Creating a Lua scripted game with openal audio

### Game Content Engineering

By Sion Williams

P059651

Contents

[Game Content Engineering 1](#_Toc191131998)

[Introduction 3](#_Toc191131999)

[Scripting 4](#_Toc191132000)

[Audio 7](#_Toc191132001)

[Bibliography 9](#_Toc191132002)

# Introduction

When we consider building and developing games in today’s day and age, we must consider the game engine. The most common analogy is that a game engine is comparable to a car engine. You can take the engine out of the car, build another shell around it, and use it again. A good example of this can be found in the difference between “Gears of War” and “Fatal Inertia”, although both games use the “Unreal Technology” game engine. One is a driving game and the other is a 3rd person shooter.

The game engine programming model was popularized by iD Software’s game “Doom” (1) and has since become an industry standard. The current leading game engine is the “Unreal technology” engine and has been used in games such as Gears of War, Fatal Inertia, Bioshock and Unreal tournament. A game engine consists of several components.

A few of these features are (2):

* Rendering Features
* Game Coding Features
* AI Features
* Modeling and Animation Features
* Physics Features
* Network Features
* Audio Features
* Miscellaneous Features (Model importing)

In recent years several big-name developers have started to use scripting extensively in their titles. This is the game coding feature shown above. Namely these are Epic Games' Unreal Tournament series (3), BioWare's Neverwinter Nights (4), and Crytek's Far Cry (5) all utilizes scripting languages.

# Scripting

In previous years when games were much simpler, there was generally not much need for scripting. The designers would explain to the programmers what they wanted and the programmers would set about hard coding the new requirements. As games became more complex, the amount of code generated also increased.

Consider the scenario where a role playing game has 100+ items in game. The programmer creates a structure for the items then stores the items in an array. The structure may look similar to:

typedef struct Item

{

char \* name; // What is the item called?

int type; // What general type of item is it?

int price; // How much should it cost in shops?

int power; // How powerful is it?

} Item;

When the game is compiled and tested, it is found that a certain item may be giving too much health. In order to fix this problem we would find the item in the array, change its power and recompile. On the second test, the power of the potion is suitable but it’s found that it costs too much. As before, the programmer must go back to the item, correct the price, recompile and then re-test.

Although the process described above is simple enough, it’s very time consuming and takes the programmer away from more pressing issues. It’s also illogical to recompile the 3D engine and sound routines every time a change is made to the game content. If an artist wants to modify some graphics, the programmer doesn’t have to recompile. The artist just makes the changes and the next time the game is run, these changes are reflected. Same goes for music and sound. The sound technician can rewrite the theme tune as often as desired, and the programmer never has to know about it.



Figure Engine Code and Item Description together (6)

The figure above shows the *engine code* and *item descriptions* as being part of the same source file. The problem with this is you can’t compile one without the other. Graphics, music, and sound, however, exist outside of the source code and are thus far more flexible.

“If you can get your game content into external files, they’ll be just as flexible as graphics and sound because they’ll only be needed at runtime.” (6) The reason for this is because they’re stored in separate files. The games only connection with this data is code that reads it from the disk. They’re loaded at runtime. At compile time they don’t even have to be on the same drive because their unrelated to the source code.



Figure Item Description as external file (6)

The figure above shows the same system, only this time the game content *item descriptions* are handled separately. This makes the game content self contained, just like the sound, music and graphics.

Effectively, what we now have is a section that can be coded in a high-level language similar to C/C++ that can be compiled independently of the game engine but loaded and executed by the engine whenever needed.

A common scripting language is Lua. Lua is a mature scripting language and has been used in many games, for example world of warcraft and sim city 4.

Due to its simplicity Lua doesn’t have variable types, instead, when writing types in Lua; “b” can be of type float, double, Boolean, int or char, all are acceptable. Lua interfaces C/C++ via a virtual stack, whenever calls are made between C and Lua, the item called is place on the stack and then removed on the other side. Lua is often used for storing variables that may need changing frequently. One application of this was with the “config.lua” file in the project. This file held parameters that enabled the graphics library to initialize the size of the rendering window. When in the early stages of the development this was needed to be changed often to achieve the correct ratio.

Another useful feature of Lua is tables; tables can hold a lot of information in a form that slightly resembles that of a structure in C/C++. This very same theory was used to create the players details written in the “player.lua” file. Using nested tables it was possible to store the players name, X and Y positions. From within the program these details were received via the stacks. Using the clever loop below, it was able to go into any depth nested table and sort each component and store them as floats.

while(lua\_next(pLuaPlayer, -2) != 0)

{

if(lua\_isstring(pLuaPlayer, -1)){ tempKey = lua\_tostring(pLuaPlayer, -2);

printf("%s = %s\n", lua\_tostring(pLuaPlayer, -2), lua\_tostring(pLuaPlayer, -1));

if(\_stricmp(tempKey, "Y")==0) {

p.StartPosY=(float) lua\_tonumber(pLuaPlayer,-1);

}

else if(\_stricmp(tempKey,"X")==0)

{

p.StartPosX =(float) lua\_tonumber(pLuaPlayer,-1);

}

}

else if(lua\_isnumber(pLuaPlayer, -1))

printf("%s = %d\n", lua\_tostring(pLuaPlayer, -2), lua\_tonumber(pLuaPlayer, -1));

else if(lua\_istable(pLuaPlayer, -1))

tableTraverse(pLuaPlayer, p);

lua\_pop(pLuaPlayer, 1);

}

With the clever use of a nested table inside a table it’s even possible to create linked lists in Lua. Although Lua can offer the game environment variables to use in game, it can also offer functions and call C/C++ functions itself. This makes creating game content less “programmer” based. “World of Warcraft” has used this to allow users freedom to create their own modifications for the game.

function MageWeaponSwap(...)

arg = {...}

table.insert(arg, 1, 18);

table.insert(arg, 1, 16);

return WeaponQuickSwap\_WeaponSwapCommon(arg);

end

Above is an example from World of Warcraft, the function simply allows the player to swap their weapons but stores the weapon firstly in a table that is initialized at the start.

Similarly, the functions created in “control.lua” updates the players X, Y positions. Due to Lua not having case’s, if else was required to achieve the same result.

function moveY (move,currenty)

if move == 2 then

return currenty + 1

elseif move == 1 then

return currenty - 1

end

end

As mentioned previously, Lua may also call functions that reside within the actual game code. The reason for doing this is to have direct access to the arguments the functions takes. As discussed previously this eliminates the need to re-compile at and after every change.

Below is an example of the above situation. Lua tells the C function that it wants to print the result of the random generator when the maximum value of the range is 16:

print(randNum(16))

The C function is as follows:

int randNumbGen(int d)

{

int random;

srand((unsigned)time(0));

int lowest=1, highest=d;

int range=(highest-lowest)+1;

random = (lowest+int(range\*rand()/(RAND\_MAX + 1.0)));

return random;

}

The way C/C++ handles this is by using a glue function. The aim of the glue function is to take the arguments from the lua stack and push the result, the code is shown below.

static int randNumGlue(lua\_State \*pLuaNumb)

{

int d = (int)lua\_tonumber(pLuaNumb,1); lua\_pushnumber(pLuaNumb, randNumbGen(d));

return 1;

}

Instantly it’s becoming visible just how useful this is to a designer and how safe it has now become for the programmers code.

# Audio

Game audio is frequently overlooked when it comes to designing and programming a good game (7). Often, developers spend so much time concentrating on other areas of the game like the AI and graphics that the audio tends to suffer. Although, in recent years games like Gear of War and Bioshock have raised the bar significantly with things like 3D sound, environmental audio and MP3 playback.

Currently there are only 2 major Audio API’s, “OpenAL” and “DirectX Audio” (7). “OpenAL” is currently the only open source option of the two and was developed by a consortium of people and companies to be a cross platform API for 3D sound. It’s the sister API to “OpenGL” for 3D graphics rendering, and the design and conventions used by OpenAL are synonymous to that of OpenGL.

The OpenAL functionality revolves around three main kinds of objects: sources, buffers, and listeners. A source is an instance of a buffer and the listener represents the ears of the in-game player.

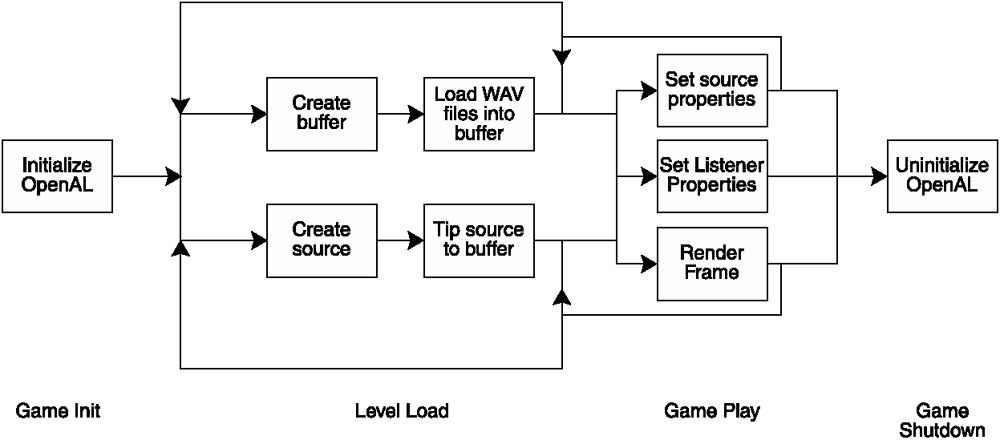


Figure Typical Flow when using OpenAL (7)

OpenAL uses these constructs so that if, for example, the game has three different helicopters, you don’t need to load three copies of the helicopter WAV file into memory. You load the WAV into a buffer, and then create three sources that reference that same buffer.

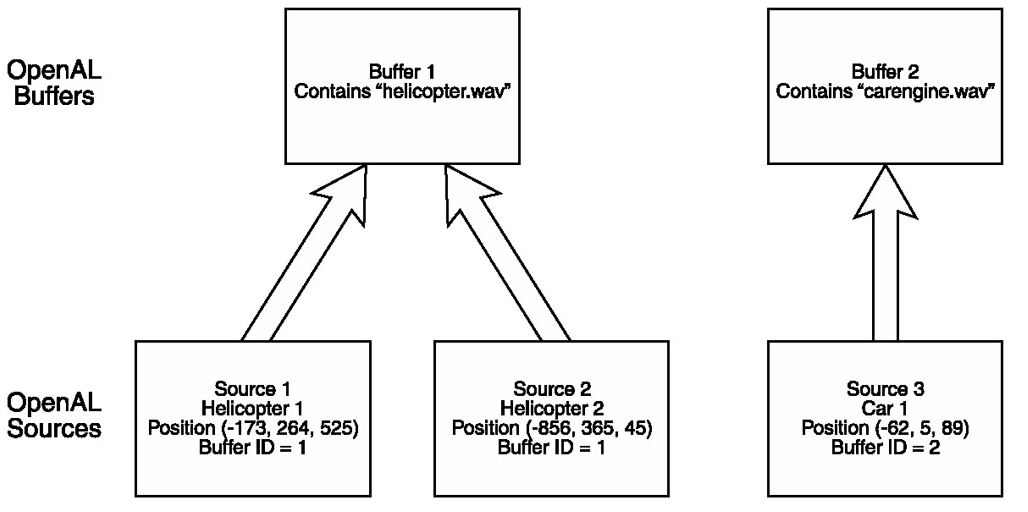


Figure Multiple source example

Another useful function of OpenAL is the support of 3D sound. In short this means that we’re able to move sound in our game world. An example of this taken from the project is that of the player’s footsteps. The playback was implemented in such a way that the source position is relative to the player’s position. This will give the sense of movement but non-visually. Below is the implementation.

SourcePos[3][0]=startX;

SourcePos[3][1]=startY;

alSourcePlay(Sources[3]);

One drawback is that OpenAL doesn’t have control over walls, where in actual fact a wall will absorb sound, OpenAL will play the audio in the same manner both with and without walls. To overcome this, the programmer will be responsible for implementing this functionality.

# Bibliography

1. **Simpson, Jake.** Game Engine Anatomy 101. *ExtremeTech.* [Online] http://www.extremetech.com/article2/0,3973,594,00.asp.

2. **Engine, Lawmaker Game.** Darkroom Studios: Technology. *Darkroom Studios.* [Online] http://www.darkroomstudios.com/content/view/13/64/.

3. **Unreal.** Unreal Technology. *Unreal Technology.* [Online] http://www.unrealtechnology.com/.

4. **Gaider, David.** Neverwinter Nights: Scripting. *Neverwinter Nights.* [Online] http://nwn.bioware.com/builders/sctutorial.html.

5. Crytek Techonology. *Crytek.* [Online] http://www.crytek.com/index.php.

6. **Varanese, Alex.** *Game Scripting Mastery.* s.l. : Premiere Press. 1-931841-57-8.

7. **McCuskey, Mason.** *Beginning Game Audio Programming.* s.l. : Premier Press , 2003. 1592000290.