Moving Cyberpunk 2077 to D3D12

Tim Green
CD PROJEKT RED
Warsaw, Poland

ABSTRACT

This talk provides an in-depth look at the D3D12 integration in Cyberpunk 2077. It discusses the transition from D3D11 (used in The Witcher 3: Wild Hunt) to the new API, while other parts of the renderer were actively being worked on. It then goes on to look at how some specific topics were approached and implemented; in particular, tasks that were previously handled by the driver, but are now the responsibility of the application, such as memory management, resource barriers, resource binding, and command list submission. It ends with a look at a material used extensively in the game, and how it worked together with D3D12.

ACM Reference Format:
Tim Green. 2021. Moving Cyberpunk 2077 to D3D12. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks (SIGGRAPH '21 Talks), August 09-13, 2021.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3450623.3464664

1 INTRODUCTION

Effective use of explicit graphics APIs, such as D3D12 or Vulkan, requires an increased level of involvement from a renderer, compared to older APIs like D3D11 or OpenGL. The application is responsible for many things that were previously handled by the driver. While this can enable more flexibility and higher performance, it comes at an increased error risk, which can lead to reduced performance or instability.

Each D3D12 integration will approach these topics differently, with different goals. For example, in managing resource states, [Meyer 2016] annotates render code with state transitions. This provides more control to the renderer, but the transitions could add clutter to the code; it also appears to force all state changes from a single thread, which would simplify implementation and eliminate the need for complicated state patching, but can create CPU sync points. [Rodrigues 2017] builds a graph of "producers" and analyses that to automate barrier generation. This can be powerful, but it seems as though complex passes requiring multiple intermediate results could lead to an explosion of producers and dependency specification boilerplate.

For Cyberpunk 2077, some decisions behind the D3D12 integration were made to help limit the impact it would have on the renderer, and to limit exposure to error-prone busywork. When possible, the approaches would lean toward keeping our existing interfaces, with functionality mimicking D3D11 (slot-based resource binding, resource state implied by usage, etc).

SIGGRAPH '21 Talks, August 09-13, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8373-8/21/08.
https://doi.org/10.1145/3450623.3464664

2 IMPLEMENTATION

2.1 Getting from there to here

The transition to D3D12 took some time, starting with an overhaul of the renderer itself before being able to move on to platform specifics. Temporary infrastructure was built to prepare the renderer for multithreaded command list generation, allowing development to continue, with the render happening across all threads before the actual command list management was available. A limited emulation of bindless textures was made to allow development of our "multilayered" material.

While new features were being prototyped and developed, part of the team moved to a separate code branch to switch to D3D12. Thanks to our separation of platform layer ("GpuApi") and renderer, and the prep work carried out in the renderer, there was little overlap between the branch and what was happening on mainline. This meant work could continue smoothly on both sides. On the branch, it was always possible to switch between D3D11 and D3D12 with a compile-time switch to verify functionality. When the time came to merge back into mainline, D3D12 performance was roughly on par with D3D11.

2.2 Details

Memory management. For most resource allocations, we use the D3D12 Memory Allocator library from the GPUOpen Initiative [D3D12MA 2019]. It is set up to use committed resources for larger allocations, and suballocation from heaps for smaller ones. D3D12's resource alignment requirements can lead to large amounts of wasted memory, so for some types of immutable buffers, we have an additional pooling system to pack multiple buffers into a single D3D resource. This saved several hundred MBs of GPU memory.

For temporary resources used during the course of rendering a frame, we use a simple memory aliasing model, allowing resources with non-overlapping lifetimes to share memory. Even without more advanced aliasing, this can reduce memory use by 400MB at 1080p on Ultra settings.

Improving memory management has been an ongoing effort. Even with multiple strides already made in this area, there is still room to do more in terms of both waste reduction and budgeting.

Resource barriers. For the most part, resource states are tracked and managed entirely within GpuApi. The tracked state is updated when a resource is used (if binding a render target, then go into RenderTarget state) and an appropriate barrier is generated to take it to that state. Since different command lists can be filled in parallel, and those command lists might use the same resources, there is an additional pass when submitting command lists to inject unknown

state transitions once the previous state is known. This can help improve batching of resource barriers, and avoid extra transitions to a default state between command lists.

In some cases, performance can be improved by controlling when a transition happens, instead of waiting for a resource to be used. We allow the renderer to set a state directly, but this does not replace or disable the internal state management. UAV and aliasing barriers are more difficult to anticipate, so they are always explicitly handled by the renderer.

Automatic state tracking in this way adds some overhead: additional code must run to update the current state, and per-command list and global states must be stored. It seemed a fair tradeoff, though, compared to the potential maintenance cost of requiring explicit transitions in the renderer.

Command list submission / Async compute. When the renderer submits a set of command lists, a pass is made over them to resolve any missing resource states, adding appropriate transitions to the preceding command list. If needed, an additional command list can be injected at the front to set up any initial states. The command lists can then be closed and submitted to D3D12.

Async compute is supported through a separate compute queue, in a simple fork-join model. All command lists that are meant to overlap are submitted together so resource states can be resolved between both direct and compute command lists. The queues are synchronized before and after. This prevents more advanced usage patterns, but it works well with the resource state management.

Root signature. A single common root signature is used for all graphics and compute workloads. It gives a binding model similar to D3D11, where each stage has some number of SRV, CBV, and UAV slots available. The ranges are split up into multiple root parameters based on expected usage patterns from the renderer (e.g. we know one particular CBV slot gets bound frequently, so it can have a separate root parameter). It is still a general-purpose signature, not hand-crafted for specific passes. In addition, all root parameters are descriptor tables; there are no root descriptors, CBVs or constants. This can add some overhead copying descriptors into a descriptor heap, and result in setting root parameters more frequently than necessary, but it was done to keep the implementation simple.

Resource binding. We use a single resource descriptor heap, split into several segments to handle different situations. There is a section managed as a ring buffer, for general purpose binding. There is a section for "resource packs," which gives a more efficient way to bind material parameters, where the referenced resources will not change. Finally, there is a large descriptor table for "bindless" resource access, primarily for textures used with our "multilayered" material. We do not use this for all resources, just where it's required. For ray tracing this is extended to include more textures and vertex/index buffers, since those must be accessible as well.

Sampler descriptors are managed in a simple cache. We found that there are very few unique combinations of sampler required, so we simply fill a single descriptor heap with the unique combinations as they come up, and keep a hash map to find existing sets.

GPU Crashes. Eventually something will go wrong. Some shader gets garbage data and loops indefinitely, or a resource isn't set correctly and an invalid memory access occurs. Debugging a GPU crash is never a fun process, and effective tools are scarce. D3D12's built-in debug layer can help with some errors and GPU-based validation can help with others, but at such a high cost it can make it difficult just to get to the point of failure. Tools like NVIDIA's Aftermath [Aftermath 2021] can help track down where in a frame a crash occurs. We developed a custom system similar to Aftermath to get additional context, provide a second opinion for comparison, and to have a similar feature on unsupported systems.

3 CASE STUDY: MULTILAYERED MATERIAL

Figure 1: Multilayered material applied to a mesh, made feasible with D3D12. By changing only a lightweight buffer specifying per-layer properties and which textures to index, different appearances are possible.

Found on most objects in the world, our "multilayered" material composites multiple base material types at render time. The base materials are a set of small, tileable textures, which are painted onto the object with a stack of masks. The goal was to have relatively small unique masks for each object, with a large material library of always-loaded textures, instead of having unique high-resolution textures for each object. This reduced the need for advanced texture streaming, since most things would be using the same set of global textures, and the remaining would be low-resolution textures for distant proxy geometry, and the packed masks.

Although we did not go "full bindless" with our material system, we did need it for multilayered materials. Assets typically have up to 20 different layers, each with 5 textures; it would be impractical to set 100 descriptors for each one, either by binding those textures dynamically, or reserving that much space in resource packs. Instead, those textures are available in the bindless region of the descriptor heap, so the material can access whatever it needs. Since different parts of a mesh can have different layers affecting them, we use Shader Model 6.0 wave intrinsics to make layer indices uniform, to ensure correct indexing of the textures.

REFERENCES

Aftermath 2021. *Nsight Aftermath SDK.* https://developer.nvidia.com/nsight-aftermath

D3D12MA 2019. *D3D12 Memory Allocator.* https://github.com/GPUOpenLibrariesAndSDKs/D3D12MemoryAllocator

Jonas Meyer. 2016. Rendering Hitman with DirectX 12. In *Game Developers Conference 2016.*

Tiago Rodrigues. 2017. Moving to DirectX 12: Lessons Learned. In *Game Developers Conference 2017.*