Vincent Van Denberghe

Performance and Architectural Study of Direct3D 11 and Direct3D 12

Graduation Work 2022-2023

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT

This graduation work consists of a project that renders the same scene in DirectX 11 and DirectX 12 using GPU instancing. This paper entails to compare the two versions of the graphics API, detailing the architectural differences in a simple rasterized scene and analyzing the performance differences between the two versions. DirectX 12 offers a lot more low-level control than its predecessor DirectX 11. However, the introduction of the command list/queue, descriptor heaps, pipeline state objects, resource barriers increase the barrier of entry for DirectX 12, making it difficult to start without prior knowledge of the DirectX render pipeline. Comparison of the two APIs' performance in the scene chosen is very similar, but the differences are plenty when looking at the underlying code and the difficulties of modern-day DirectX 12 video games are not that apparent.

INTRODUCTION

DirectX 11 has been the standard for graphics rendering in video games for many years. However, with the recent rise of DirectX 12, developers are now presented with a new set of tools that promise greater control and support for advanced features such as hardware ray tracing and variable rate shading.

The arrival of the 9th generation of video game consoles, such as the PlayStation 5 and Xbox Series, has increased the demand for better graphics performance. DirectX 12 offers several advantages over its predecessor, including improved CPU utilization and support for low-level hardware abstraction. However, the transition to the new API has not always been smooth, as demonstrated by games such as Borderlands 3 and Final Fantasy 7 Remake, which have experienced performance issues when using DirectX 12.

The goal of this paper is to explore the capabilities of DirectX 12 and to create a rasterizer using the new API. The differences between DirectX 11 and DirectX 12 will be benchmarked and the performance and code architecture of both versions will be analyzed. The challenges of implementing DirectX 12 in game engines will be discussed, as well as the potential role of D3D11On12 in porting existing games to the new API.

DirectX 12 offers significant improvements over DirectX 11, but the transition to the new API is not without challenges. By comparing the performance and code architecture of the two versions, we can better understand the advantages and disadvantages of DirectX 12.

RELATED WORK

The differences between DirectX 11 and DirectX 12 are numerous but can be boiled down to a simple topic: increasing performance by handling agency back to the user. What sets the 2 versions of the API apart is that DirectX 12 provides a lower level of hardware integration than previous version of DirectX, enabling significant improvements to multi-core CPU scaling for video game titles written using the API.

For example, where DirectX 11 has automatic memory management, DirectX 12 titles are responsible for their own memory management. Furthermore, DirectX 12 makes use of command queues and lists, descriptor tables and pipeline state objects to reduce GPU overhead[1].

1. DESIGN

1.1. API

This project will use the Win32 API, while following the DirectXTK11[2] and DirectXTK12[3] (DirectX 12 Tool Kit) tutorials using its Direct3D Game Visual Studio Templates[4]. This is a public series of tutorials created by Microsoft. These tutorials for designing this project's DirectX code will be followed, including their best practices.

1.2. SCENES

The application will consist of a single scene with meshes. Using GPU instancing[5,6] to render the same object a great number of times, the goal is to stress the GPU as much as possible while eliminating the CPU bottleneck, in the hopes of getting clear performance readings. DirectX 12 mesh shading will not be used, due to time constraints as well as to maintain feature parity between the two API versions.

1.3. CLASSES

The code will be written in a way that facilitates comparisons between similar aspects of the rasterizer pipeline. The initialization of both API versions is naturally different, so initialization is split up into two classes that inherit from the same base class, and the sub classes then implement/override the base class functions. This will make comparisons much easier, as one-to-one comparisons between the two APIs become possible.

1. BENCHMARKING

At runtime, FPS will be logged and saved to a CSV file, which can easily be read using Microsoft Excel. This CSV file will contain a timestamp, current FPS and the currently used graphics API. Performance will only be logged like this once every second.

A second goal of this paper is to try to pinpoint what can go wrong when creating games with DirectX 12, and why games tend to struggle when using DirectX 12. A third version, using the D3D11On12[7] API, a mechanism by which developers can use D3D11 interfaces and objects to drive the D3D12 API, might provide an interesting middle-ground between the two graphics APIs. However related, this specific method will not be implemented in this project.

CASE STUDY

1. INTRODUCTION

Figure 1. Image of program scene

The scene used was inspired by Microsoft's Basic Instancing scenes[8] for DirectX 11 and DirectX 12.

The classes used in the DirectXTK (DirectX Tool Kit)[2,3] are not representative of actual DirectX programming, as they provide abstractions of DirectX functions. The Simple Instancing scene found on Microsoft's XBoxUWPSamples[8] GitHub page used GPU instancing to render multiple instances of a mesh, using DirectXTK functions only for assigning buffer constants in DirectX 12, which was excused as the abstraction was not pertinent to this paper's subject.

Some changes were made compared to the original scene: the cube was swapped out for a Stanford Dragon model, taken from the Stanford 3D Scanning Repository[9], and the initialization of the vertex buffer for DirectX 12 was updated to use an additional upload buffer.

2. SETUP

CPU: AMD Ryzen 7 4800H 16 GB memory

GPU: Nvidia Geforce RTX 2060M 4GB VRAM

OS: Windows 11 Home

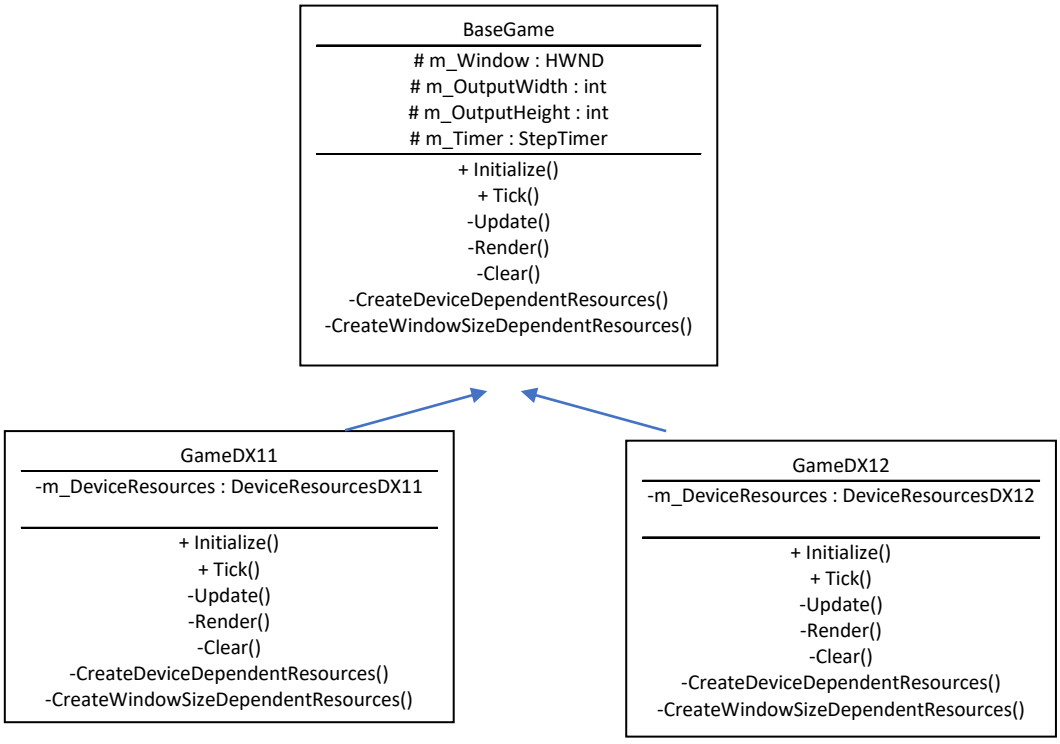
IDE: Visual Studio 2022 Enterprise

Additional debugging software: PIX for Windows (for DirectX 12 graphics debugging)

App window resolution: 800 x 600

3. PROJECT SETUP

This project uses the Win32 API with the "Direct3D12/11 Win32 Game DR" Visual Studio template downloaded[4] following the instructions on the DirectX TK GitHub page. One of the reasons this template was chosen, was that it came with a preconfigured Game class, implemented with Win32. Another reason this template was chosen, were the pre-initialized DirectX resources in the form of the DeviceResources. To facilitate future code comparison, a class structure was implemented as shown in the UML diagram below.



To separate the renderers from each other, a variable is stored in a global namespace. Using a menu toggle "Render Mode" you can select the desired render mode at any time, releasing the current API's resources, and initializing the selected ones.

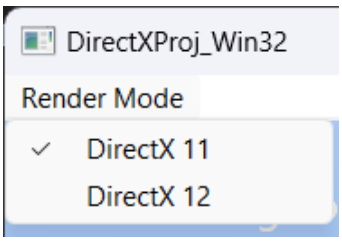


Figure 2. Render mode selection

Inside of the Device Resources files, all DirectX resources were stored inside ComPtr's instead of unique or shared pointers of the C++ Standard Library. As ComPtr's are designed to work with interfaces, not dynamically allocated memory[10,11], they are ideal when working with DirectX resources.

4. DIRECTX INITIALIZATION

While initializing DirectX, the first differences between the two version of the graphics API come to light. In the Device Resources classes, initialization of DirectX resources is split up between two different functions: *CreateDeviceResources* and *CreateWindowSizeDependantResources*. The device, device context, DXGI Factory and several other resources are initialized in the former. The swap chain, render target and depth stencil view are initialized in the latter, as the window is resizable, and so these resources would need to be recreated.

4.1. DIRECTX 11

In DirectX11, the first thing to do is setup the *DXGIFactory2*[12], then call *D3D11CreateDevice* to initialize the device[13] and device context[14]. These respectively represent the display adapter, used to create resources and to enumerate the capabilities of the display adapter, and the circumstance/setting in which the device is used. The next step is to create the swap chain and link the DXGIFactory to the application window. Then, the render target, depth stencil and depth stencil view are initialized. Lastly, the *D3D11_Viewport* is created so the render viewport is set to the entire window.

4.2. DIRECTX 12

For DirectX12, the *DXGIFactory2*[12] is first initialized with the *D3D12Device*[15] afterwards. However, before creating our swap chain, there are several other things that need to be initialized: the *Command Queue*[16], *Descriptor Heaps*[17] *Command Allocator*[16], *Command List*[16] and *Fence*[18]. These are explained in more detail below. The *Command List*[16] is left open for now, as it is needed for initialization of the buffers in the DirectX 12 game class. The swap chain is created as well as the render target views for each of the back buffers. The depth buffer is created as a *Resource*[19] with a default heap type, as a 2D resource.

The *Command Queue* and *Command List*[16] were created with the goals of enabling reuse of rendering work and of multi-threaded scaling. This differs from previous versions of DirectX in three ways. First, elimination of immediate context, enabling multithreading. Second, the application now owns how rendering calls are grouped into GPU work items, as it can now re-use items to save on performance. Last, the application now controls exactly when items are submitted to the GPU, enabling the first two items. To execute work on the GPU, the app must explicitly submit a command list to a command queue associated with the Direct3D device. Several commands can be pushed to the queue before it is executed and can also be reused in this way. But the application is responsible for ensuring that the direct command list has finished executing on the GPU before submitting it again.

The *Descriptor Heap*[17] contains object that aren't part of a Pipeline State Object (PSO), such as Shader Resource Views (SRV) and Constant Buffer Views (CBV) as well as Render Target View (RTV) and Depth Stencil View (DSV) as they are used in the app. They encompass the bulk of memory allocation required for storing the descriptor specifications of object types that shaders reference for as large of a window of rendering as possible. They mirror what most GPU hardware does and can therefore only be immediately edited on the CPU.

A *Fence*[18] is an object used to synchronize the CPU and one or more GPUs.

In DirectX 12 the explicit *Texture2D, RenderTargetView, DepthStencilView,* etc no longer exist. Instead, they are replaced with *Resource Barriers*[20]. Resource Barriers move the responsibility of per-resource state management from the graphics driver to the app. In DirectX 11, drivers tracked this state in the background, which significantly complicates any sort of multi-threaded design and is expensive on the CPU as well. This way, a texture resource can be accessed as Shader Resource View or Render Target View.

5. CLASS INITIALIZATION

As the project needs to be able to switch between DirectX 11 and DirectX 12 on-the-fly, an abstract base class containing all the necessary functions was defined. This class was then inherited from twice: once for a DirectX 11 version and once for a DirectX 12 version (which were named accordingly).

Inside the constructor for both classes the Device Resources are created, then, from the environment the class is constructed in, the *Initialize* function is called. Here, the Device Resources as well as some other objects that manage class resources are initialized.

Both classes have an array of custom *Instance* objects, and two arrays of *XMVECTOR* for rotation and velocities. The *Instance* struct contains two *XMFLOAT4* variables, one for rotation, and one for position and scale. Their use will be described in a later section.

5.1. DIRECTX 11

The DirectX 11 version has six buffers: a vertex buffer, index buffer, instance buffer, color buffer, vertex constant buffer and pixel constant buffer. It also has *ID3D11InputLayout*[21], *ID3D11VertexShader* and *ID3D11PixelShader* objects. In the *CreateDeviceDependentResources* function, the input layout is described, and the vertex and pixel shaders are read from their CSO files[22] (Compiled Shader Object). Shaders will be covered in more detail in a later section. Subsequently, data is read from the OBJ file and written into buffers. Vertex information is written into the vertex buffer, index information is written into index buffer (every vertex in the buffer is unique). The instance buffer is initialized with a nullptr, and the color buffer is initialized with random colors. Next, the two constant buffers are initialized for the vertex shader and pixel shader. All the other arrays get initialized as empty, at maximum capacity.

5.2. DIRECTX 12

The DirectX 12 version is quite a bit different in setup. It has two vertex buffers, two index buffers, two color buffers and 1 instance buffer. Information such as the location of the mapped instance data, the virtual GPU address of the instance buffer, and details about three *Vertex Buffer Views*[23] and one *Index Buffer View*[23] are also stored. The class also has a *Fence* object and a *FenceEvent* object. The Root Signature[24] and Pipeline State[25] are also stored as well as a *Graphics Memory* object (defined in DirectXTK). After initialization of the Device Resources, the *Root Signature*[24] is initialized. Then, the compiled vertex and pixel shaders are read, the input layout is defined,

and put into a *Pipeline State Object*[25] *(PSO)*. The *Root Sisgnature* is assigned to the PSO, as well as the data read from the vertex and pixel shaders, along with a wide array of other information. Next, just like in the DirectX 11 version, the data read from the OBJ file is written to the buffers, but the way this is done, is quite different. In DirectX 12, buffers are replaced by *Resources*, and have states that can be manually transitioned between by the application. To initialize buffers with the correct data in an efficient way, two buffers are needed: the buffer that will be sent to the GPU and an upload buffer[26]. Data is read into the upload buffer, and then copied to the actual buffer. This is done for the vertex buffer, index buffer and color buffer. For the instance buffer, *map*[27] is used to map a local variable to the buffer, so we can write modified instance data to the buffer. This is done to improve performance, as while it is possible to use only one upload buffer instead of two, this is not ideal as the data would need to be copied to the GPU every time it needs it. Without this optimization, the application had sub-par performance. The *Buffer Views*[23] are also initialized with the virtual GPU address of the buffer, size of the stride (one entry in the buffer), and the full size of the buffer. Afterwards, the *Command List*[16] is closed and executed, so the data is pushed onto the GPU.

The *Root Signature*[24] links command lists to the resources shaders require. As the application only uses one shader, only one Root Signature is necessary. It is possible to reuse the same *Root Signature* for different PSOs in the same way command lists are reusable. It defines the access a shader has to certain pipeline stages. In this project, access to the hull, domain and geometry shaders is denied in this way. The two constant buffers are also defined as such, one being visible[28,29] to the vertex shader, the other to the pixel shader.

The 2-buffer pattern consists of two buffers with two *Resource Barriers*: one that has default heap type[30], and *D3D12_RESOURCE_STATE_COPY_DEST* and another with an upload heap type and *D3D12_RESOURCE_STATE_GENERIC_READ*. The former is the buffer that gets pushed to the GPU, and the latter is the upload buffer used as an intermediary to copy data onto the actual buffer. Once the *Resources* are created, the object data is put into a *D3D12_SUBRESOURCE_DATA* object, copied into the upload buffer and then copied onto the actual buffer. Then the actual buffer's *D3D12_RESOURCE_STATE_COPY_DEST* state is set to transition to *D3D12_RESOURCE_STATE_INDEX_BUFFER* or *D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER*.

6. GAME LOOP

The project uses a normal Update-Render game loop, as provided in the used template.

In the game class' *Tick* function, the *Update* and *Render* functions are executed.

The *Update* function updates all currently rendered instances based on their velocity, and reflects them when they reach the border, so they don't fly away too far. In DirectX 11, the instance and constant buffer data are updated here. In DirectX 12 this happens later.

The *Render* function has quite a bit of differences as it uses the respective graphics APIs more than in the *Update* function. First, the back buffers are cleared, to ensure that no data from previous frames will be shown again. This happens in the *Clear* function.

6.1. DIRECTX 11

In DirectX 11, the *Render Target View* and *Depth Stencil View* are cleared, the *Render Target* is set, together with the *Viewport*[31]. When the back buffers are cleared, the GPU is prepared to render to upcoming frame. The *Input Layout*[21] is created, the *Primitive Topology*[32] is set to *D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST*, and the vertex buffers are initialized. Next, an array of *ID3D11Buffer* pointers is created with the vertex buffer, instance buffer

and color buffer. With a call to *IASetVertexBuffers*[33] the three vertex buffers are set at once. Then, with *IASetIndexBuffer*[34] the index buffer and format of the index buffer are set. The constant buffers and shaders are set next. When everything is configured, a call to *DrawIndexedInstanced* [35] draws all instances. Then, *Present* is called to render the new frame.

6.2. DIRECTX 12

In DirectX 12, before the back buffer is cleared, the fence is checked to see if the GPU is keeping up with the CPU[36]. After executing everything that was on a command queue, the fence signal is incremented (this would happen at the end of the *Render* function). This function is a nonblocking operation, so when executing it, the fence value might not get changed immediately. If the GPU has not caught up yet, the program waits for a single frame. If the GPU and CPU are in sync, the command list is prepared to render a new frame by resetting the command list as well as the allocator and changing the *Resource Barrier* from *D3D12_RESOURCE_STATE_PRESENT* to *D3D12_RESOURCE_STATE_RENDER_TARGET*. The back buffers are then cleared. There is not much different here when compared to DirectX 11. After setting the Root Signature and Pipeline State, the constant buffers are configured using DirectX ToolKit helper functions to copy data from the buffers using memcpy. Additionally, the Primitive Topology is set. Then, the 2nd vertex buffer is created (initialization of the instance buffer was skipped in the Game class' *CreateDeviceResources* function) by copying the instances to the address we earlier mapped out instance data to. With *IASetVertexBuffers*[37] the vertex buffers are sent to the GPU using the *Vertex Buffer Views*[23]. In DirectX 12 however, the stride information, size (and format for the *Index Buffer View*[23]) are stored inside a *Buffer View* struct, so there is no longer any need to pass that information along to the function directly, like it is with DirectX 11. After setting the Index and Vertex buffers, the scene is set to draw with a call to *DrawIndexedInstanced* [38], similar to DirectX 11. Before we return from the *Render* function, the *Command List* is closed, executed, and the scene is drawn.

7. GPU INSTANCING

Rendering a mesh multiple times is straightforward, as detailed above. This section provides a more detailed description on the shader configuration for this project.

In the *D3D11_INPUT_ELEMENT_DESC* and *D3D12_INPUT_ELEMENT_DESC*, certain entries in the input layout are reserved for the instance. From the OBJ file, vertex position and normal are read, which each have their own entry in the input layout. The other three entries are ones that can only be filled in by the instance themselves: rotation, position & scale, and color. They are easily distinguishable by their *INPUT_CLASSIFICATION*[39]. The *INPUT_CLASSIFICATION* describes if the data in the input is per-vertex data (data that is read from the vertex buffer), or per-instance data (data read from the instance buffer).

So, to properly construct an instanced vertex, it needs two separate inputs, which are unrelated to one another: the original data, and the modified data. Hence, the need for (at least) two vertex buffers, one for vertex data, and one for the instance data.

8. SHADERS

The shader language used for the shaders is HLSL[40]. Creating a shader in Visual Studio is not difficult, as it has built-in functions for it. The shader code was taken from the *XBoxUWPSamples*, the same project this project's code is based off. The shader code read in the *CreateDeviceDependentResources* function in the Game class isn't read

from the HLSL file, however, but from a CSO file. By including the shader in the project, when building the solution, the shaders automatically get compiled together with it, eliminating shader runtime compilation stutter.

9. PERFORMANCE COMPARISON

The previous section details the architectural differences, while this section will delve deeper into the actual performance differential between DirectX 11 and DirectX 12.

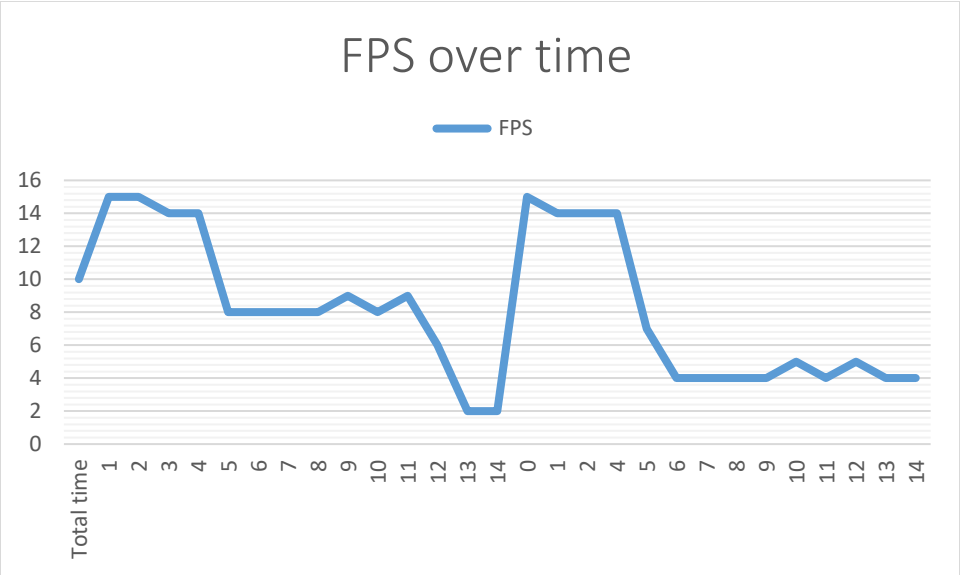


Figure 3. Chart showing FPS over time

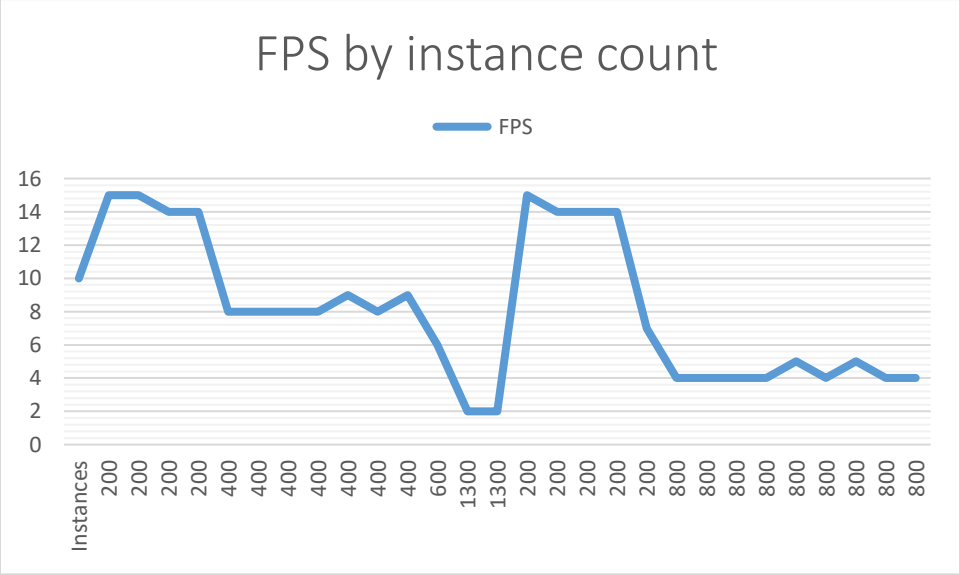


Figure 4. Chart showing FPS by instance count

The first graph details the FPS count (frames per second) on the y-axis, and the elapsed time for both DirectX 11 and 12 (0 to 14 seconds) on the x-axis. The second graph shows FPS count with the number of instances rendered at that time. The dip in framerate at the 13-14 second mark is due to switching between APIs. The graphs show there isn't really a difference in performance between the two versions. This means there is no inherent difference between the two versions of the graphics API, and any optimization needs to be implemented by the user.

DISCUSSION

While the changes between DirectX 11 and DirectX 12 are numerous, there are clear parallels between each step of the render pipeline. Generally, what takes a single step in DirectX 11 takes quite a bit more work to achieve the same result, facilitating side-by-side comparisons. The low-level nature of DirectX 12 raises the barrier of entry quite a lot for people looking to learn DirectX. The command list, descriptors, and PSOs are complicated to understand. However, using DirectX 11 can provide the same result in a simpler and more direct way. Low-level languages like this provide more control over how things are done, but also increases the potential for errors.

This project's usage of DirectX 12 does not compare to the way modern-day AAA games would utilize it. Therefore, it is not possible to determine what the biggest performance detriment is in video games. D3D11On12 might provide an interesting middle-ground when looking to port a graphics engine from DirectX 11 to DirectX 12, as Microsoft provides a guide on porting to DirectX 12[41].

This project is also quite unoptimized. It is single threaded, has an unoptimized index buffer and no memory management. The DirectX 12 scene uses around twice as much memory as the DirectX 11 version, and all mesh vertexes in the vertex/index buffers are unique, which further increases memory usage. This is not an optimal solution for eliminating runtime shader compilation stutter.

The implementation of pre-compiled shaders in this project is not optimal either. Because every computer's hardware configuration is different, shaders have to be compiled for each computer a game runs on. A rebuild of this project is required after deleting the CSO files to make them work on different hardware.

FUTURE WORK

This project uses basic rasterization functions and doesn't make use of many of the new features implemented in DirectX 12.

The Command List/Queue introduced in DirectX 12 allow for multithreading, something that is difficult to do in DirectX 11. The project currently uses no multithreading, so adding this would certainly improve performance. This optimization would be heavily based on Microsoft's own multithreading sample[42].

Memory management[43] could also be introduced in the DirectX 12 scene.

As mentioned above, this project's implementation of PSO's is suboptimal, so looking into a way to make shaders compile on first startup would help eliminate runtime shader compilation stutter[22].

A basic scene manager could also be introduced to increase the flexibility of the overall scene and necessitate a material/texture load system.

Other optimizations could be inspired by Microsoft's MiniEngine graphics engine[44].

BIBLIOGRAPHY

1: *Important Changes from Direct3D 11 to Direct3D 12 - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/important-changes-from-directx-11-to-directx-12#porting-from-direct3d-11

2: *Getting Started · microsoft/DirectXTK Wiki*. (n.d.). Retrieved from https://github.com/Microsoft/DirectXTK/wiki/Getting-Started

3: *Getting Started · Microsoft/DirectXTK12 Wiki · GitHub*. (n.d.). Retrieved from https://github.com/microsoft/DirectXTK12/wiki/Getting-Started

4: *DirectX Visual Studio Templates Wiki · GitHub*. (n.d.). Retrieved from https://github.com/walbourn/directx-vs-templates/wiki

5: *Using System-Generated Values - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-input-assembler-stage-using#instanceid

6: *C++ - Instancing with Directx11 - Game Development Stack Exchange*. (n.d.). Retrieved from https://gamedev.stackexchange.com/questions/170192/instancing-with-directx11

7: *Direct3D 11 on 12 - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/direct3d-11-on-12

8: *Xbox-ATG-Samples/UWPSamples/IntroGraphics at main · microsoft/Xbox-ATG-Samples*. (n.d.). Retrieved from https://github.com/microsoft/Xbox-ATG-Samples/tree/main/UWPSamples/IntroGraphics

9: *The Stanford 3D Scanning Repository*. (n.d.). Retrieved from http://graphics.stanford.edu/data/3Dscanrep/

10: *DirectX - ComPtr vs. C++ shared_ptr*. (n.d.). Retrieved from https://social.msdn.microsoft.com/Forums/SqlServer/en-US/5a376d34-f74b-4594-b87f-d070a3e00199/directx-comptr-vs-c-sharedptr?forum=windowsgeneraldevelopmentissues

11: *ComPtr Class | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/cpp/cppcx/wrl/comptr-class?view=msvc-170

12: *IDXGIFactory2 (dxgi1_2.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/dxgi1_2/nn-dxgi1_2-idxgifactory2

13: *ID3D11Device1 (d3d11_1.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d11_1/nn-d3d11_1-id3d11device1

14: *ID3D11DeviceContext1 (d3d11_1.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d11_1/nn-d3d11_1-id3d11devicecontext1

15: *ID3D12Device (d3d12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d12/nn-d3d12-id3d12device

16: *Design Philosophy of Command Queues and Command Lists - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/design-philosophy-of-command-queues-and-command-lists

17: *Descriptor Heaps Overview - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/descriptor-heaps-overview

18: *ID3D12Fence (d3d12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d12/nn-d3d12-id3d12fence

19: *ID3D12Resource (d3d12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d12/nn-d3d12-id3d12resource

20: *Using Resource Barriers to Synchronize Resource States in Direct3D 12 - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/using-resource-barriers-to-synchronize-resource-states-in-direct3d-12

21: *ID3D11InputLayout (d3d11.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d11/nn-d3d11-id3d11inputlayout

22: *Compiling Shaders - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-part1

23: *Resource binding overview - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/resource-binding-flow-of-control

24: *Root Signatures Overview - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/root-signatures-overview

25: *Managing Graphics Pipeline State in Direct3D 12 - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/managing-graphics-pipeline-state-in-direct3d-12

26: *Using Resource Barriers to Synchronize Resource States in Direct3D 12 - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/using-resource-barriers-to-synchronize-resource-states-in-direct3d-12

27: *ID3D12Resource::Map (d3d12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12resource-map

28: *CD3DX12_ROOT_PARAMETER structure (D3dx12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/cd3dx12-root-parameter

29: *D3D12_SHADER_VISIBILITY (d3d12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d12/ne-d3d12-d3d12_shader_visibility

30: *D3D12_HEAP_TYPE (d3d12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d12/ne-d3d12-d3d12_heap_type

31: *D3D11_VIEWPORT (d3d11.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d11/ns-d3d11-d3d11_viewport

32: *Primitive Topologies - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-primitive-topologies

33: *ID3D11DeviceContext::IASetVertexBuffers (d3d11.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11devicecontext-iasetvertexbuffers

34: *ID3D11DeviceContext::IASetIndexBuffer (d3d11.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11devicecontext-iasetindexbuffer

35: *ID3D11DeviceContext::IASetIndexBuffer (d3d11.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11devicecontext-iasetindexbuffer

36: *ID3D11DeviceContext::DrawIndexedInstanced (d3d11.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11devicecontext-drawindexedinstanced

37: *ID3D12GraphicsCommandList::IASetVertexBuffers (d3d12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12graphicscommandlist-iasetvertexbuffers

38: *ID3D12GraphicsCommandList::DrawIndexedInstanced (d3d12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12graphicscommandlist-drawindexedinstanced

39: *D3D12_INPUT_ELEMENT_DESC (d3d12.h) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/d3d12/ns-d3d12-d3d12_input_element_desc

40: *High-level shader language (HLSL) - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl

41: *Porting from Direct3D 11 to Direct3D 12 - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/porting-from-direct3d-11-to-direct3d-12#submitting-work-to-the-gpu

42: *Direct3D 12 multithreading sample - Code Samples | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/samples/microsoft/directx-graphics-samples/d3d12-multithreading-sample-win32/

43: *Memory Management in Direct3D 12 - Win32 apps | Microsoft Learn*. (n.d.). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d12/memory-management

44: *microsoft/DirectX-Graphics-Samples: This repo contains the DirectX Graphics samples that demonstrate how to build graphics intensive applications on Windows.* (n.d.). Retrieved from https://github.com/microsoft/DirectX-Graphics-Samples

APPENDICES

You can find my project code on GitHub, linked here: https://github.com/Vincent-VD/DX11vDX12

Parts of this paper were written with help of ChatGPT: https://openai.com/blog/chatgpt/