Code Sample: Parallel Processing with Direct3D 12

File(s): Download the Code Sample

License: Intel Sample Source Code License Agreement

Optimized for...

<table>
<thead>
<tr>
<th>OS:</th>
<th>Microsoft Windows 10 (64 bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware:</td>
<td>GPU required</td>
</tr>
<tr>
<td>Software:</td>
<td>Microsoft Visual Studio 2017, Direct3D 12, C++</td>
</tr>
<tr>
<td>Prerequisites:</td>
<td>Familiarity with Visual Studio, Direct3D API, 3D graphics, parallel processing.</td>
</tr>
</tbody>
</table>

Tutorial Article

Parallel Processing with DirectX 3D* 12

Introduction

The idea behind this project was to provide a demonstration of parallel processing in gaming with Direct3D 12. It expands upon the results from the paper “Comparison of Intel i5-Core and i7-Core systems with Visualizations in OpenGL and VR/Oculus” (see References section) and extends the code there to contain a Direct3D12 renderer. It also re-implements the previous particle system as a Direct3D 12 compute shader.

1) Modify code to add a CPU Direct3D 12 renderer
2) Moving to the GPU
3) Closer look at differences between CPU and GPU

Get Started

Modify code to add a CPU Direct3D 12 renderer

The first task is to add a Direct3D 12 “renderer” to the particle system used in the i5-vs-i7 article. The software design makes this very easy to do since it very nicely encapsulates the concept of rendering. The first step is to define an interface to the renderer and then write an event loop. To improve performance, I wrote a custom upload heap. Next I looked at the compute shader and the actual Direct3D 12 rendering code, before discussing issues surrounding the vertex buffer view.

Moving to the GPU

We can improve performance by moving the renderer from the CPU to the GPU. Better than separating processing between separate threads, is to separate it between multiple processors: CPU and GPU.

The first thing I considered after struggling with getting every single field in every single structure in the CPU portion to be correct and consistent, was facing the prospect of two to three times more work for the GPU compute problem. I quickly decided the right thing to do was to look for some kind of “helper” framework and, thus, chose MiniEngine: A DirectX 12 Engine Starter Kit. I will cover how I installed and customized MiniEngine for this project. Through the use of MiniEngine, the 500+ lines of code to render using the CPU is reduced to about 38 lines of setup code and 31 lines of rendering code (69 lines total) for the GPU, so my work paid off.

The GPU renderer consists of setup and rendering code. Setup contains configuring the root signature, vertex inputs and obtain the formats used for color and depth. Finally, I configure the graphic PSO and the view and production matrices. Rendering code is broken into obtaining the context, describing the transitions, clearing the color and depth before updating the matrix with the new values and then drawing the frame.

Closer look at differences between CPU and GPU
For best performance I use two buffers of particle data and render one while the other is being updated for the next frame by GPU compute. I briefly talk about this before taking a deeper look at changes required to implement a particle rendering system on the GPU, in particular, differences between the algorithms.

References


Updated Log

Created March 20, 2018

Appendix 1:

N/A