Master's Project
Practical Uses of Intel's Iris Extensions

Bryan Pawlowski
Computer Grapgics, Oregon State University

December 27, 2014

Contents

1 Introduction

Since the advent of computer graphics, engineers have been trying to find new ways to create graphics hardware solutions intended to yield higher performance and image quality. However, image quality can mean many things to many people. Individuals may use grapchics hardware to create vibrant and abstract experiences, where others may create an experience where the images being created are as photorealistic as possible. The question that graphics hardware engineers must ask, is, "What can we make that will help developers create a higher quality experience?" For Intel, their answer lies within the Haswell processor, with the introduction of what they have dubbed as "Iris Graphics Pro."

2 Background

Before we get into the gory details of what my project is all about, there are some key concepts that must be covered. This project utilized a lot of different concepts withint Computer Science outside of the realm of graphics. We first need to cover what the hardware is and why it is different, then take a general look at the graphics pipeline, and finally discuss two key topics in parallel programming. My goal is to make sure my rationale for this project is completely clear.

2.1 What is Iris Pro?

Iris Pro is the name for Intel's integrated graphics solutions that shipped with the high end of their Haswell product line. This iteration, Iris Pro included two hardware extensions that provided capabilities that haven't really been offered to graphics programmers in the past, and are meant to increase performance of realtime graphics applications on high-end Intel chips.

2.1.1 Shared Memory

One extension, the shared memory extension, is self explanatory. The CPU on which my project was run, the core i7 4750HQ, was actually manufactured as a FPGA, where there was a top layer added on top of the chip for 128MB of L4 Cache. The purpose of this new, particularly large cache level was to create a memory bridge between the CPU and the GPU sides of the chip. As operating systems are not used to this type of an architecture for graphics programming, there did not exist a streamlined API for using this feature at the time of my project. There are techniques and procedures to utilize this extension, but my project focuses on the other extension, Pixel Synchronization.

2.1.2 Pixel Synchronization

With the Pixel Synchronization extension, the pipeline is now capable of creating a barrier during the pixel shader stage by pixel, where multiple instances of the pixel being rendered will happen in-order and not in parallel, based on the primitive number. Synchronizations capabilities are new to the realtime graphics world and as with any new technologies, the potential is far from being tapped to its fullest.

2.2 The Graphics Pipeline

In computer graphics, objects are described mathematically as a group of dots, then these dots are given to the graphics hardware, with information on how to conntect all of the dots, how to fill in the space between the dots, etc. 3D vertices on into the pipeline, and a pixel with a color comes out of the other end, basically. The two main steps of the pipline are the vertex shader, and the pixel/fragment shader.

2.2.1 The Vertex Shader

The Vertex shader is the area of entry for every 3D point. Within the vertex shader, the GPU generally stores key information and properties each vertex has, before passing it through the pipeline for these values to be interpolated. The purpose of this, is that it greatly reduces the amount of space needed to express the looks and properties of an object. If we can describe the key pieces of information at a few points on the object, the idea is that the GPU will be able to make informed assumptions about the other parts of the model where we have provided no information.

2.2.2 The Pixel/Fragment Shader

Depending on Direct 3D or OpenGL, this stage of the pipeline is either called the Pixel Shader, or the Fragment Shader, respectively. Prior to the Pixel Shader stage, the 3D object has been transformed, and the object's properties have been interpolated across its entire surface. After this interpolation takes place, the GPU then decides where in screen the object is, then runs the pixel shader to color those specific pixels. Most rendering techniques do most of their lighting effects and color computations at this stage of the pipeline to help create a smoother coloring of the model. In a traditional pipeline, the pixels are evluated in parallel, making rendering a potentially fast process. The tradeoff, however, is that with increased rendering speed comes a lack of knowledge of other parts of the object being evaluated. The more naive the pixel is about its neighbors or its surroundings, the quicker the computation will finish, and the final image displayed.

2.3 Parallel Programming

Pixel synchronization and my implementation of these different render techniques borrow concepts from parallel computing. As vertices are passed through the pipeline and on to the pixel shader, these parts of the pipeline are actually happening in parallel for all of the different instances of vertices and pixels. The problem with doing anything depth-based in realtime graphics, is that each fragment or potential pixel doesn't really have any knowledge of any of the neighboring pixels, they do not execute in any specific order, either. In order to know if one potential pixel is deeper than another potential pixel, we need a predictable way of determining this. If we were to naively try to gather this information within the pixel shader, we would run in to a race condition. We also need a way to share this information across different instances of the pixel shader.

2.3.1 Shared Resources

To determine the depth of a certain spot of a 3D model within the pipeline, we need some sort of a way for the related potential pixels to communicate with each other. To do this, we use a feature of Direct3D 11 called Unordered Access Views. Unordered Access Views (UAVs) are basically a read/write texture that can only be accessed within the GPU, during the Pixel Shader stage. UAVs can be used in either a 1 or 2 dimensional array. The size of the UAV is specified to the graphics device before rendering begins. Within the shader, the programmer must take care to list the UAVs in the same order as they are initialized within the CPU side of the application. Once properly initialized, the UAVs are identified within the shader as RWTexture2Ds.

2.3.2 Synchronization and Barriers

Oftentimes, in parallel programs, we must make sure our threads (or in the case of this project, pixels) are synchronized and executed in some sort of order. The Pixel Synchronization extension can be likened to a barrier in parallel programming, however, there is an extra step before the pixels are released to execute one-at-a-time. The pixels in flight are only synchronized across other instances of the pixel shader that are executing at that same [X, Y] coordinate. All pixels executing at that coordinate are gathered and then ordered by primitive number, then execute with a priority of lowest primitive number to highest. I hypothesize that a feature may pop in up in the future where the programmer may decide whether pixels are ordered in increasing or decreasing order. This would be useful, for instance, if a 3D object is facing away from the camera (rather, facing INTO the scene). This gives the programmer the power of ordering the execution in a front-to-back or back-to- front manner, which would greatly simplify depth-based render techniques.

2.4 The Iris Connection

2.5 Rationale

3 Subsurface Scattering

3.1 Previous Implementations

3.1.1 Wrapping Approximation

3.1.2 Depth Mapping

3.1.3 Texture-Space Diffusion

3.2 Adding Iris Extensions

4 Refraction

4.1 ”Fake” Convex Object Refraction

4.2 Better Convex Refraction With Iris

5 Single-Bounce Raytrace/Hit-Detection

6 Conclusions and Future Work