Whitepaper

Optimizing *Marble Blast Ultra*

Eric Preisz
eric@venagen.com

Version 1.0 - February 2008

Optimizing *Marble Blast Ultra*

Abstract

Marble Blast Ultra (MBU) is an *Xbox Live Arcade* and *InstantAction.com* launch title developed by GarageGames.com, Inc. in Eugene, Oregon. Using a process taught at Full Sail Real World Education, located in Orlando, FL, we will examine and improve the performance of this application by 53% - 66% with little decrease in fidelity across different PC configurations.

Introduction

The method used to optimize MBU relies on a detection process that focuses on system, application, and micro optimizations to balance the most efficient optimizations with the opportunities that yield the highest return on our efforts.

The system level of optimization focuses on the utilization of our hardware. We measure utilization as a percentage of time that the hardware is not idle. For example, a common measurement of utilization is 100% CPU utilization and 25% GPU.

System level optimizations include balancing. Balancing is when we move work from an over utilized resource, to an underutilized one. Moving the updates of a particle system from vertex buffer locking and CPU processing to a GPU method would be a useful optimization for the utilization example in the prior paragraph.

Application level optimizations are class or function sized optimizations. These optimizations are typical in every game engine. In fact, one may think of a game engine as a collection of application level optimizations. Quad trees, render state batching, or sorting algorithms are all examples of application level optimizations.

Eric Preisz is a course director for optimization at Full Sail Real World Education in Orlando, Florida and a managing director at Venagen. Venagen is looking for the opportunity to optimize your application. Contact Eric Preisz at eric@venagen.com

Micro optimizations are the fodder for mailing lists and arguments; these line by line optimizations are easy to discuss in terms of latency and throughput, yet their use to increase performance tend to vary from processor-to-processer and configuration-to-configuration. Unless your detection process points you at a specific area ripe for micro-optimizations, yield to the suggestions of an optimizing compiler. Shaders, however, are an exceptional opportunity for micro optimizations, especially if optimizing your shaders means fewer shader instructions.

Understanding these levels of optimization is important to our task of optimizing a game. Using the process of detection utilized at Full Sail Real World Education, we attempt optimizations in the following order: system, application, micro.

When we optimize a game and the frame rate increases, we must repeat the process from the beginning. A game flows through our instructions like water through a stream – remove a rock (bottleneck/hotspot) and the water flows using new paths. Even the smallest optimization can change what resource limits our applications performance.

In a perfect world, we would solve every system level optimization before moving to the application level. Likewise, we would optimize the application level before optimizing micro. Doing this helps us to find the optimizations with the biggest return on investment. In the real world, milestones and deadlines may prohibit the largest opportunity for frame rate increase if the implementation of the optimization is complex. In these cases, programmers are likely to choose a less time consuming optimization.

Even more important than finding the optimization that yields the biggest gain is ensuring that we don't perform an optimization that yields no gain. When we incorrectly optimize a fast parallel stage, the stage that limits our frame rate may still limit our performance as it did before the unprofitable optimization.

Understanding the three levels of optimization helps us in our task. As developers, we need not just optimize our games, we must optimize them efficiently. Large game engines can contain 500,000 to 1.5 million lines of code; understanding what code to optimize is as important as the solution to what you discover.

This paper will explore system, application, and micro optimizations, used to optimize a title developed by Garage Games. The tool we used for GPU optimizations NVIDIA's PerfHUD 6.0.

Benchmark

For our benchmark, we chose an average scene that contained many of the effects used throughout the game. To make our benchmark representative of

the final deliverable required for in-browser rendering, we must optimize the application in windowed mode.

Figure 1. Pre-optimization benchmark running at 78Hz.

Figure 2. Post-optimized benchmark running at 130Hz.

By choosing an “average frame” we are hoping to achieve uniform optimizations across the entire application. This paper will only address several iterations of optimization using one benchmark. This process, to ensure a performance increase across many configurations of PC, should be run on many

different GPUs and CPUs. In this particular case study, we executed the majority of our application using a 1.18 GHz Intel Core 2 Duo and a NVIDIA 8300 GS.

Detecting Errors

Before diving into the optimizations, it is advisable to search for common API errors or bad practices. Using the Debug Console, PerfHUD indicates that there is a vertex buffer being locked every frame.

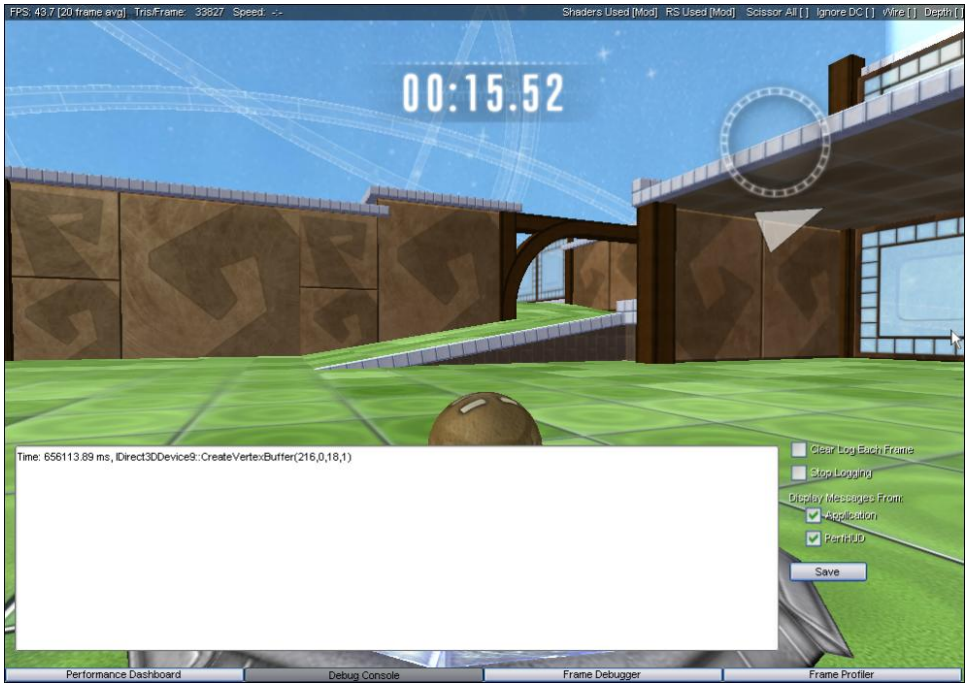

Figure 3. The Frame Debugger displays a vertex buffer creation call that occurs during game runtime.

Placing a breakpoint inside the engine reveals the misuse of an enumeration of the engine's graphics layer that is causing the creation and deletion of a vertex buffer. A simple switch of the enumeration, converts the vertex buffer to dynamic, which also removes the call from the Debug Console.

The frame rate increase is minimal at best, but now that we are free of the API "abuse", we are able to continue the process of optimizing the application.

First Optimization

First Optimization: System Level View

Our first goal is to determine our system utilization. This is easily done in the Performance Dashboard. PerfHUD collects data directly from hardware counters. One of them, GPU Idle, will help us determine if we should be optimizing the CPU or the GPU.

GPU Idle measures the idle state of the graphics card across each frame. Through the customization of the interface available, we will select GPU Busy, which is simply 100% - GPU Idle %.

Using this number will provide us insight on whether we are more CPU or more GPU bound. By optimizing the resource that is bounding our application's performance, we are likely to find the biggest bottleneck or hotspot.

Figure 4. GPU Utilization is near 100%. This data is represented by the green line that is near the top of the chart.

At this stage, our application is GPU bound, which we are able to determine by PerfHUD's reading of GPU Busy. Figure 4 shows a GPU Busy number that is near 100%.

Empirical testing, performed by throttling CPU instructions and plotting frame rates performance against GPU Busy output, suggests that an application is equally CPU and GPU bounded near 85% GPU utilization. Therefore, when reading GPU Busy numbers, we suggest that a GPU busy that is less than 80% is likely CPU bound. An application with a GPU Busy number that is greater than 90% is mostly GPU bound.

Knowing that we are GPU bound doesn't tell us enough to apply a solution. The GPU contains many parallel units performing different operations on the data. These are arraigned in a deep pipeline on the GPU, similar to multiple vehicle assembly lines. These assembly lines perform parallel operations on batches of triangles and pixels that share similar render state. When optimizing the GPU, we must focus on which unit is the bottleneck, slowing the entire pipeline. We should focus our optimizations there.

By continuing in the Performance Dashboard view, we are able to gather more information about our benchmark. Figure 5 shows that shader (~60%) and texture (~45%) utilization is the highest, when compared to the utilization of vertex assembly (<~5%) and raster operations (~17%). Using this information, we build an educated guess that we should focus on shader and texture processing.

With shader model 4.0, shaders are unified, meaning that vertex, geometry, and pixel shaders all share the same shader pool. Figure 5 also shows us that the majority of shader utilization lies in the pixel shader.

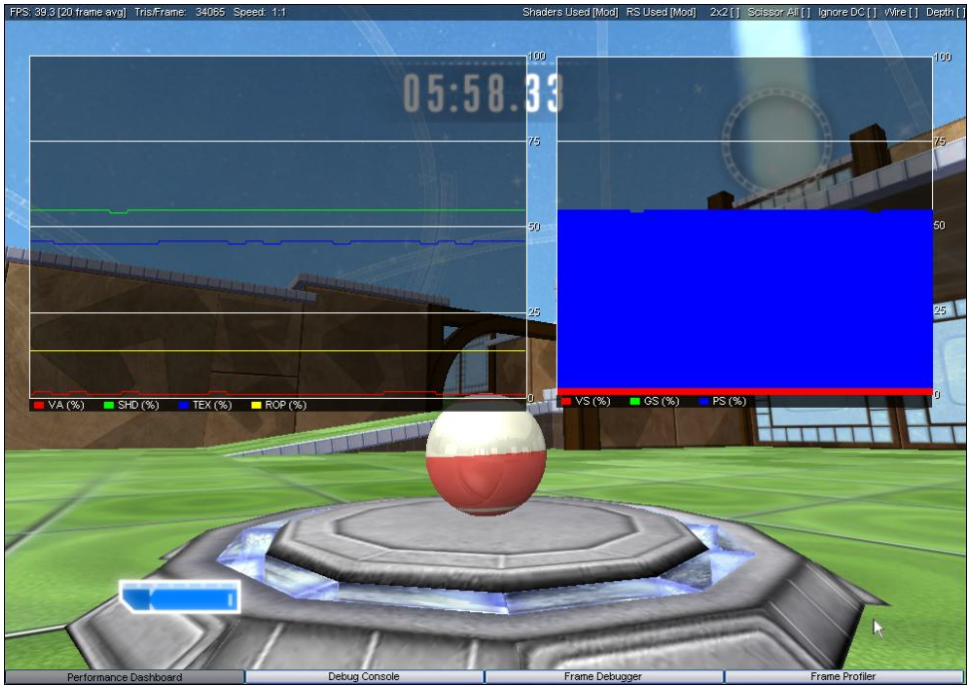

Figure 5. The Performance Dashboard indicates that the shader unit is operating near 60% (green line) and the texture unit is operating near 45%. On the right, we see that the 60% of the shader is dominated by work in the pixel shader.

Understanding this data is useful as we continue. Pressing F8 activates one of the most useful features of PerfHUD, the Frame Profiler. When entering the Frame Profiler, PerfHUD runs a series of tests on a single frame, measuring the performance at different units along the GPU pipeline. When the tests are complete, PerfHUD presents the data by sorting the draw calls into groups sharing common state, calling these State Buckets. At first, the largest State Bucket (and the slowest draw call in that State Bucket) is selected. For more information on state buckets, see the PerfHUD user guide. This draw call represents a significant optimization opportunity.

The current State Bucket contains many draw calls, but one draw call seems particularly costly. Figure 6 displays the slow draw call, highlighted in the scene by PerfHUD in wireframe. Notice that the draw call generates 257,662 pixels.

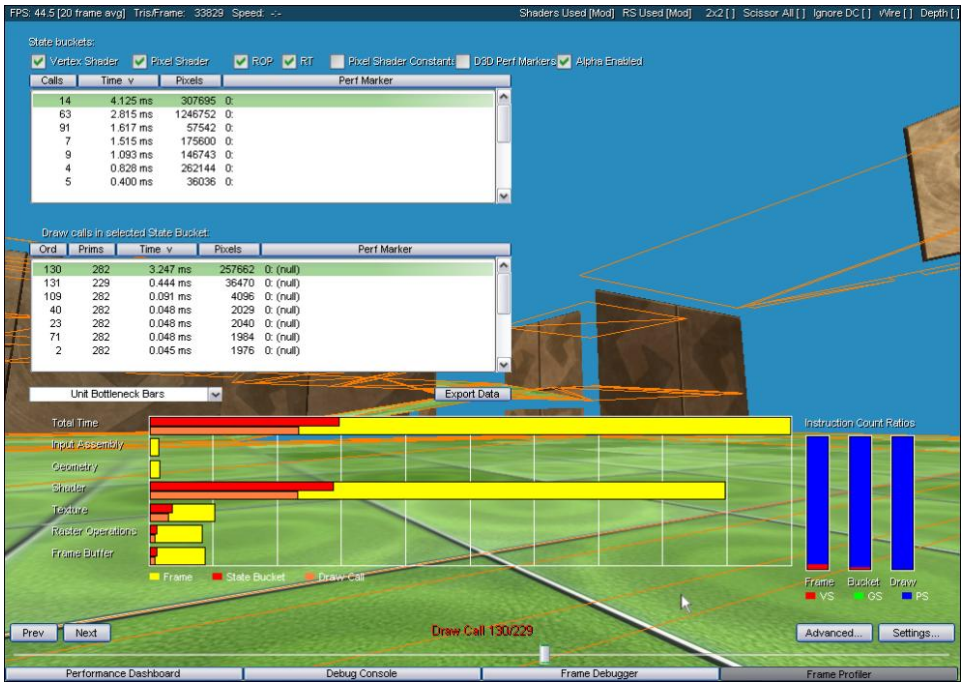

Figure 6. Draw call 130 takes 3.247 milliseconds and draws 257,662 pixels.

There are many optimizations we can apply to this draw call. Using the knowledge gained from our system level, an appropriate application optimization is available.

First Optimization: Application Level

Using the knowledge from our system level analysis, we choose to perform a z-pass to help reduce the amount of pixels generated by the draw call. We implement this by drawing parts of the scene twice, disabling color-writes in the first pass. This optimization works under the following conditions:

- Low number of draw calls.
- Under utilization of vertex assembly, vertex shader, and geometry shader.
- High utilization of pixel shader resource due to processed pixels that are occluded.

This profile fits the description of our current application; when this optimization is applied, we see a frame rate increase. The z-culling of those pixels has reduced the number of pixels from 257,662 to 115,868. This optimization is particularly interesting since doubling the number of draw calls actually increased overall frame rate.

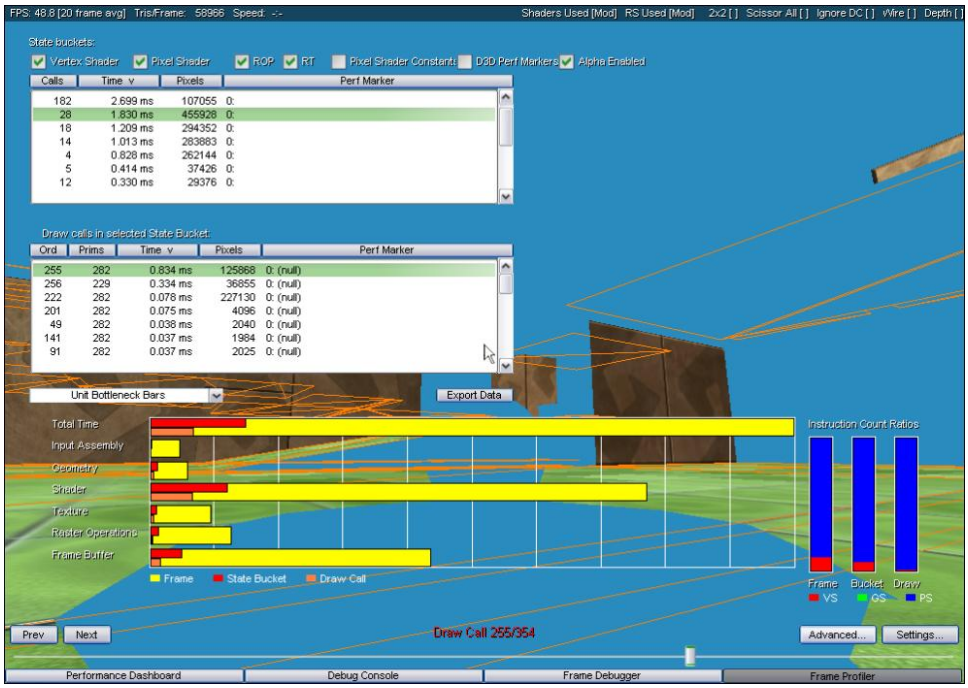

Figure 7. Using a z-pass, the application culls pixels more efficiently.

Now that we have increased the performance, we must move back to a system level view and determine if we have drastically changed the performance balance between CPU and GPU.

Second Optimization

Second Optimization: System Level

Moving back to the Performance Dashboard, PerfHUD shows that we are still GPU bound and our shader utilization is a few percent less than in the past view. Since we are still mostly limited by shaders, and more specifically pixel shaders, we will again run the Frame Profiler.

Second Optimization: Micro Level

After running the Frame Profiler, PerfHUD points to the same draw call. There is no easily implemented application optimization, except for possibly better culling; but, that wouldn't change our pixel performance since pixels outside the viewport are already culled during rasterization.

The remaining optimization on this particularly costly draw call is to micro optimize the pixel shader.

The developer, GarageGames, was already aware that this pixel shader was costly and had already revised the shader to perform using less texture samples.

Using the Pixel Shader Advanced State Inspector feature of PerfHUD, we are able to replace this shader and compile it to test if our new shader is more optimal. We can also toggle between the original and modified shader to confirm the performance change in the Performance Dashboard.

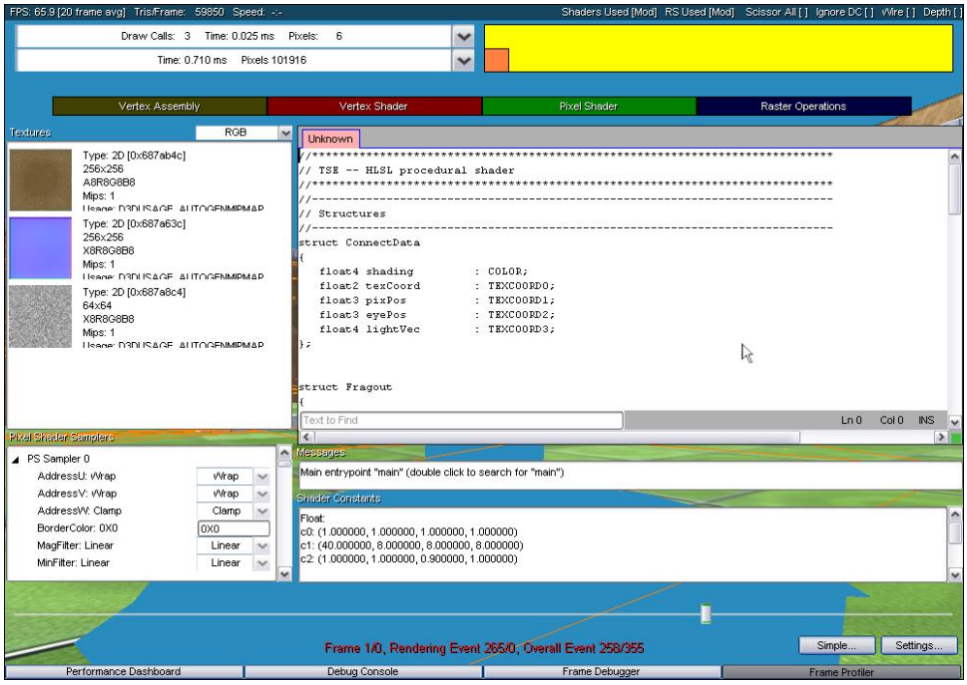

Figure 8. Using PerfHUD, we are able to replace the current shader in the Advanced State Inspector.

Third Optimization

Third Optimization: System Level

Having performed the optimization, we review the Performance Dashboard and find only a slight change in shader performance and GPU utilization. The Frame Profiler is showing a new, slowest state bucket and is pointing at a fixed function draw call used to render the glow buffer into the framebuffer.

Instead of continuing to look for an increase in shader performance, we will look at increasing the performance of the next highest utilized GPU sub-unit, which, according to PerfHUD, is the texture unit.

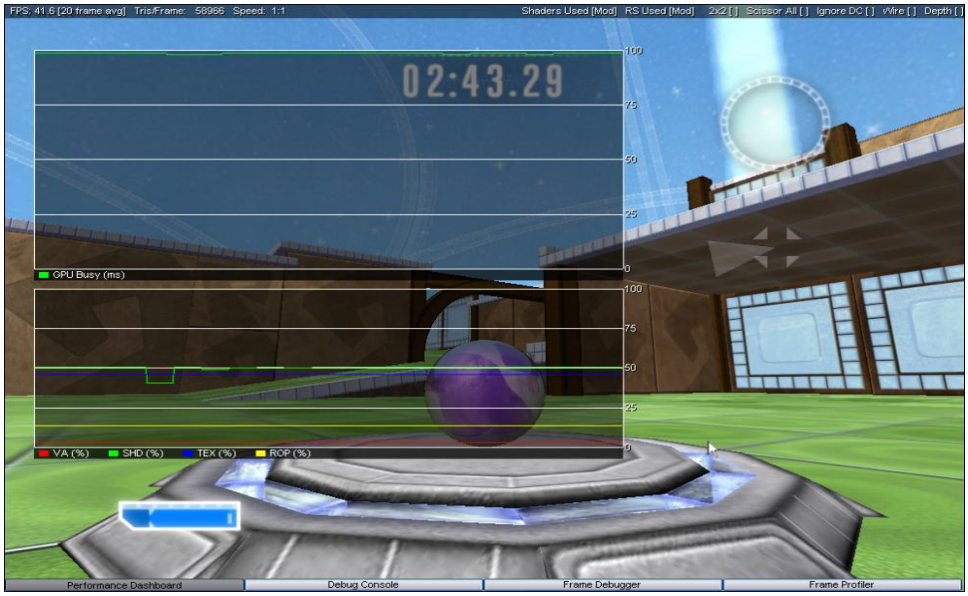

Figure 9. Performance Dashboard shows that we have reduced the utilization of the shader unit.

Third Optimization: Micro level

The easiest way to test texture performance is to use the "ctrl+t" key combination while in the Performance Dashboard. This causes PerfHUD to substitute a 2x2 texture for every texture in the scene. When we perform this operation, the frame rate should increase, but does not. The texture unit utilization stays the same.

Detecting the next optimization requires an obscure observation. Since the ctrl+t key only affects the draw calls, then a potential optimization lies in another call. In this case, the trouble call is a stretchrect with a *src* that contains an effect buffer and a *dest* that contains a pointer to the backbuffer.

Stretchrect is supported in hardware, so the use of the call is acceptable; however, the resolution src texture is quite large at 512x512. That means for that stretchrect, you will be reading and filtering 262,144 texels. Reducing the resolution to 256x256 reduces resolution, but increases performance significantly. The pixel utilization increases significantly and texture unit utilization drops.

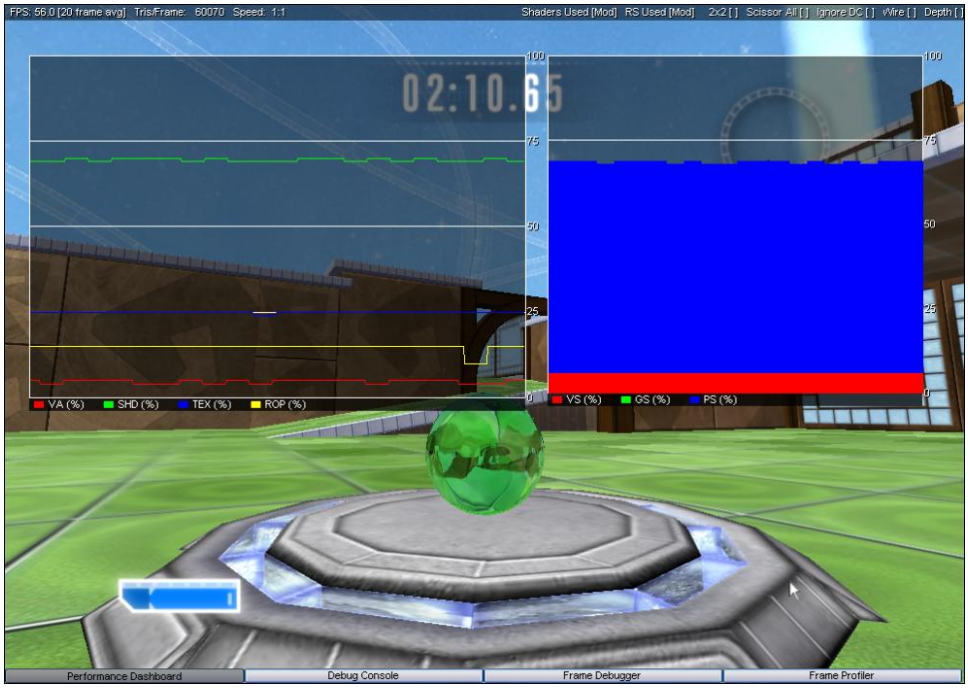

Figure 10. After reducing the resolution of the strechrect, pixel utilization increases and texture utilization decreases.

Final Optimization

Final Optimization: System Level

There are no more obvious texture or pixel shader optimizations and we are still GPU-bound. This final optimization is difficult to detect, yet yields a significant performance gain.

The GPU is a deep pipeline with a large latency and incredible throughput. This deep pipeline allows the CPU to be three frames ahead of the GPU. This creates noticeable latency for user input when frame rates are low.

Due to a bug, the application reduces this latency in a costly way. The application mistakenly falls back to a method that locks a dynamic texture with a NULL flag forcing a CPU/GPU sync and download from video memory.

You can visualize the overhead in the Performance Dashboard by viewing the driver time. The wasteful driver time occurs during the download and sync.

Figure 11. Driver time is reduced by removing the rendering "fence".

Conclusion

Our goal was to increase performance without sacrificing appearance. We were able to meet this goal with almost all of our optimizations. The stretchrect optimization does reduce fidelity in areas behind refracted glass, but the impact on the scene is minimal and acceptable from an artistic standpoint.

With help from PerfHUD, our optimizations yielded a 53% frame rate increase from the starting 40.1 Hz to 61.4 Hz on the 8300GS.

By using the ctrl+s key in the Performance Dashboard we are forcing all pixels to be rejected in rasterization. Since the frame rate is still increasing, we can conclude that there are still more opportunities for performance increase by optimizing the pixel shader, texture, or ROP units.