Optimizing XuanYuan with Intel® Graphics Performance Analyzer

on Intel® HD Graphics

Intel Corporation: Cage Lu, Kiefer Kuah  Tencent Corporation: Tim Cong

Abstract

Engineers from Intel and Tencent Corporations worked collaboratively to improve the performance of XuanYuan by 1.34x on test PCs with Intel® HD 3000 Graphics. The team used the Intel® Graphics Performance Analyzer (GPA) to analyze a representative scene and find bottlenecks. Improvements were made to shadow map generation, skybox culling, terrain rendering, UI image caching, and atlasing smaller textures. All these changes increased the frame rate from 53 frames per second (fps) to 71 fps providing an exceptional gaming experience. The performance gains from these optimizations will propagate to systems with the new 3rd generation Intel® Core™ processors, codenamed Ivy Bridge. The frame rate on the test Ivy Bridge system improved by 1.30x from 66 fps to 86 fps.

Introduction

Intel HD graphics have continued to improve 3D performance with many newly released games seeing significant improvement. Starting with the 2nd generation Intel® Core™ processor family, codename Sandy Bridge, Intel HD Graphics are now part of the same silicon as the CPU. In other words, new Intel Core processors have graphics processors built in. According to a report by Jon Peddie Research, 60% of the graphics processors shipped in the third quarter of 2011 were Intel HD graphics. To capture this significant install base, we encourage all game developers to optimize their games for Intel HD Graphics.

Intel is the known leader in performance analysis tools for CPUs and multi-threading. Intel® GPA continues this tradition, providing a robust and powerful solution for GPU performance analysis. Intel GPA tool helps game developers analyze, evaluate, and optimize performance. Intel GPA consists of three components: the GPA System Analyzer Heads-up Display (HUD), the GPA Frame Analyzer, and the GPA Platform Analyzer. The GPA System Analyzer HUD provides real-time performance monitoring at the system level. The GPA Frame Analyzer dives into the details of a captured game frame. The GPA Platform Analyzer visualizes the execution profile of the tasks in your game over time. Intel GPA can help you find hotspots and optimize your game for better performance.

Tencent is the largest online game company in China, and XuanYuan is a Chinese language MMORPG developed by Tencent Aurora Studio and released in 2011. In this paper, we describe some of the ways we used Intel GPA to find and implement several optimization opportunities. The game frame rate improved from 53 fps before optimization to 71 fps after optimization on our test hardware, an improvement of 1.34x.
The analysis and optimization were done on a PC with a 2nd generation Intel Core i7 processor, codename Sandy Bridge, with Intel HD Graphics 3000. This version of Intel HD Graphics was launched in 2011, and its performance is comparable to some mainstream discrete graphics cards. Tencent wanted XuanYuan to be playable on a broad range of PCs. Even though the game was already running well on Intel HD Graphics 3000, we wanted to further increase the speed of the game.

We started by selecting a scene from the game that is representative of the actual game play (Figure 1). This scene was outside of ShangYang city and had about 15 NPCs visible at once. The initial frame rate of this scene was 53 fps. Our goal was to raise it to >60 fps without significantly reducing the visual appeal of the scene.

![Figure 1. The scene from XuanYuan used in the performance analysis.](image)

**System Level Analysis**

The GPA System Analyzer Heads-up Display (HUD) outputs system level metrics in real time while the game is running. It supports dozens of performance counters on the CPU, DX runtime, and GPU. And it also supports experimentation via Direct3D pipeline state overrides, which are useful in locating bottlenecks in the game.

Using the System Analyzer HUD, we found that:

1. Most of the computation work was being performed on one main thread. What this meant was that the CPU could potentially become a bottleneck if too much work was given to that one thread. This could happen, for example, if the scene was populated with hundreds of characters
because character animation would all be done on the main thread. Dividing the work up into multiple threads eases this potential bottleneck as we proved by multithreading a scene with hundreds of characters for approximately a 2X improvement. We do not describe those changes in this paper as the scene we used in the analysis for this paper did not have hundreds of characters.

2. The number of state changes was ~6k per frame or an average of 7.5 times per draw call. This was an abnormally high ratio and suggested that draw calls were not sorted for efficient state management. Reducing the number of state changes could benefit performance.

3. With the Null Hardware override, the game reached 100 fps. With the Disable Draw Calls override, the frame rate was also at 100 fps. By seeing what the frame rate would be when these overrides were in place showed us what the game could do if the overhead from GPU and the driver, respectively, were not factors, i.e., the GPU and driver had infinite capability. The significant increase in frame rate from 53 fps to 100 fps with the Null Hardware override suggested that the bottleneck was in the GPU. As a comparison, we also tested a crowded scene that had 200 characters. No significant frame rate changes were noticed with the Null Hardware override. This lack of change indicated that the CPU was the bottleneck for this artificial scene.

Frame Analysis

The GPA Frame Analyzer zooms in on the details of a captured frame. It outputs GPU metrics and DirectX metrics and states, such as textures and shaders, of every draw call in the captured frame. Figure 2 shows a screen shot from the GPA Frame Analyzer of a captured frame from XuanYuan.

![Figure 2. The analysis of a captured frame from XuanYuan loaded in GPA Frame Analyzer.](image-url)

From the Frame Analyzer, we obtained a breakdown of the time consumed in the frame.

1. Shadow map generation: ~170 DrawPrimitive calls, 15.9% frame time. All of the objects in the scene had real-time shadow. We could substitute real-time shadows with static shadow for some of the objects with minimal impact to visual quality for a significant performance increase.

2. Skybox: 3 draw calls, 4.7% frame time. The skybox was not visible in this scene. We can do
visibility culling before drawing it. Another possible optimization was to render the skybox after all other opaque scene geometries were drawn to take advantage of Early-Z culling.

3. Terrain: 50 draw calls, 34.9% frame time. Like most other online games, drawing the terrain took up significant time and was an important target for optimization.

4. Characters and other objects on the ground: ~240 draw calls, 22.4% frame time. No obvious optimization opportunity was found.

5. Post processing: 8 draw calls, 8.4% frame time. No obvious optimization opportunity was found.

6. UI: 55 draw calls, 7.4% frame time. Many of the elements in the UI could be merged into a smaller number of draw calls to reduce the time spent displaying them.

**Optimization Strategy and Results**

1. **Shadow map generation**
   The game generated real-time shadow maps for all of the objects in the scene through ~170 draw calls. But displaying real-time shadows is not necessary for certain objects like terrain and distant objects. Off-line rendered static shadows could be used instead. This would save the number of draw calls and improve performance.

   Figure 3 shows the number of calls for the shadow map generation before and after optimizing. The screen shot on the left shows that the original shadow map code had 176 draw calls. After optimization, generating the shadow map required only 45 draw calls, as shown in the screen shot on the right. The time consumed by shadow map generation was reduced from 2.97 ms to 1.03 ms.

![Figure 3. Before and after optimizing shadow map generation.](image-url)
2. Skybox
The skybox was not visible in this game scene. However, as shown in Figure 4 (in yellow), Frame Analyzer showed that the game was rendering it anyway. We added testing whether the skybox was visible in the scene so that it would not be rendered if it was not visible. This enhancement saved 0.88 ms.

![Figure 4](image.png)

Figure 4. The game rendered the skybox (shown in yellow) even when it was not visible.

3. Terrain
The terrain was the most compute-intensive object to render in the scene. This is a common problem in MMORPG games. We found that a significant portion of the terrain was using the same textures and shaders. The time to render this portion of terrain was 5.1 ms, which was 78% of the total time of 6.5 ms for all the terrain. Figure 5 shows the draw calls rendering this portion of the terrain (in yellow). The pixel shader had 89 instructions, 13 of which were texture instructions and 76 were arithmetic instructions.
Figure 5. The portion of the terrain that uses the same textures and shaders took 78% of the total time to render all of the terrain. Its pixel shader had 89 instructions.

Under the Shader tab in Frame Analyzer, we viewed the pixel shader and found that it consumed 90% of the frame time. We simplified the shader for medium and low settings, removing the light map, normal map, AO map, etc. Figure 6 shows the captured frame in Frame Analyzer after changes were made to the pixel shader. The pixel shader now has 31 instructions, 8 of which are texture instructions and 23 are arithmetic instructions. These improvements reduced the time consumed by this portion of terrain from 5.1 ms to 3.4 ms.

Figure 6. The time to render this portion of the terrain was reduced to 3.4 ms after optimizing the pixel shader.

To create a version of the game that was optimized to run on slower-speed processors, the pixel shader was further simplified to only two texture load instructions. The time consumed to render the terrain was reduced to 0.93 ms (Figure 7).
Figure 7. Further simplification of the pixel shader for lower performance hardware.

4. UI
Displaying the UI consumed 7.4% frame time. Figure 8 is a screen shot of the UI from the game.

Figure 8. An example of XuanYuan's UI.
With Frame Analyzer, we found that the character image at the top left corner of the screen was rendered using nine draw calls (Figure 9). The game issued these nine draw calls every time this image was displayed.

**Figure 9.** Nine draw calls are used to render the character image for the UI.

This image was not expected to change from frame to frame. Therefore, we could re-use the image across multiple frames and not render it every time. To do this, we rendered the image to an image buffer and then used the buffered image for multiple frames. Figure 10 shows the frame after the optimization. Re-using the buffered image took only 1 draw call instead of 9.

We extended this optimization to other UI elements in the game. With these optimizations, the time consumed to display the UI was reduced from 1.38 ms to 1.17 ms.

**Figure 10.** An image buffer is used to store the UI image for re-use across multiple frames.
5. State Changes
State changes incur cost in the application, Direct3D runtime, graphics driver, and graphics hardware. Reducing state changes can improve game performance. A detailed analysis of the game in Frame Analyzer showed that there were many draw calls that used very small textures, e.g. 32x32, 64x64, and 128x128. This was inefficient. Figure 11 is a screen shot of two neighboring draw calls using small textures.

![Figure 11. Two neighboring draw calls that use small textures.](image)

These neighboring draw calls used a set of very small textures (Figure 12). Each draw call was preceded by 3 SETTEXTURE API calls. Because the textures set in the second and third SETTEXTURE calls were small, we merged them into one to reduce the number of state changes. Then, we only needed to call SETTEXTURE once to set the texture, and used it across several draw calls. The uv texture coordinates from the smaller textures were remapped to the merged texture.
Figure 12. Small texture sizes.

Figure 13 and Figure 14 are screen shots from the GPA Frame Analyzer after the optimization. We merged the textures used into three draw calls. The number of SETTEXTURE API calls was reduced from 9 to 4, which decreased the number of state changes by more than 500 for every frame. We did not see this optimization translate into a gain in frame rate. It is possible that the optimization reduced CPU time spent in the Direct3D runtime and the graphics driver, but the performance improvement was hidden by bottlenecks on the GPU.

Figure 13. The number of SETTEXTURE API calls was reduced from 9 to 4 after merging the textures.
Figure 14. Several smaller textures were merged.

Optimizations Propagate to Ivy Bridge

The analysis and optimizations described in this paper were done on Sandy Bridge with Intel HD Graphics 3000. Intel’s new 3rd generation Intel Core processors, codenamed Ivy Bridge, have higher performance GPUs running Intel HD Graphics 4000. The optimizations we made to the XuanYuan game and the resulting gains will propagate to the new processors. The gain measured on our pre-release Ivy Bridge systems showed an improvement of 1.30x, from 66 fps before optimization to 86 fps after optimization. In some cases, developers should consider trading some of the performance improvement for power optimization. When running games on battery powered systems, game developers may want to consider placing a cap on the frame rate to optimize power use to extend battery life. In this game, the frame rate was increased sufficiently such that it can be capped to conserve power and give users longer play times between battery recharges. Games designed to be power aware benefit overall user experience on notebooks and Ultrabook™ devices. Ultrabook devices are equipped with Intel HD Graphics and Ivy Bridge processors.

Summary

We completed performance analysis and optimization for the XuanYuan game using Intel GPA. We used Intel GPA to do in-depth frame analysis, run state override experiments, modify shaders, and see the effects of the modifications in real time. We also reduced the excessive number of state changes. Table 1 is a summary of the optimizations implemented and the resulting performance gain. After these optimizations, the frame rate increased by 1.34x on Intel HD Graphics 3000 from 53 fps to 71 fps and by 1.30x on Ivy Bridge from 66 fps to 86 fps.
Table 1 Optimizations conducted on Intel HD Graphics 3000.

<table>
<thead>
<tr>
<th>Optimized item</th>
<th>Optimizing method</th>
<th>Before optimization</th>
<th>After optimization</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow map generation</td>
<td>Use static shadow for some less important objects</td>
<td>2.97 ms</td>
<td>1.03 ms</td>
<td>1.94 ms</td>
</tr>
<tr>
<td>Skybox</td>
<td>If skybox is not visible, don’t render it</td>
<td>0.88 ms</td>
<td>0.0 ms</td>
<td>0.88 ms</td>
</tr>
<tr>
<td>Terrain</td>
<td>Reduce the number of textures from 13 to 8, and simplify pixel shader</td>
<td>5.1 ms</td>
<td>3.4 ms</td>
<td>1.7 ms</td>
</tr>
<tr>
<td>UI</td>
<td>Merge UI elements</td>
<td>1.38 ms</td>
<td>1.175 ms</td>
<td>0.205 ms</td>
</tr>
<tr>
<td>State changes</td>
<td>Merge small textures</td>
<td>~6000 state changes/frame</td>
<td>~5500 state changes/frame</td>
<td>Reduce ~500 state changes/frame</td>
</tr>
</tbody>
</table>

About the Authors

**Cage Lu** is an Application Engineer at Intel. For several years, he has been helping big gaming ISVs in China to optimize game client performance on Intel platforms.

**Tim Cong** has more than 8 years of experience in game development. He participated in the earliest 3D MMORPG development in China. Now Tim is working for Tencent, focusing on game and engine development.

**Kiefer Kuah** is a software engineer at Intel. He optimizes games on Intel platforms.

Appendix A:

---

1 Sandy Bridge Platform Configuration:
CPU: Intel Core i7-2720QM @ 2.20 GHz, with HD 3000 graphics
Memory: 8 GB DDR3 1333Mhz
HD: Intel 160 G SSD
OS: Windows 7 Professional, 64 bit
Graphics Driver: 8.15.10.2509
Ivy Bridge Platform Configuration:
CPU: Intel Ivy Bridge Core i7 @ 2.30 GHz, with HD 4000 graphics
Memory: 8 GB DDR3 1333Mhz
HD: Intel 160 G SSD
OS: Windows 7 Professional, 64 bit
Graphics Driver: 8.15.10.2568