Scripting

IMGD 4000


Outline

• Scripted behavior
• Scripted language
• Lua
• Blueprints

Scripting

• Two senses of the word in games*
  – “scripted behavior”
    • Having NPCs follow pre-set actions
    • rather than choosing them dynamically
  – “scripting language”
    • Using an interpreted language
    • to make the game easier to modify
• The two are related:
  – A scripting language is good for writing scripted behaviors (among other things)

* also “shell scripts”, which are not today’s topic

Scripted Behavior

• One way of building NPC behaviors
• Other way is simulation-based behavior
  – e.g., goal/behavior trees
  – genetic algorithms
  – machine learning
  – etc.
Scripted vs. Simulation-Based Behavior

• Example of scripted behavior (in combat game)
  – Fixed trigger regions
    • When player/enemy enters predefined area
    • Send pre-specified waiting units to attack
  – Doesn’t truly simulate scouting and preparedness
  – Easily found “exploit”
    • e.g., Mass outnumbering force just outside trigger area
    • Attack all at once

Non-scripted (“simulation-based”) version?
• Send out patrols
• Use reconnaissance information to influence unit allocation
• Adapt to player’s behavior (e.g., massing of forces)
• Can even vary patrol depth depending on stage of game

Advantages of Scripted Behavior

• Much faster to execute
  – Apply simple rule versus run complex simulation

• Easier to write, understand and modify
  – Than sophisticated simulation

Disadvantages of Scripted Behavior

• Limits player creativity
  – Players try things that “should” work (based on their own real-world intuitions)
  – Disappointed when they don’t

• Allows degenerate strategies
  – Players learn limits of scripts
  – and exploit them

• Games need many scripts
  – Predicting their interactions can be difficult
  → Complex debugging problem
Stage Direction Scripts

- Controlling *camera movement* and “bit players”
  - Create guard at castle drawbridge
  - Lock camera on guard
  - Move guard toward player
  - and so on
- Better application of scripted behavior
  – Doesn’t limit player creativity as much
  – Improves visual experience

Outline

- Scripted behavior (done)
- Scripted language (next)
- Lua
- Blueprints

Scripting Languages

*You can probably name a bunch of them...*

Two flavors:

1. Custom languages tied to specific games/engines
   - UnrealScript, Blueprints, QuakeC, HaloScript, Linden Script (LSL), ...
2. General purpose languages
   - Tcl, Python, Perl, Javascript, Ruby, Lua, ...
   - “Modern” trend, especially with Lua

*Often used to write scripted behaviors*

Custom Scripting Languages

- Custom scripting language tied to specific game, which is just idiosyncratically “different” (e.g., QuakeC) doesn’t have much to recommend it
- However, game-specific scripting language that is *truly natural* for non-programmers can be very effective:
Custom Languages and Tools

“Designer UI” from Halo 3 (Microsoft, 2007)

General Purpose Scripting Languages

What makes a general purpose scripting language different from any other programming language?

- Interpreted (byte code, virtual machine)
  - Technically, property of implementation (not language per se)
  - Faster development cycle
  - Safely executable in “sandbox”
  - Recently JIT native compilation also
- Simpler syntax/semantics
  - Untyped
  - Garbage-collected
  - Built-in associative data structures
- “Plays well” with other languages
  - e.g., LiveConnect (Java/javascript with Web browser), .NET (Microsoft framework for interoperability), Lua stack (integrates with many languages)

General Purpose Scripting Languages

But when all is said and done, it looks pretty much like “code” to me....☺

e.g., Lua

```lua
function factorial(n)
  if n == 0 then
    return 1
  end
  return n * factorial(n - 1)
end
```

So it must be about something else...

Now go back...

To the world of C++ engines....
Scripting Languages in Games

So it must be about something else...
Namely, the game development process

• For technical staff
  – Data-driven design (scripts viewed more as “data,” not part of codebase)
  – Script changes do not require game recompilation

• For non-technical staff
  – Allows parallel development by designers
  – Allows end-user extension

A Divide-and-Conquer Strategy

• Implement part of game in C++
  – The time-critical inner loops
  – Code you don’t change very often
  – Requires complete (often very long) rebuild for each change

• and part in scripting language
  – Don’t have to rebuild C++ part when change scripts
  – Code you want to evolve quickly (e.g., NPC behaviors)
  – Code you want to share (with designers, players)
  – Code that is not time-critical (can migrate to C++ later)

General Purpose Scripting Languages

But to make this work, you need to successfully address a number of issues

• Where to put boundaries (APIs) between scripted and “hard-coded” parts of game
• Performance
• Flexible and powerful debugging tools
  – even more necessary than with some conventional (e.g., typed) languages
• Is it really easy enough to use for designers!?*

A Note About Performance

• Scripting languages with game engine can often take performance hit
  – e.g., UE4 blueprints take performance hit about 10x over C++
  [Billy Bramer 2014, Epic Staff and UE4 developer]

• But this performance hit is often not an issue
  – Remember, working code more important than high performance code!
  – Performance only matters if it impacts playability/quality

• There is big difference between "slower than C++" and "too slow to be used"
Guidelines for Scripting vs. C++ (1 of 2)

• For engines that support both (e.g., UE4), generally
  – Content creators and game designers use script
  – Engineers use C++, but sometimes script depending upon feature
• For performance, vast majority of game programming cases, scripting fine
  – Level of comfort (or interest learning) can help decide
• Do “heavy lifting” in C++
  – A few nodes call back into C++ functionality no problem
• Just like don’t put performance heavy code inside Object
  tick/update function, don’t put performance heavy code in script
  – e.g., computing A* for large map each tick
• With engine, parts of game that are performance critical in C++, so don’t worry about script performance
  – e.g., game loop and collision detection already C++
• Scripting often good for triggered events
  – e.g., triggered when event happens, like taking damage

Guidelines for Scripting vs. C++ (2 of 2)

• Engineers intentionally expose custom C++ methods and variables for content creators (e.g., artists and designers) to use to
  a) Speed up content creators workflow, without them dealing with nitty-gritty details
  b) Protect content creators against intricacies of feature that might be error-prone or confusing
    • (e.g., content creator doesn’t care about math involved with placing building on unlevel terrain)
  c) Expose properties so content creator can just adjust and view
  d) Provide method for operation content creator repeats often, thus doing operation “behind the scenes”

Outline

• Scripted behavior (done)
• Scripted language (done)
• Lua (next)
• Blueprints

Most Popular Game Scripting Language?

• Lua
• Has come to dominate other choices
  – Powerful and fast
  – Lightweight and simple
  – Portable and free
• See http://www.lua.org/
Lua Language Data Types

- **Nil** – singleton default value, nil
- **Number** – internally double (no int’s!)
- **String** – array of 8-bit characters
- **Boolean** – true, false
  - Note: every non-boolean except nil coerced to true, e.g., “”, 0 are true
- **Function** – unnamed objects
  - Can be stored, passed as arguments, returned as results
- **Table** – key/value mapping (any mix of types)
  - Used heavily when manipulating data
- **UserData** – opaque wrapper for other languages
- **Thread** – multi-threaded programming (reentrant code)

“Promiscuous” Syntax and Semantics

- **Optional** semi-colons and parens
  - A = 10; B = 20;
  - A = 10 B = 20
  - A = foo();
  - A = foo
- **Ignores** too few or too many values
  - A, B, C = 1, 2, 3
  - A, B, C = 1, 2, 3, 4
- **Uptake?** Flexible and forgiving, but can lead to a debugging **nightmare!**
- **Moral:** Only use for small procedures

Lua Variables and Assignment

- **Untyped:** any variable can hold any type of value at any time
  - A = 3;
  - A = “hello”;
- **Multiple values**
  - In assignment statements
    - A, B, C = 1, 2, 3;
  - Multiple return values from functions
    - A, B, C = foo();
Lua Operators

- **arithmetic:** +  -  *  /  ^
- **relational:** <  >  <=  >=  ==  ~=
- **logical:** and  or  not
- **concatenation:** ..

... *with usual precedence*

Lua Tables

- **Heterogeneous** associative mappings
- Used a lot
- Standard array-ish syntax
  - Except any object (not just int) can be “index” (key)
    mytable[17] = “hello”; mytable[“chuck”] = false;
  - Curly-bracket constructor
    mytable = { 17 = “hello”, “chuck” = false };
  - Default integer index constructor (starts at 1)
    test_table = { 12, “goodbye”, true };
    test_table = { 1 = 12, 2 = “goodbye”, 3 = true };

Lua Control Structures

- Standard *if-then-else*, *while*, *repeat* and *for*
  - with break in looping constructs
- Special *for-in* iterator for tables
  
  ```
  data = { a=1, b=2, c=3 };
  for k,v in data do print(v..“ “..k) end;
  ```
  produces, e.g.,
  ```
  a 1
  c 3
  b 2
  ```
  (order undefined)

Lua Functions

- Standard parameter and return value syntax
  ```
  function (a, b)
    return a+b
  end
  ```
  - Inherently unnamed, but can assign to variables
    ```
    foo = function (a, b) return a+b; end
    foo(3, 5) \rightarrow 8
    ```
Other Lua Features ...

- Object-oriented style (alternative dot/colon syntax)
- Local variables (default global)
- Libraries (sorting, matching, etc.)
- Namespace management (using tables)
- Multi-threading (thread type)
- Bytecode, virtual machine
- Some features primarily used for language extension

See [http://www.lua.org/manual/5.3](http://www.lua.org/manual/5.3)

Outline

- Scripted behavior  (done)
- Scripted language  (done)
- Lua  (done)
- Blueprints  (next)

So what’s all this got to do with UE4?

- Game engine core of UE4 is coded in C++ ...
- UE4 provides blueprints “scripting languages”  
  – Also some Macros that can act like scripts
- So this is the divide-and-conquer paradigm we discussed!

Scripting and Unreal Engine

UE1 and UE2
- Designed for First Person Shooters (FPS)
- UnrealScript game scripting language

UE3
- Kismet visual scripting added
- More modular game classes
- But still very FPS centric

UE4
- UnrealScript replaced with Blueprints
- Game genre agnostic
- Lots of sample projects!
UnrealScript vs. C++ vs. Blueprints

UnrealScript was:
- An object-oriented scripting language
- Similar in syntax to C, C++, Java, but also somewhat different
- Compiled to virtual machine byte code
- Added features, such as States, Timers, Delegates

Blueprints are:
- Visual scripting designed to be artist and designer friendly
- Using same virtual machine as UnrealScript
- Almost as powerful as UnrealScript, in some ways better
- Extensible by users with features

C++ has:
- Always been part of UE game programming
- Tight bi-directional integrations with virtual machine
- Been extended with added macros in UE4 to replace UnrealScript for coders

What are Blueprints?

- Blueprints are scripting language built on top of C++
  - Behind scenes, calling existing C++ methods
  - Used to automate existing functionality already coded in C++
- Blueprint class (often shortened to blueprint)
  - Created inside Unreal Editor visually (e.g., contrast to typing code in VS)
  - Define new class or Actor can be placed into maps as instances that behave like any other type of Actor.
- Thus, scripting that can be extended by game programmer

UE4 Blueprint Example

Player presses "dash" and if character can jump, launch character based on character's current velocity with an upward (z) velocity.

Created in Blueprint Editor
Blueprint Types (1 of 2)
• Two types of blueprints — level and class
  • Level blueprint — one per level
    — Reference and manipulate Actors within level
    — Control cinematics (using Matinee Actors)
    — Manage level-related resources (e.g., checkpoints)
    — Interact with blueprint classes in level (e.g., reading/setting variables, triggering events)
• Blueprint class (next slide)

Blueprint Types (2 of 2)
• Blueprint class — interactive assets (e.g., doors, switches, destructible scenery)
  — Contain necessary scripts to respond to player interactions (collisions)
  — (e.g., make them animate, change materials, play sound effects)
  — Broadly, these are “game objects”
• Prefabs (scripts to pre-fabricate game world)
• Pawns (e.g., player characters)
• HUD (heads-up display, such as a status bar)
• As a sub-set, a “data-only” blueprint contains only node graphs and variables from parent.
  — Cannot add new attributes.
  — Allows tweaking of attributes

Blueprints Compared to C++
• Blueprints
• Blueprint Asset
• UBlueprintGeneratedClass
• ParentClass
• Variable
• Graphs/Events
• Class Defaults
• Components List
  • C++
  • .h/.cpp files
  • UClass
  • : [ClassName]
  • UProperty()
  • UFunction()
  • native constructor
  • native constructor

Examples of Blueprints
1. Create custom prefab with construction script (good for artists/designers)
   — Executes when Actor placed in editor (not gameplay)
   — Useful for custom props (e.g., light fixture that matches material to color, e.g., randomly scatter leaves)
   — May be time consuming to create initially, but may save time later
2. Create playable character
   — Manipulate camera behavior, setup input vents (e.g., mouse, keyboard), create Animation Blueprint
   — Character Blueprint has moving, jumping, swimming and falling built-in
   — Only need to add input events
3. Create HUD
   — Contains event sequence and variables, but is assigned to GameMode asset (not level)
   — Read from other blueprints then display (e.g., read health and display health bar)
   — Add hit-boxes for buttons
Create 3rd Person Game Using Just Blueprints

https://wiki.unrealengine.com/Blueprint_3rd_Person_Game_Creation_Tutorial_Playlist

See also, the Blueprint Essentials Tutorial Playlist:
https://wiki.unrealengine.com/Blueprint_Essentials_Tutorial_Playlist

But Lua cannot stand alone...

• Why not?

• Accessing Lua from C++
• Accessing C++ from Lua

Connecting Lua and C++

• Lua virtual stack
  – bidirectional API/buffer between two environments
  – preserves garbage collection safety

• data wrappers
  – UserData – Lua wrapper for C data
  – luabind::object – C wrapper for Lua data
Lua Virtual Stack

- both C and Lua env’ts can put items on and take items off stack
- push/pop or direct indexing
- positive or negative indices
- current top index (usually 0)

Accessing Lua from C

C

Lua

Accessing Lua Global Variables from C

- C tells Lua to push global value onto stack
  ```
  lua_getglobal(pLua, "foo");
  ```
- C retrieves value from stack
  - using appropriate function for expected type
    ```
    string s = lua_tostring(pLua, 1);
    ```
  - or can check for type
    ```
    if ( lua_isnumber(pLua, 1) )
    { int n = (int) lua_tonumber(pLua, 1) } ...
    ```
- C clears value from stack
  ```
  lua_pop(pLua, 1);
  ```

Accessing Lua Tables from C (w. LuaBind)

- C asks Lua for global values table
  ```
  luabind::object global_table = globals(pLua);
  ```
- C accesses global table using overloaded [] syntax
  ```
  luabind::object tab = global_table["mytable"];
  ```
- C accesses any table using overloaded [] syntax and casting
  ```
  int val = luabind::object_cast<int>(tab["key"]);
  ```
  ```
  tab[17] = "shazzam";
  ```
Calling Lua Functions from C (w. LuaBind)

- C asks Lua for global values table
  
  ```
  luabind::object global_table = globals(pLua);
  ```

- C accesses global table using overloaded [ ] syntax
  
  ```
  luabind::object func = global_table["myfunc"];  
  ```

- C calls function using overloaded ( ) syntax
  
  ```
  int val =
    luabind::object_cast<int>(func(2, "hello"));
  ```

Accessing C from Lua

- C exposes function to Lua
  ```
  void MyFunc (int a, int b) { ... }
  module(pLua) [
    def("MyFunc", &MyFunc)
  ];
  ```

- Lua calls function normally in scripts
  ```
  MyFunc(3, 4);
  ```  
  [See more details and examples in Buckland, Ch 6.]

Calling C Function from Lua (w. LuaBind)

- C "exposes" function to Lua
  ```
  void MyFunc (int a, int b) { ... }
  module(pLua) [
    def("MyFunc", &MyFunc)
  ];
  ```

- Lua calls function normally in scripts
  ```
  MyFunc(3, 4);
  ```  
  [See more details and examples in Buckland, Ch 6.]