Zach Giannuzzi, Zac Goodsell, Tyler Gorman
Rough Draft - Modding DoTA 2 with Lua
Note - Still a work in progress Last 2 pages are ideas that might be incorporated and citations for now not cited correctly just added for fillers to be done correctly later. After all scripting is completed, we will grab in game captures of example code.

Background

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(Scripting API).Lua has been used in applications such as Adobes photoshop lightroom and is currently "The leading scripting language in games"(Lua.org) such as World of Warcraft and Angry Birds. The research and learning of Lua will be done by modding the video game Dota 2. Many of Lua's features and how Lua works with Dota 2 will be explored, specifically methods, key-value pairs, abilities, scripts, game events and others.

Key Value Pairs/Tables

Key value pairs and tables are very important in understanding Luas 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 (). 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"])
```

Ex. In this example 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.

Methods

Remember that a table Lua is an object, tables have a state , as an identity that is independent from their values(Lua.org). 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)
```

Ex. Here a table called account is created with a 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 so the function knows what it is being called on. The second call is the same thing however the use of : operator which hides the self parameter.

When calling methods for dota 2 mod the : operator is how we access the various Game API functions.

DoTA 2 Code

When DoTA 2 was being developed, it wasn't done completely in Lua but instead Lua was integrated with C++. The reason DoTA 2 was developed in Lua is because it is a scripting language (quora). 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 (valve).

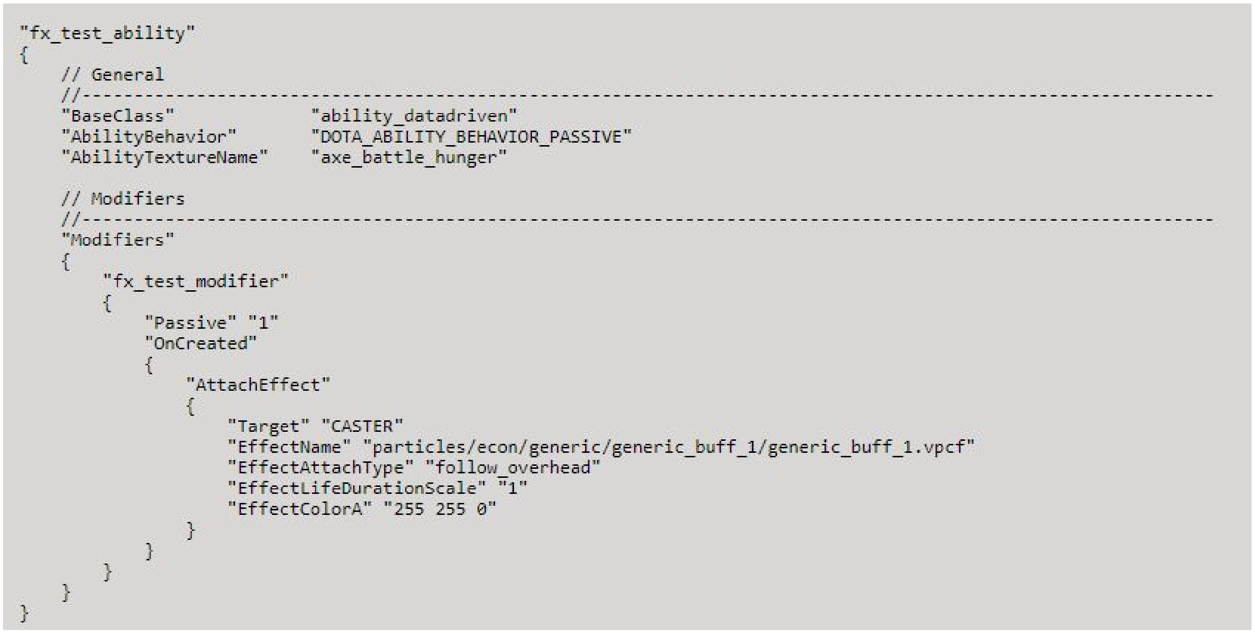
When developing games, integrating a scripting language not only makes the development easier but also modding is easier. All one really needs in order to modify the Lua code is a text editor, which makes it easy for scripts to be easily modified by the end user (valve). Also, Lua is able to handle core 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 (quora). 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 in-game modes, game rules, abilities, hero interactions, neutrals, AI, and many more (valve). 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 (valve).

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 (valve). 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 (valve).

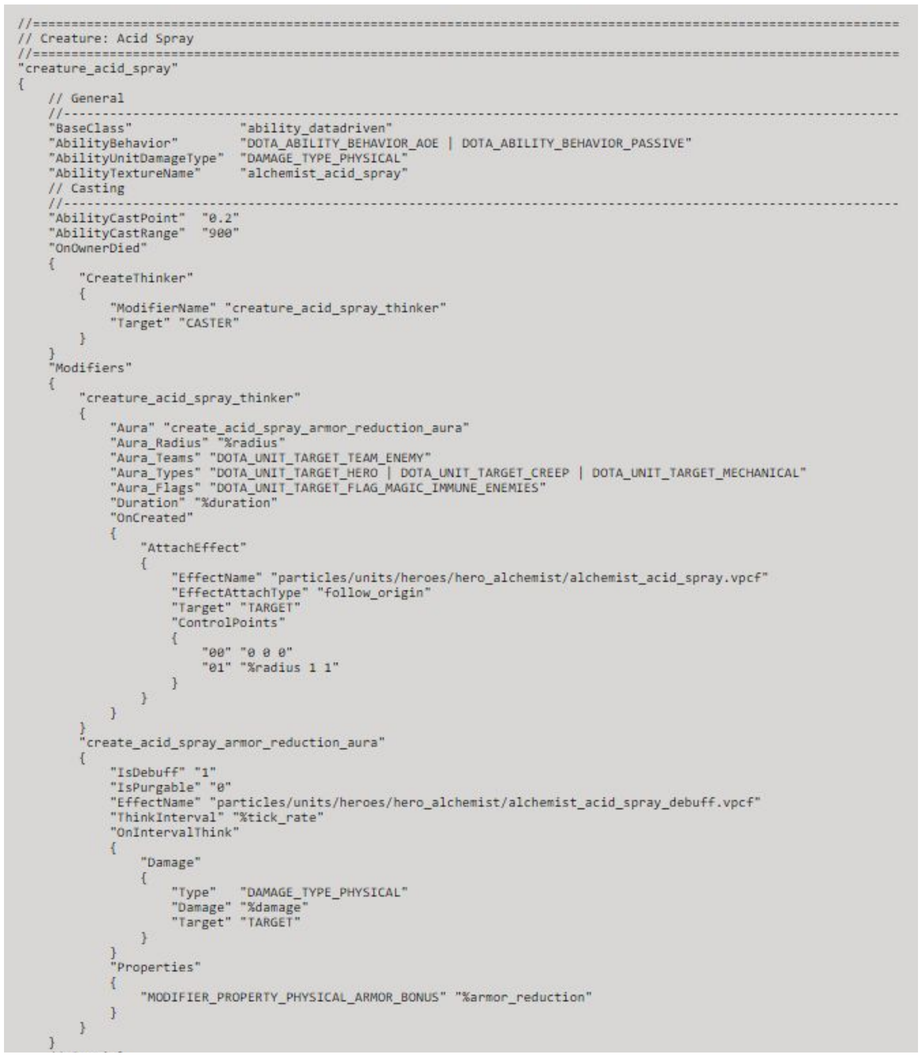


```
"fx_test_ability"
{
    // General
    //-------------------------------------------------------------------------------------------------------------
    "BaseClass"              "ability_datadriven"
    "AbilityBehavior"        "DOTA_ABILITY_BEHAVIOR_PASSIVE"
    "AbilityTextureName"     "axe_battle_hunger"

    // 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"
                }
            }
        }
    }
}
```

EX. 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 (valve).

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 (valve).



```
//=========================================================================================================
// 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"
            }
        }
    }
```

EX. 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, aka 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 units physical armor bonus.

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++ (mod). These ability and modifier action give the user the choice of overriding those functions in the script.

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 (mod). 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 exists. The script and all the variables and functions can be accessed through the object, which remains available for however long the objects exists for (valve).

Game Events

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

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

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

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

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 (Valve).

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 (Valve API) -

- 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.

Extra Stuff

Ideas-

- Background of Lua
- How it's a scripting language
- How does it interact with everything?
- **Different add on folders?** → The way the channel derives data will be gone over, relatively important in understanding the modding aspect of Lua
- Scripting uses methods
- Dota coded in C
- API's
- Mod dota site
- Intro to modding games. With scripting?
- **Map editor aka HAMMER → small section will be added**

Sources -
https://developer.valvesoftware.com/wiki/Dota_2_Workshop_Tools/Scripting/API
https://developer.valvesoftware.com/wiki/Dota_2_Workshop_Tools/Scripting
https://www.quora.com/Why-did-Valve-choose-Lua-instead-of-C++-to-develop-Dota-2
http://www.lua.org/pil/contents.html

DOTA working w/ LUA

- After addon_game_mode Precache & Activate are finished, the first function to be executed in the barebones.lua file is GameMode:InitGameMode().
    - In here the game starts by initializing all sorts of rules and functions, which are registered over the GameRules and GameMode entities. For this, many variables are defined on top of the file to help organize options like gold settings, kills, custom levels, etc.
- Just as KV, Lua is Case Sensitive. Also, the placement of the functions within your main Lua file doesn't generally matter. All the script lines within a function call will be run one after another, potentially on the same *frame*; one frame in Dota is 1/30 of a second.
- Dynamic_Wrap is a function to ensure that the script_reload command also reloads the listeners. script_reload restarts Lua scripts at runtime, unlike DataDriven files which require the game to be fully restarted.
    - EX -> **ListenToGameEvent**('dota_player_gained_level', **Dynamic_Wrap**(GameMode, 'OnPlayerLevelUp'), self)
- DeepPrintTable is a Global Valve-made function which will display the information of the table passed

- There is one more thing to consider: the “wait one frame” issue. Because all units are actually spawned at the (0,0,0) coordinates and then moved to the desired position, **many times you’ll need to create a 0.03-second timer (1 frame) for some scripts to work**, and this is one of those cases.

Dota Variables

  - **.caster**, the entity that started the ability.
  - **.target**, the target of the ability (can be the same as the caster in some cases)

Lua Specific

- Calling a method -> Class:function(variables)
  - Note the use of: colon before the function. In Lua, this is how we access the various **Game API functions**.
  - We say that GameRules is an **HScript** or a **handle**. Handles are basically huge tables, with all the pertinent info of the entity. Over the Scripting API page you’ll see many different types of functions which can use different handles
  - The 3rd and last main element of InitGameMode are self-defined variables to track info. These use the self. entity, which is a local reference to the GameMode entity, seen through all the functions inside the main Lua file. Adding information to an entity like entity. is loosely called “indexing” and is basically adding another entry to the big table of that entity. <- Kind of mixed between dota and Lua, difference being its generic enough that multiple games use this type of call back
- Concatenating strings in Lua is “string1” **..** “String2”
- **Moreover, access to local variables is faster than global ones.**
- Values work like **table.table.variable**

- Key-Value pairs are used generally in Lua
  - for key, unit in pairs(units) do
  -    print(key,value)
  -    unit:ForceKill(true)
  - End

    - The key, unit are the chosen names to refer to the position and value inside the *units* table, which will be read in pairs. Using ‘_’ as the name of the key is a good convention when you want to make it clear that the first parameter won’t be used. The 2nd parameter, unit, is used to itea handles of the units found.
- Comment is --[[text goes here]]