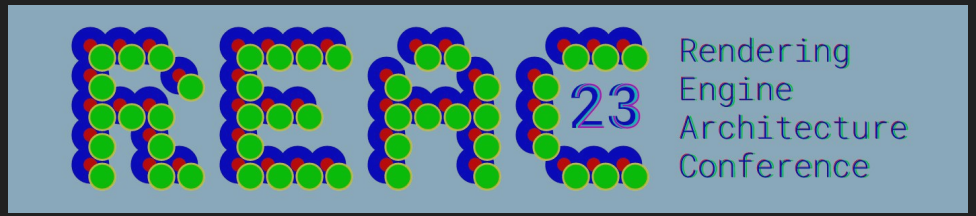
Far Cry Dunia Engine Shader Pipeline

Long-term Vision & Lessons Learned



About us

Jendrik Illner

Senior Rendering Engineer at Santa Monica Studio since 2022

This talk is about my time at Ubisoft, junior 3D programmer on FC Primal in 2015 to 3D technical lead on FC6 in 2021

Cong Hao He

3D Programmer at Ubisoft Montreal since 2019

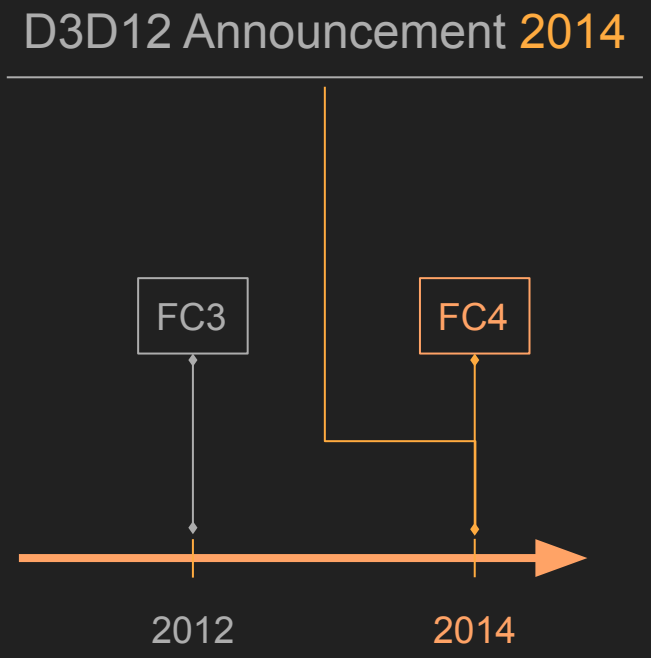
Coming from a background of graphics driver and SDK development, this talk is about my first AAA production experience.

Presentation Overview

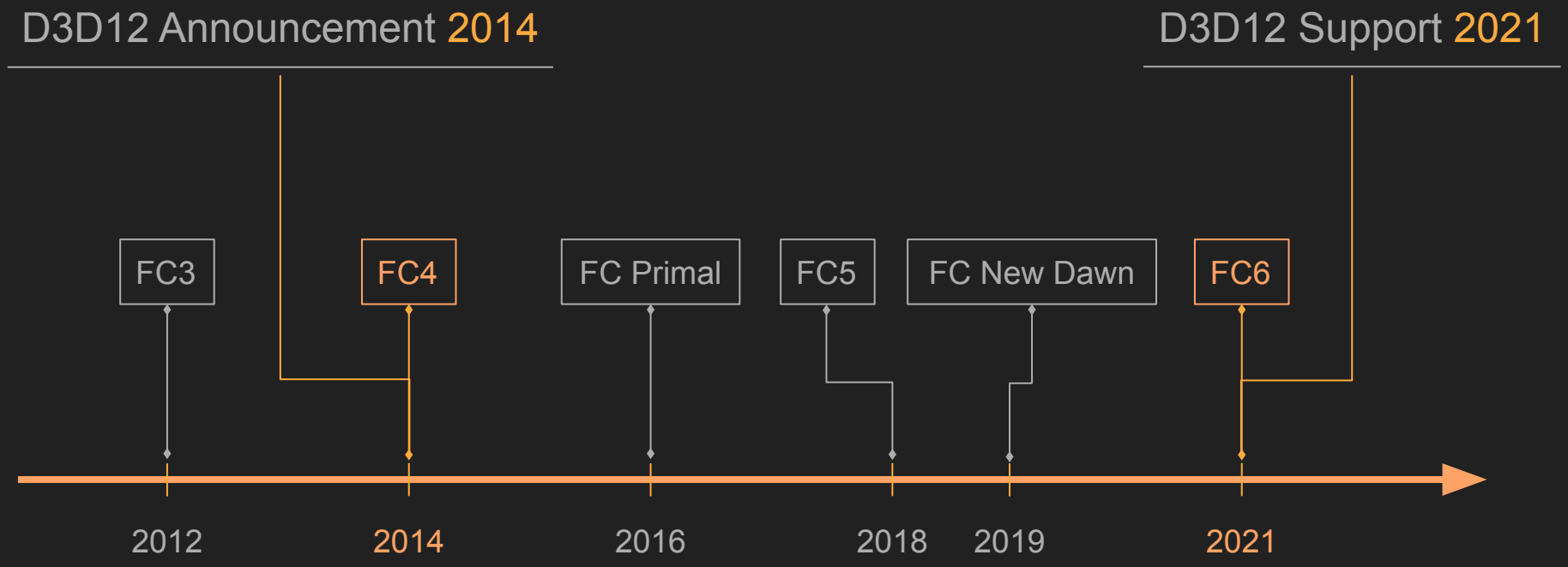
- Challenges from a legacy codebase
- Long-term tech vision and planning
  - Architectural & workflow preparation during Far Cry 5
- Executing the vision
  - Improving the Architecture and getting ready for production
- Fine-tuning User Experience

Let's go back in time a bit

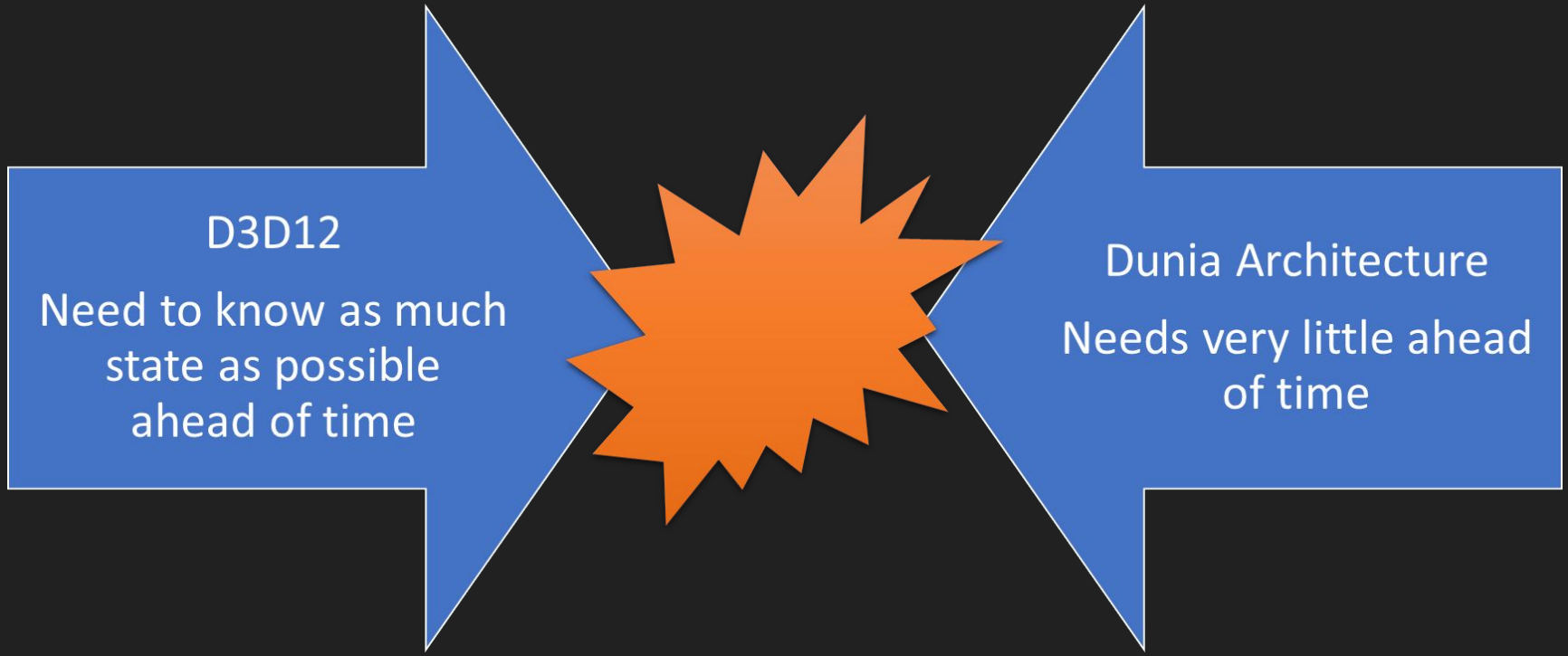
Far Cry 4 + D3D12 announcement



Far Cry 6 + D3D12 support



Why did it take so long?



Far Cry Code Challenges

- Aimed at D3D11 level architecture
  - Dynamic states, set from code, combined just before the draw call
  - Art is free to arbitrarily apply shaders to all kind of meshes
- But front-end system also in need of upgrade
  - Terrain, Water, GI, Sky ….
  - Updating to physical based lighting …

What we need to achieve

Update the backend for new APIs

+

Develop new features

+

Keep shipping on a regular schedule

Long-term Technical Vision and Planning

Long-term Technical Vision and Planning

The rest of the talk is about how we solved these

D3D12 architectural challenges can be broken into three groups

Resource States / Barriers

Render State Management (PSO)

Memory Management

This talk will mostly focus on the shader state side

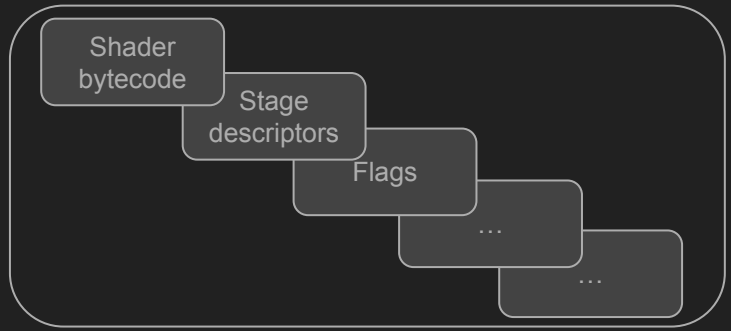
PSO Refresher

What's in a Pipeline State Object?

Easier to say **what is *not***:

- Resource bindings
- Viewports
- Scissor
- Blend factors
- Depth/Stencil reference values
- Topology

► Everything else is part of the PSO



Why talk about this today?

Image source: Digital Foundry

(By the way) You Can Use Vulkan Without Pipelines Today ...

Dunia Shader Authoring

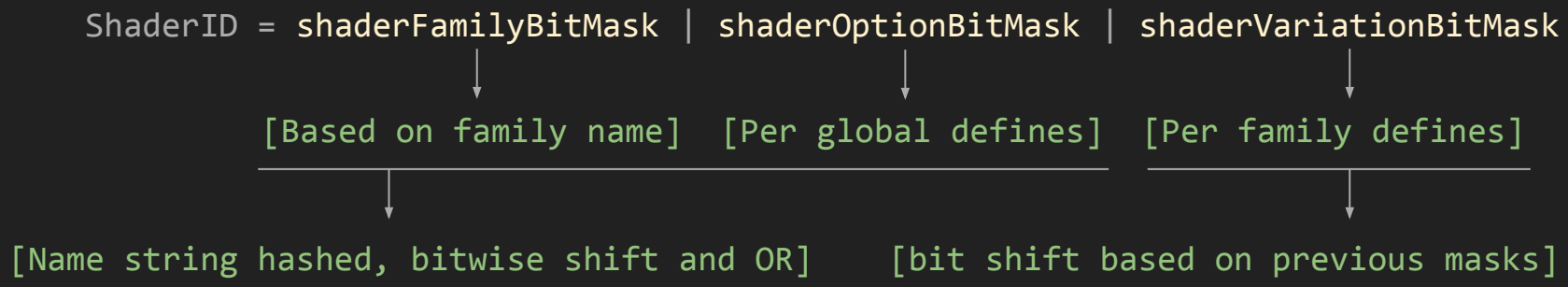
Expressing Material connections

- Our shaders are authored as ***Shader Families***
  - Group of shaders that implement similar material responses
  - Skin, Car Paint, Vegetation, Weapons, ….

FC6 - Shader ID to PSO transition

Each shader has a ***ShaderID***: a 32-bits uint

Composition:



This ShaderID is referring to a pair (vertex/pixel) or compute shaders

Shader Pipeline Goal

- Goal: “Each ShaderID in the engine maps to a PSO”
  - Reduce the runtime overhead
  - Reduce possibility for mistakes in state getting out of sync

```
commandList.Draw( ShaderID, DrawCallBindingInfo );
```

Identify the challenges to reach the goal

- Shader and Pipeline state is fully separate
  - Shaders are in Data (hlsl) and Pipeline state is in Code (C++)
- No Art restrictions
  - Every model could be used with any shader and it was kinda expected to work
  - Nearly every object in the world could switch to a fully different and unrelated shaders at any point in game

The Plan

- Extend the shader compiler to allow state parsing
- Move PSO states into each family

```
technique {
      BlendOp0 = Add;
      RenderTargetFormat0 = R8_UNORM;
      ....
}
```

Can Art Help?

- Define clear rules for shader and mesh combinations
  - Shaders define supported vertex format, only models with matching layout can use a shader
  - Only shaders from the family can be switched on the same model
- If a rule is broken, shaders will be replaced with a placeholder material in development builds

► The clearly defined restrictions gave the teams enough time to adjust

  - Art pipelines for AAA are long
    - Need as much time to adjust as possible
    - Adjust workflows before it’s needed!

Long-term Technical Vision and Planning

- Have a plan for FC5 to get closer to solve PSO challenges
- What about barriers?
  - That’s a topic for another talk, but here is a brief overview

Resource States / Barriers

Render State Management (PSO)

Memory Management

Barrier Work (Overview)

- One of FC5's main task
- Annotate resources as it flows through the frame in the form of:
  - Resource Stage + Resource Usage

```
INSERT_TRANSITION( trackingContext, gBufferRT0, ResourceUsage::PIXEL_SHADER_WRITE );
```

- High-level barrier system based on explicit binding model
  - Each time a user would request a new binding for a texture, render-target, buffer etc.
  - Explicitly specify the usage
- Resolve dependencies just before draws/dispatches to issue correct barriers
- Pros: Simple to debug, lightweight system
- Cons: Manual annotation is time consuming and error prone

- Release

Tech Milestone

- D3D12 compatible barrier system in place, and functional on consoles
  - Not used on PC (but validation system is running in development builds)
- Data restrictions in place and respected by artists
- The shader system prepared for the introduction of PSO state
  - Data builder pipeline complete, data loading infrastructure
  - First pass of render state in shader files

- Production Begins

FC6 challenges

- New lead studio (Ubisoft Toronto instead of Montreal)
- Lots of technical must haves:
  - D3D12 for PC
  - Vulkan for Stadia
  - PS5
  - D3D12 for XBSX
- What is available:
  - D3D11 for PC, XBox
  - PS4

► Lots of work ahead, but we are prepared and have a plan

High Level Plan

1. Start working on a PC Platform (either Vulkan or D3D12)
    - Our PC/Stadia team picked Vulkan as they would be responsible for Stadia as our first external technical milestone
    - New Targets (PS5/XBSX/Stadia) followed shortly after (had to get the engine up-and running on the targets first)
2. …. Do general porting work
3. Barrier system from FC 5 is ready for the task!
    - Huge time saver for FC 6
    - Not perfect immediately, cases not caught, custom validation layer disagrees etc
    - + Developers already well aware of working with barriers
4. Once we had a basic version working, it's time to take advantage of the new APIs

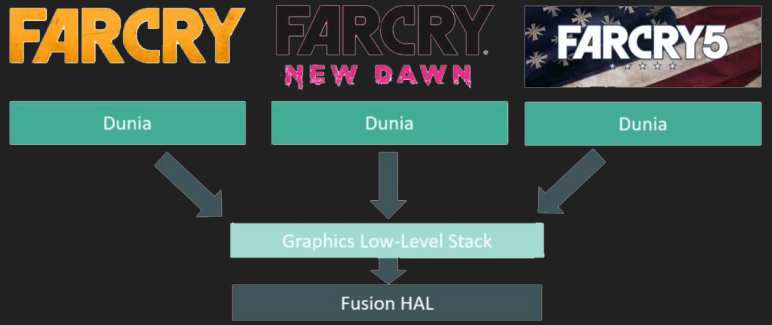
FC6 - Transition to Modern APIs

New APIs: *Vulkan, D3D12 and AGC*
New Platform: *Stadia, XBSX, PS5*

How to approach this with limited resource?

Shared Ubisoft initiative (Fusion):
A low-level API abstracting common APIs

- “Port” once to this new API
- Platform specific work can be shared
- Reuse between games & engines



On Demand PSO

At this time we had implemented the obvious PSO integration

1. Cache all render state; calls from the runtime
2. On each Draw Call, hash the whole state + look up PSO in a Dictionary
    - If not ready, skip the draw call until the PSO is ready at some point in the future

► Never stall rendering, but certain objects might be missing for few frames

Skipping Draw Calls

Dunia Philosophy:

If Dispatch or Draw shaders are not ready, do *not stall. Skip* the call

Compute shader dispatch

```
// Shadow tiles
bool isSuccess = device.Dispatch(BuildTiles);

if (isSuccess)
    isSuccess = device.Dispatch(BuildIndirectArgs);

if (isSuccess)
    isSuccess = device.DispatchIndirect(ResolveShadow);
```

Draw Shader

```
// A class monitoring shader readiness
ShaderMonitor monitor;

device.Draw(ShaderID_0);
device.Draw(ShaderID_1);

bool isSuccess = monitor.AnyShadersNotReady();

if (isSuccess)
    // Do something
```

Sidenode: UE5 supports a similar way now since 5.2 (for draw calls)

Expanding Skipping Draw Calls

Very helpful when porting to a new API as well

- Hook up the validation system to the skip conditions
- If a draw call triggers a warning/error, skip the draw call instead of potentially hanging the GPU
    - We had allow lists of errors that we know are “safe”

Iterative Improvements

We focus on making iterative improvements

- We have a system that does a decent job of getting the game running
- QA can test, we can develop and improve all systems from here in parallel

Improve Shader Variations

- Started with a long list of variations
- All of same importance, but that’s not true

Variations

- Vertex Formats
- Output Formats
- Burning
- Wetness
- Wind influence
- Skinning
- …..

Improve Shader Variations

- Started with a long list of variations
- All of same importance, but that’s not true

Variations

- Vertex Formats
- Output Formats
- Burning
- Wetness
- Wind influence
- Skinning
- …..

Essential vs Optional

Essential

- Vertex Formats
- Output Formats
- Skinning

Optional

- Burning
- Wetness
- Wind influence

Anything that only adds additional information to an object is marked as optional

- For example when a object starts burning we don’t mind if the shader is not swapped for a couple of frames. As the objects gets burned slowly over time anyway
- QA will flag cases where this might be cause undesirable artifacts

Fallback to less specific PSOs

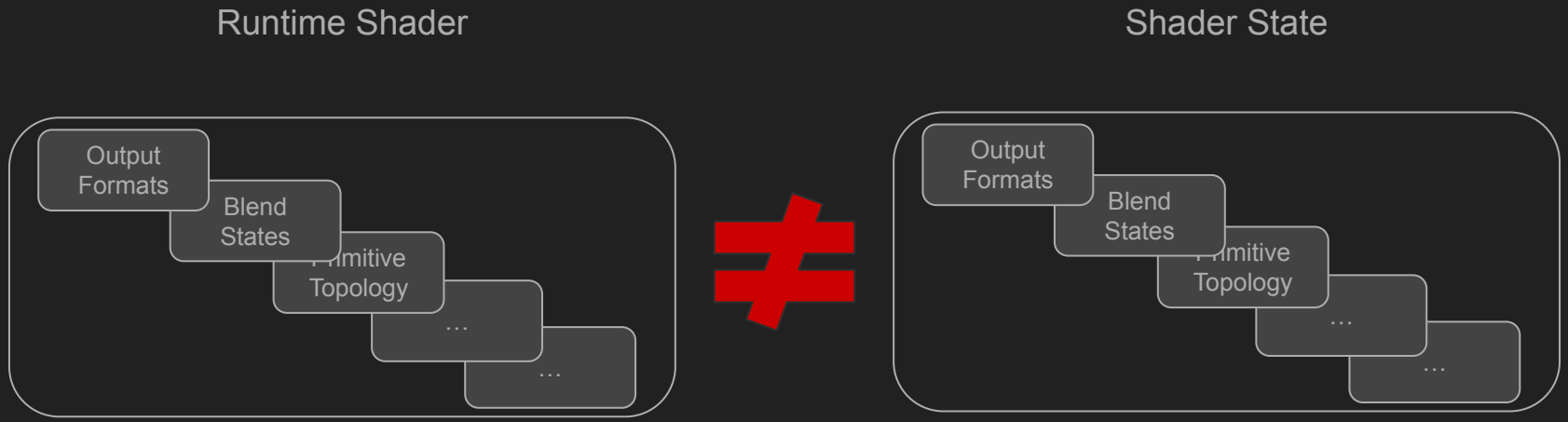
- Now when we hash the PSO, start looking for fallback options if it’s not ready
  - Try to remove optional defines and see if we can find a match
  - If we find one, use it instead
  - Otherwise keep searching
  - If we don’t find a match, we are back to skipping draw calls

Map ShaderID to PSO - The Journey begins

Transition Start

- Start with work on Xbox Series X
  - Why? AAA reality of our team in Montreal owning the xbox platform
  - And we had the time in the schedule set aside and we thought it was important
  - Less worried about regressions as it’s the least “tested” platform at this stage

Two Sets of States



The two sets should match, but of course they don’t as the Shader State was setup during FC5 but unused up to this point!

Validation system needed to catch the mismatches

Validation System

Introducing a state validation system

Different validation mode available:

1. Validate state & prefer runtime state
2. Validate state & prefer runtime state & log warning
3. Validate state & prefer data state & log warning
4. Validate state & skip draw call & log warning
5. Only use Data state

► We want to get to Level 5 but had to start with level 1

Turn on & hope for the best

- … And >90% cases are falling back to the runtime state
- What are the most common issues?
  - Render Target Formats, Blend Settings
- Fix these up
  - Sounds bad, but most state is in common include files
- Keep on iterating on this on programmer only
  - Until we can boot and play the game for a couple of minutes

```
technique {
#include "BlendedStates.inc.fx"

#if defined(TWO_SIDED)
        CullMode = None;
#endif
}
```

Expanding the Test Audience

- At this stage we were more confident and started enabling a warning on screen if a mismatch is detected
  - First on a small number of testers machine (the local 3D Programmer QA)
  - Expanded globally once local QA didn’t see any warnings regularly

- Now QA would record information into JIRA as any other bugs

Automatic QA Helper

Most issues resolved via code change near the end of production

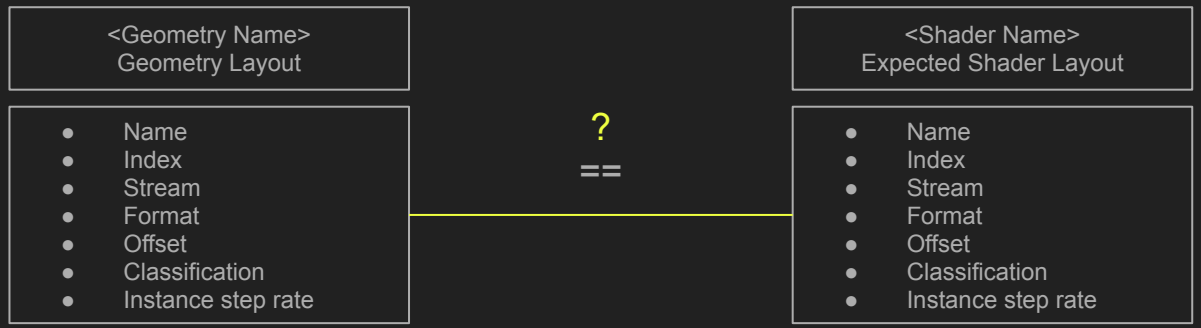
- But there are a couple of cases left that need art to resolve
- We want to provides technical directors with enough information to track down the causes and assign issues to the right person without programmer involvement

JIRA Generation

Custom internal API to generate a Jira from runtime code on any platform

We validate geometry layout vs. what the shader expect

- Mismatch information directly available in Jira bug report
- One JIRA for each geometry + shader combination
- Directly assigned to the responsible Technical Director



Final Testing and Validation

Test Coverage

- Environments
  - Urban, dense vegetation, open field, etc. in day/night time and various weather conditions



- Gameplay settings

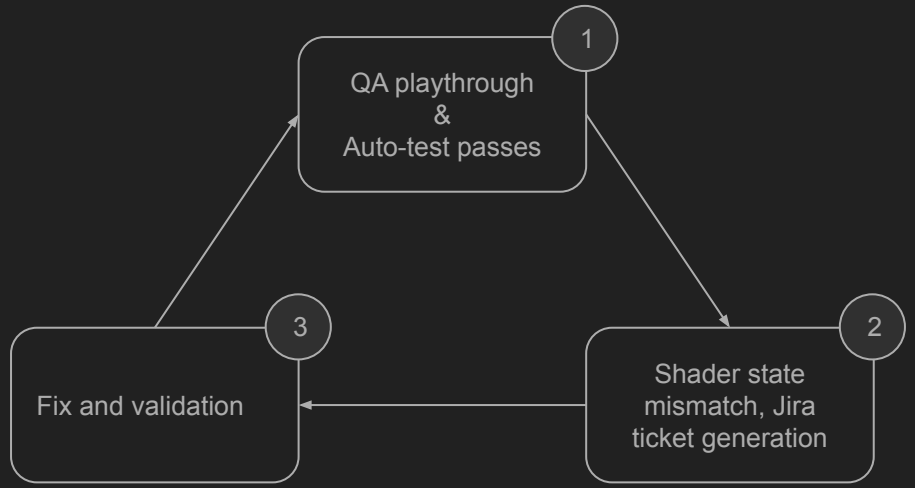


Automatic Testing

- Spawn at fixed locations, deterministic
  - Automatically collects screenshots, profiler perf captures, and Pix captures
- Spawn in all map sectors, bot randomly playing the game
  - Catch regressions, also perfect for detecting shader state mismatch

Final Testing Process

- A combination of manual testing by QA team and (mostly) automatic scripts
  - QC team: mission, cutscenes, game menu
  - Automation: all map sectors and environment setting permutations
- An iterative process to detect and fix most of state issues



Interesting Edge Cases

Noticed a common pattern, whenever we would aim at an enemy we would get an assert with mismatch

- What was this all about?
  - Outline shader!
- Effect that bypasses the shader override system
- And needs to be applied to all kinds of model formats

Add a variation for each vertex format?

Custom Vertex Fetch

- Decoupling the Vertex Format from the format the shader expects
  - Using Programmable Vertex Pulling
- The Shader only needs to to know Position + Normal
  - Doesn’t care if it has two set of normals, tangent space or anything else
- Each Model exposes a Structured Buffer View onto the data streams
  - The C++ logic can query the mesh and bind these for the outline shaders
  - If it gets applied to an object that doesn’t offer the views a dummy stream will be bound

► Designed for reduced shader variation count but proved to be a very flexible architecture as a side effect

Each ShaderID in the engine maps to a PSO!

- This was a great amount of work, we are grateful we broke this work down over multiple games
- The runtime is super simple, most cases will never need to deal with render state
  - All state is managed centrally in a couple of common header files
  - Shaders own the whole state in a single place
  - No more state leaking
- Really worth the effort and an engine architecture is better if designed for it
  - Large AAA games can do it, we proved that!

Fine-tuning User Experience

Shader Pre-compilation Process

We have all the runtime side of things in-place, but how to improve the first start experience?

We have a couple of more techniques

1. Collecting runtime PSO usage
2. Compiling Shaders on first load

Collecting PSO Usage

- Large amount of Variations that could be used, but most aren’t
- Each run connected to a database
  - Track what kind of shaders are used
  - Consolidated all runs into a single set of PSOs
- Final list of the ~10k “most used” PSOs
  - Some forced shaders added manually (e.g. copy shaders, clear shaders)
  - All shaders are compiled on boot of the game and are ready by the time main menu loads

Priming during loading screens

The second part happens during each loading screen

1. At the end of the loading process, don’t immediately hide the loading screen even when the player is loaded into the world
2. Render the world behind the loading screen
3. Spinning the character around itself
4. This primes the virtual textures and shader variations
5. Remove the loading screen only once both parts are done

► This way we can be sure all shaders for the new location and virtual textures are ready before the player can see them

Conclusion

Lessons Learned

1. When major technology change is required, identify and tackle the architectural changes needed first
2. Start setting up the supporting systems early
3. See restrictions as an opportunity for better design, not annoyances
4. Use iterative design process, each step gets closer to a shippable solution
5. Think long term, don’t limit yourself to current project timeframe

Acknowledgements

Special thanks to all members of 3D graphics team who contributed over the years to make Far Cry games possible.

Thank You & Questions

Sources

Advanced Graphics Summit: 'Marvel's Spider-Man' Remastered: A PC Postmortem - gdcvault - video

The Challenges of Rendering an Open World in Far Cry 5 - advances.realtimerendering

OpenGL Insights - Programmable Vertex Pulling

UE5 Release Notes - unreal engine 5.2 release notes