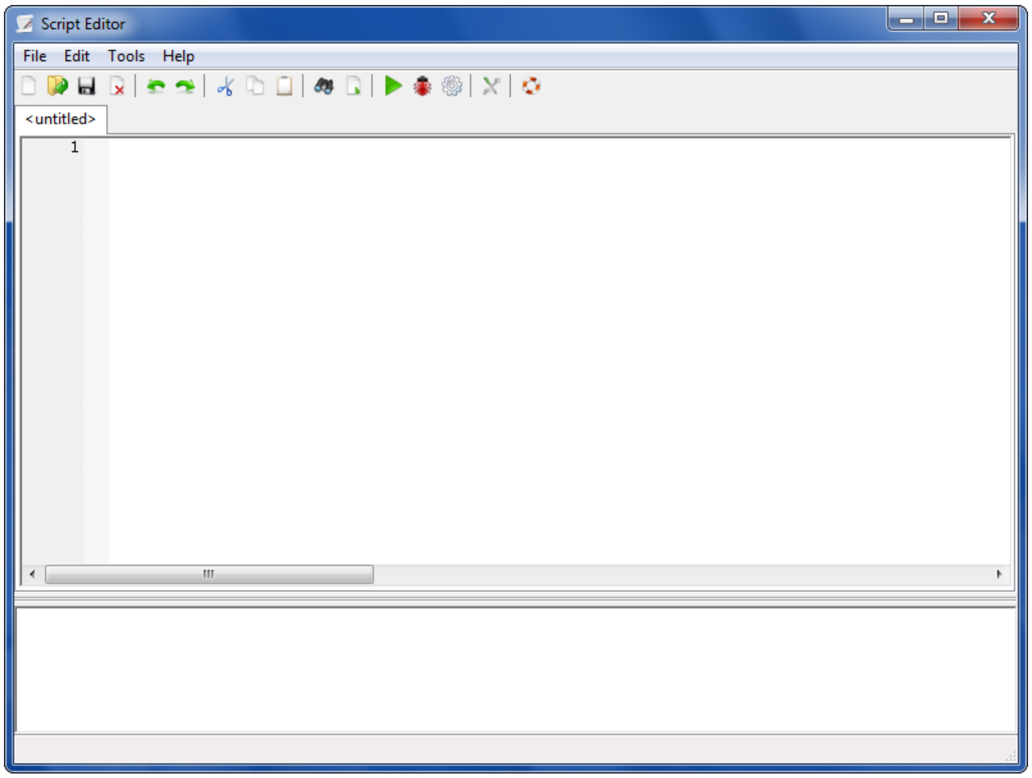
Getting Started with Lua

Leadwerks Engine utlitizes the Lua scripting language. Lua is used in many games including Crysis, S.T.A.L.K.E.R.: Shadow of Chernobyl, and Garry's Mod. More information about the Lua scripting language is available at www.lua.org.

Scripts in Leadwerks Engine can be run on both a global and per-model approach. The entire flow of a program can be written in script, or per-model scripts can be used with a game written in another programming language. In this lesson we will learn how to write a simple program using Lua script.

The Script Editor

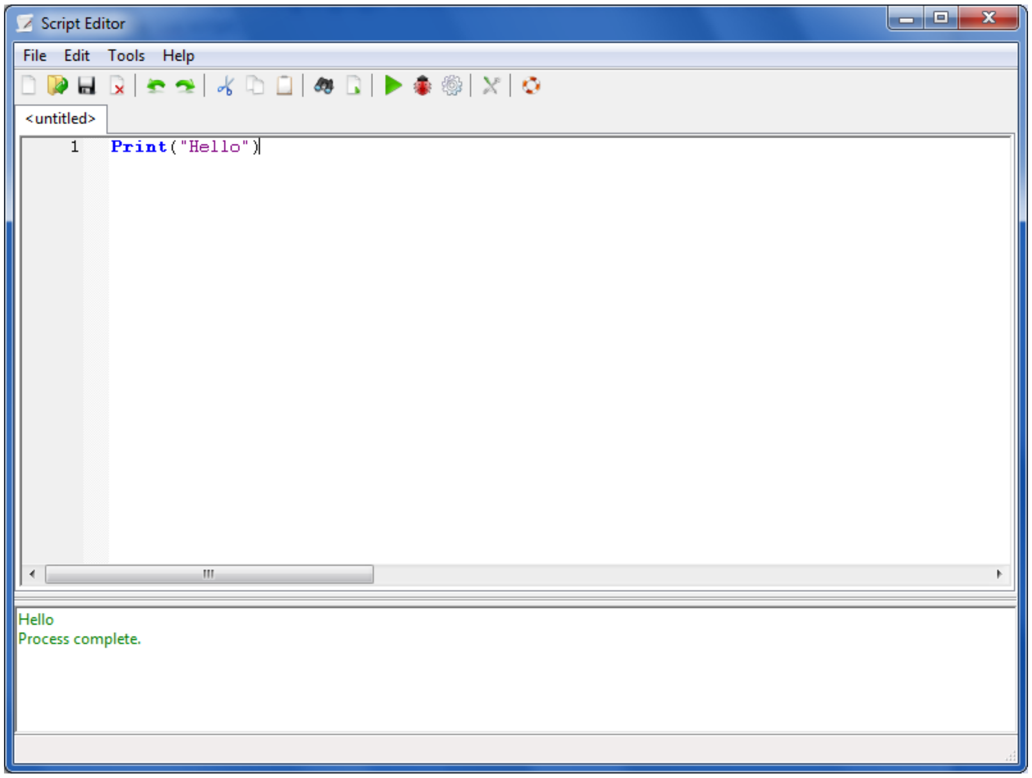
The script editor can be used to write, edit, run, and compile Lua scripts. When we open the script editor it starts with a blank text area:



Enter this text into the text area and press F5 to run the script:

```
Print("Hello")
```

The printed output will be visible in the output log at the bottom of the script editor window:



```
Print("Hello")
```

```
Hello
Process complete.
```

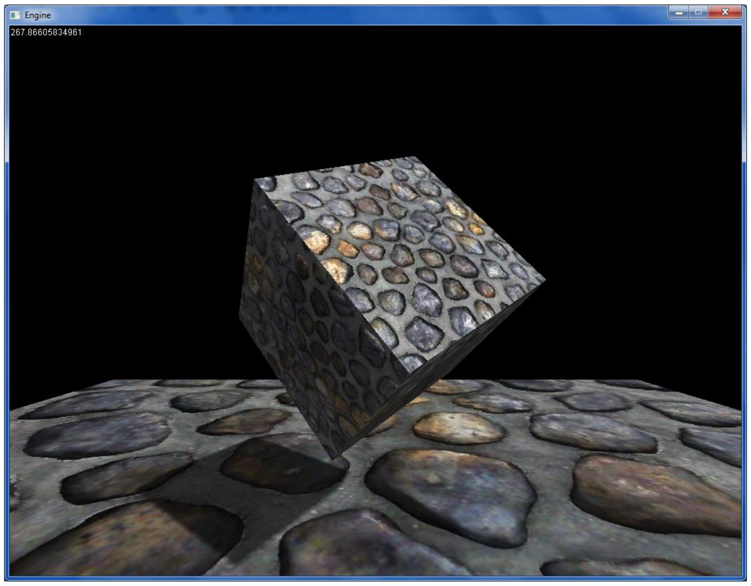
What is happening when we run the script? The script editor saves the script and launches the engine interpreter. The interpreter is a standalone application that loads a script and runs it. The interpreter can be used to make standalone games that are controlled entirely with script. The script editor saves all open scripts and launches the interpreter, passing the name of the selected script in the command line. If a script has not been saved yet and has no name, as our untitled script here does, the script editor will save the text in a temporary file with a random name. The temporary file will be deleted after the script program completes.

Now let's try a more advanced script. This code will create a graphics window, create some objects, and run a rendering loop until the user presses the escape key:

```
require("Scripts/constants/keycodes")

--Register abstract path
RegisterAbstractPath("")

--Set graphics mode
if Graphics(1024,768)==0 then
        Notify("Failed to set graphics mode.",1)
        return
end

world=CreateWorld()
if world==nil then
        Notify("Failed to initialize engine.",1)
```

```
	return
end

gbuffer=CreateBuffer(GraphicsWidth(),GraphicsHeight(),1+2+4+8)

camera=CreateCamera()
camera:SetPosition(Vec3(0,0,-2))

light=CreateSpotLight(10)
light:SetRotation(Vec3(45,55,0))
light:SetPosition(Vec3(5,5,-5))

material=LoadMaterial("abstract::cobblestones.mat")

mesh=CreateCube()
mesh:Paint(material)

ground=CreateCube()
ground:SetScale(Vec3(10.0,1.0,10.0))
ground:SetPosition(Vec3(0.0,-2.0,0.0))
ground:Paint(material)

light=CreateDirectionalLight()
light:SetRotation(Vec3(45,45,45))

while AppTerminate()==0 do

	if KeyHit(KEY_ESCAPE)==1 then break end

	mesh:Turn(Vec3(AppSpeed()*0.5,AppSpeed()*0.5,AppSpeed()*0.5))

	UpdateAppTime()
	world:Update(AppSpeed())

	SetBuffer(gbuffer)
	world:Render()
	SetBuffer(BackBuffer())
	world:RenderLights(gbuffer)

	DrawText(UPS(),0,0)
	Flip(0)
End
```

If you copy this text to the script editor and press F5, you will see something like this:



```
267.86605834961
```

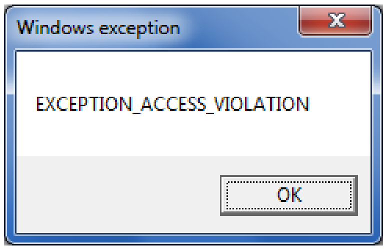
Debugging Scripts

We can also debug scripts. When we debug a script, it is run with the debugger instead of the interpreter. The debugger is an executable called engine.debug.exe. The debugger is the same as the interpreter, but it is compiled with debugging options enabled. The debugger runs more slowly than the interpreter, but it is better at catching errors.

Let's make a script that creates an error. This code will cause an error to occur, because the Point() method is being used to point the created mesh at an entity that doesn't exist:

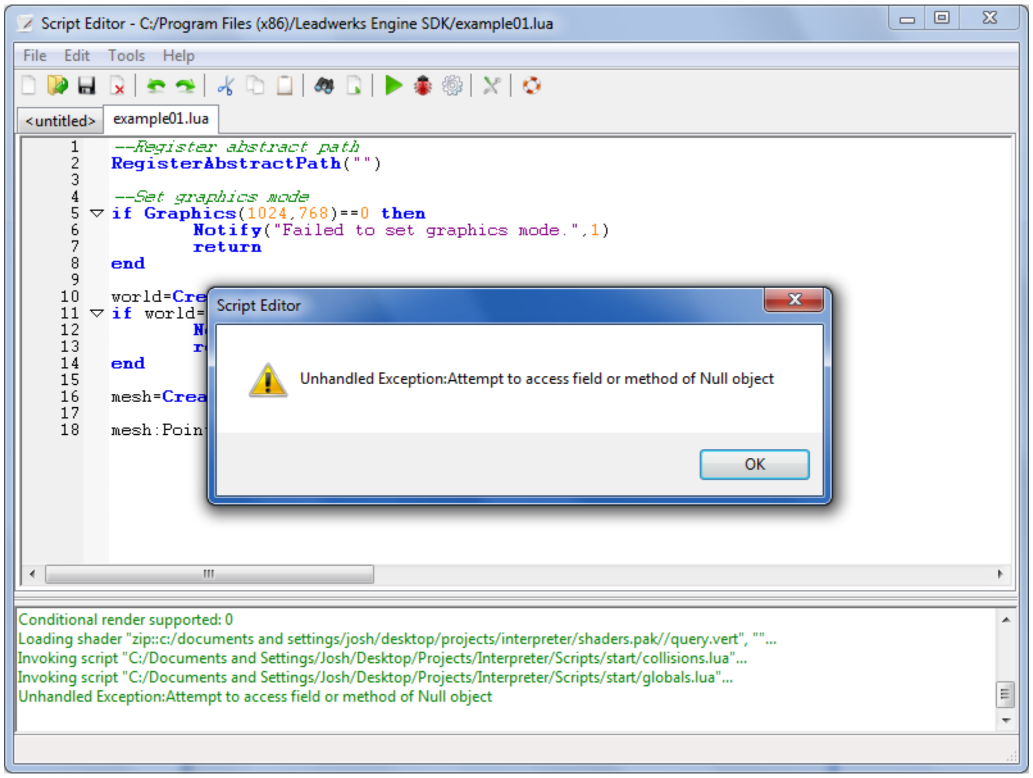
```
--Register abstract path
RegisterAbstractPath("")

--Set graphics mode
if Graphics(1024,768)==0 then
	Notify("Failed to set graphics mode.",1)
	return
end

world=CreateWorld()
if world==nil then
	Notify("Failed to initialize engine.",1)
	return
end

mesh=CreateMesh()

mesh:Point(nil)
```

When we run this script, an error occurs:



```
EXCEPTION_ACCESS_VIOLATION
```

This is not terribly informative. All we know is that something is wrong. We can get more information about the program if we use the debugger. The debugger can be run by pressing F4. When we run the debugger we get an error telling us the problem is a Null object. This makes sense, because we passed nil (the Lua equivalent to Null) in the Point() method:


Script Editor - C:/Program Files (x86)/Leadwerks Engine SDK/example01.lua
File Edit Tools Help
<untitled>
example01.lua
Script Editor
OK

```
--Register abstract path
RegisterAbstractPath("")

--Set graphics mode
if Graphics(1024,768)==0 then
        Notify("Failed to set graphics mode.",1)
        return
end

world=Cre
if world=
        N
        r
end

mesh=Crea

mesh:Poin
```

```
Unhandled Exception:Attempt to access field or method of Null object
```

```
Conditional render supported: 0
Loading shader "zip::c:/documents and settings/josh/desktop/projects/interpreter/shaders.pak//query.vert", ""...
Invoking script "C:/Documents and Settings/Josh/Desktop/Projects/Interpreter/Scripts/start/collisions.lua"...
Invoking script "C:/Documents and Settings/Josh/Desktop/Projects/Interpreter/Scripts/start/globals.lua"...
Unhandled Exception:Attempt to access field or method of Null object
```

Compiling Scripts

Lua has the ability to compile scripts into precompiled byte code files. These precompiled files will load faster and can be used to protect your source code. Precompiled Lua files do not run any faster than uncompiled Lua files. To compile a script, press the F6 key. You will be prompted to save the script as a file if you have not already. The script editor will then run the luac.exe utility. This is the Lua script compiler. The Lua script compiler will load the Lua script file and save a compiled Lua file with the same name, using the extension ".luac". The precompiled .luac file can then be run with the script interpreter.

Running a Game

The script interpreter can be used to run a game without the script editor. By default, the interpreter will load the Lua file "start.lua" if it exists. If name of the script file to be run can also be passed to the interpreter in the command line. If the command line the interpreter is launched with consists of only one parameter, the interpreter will assume that is the script file to load. This allows the user to simply drag a Lua file onto the interpreter to launch the script. The script file to load can also be indicated using the +script switch, where +script is followed by the name of the Lua file to run. The diagram below illustrates how the script to run can be passed to the interpreter in the command line. As you can see, we can run either

Command line	Script that is run
engine.exe	start.luac or start.lua (depending on which is available)
engine.exe "example01.lua"	example01.lua
engine.exe +script "example01.lua"	example01.lua
engine.exe +script "example01.luac"	example01.luac

Script Hooks

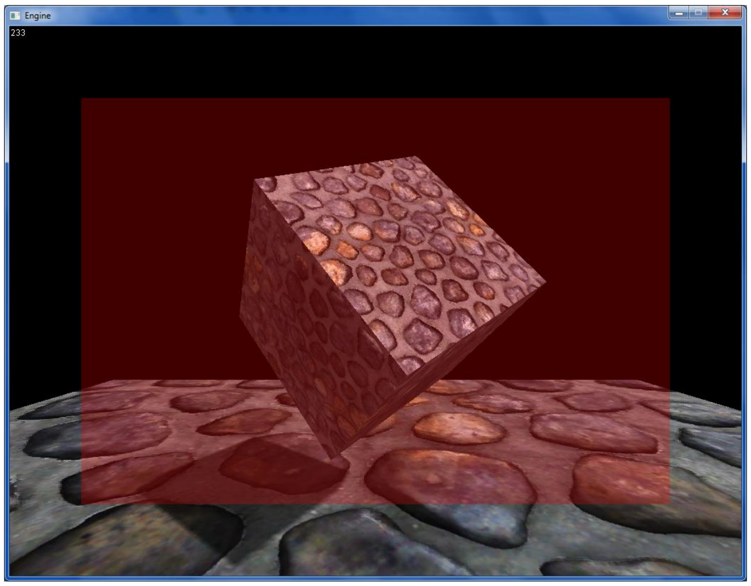
There are a several predefined hooks you can declare in a script. The engine will call these functions at various points in the program.

Function Name	Arguments	Description
FlipHook	<none>	Called prior to Flip() function.
UpdateWorldHook	world	Called each time a world is updated.
UpdatePhysicsHook	world	Called once for each physics step.

This code will use the Flip hook to draw a 2D overlay on the scene, without having to keep the drawing code in the main loop:

```
require("scripts/constants/keycodes")

--Register abstract path
RegisterAbstractPath("")

--Set graphics mode
if Graphics(1024,768)==0 then
	Notify("Failed to set graphics mode.",1)
	return
```

```
end

world=CreateWorld()
if world==nil then
       Notify("Failed to initialize engine.",1)
       return
end

gbuffer=CreateBuffer(GraphicsWidth(),GraphicsHeight(),1+2+4+8)

camera=CreateCamera()
camera:SetPosition(Vec3(0,0,-2))

light=CreateSpotLight(10)
light:SetRotation(Vec3(45,55,0))
light:SetPosition(Vec3(5,5,-5))

material=LoadMaterial("abstract::cobblestones.mat")

mesh=CreateCube()
mesh:Paint(material)

ground=CreateCube()
ground:SetScale(Vec3(10.0,1.0,10.0))
ground:SetPosition(Vec3(0.0,-2.0,0.0))
ground:Paint(material)

light=CreateDirectionalLight()
light:SetRotation(Vec3(45,45,45))

function FlipHook()
       SetBlend(1)
       SetColor(Vec4(1,0,0,0.25))
       DrawRect(100,100,GraphicsWidth()-200,GraphicsHeight()-200)
       SetColor(Vec4(1,1,1,1))
       SetBlend(0)
end

while AppTerminate()==0 do

       if KeyHit(KEY_ESCAPE)==1 then break end

       mesh:Turn(Vec3(AppSpeed()*0.5,AppSpeed()*0.5,AppSpeed()*0.5))

       UpdateAppTime()
       world:Update(AppSpeed())

       SetBuffer(gbuffer)
       world:Render()
       SetBuffer(BackBuffer())
       world:RenderLights(gbuffer)

       DrawText(UPS(),0,0)
       Flip(0)
end
```



A more advanced hook system based on Lua code can be used to run multiple script functions. The "Scripts\hooks.lua" file contains code to do this. The following script functions can be used to add, remove, and run script hooks:

> ***AddHook***(*name, function*)
>
> ***RemoveHook***(*name, function*)
>
> ***RunHooks***(*name*)

If we include the hook script our example above can now be made to look like this. Notice the AddHook function is called *after* the DrawOverlay function is declared. If it were called before the function was declared, there would be no function to pass to AddHook:

```
require("scripts/constants/keycodes")
require("scripts/hooks")

--Register abstract path
RegisterAbstractPath("")

--Set graphics mode
if Graphics(1024,768)==0 then
	Notify("Failed to set graphics mode.",1)
	return
end

world=CreateWorld()
if world==nil then
	Notify("Failed to initialize engine.",1)
	return
end
```

```
gbuffer=CreateBuffer(GraphicsWidth(),GraphicsHeight(),1+2+4+8)

camera=CreateCamera()
camera:SetPosition(Vec3(0,0,-2))

light=CreateSpotLight(10)
light:SetRotation(Vec3(45,55,0))
light:SetPosition(Vec3(5,5,-5))

material=LoadMaterial("abstract::cobblestones.mat")

mesh=CreateCube()
mesh:Paint(material)

ground=CreateCube()
ground:SetScale(Vec3(10.0,1.0,10.0))
ground:SetPosition(Vec3(0.0,-2.0,0.0))
ground:Paint(material)

light=CreateDirectionalLight()
light:SetRotation(Vec3(45,45,45))

function DrawOverlay()
	SetBlend(1)
	SetColor(Vec4(1,0,0,0.25))
	DrawRect(100,100,GraphicsWidth()-200,GraphicsHeight()-200)
	SetColor(Vec4(1,1,1,1))
	SetBlend(0)
end

AddHook("Flip",DrawOverlay)

while AppTerminate()==0 do

	if KeyHit(KEY_ESCAPE)==1 then break end

	mesh:Turn(Vec3(AppSpeed()*0.5,AppSpeed()*0.5,AppSpeed()*0.5))

	UpdateAppTime()
	world:Update(AppSpeed())

	SetBuffer(gbuffer)
	world:Render()
	SetBuffer(BackBuffer())
	world:RenderLights(gbuffer)

	DrawText(UPS(),0,0)
	Flip(0)
end
```

The Lua hook system can be used to add and remove bits of functionality to the program, without having to alter the main program loop. For example, a GUI system can be added that uses the Flip hook to draw, but does not require any code changes in the main program loop.

Class Scripts

A model can have a .lua script associated with it. Pre-defined functions are called at various parts of the program for both the model reference, and for individual model instances. No individual functions are required. If a function is not present in the script, it will be skipped. Class scripts will work with a Lua program, or with C/C++ and other languages.

Classes and Objects

In the class scripts, an "object" is a Lua table associated with an engine model. A class is the table associated with an engine ModelReference object. This is a Lua table that is used to create all the objects. The class has a reference to the model reference, and has an instances table where all objects of the class are inserted into. So on the engine side you have a ModelReference with an instances list where all the models of that type are stored. On the Lua side you have the class table, which has an instances table (like a list) where all the objects of that class are stored.

Lua	Engine
class	ModelReference
object	Model

In Leadwerks Engine, the ModelReference contains a list where all its instances (Models) are stored. If you have a modelreference, you can iterate through its instances (Models) as follows:

```
for model in iterate(modelreference.instances) do

end
```

In Lua, the class contains a table where all its instances are stored. If you have the class you can iterate through its instances (objects) as follows:

```
for model,object in pairs( class.instances ) do

end
```

A class table contains a handle to the ModelReference, and an object contains a handle to the Model. If we have a ModelReference we can retrieve the class as follows:

```
class = classtable[ modelreference ]
```

If we have a Model we can retrieve the object as follows:

```
object = objecttable[ model ]
```

We can also retrieve a class by name, if at least one instance of it exists in the scene:

```
class = classnametable[ "light_directional" ]
```

Let's try a few exercises to make sure we understand these relationships. Each problem will state the information we have, and what we want to get or do.

Exercise 1

We have a model, and want to get the class associated with its reference:

```
class = classtable[ model.reference ]
```

Alternatively we can first get the object associated with the model, then retrieve the class:

```
object = objecttable[ model ]
if object~=nil then
       class = object.class
end
```

Exercise 2

We have an object and want to iterate through all objects of that class.

```
for model,object in pairs(object.class.instances) do

end
```

The following is a somewhat backwards approach, but it still works:

```
for model in iterate(object.model.reference.instances) do
       object = objecttable[ model ]

end
```

Exercise 3

We want to iterate through all directional lights in the scene.

```
class = classnametable[ "light_directional" ]
if class~=nil then
      for model,object in pairs(class.instances) do

      end
end
```

Here's another way. It's probably not as efficient, but it still works:

```
class = classnametable[ "light_directional" ]
if class~=nil then
      for model in iterate(class.modelreference.instances) do
            object = objecttable[ model ]

      end
end
```

Note that if no directional lights exist in the scene, classnametable["light_directional"] will return nil. For this reason we add an if statement to make sure the class even exists.

Class Functions

class:CreateObject(model)

This is called when a new instance of a model is created. You can add your own custom initialization code here. For example, you might add headlights to a car.

class:InitDialog(propertygrid)

This initializes the properties editor for the class in Leadwerks Editor. This function is only called by the Editor, and uses GUI functions not supported by the core engine.

class:Free()

This will be called when all instances of a class are freed. The class is no longer needed, so it will be deleted with this function. Use this to clean up any per-class objects you have created.

Object Functions

object:SetKey(key, value)

This can be used to make entity keys control entity settings and appearance. For example, you can use this to allow control of the entity color in the editor. If the command returns 1, the key will be added to the entity's properties, otherwise it will be discarded.

object:GetKey(key, value)

This can be used to return entity data as a key. For example, you can retrieve the entity's color and return it as a key to allow control of the entity color in the editor. The value passed to the function is the value that was stored in the key table, if it exists. You can use this value by returning it from the function, or override it by returning a different string value.

object:Collision(body, position, normal, force, speed)

This function is called whenever this entity hits or is hit by another physics body. The position, normal, and force are Vec3 objects, and speed is a float value.

object:UpdateMatrix()

This is called whenever the model matrix changes, whether it is moved by programming, physics, or animation.

object:ReceiveMessage(message, extra)

This will be called any time a message is received by the model. This can be used to make objects communicate information to one another.

object:Update()

If present, this will be called once each time the world the model resides in is updated.

object:UpdatePhysics()

If present, this will be called once for each physics step of the world the model resides in.

object:Render(camera)

This is called before a model is rendered. It is not called if the model is offscreen.

object:SetTarget(target, index)

This function is called whenever an entity target is set.

object:Reset()

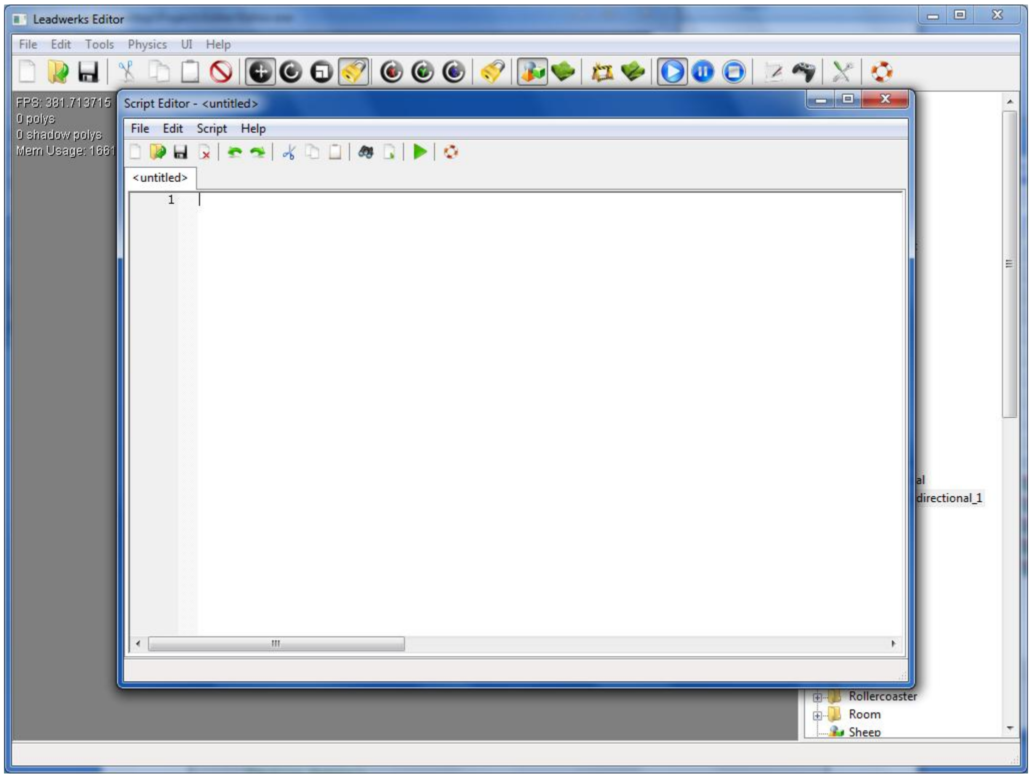
This function is called whenever a model's reset matrix is reset. This can happen if physics are reset in the editor, or if a model is manually moved in the editor. Models that use internal joints can be updated with this function.

object:Free()

This is called when the model is freed. Use it to clean up any extra objects you may have created. Make sure you remember to call the super:Free() function, which will perform required cleanup tasks.

The Integrated Script Editor

Leadwerks Editor features an integrated script editor that lets us write and run scripts in real-time. Start Leadwerks Editor and select the Tools>Script Editor menu item to open the integrated script editor:



The integrated script editor.

We can write and run scripts just like in the standalone script editor. What we do in one script affects the engine Lua state, and can affect other scripts. To demonstrate how the Lua state works, let's perform a simple test. Type the following text in the integrated script editor:

```
a="Hello!"
```

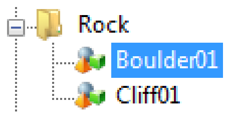
Press F5 to run the script. Now delete the previous code and type this:

```
Notify(a)
```

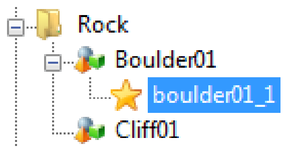
Press the F5 key again to run the script. You can see the "a" variable is accessible. The Lua state won't be reset until a new scene is opened, or a game script is run.

Custom Classes

Now we're going to learn how to make our own custom scripted classes. If the integrated script editor is still open, close it. Find the *class node* for the Rock>Boulder01 class. The class node has an icon with three colored shapes. This represents a class of objects you can create:



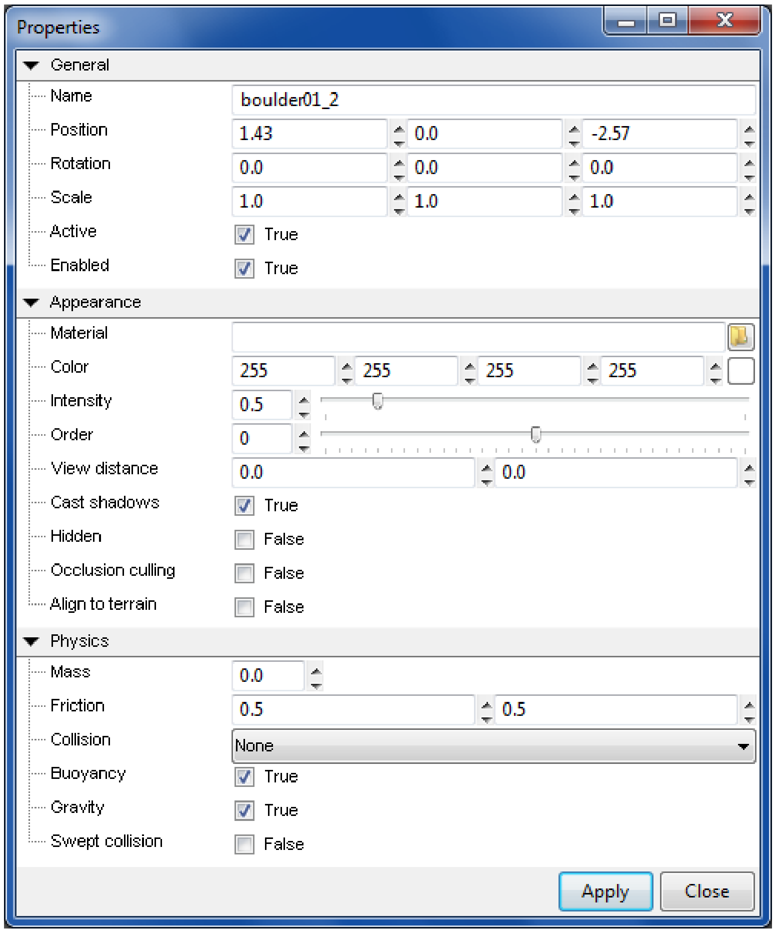
Click the class node and drag it into the scene to create an instance of the class. Below the class node, a new item will appear with a star next to it. This is an *object node*, and represents an individual instance of the class that has been created.



When you double-click on a class node, the class script is opened in the integrated script editor. The class script for the Rock>Boulder01 class looks like this:

```
require("scripts/class")

local class=CreateClass(...)
```

This is all we need for basic functionality and properties. If you click on the object node for the Rock>Boulder01 instance we created, the properties editor is opened, and the editable properties for this object are shown:

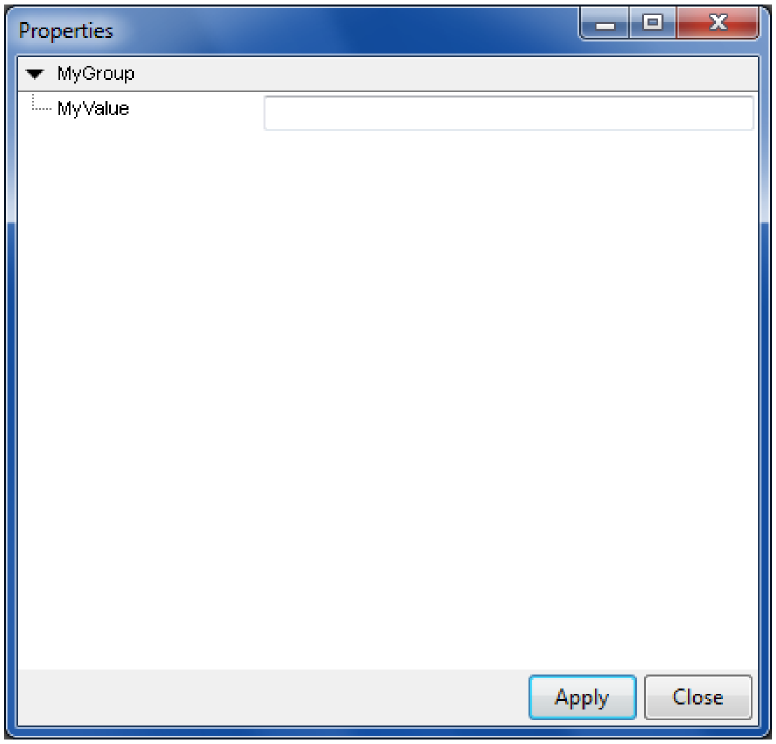


Name	boulder01_2			
Position	1.43	0.0	-2.57	
Rotation	0.0	0.0	0.0	
Scale	1.0	1.0	1.0	
Active	✓ True			
Enabled	✓ True			
Material				
Color	255	255	255	255
Intensity	0.5			
Order	0			
View distance	0.0	0.0		
Cast shadows	✓ True			
Hidden	False			
Occlusion culling	False			
Align to terrain	False			
Mass	0.0			
Friction	0.5	0.5		
Collision	None			
Buoyancy	✓ True			
Gravity	✓ True			
Swept collision	False			

These properties are defined in the "Scripts\class.lua" file. We can define our own properties by overwriting the class:InitDialog function with our own. Copy this text to the Rock>Boulder01 class script and save it.

```
require("Scripts/class")

local class=CreateClass(...)

function class:InitDialog(propertygrid)
        local group=propertygrid:AddGroup("MyGroup")
        group:AddProperty("MyValue",PROPERTY_STRING)
end
```

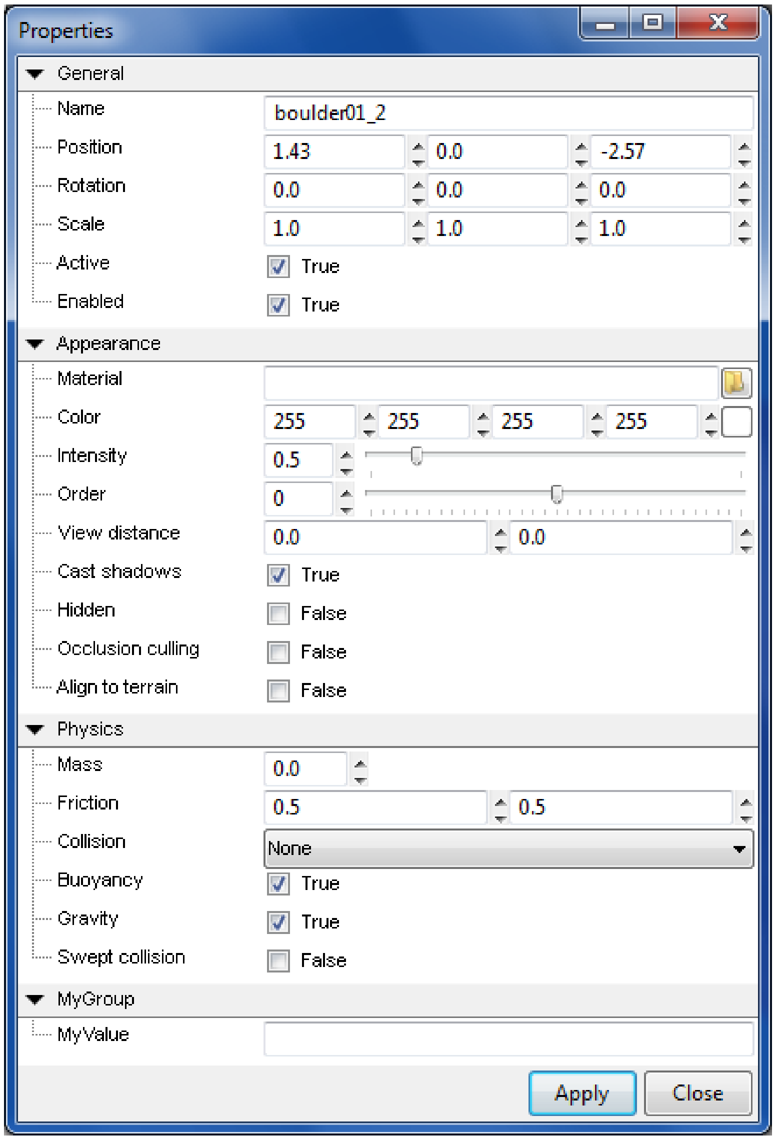
The properties dialog will change to this:



Most of the time we want to add our own custom properties in addition to the basic properties all classes use. We can use both the basic properties and our own if we simply call the class.super:InitDialog function prior to adding custom properties:

```
require("Scripts/class")

local class=CreateClass(...)

function class:InitDialog(propertygrid)
       self.super:InitDialog(propertygrid)
       local group=propertygrid:AddGroup("MyGroup")
       group:AddProperty("MyValue",PROPERTY_STRING)
end
```

Here is the result:


Properties

General

Name	boulder01_2
Position	1.43 / 0.0 / -2.57
Rotation	0.0 / 0.0 / 0.0
Scale	1.0 / 1.0 / 1.0
Active	✓ True
Enabled	✓ True

Appearance

Material	
Color	255 / 255 / 255 / 255
Intensity	0.5
Order	0
View distance	0.0 / 0.0
Cast shadows	✓ True
Hidden	False
Occlusion culling	False
Align to terrain	False

Physics

Mass	0.0
Friction	0.5 / 0.5
Collision	None
Buoyancy	✓ True
Gravity	✓ True
Swept collision	False

MyGroup

MyValue	

Let's add some behavior. To make the model turn each frame, first we add a class:CreateObject function to override the default class:CreateObject function in class.lua. Then we add an object:Update function to make the model turn each frame:

```
require("scripts/class")

local class=CreateClass(...)

function class:CreateObject(model)
        local object=self.super:CreateObject(model)

        function object:Update()
                self.model:Turn(Vec3(0,1,0))
        end
```

```
end
```

Save the script, and your rock will start spinning on its own!

Let's say we wanted to make a class that for some reason creates a cube that is associated with the object. This might not be a realistic scenario, but it is useful for demonstrating object cleanup. Notice that object:Free function first checks to see if the cube is non-nil (always a good idea) and then frees it. The important point is the object:Free function then calls self.super:Free, which is the base object:Free function. Without the call to self.super:Free, the object won't get removed from the object table, and the class might never get freed from memory, even when all instances of it are freed.

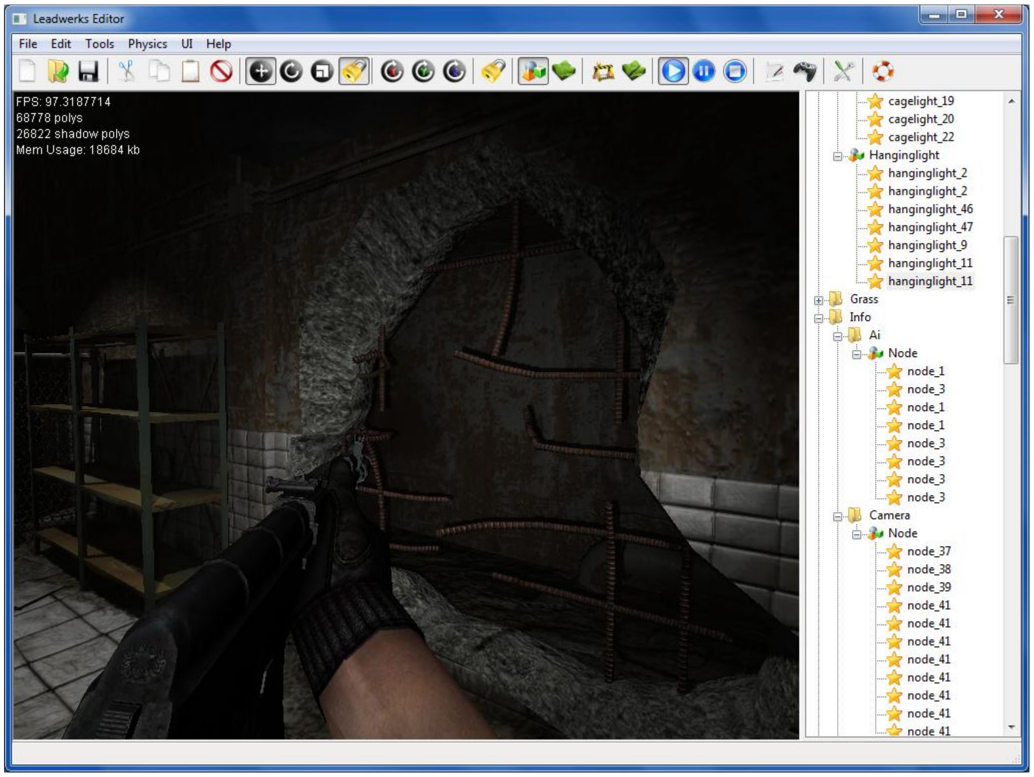
```
require("scripts/class")

local class=CreateClass(...)

function class:CreateObject(model)
	local object=self.super:CreateObject(model)

	object.cube=CreateCube()
	object.cube:SetScale(Vec3(4,4,4))

	function object:Free()
		if self.cube~=nil then
			self.cube:Free()
			self.cube=nil
		end
		self.super:Free()--Don't forget this!!
	end

end
```

Class scripts allow some very advanced functionality. You can add lights, roads, and even AI using class scripts. Browse around the different class scripts included in the engine and see how they work.

The template.lua file contains all the predefined class and object methods. You can copy this text to a new class script and uncomment the functions you want to use.

Game Scripts

You can switch to game mode in the editor by pressing Ctrl+G. When the editor switches to game mode, a temporary copy of the scene is saved in memory. The game script is loaded and run. The game script is defined in the Options>Paths dialog. After the game script returns to the main program, the scene is cleared, the Lua state is reset, and the scene is restored from the copy saved in memory. This allows you to run a fully operational game in the editor without having to clean up after it is done.



Playing a game in Leadwerks Editor.

When you open or write a script in the integrated script editor, the Lua state is not recreated after the script runs. This allows you to modify internal values or set variables, but the editor will not clean up any damage you do. Conversely, when the "Switch to Game" mode is enabled, the editor reloads the scene from memory and creates a clean new Lua state when the you switch back to the editor.

Remember, after a game script is run, the Lua state is cleaned up. When you run a script from the integrated script editor, any changes you make are left in the Lua state.

Startup Scripts

Whenever the Lua state is reset (after creation of the first world or when a new scene is loaded in the editor) all scripts in the "Scripts\start" folder are run, in no particular order. You can use startup scripts to define collision rules or declare global values.

C++ and Lua

The C++ programming language can interface seamlessly with Lua. Class scripts and startup scripts work with any program, including a C/C++ project.

C/C++ can also use the Lua command set to perform advanced actions. One common need is to set a global variable in Lua to an object the user creates. Many of the class scripts rely on the "fw" variable being set to a framework object the application creates. Fortunately, the Lua command set allows us to easily set this value directly:

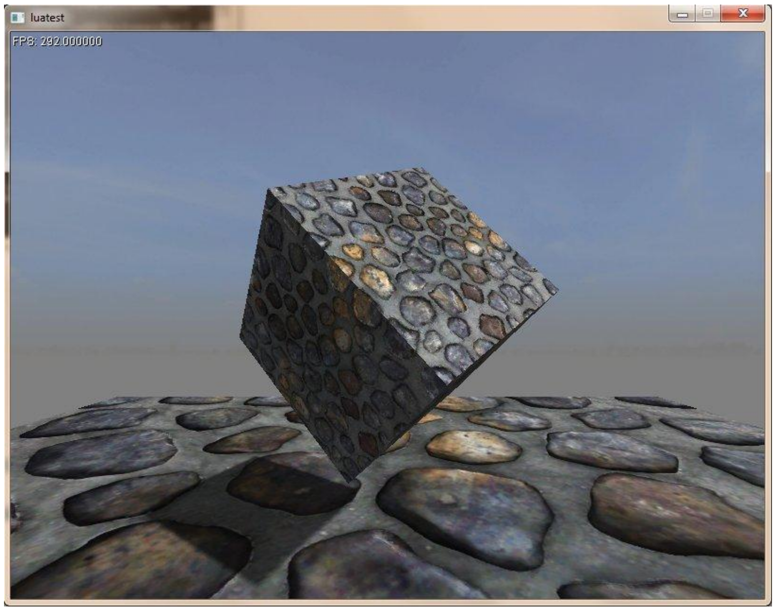
```
TFramework framework=CreateFramework()
unsigned char L=GetLuaState();
lua_pushobject(L,framework);
lua_setglobal(L,"fw");
lua_pop(L,1);
```

Lua scripts can then use the "fw" variable to access the framework commands.

This program will create a framework object load the environment_atmosphere model. The class script for this model will use the framework object to enable a skybox:

```
#include "engine.h"

int main( int argn, char* argv[] )
{
	Initialize() ;
	RegisterAbstractPath("C:/Leadwerks Engine SDK");
	SetAppTitle( "luatest" ) ;
	Graphics( 800, 600 ) ;
	AFilter() ;
	TFilter() ;

	TWorld world;
	TBuffer gbuffer;
	TCamera camera;
	TMesh mesh;
	TLight	light;
	TMesh ground;
	TMaterial material;

	world = CreateWorld() ;
	if (!world) {
		MessageBoxA(0,"Error","Failed to create world.",0);
		return Terminate();
	}

	TFramework framework=CreateFramework();
	TLayer layer = GetFrameworkLayer(0);
	camera=GetLayerCamera(layer);
	PositionEntity(camera,Vec3(0,0,-2));
```

```
        //Set Lua variable
        BP L=GetLuaState();
        lua_pushobject(L,framework);
        lua_setglobal(L,"fw");
        lua_pop(L,1);

        LoadModel("abstract::environment_atmosphere.gmf");

        material=LoadMaterial("abstract::cobblestones.mat");

        mesh=CreateCube();
        PaintEntity(mesh,material);

        ground=CreateCube();
        ScaleEntity(ground,Vec3(10,1,10));
        PositionEntity(ground,Vec3(0,-2,0));
        PaintEntity(ground,material);

        light=CreateDirectionalLight();
        RotateEntity(light,Vec3(45,45,45));

        // Game loop
        while( !KeyHit() && !AppTerminate() )
        {
              if( !AppSuspended() ) // We are not in focus!
              {
                    // Rotate cube
                    TurnEntity( mesh, Vec3( 0.5f*AppSpeed() ) );

                    // Update timing and world
                    UpdateFramework();

                    // Render
                    RenderFramework();

                    // Send to screen
                    Flip(0) ;
              }
        }

        // Done
        return Terminate() ;
}
```



Conclusion

We have learned how to use the script editor to run, debug, and compile scripts. We learned how the interpreter can be used to run a standalone game. We also learned how class scripts can be used to make powerful interactive objects. Finally, we learned a little about the Lua command set, and how a C/C++ program can interact with Lua. This is the basis for writing games with script in Leadwerks Engine.