Scriptable Render Pipeline

Future of Rendering in Unity

Aras Pranckevičius
Problem In Pictures
Why is that a problem?!
Unity’s Render Pipeline Today, In Theory

- Forward or Deferred
- Configurable
  - Custom shaders, both for materials and lighting
  - Compute shaders
  - Custom post-processing effects
  - Command Buffers
- Works well on all platforms
Unity’s Render Pipeline Today, In Practice...

- Big black box system
- Hard to configure right
- Flexibility is not awesome
- Performance is not awesome
- “One Size Fits All” trap
- Often does not use platform specific strengths
- Changing the behavior is hard

:(
New Goals!

- Small C++ core
- Expose APIs
- High level “render loop” logic in C#
What do we want our renderer to be?

Lean

- Minimal surface area
- Testable
- Loosely coupled
What do we want our renderer to be?

User Centric

- Lives as extension or in user’s project directly
- Debuggable
- Extend and modify
- Fast iteration time for changes
What do we want our renderer to be?

Optimal

- Perform fast, duh
- Optimal for:
  - Particular platform
  - Particular application type
- Allow removing things your project does not need
What do we want our renderer to be?

Explicit

- Does exactly what you tell it. Nothing more. Nothing less.
- No magic
- Clean API
Scriptable Render Pipeline
Engine (C++) vs userland (C#) split

- If it’s perf critical, it’s done in engine/C++
  - Future: maybe in C# if we can make it fast (ongoing research)
- Engine C++ code:
  - Culling
  - Sorting / Batching / Rendering sets of objects
  - Internal graphics platform abstraction
- C# / shader code:
  - Camera setup
  - Lighting / shadows setup
  - Frame render passes setup / logic
  - Shader & compute code
This is not rocket surgery

● High level code / config to describe rendering idea is not new:
  ○ “Destiny’s Multi-Threaded Rendering Architecture”, Natalya Tatarchuk, GDC 2015
  ○ “Framegraph: Extensible Rendering Architecture in Frostbite”, Yuriy O’Donell, GDC 2017

● Should it be data (graph / config files) or code (C# / Lua / …)?
  ○ We went for code
  ○ Programmers like code more than noodle graphs :)
  ○ Some decisions are branchy and game state dependent
Main C# APIs

- Cull specific views
- Render subset of visible objects
  - With info on what material/shader passes to use
  - With sorting flags
  - With “what kind of per-object data to set up” (light probes, per-object light lists, etc.) to set up
- Already existing APIs for:
  - Setting up render passes / render targets
  - Setting up shader constants / global resources
  - Dispatching compute shaders
  - Rendering individual meshes (for special fx / post fx)
- APIs build a “command buffer” that is later analyzed/executed
C#?! U MAD?!?!

- This is high-level code operating on frame structure
- No per-visible-object C# bits
- Actually runs faster and schedules better than our old C++ render loops!
- We also have a bunch of threading / no-GC things cooking for C#, soon...
Want to ship out of the box

- PC/Console/High-Mobile pipeline *(codename “HD”... naming is hard!)*
- Low-end mobile pipeline
- VR
HD Pipeline

- PBR, GGX, area lights, FPTL/clustered, aniso GGX, layered, SSS, ...
  - All the buzzwords :) 
- Requires compute shader support
- Watch it live! github.com/Unity-Technologies/ScriptableRenderLoop
Great. When?

- “Experimental” in Unity 5.6 since last year!
  - [unity3d.com/unity/beta](https://unity3d.com/unity/beta)
  - [github.com/Unity-Technologies/ScriptableRenderLoop](https://github.com/Unity-Technologies/ScriptableRenderLoop)
  - API keeps on changing

- Want to ship “for reals” in release after 5.6