A Scriptable 2D Game Engine

Cyrus Hanlon

April 27, 2018

Contents

Chapter 1

Introduction

There are many game engines that already exist but nothing combines ease of use with raw speed.

From Unreal Engine [5] with its complete design versus much lighter weight engines such as LÖVE [12], there is a very wide range of technologies and techniques at work. There is a definitive gap between large heavy engines and the small and lightweight that this project will fill. By having many features written into the core engine but keeping a lightweight and easy to use API at the front, it will be possible to quickly learn how to prototype games, while aiming to achieve excellent real world performance.

A very important part of any game engine is the interface between developer and the inner workings. There are many scripting languages that can be chosen from and there are many important factors when selecting a language to use, the most important factors include: popularity, execution speed and learning curve. Lua is at the top of the pile when it comes to games. Other languages such as JavaScript and Python are extremely popular in other areas but are not anywhere near as often used in games as they are more difficult to bind.

Embedded scripting languages need to have access to core elements of the game engine to allow for developers to create fully fledged games without having to touch the lower level. Having the core engine open source also

allows game developers to extend the Lua interface as required and push the changes to the repository. As many features as possible should be included, such as resource management, camera management and physics, all should make the development process simpler for the end developer.

Over the course of this report the design, implementation and technologies involved will be discussed including library choice for the various aspects as required.

1.1 Objectives

Allow interaction with game objects from both C++ and the embedded high level scripting language. This will be extremely important for ensuring that the designed game engine will be highly functional and easy to use.

Have various entry points that should be synonymous with what other game engines have such as standard entry points like initialise, update and draw.

Ensure full expandability. This will be a major factor when it comes to designing and implementing the core engine, which must be as general purpose as possible and should be tweakable for any use case.

Improve performance beyond LOVE2D by using techniques such as moving more of the core processing to the C++ side.

A maintainable codebase is very important for expandability and will ensure a fluid development process through the project and into the future beyond the project.

Document the engine and scripting language API. Implementing a game in a short space of time is very difficult without documentation since developers without any intimate knowledge of the engine will still be required to fully utilise the various features of the engine.

Compete in a game jam to verify how effective the various aspects of the game engine are under high stress rapid development.

Chapter 2

Literature Review

In this chapter the existing technologies surrounding the project's intended market will be described and discussed and technical details necessary for understanding the work will be explained.

2.1 Lua

2.1.1 About Lua

Lua was created for extending applications, there was an increase in demand for customisation and no language was available that combined procedural features with powerful data description facilities. [15] Lua came from the languages SOL and DEL [14] both of which were made by the company Tecgraf.

SOL is an acronym for Simple Object Language and was a specialised data description language used for customising software. The SOL syntax was strongly influenced by the syntax of BiBTeX [9]. DEL is an acronym for Data Entry Language, an entity in DEL is effectively an object or record in other languages [14]. Lua was created as a unique combination of the two more primitive languages, it was designed to avoid unique syntax so that anyone could use it effectively with limited training and for the language to

have built-in type checking.

Lua is written in C which allows it to be very cross platform [11], it is used in projects ranging from Cheat Engine [2], an open source memory scanner and hex editor for Windows, to Android and iOS games using something such as Cocos2d [3], an open source software framework used in the creation of apps or games. Therefore Lua is widely used thanks to its simple learning curve, ease of implementation using the provided minimalistic C API and its fast execution when compared to similar languages. It is also used in many non-game settings such as in embedded systems using eLua [4] thanks to its low memory footprint.

Lua allows for loading modules at run time that were not required at build time, this allows for extremely flexible program extensions. A module is made available through the use of an ordinary table with functions or variables being accessed through the table. Loading, compiling then executing Lua files from within Lua is easily possible and allows for things such as hot reloading code, vastly improving the development rate, as developers could just save a file and the Lua program would simply start using it.

2.1.2 Running Lua

Lua is not interpreted directly but is instead compiled into bytecode, this is then run on the Lua virtual machine. Because of this Lua is a fast language, considering how high level it is. Lua used a stack based virtual machine initially but with the release of Lua 5.0, Lua began to use a register based machine. It also uses a stack for allocating activation records where the registers live [8].

First class functions with lexical scoping is difficult to implement for languages that use a stack based vm, Lua uses a stack for local variables and a heap for when they go out of scope while being referenced to by nested functions [8]. Using this concept Lua successfully implements features such as closures that rely on register based VMs.

Lua can also be used from the command line which allows for productivity scripts to be written and used without the need for it to be embedded into

another application. Lua can also be used as a standalone program using the tool srLua which was created by one of the Lua authors [6], this allows the language to be used on any platform.

2.1.3 Syntax

The syntax of Lua was designed so that it combined simple procedural syntax with powerful data description constructs, the general syntax is very easy to learn with its roots in SOL and DEL, both were designed for ease of use. It borrows much of its syntax from other languages, comment syntax, for example, is similar to that of Haskell and SQL. The control flow is similar to that of pascal using keywords such as `then` and `end` for `if` statements. It is, however, renowned for being very easy to write with its simple syntax and extremely powerful table constructs. A quirk when working with Lua is the fact that tables start at an index of 1 instead of 0 as is very common throughout programming languages.

An example of Lua's control statements are shown in listing 2.1. The use of `elseif` as a single keyword is useful and avoids the need to use an `end` per `if`. The conditions in each of the statements do not need to be wrapped in parenthesis and allow for logical operators.

```
1  local x = 30
2  if x > 25 then
3      print("x is less than 25")
4  elseif x < 35 then
5      print("x is greater than 35")
6  end -- prints out x is less than 25
7
8  while true do
9      print("loop!")
10 end -- this loops forever printing loop!
11
12 repeat
13     print("loop!")
14 until false -- this loops forever printing loop!
15
16 for i=1, 10, 3 do
17     print(i)
```

```
18 end -- prints 1 4 7 10
```

Listing 2.1: Lua Syntax Sample 1

There are two types of `for` loop in Lua, one is a numeric `for` and is shown in listing 2.1, the other is the generic `for` that uses an iterator and allows for the traversal of all values. The numeric `for` uses a very simple syntax and so is very easy to write, the first argument is the starting value of the iteration variable, the second is the end value and the third is the value to add with each loop.

The logical operators in Lua are `and`, `or` and `not`. Logical operators and control statement expressions take `nil` and `false` as `false` and anything else as `true` [10]. This allows for some unique syntax that more advanced Lua programmers will use to reduce the amount of code required.

Each of the three logical operators behave differently when used in this way. The `and` operator returns the first argument if it is evaluated to `false` and otherwise returns the second argument. The `or` operator returns the first argument if this value isn't `false` and returns the second argument otherwise. The `not` operator always returns `true` or `false` by inverting the boolean. The Lua 5.3 manual [10] has an excellent set of examples as shown in listing 2.2.

```
1 10 or 20 --> 10
2 10 or error() --> 10
3 nil or "a" --> "a"
4 nil and 10 --> nil
5 false and error() --> false
6 false and nil --> false
7 false or nil --> nil
8 10 and 20 --> 20
```

Listing 2.2: Lua Syntax Sample 2 (taken from [10])

Using this behaviour of logical operators allows for quickly and easily assigning variables using, for example, a configuration file.

Lua allows for assigning multiple variables on the same line as shown in listing 2.3. When used in conjunction with returning multiple values from a

function, this can reduce the amount of code required and therefore improve development time.

```
1 a, b = 100, 200
2 print(a + b) -- prints out 300
```

Listing 2.3: Lua Syntax Sample 3

With the introduction of Lua 5.2, the `bit32` library added support for bitwise operations and with Lua 5.3 came native support for bitwise operators. Behaviour is exactly the same as in any other language with support for them as is shown in listing 2.4.

```
1 print(1 & 2) -- bitwise and prints out 0
2 print(3 | 4) -- bitwise or prints out 7
3 print(5 ~ 6) -- bitwise xor prints out 3
4 print(7 >> 1) -- bitwise left shift prints out 3
5 print(8 << 1) -- bitwise left shift prints out 16
6 print(~9) -- bitwise not prints out -10
```

Listing 2.4: Lua Bitwise Operations

Lua has some syntactic sugar which helps to simplify what the Lua developer will have to interact with.

```
1 -- Inline declaration of a table
2 local tab = {
3     foo= 45,
4     tab = {
5         x = 23
6     },
7     place = 10
8 }
```

Listing 2.5: Syntactic Sugar Sample 1

In listing 2.5 we can see how to inline declare a table with a sub table that is also inline declared.

```lua
function tab.func(self, y)
    print(self.tab.x + y)
end

tab1.func(tab, 25)
```

Listing 2.6: Syntactic Sugar Sample 2

We can see how to declare a member function using some syntactic sugar in listing 2.6. Which is equivalent to listing 2.7.

```lua
-- Declaration of an example member-like function
function tab:func(y)
    print(self.tab.x + y)
end

tab:func(25)
```

Listing 2.7: Syntactic Sugar Sample 3

There are also several ways to access entries within a table all of which can be mixed and matched based on the programmers preference.

```lua
print(tab.place)    -- prints out 10
print(tab["place"]) -- prints out 10
print(tab.tab["x"]) -- prints out 23
```

Listing 2.8: Syntactic Sugar Sample 4

Lua's support for coercion allows for the automatic conversion of strings and numbers, this helps developers with rapid development as they would not need to have to think about converting types in many situations when Lua itself handles it for them. Bitwise operators always convert numbers to integers, mathematic operations always convert operands to floats [10].

String concatenation using the `..` operator converts numbers to strings wherever is needed but using the `string.format` function will allow for fine tuned control of how the number will be converted.

Examples of Lua's coercion in action are shown in listing 2.9.

```lua
print(1 .. 2.2 .. "three") -- prints out 12.2three
print("1" & "2") -- prints out 0
print(1.1 & 2) -- errors out
print("1.1" / 2.2) -- prints out 0.5
print(string.format("%f", "5.5")) -- prints out 5.500000
```

Listing 2.9: Coercion Sample 1

Lua's tables have a length operator in the form of `#`, it returns the length of a table as would be found by the `ipairs` operator. It counts through integers starting at 1 and stopping when the first nil value is found, this means that the operator is only useful for finding the length of array-like tables.

2.1.4 Functions

Lua allows for a function to return multiple values, this is an interesting concept and can have some downsides, it can make it quite difficult to correctly name a function to reflect the return values. It is a very powerful feature and if handled correctly can improve productivity especially when combined with Lua's variable number of arguments. An example is shown in listing 2.10.

```lua
function sum(...)
    local args={...}
    local ans = 0
    for k, v in pairs(args) do
        ans = ans + v
    end
    -- returns the sum of all values and the number of values
    return ans, #args
end
```

Listing 2.10: Functions Sample 1

Functions, in Lua, are first class values with proper lexical scoping [7]. A first class value in Lua means that a function is effectively the same as any other regular variable, this allows a Lua developer to pass functions into other

functions as arguments, return functions, use functions of keys and values of tables and can be assigned to a variable. This is a very valuable trait for a high level scripting language to have as it allows for very interesting techniques such as functional programming, closures are fully supported in Lua as a result of this.

Lua has proper tail calls which allows for recursion. If the last thing that a function does is call another function, the calling function is no longer required and is no longer required on the stack, this allows for an infinite recursion without overflowing the stack [7]. An example of a Lua tail call is shown in listing 2.11.

```
1 function func(x) return 50 end
2
3 function tailCall(x)
4     return func(x)
5 end
```

Listing 2.11: Functions Sample 2

Without proper tail calls, recursion could overflow the stack but with proper tail calls this is impossible and fits in perfectly with a high level scripting language such as Lua.

2.1.5 Tables

Tables are the only data structure in Lua but are extremely flexible and can be used to provide the functionality of almost any data structure as required. They are very similar in use to a hash map and allow for any variable of any type to be assigned to a key or value within the same table with the exception of `nil`, it is also possible to treat a table similar to an array as shown in listing 2.12.

```
1 -- declaration of a table using numerical indices
2 local array = {
3     "this is a string",
4     105,
5     function(x) print("this is a function"..x) end
```

```
}

print(array[1]) -- prints out this is a string
print(array[2]) -- prints out 105
array[3]("!") -- prints out this is a function!
```

Listing 2.12: Tables in Lua Sample 1

Lua has a variety of built in functions for the use of manipulating and getting data from tables all available through the table library [10].

To loop over all key-value pairs in a table lua provides the `ipairs` and `pairs` iterators. Used with a `for` loop a Lua developer is provided with a powerful method of quickly and easily accessing anything within a table as shown in listing 2.13.

```
local tab = {
    foo = "this is a string",
    105,
    "this is another string",
    [333] = function(x) print("this is a function"..x) end
}

for k,v in pairs(tab) do
    print(k, v)
end
--[[ prints out the following:
1 105
2 this is another string
333 function: 0x1663b90
foo this is a string
]]

for k,v in ipairs(tab) do
    print(k, v)
end
--[[ prints out the following:
1 105
2 this is another string
]]
```

Listing 2.13: Tables in Lua Sample 2

The `pairs` function iterates over all elements in a table while the `ipairs` function will return all key-value pairs where the keys are numbers, starting at the first index and stopping when it hits the first nil value. [7]

Getting the length of a Lua array is found by calling `table.getn(tab)` however, to find the size of a table with mixed key types or non-consecutive indices, a generic `for` loop with a count variable would have to be used. In cases where mixed key types are used it shouldn't be required to know the size as the pairs cannot be accessed by a normal for loop in any case.

Weak tables are tables that consist of elements that are weak references, this can be the keys or the values, if the only reference left to an object is a weak reference the garbage collector collects it. They are often used when a word needs to be reserved without actually creating an object in memory.

2.1.6 Metatables and Object Oriented Programming

All tables in Lua are permitted to have a metatable[7]. They allow for changing the behaviour of various aspects of a lua table, such as the add operator, by setting the `__add` value to a new function this code will be called instead of the default add functionality as is shown in listing 2.14.

```
1  local metatable = {
2      __add = function (a, b)
3          return 5 -- the result will always be 5
4      end
5  }
6
7  t1 = {}
8  -- set the meta table of t1 to metatable
9  setmetatable(t1, metatable)
10 local t2 = t1 + t1
11 print(t2) -- prints out 5
```

Listing 2.14: Metatables Sample 1

These functions are called metamethods and allow for any interaction with a table to be altered. While classes do not exist in Lua, using a meta table and first class value functions, behaviour similar to object oriented programming

can be achieved especially with the use of factory methods. An example of a class is shown in listing 2.15. On line 13 a table with the metatable `Dog` is created and returned to `myDog`.

```
1  Dog = {}
2  function Dog:new()
3      local o = {}
4      setmetatable(o, self)
5      self.__index = self
6      return o
7  end
8
9  function Dog:bark()
10     print("Bark!")
11 end
12
13 local myDog = Dog:new() -- creates instance of Dog
14 myDog:bark() -- prints out Bark!
```

Listing 2.15: Metatables Sample 2

While this concept is very useful, if the `new` function is ever changed by mistake, a difficult to locate bug could be created.

2.1.7 Coroutines

A coroutine is similar to a thread in that it has its own line of execution, its own stack, its own local variables, and its own instruction pointer but shares global variables and almost anything else with other coroutines. The main difference to a thread is that a program with threads runs things concurrently while a coroutine in Lua is collaborative and at any given time only 1 is running [7]. While there may not be a performance gain from using coroutines, it allows a developer to easily orchestrate multitasking that would otherwise be difficult to implement.

To create a coroutine it is as simple as calling the `coroutine.create` and passing a function as shown in listing 2.16.

```lua
co = coroutine.create(function (a,b)
    coroutine.yield(a + b, a - b)
end)
print(coroutine.resume(co, 20, 10)) -- prints out true 30 10
```

Listing 2.16: Couroutine Sample [7]

2.1.8 Garbage Collection

Lua automatically manages memory, Lua destroys objects when all references to them are lost. There is no function to delete them manually, a program can only create objects which means that the developer will not need to bother with the majority of memory management [7]. Lua has no issues dealing with cyclic references unlike simple garbage collectors. Lua allows for manually calling the garbage collector using the `collectgarbage` function as even the best garbage collector isn't perfect, the Lua garbage collector sometimes needs some help. It is even possible to manually pause, stop and restart the garbage collector to allow for fine tuned adjustments to the use of memory in a Lua program, it is also possible to get the total amount of memory being used.

In very high performance scenarios a Lua developer may need to assume control of the garbage collector, a simple example is shown in listing 2.17.

```lua
collectgarbage("stop") --stops the garbage collector
print(collectgarbage("collect")) --prints out 0
mytable={}
for i=1,10000,1 do mytable[i]=i end --fill the table with 10000 items

print(collectgarbage("count")) --prints out 285
mytable = nil -- we remove the reference to the table
print(collectgarbage("count")) --prints out 285

collectgarbage("collect") --runs the garbage collector
print(collectgarbage("count")) --prints out 28
```

Listing 2.17: Manual Garbage Collector Sample

Although the reference to the table is removed the table still exists in memory until the next pass of the garbage collector.

2.1.9 The Lua C API

Lua is an embedded extension library and cannot be used as a stand alone program, a program can use Lua with little input from a programmer using the provided C API. The API puts flexibility and simplicity above all else, this can sometimes make it difficult to implement for an inexperienced C developer as there is no built in type checking and no clean errors, this is down to the developer to handle [7].

A major component of the Lua C API is that new functions can be registered with the virtual machine allowing for behaviour that would be otherwise impossible to implement in pure Lua. Class like objects called userdata can also be implemented in C and used in Lua allowing for interaction between the program and Lua, in the case of a game engine, objects such as sprites could be accessed in both C and Lua and provide a developer with the ability to create high performance games in a much easier to use package than C.

An example of registering a function in the Lua vm is shown in listing 2.18.

```
1
2  static int l_sin (lua_State *L)
3  {
4      double d = lua_tonumber(L, 1); /* get argument */
5      lua_pushnumber(L, sin(d)); /* push result */
6      return 1; /* number of results */
7  }
8  void registerSin(lua_State *L)
9  {
10     lua_pushcfunction(l, l_sin);
11     lua_setglobal(l, "mysin");
12 }
```

Listing 2.18: Sample function implementation in C API [7]

2.2 Game Engines

There are many game engines already out there covering a wide range of performance and development styles, engines such as Unreal Engine [5] and Unity [18] have integrated development environments and require all development to be done through them. Other game engines, such as LÖVE [12] can be developed in any text editor using any work flow that the developer chooses, this improves prototyping times and allows developers to get comfortable much quicker as they are using tools that they are well used to.

Open source allows for the collaborative development of any project and would therefore be ideal for a game engine, by allowing people to fork and extend a game engine a very wide variety of features can be implemented, enabled and disabled based on the individuals needs.

2.2.1 LÖVE

LÖVE [12] provides a limited set of features and leaves the majority of the work on the internals of the engine to the game developer. This allows for very fast prototyping of small example games but when creating a larger game things can get difficult. The engine does include some things at a lower level such as physics, GLSL shaders, image loading, TCP and UDP networking. Because LÖVE doesn't manage entities there is quite a low performance ceiling especially with the addition of lighting and particles, other engines address this at the cost of development time. LÖVE is completely open source and completely free.

2.2.2 Unity and Unreal Engine

Unity [18] and Unreal Engine [5] both manage every aspect of the game while still allowing massive extensibility. Both have unique development flows and both take a very long time to master, because of just how massive they are, development can be very slow when compared to lightweight prototyping engines. The Unity engine is not fully open source but allows source access for certain modules of the engine while the latest versions of Unreal Engine

are fully open source, both adopt similar payment patterns making the developer pay based on the number of sales rather than a fixed fee, this allows for smaller studios and individuals to create very professional products.

2.2.3 Cocos2d

Cocos2d is an open source framework designed for creating cross platform games, apps and other GUI based programs [3], it has support for scripting in both Lua and JavaScript but requires C++ code to initialize any scripts. There is an IDE created for cocos2D that provides similar workflow to the full engines while having the flexibility of text only engines such as LÖVE.

Chapter 3

Design

In this section of the report the design and library choices of the game engine will be identified and shown alongside full explanations of how they fit into the product as a whole.

3.1 Requirements

Using an agile methodology will be an important part of the development of this project as it provides a far superior work flow when compared to far more static methodologies, for example, the waterfall method. This will result in an improved rate of work and the addition of currently undiscovered ideas. Although an agile methodology will be in use, the requirements of the project are still fixed and all efforts will be devoted to ensuring that they are fulfilled.

For a game engine to be complete, a wide variety of features are required especially when a high level scripting language is the only input for developers to use when creating a game.

The engine will provide a Lua interface to all required core engine features using the Lua C API.

The engine will need the ability to render, reposition, angle and animate

sprites provided from a spritesheet.

Physics will also need to be provided by the engine to allow for much improved performance over having physics implemented in Lua.

Both the physics and rendering capabilities of the game engine will need to be fully accessed through the provided Lua API. The Lua API should be consistent and easy to use, matching, where possible, the Lua style completely.

3.2 Library Selection

Parts of the engine will be handled by external libraries, an important property of these libraries is performance and modernness as well as cross platform capability. Ideally the libraries will be open source.

Rendering, physics, scripting and sound will be handled by external libraries and implemented in the game engine. For rendering and sound there are several choices: SDL, SFML, DirectX and OpenGL with a sound library. SDL is widely used and has very good performance but it is written in pure C so has no object oriented code and is therefore quite dated when compared to SFML. DirectX is only available on the Windows and Xbox platforms and is therefore not ideal for this project as cross platform is a requirement. OpenGL with a sound library has the best performance of all of the options but is also the most difficult to use and both SDL and SFML wrap OpenGL and provide access to the context so would be better options.

SFML or Simple and Fast Multimedia Library is written in C++ and is available on Linux, Mac, Windows and FreeBSD and is fully open source with its code available on Github. SFML handles everything from window management and input to sound and networking, the performance is very similar to that of SDL but provides a modern object oriented API unlike SDL. These reasons make SFML the perfect choice for the projects requirements.

For the physics library there is only one real option when it comes to games. Box2d is an open source C++ physics engine for simulating rigid bodies in

2D [1]. It has excellent performance and is well proven for its suitability for this project as many games and game engines on all platforms use it to great effect.

As for scripting the game engine there are several options such as C#, Javascript and Lua. For the purposes of this project, Lua is the best option due to it's simple binding process and powerful syntax. Many games use Lua as their embedded scripting language such as Garry's Mod, World of Warcraft, Angry Birds and many, many more.

For the UI an external library created to provide UI capabilities to SFML could be used such as SFGUI [16] or TGUI [17]. This will need binding for use in Lua which could prove to be challenging, however getting the minimum working example would not be too difficult. It may be better for the development time if only text rendering is implemented as other UI elements can be achieved using sprites.

3.3 Engine Design

The first step to creating a game engine is to get each of the libraries working correctly in a single project. As the project will be completed in an agile fashion it may be difficult to know exactly what will be created and in what order, however, this does not mean that designs cannot be drawn up outlining potential solutions.

The engine will require a variety of features including an entity management system, such as the entity component system, this will allow for dynamic game objects that are easily pieced together by a developer. This system will need to manage a wide variety of entities in various forms including drawing, animating, playing sounds and others. An example entity that will behave as a player controlled car would have the class diagram as shown in figure 3.1.

For a car to be functional in a game it will need to be drawn and animated, have a physical presence in the world and receive player input. Behaviour can be easily placed into new components when repeated code is required, in the static inheritance system this is not as simple.

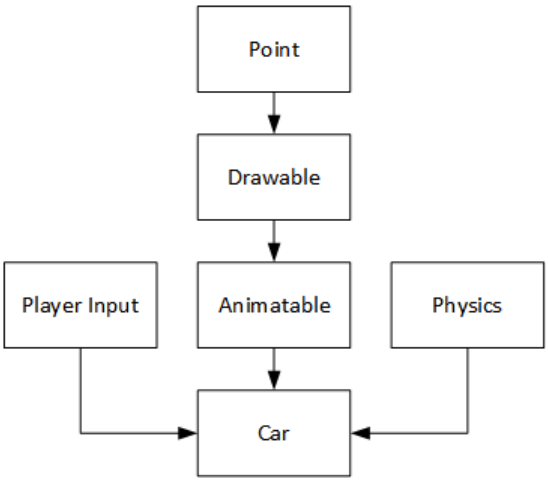


Figure 3.1: ECS Class Diagram Example

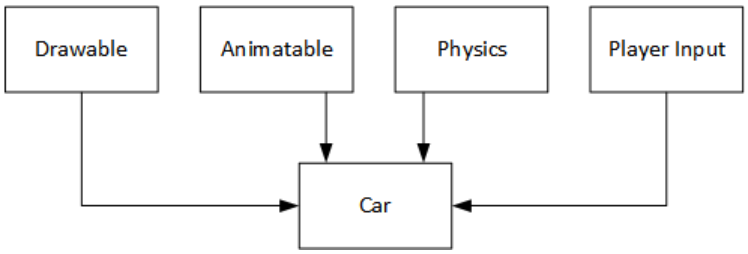


Figure 3.2: ECS Class Diagram Example

Another example of the same in game behaviour in the more traditional class hierarchy system is much simpler to implement but does not allow for the dynamism of the entity component system as is shown in figure3.2 and the diagram is clearly more complicated and may result in repeated code. For the sake of development it may be better to first implement the more traditional system and then convert over to the ECS style.

Lua script management will be entirely handled within the core of the game engine allowing for creating and removing objects, managing object behaviour, physics interactions, UI, the camera and more. The engine will be completely open source and will be easily expandable and improvable by

the community as a result.

Loading and saving of the game state should be integrated into the game engine to speed up development time for the game developer. Level loading should also be integrated, an external open source level editor application could be used for the creation of the levels, a JSON or other data file could then be loaded into the game engine reducing the development time even further. Settings should also be managed by the game engine allowing the Lua developer to load and save any required key value pairs using a provided API. By implementing these features in the game engine the game developer can focus more on the actual game rather than semantics.

Documentation is a requirement for people to be able to use the engine to create games effectively and helps to reduce the learning curve. Each part of the documentation should be completed after the completion of every Lua accessible feature, to ensure that it is appropriately maintained. Some kind of automatic documentation tool could be used similar to that of Golang's GoDoc or Java's javadoc, this would allow for the the documentation to stay up to date given any changes to the underlying code. Well documented examples could mitigate the requirement for extensive documentation, however, both would be ideal.

For the game engine to function well there are a number of required systems that must be incorporated into the core of the engine. There are 5 main elements, the window, the physics world, the resource manager, the gamestate manager and the Lua state. Designing the core of the engine to be robust and lightweight is important. Anything that the game engine is used for will rely heavily upon it, the better the design, the better the implementation.

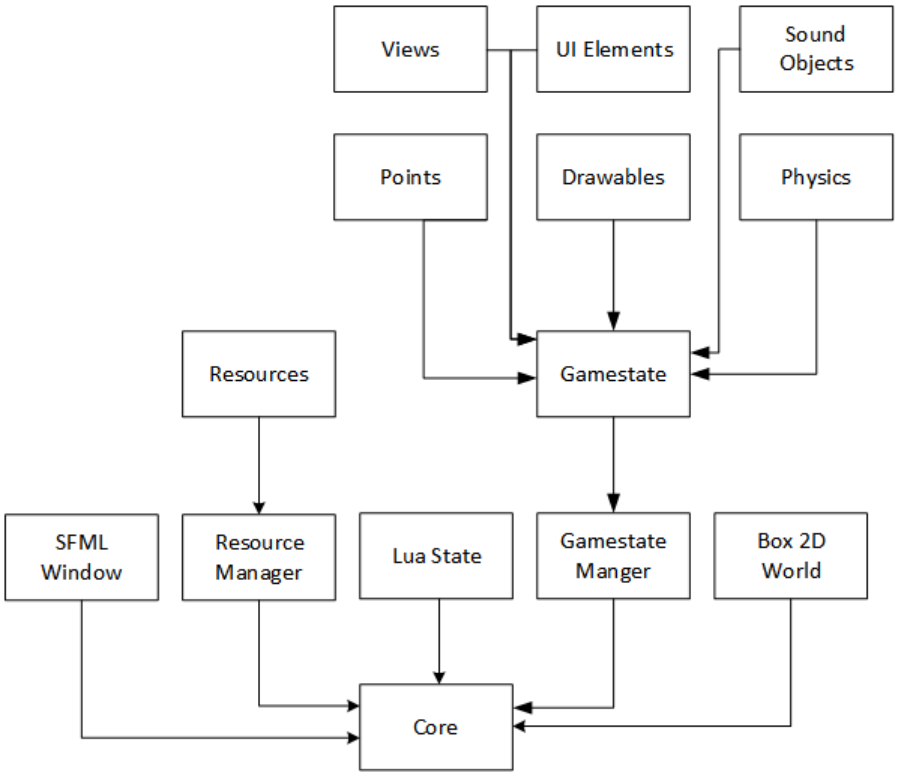


Figure 3.3: Core Systems Diagram

In figure 3.3 the design of the core is outlined, any lateral connections will be available to each system through the core, drawable components and sound objects will have access to the resource manager.

The storage of all game objects within the engine will be handled by the gamestate, allowing for the use of multiple game states will allow for the use of main menus, loading screens, pausing and resuming and much more. For this reason a strong design is very important

```
1 class Gamestate
2 {
3     Map of Sounds
4     Map of UI
5     Map of Points
6     Map of Views
7
8     SetPaused
9     GetPaused
```

```
10
11      Think
12          Points Think
13      Draw
14          Points Draw
15  }
```

Listing 3.1: Gamestate Pseudo Code

An example definition of the gamestate class written in pseudo code is shown in listing 3.1. Using polymorphism all objects that need to think or be drawn can be stored in the points map and then at every think time, all objects will think, and at every draw time, all objects will draw. By overriding the think and draw functions, different behaviour can be achieved for each sub class of point.

Reusing resources will ensure that the memory footprint of a game is kept as low as possible and this will reduce the number of file operations thus improving the performance.

```
1   class ResourceManager
2   {
3       Map of ResourceInterface
4
5       Exists
6       Load
7       UnLoad
8   }
9
10  class ResourceInterface
11  {
12      UseCount
13      Resource
14  }
```

Listing 3.2: Resource Manager Pseudo Code

An example of a highly generic resource manager is shown in listing 3.2, alongside an example of a resource wrapper class. It is highly generic, as within the resources map, all types of resource may be stored. The exists function will return if a given resource is currently in the resources map. The

load function will check that the map has the resource already and return it if so, otherwise, it will load from file and add to the map before returning the resource. The unload function reduces the use count of the resource by one and then checks if there are 0 uses, if there are the resource is removed from the map.

For input handling, there are two approaches that any game developer should have the option of choosing from, real time or as and when checking. Both should be accommodated, with specially named functions being called in Lua for the real time solution and functions that may be called anywhere in Lua querying whether a key is pressed for the as and when solution.

Animations are extremely useful to any game as they provide a sense of polish when used correctly. For this reason the animation system needs to be very simple to use but still very powerful.

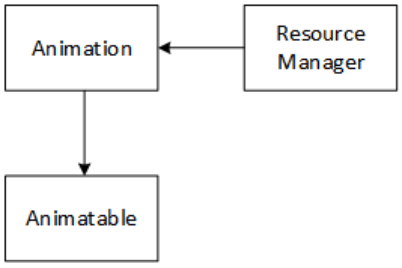


Figure 3.4: Animatable System

In figure 3.4 an outline of what is required for a functional animation system is shown. The resource manager will hold the sprite sheet for all animations so that the whole sprite sheet is only required to be loaded in a single time. The animation is then stored in the animatable so that any object can have as many animations as required.

The animation class holds all information required to animate from a sprite sheet including frame rate, frame count, frame size, the positions of each frame on the sprite sheet, the sprite sheet URI and various flags. All should be easily configured from Lua to ensure that the system is effective.

Using the entity component system uses composition over inheritance and allows for far greater flexibility than the standard inheritance approach, it also makes maintenance and expansion easier, this makes it an ideal

methodology to use in this project.

```
class Component
{
    Name
    ParentID
}

class DrawableComponent : Component
{
    SetViewTarget
    SetPosition
    SetAngle
    SetTexture

    Draw
}
```

Listing 3.3: Resource Manager Pseudo Code

All components inherit from a base component which provides functionality for setting and getting the parent of the component and provides the name of the component as is set in the child constructor. This is shown in listing 3.3 In addition to all game objects being composed of a variety of components there are systems required to handle these components. These systems know how the components must be used and must iterate over all components of its given type in order to provide the engine with that components functionality.

Additional components, beyond the core components, could include a networkable component, a health component, an AI component and many more.

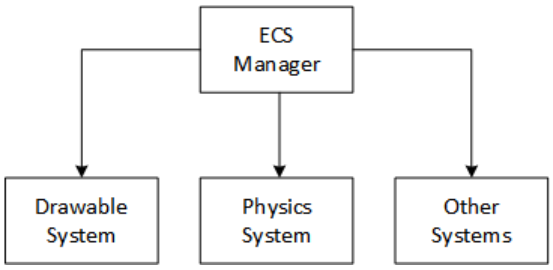


Figure 3.5: Animatable System

Handling the components will be done by a system per component all

handled by an ECS manager as is shown in figure 3.5.

3.4 Running

Testing game code needs to be a simple process, to achieve this, calling the game engine from the command line with a single argument is the best approach as this will also allow for the dragging and dropping of Lua files onto the exe. This method also means that an IDE would be able to launch the game as though it was anything else.

Chapter 4

Implementation

In this section the process of implementing the game engine and any issues faced will be described and discussed.

An important first step was to get Lua, SFML and Box2D building together within a single cpp project. Once the project was building correctly, the next step was to create some example classes and a simple main program loop to create a test bed for future work. Setting up a Github repository was another important step, this is a big project, change control is extremely useful especially when the project is spread over many source files.

At each major stage in the development cycle testing the game engine on other platforms was ensured as a cross platform game engine is far more useful to a developer than one that is locked down to a specific platform. Doing so should never take much effort since all code will be written using cross platform libraries using modern C++.

Creating simple classes that showed that SFML works allowed for the creating of the first classes as shown in the class diagram. The drawable and point classes were then created as well as the surrounding framework to allow for the storing and retrieval of the objects from both Lua and in cpp tests.

The point class is purely virtual and cannot be instantiated, it provides a think function that will be overridden by all sub classes that are required to do work every cycle of the main loop. This style of programming reduces

the amount of duplicated code in the core of the engine.

The drawable class provides the ability to draw a sprite at a set position and inherits from the point class so that it can be stored in the same vector as other entities and can then be casted as is required. This is a standard example of using the type theory of polymorphism [13] and allows for all subtypes to be handled by the same functions while having, potentially, fully unique behaviour.

Creating the resource manager proved to be challenging but ultimately very satisfying when it worked correctly. It allows for the end user of the game engine to not have to bother with resource management as the core engine will handle it, all resources are loaded from file for the first time but are otherwise reused by passing a smart pointer to the resource, when the resource is no longer in use anywhere it is unloaded from the resource manager.

The initial versions of the resource manager used raw pointers, this caused a few issues with leaving dangling pointers when trying to write modern code, to resolve this issue, smart pointers were used. The shared pointer class from the C++ standard library was the perfect type of smart pointer for use in this case as the resources were being shared an additional time with each use. Shared pointers also allowed for getting the use count of itself as opposed to raw pointers which have no idea how many times they are being used.

```cpp
template<class T>
std::shared_ptr<T> load(std::string uri)
{
    if (this->resources.count(uri) != 1)
    {
        std::shared_ptr<T> rsc = std::shared_ptr<T>(new T);
        rsc.get()->loadFromFile(uri);

        Resource<T>* newResource = new Resource<T>(rsc);

        resources.insert({uri, newResource});
    }

    //resources[uri]->Useage++;
    auto rsc = resources[uri];

    return dynamic_cast<Resource<T>*>(rsc)->resource;
}
```

Listing 4.1: Resource Manager generic load function

As shown in listing 4.1 C++ templating is used to ensure that the load function of the resource manager is capable of handling all types of resource. The first thing that the function does is check if the resource is already in the map, if it isn't, it then creates a smart pointer and loads the resource from file and inserts it into the resource map. The resource is then pulled from the map using the uri and returned to the caller. The usage is not incremented as the resource manager simply uses the count functionality of smart pointers instead of manually handling them.

Up to this point there was no Lua integration at all, to begin the game engines API, initialisation and a think function were provided, this allowed the cpp environment to call the init function when the Lua world needed to be created and called the think function every iteration of the main loop.

User input is a requirement too, so getting when mouse buttons were pressed and released and when keyboard keys were pressed and released were all provided to the Lua API in the form of hooks that were called as and when the events happened. Getting the current mouse position and the state of all buttons and keys was also provided so the end user of the engine can

decide their preferred method of handling input within their game.

Interaction with the window through SFML allows for the user of the game engine to ensure that their game works on all resolutions. Getting the resolution of the window is currently added to the API but setting the size of the window is not. Adding this to the Lua API also provides control to the lower level systems that would otherwise be fully inaccessible from Lua.

The resource manager is at the core of all file based operations in the game engine, allowing for its expansion to hold other resources has meant that the implementation of sound within the engine was trivial, using only systems that already existed. Support for sound in Lua was then also added to allow for use of sound effects by the game developer.

The animation system must be tightly integrated into the core of the engine to ensure that there will be excellent performance passed forward to the Lua API. By extending the drawable class some rendering code had to be duplicated in order for the correct behaviour to be possible.

The animation class handles all of the frames, settings and timing for all animations within the game engine, having a class handling all of this allows for various systems and processes to use the same code. It also ensures that all animation code is easy to find and modify should the need arise. The UI system could use the animation class just the same as the animatable class could. Seamless integration with Lua was required since animations are an important part of any game, ensuring that this was achieved was very difficult.

In early versions of the animation system the sprite sheets were cut up into frames and stored in the resource manager separately from the animation class. Through stepwise refinement the system was changed to simply require integer rectangles of the frames, this simplified the system and reduced the reliance on the resource manager, the performance improved as a result.

```
    local ss = "resources/textures/metalslug_mummy37x45.png"

    local anim = Animation.New("walk", ss, 37, 45, 50, 18)
    anim:SetLooping(true)
    anim:Regenerate()

    local testDrc1 = DrawableComponent.New()
    testDrc1:SetOrigin(18, 22)
    testDrc1:AddAnimation(anim)
    testDrc1:SetAnimation("walk")
    testDrc1:SetAnimates(true)
```

Listing 4.2: Lua API Animation

Integration with Lua was then finalised as is shown in listing 4.3, this style fits with the Lua style and is very easy to learn and use effectively.

Box2D was added to the engine and to the Lua API. Binding the fixture definitions and body definitions was an interesting challenge but was difficult to decide the best method. It was clear that using a table with only the required parameters was the best option, as having to use a large selection of setter functions would have been painful for the end developer.

```
box.physics = PhysicsComponent.New(
    b2BodyDef.New({
        active = true,
        type = "dynamic"
    }),
    b2FixtureDef.New({
        density = 1,
        friction= 1,
        restitution = 0.3
    }),
    1, 1)
```

Listing 4.3: Example of creating a Box2D physics component

As shown in listing 4.3 the Lua required to create a physics component is very simple and can easily be modified to allow for any situation. The keys in the definition tables are the same as the ones found in the Box2D documentation.

Allowing for the deletion of the bodies proved to be reasonably difficult as just removing the instance of the physics component was not enough, the body also had to be removed from the physics world.

Management of what the players can see needs to be handled by the game engine to simplify the Lua code required to achieve the same thing, this can be accomplished by using the SFML view class and setting the render target on the drawable objects. This is easily exposed to Lua allowing the game developer to manage a complex feature. Using a single view as the camera to navigate the game world and a second view for the UI would be the perfect use case for this system. As it stands the system is quite naive and can result in looping through the drawable components more than once, this should be improved upon in the future.

Before the core of the engine got too big it was required to convert to the ECS design from the standard hierarchy design because the ECS style improves the expandability and makes the engine easier to use especially from the Lua API. Composition of entities could be left entirely down to the game developer rather than the engine developer. Conversion to an ECS based design proved to be quite straight forward as only the parts of the engine that directly interacted with the standard approach classes had to be changed.

Implementation of the ECS manager and the component systems was mostly a case of copying and modifying existing code before deleting the old classes, the Lua API did not need to change very much. Adding the add component functions was trivial. All references to the old system have now been removed and replaced with their ECS counterparts.

The animation system was almost implemented as designed but a clear improvement was found in that the drawable and animatable components could be combined, doing so removed any duplicated code and simplified the process for the end developer as they no longer need to think about overlapping behaviour. The animatable and animation classes were then exposed, in full, to Lua.

In order for the engines features and performance to be showcased, a pair of examples were created with the intention of releasing them with the source

code. First, a version of snake was created to show examples of using key pressed hooks for user input, text rendering, resource management as well as the general style of coding in Lua. A second code sample, called fall, was created to showcase the Box2D physics integration, performance and the camera system. Some developers prefer to learn by example rather than by reading documentation so this option was provided.

Testing something as broad as a game engine can prove to be difficult as there are so many components that could be causing issues. To test that each feature was working correctly, small code samples were created and ran with any issues being picked up along the way. During the development of both examples several issues were spotted and quickly patched, this style of testing is ideal for this style of project. Creating robust code can only be achieved through testing, to effectively test and improve the engine people need to use it and fix any issues they find by making pull requests on Github.

Chapter 5

Evaluation

In order to properly evaluate the product created over the course of this project, a game jam style event was organised in order to allow impartial developers to use the engine and provide their opinion on it's effectiveness.

For the game jam several participants were found with a wide range of skill, starting from first year computer science students to passionate hobbyist turned professional. This was done in order to get different perspectives on the game engine. A series of questions were asked after using the engine in order to ascertain their opinions.

Starting from the least experienced and moving towards the most experienced the results of the questionnaires will be analysed. The questionnaires can be found in the appendix of this report.

First is a first year computer science student called Roman. Overall, he provided very positive feedback across the board. They enjoyed using the engine and found it to be very easy to pick up and create with, they would also choose to use this engine if they had a game idea they would like to prototype.

Next is an engineer graduate, called Henry, with no education or professional experience in programming for games or otherwise, they managed to produce a simple zombie survival game within a few hours of starting. They praised the ease of setting up the engine and liked the fact that the engine was very

easy to use with little to no experience. A suggestion was made for improved support of different resolutions, this could be achieved by abstracting the positions and sizes to be virtual ones that are fully unrelated to the size and shape of the window.

Finally a professional developer, called Jim, with years of experience developing in many technologies including scripting in Lua for various games produced an excellent showcase of the engine utilising many of the major features of the engine with no prompts other than the use of the examples and Lua binding source code. Given his experience with Lua in other settings he found it very easy to pick up given just the sample code provided with the engine. During his time developing in the game jam a few points were brought up and the engine was quickly extended to cover his requirements. This shows that the engine is very easy to extend which is one of the objectives of the project.

Initially there were no physics contact hooks but within 15 minutes they were added to the engine as well as hooks provided to Lua.

Constructive criticism included providing better documentation for inexperienced developers, this is reasonable and should be one of the first things to be completed if the project is continued to be developed. Further constructive criticism included the lack of a large standard library such as a vector library, which is very relevant to game development, another very reasonable point.

In figure 5.1 a print screen taken whilst playing the game is shown. The objective is to collect as many gold rocks as you can in the hopper to the left of the screen by first shooting the large rocks to break them down into tiny rocks.

All work completed on the project has been towards achieving the objectives. Almost all objectives have been completed entirely, however, the fulfilment of the requirement for a fully featured Lua API could be improved upon. The API is complete but is missing some nice to have features such as a standard vector library and full control over the game window as per the feedback from the game jam. The lack of thorough documentation is also an issue, this can quickly be dealt with using automated tools and comments in the correct style. The game samples that are provided with the engine provided enough

Figure 5.1: Animatable System

documentation for even a total beginner to create something interesting and unique.

The engine has various entry points from C++ to Lua that are synonymous with other game engines. All of the hooks that relate to user input, physics collisions as well as the think and init functions all provide enough for almost any developer to complete any task.

The performance is superior to that of LOVE2D as the core engine handles much of the expensive processing in C++. As the physics engine and all rendering is taken care of in C++ much of what reduces the performance of LOVE2D has been improved upon.

The engine is proven to be easily expandable as the design was created from the ground up with expandability in mind and during the game jam several features were quickly and effectively implemented.

Chapter 6

Conclusion

Over the course of this project a scriptable 2D game engine written in C++ has been designed and created, the engine itself is, in its current state, a viable prototyping tool that may be used by anyone as the code is fully open source and available on Github. The engine handles a wide range of things for the game developer including resources, animations, sounds, the camera, user input and much more. It is very easy to extend and improve.

A game jam has been successfully organised in order to effectively evaluate the game engine, this proved the engines effectiveness and ease of use as people with no development experience were able to create games and people with a lot of experience found it to be similar in style to other engines and games.

In the future there are a wide range of potential features that could make this project even greater. Code that is required for every game such as loading and saving levels, a larger standard library that could help improve development times even further, additional components and much more.

Adding support for networking along with a networkable component would be a great addition to the engine allowing for multiplayer games to quickly be implemented using just the Lua API. An additional component that could be added is an AI component, this would handle all behaviour for allied or enemy units and would open the door to more advanced game play

mechanics with minimal code changes.

Replacing or supplementing the current SFML renderer with a 3D one would allow the engine to eventually compete with the likes of Unity while still providing an easy to use Lua API, this would allow for even new developers to create 3D games.

While completing this project I have learned a multitude of new techniques thanks to thorough research and a very interesting project. I have explored many ideas that I have been passionate about for years and am very happy that I have had the opportunity to create this product.

Bibliography

[1] *Box2D*. http://box2d.org/about/.

[2] *Cheat Engine*. http://www.cheatengine.org.

[3] *COCOS2D*. http://cocos2d-x.org/.

[4] *eLua*. http://www.eluaproject.net/.

[5] Epic Games. *Unreal Engine*. https://www.unrealengine.com. 1998.

[6] Luiz Henrique de Figueiredo. *Libraries and tools for Lua*. http://webserver2.tecgraf.puc-rio.br/~lhf/ftp/lua. 2017.

[7] Roberto Ierusalimschy. *Programming in Lua, Third Edition*. 3rd. Lua.Org, 2013. ISBN: 859037985X, 9788590379850.

[8] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. "The Implementation of Lua 5.0". In: (2005).

[9] Leslie Lamport. *Latex: A Document Preparation System*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-15790-X.

[10] *Lua 5.3 Manual*. https://www.lua.org/manual/5.3/manual.html. 2017.

[11] *Lua source code*. https://github.com/lua/lua.

[12] *LÖVE*. https://love2d.org. 2008.

[13] *Polymorphism*. http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf.

[14] Roberto Ierusalimschy and Luiz Henrique de Figueiredo and Waldemar Celes. In: *IN PROCEEDINGS OF V BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES*. 2001.

[15] Roberto Ierusalimschy and Luiz Henrique de Figueiredo and Waldemar Celes. "Lua – an extensible extension language". In: *Software: Practice & Experience 26*. 1996, pp. 635–652.

[16] *SFGUI*. https://github.com/TankOs/SFGUI.

[17] *TGUI*. https://tgui.eu/.

[18] Unity Technologies. *Unity*. https://unity3d.com. 2005.

Scriptable 2D Game Engine

Course-Specific learning outcomes

Throughout the course of this project several learning outcomes will be covered as well as potentially others.

- Use knowledge, abilities and skills for further study and for a range of employment in areas related to scientific and technical computing.
- Interpret legislation appropriate to computer professionals and be aware of relevant ethical issues and the role of professional bodies;
- Analyse, design, and implement algorithms using a range of appropriate languages and/or methodologies;
- Apply the principles and operation of languages, compilers and interpreters;
- Demonstrate effective communication, decision making and creative problem solving skills, and identify appropriate practices within a professional, legal and ethical framework;

If the engine is deemed during research to require an AI system the below would be applicable.

- Critically appraise and apply suitable artificial intelligence techniques for a variety of software systems.

Project Background

Game development can be very difficult without the correct tools, especially the prototyping phase. This project would allow for the creation of a game engine that is fit for prototyping an idea as well as being adapted for the final production version.

Prototyping is difficult as engines with excellent performance such as Unreal Engine or even Unity are very heavy and slow to work in and would take a vastly larger amount of time to get to a high-quality working example, especially in 2D. LOVE2D is excellent for quickly prototyping a game but the performance would suffer when the game began to grow in scope, a great way to remedy this is to have most of the heavy-duty code running on the C++ side. The game engine that needs to be created is one that is easy to prototype in as well as have the performance to handle a fully-fledged game.

Similar to how Unreal Engine 4 has high level drag and drop programming in the form of blueprints as well as full engine access in C++, an engine that is much lighter weight and easy to develop in is yet to be made especially for 2D game development.

Aim

Design and implement a viable open source 2D game engine using C++ that implements an interface to the backend for use in a high-level embedded scripting language for rapid and powerful development.

Objectives

Research and review what existing 2D game engines use for rendering, sound and which scripting language was used, as well as design decisions such as how much control the embedded language gets and how much of the game engine is written in the embedded language versus C++. Research into game engine design techniques will also be a necessity as engine design is a very deep and complex subject.

Allow interaction with game objects from both C++ and the embedded high level scripting language. This will be extremely important for ensuring that the designed game engine will be highly functional and easy to use.

Have various entry points that should be synonymous with what other game engines have such as standard entry points like initialise, update and draw.

Ensure full expandability. This will be a major factor when it comes to designing and implementing the core engine as the engine will need to be as general purpose as possible and should be tweakable for any use case.

Improve performance beyond LOVE2D by using techniques such as moving more of the core processing to the C++ side.

A maintainable codebase is very important for expandability and will ensure a fluid development process through the project as well as into the future beyond the project.

Document the engine and scripting language API. Implementing a game in a short space of time is very difficult without documentation as developers without any intimate knowledge of the engine will still be required to fully utilise the various features of the engine.

Compete in a game jam to verify how effective the various aspects of the game engine are under high stress rapid development.

Problems

There are many potential problems that could be run into. With each objective comes potential failure points that would need to be worked through as they are encountered.

Interaction with game objects from both C++ and the high-level scripting language could be very difficult and would take an excellent infrastructure plan to allow for it to be entirely implemented especially in a fluid and simple way.

Expandability may also prove to be a problem when it comes to the implementation as engine design is a very complicated subject, a lot of research will have to go into this part of the project to ensure that the design allows for the required feature set.

Ensuring that the documentation is all up to scratch could be very time consuming and could end up pushing the timetable out.

Required Resources

For the completion of this project no additional resources are required.

Timetable and Deliverables

Research – In depth research will be required before starting any of the other objectives to ensure that the design is on target.

Engine Implementation – The entire development process should be entirely possible within approximately 20 weeks unless there are any major hold ups.

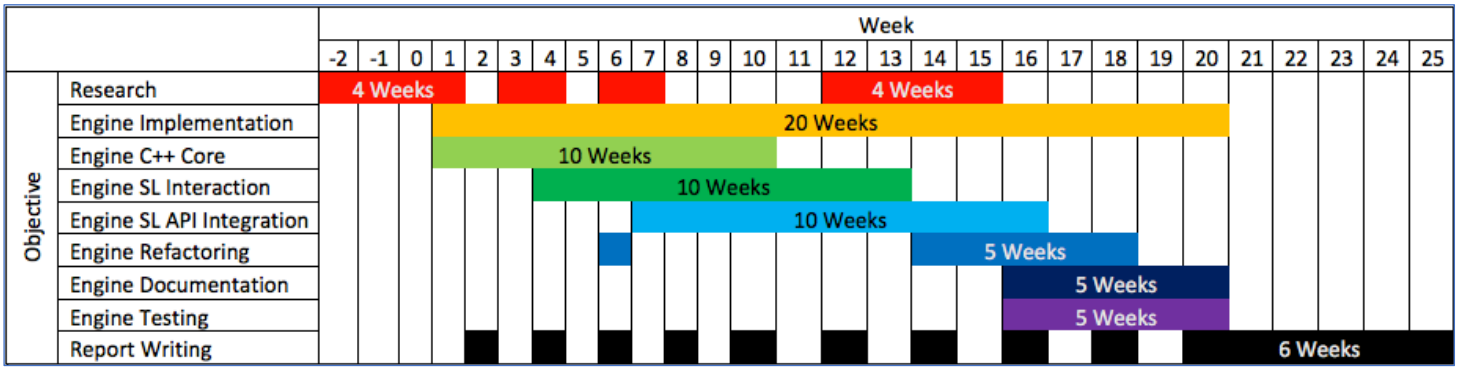
Engine C++ Core – The core of the engine will be very important for ensuring that the performance will be better than equivalents.

Engine SL Interaction – Ensuring that the scripting language has enough core engine interactions through things like keypresses and other hooks should take around 10 weeks.

Engine SL API Integration – Designing and implementing the scripting language API should take around 10 weeks.

Engine Refactoring – Refactoring will be completed in minor cases throughout the project, but the major refactoring that will be required after the engine is implemented should take around 5 weeks.

Engine Documentation and Testing – Documenting the engine will take place while testing every part of the engine and should take around 5 weeks.



Cyrus Hanlon 14061650

Huw Lloyd (supervisor) 29/10/2017

The MANCHESTER METROPOLITAN UNIVERSITY
Faculty of Science and Engineering

RISK ASSESSMENT COVER SHEET

REFERENCE NUMBER: NPC / 090517 / JDE1.49

SCHOOL: Computing, Mathematics & Digital Technology

TITLE OF WORK: CMT Projects involving software development only

LOCATION OF WORK: John Dalton Building computing facilities, computers at student's own home etc.

INTENDED ACTIVITIES (attach methods sheets (e.g. standard operating practices) and work schedules to this form):

General use of computers to develop and test software. Method sheets and work schedules not applicable.

PERSONS AT RISK (list names of all individuals (including status e.g. staff/student), and/or unit(s) / course(s) undertaking the activity. For students please indicate course and level, for staff give contact email / phone number):

Undergraduate students.

HAZARDS (provide a summary of the hazards anticipated and attach detailed assessments with appropriate risk control methods to this form):

Repetitive Strain Injury – work related upper limb disorder

Back injury resulting from improper posture

Eye strain

Fatigue Stress

Possible risk from 240v electrical mains supply

Are these hazards necessary in order to achieve the objectives of the activity?

Yes

Hazard Rating (delete as appropriate): **Low**

HAZARDOUS SUBSTANCES/MATERIALS USED AND HAZARD CLASSIFICATION

(appropriate COSHH data sheets / risk assessments must be attached to this form**):**

ALL CONTAINERS OF HAZARDOUS SUBSTANCES SHOULD BEAR CORRECT HAZARD WARNING LABELS.

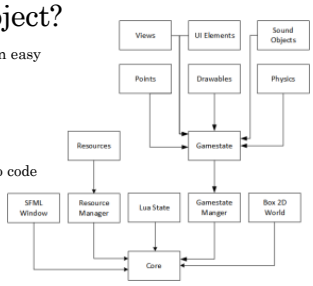
NAME OF MATERIAL *Please provide also approximate quantity and concentration if applicable.*	**HAZARD CLASS**	**HAZARD LABEL**	**DISPOSAL** *Hazardous materials must not be removed from laboratories. List disposal arrangements for* <u>all materials listed below</u> *in the location where* <u>the work will be carried out:</u>

2D Game Engine

By Cyrus Hanlon

What is my project?

- 2D Game engine that provides an easy to use and powerful Lua API
- High performance and low game development times
- Easily expandable
- Complete, users will only need to code in Lua



Major features - 1

Simple to use Lua API – (Peer review proves that)

ECS – (Easily Expandable without much development experience new components can be added to the game engine)

SFML – (Good performance with low development overhead)

Animations – (All handled by the engine so the Lua developer doesn't need in depth knowledge to implement polished games)

Major features - 2

Box2D – (Used in 2D games on all devices so has proven track record)

Resource management – (Lua developers don't need to bother with directly managing assets, they can refer to them by path)

Input Handling - (Takes out all direct hardware access from the Lua developer, could be expanded to handle controllers etc)

Expandability – (All systems are robust and generalised so third party developers can easily add new features)

How it works

```
local box = {}
box.drawable = DrawableComponent.New()
box.drawable:SetTexture("resources/textures/box.png")
box.physics = PhysicsComponent.New(
    b2BodyDef.New({
        active = true,
        type = "dynamic"
    }),
    b2FixtureDef.New({
        density = 1,
        friction= 1,
        restitution = 0.3--math.random( 50 ) / 100
    }),
    1, 1)
box.entity = Entity.New("ball"..#boxes)
box.entity:AddDrawable(box.drawable)
box.entity:AddPhysics(box.physics)
```

- This code is simple in Lua
- Behind the scenes a lot is going on

How it works

```
local box = {}
box.drawable = DrawableComponent.New()
box.drawable:SetTexture("resources/textures/box.png")
box.physics = PhysicsComponent.New(
    b2BodyDef.New({
        active = true,
        type = "dynamic"
    }),
    b2FixtureDef.New({
        density = 1,
        friction= 1,
        restitution = 0.3--math.random( 50 ) / 100
    }),
    1, 1)
box.entity = Entity.New("ball"..#boxes)
box.entity:AddDrawable(box.drawable)
box.entity:AddPhysics(box.physics)
```

- Example of creating a box in Lua
- Create both component types
- Create entity
- Add the components to the entity

How it works

```
local box = {}
box.drawable = DrawableComponent.New()
box.drawable:SetTexture("resources/textures/box.png")
box.physics = PhysicsComponent.New(
    b2BodyDef.New({
        active = true,
        type = "dynamic"
    }),
    b2FixtureDef.New({
        density = 1,
        friction= 1,
        restitution = 0.3--math.random( 50 ) / 100
    }),
    1, 1)
box.entity = Entity.New("ball",.#boxes)
box.entity:AddDrawable(box.drawable)
box.entity:AddPhysics(box.physics)
```

- Create userdata objects
- Insert components into appropriate containers in the gamestate
- Request a resource from the resource manager and set it on the drawable
- Create Box2D physics object and add to the Box2D world
- Create entity and place it in the entity container in the gamestate
- Attach the components to the entity by setting the parent property

Evaluation - Peer Review

- Jim | 27
  - Programmer with 3 years professional experience
  - 5+ years making game modes for Garrysmod in Lua
- Wessel | 20
  - 1 year of education in programming
  - CS Student
- Roman | 20
  - 1 year of education in programming
  - CS Student
- Henry | 23
  - Personal interest in programming
  - Engineering graduate

Examples

- Snake
  - Simple complete game example
- Fall
  - Box2d and SFML camera example
- Quarry
  - Made by Jim
- Zombie Survival
  - Made by Henry

Evaluation Questionnaire

Name: Henry Gibson

Did you enjoy using the game engine?

Yes, I have tried several times before to make a simple game in a 2D game engine, but there has always been so much initial setup required and initial knowledge, that I've always given up, however with this I was able to very easily get things working.

Did you find using the game engine to be intuitive?

Yes, it was extremely easy to get started, zero initial set-up required. Lua was really easy to use, based on my limited experience with MATLAB and Python for engineering applications.

Did you find it's feature set to be complete?

I was able to do everything I would expect from a simple 2D game engine, I feel that any further functionality would be easy to make within the .lua file.

Would you need the support of a technical person to be able to use the engine?

I think that anyone with a basic understanding of programming (not necessarily experience) would be able to use the engine, any questions regarding lua syntax can easily be found online.

Was the engines API consistent to use?

Yes

Would you imagine that most users of the engine would find it easy to learn how to use?

Yes

Did you need to learn a lot before you could get going with the engine?

Definitely not

Would you use this engine to prototype a 2D game?

Yes

Are there any features that you thought should be included?

Maybe more support for different screen sizes

If you have any further points you wish to put forward, please write them below.

Evaluation Questionnaire

Name: Roman Reichardt

Did you enjoy using the game engine?

Yes.

Did you find using the game engine to be intuitive?

Yes, I had no issues getting used to it.

Did you find it's feature set to be complete?

Yes, all the functionality I desired was present.

Would you need the support of a technical person to be able to use the engine?

No, the system is very easy to use.

Was the engines API consistent to use?

Yes.

Would you imagine that most users of the engine would find it easy to learn how to use?

Depends on their knowledge about computer science but overall I think it is a easy-to-use engine.

Did you need to learn a lot before you could get going with the engine?

No, basic programming knowledge is sufficient.

Would you use this engine to prototype a 2D game?

Yes.

Are there any features that you thought should be included?

No not in particular.

If you have any further points you wish to put forward, please write them below.

Evaluation Questionnaire

Name:

Jim van Kouwen

Did you enjoy using the game engine?

Yes, it was a lot of fun!

Did you find using the game engine to be intuitive?

Having coded Lua in other game engines this all came very natural to me. The provided examples provided everything I needed to know to get started.

Did you find it's feature set to be complete?

Although the engine provides everything you need, you do need to implement some generic things yourself that would have been nice to be provided, like a vector library in Lua or additional functions to look up game objects.

Would you need the support of a technical person to be able to use the engine?

Not at all.

Was the engines API consistent to use?

Yes.

Would you imagine that most users of the engine would find it easy to learn how to use?

For me the examples made it easy. Inexperienced developers might need a bit more documentation.

Did you need to learn a lot before you could get going with the engine?

No.

Would you use this engine to prototype a 2D game?

Yes, I think this is very suitable to quickly whip up a concept and trying things out.

Are there any features that you thought should be included?

As above, 2d vector functions and math, and ways to look up game objects. All is possible to implement myself but since everyone will need this it would be nice to be provided out of the box.

If you have any further points you wish to put forward, please write them below.

It was a lot of fun working with this engine! I would like to thank Cyrus for his efforts. When I had a few requests he was able to implement them quickly.

OneDrive Link: https://stummuac-my.sharepoint.com/:u:/g/personal/14061650_stu_mmu_ac_uk/Ee0M7wLuyvdBiu2FrcX5bDQBQbpX4jA4n_ojZZJ9GXafTw