Master of Science in Engineering: Game and Software Engineering
Joakim Sjöberg
Filip Zachrisson

3D Programming 3:

Project

- A performance test of the multi engine in DX12

DV2551
3D-programming 3

1 Introduction

We were interested in investigating the multi engine of DX12. DX11 only has one type of queue, were DX12 got three different types of queues.

These queues can work in parallel of each other unless the data is dependent on each other (which it is in our case). The copy queue is used for cases were the only job is to send data from CPU→GPU. The compute queue can do all that a copy queue can and more such as "dispatch" to work with compute shaders. The direct queue can do everything the other queues can and more such as "draw" to work with vertex and pixel shaders.

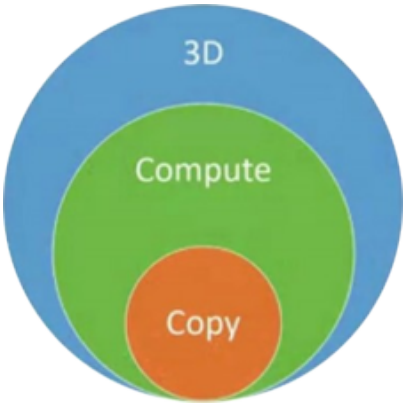

Figure 1: Multi engine

We made a test were we copy a vector with *colorData* with the copy queue, then modify the color on the GPU-side using a compute queue, and finally draw two objects with a direct queue were one of the objects is using the color from the earlier stages.

2 Implementation Methodology

We decided early on that we wanted to create a mini engine in this course which we can use as a starting point in our later courses such as "big game project" in the last year of our education. Our engine is still work in progress of course, but now it will work as a solid foundation in before future courses as well.

2.1 Implementation Specifics

The following section will explain a few details about how we used DX12 in our engine.

2.1.1 Fences

We use one fence in our application, were we increment it three times each frame (once for each queue). Since we need the copy queue to finish before the compute queue starts executing its command lists, the copy queue signals when its finished copying the data, and only then the compute queue will start executing it's command lists. The same procedure happens between the compute queue and the direct queue, and finally the copy queue has to wait for the direct queue to finish before accessing its resource.
Also, all our command lists and command allocators and double buffered, which means that the CPU can start recording command lists one frame ahead of were the GPU is currently at.

2.1.2 Multi-threading

In the engine we implemented an abstract thread-pool that is used mainly for recording command lists, but could later on be used by the game developers with their own functionality. Currently the benefits from multi-threading is pretty small, since we only have two tasks in our project(Forward Rendering and Blend).

2.2 Engine features

The engine has support for creating objects and choosing *DrawOptions* how to draw them. *RenderTask* determines how an object is rendered. The only two implemented *RenderTask* so far are "forward rendering" and "blend", both of them are used in the project. But it is easy to add another task and. Future work could be to implement more *RenderTask*, such as *ShadowTask*, *BloomTask* etc.
The engine is abstract from future work such as creating a game, which means that people without any knowledge of DX12 can still work with our engine and create a game using it.

3 Results

3.1 Specs

- i7-6700K @4.6GHz (4-Core + Hyper-threading)
- GTX 960 @1280/3500/2560MHz (Core/Memory/Shader)
- 2x Corsair DDR4 8GB @1066MHz

3.2 Benchmarking

When we benchmarked our application, we measured all command list recordings of each task, as well as the sum of them. The following tabular is our result when measuring the time it takes to execute each task on the GPU.

Average of 10000 Frames (ms)		
Task	Different Queues	Direct Queue Only
Copy Color	0.001747	0.003857
Compute	0.000151	0.004703
Forward	0.098126	0.111821
Blend	0.016185	0.018538
SUM	0.116209	0.138919

4 Analysis

The result from the benchmark shows us that it was faster to use all the queues instead of just the direct queue, our reasoning being that the GPU can do certain things in parallel this way. The test was very simple and the benchmark can differ when using different test benches then the one we used.

5 Conclusions

We both worked the same amount, often together. The grade we aimed for is A and we accomplished this by first implementing E and then moving up towards A. The correlations between our specific implementations are:

- D - WaitForFrame();
- C - ThreadPool $\rightarrow$ AddTask();
- B - CopyColorTask
- A - ComputeTestTask

5.1 Future Work

- Implementing flags in Thread to check if all tasks with a certain flag is done.
- Double buffering resources to be able to execute the copy and compute queue before the last direct queue is executed.
- More tasks, for example shadows, post processing effects ...