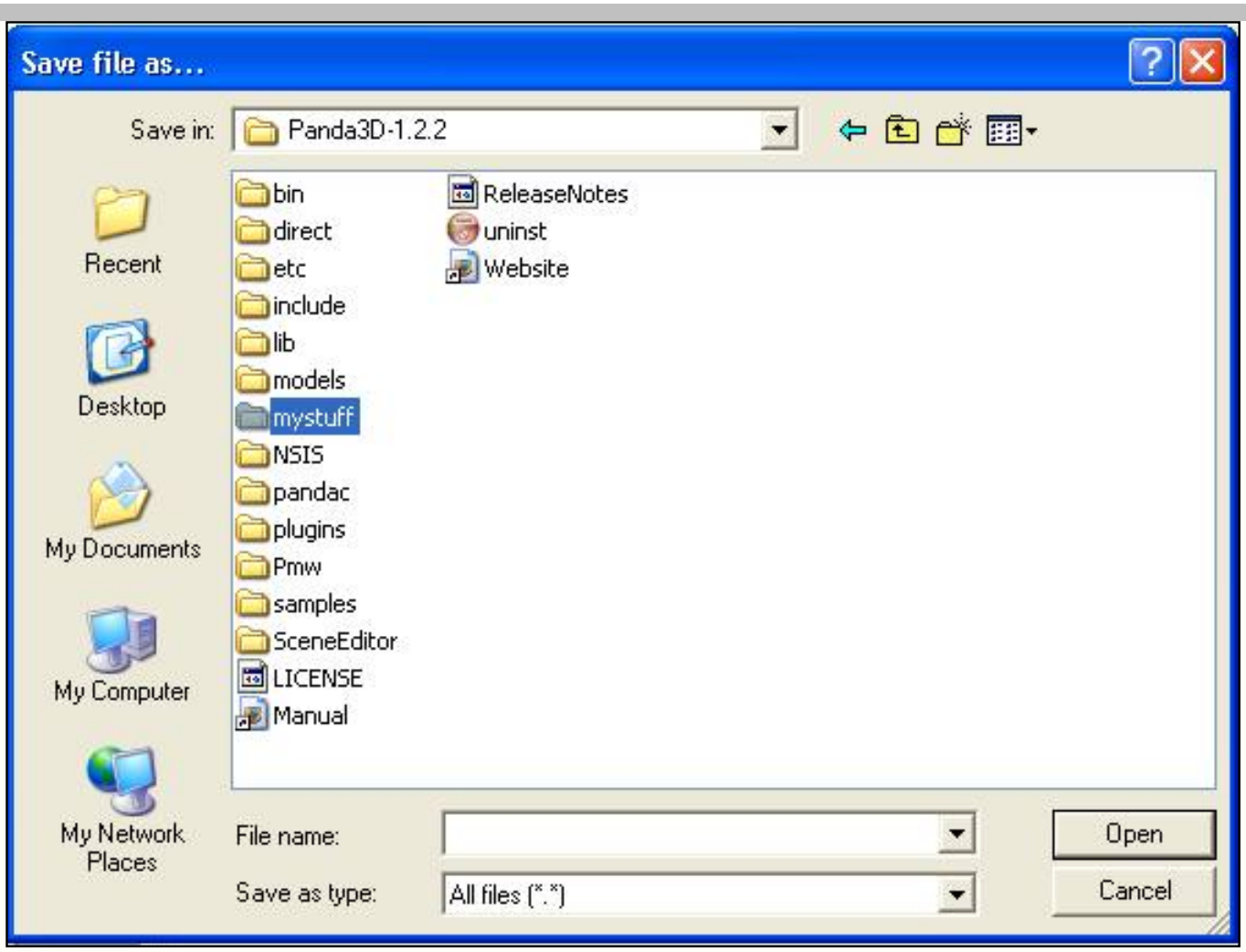


3. Well done! Youâ€™ve just written your first Panda3D script Very Happy. Itâ€™s not much, but those few lines of code tell Panda3D to start. But before you can run this script, you must save it. So click â€˜File > Save Asâ€™ on the top menu bar and a new window should open.

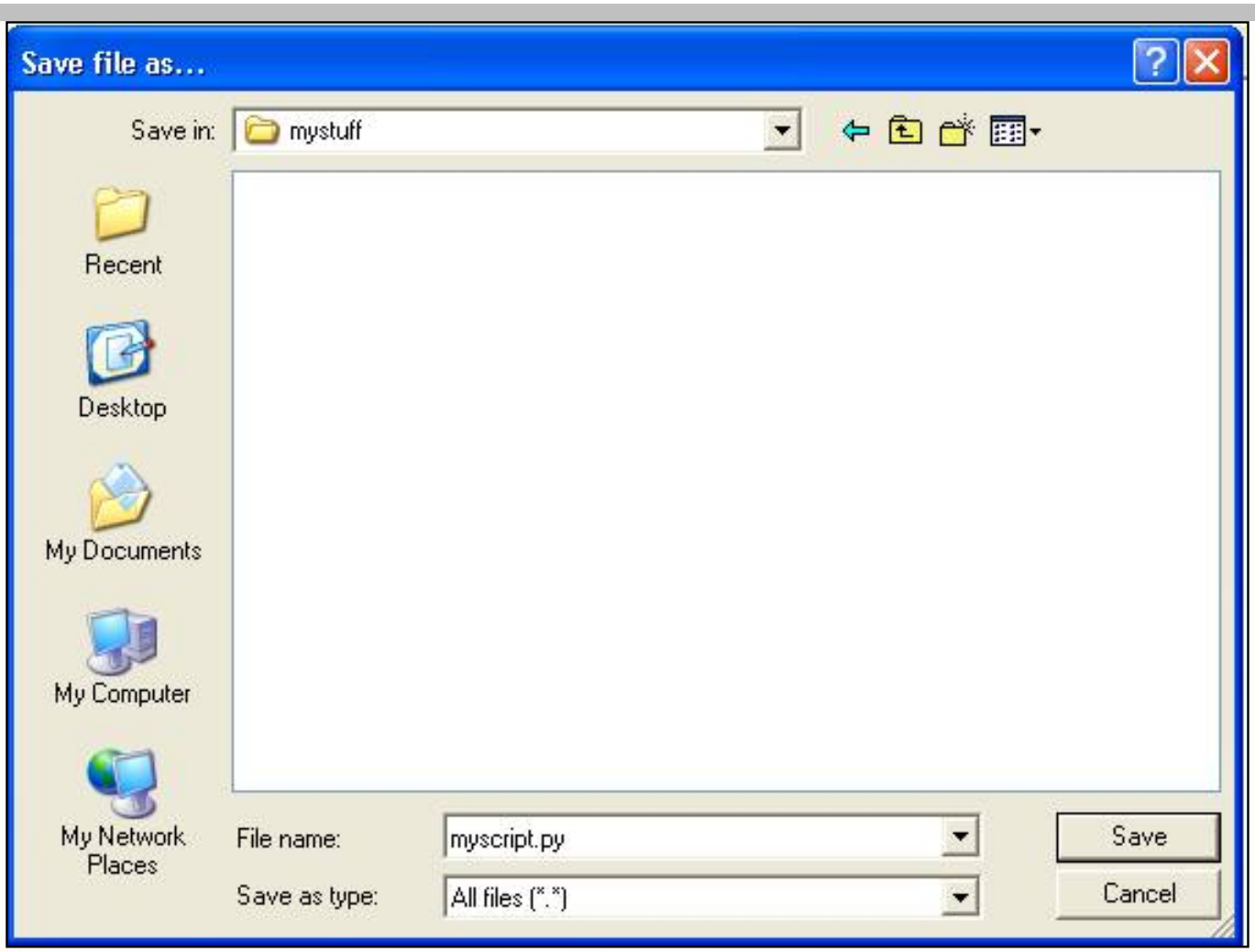


4. At the top of this new window there is a â€˜Save in:â€™ text box, which is pointing to â€˜PyPE-2.4-win-ansiâ€™, you DONâ€™T want to save your script there, so click the little down arrow beside the text box, then scroll down the list and click â€˜Local Disk (C)â€™, then double-click on the â€˜Panda3D-1.2.2â€™ folder to open it.

5. Find the â€˜mystuffâ€™ folder that we created earlier and double-click on it to open it.



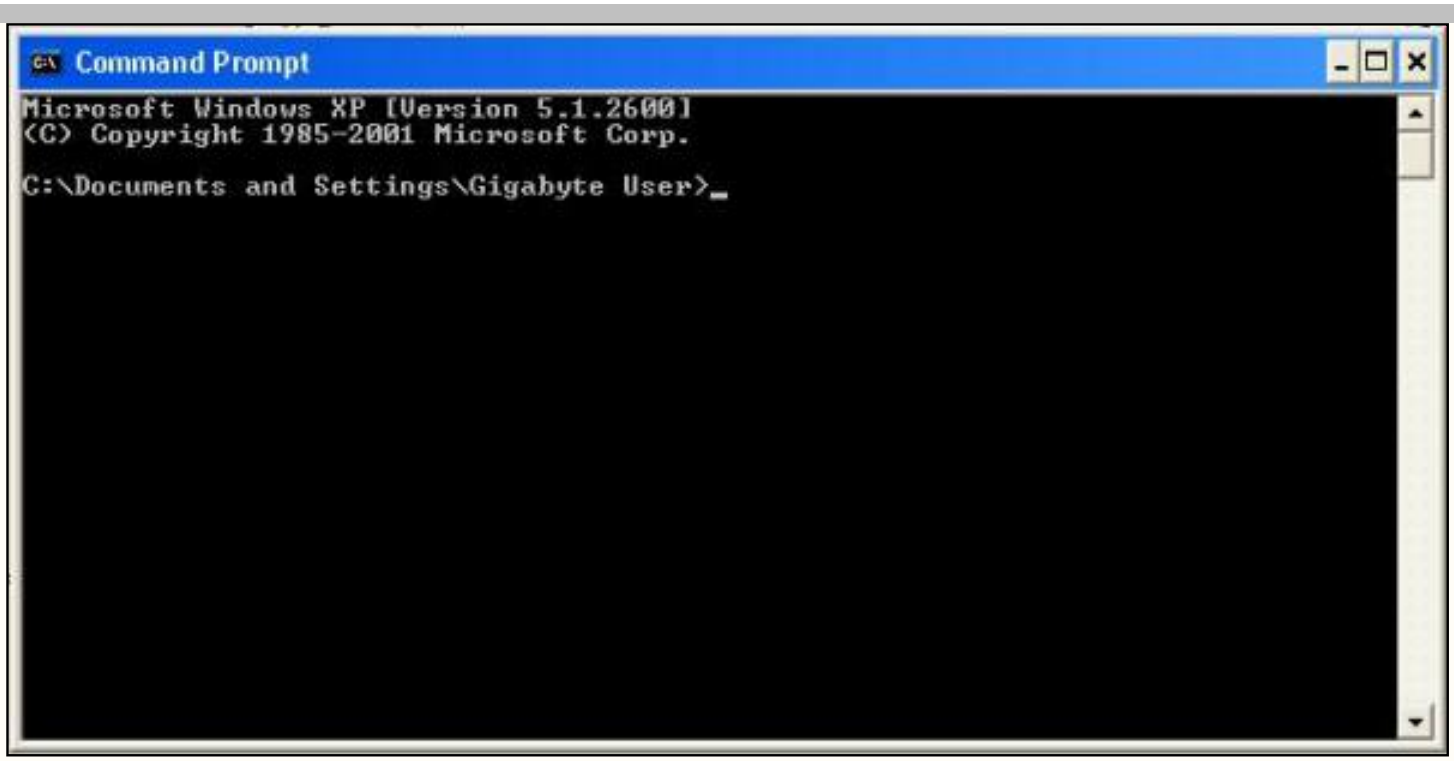
6. Good! Now type a name for your script in the â€˜File name:â€™ text box. I called mine myscript.py (make sure you put .py on the end of the name). Then click the â€˜Saveâ€™ button.



7. Very good! You can now close PyPE.

Running Your Script and Starting Panda3D

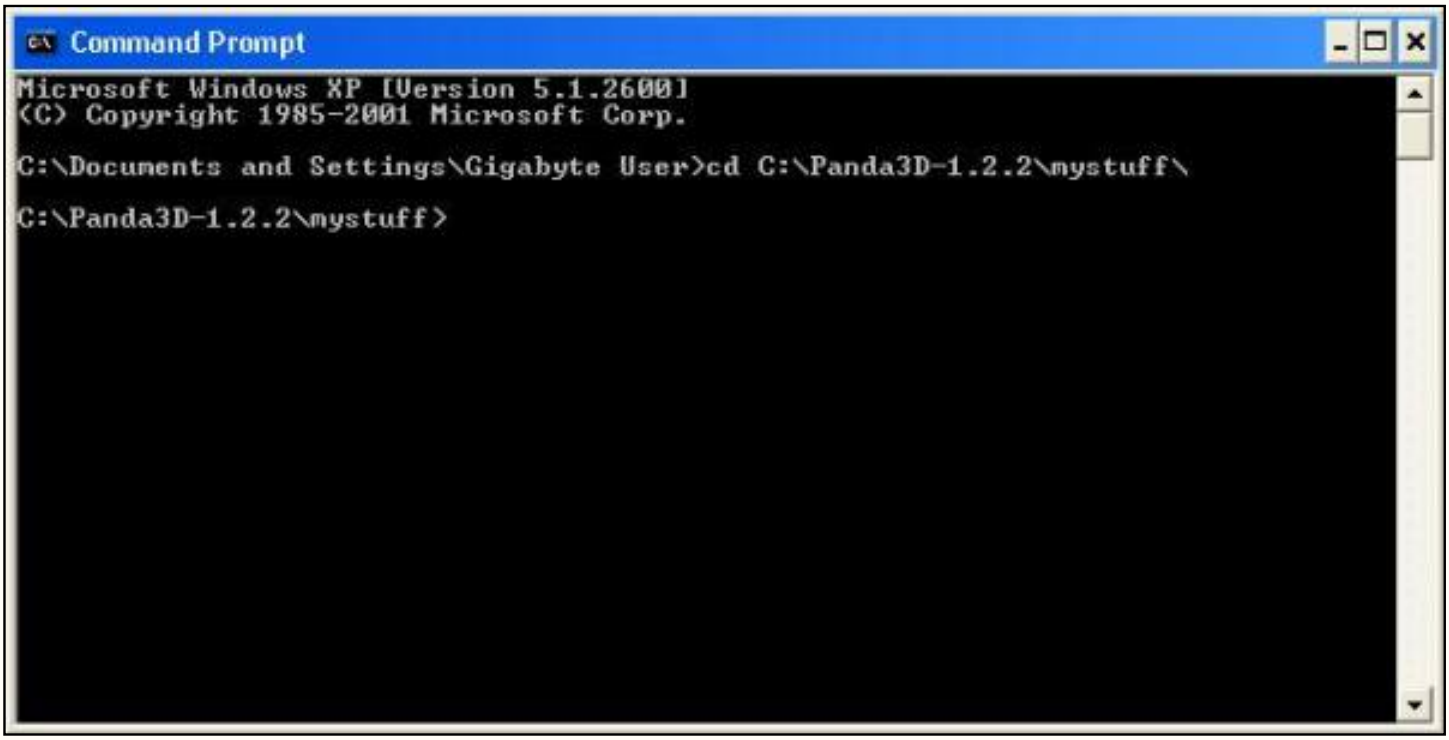
1. You now run your script by using your computerâ€™s â€˜Command Promptâ€™. You access this by clicking â€˜Start > All Programs > Accessories > Command Promptâ€™. When it opens, it should look something like this:



2. At the moment itâ€™s pointing to its default directory, which in my case is â€˜Documents and Settingsâ€™ (it doesnâ€™t matter if yours is different). We need to change the directory to the one where we saved our script. To do this, we type cd. This stands for â€˜change directoryâ€™. So type the following text behind the > symbol.

```
cd C:\Panda3D-1.2.2\mystuff\
```

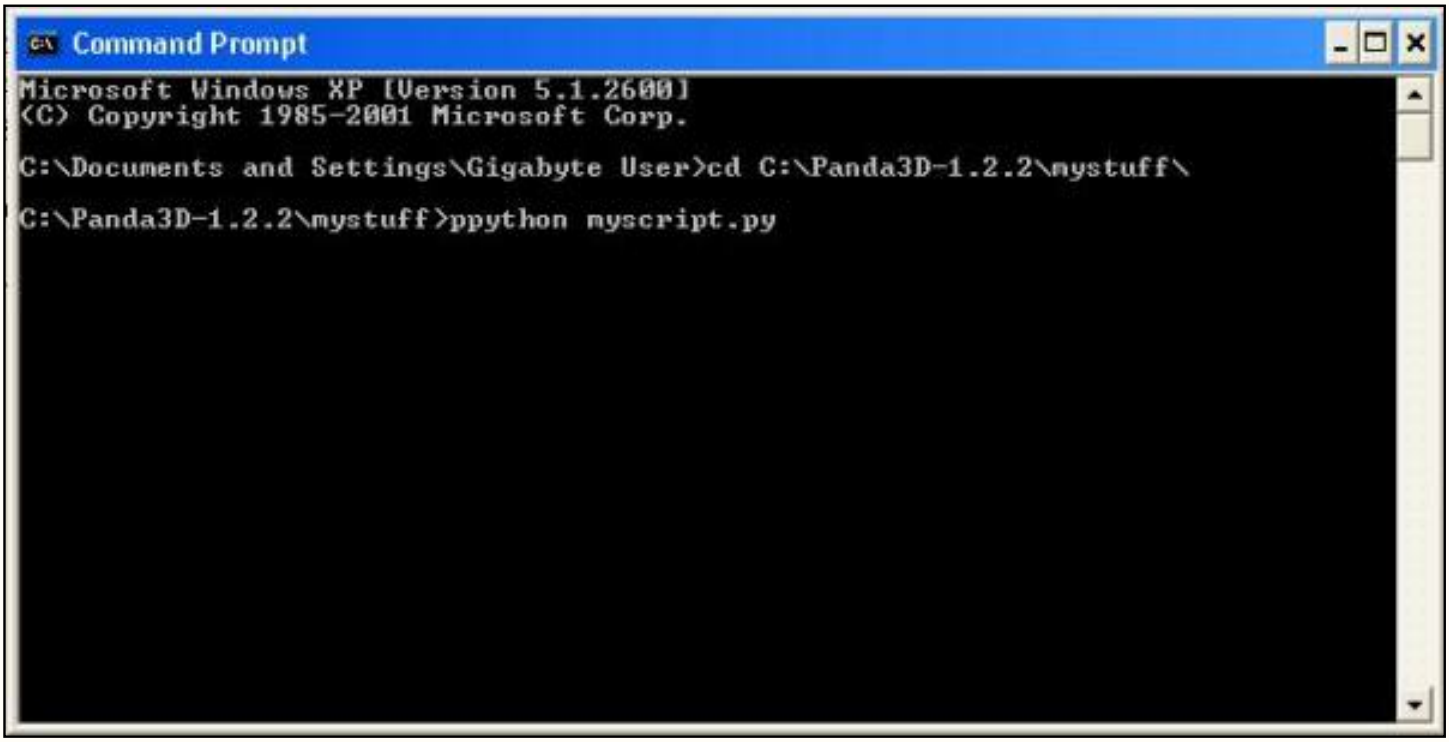
Please note that itâ€™s case sensitive and must match exactly. Then press the â€˜Enterâ€™ key on your keyboard. You should now have the following on the Command Prompt:



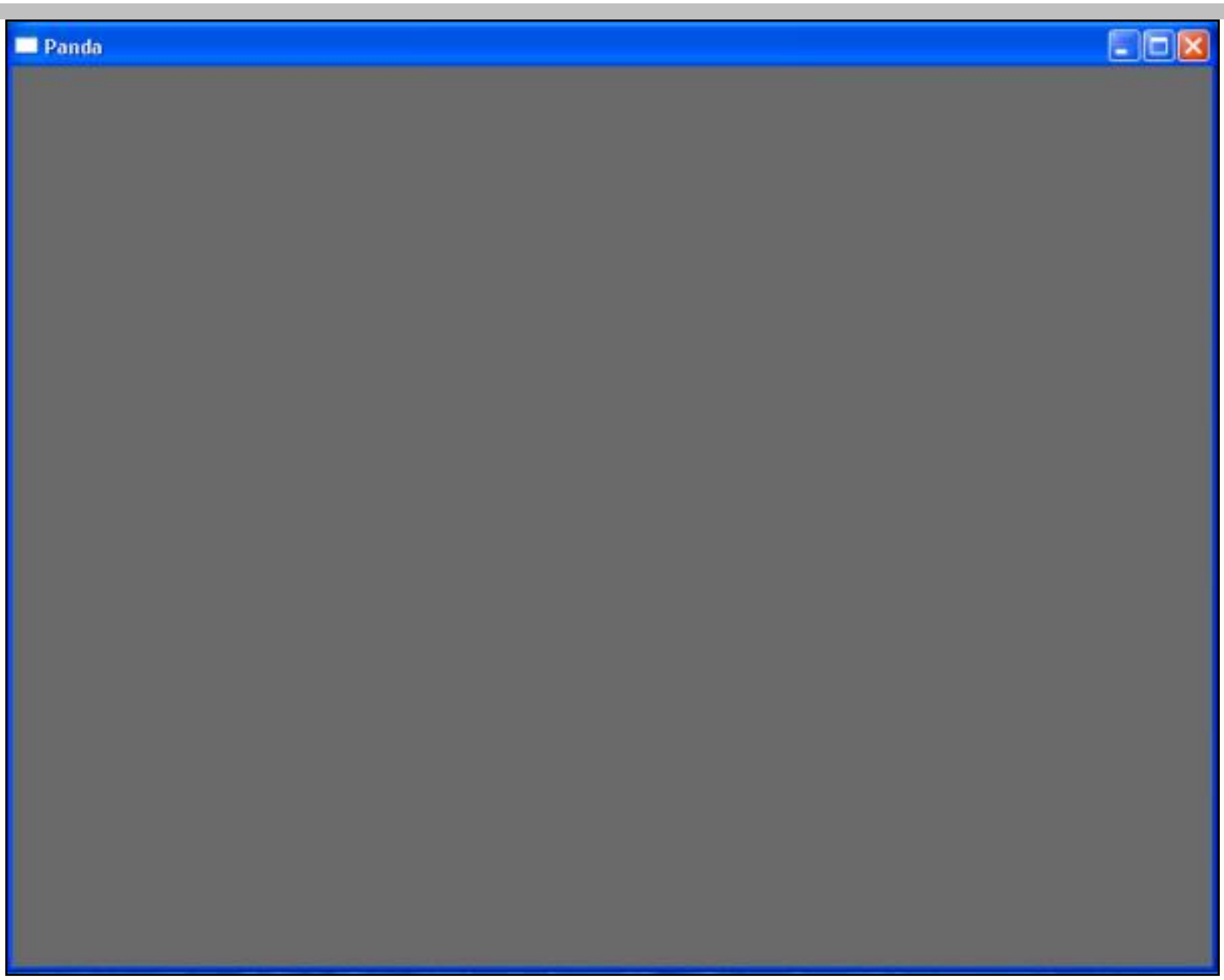
3. Good! This means that itâ€™s now pointing to the right directory. To run your script, and start Panda3D, type the following text behind the > symbol:

```
ppython myscript.py
```

Make sure you type ppython (the extra â€˜pâ€™ tells it to use the special Panda3D version of python and not just the regular version of python).



4. Now press the â€˜Enterâ€™ key on your keyboard. If all is well, Panda3D will start and you should see the main rendering window appear.



This is a empty program it won't do anything. Now you hopefully understand how to write a panda programm

Panda3D Manual: Configuring Panda

In the *etc* subdirectory, you will find a configuration file *Config.prc*. This controls several of Panda's configuration options - does it use OpenGL or DirectX, how much debugging output does it print, and so forth. The following table lists several of the most commonly-used variables.

Variable	Values	Default	Details
load-display	pandagl pandadx8	pandagl	Specifies which graphics GSG to use for rendering (OpenGL or DirectX 8)
win-width	Number of pixels	640	Specifies the width of the Panda3D window
win-height	Number of pixels	480	Specifies the height of the Panda3D window
win-origin-x	Number of pixels	100	Specifies the x dimension of the upper left corner of Panda3d window
win-origin-y	Number of pixels	100	Specifies the y dimension of the upper left corner of Panda3d window
fullscreen	#t #f	#f	Enables full-screen mode (true or false)
undecorated	#t #f	#f	Removes border from window (true or false)</\ td>
cursor-hidden	#t #f	#f	Hides mouse cursor (true or false)
show-frame-rate-meter	#t #f	#f	Shows the fps in the upper right corner of the screen (true or false)</\ td>
audio-cache-limit	number	32	limits the number of sounds you can load
notify-level-[package]	fatal error warning info debug spam	info	Sets notification levels for various Panda3D packages to control the amount of information printed during execution (fatal being least, spam being most)
model-path	Path string	480	Adds specified path to the list of paths searched when loading a model

texture-path	Path string	480	Adds specified path to the list of paths searched when loading a texture
sound-path	Path string	480	Adds specified path to the list of paths searched when loading a sound
load-file-type ptloader		Enabled	Allows the loading of file types for which converters have been written for in pandatool
audio-library-name	fmod_audio miles_audio null	fmod_audio	Loads the appropriate audio drivers. Miles is a propriertary audio, so only select that option if you currently have it.
want-directtools	#t #f	#t line commented out	Enables directtools, a suite of interactive object/camera manipulation tools
want-tk	#t #f	#t line commented out	Enables support for using Tkinter/PMW (Pythonâ€™s wrappers around Tk)

top

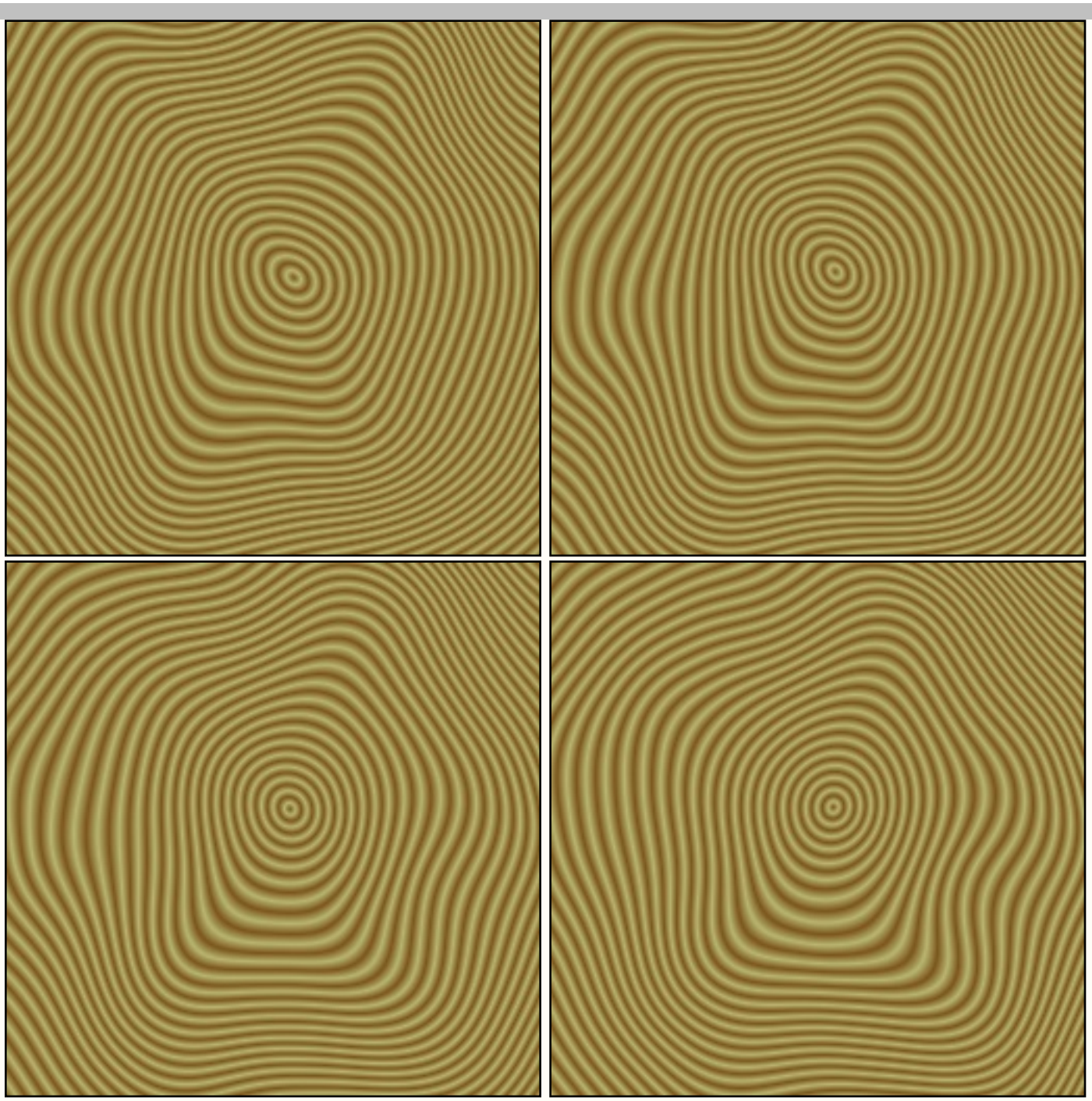
Panda3D Manual: Woodgrain Example

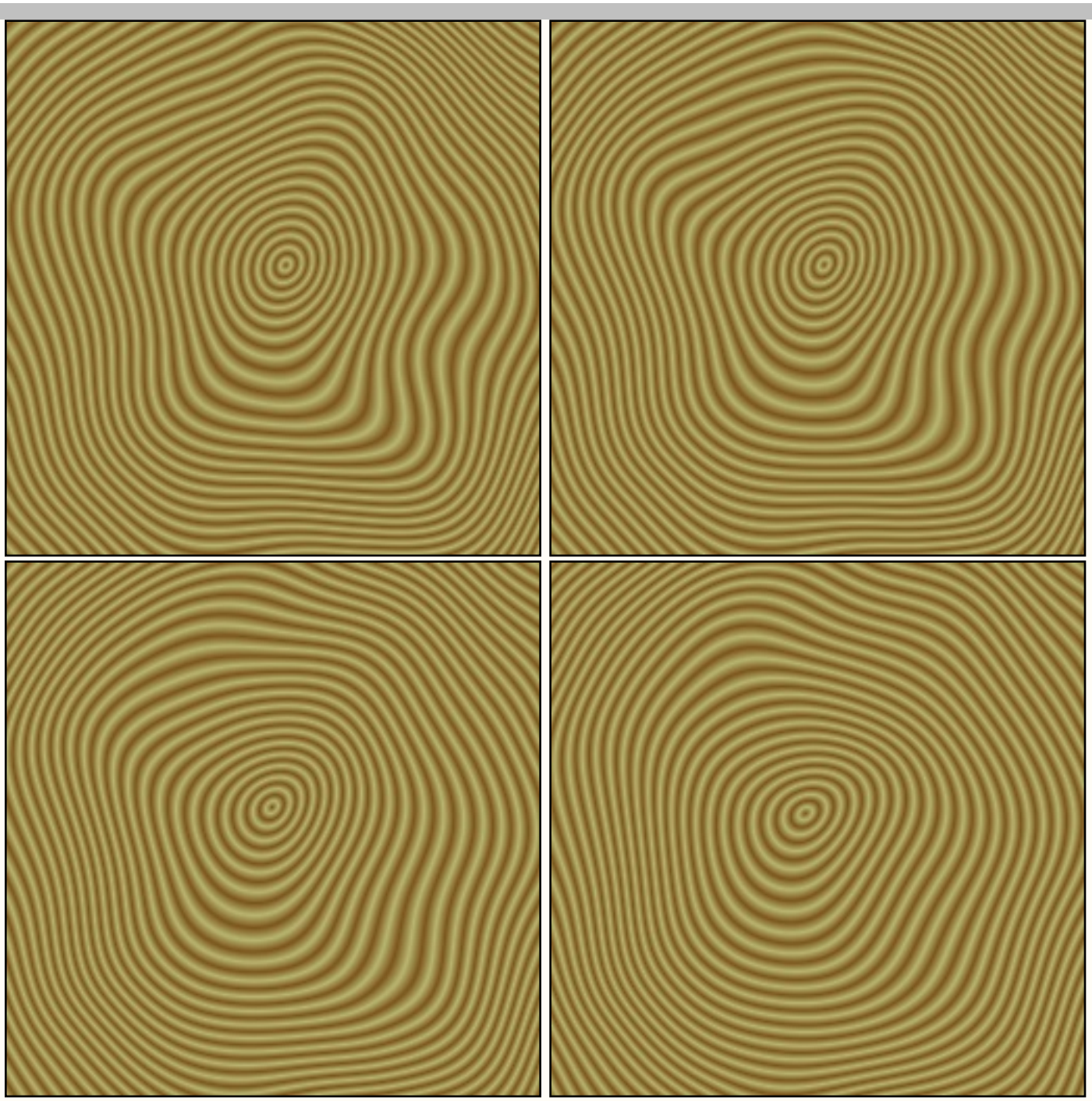
The following program will generate and write out a 3-D texture to simulate woodgrain:

```
from direct.directbase.DirectStart import *
from pandac.PandaModules import *
import math

# These constants define the RGB colors of the light and dark bands in
# the woodgrain.
lightGrain = (0.72, 0.72, 0.45)
darkGrain = (0.49, 0.33, 0.11)

def chooseGrain(p, xi, yi, radius):
    """ Applies the appropriate color to pixel (xi, yi), based on
    radius, the computed distance from the center of the trunk. """

    # Get the fractional part of radius.
    t = radius - math.floor(radius)

    # Now t ranges from 0 to 1.  Make it see-saw from 0 to 1 and back.
    t = abs(t - 0.5) * 2

    # Now interpolate colors.
    p.setXel(xi, yi,
             lightGrain[0] + t * (darkGrain[0] - lightGrain[0]),
             lightGrain[1] + t * (darkGrain[1] - lightGrain[1]),
             lightGrain[2] + t * (darkGrain[2] - lightGrain[2]))

def calcRadius(xn, yn, x, y, z, noiseAmp):
    """ Calculates radius, the distance from the center of the trunk,
    for the 3-d point (x, y, z).  The point is perturbed with noise to
    make the woodgrain seem more organic. """

    xp = x + xn.noise(x, y, z) * noiseAmp
    yp = y + yn.noise(x, y, z) * noiseAmp

    return math.sqrt(xp * xp + yp * yp)

def makeWoodgrain(texSize, texZSize, noiseScale, noiseZScale,
                  noiseAmp, ringScale):

    """ Generate a 3-D texture of size texSize x texSize x texZSize
    that suggests woodgrain, with the grain running along the Z (W)
    direction.  Since there is not as much detail parallel to the
    grain as across it, the texture does not need to be as large in
    the Z dimension as in the other two dimensions.

    The woodgrain shape is perturbed with Perlin noise to make it more
    organic.  The parameters noiseScale and noiseZScale controls the
    frequency of the noise; larger numbers make smoother rings.  The
```

```
    parameter noiseAmp controls the effect of the noise; larger
    numbers make more dramatic distortions.

    ringScale controls the number of rings visible in the cross
    section of the texture.  A larger number makes more, denser rings.
    """

    # First, create the two PerlinNoise objects to perturb the rings
    # in two dimensions.  This class is defined in Panda3D.
    xn = PerlinNoise3(noiseScale, noiseScale, noiseZScale)
    yn = PerlinNoise3(noiseScale, noiseScale, noiseZScale)

    # Start by creating a empty 3-D texture.
    tex = Texture('woodgrain')
    tex.setup3dTexture()

    for zi in range(texZSize):
        z = float(zi) / float(texZSize - 1) - 0.5

        # Walk through the Z slices of the texture one at a time.  For
        # each slice, we create a PNMImage, very much as if we were
        # reading the texture from disk.
        print zi
        p = PNMImage(texSize, texSize)

        # But instead of reading the PNMImage, we fill it in with the
        # ring pattern.
        for yi in range(texSize):
            y = float(yi) / float(texSize - 1) - 0.5
            for xi in range(texSize):
                x = float(xi) / float(texSize - 1) - 0.5

                radius = calcRadius(xn, yn, x, y, z, noiseAmp)
                chooseGrain(p, xi, yi, radius * ringScale)

        # Now load the current slice into the texture.
        tex.load(p, zi)

    return tex

# Create a 3-D texture.
tex = makeWoodgrain(texSize = 256, texZSize = 8, noiseScale = 0.4,
                    noiseZScale = 0.8, noiseAmp = 0.12, ringScale = 40)

# Write out the texture.  This will generate woodgrain_0.png,
# woodgrain_1.png, and so on, in the current directory.
tex.writePages(Filename('woodgrain_#.png'))
```

The resulting images look like this:





top

Search

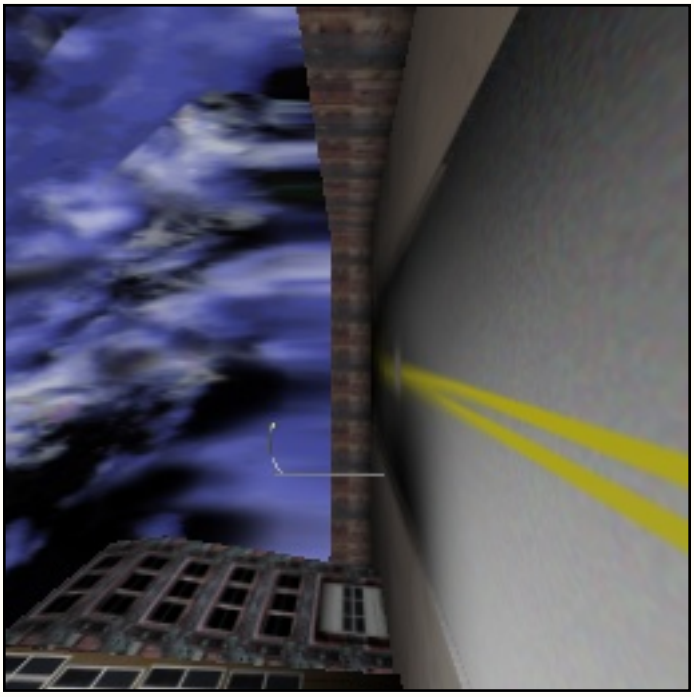
Panda3D Manual: Sample Cube Map

The following sample code loads up an environment, puts the camera in the center of it, and generates the six faces of a cube map from the point of view of the camera:

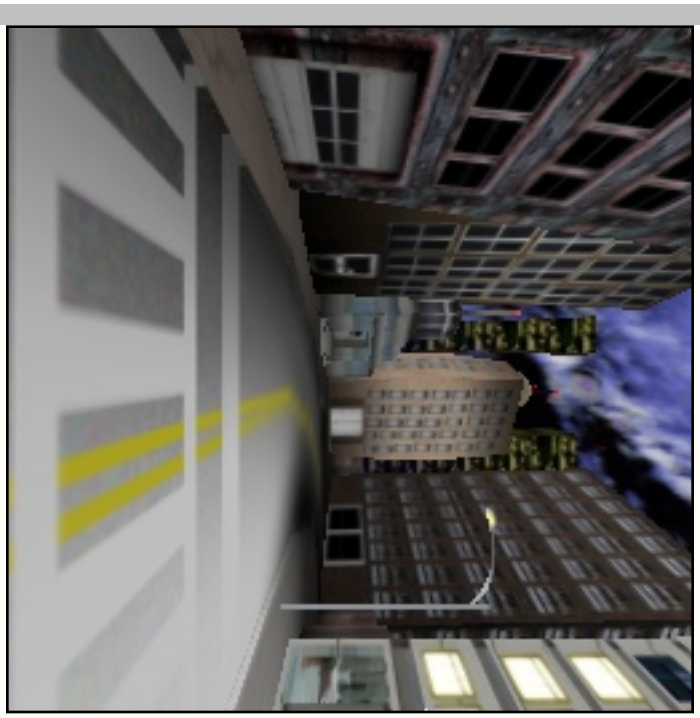
```
scene = loader.loadModel('bvw-f2004--streetscene/street-scene.egg')
scene.reparentTo(render)
scene.setZ(-2)
base.saveCubeMap('streetscene_cube_#.jpg', size = 256)
```

These are the six faces generated:

Right:

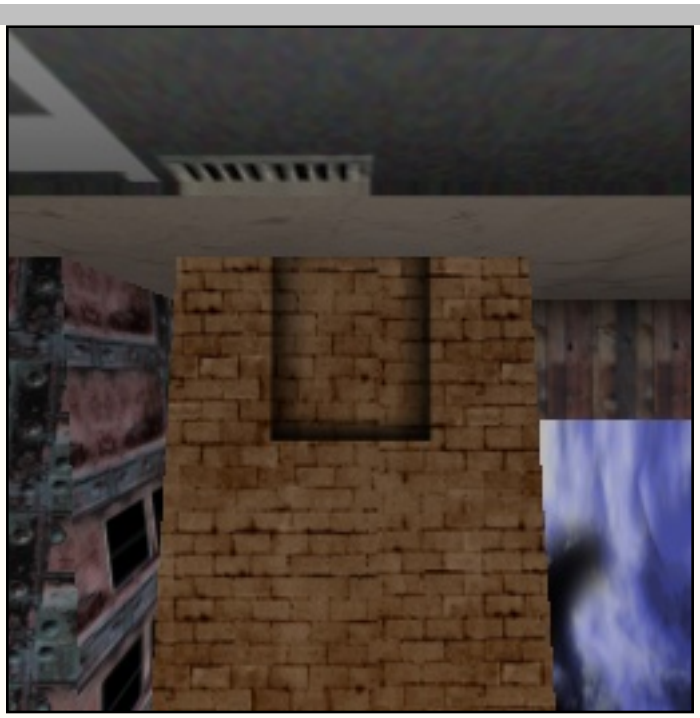


Left:

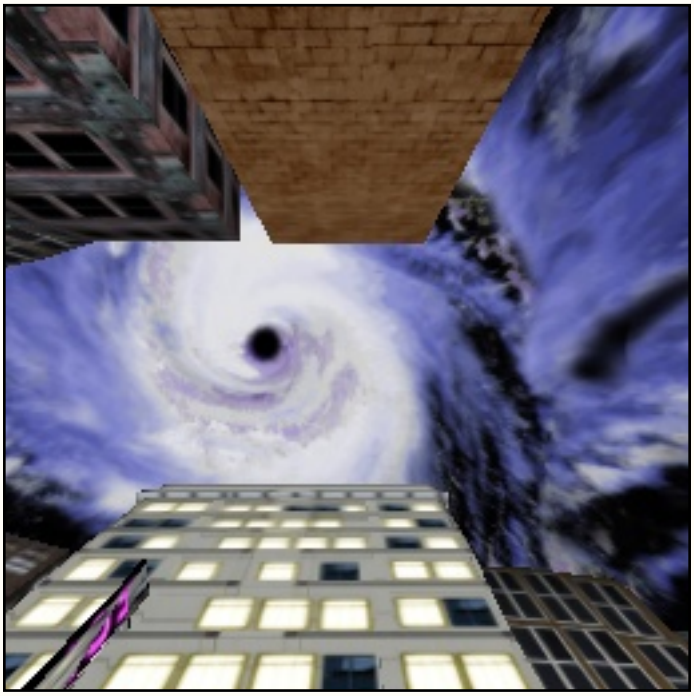


Front:

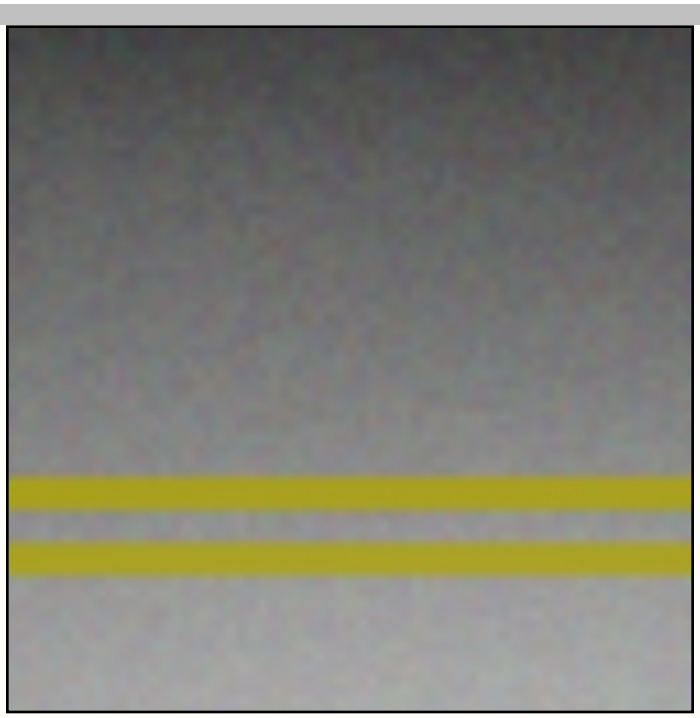
Back:



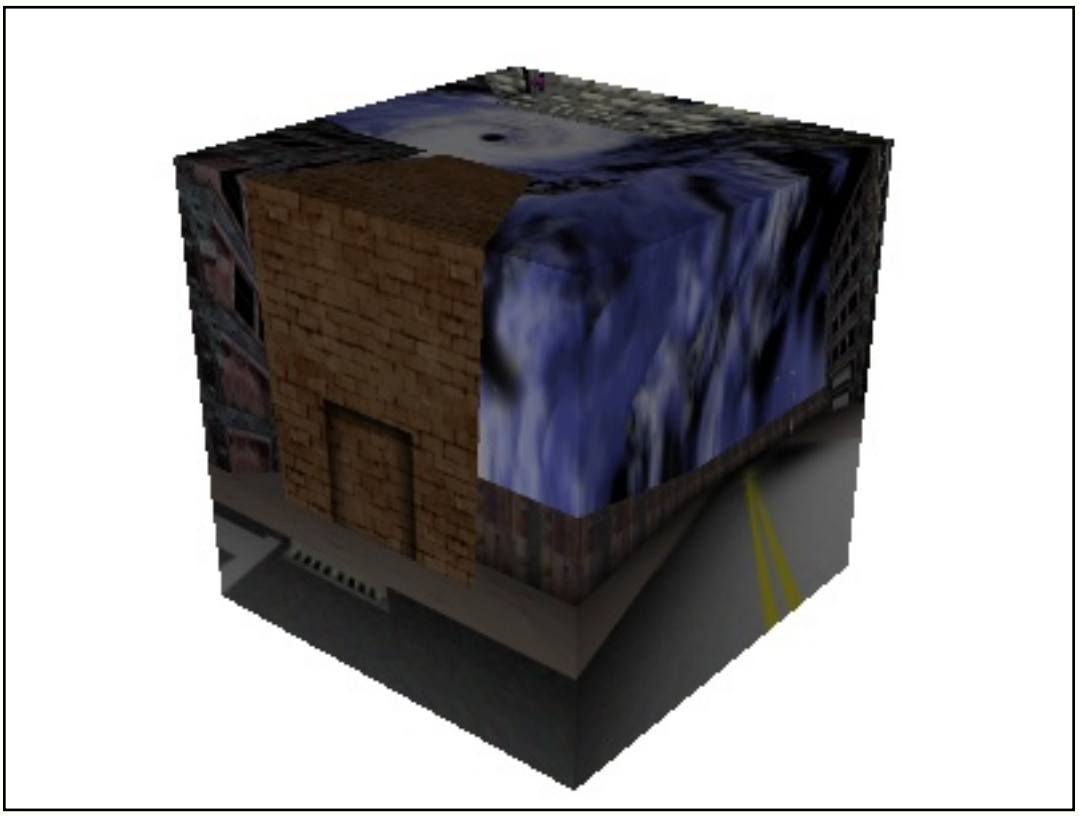
Top:



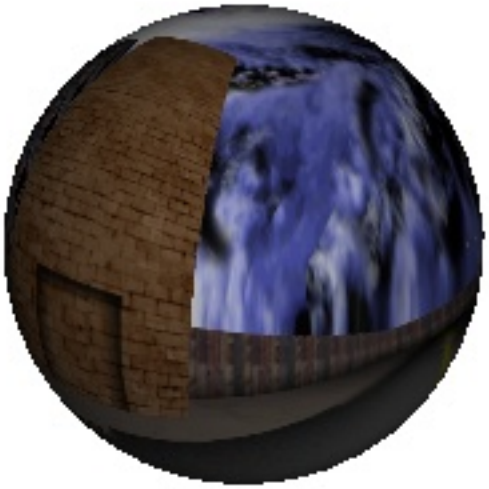
Bottom:



And when they are assembled into a cube map, it looks like this:



Or, when we apply that cube map to a sphere, you can see there are absolutely no seams between the edges:



top

Search

Panda3D Manual: Jam-O-Drum

<<prev top next>> Search

There is currently no text in this page, you can search for this page title in other pages or edit this page.

<<prev top next>> Search

Panda3D Manual: Debugging and Performance Tuning

There is currently no text in this page, you can search for this page title in other pages or edit this page.

Panda3D Manual: Search

There is no page titled "Jam-O-Drum". You can create this page.

For more information about searching Panda3D Manual, see Searching Panda3D Manual.

Showing below **1** results starting with #**1**.

View (previous 20) (next 20) (20 | 50 | 100 | 250 | 500).

No page title matches

Page text matches

1. Main Page (7,605 bytes)
   213: [[Jam-O-Drum]]

View (previous 20) (next 20) (20 | 50 | 100 | 250 | 500).

Search in namespaces:
☑ (Main) ☐ Talk ☐ User ☐ User talk ☐ Panda3D Manual ☐ Panda3D Manual talk
☐ Image ☐ Image talk ☐ MediaWiki ☐ MediaWiki talk ☐ Template ☐ Template talk
☐ Help ☐ Help talk ☐ Category ☐ Category talk
☐ List redirects
Search for Jam-O-Drum Search

Back to the Manual

Panda3D Manual: Search

There is no page titled "Debugging_and_Performance_Tuning". You can create this page.

For more information about searching Panda3D Manual, see Searching Panda3D Manual.

Showing below **0** results starting with #**1**.

No page title matches

No page text matches

Note: Unsuccessful searches are often caused by searching for common words like "have" and "from", which are not indexed, or by specifying more than one search term (only pages containing all of the search terms will appear in the result).

Search in namespaces:

☑ (Main) ☐ Talk ☐ User ☐ User talk ☐ Panda3D Manual ☐ Panda3D Manual talk
☐ Image ☐ Image talk ☐ MediaWiki ☐ MediaWiki talk ☐ Template ☐ Template talk
☐ Help ☐ Help talk ☐ Category ☐ Category talk
☐ List redirects

Search for Debugging_and_Pe Search

Back to the Manual