Intel Graphics Performance Workshop
for 3rd Generation Intel® Core™ Processor (Ivy Bridge)

Introduction

The Intel Graphics Performance Workshop presents a tutorial illustrating the use of the Intel® Graphics Performance Analyzer (GPA) tool suite to guide rendering performance optimization of a D3D11 graphics application. The workshop materials consist of a representative D3D11 sample application “City Racer” and an analysis and optimization tutorial suitable for a classroom presentation or individual developer consumption.

City Racer Sample Application

The sample application simulates a road race of hovercraft vehicles through a stylized city setting. The combined city and vehicle geometry consists of approximately 4.4M vertices (1.5M triangles) per frame, with diffuse- and specular-mapped materials lit by a single directional (sun) light. Scene depth is emphasized with an aerial perspective fog effect, and in-vehicle views are rendered with a velocity-sensitive motion blur. Six incremental performance optimizations are included in the source, selectively
included via conditional compilation directives, to increase overall rendering performance in steps corresponding to optimizations identified in the tutorial.

**Optimization Tutorial**

The initial state of the sample code is sub-optimal, including several design choices that constrain overall rendering performance. The tutorial leads a developer through six successive optimization steps. These six steps are:

1. Render target bit depth reduction
2. Switch to accumulation buffer blur technique
3. Reduce blur buffer resolution
4. Eliminate unnecessary clears
5. Add frustum culling
6. Switch to GPU instancing

At each step the application is analyzed with the Intel Graphics Performance Analyzer (GPA) suite to identify a performance bottleneck. An appropriate optimization is then applied to the sample code to overcome the bottleneck, and Intel GPA is re-run to measure the performance gained. The optimizations applied are generally in line with guidelines provided in *DirectX* Developer’s Guide for Intel® Processor Graphics - Maximizing Graphics Performance on the New Intel® Microarchitecture Codenamed Ivy Bridge. Over the course of the tutorial a variety of optimizations are applied that cumulatively improve the rendered frames-per-second (fps) roughly twofold.

**City Racer Sample Application**

**Prerequisites**

The City Racer sample is built with Microsoft Visual Studio* 2008 or later, with solution files provided for 2008 and 2010 versions. City Racer utilizes the Direct3D 11 API and the DXUT framework, requiring that the Microsoft DirectX* SDK be installed. Additionally, profiling builds include markup commands provided by Intel’s common ittnotify.h header file installed with Intel GPA v4.2 or later.

**Sample Design**

City Racer is logically divided into race simulation and rendering subcomponents. Race simulation includes modeling of vehicle acceleration, braking and turning parameters and steering AI for track following, and collision avoidance. Race simulation code is in the track.cpp, vehicle.cpp, and SceneDescription.cpp files. Race simulation is not affected by any of the optimizations applied over the course of the tutorial.

The rendering component consists of drawing the vehicles and scene geometry using the D3D11 API and the DXUT framework. This code is entirely contained within main.cpp and is the target for all performance optimizations. The initial version of the rendering code is intended to represent a first-pass
effort, containing several performance-limiting design choices. Effort was made to include such initial choices that might be naïve or overkill, but not silly, e.g., over-specifying the render target bit depth, but not individual per-primitive Draw() calls.

Mesh and texture assets are loaded while parsing a `scene_description.txt` file commonly used in Intel samples, slightly modified for this sample’s particular requirements. Individual meshes are tagged as either pre-placed scenery items, instanced scenery with per-instance transformation data, or vehicles for which the simulation provides transformation data. Two camera modes are available, one free and one vehicle-locked. When the vehicle camera is active, a second render pass is added to produce a radial motion blur effect. All optimizations are targeted at the blurred vehicle camera mode.

On sample start-up, the scenery data is loaded, the simulation is initialized, and simulation step and rendering are interleaved on a single thread. The application is entirely GPU-bound, so no effort was made to thread the CPU code. On the contrary, for the optimization that specifically targets CPU performance a synthetic load is also inserted into the single CPU thread to force rendering to be CPU-bound to better demonstrate the performance gain.

**Optimization Tutorial**

The optimization tutorial is intended to be equally useful in either a classroom presentation or done individually. The tutorial walks through six optimization steps that improve rendering performance without negatively impacting visual quality. For each step, Intel GPA is used to capture and analyze a ‘before’ frame and/or trace, then a code change is made and Intel GPA is run again to verify that the change successfully addressed the performance problem.

Performance numbers reported are from a ‘Chief River’ SDP system using a 3rd generation Intel® Core ™ processor code-named Ivy Bridge, a mobile CPU with onboard GT2-level GPU. The operating system is Windows 7*, 64-bit, with 4 GB of DRAM installed. FPS numbers are taken on a 1920x1200 monitor in full-screen mode, i.e., at slightly greater than 1080p resolution.

**Optimization Step 1**

Initial performance of vehicle-camera rendering is around 50 ms/frame (20 fps). Intel GPA trace analysis finds that the application is bottlenecked by the GPU (this remains true until the final optimization step). GPU frame analysis finds that the final blur pass consumes nearly 28 ms, with Sampler Busy at 100% and EU idle at around 70%. The scene is therefore heavily texture bandwidth bound in the GPU.
The sample uses a floating-point render target, initially set to a 128 bit-per-texel depth format DXGI_FORMAT_R32G32B32A32_FLOAT. Changing to DXGI_FORMAT_R8G8B8A8_UNORM reduces the bit depth and corresponding texture bandwidth by a factor of 4. After making this change, frame time improves to 25 ms (40 fps).

**Optimization Step 2**

Step 2 again focuses on reducing texture bandwidth. The initial blur algorithm samples the render target texture along a radially aligned line segment with the number of samples varying from 1 (non-blurred) to as many as 25 samples for the extreme high-speed, small-depth-value corner pixel. Although the blur pass time has been reduced to around 7.5 ms, the GPU is still texture bandwidth bound.

An alternative blur algorithm is implemented that continually blends the scene into a traditional motion-blur-style accumulation buffer. This adds a pass to update the accumulation buffer, but the final blur blend then requires only three texture samples (2 color, 1 depth) per output pixel, blending between the full-resolution render target and the blurred buffer. Implementing this change improves overall rendering performance to 21 ms (48 fps).

**Optimization Step 3**

At step 3 the full blur pass (now actually 2 passes, the accumulation pass and the final blur blend) has been reduced to 3.5 ms, and the GPU is still texture bandwidth bound.
At this step we realize that for the blurred image stored in the accumulation buffer, a full-screen-resolution image is overkill. The blur pass render targets are adjusted to down-sample to a quarter-resolution buffer during the accumulation pass, and then blended back into a full-resolution buffer in the final pass. This further reduces blur-pass bandwidth and overall blur time to just under 2 ms, yielding an overall frame time of 19.6 ms (51 fps).

Optimization Step 4

An initial step in a typical D3D application is to clear the render target and back buffer color and depth data at the start of each frame. This clear operation is small but not free, consuming just over 1 ms on the GPU.

This application is set between a plane and a hemispherical skybox and does not alpha blend, so we know that every pixel in the scene will be rendered opaquely in every frame. This allows us to skip clearing the color buffers, clearing only depth data (100 us). This small adjustment saves about 1 ms per frame, bumping the rendering speed to 18.7 ms (53 fps).

Optimization Step 5
The city scene includes a large number of identical traffic barriers, each containing over 1300 primitives. Drawing these barriers consumes over 9 ms of GPU time, even though most are clipped and do not contribute to the final scene. We now add a view frustum culling step to avoid submitting barrier instances not within the camera’s view. The effect of this optimization is variable depending on camera orientation, but for the frame captured here it reduces the barrier drawing time to approximately 2 ms, resulting in an overall frame time of 14 ms (71 fps).

**Optimization Step 6**

The final optimization step illustrates a best practice for rendering, but its primary improvement is to reduce workload on the CPU. Because the application at this point is still GPU-bound, the performance gain at this point is barely visible in the final frame rate. To make it more visible, we introduce a synthetic load on the rendering thread that consumes 2/3 of CPU cycles. With this modification the application becomes heavily CPU-bound, and total frame time is 24 fps.

![Figure 6 - Road barriers consume over 9 ms, even though 90% of the primitives are clipped.](image)

![Figure 7 - CPU-bound application, with 660 instances drawn individually, consuming over 5 ms, or roughly half the CPU draw time.](image)

Until this point the application has used the SDKmesh-provided Render() call to do shader resource setup and Draw() submissions. For instanced data, the individual transformations are loaded into a constant buffer and Render() is called iteratively for each instance. In Figure 5, the Intel GPA System...
Analyzer identifies these calls within the ‘Instanced Model’ tag consuming over 5 ms or over half the overall CPU rendering time.

D3D11 provides a DrawInstanced() API call, but SDKmesh does not provide a means to take advantage of it. The optimization presented in this step extracts the individual vertex buffers and texture resources from the SDKmesh container and draws all 660 instances using a single DrawInstanced() call.

![Figure 8 - Greatly expanded view of CPU rendering time, after GPU instancing has been implemented. 'Instanced Model' time is barely visible as a cyan line near the left end of the rendering thread, now consuming only .03 ms](image)

With GPU instancing implemented, the CPU time required to draw all 660 instances is reduced to 30 us, now barely visible in the CPU rendering timeline. Overall frame time is reduced 28 ms (36 fps).

Although this step is somewhat contrived to exaggerate the effect of GPU instancing, it does effectively demonstrate the possible CPU benefits, reducing 5 ms of setup and draw time to effectively zero. Actual games will undoubtedly have higher CPU loads than this relatively simple sample and will benefit from instancing without such contrived loads.

**Further Optimizations**

The tutorial explores only a few of the optimization opportunities—many more remain. For example, there are many repeated objects in the scene (street lamps, intersection markings, some buildings), but only the most expensive object (road barriers) are culled and instanced during the tutorial. By targeting just the most expensive object we demonstrate the technique and the performance gain, and leave some performance optimizations for the user to recover in a hands-on, “see one, do one” style workshop.

Another obvious optimization would be a geometry Level of Detail (LOD) system, replacing a highly detailed model in the foreground with lower resolution versions in the distance. This could be a significant performance improvement, but would require large-scale code changes and multiple models per object. The required changes are beyond the scope of this workshop, but may be included in future versions of the workshop.

The DXUT library and SDKmesh container classes are used in the sample for expediency and clarity of the rendering code, but they are not highly optimized for rendering performance. These components could be replaced with custom performance-optimized libraries—but again, such a major undertaking is beyond the intended scope of this tutorial. Future versions of this workshop may utilize in-house replacements that may be more amenable to performance tweaking.
Conclusion

This workshop has taken a representative application and used the Intel Graphic Performance Analyzer tool suite to examine the application behavior and make targeted changes to improve performance. The changes made and improvements realized were:

1 – Render Target Bit Depth       50 ms -> 25 ms (100%)
2 – Accumulation Buffer Blur      25 ms -> 21 ms (19%)
3 – Blur Buffer Resolution        21 ms -> 19 ms (9%)
4 – Eliminate clears              19 ms -> 18 ms (5%)
5 – Frustum Culling               18 ms -> 14 ms (28%)
Overall GPU-centric Optimization  50 ms -> 14 ms (257%)

6 – GPU Instancing (not cumulative) 41 ms -> 28 ms (42%)

In summary, from the initial implementation to the best optimized version, we demonstrate rendering performance improvement from 20 fps to 71 fps, for a total gain of 257%. Since this sample is a known sub-optimal implementation to start with, a developer applying knowledge gained from this tutorial may not likely see the same absolute performance gain. Nevertheless, the primary goal of the sample is not to demonstrate the performance of this specific sample application, but to educate developers about the potential performance gains to be found in the recommendations in the DirectX* Developer’s Guide for Intel® Processor Graphics and to illustrate the usefulness of the Intel GPA tool suite in finding those improvements.