Development Languages

Overview

- What games need from a language
- C++
- Scripting

Language

- First (and hardest to change) architectural decision
- Games have some unusual architectural requirements
  - Performance is far more important than most other domains.
    - Garbage collected languages can be particularly dangerous at scale.
  - BUT there is also an unusually large amount of complicated logic, which needs good abstractions.
  - Extreme portability is often important
- Many games have server components, which have different requirements from client but also from other web apps.
  - Server can afford to give up more performance than client (GC is probably fine), but not as much performance as other web apps (Game server are frequently no I/O bound, which most web services are)

Language

- C++ is by far the most common language for game (client) development
  - C# is probably second (Due to Unity)
  - Some outliers as well, from relatively common to very niche (Minecraft: Java, Jak & Daxter: GOAL)
- C++ provides a good mix of performance and abstraction
  - And is available almost everywhere
- More variety on the server, but still slightly different from other web tech stacks
  - C++ (To share logic with the client for multi-player games)
  - Java (and related JVM languages like Scala)
  - Go (For performance)
  - Erlang (For stability)

C++ Alternatives

- More modern system programming languages might eventually be able to replace C++ for games
  - i.e. Go / Rust / Swift
  - Or fast interpreted languages (Java, C#)
- More expressive in some areas than C++, generally safer
- All have problems though
  - Slower than C++
    - And, crucially, less control
  - Go / Java / C# : Garbage collection
  - Rust : Unreasonably strict
  - Swift: Poor cross-platform support
- Another interesting effort: Jai
- Huge momentum to overcome for any of these though

Scripting Languages

- C++ isn't the best choice for all problems
  - Complicated feature set, syntax
  - Low-level, dangerous
  - Lengthy shutdown/compile/link/rerun development process
- You can get lots of benefits from using a higher level language
  - Expressiveness, iteration, safety
  - More accessible to designers (and end users)
- Games usually try to play it a little safe and use a mixture
  - C/C++ “engine”
  - a high level language bolted on for “scripting”

Advantages of Scripting

- Expressiveness
  - Scripting languages employ higher level constructs than C++
- Reduced iteration time
  - Interpreted, or byte-code compiled on-the-fly
  - Live reload
- Useful features
  - Automatic resource management (garbage collection)
  - No pointers, safer data structures
  - Dynamic typing
- Accessibility
  - Easier for casual programmers to grok than C++
- Customizable
  - Write your own language, or change an existing one to suit your needs

Problems

- Additional work to bind the chosen language to C++
  - These bindings are often non-trivial, needing complicated template code, parsing header files, etc.
- Standard C++ tools are not script aware
  - Debugger, compiler, profiler
  - Script is a “black hole” to C++
- Multi-language debugging is harder
- Script code runs slower
  - Not always significant. 90/10 rule
- Harder to keep memory usage in check
- Having non-programmers write code can be a disaster

Division of Labour

- Script can can be employed with varying levels of commitment
  - Loading only, scripts just configure C++ components at load time
  - Small number of high level scripts running
  - Entity behaviour scripted
    - Each entity is running a little script (usually a state machine of some sort)
  - Write the whole game in the script language
    - Not really scripting any more then, is it?
    - Use C/C++ only to accelerate the performance-critical stuff

Common uses of scripting

- Mission scripting
  - Track high level goals for player (location to go to)
  - Position mission specific items / entities
- AI entity behaviours
  - Find nearest entity of another type
  - Issue a pathfind
- Controlling cut scenes
  - Position entities and cameras, trigger animation and dialog tracks
- Animation trees / blending
  - If this button gets pressed in this frame range of an animation play this animation

Scripting Philosophy

- Where a team stands on this depends on
  - the type of game they are writing
  - the programming ability of the scripters, and
  - how much performance they are willing to trade for the benefits of scripting

Concurrency

- It is often convenient to model an entity's behaviour/brain/state machine as an independent thread of execution
- Scripts provide a convenient mechanism to support this
- Often built in language support
  - E.g. Lua co-routines
- Script threads are co-operatively scheduled, so there are fewer issues compared to pre-emptive OS-level threads
  - No locking primitives, race conditions, etc.
  - Better performance
    - Cheap context switch

Choice of (Scripting) Languages

- Lua
  - Widely used in the game industry
  - Designed for embedding
  - Byte-code interpreted
  - Dynamically typed
  - Very flexible tables (associative arrays) as the fundamental data structure
  - Simple, clean syntax
  - Garbage collection
  - Not natively object-oriented, but can fake it effectively
  - Good performance (for a scripting language)

Choice of Languages

- Java / C#
  - Statically typed
  - Very rich standard libraries
  - Object oriented
  - Object reflection, serialization
  - Properties (C#)
  - Good performance
- Significant safety benefit over C++, but not much gain in expressiveness
- Generally these languages are too close to C++ to offer huge benefits as a scripting language
  - A decent choice as an alternative to C++
  - Unity does offer C# as (a) scripting language, though

Choice of Languages

- Perl / Python / Ruby / PHP
  - Very different syntax, but similar feature sets
  - Rich standard libraries
  - Dynamically typed
  - Garbage collection
  - Generally pretty slow
  - Large memory footprint
- All are intended to be much more self contained application languages
  - Fairly hostile to embedding in C++
- Optimised for text processing, which isn't a big concern for games

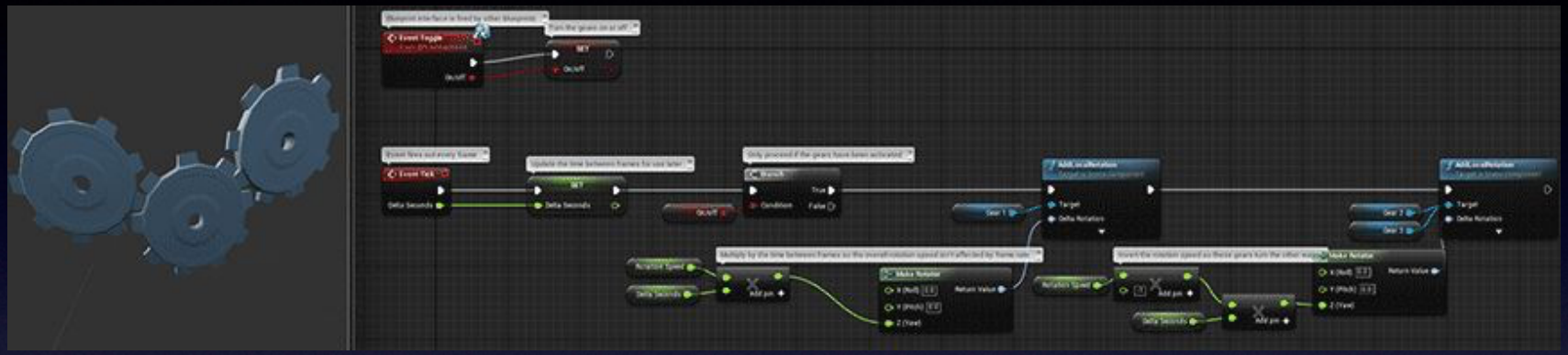
Choice of Languages

- Lisp, Scheme, etc.
  - Very expressive syntax
    - Not terribly readable, though
  - Dynamically typed, GC'd, interpreted or compiled
  - Easy to write an interpreter for
  - Garbage collection
  - Decent performance (if you have a good implementation)
  - Doesn't play well with C++
  - Hard for people trained in imperative programming styles to learn
- JavaScript
  - Lots of advantages due to ubiquity in web programming
    - Easy to find/train developers
    - Very fast and embeddable implementations out there (JavaScriptCode, V8, SpiderMonkey)

Choice of Languages

- Other embeddable languages:
  - Small, Squirrel, Io, interpreted C (LCC), Tcl, etc.
  - There are a lot of choices out there
- Roll your own?
  - Get exactly the mix of features you want
  - Seamless C++ bindings, state-machines, concurrency, event handling, resource management
  - Less common even at AAA level now
    - UnrealScript: Java-esqe language with game-oriented features, deprecated in favour of Blueprint
    - GOAL: Naughty Dog's in house lisp variant, ditched when their Lisp guy retired, though they now use an off-the-shelf Scheme interpreter (and more recently Racket) as a scripting language

Visual scripting languages



- Get away from text oriented programming environment
- Popularized by Kismet in UE3 and now Blueprint in UE4
- Even more accessible to non-developers

Summary

- C++, for better or worse, still the big dog in game development
- It's worth using scripting to gain some productivity and expressiveness back that C++ takes away
- Lots of approaches and languages
  - Pick a commitment level
  - Lua is often the default for text scripting these days
  - Visual scripting is getting to be more of a big deal as well