Lua and Fable

Jonathan Shaw

How does Fable use Lua?

- AI
- Quest scripts
- Camera
- Start-up settings
- Miscellaneous scripts
- Debugging
- Patching

A (very) brief history

- The original Fable used C++ as its “scripting” language, using macros to make it more “user friendly”.
- We used micro-threads (or fibres) to give each AI entity and each quest its own non-pre-emptive thread.
- We liked micro-threads, but didn’t like C++ as a scripting language.

Enter Lua

- Like all good scripting languages Lua is easy to write, and can give you almost-instant feedback when changing scripts while your game is running.
- Lua's first class support for coroutines made Lua an ideal choice for Fable.
- Fable II and III make *extensive* use of coroutines.
  - On busy levels we can have upwards of 100 coroutines running.

AI

- An AI entity in Fable II and III has a brain, defined in Fable's general game editor.
- A brain has a list of behaviour groups with priorities (some of which can be the same).
- A behaviour group has a list of behaviours with priorities (some of which can be the same).

Deciding what behaviour to run

- Iterate through the behaviour groups, starting with the highest priority – see if they're valid to run.
  - E.g. a "work in the factory" behaviour group isn't valid to run if it's night time.
- For each valid behaviour group, iterate through the behaviours, starting with the highest priority, to see if they're valid to run.

Same priority behaviours

- If some behaviours or behaviour groups have the same priority, we want to iterate through them in a random order.
- Custom iterators in Lua were the perfect solution for this design.
- Sadly, very nice to write, but a performance bottleneck, so we moved the logic to C++.

Quests

- A quest will have one “master” coroutine, with a number of “child” coroutines – when the master coroutine is finished we automatically kill the child coroutines.
- We have the concept of “entity threads” – these are automatically tied to specific entities and die when the entity dies (or suspend when the entity is unloaded when the player changes level).

Saving quests

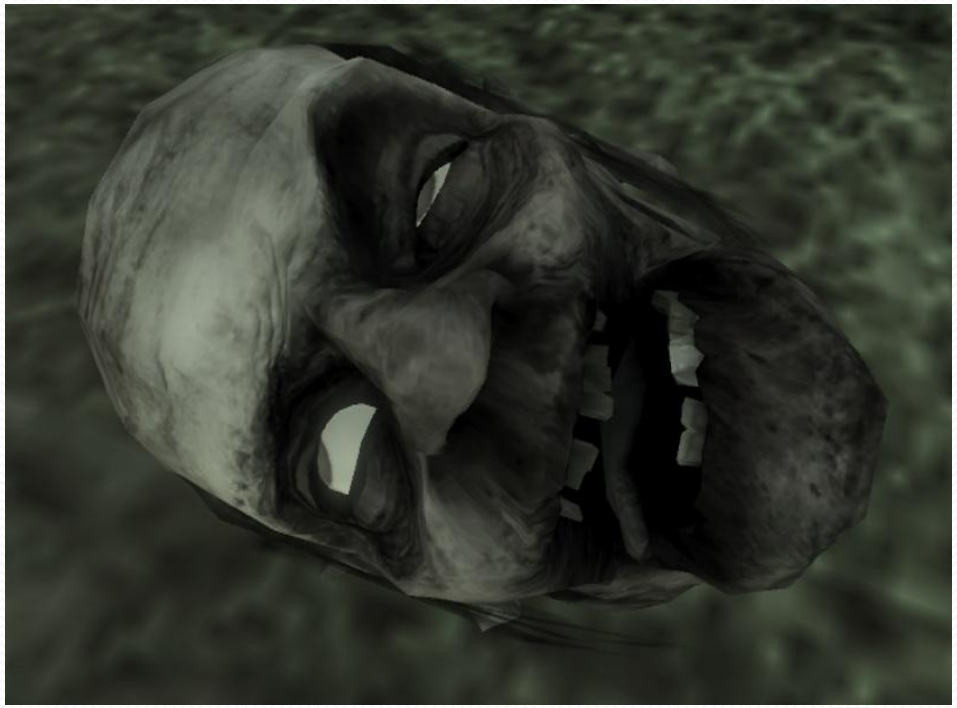
- We wanted players to be able to save the game wherever they were in the game and for it to be a "perfect save" – *without quest scripters having to do anything to make it work.*
- Quite a challenge, until we learned about Pluto for Lua, by Ben Sunshine-Hill.

The patch challenge

- Using Pluto allows you to save coroutines and to resume them where they left off after loading the save game.
- What if you want to patch a function that a coroutine was yielded in?
  - If you change the line numbering (of the resultant Lua byte code) before the final yield in the function, you can't.

Nasty tricks

- This forces you to come up with some “creative” solutions.



Making it a little easier

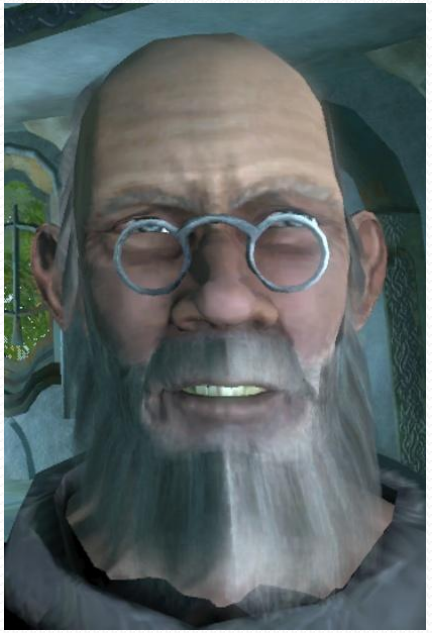
- In Fable III some of our scripters have started changing how they format their while loops:

```
while true do
  -- Check for various
  -- conditions and act
  -- on them
  coroutine.yield()
end
```

```
while true do
  coroutine.yield()
  -- Check for various
  -- conditions and act
  -- on them
end
```

Bring on the Watch Dogs

- Lua is extremely flexible and inspectable.
- Add scripts to run in their own coroutines and inspect the state of the known problem scripts – when they detect the problem, fix it and shut down.



Lua/C++ interface

- We wanted it to be as easy as possible to register C++ functions in Lua.
- The more functions exposed to Lua, the more power you have when it comes to debugging and releasing DLC without requiring a Title Update.
  - There were features in Fable II's second DLC that we had to drop because the one function the scripts needed hadn't been exposed.

Lua Plus Call Dispatcher

- We use Joshua Jensen's Lua Plus Call Dispatcher.
- This lets you expose a function with any arbitrary signature to Lua (it doesn't have to be a `lua_Cfunction`).
- You can describe to the call dispatcher how to interpret any type you might want to expose to Lua.

Hiding the Lua stack

- For a C++ programmer used to dealing with objects, it can be useful to be able to treat Lua tables and functions as C++ objects.
- Using inspiration from LuaPlus we wrote CLuaTable and CLuaFunction classes.
- These use the Lua registry to keep handles to the table and functions alive even if no handles to them in Lua remain.

A quick example

```
class CEntity; class Vec3;
Vec3 CEntity::GetPos() const;

luaL_newmetatable(L, “EntityMetaTableName”);
lua_pushvalue(L, -1); // Pushes the metatable
lua_setfield(L, -2, "__index"); // mt.__index = mt

CLuaTable entity_mt (L); // This pops the metatable
  off the stack
entity_mt.RegisterDirectMemberFunction<CEntity>(
  “GetPos", &CEntity::GetPos);
```

Using that in Lua

```
local hero = GetHero()
local trigger = GetEntity("Trig1")

while DistanceBetweenEntities(hero, trigger) > 10 do
        coroutine.yield()
end

-- The hero is now within 10 units of the trigger
```

Debugging Lua

- Lua provides debugging hooks to allow the creation of a debugger application without needing to modify any of the Lua source code.
- Our Lua Debugger supported breakpoints, step into, step over, per-coroutine breakpoints, a watch window and the ability to break into Lua from a C++ assert.
- Saving a file in the debugger would automatically reload it in the game.

Debugging *with* Lua

- Our in-game debug console ran in the game's Lua environment.
- This meant everything scripts or AI could do, you could do in the console (including inspecting the current state of the AI and quests).
- On Fable III we have remote access to this same console (including niceties like auto-complete).

Automation

- Having remote access to the Lua environment means we can very easily write external scripts (ours are in Python) to drive the game however we like.
- We change start-up settings (so Lua is one of the first systems in the game we initialise) and test that every level in the game loads successfully, and get performance metrics from the game using this system.

Profiling Lua

- This is something we struggled with, partly due to our heavy reliance on coroutines.
  - Profilers that hooked into function entries and exits wouldn't stop the timer during yields, giving very skewed results.
- For Fable II, we didn't have anything more sophisticated than manually timing suspected expensive functions and printing times to the debug output channel.

Memory usage

- When Fable II shipped Lua was using roughly 8 to 15Mb of memory.
- We had a custom allocator designed for many small, fragmenting allocations.
- Tracking memory “leaks” in Lua was extremely difficult.
- We found that pre-compiling our Lua scripts massively reduced fragmentation (in addition to the small improvement in loading time).

Reducing memory footprint

- Using Lua 5.1's module system made demand-loading quest scripts very simple.
- Changing lua_Number to float from double saved us 1 or 2Mb, because those extra 4 bytes exist in every Lua type, whether table, Boolean, function, etc.
- The problem with floats comes when you want to pass 32 bit hashes or UIDs to and from Lua (because integers above 8 million lose precision when converted to floats).
  - Light user data was the solution.

Taming the garbage collector

- We turn the garbage collector off and manually run it once per game frame.
- We played around with the step size to try to balance the time the collector takes and the collector having enough time to clean up the general memory churn in any given frame.
- Even when we found a number we were reasonably happy with, the collector would sit at about 3ms, often spiking over 6ms.

Smoothing the spikes

- One of our colleagues from Turn 10 gave us the solution.
- From this:

```
lua_gc(L, LUA_GCSTEP, 120);
```

- To this:

```
double start_time = LHTiming::GetTime();
double end_time = start_time +
  g_SecondsForGarbageCollection; // 2ms
do
{
  lua_gc(L, LUA_GCSTEP, 0);
}
while(LHTiming::GetTime() < end_time);
```

Lua and Fable III

- We were very happy with our use of Lua in Fable II, so haven't really changed that much with Lua in Fable III.
- Our quest scripts now support mid-quest "skip points" so that when debugging a quest you can now skip (forwards or backwards) to a decent number of points in the middle of quests.
- We're using Kore in Fable III, which gives us improved performance and improved tools, including a profiler.

Conclusion

- Lua lets us develop systems quickly with fast iteration.
- Performance is an issue and can be too much of a “black box”.
- Having a rich and full interface between your game and scripting language will serve you well when it comes to extending the game.

References

- Lua:
  - http://www.lua.org
- Pluto for Lua:
  - http://luaforge.net/projects/pluto/
- LuaPlus:
  - http://luaplus.org
- Kore
  - http://www.kore.net