Basic Irrlicht And Python Integration

The Beginning

Thanks for downloading this tutorial. Today, I'm going to show you how to integrate the Irrlicht Game Engine, which can be obtained at http://irrlicht.sourceforge.net with the Python Scripting Language, which can be obtained at http://www.python.org . I'll be using Python 2.5.4 and version 1.5 of the Irrlicht SDK. I'm assuming you have basic familiarity with C++, the Irrlicht Engine and the Python Scripting Language. Basically, I'm here to show you how to put 'em all together.

Some Theory For You All

Essentially, there are two types of programming languages, compiled languages and interpreted languages. Compiled languages are languages include Assembler, C, C++ while interpreted languages include Java, Python, Ruby, Lua, etc…

Compiled languages are generally faster than interpreted languages. This is because applications written in compiled languages are converted to binary, which the computer can understand directly, reducing the amount of overhead required to perform certain actions. Interpreted languages on the other hand, are usually slower than compiled languages. This is because they are usually converted to a form of intermediate code, which is processed by a program called an interpreter into a form that the computer can understand. The thing is, with the interpreted language, it has to be converted to machine readable form every time it is run, while with a compiled language the program only need to be converted to machine code once, and can be run without any further translation being necessary.

Why Do I Need To Embed A Scripting Language In My Game?

Well, because everybody's doing it, of course ☺. But seriously, everybody IS doing it. It has a number of advantages.

Remember how I just said that interpreted code has to be translated to machine code EVERY time it is run, while a compiled language has to be translated only once? Here's the thing, you can embed a scripting language into your application to provide additional functionality to the end user, without having to expose the source code (think about Quake and QuakeC) or to speed up the development process(getting over the need to recompile the whole program every time you make a minor change).

You see, you can essentially turn your program into an interpreter, providing an environment for the scripting language to run in. Your scripts can then be stored in files separate from your main executable, and loaded into the program every time it is run (as opposed to compiled alongside the main executable). This saves time, especially with larger projects (ever had to wait 2 hours for a program to compile? I have, its quite annoying).

There are lots of other scripting languages I could have used. I could have used Lua, Ruby, GameMonkey etc. But eventually, I settled on Python. First, it's free (although practically all of them are). Next I already had it installed (programmers are inherently lazy ☺). Third, it's relatively well-documented, although it's sort of difficult to understand the C++ code required to embed the Python interpreter inside a C++ program. Fourth, Python itself is a pretty powerful language, supporting lots of OOP and functional programming techniques.

Starting To Code

First of all, you need to link the libraries Irrlicht.lib and python25.lib. There's a quick catch as it pertains to linking Python however. If you're using MSVC and you're running a debug build, its going to tell you that it can't find python25_d.lib. To save yourself the trouble of having to mess with the project configuration settings, just go to the libs subdirectory of where you installed Python, and make a copy of python25.lib called python25_d.lib. Oh yeah, also, remember to set the appropriate include and library directories for your compiler. Obviously, you're going to need Irrlicht.h and Python.h. Some other libraries which I ended up needing were string.h (yes string.h, not cstring or just string…you get a lot of screwed up namespace errors if you include those, since both Irrlicht and the STL have string classes, If you look in my source code, you'll see a section commented with information about the problem I had with that).

Line By Line Analysis

I'm going to analyze the source code line by line in this section, so here goes:

```
#include<Irrlicht.h>
#include<Python.h>
#include<stdio.h>
#include<vector>
#include<string.h>
#include<stdlib.h>

using namespace std;

using namespace irr;
using namespace core;
using namespace video;
using namespace gui;
using namespace scene;
using namespace io;
```

If you need an explanation…seriously…psssh

```
IrrlichtDevice * device;
IVideoDriver * driver;
ISceneManager * smgr;
IGUIEnvironment * guienv;
IGUIFont * default_font;
vector<ITexture *> texture_array; //Our array of textures
```

Some global variables (please, don't attack me on my use of these, I'm trying to keep it simple here).

```
static PyObject * PyIrr_LoadTexture(PyObject * self,PyObject * args)
{
	//Watch this, tricky,remember to pass string ADDRESS to
//PyArg_ParseTuple
	char * tex_name;
	PyArg_ParseTuple(args,"s",&tex_name);
	texture_array.push_back(driver->getTexture(tex_name));
	/*The line below is sorta kludgy, but it works.It won't hold up if you
remove a texture
	from the array though,so watch your step, kid. I'll leave it to you to
come up with
	a more intuitive method of storing the textures*/
	return Py_BuildValue("l",texture_array.size() - 1);
};
```

Here's our first function that will be used in our script. This function is responsible for loading textures. Rather than having to wrap the whole ITexture and IVideoDriver class into Python, I decided to take the easy way out, and assign an integer identifier to every texture loaded. The integer would represent the texture's index in the texture_array vector.

Here goes the line by line of the function. First, the function prototype. Take a good look at it:

```
static PyObject * PyIrr_LoadTexture(PyObject * self,PyObject * args)
```

Every function in C++ that you want to link to Python has to have a return type of `static PyObject *` to enable the Python interpreter to understand it. Just take that as a given.In fact, the general format of the function prototype I gave above is the same for every C++ function that you want to use to extend the Python language.In other words:

```
static PyObject * [function_name](PyObject * self,PyObject * args)
```

Note that [function_name] is the name the function will go by in the C++ program, not the Python program. To give the function the name that it will go by Python script will be done later. Note that every function of this type takes two parameters, no matter how many parameters the function takes in script. Lets not rush now, kids ☺.

```
char * tex_name;
PyArg_ParseTuple(args,"s",&tex_name);
```

These two lines read the data from the parameters passed to the script function. In our Python script, we would call the function using:

tex_id = load_texture(“texture.PNG”)

As you see, the function in Python takes only one parameter, a string. So why does our C++ definition for the function always take 2 parameters? Well, the explanation of the first parameter is a bit complex(it has something to do with classes, I think), but the second one actually contains the data we want. No matter how many parameters the function has in the Python script, the parameters are put into a tuple(a Python datatype which basically acts as a container) and passed back to the C++ program in the “args” variable.

So how do we extract the data from this tuple?

Look at the function called PyArg_ParseTuple. In this case, it has three parameters. The first parameter is the tuple datatype in which the function parameters are passed. The second is the *format string* of the data in the tuple. This specifies how the data in the tuple is extracted. If you’ve ever used standard C++ functions such as scanf and sscanf and printf, this should seem familiar. The third parameter(actually there can be more than one third parameter, so to speak) is/are the addresses of the variables where the extracted data should be placed. In this case, the only parameter our function takes is a string. A quick check in the python documentation says the format specification for strings is “s”. Therefore, we extract that data and put it in the string called tex_name. You don’t need to worry about allocating the memory for the string, Python has already taken care of that for you (after all, how would you know how big the string is?). Just provide the variable for the string data to be put into. You can check the Python format specifications in the Python documentation. If this section was a bit unclear to you, check out the MSDN documentation for the sscanf function, which has a similar format.

```
texture_array.push_back(driver->getTexture(tex_name));
```

That line loads the texture specified by the string and puts it into our array of textures, so that it can be used later. We don’t have to worry about the Irrlicht engine reference counting here, its okay as is.

```
return Py_BuildValue("l",texture_array.size() - 1);
```

Don’t panic. That line is pretty straightforward. The first parameter is the format specification. In this case, we want to return an ID number. This would basically be the same as a long datatype in C++, or a s32 according to the Irrlicht specifications. The Python format specification for this is “l”. This function essentially creates the object containing the return value of the function, in this case, the texture ID.

So, that's the first function we've dissected. Not that bad, was it?

Here's the next one:

```
static PyObject * PyIrr_SetTexture(PyObject * self,PyObject * args)
{
	s32 tex_id,node_id;
	PyArg_ParseTuple(args,"ll",&node_id,&tex_id); //This is your new best
//friend, seriously
	/*Quite similar to the scanf family of functions, don't you think? It
take a format
	string and some input data, and analyzes the input data and gives you
the result
	in a manner specified by the format string*/
	ISceneNode * node = smgr->getSceneNodeFromId(node_id);
	if(node != NULL)
	{
		node->setMaterialTexture(0,texture_array[tex_id]);
	};
	/*This line returns a value of Py_None, which is more or less the same
thing as a
	function with a return type of void in C++, in other words, no output
values */
	return Py_BuildValue("");
};
```

Pretty straightforward. First, we declare two s32 variables called tex_id and node_id. Then, we parse the tuple to extract the data passed to the function in the script. As you can see, our format string is "ll" or two integers. The function would be called in script by:

set_texture(12,2)

In other words, to extract more than one parameter from a function, we simply just put together all the letters representing the format specifications of the input parameters in the section for the format string, in the order they appear. So if a function takes 3 floats as parameters, the format string is "fff". If it takes two long integers, two floats and a string, the format string is "llffs". You should get the idea by now.

```
ISceneNode * node = smgr->getSceneNodeFromId(node_id);
	if(node != NULL)
	{
		node->setMaterialTexture(0,texture_array[tex_id]);
	};
```

Those lines above simply get the scene node specified by the node ID passed to the function and attaches the texture specified by the texture ID. There's some rudimentary error checking there, but the application in general could do with a LOT more error checking. I'll leave that up to you.

```
return Py_BuildValue("");
```

“Whoa,” you say, “What the hell does THAT mean? There’s no format specification, and no data to put into the object we create.

To explain this, lets think in C++ terms. When a function has a return type of void, that means it doesn’t return anything, right? That’s basically the same thing for the Python value of Py_None, almost the same as the C++ NULL. In other words, the function has no return type because it doesn’t return anything. I hope that makes sense.

```
static PyObject * PyIrr_AddCubeSceneNode(PyObject * self,PyObject * args)
{
	s32 node_id;
	float size;
	float px,py,pz,rx,ry,rz,sx,sy,sz;
	PyArg_ParseTuple(args,"lffffffffff",&node_id,&size,&px,&py,&pz,&rx,&ry,
&rz,&sx,&sy,&sz);
	ISceneNode * node = smgr->getSceneNodeFromId(node_id);
	if(node == NULL)
	{
		node = smgr-
>addCubeSceneNode(size,NULL,node_id,vector3df(px,py,pz),vector3df(rx,ry,rz),v
ector3df(sx,sy,sz));
		node->setMaterialFlag(EMF_LIGHTING,false);
	}
	else
	{
		return Py_BuildValue("");
	};
	return Py_BuildValue("l",node_id);
};
```

Most of this should be straightforward by now, so I’ll just skip to PyArg_ParseTuple.

```
PyArg_ParseTuple(args,"lffffffffff",&node_id,&size,&px,&py,&pz,&rx,&ry,&rz,&s
x,&sy,&sz);
```

Check THAT format string out.If you count it up, that means the function takes 11 parameters. That’s one long integer and 10 floating point variables. The long integer is the ID of the cube scene node. The first floating point number is the size of the cube. The next three are the X,Y and Z components of the position vector. The three after that are the X,Y and Z rotation vector. The last three are the X,Y and Z of the scaling vector.

Next, we test if the scene node with the ID we requested exists already.

```
ISceneNode * node = smgr->getSceneNodeFromId(node_id);
```

This should be NULL if the node doesn’t exist(which is what we want). If it is NULL, we create the node and make the function return the ID of the newly created node. We also turn off the lighting, so we can see it(I forgot this initially and couldn’t see anything, so I kept on thinking something was wrong with my script…just some info for you all so you don’t make the same mistake). If the node already exists, we return Py_None or NULL, whatever you want to call it, since we don't want to create two nodes with the same ID. Otherwise, if we just created the node, we return the ID.

I'm not going to bother to go through the PyIrr_DrawText function since it would basically be a rehash of what I covered above. The only thing of note in that function is to see how I got a char * from the parameters passed to the Python function, and converted it to a wchar_t*. It's a pretty common operation in Irrlicht, especially if you're like me and find Unicode annoying and uncomfortable, although useful.

```
static PyMethodDef irr_funcs[] =
{
      {"set_texture",PyIrr_SetTexture,METH_VARARGS,"Adds a texture to a scene
node"},
      {"draw_text",PyIrr_DrawText,METH_VARARGS,"Renders text to the screen
with default font"},
      {"add_cube",PyIrr_AddCubeSceneNode,METH_VARARGS,"Adds a cube scene
node"},
      {"load_texture",PyIrr_LoadTexture,METH_VARARGS,"Loads a texture"},
      {NULL,NULL,0,NULL}
};
```

The PyMethodDef data type is a datastructure that associates the C++ function with the Python function. The first component of a PyMethodDef structure is the name which the function it refers to will go by in the Python script. The second component is the C++ function which will be invoked every time the function is called from a Python script. The third component specifies the argument passing convention for the function parameters. To save yourself from having an unnecessarily high blood pressure, I suggest leaving it at METH_VARARGS, although the adventurous ones among us can check the Python docs for the alternative values. The fourth parameter is a string that contains a description of the function. Check the comments in the source code for a little tidbit about how it comes in handy. Note that the functions are declared as an array of PyMethodDef, and the last member of the array is always `{NULL,NULL,0,NULL}`.

```
PyMODINIT_FUNC init_irr(void)
{
    Py_InitModule("irr",irr_funcs);
};
```

This function is responsible for initializing our module to expose the functions to the scripting language. Note that the return type is `PyMODINIT_FUNC`, which is a macro defined by the Python-C++ API. You need to remember use it to link the function. The Py_InitModule function takes two parameters, the name which the module will go by in the script, and the functions which the module defines in the script.

The next function involves a bit of Python black magic. There are LOTS of functions in the Python-C++ API that allow you to run a python script. However, most of those use the FILE structure, which is problematic, since it is not defined in a standard manner across platforms or compilers. In other words, unless you're using the exact same compiler with the exact same settings that your version of Python was built with, its not going to work. And since its highly unlikely that you are, we have to use this quick fix to get Python scripts to work.

```
void ExecuteScript(irr::core::string<char> scriptname)
{
      irr::core::string<char> result;
      result = irr::core::string<char>("execfile('");
      result.append(scriptname);
      result.append("')");
      PyRun_SimpleString(result.c_str());
};
```

Python has a built in function that allows you to run a Python script from inside a Python script (let’s hear it for interpreted languages baby!!! WHOOO!!!!). The syntax of this function is execfile([filename as string]). The PyRun_SimpleString runs some Python code specified in a string. We do this little hack where we create a string that goes like “execfile([scriptname])”, which makes the Python interpreter run a script containing the instruction to run a script ☺. Hope that wasn’t confusing.

```
init_irr(); //Initialize our module
ExecuteScript("script.pys");
while(device->run())
{
      driver->beginScene(true,true,SColor(255,0,0,0));
          ExecuteScript("script2.pys");
          smgr->drawAll();
            guienv->drawAll();
      driver->endScene();
};
```

This part doesn’t really need any explanation. We have two scripts. The first script, called script.pys, contains the code to create the cube. The second script renders my name at the top left corner of the screen. Note that the second script is executed inside the render loop because my name needs to be redrawn every frame. The first script is executed before it, because we only want to create the cube once. If we put it in the render loop, we would keep on creating cube scene nodes, until your computer runs out of memory and blows up, killing you. And we wouldn’t want that now, would we? ☺

That’s about it for this tutorial. See you next time. Feel free to mess around and redistribute the sources with any modifications you see fit. Just remember to leave my name in the header if you redistribute this PDF document.

Bye!!