Modding Dota 2 Using Lua

Zach Giannuzzi, Zac Goodsell, Tyler Gorman

Siena College

Abstract

For this project we wanted to use Lua's scripting ability to modify DoTA 2. The plan for our mod was to create a dodge ball like game. First, we learned about Lua and its scripting abilities. Next we started modding the game by creating a map for our dodgeball game to be played on. This was fairly straight forward after learning how to use the HAMMER map making tool from Valve. After making the map we started to focus on making the items and the interaction of the item and the users. Using key values in Lua we were able to identify when a user picked up a ball and if a thrown ball hit an enemy.

1 Overview

Lua is a scripting language originally developed in 1993, since then, Lua is currently on its 5th version, and supports procedural, object-oriented, functional, and data-driven programming. Simmarly to Python, Ruby, and Javascript, Lua is dynamically typed, meaning you don't have to specify a data type at compile time, the type will be determined at runtime. However the DOTA 2 engine is written primarily in C++, which is statically typed. Therefore we will still need to be aware of the data types when calling the API [4]. Lua has been used in applications such as Adobes photoshop lightroom and is currently "The leading scripting language in games such as World of Warcraft and Angry Birds" [1]. The research and learning of Lua will be done by modding the video game Dota 2. In Section 2, we discuss basic functionality of Lua, this includes, types and values, expressions, statements, key-pair values, tables, and methods.

DoTA 2 was developed using both C++ and Lua. C++ is used to handle a majority of the background functions such as AI bots [3]. The reason the entire games wasn't developed using C++ was because a scripting language was needed to develop in game behaviors. Using a

scripting language also makes it easier to develop a game because code is compiled as instructions and can be seen as soon as you write it without having to recompile machine code. Data driven ability are use throughout a majority of Lua in DoTA 2. These abilities have npc_abilities.txt key values files that contain ability metadata. Entity scripts are used for adding new functionality to the game. These scripts are convenient in a data driven fashion script by executing scripts when certain in game events happen. Script filters allow for the creation of custom game with the ability to modify in game actions. Each script passes the values from an event into a table and by editing those values the user can modify in game behaviors.

2 Basic Functionality of Lua

2.1 Types and Values

As stated above, lua is a dynamically typed language. In Lua there are eight types: nil, boolean, strings, userdata, function, thread, and table. The type function can be used at anytime to know what type of a given value. Lua's nil is equivalent to java's null value, as it is used as a non value. Lua's boolean types are mostly the same as other languages in that Lua uses the true and false values. Similar to Javascript, Lua does not have an integer type, rather, Lua uses double-precision floating-point numbers to represent all numbers. The reasoning behind this is "when you use a double to represent an integer, there is no rounding error at all (unless the number is greater than 100,000,000,000,000)" [2] therefore there is no need for an integer type. Strings in Lua are a sequence of characters, and similar to C, strings are immutable. Tables and functions will be discussed further in the paper.

2.2 Expressions

Lua has all the usual operators as other programming languages, Lua's relational operators are similar to Java's relational operators except ~= in Lua is equivalent to Java's !=. Lua compares everything according to their type, for example "5" = 5 returns false. Lua has 3 logical operators, and, or, and not. Lua uses shortcut evaluation, like java will, meaning it will only evaluate a second operand when necessary. String concatenation uses the ".." operator to concatenate strings, and if you are concatenating a string with a number, Lua will convert the

number to a string. All of the operators in Lua are left associative, except for the exponentiation and concatenation operators which are right associative [2].

2.3 Statements

Assignments in Lua are the same as most languages, and also supports multiple assignments. If there is an assignment such as a,b,c = 0,1 Lua will adjust the number of values to shorter than the assigned variables and give nil to all other extra variables that were not assigned. Another assignment operation Lua can do is assign several variables to a function.

```
a, b = f()
```

Figure 1 an example of assigning variables to a function.

In figure 1, a and b are assigned to a function, for example let's assume that the function f() can return multiple values, the first value returned will be assigned to a, and the second result will be assigned to b.

Variables are always considered global unless specified with the local keyword, by declaring a variable as local, the scope of the variable is limited to the scope where the variable was declared. Control Structures in Lua all have an end terminator at the end of a control structure block. Lua has an if then else, a while loop , numeric and generic for loops, and repeat-until control structures [2].

2.4 Key Value Pairs/Tables

Key value pairs and tables are very important in understanding Lua's functionality with Dota 2. Lua uses a data structuring mechanism known as tables. Tables in Lua can represent many different data structures such as, ordinary arrays, symbol tables, queues and much more. The tables themselves can have as many entries and different types as you want. Tables in Lua are objects, there is also no way to declare a table, a table is created using a constructor expression, you will see this in the example below. The way it works is the table uses associative arrays, associative arrays can be index with not just numbers but strings and any other value of the language [2]. For example

```
a = {}
k = "x"
a[k] = 10
a[20] = "great"
print(a["x"])
k = 20
print(a[k])
a["x"] = a["x"] + 1
print(a["x"])
```

Figure 2. An example of how tables work.

In figure 2 the table is declared. Next give k = “x”, after that, we assign a entries in the table a[k] = 10 and a[20] = “great”, keep in mind that a[20] is not an index 20, but rather 20 is key and the value is “great”, when you print a[“x”] you get 10, if you were to set k = 20 and print a[k] you would now get “great”, the last line will print 11.Each reference to a key has a value associated with it, hence key value pairs. The tables contain all info, functions etc, that are needed to mod Dota 2.

2.5 Methods

Remember that a table in Lua is an object, tables have a state, and an identity that is independent from their values [2]. The way the example is set up below the function below essentially acts a method.

```
Account = { balance=0,
            withdraw = function (self, v)
                         self.balance = self.balance - v
                       end
          }

function Account:deposit (v)
  self.balance = self.balance + v
end

Account.deposit(Account, 200.00)
Account:withdraw(100.00)
```

Figure 3. Example of a Method

Figure 3 shows a table called account which creates balance = 0 , and a withdraw function, a function called deposit is then also made on the following line, here we see 2 different ways to call the functions however either can be used. The Account.deposit(Account, 200.00) is using the

self relation (Account), so the function knows what it is being called on. The second call is the same thing, however the use of the : operator passes the self parameter, much like an object calls its own method. When calling methods for the dota 2 mod, the : operator is how we access the various Game API functions.

3 DoTA 2 Code

3.1 Game Development

When DoTA 2 was being developed, it wasn't done completely in Lua but instead C++ was integrated with Lua. The reason DoTA 2 was developed in Lua is because it is a scripting language [6]. This is because scripting languages aren't compliled at as machine code but instead as an instruction. A majority of the game is run by C++ because a scripting language simply can't do it. Game objects like AI bots are written in C++ because if the AI bots were open source this would allow for people to easily modified parts of the game that would make it unfair. The source code for DoTA 2 isn't open to the public. However, it is possible to develop many different game modes with the help of Lua [4].

When developing games, integrating a scripting language not only makes the development easier but also modding is easier. All one really needs in order to modify the Lua code is a text editor, which makes it easy for scripts to be easily modified by the end user [4]. Also, Lua is able to handle core actions like garbage collection by itself which allows for the user to focus more on higher game logic. By using a scripting language, the user has the ability to change in game behaviors without having to recompile the code because Lua code is compiled at runtime [6]. Also this runtime compiling allows for the user to debug easily. A majority of Lua is used for customization, while C++ is used primarily for functionality.

Scripting in Dota 2 is handled by the VBScript virtual machine using the Lua programming language. Lua is launched at run time when DoTA 2 loads the add-ons and manipulates most features of the game. Lua scripts can control the events that happen such as

in-game modes, game rules, abilities, hero interactions, neutrals, AI, and much more [4]. All of DoTA 2 workshop tools for writing scripting addons are handled with Lua. The way that the scripts are written with Lua in DoTA 2 are very similar to those written and coded in other languages, therefore making it easy for someone with a background in a different language to write code in Lua [3].

3.2 Abilities Data-Driven

Most abilities in Dota 2 are defined by the C++ code. These abilities have npc_abilities.txt key values files that contain ability metadata. These pieces of metadata alongside with all the actual behavior specified in the code define how things work in the game. The data-driven ability system is a method to create custom abilities for Dota 2 [4]. These custom abilities allow for the user to assign special properties and events to occur when they're used.

Here is what a simple data-driven ability looks like. This example is a passive ability that applies a visual effect to the unit that has the ability [3].

```
"fx_test_ability"
{
    // General
    //-------------------------------------------------------------------------------------------------------------
    "BaseClass"             "ability_datadriven"
    "AbilityBehavior"       "DOTA_ABILITY_BEHAVIOR_PASSIVE"
    "AbilityTextureName"    "axe_battle_hunger"

    // Modifiers
    //-------------------------------------------------------------------------------------------------------------
    "Modifiers"
    {
        "fx_test_modifier"
        {
            "Passive" "1"
            "OnCreated"
            {
                "AttachEffect"
                {
                    "Target" "CASTER"
                    "EffectName" "particles/econ/generic/generic_buff_1/generic_buff_1.vpcf"
                    "EffectAttachType" "follow_overhead"
                    "EffectLifeDurationScale" "1"
                    "EffectColorA" "255 255 0"
                }
            }
        }
    }
}
```

Figure 4. The ability called fx_test_ability is a passive ability with the texture based off of the hero Axe's ability called battle hunger. The modifiers section holds fx_test_modifier which is specific to this custom ability. When this passive is created , it attaches an effect onto the caster based off generic_buff_1.vpcf with the color yellow overhead the hero [4].

Here is another more complex example of an ability that waits for the owner to die. On death, a thinker is created with an acid pool visual effect and an aura modifier which reduces armor and applies a damage over time effect [4].

```
//=================================================================================================================
// Creature: Acid Spray
//=================================================================================================================
"creature_acid_spray"
{
    // General
    //-----------------------------------------------------------------------------------------------------------------
    "BaseClass"             "ability_datadriven"
    "AbilityBehavior"       "DOTA_ABILITY_BEHAVIOR_AOE | DOTA_ABILITY_BEHAVIOR_PASSIVE"
    "AbilityUnitDamageType" "DAMAGE_TYPE_PHYSICAL"
    "AbilityTextureName"    "alchemist_acid_spray"
    // Casting
    //-----------------------------------------------------------------------------------------------------------------
    "AbilityCastPoint"  "0.2"
    "AbilityCastRange"  "900"
    "OnOwnerDied"
    {
        "CreateThinker"
        {
            "ModifierName" "creature_acid_spray_thinker"
            "Target" "CASTER"
        }
    }
    "Modifiers"
    {
        "creature_acid_spray_thinker"
        {
            "Aura" "create_acid_spray_armor_reduction_aura"
            "Aura_Radius" "%radius"
            "Aura_Teams" "DOTA_UNIT_TARGET_TEAM_ENEMY"
            "Aura_Types" "DOTA_UNIT_TARGET_HERO | DOTA_UNIT_TARGET_CREEP | DOTA_UNIT_TARGET_MECHANICAL"
            "Aura_Flags" "DOTA_UNIT_TARGET_FLAG_MAGIC_IMMUNE_ENEMIES"
            "Duration" "%duration"
            "OnCreated"
            {
                "AttachEffect"
                {
                    "EffectName" "particles/units/heroes/hero_alchemist/alchemist_acid_spray.vpcf"
                    "EffectAttachType" "follow_origin"
                    "Target" "TARGET"
                    "ControlPoints"
                    {
                        "00" "0 0 0"
                        "01" "%radius 1 1"
                    }
                }
            }
        }
        "create_acid_spray_armor_reduction_aura"
        {
            "IsDebuff" "1"
            "IsPurgable" "0"
            "EffectName" "particles/units/heroes/hero_alchemist/alchemist_acid_spray_debuff.vpcf"
            "ThinkInterval" "%tick_rate"
            "OnIntervalThink"
            {
                "Damage"
                {
                    "Type"   "DAMAGE_TYPE_PHYSICAL"
                    "Damage" "%damage"
                    "Target" "TARGET"
                }
            }
            "Properties"
            {
                "MODIFIER_PROPERTY_PHYSICAL_ARMOR_BONUS" "%armor_reduction"
            }
        }
    }
```

Figure 5. The above script does a plethora of things. First, and most importantly, "OnOwnerDied" is triggered on the death of the associated hero. It holds CreateThinker which links the ModifierName to "creature_acid_spray_thinker" via KV pairs. The target of this modifier is the caster, the one that dies. Modifiers holds "creature_acid_spray_thinker" and "creature_acid_spray_armor_reduction_aura" (referred to further as Mod1 and Mod2). Mod1 calls Mod2 in it, sets the aura's radius, and sets a bunch of flags regarding it while also attaching effects to the ability (notice that Target kv is used multiple

times, however, they have completely different meanings to one another). Mod2 sets flags such as IsDebuff and IsPurgable to let the game know that this is a debuff being applied and while it is on a hero, it cannot be purged off. ThinkInterval is the rate at which it inflicts “OnIntervalThink” (and in this case, it happens to be damage). Lastly, it applies a negative modifier to the unit’s physical armor bonus.

3.3 Lua Abilities and Modifiers

Lua is capable of specifying abilities and modifiers entirely in itself. The abilities and modifiers have more advanced logic in their effects. Lua-derived abilities and modifiers behave similarly to their in-game counterparts coded in C++ [5]. These ability and modifier action give the user the choice of overriding those functions in the script. This is where methods are labeled under scriptName:Function and can rely on the self keyword point at its own information.

3.4 Entity Scripts

Entity scripts can be used for adding new functionality or logic to in game objects. These are convenient when scripting in an event driven fashion [5]. Entity scripts are assigned by adding the script file name to an in game object. The scripts are executed when the object spawns, loading into a script scope specific to each object. The script and all the variables and functions can be accessed through the object, which remains available for however long the objects exists for [3].

3.5 Game Events

ListenToGameEvent, when given an event name and function, will call the function given every time the given event happens.

```
function LevelUpMessage (eventInfo)
    Say(nil, "Someone just leveled up!", false)
end

function Activate ()
    ListenToGameEvent("dota_player_gained_level", LevelUpMessage, nil)
end
```

Figure 6. An example - Every time a player levels up, we show a message to all players:

3.6 Script Filters

Script filtering allows the user to make custom games with the ability to get and modify in game actions. Each script passes the value from an event that describes what is going to happen into a table. The user is then able to modify these table values to change in game behaviors [4].

The best way to see what values are in each filter is to print the contents of the table when the filter is called.

Some examples from the API [3] -

- ExecuteOrders - All Player issued and forced actions to get routed through here.
- RuneSpawn - A rune is about to spawn on the given spawner.
- Damage - Damage is about to be applied.
- ModifyGold - A player is having their gold value modified for the specified reason.
- ModifyExperience - A player is having their gold value modified for the specified reason.

3.7 Our Style of Lua

There is a lot of information floating around about the current way to code in Lua in relation to DoTA2. The majority of which seems to recommend the data-driven system. This however, is old data from right before and after the reborn client came out (2015). Although it is still entirely possible to mod with data-driven scripting, it is dying out as it seems to becoming more and more depreciated. From speaking to individuals within the DoTA2 modding community in a public chat, it was learned that the best/most widely used way is through lua-abilities. Unfortunately, this was learned too late in the process resulting in the main item being data-driven, but the saving grace is that each method in the data-driven model calls a lua script making our work somewhere in between. The key issue with this being that the self pointer must be passed through the method, and a colon (:) cannot reference itself. Figure 7 gives a good idea on why a mixed Lua style would not be appropriate for future code.

```lua
--[[Hey, you. You're finally awake. You were trying to code the project the night
before, right? Walked right into that Volvo ambush, same as us, and that neckbeard over there.
But seriously, drops and item on death and gives it a charge.]]
function DropItemOnDeath(keys) -- keys is the information sent by the ability
    print( '[item_dodgeball] DropItemOnDeath Called' )
    local killedUnit = EntIndexToHScript( keys.caster_entindex ) -- EntIndexToHScript takes the
    local itemName = tostring(keys.ability:GetAbilityName()) -- In order to drop only the item
    if killedUnit:IsHero() or killedUnit:HasInventory() then -- In order to make sure that the
        Say(nil, ("Goodbye: "  .. keys.ability:GetInitialCharges()), false)
        for itemSlot = 0, 5, 1 do --a For loop is needed to loop through each slot and check if
            if killedUnit ~= nil then --checks to make sure the killed unit is not nonexistent.
                local Item = killedUnit:GetItemInSlot( itemSlot ) -- uses a variable which gets
                if Item ~= nil and Item:GetName() == itemName then -- makes sure that the item
                    local newItem = CreateItem(itemName, nil, nil) -- creates a new variable wh
                    CreateItemOnPositionSync(killedUnit:GetOrigin(), newItem) -- takes the newI
                    newItem:SetCurrentCharges(1)
                    killedUnit:RemoveItem(Item) -- finally, the item is removed from the origin
                end
            end
        end
    end

end
```

Figure 7. This shows how our model uses keys as a reference to self, as opposed to fileName:DropItemOnDeath. This method is originally called when the method OnOwnerDied of the item is ran. It can be compared against newItem or killedUnit which are of class item or entity respectively. They are able to call methods using a colon as it is their class type method being called. For our keys, we must reference keys.ability which is our item, as opposed to self:GetAbilityName

3.8 Our project - Dota Dodgeball

Our project is a gamemode under the guise of Dodgeball, but it still keeps the DoTA2 flair. We use a custom made dodgeball item to throw dodgeballs around. The associated ability uses another hero's particles to act as our dodgeball. This ability creates a linear projectile associated with these particles. In a normal game of dodgeball, when you throw a ball it is gone and it is more than likely on the other side of the court. The same is true for our gamemode. If the ball hits an enemy entity, it stops on sight. This player is out (killed and forced to respawn) and the ball is removed from the throwing player's inventory, spawning a new item under the targets previous position. If the ball misses and does not come into contact with anyone, it is still removed from the thrower's inventory, but the ball is placed in the centermost of the opposing team's side. This is just due to convenience, instead of having to create an absurd amount of logic. Each player in the game spawns in with one dodgeball. This means that a game with 2 players will have 2 dodgeballs, while one with 10 players would have 10 dodgeballs. Currently, the way to win is by getting 5 kills in total on one team. What makes this a little different than a

normal game of dodgeball is the ability to pick a hero with a variation of abilities. Currently there is one hero available (puck) out of the 119 present in the game. Our hero puck, has one ability available called phase shift. It is the default ability in which he disappears from the map and is untargetable/unhittable but with a modified, lower cooldown. Figure 8. Is a link to a video demonstrating these features.

Figure 8. https://youtu.be/PbI85pVhHno

3.9 Future plans

Needless to say, our project was an ambitious one for the relatively short amount of time we had to work on it (only a few weeks). Since this specific paper was dedicated more towards Lua than it was to the actual mod we created, the following sections will go into each separately.

3.9.1 Lua work

Since this language is targeted to be a language for scripting, the future could be a lot of different things. To start, a lot of other games have the ability to be modded with this language, which could be interesting to see the contrasts of how smooth it is to work in these different games. A thing to keep in mind is that although outdated, the majority of DoTA2 functions are listed as an API on a dev wiki for public use. Something like the game Binding of Issac did not have original mod support, and it's "API" is just a list of names of methods to call. Although Lua is designated as a game scripting language, it has practical uses as well. Since there is such a wide variety of functionality to Lua, things like web scraping are possible through packages such as LuaSec and LuaSoc. We could explore the interaction of games and outside programs working in conjunction as another avenue for potential research.

3.9.2 Expanding this mod/ DoTA2 modding in general

Since there was a time restriction, once the main understanding of Lua and DoTA2 happened, the project was close to being over. Given more time, we could expand it in a number of ways. First, we could change the particles to make it look like an actual dodgeball. Next, we could add more heroes and abilities in this mode to make it have a more DoTA2 feel to it. We could also update the map, putting entities around the playfield to expand where the ball could

go when missing a target. This would allow for a more dodgeball feel to the game mode, and not one that is restricted by the client (i.e instead of spawning the item in the center of the opposing players field of play on a miss making the gameplay feel stiff).