Building Performant VR Applications for Multi-Domain Modeling and Simulation

Josh Klint
Leadwerks Software
ATA Aerospace
Los Angeles, CA
josh@ultraengine.com

Doyle Towles
Peraton
Pasadena, CA
doyle.towles@peraton.com

Ben Gavares
Northrop Grumman
Greenbelt, MD
benjamin.j.gavares@ngc.com

Michael Juliano
Northrop Grumman
Pasadena, CA
michael.juliano@ngc.com

ABSTRACT

Recent advances in virtual reality hardware have lowered costs and improved the portability and ergonomics of virtual reality training systems. Advances in hardware and software have been driven largely by the entertainment sector and adopted for modeling and simulation. However, while working with a team at NASA's Goddard Space Flight Center we discovered problems of scale and performance that game-centric technologies fail to address.

Maintaining high framerates in VR is important in order to prevent the user from experiencing motion sickness. Some of our projects involved the display of highly detailed CAD models in VR. This presented severe performance challenges to our existing software and required a new renderer design that more closely conforms to the graphics hardware. An orbital mechanics simulator we developed involves large scales beyond the precision of 32-bit floating point math. The use of 64-bit double floats provides improved precision, but the actual implementation proved to be more nuanced and required balance with our goals of improved performance.

To solve these problems, we first performed an analysis of graphics hardware. Notable hardware trends were used to create a design framework that provided guidance during implementation of a new Vulkan-based renderer. A series of benchmarks was devised to test our new design against our original OpenGL renderer. Our results demonstrate that the techniques described here yielded more efficient GPU utilization and order-of-magnitude performance gains in a variety of scenarios.

ABOUT THE AUTHORS

Josh Klint has more than ten years entrepreneurial experience as the founder of Leadwerks Software, was involved with Valve during the early development of their VR technology, and assists with projects at Goddard Space Flight Center as a software engineer for ATA Aerospace. He is a contributing author to *Game Engine Gems 3* (2016) and speaker at the Game Developers Conference (2013). Josh has a bachelor's degree in biology from UC Davis and an MBA from Sacramento State University.

Doyle Towles has 35 years professional experience in mechanical design, analysis, systems engineering and project management for manned and robotic aerospace missions. He led the Systems Engineering and Advanced Technologies team, supporting JPL on the Mars Science Laboratory and Mars 2020, as well as satellite servicing on Hubble Space Telescope, Robotic Refueling Mission, and Restore-L. Doyle also led a visualization team which generates virtual reality, animation, graphics, and video production for training and simulation.

Ben Gavares is a senior mechanical designer and a 10-year veteran of the Goddard Space Flight Center's engineering division. His focus for the past decade has been taking a mission from concept to reality as effortlessly as possible. Ben began his spaceflight career with the Hubble Space Telescope servicing mission 4, where he acted as a mechanical designer for three years. He is currently working on the development of VR applications for NASA's Code 540 Division, while continuing his role as Senior Designer.

Michael Juliano has over twenty years' experience in the video games industry—thirteen as a 3D artist focusing on environments and level design; and the remainder as a software engineer. He served as Lead Environment Artist with ImageSpace Incorporated; and, as a solo effort, developed a spaceflight simulator with a focus on proper orbital physics and spacecraft systems simulation. He is currently a designer with Northrop Grumman, working on the development and integration of VR software applications.

Building Performant VR Applications for Multi-Domain Modeling and Simulation

Josh Klint
Leadwerks Software
ATA Aerospace
Los Angeles, CA
josh@ultraengine.com

Doyle Towles
Peraton
Pasadena, CA
doyle.towles@peraton.com

Ben Gavares
Northrop Grumman
Greenbelt, MD
benjamin.j.gavares@ngc.com

Michael Juliano
Northrop Grumman
Pasadena, CA
michael.juliano@ngc.com

INTRODUCTION

Recent advances in virtual reality hardware have lowered costs and improved the portability and ergonomics of virtual reality training systems. Advances in hardware have been driven largely by the entertainment sector (Smith, Desnoyers-Stewart, and Kratzig, 2019). Several popular development platforms and middleware packages for creating VR applications have likewise been developed first for games and then adopted for simulation purposes. While working with a team at NASA's Goddard Space Flight Center we discovered two problems that game-centric technologies fail to address.

An orbital mechanics simulator we developed involves large scales beyond the precision limits most games use. Computer games typically use 32-bit floating point values for object orientations (translation, rotation, and scale). Graphics hardware support for 64-bit floating point math was first introduced in Nvidia's GPUs in 2010 (Whitehead and Fit-Florea, 2011), but is rarely used. Exceptions do exist, but these are typically for a specific game in a genre that demands it, such as space and flight simulations. Projects under consideration required a more robust integration with 64-bit float support for graphics, rigid body dynamics, line-of-sight testing, and pathfinding.

We also experienced significant challenges with rendering performance. Performance in VR has always been a major consideration. VR involves the rendering of two images, and head-mounted displays for VR usually render at higher framerates (90-144 Hz) than a typical computer monitor does. A rendering process designed for a 60 Hz computer monitor can safely take up to 16.7 milliseconds to complete, but the same process must be performed twice in 6.9 milliseconds for VR rendering, once for each eye. The constraints become even tighter when detailed engineering models are used. Games are typically designed with a budget for polygons, texture data, and other parameters. The art team works within that budget to develop content, with the expectation that the final product will run at the desired framerate on the target hardware. In other words, the target hardware determines the budget for polygon count, texture resolution, and other parameters. When starting simulation projects, we quickly discovered that polygon counts were predominantly determined by the design of the real object. Small details could not be optimized away, because those were often the details the user was interested in viewing in VR.

The demands of our NASA VR projects forced us to rethink renderer design. In this paper we will describe our research into graphics hardware trends, describe our formulation of a framework for renderer design, describe how design decisions were weighted against our framework, and review the results of our resulting Vulkan-based renderer compared to the older OpenGL renderer we started with, as well as the Unity 3D engine.

HARDWARE ANALYSIS

Our first step was to assess modern graphics hardware trends to see how we could make our renderer conform more closely to modern GPUs and continue to scale with future hardware. Our previous rendering architecture was developed around 2010, so it was worth re-examining to see if our assumptions about optimum renderer design were still valid. In our examination, we found three notable hardware trends.

First, we found that processing power on both the GPU and CPU was increasing at a faster rate than memory speed (Knarr, 2019). The GPU is a massive parallel processor, and the number of cores is a good indicator of overall rendering speed. Between the release of the Nvidia GeForce 280 in 2008 and the GeForce 2080 in 2018, the number

of GPU cores increased by more than ten times (see Figure 1). During the same period memory speed experienced a more moderate 300% increase (see Figure 2).

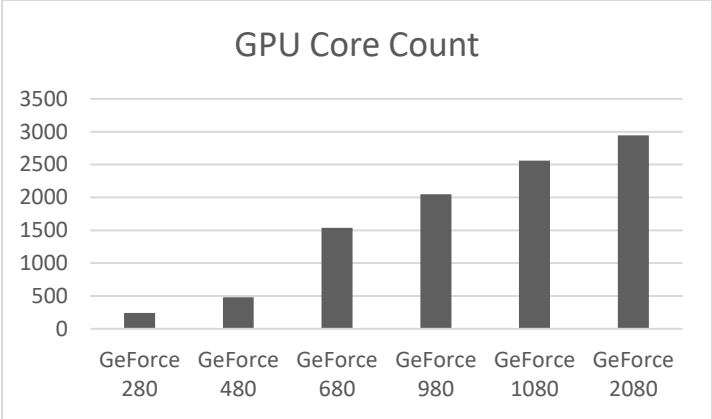

Figure 1. Core counts in select Nvidia GPUs from 2008-2018

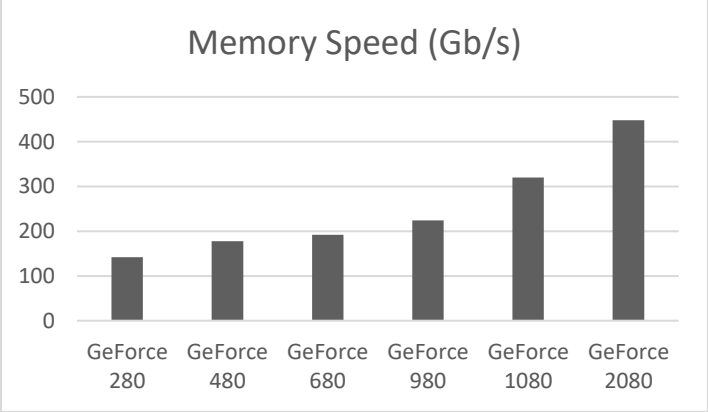

Figure 2. Memory speed in select Nvidia GPUs from 2008-2018

Second, we observed that popular real-time renderers typically demonstrate decreased GPU utilization as scene complexity increased. This is caused when the CPU rendering code creates a bottleneck for instructions sent to the GPU, and is sometimes referred to as "GPU starvation", as the GPU is sitting idly waiting for commands. We hypothesized that eliminating this bottleneck would result in higher framerates and performance limited by the GPU alone.

Finally, we noted that graphics card shader logic support has improved significantly since the first implementations of programmable shaders. Features like variable-length *for* loops and branching *if* statements that would have been unsupported or nonoptimal in the past were now perfectly safe to use (Akenine-Möller, Haines, and Hoffman, 2018).

Based on these observations we developed three directives to guide our new renderer design:

- Reduction of data passed over the PCI bridge and increased reliance on the GPU for processing and decompression of data.
- Better utilization of multicore processors to prevent CPU bottlenecks from slowing down the GPU.
- Increased reliance on complex shader logic.

Renderer Design Framework

We next developed a design framework based on our directives. In order to reduce the amount of data passed over the PCI bridge to the GPU we identified three methods:

- Store more data persistently in video memory instead of resending it each frame.
- Update data in video memory less frequently, when possible.
- Whenever possible, use some form of simple compression for data that is sent to video memory. This necessarily requires more processing time on the CPU for compression and on the GPU for decompression, and that is a tradeoff we are intentionally making.
- Dedicate more CPU threads to optimal preparation of data for submission to the GPU.

In order to maximize GPU utilization and prevent the phenomenon of GPU starvation, we decided to isolate all graphics instructions onto one dedicated rendering thread. The idea was that the rendering thread would always cycle at the maximum frequency, and the rest of the 3D engine architecture must conform to support that.

Because of our decision to embrace complex shader logic, we decided to make liberal use of conditional statements, loops, and buffer lookups, with the expectation that this would result in overall faster framerates, despite the increased computational complexity in the pixel pipeline. This causes our pixel shaders to be relatively more expensive. To mitigate this, we implemented an early Z-pass, which ensures that we are not wasting time drawing occluded pixels.

IMPLEMENTATION

To achieve usable framerates in VR we focused on maximizing GPU utilization. Our thinking was that as long as the GPU utilization was under 100%, some unutilized processing capacity existed, and our framerate was therefore slower than it needed to be. To this end we sought to minimize processing time in the rendering thread, and to minimize GPU memory bandwidth usage.

The first step was to reorganize our single-threaded architecture into multiple threads. Our final design uses one thread for the main logical loop running at a constant 60 Hz (see Figure 3). The logic thread executes all user-defined code for the specific application that is being developed, and it is given a full 16.667 milliseconds to execute without affecting the framerate. The logic thread interfaces with a separate rendering thread. When an object state in the logic thread changes, for example the position of a vehicle is updated, that information is sent to the rendering thread. Animation also interfaces with the logic thread in a thread pool of variable size. This allows the logic thread to access the animated orientations of limbs before passing that information to the rendering thread.

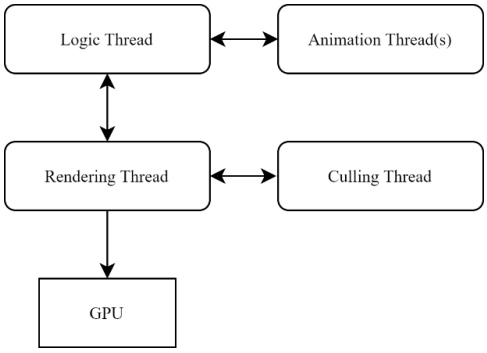

Figure 3. Diagram of partial multi-threading structure.

We initially thought this approach might be unworkable due to the latency this introduces to the system. Latency is a major concern for VR, and this design produces a significant delay in the time between when an object moves and when the displayed position is fully updated. In practice, we found that not all latency matters. Latency in the orientation of the user's headset and hand controllers should be minimized, but latency related to objects in the rest of the world does not produce any negative effects. Since there is no connection to the user's body, there is no perceived delay. We update the orientation of the camera and hand controllers in the rendering thread, just before drawing each frame, and this is sufficient to give the entire simulation a fluid feeling, even if the perceived orientation of other objects is one frame behind their true position in the simulation.

Culling

To maximize GPU utilization, we isolated our culling routine on a separate thread. Our initial design placed a culling thread in between the logic and rendering thread. Data was passed from the logic thread to the culling thread, and then forwarded to the rendering thread. However, we found that this design produced excessive latency. Our final design made the culling thread branch off from the rendering thread. Object state changes flow directly from the logic thread to the rendering thread and are then forwarded to the GPU. Camera visibility sets are calculated asynchronously. We found that a much lower culling frequency of 45 Hz was acceptable in most usage scenarios. The culling thread intermittently returns visibility sets to the rendering thread. Several frames are typically drawn using the same visibility set before a new one is received.

Swept Frustums

The only drawback of reusing visibility sets to draw multiple rendering frames was that sometimes an object could suddenly "pop" into view. This was most apparent when the camera was turned rapidly or when a camera was moving backwards at a high velocity, low to the ground. To solve this problem, we experimented with warping the culling frustum to account for the camera inertia. In the culling thread, the difference between the camera rotation and position during the last two frames received is used to calculate the camera's translational and rotational velocity. Then a third orientation is extrapolated to estimate where we expect the camera to be in the next keyframe. A convex hull is calculated that encapsulates all three frustums (see Figure 4). This convex hull is then used to test whether objects intersects the camera frustum volume. This technique significantly reduces the artifacts we observe when turning or moving the camera at a fast rate. When enabled, this feature can also cause a decreased framerate when the camera is rapidly turned, because the resulting visibility set includes more objects due to the larger frustum volume.

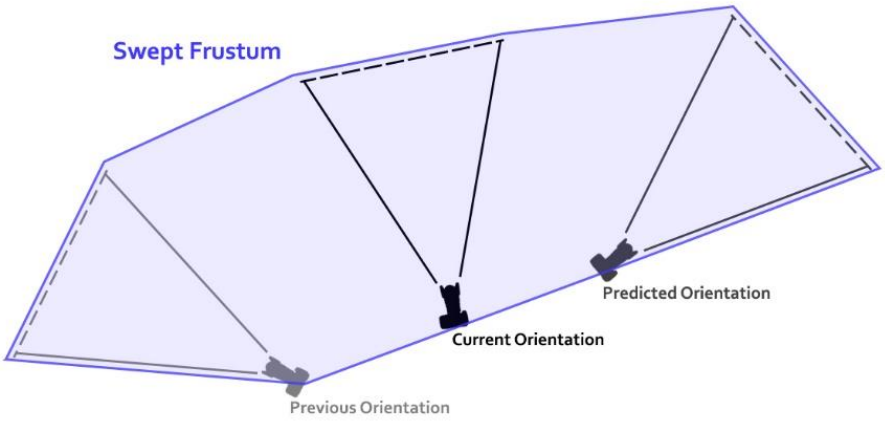

Figure 4. Convex hull encapsulates the previous, current, and extrapolated next camera frustums.

Float Precision

We had very specific and robust goals for our new renderer implementation. We wanted to extend the visual capabilities of our current orbital simulation by allowing more and higher-detailed objects, represented with millimeter precision at the outer reaches of the solar system. We also wanted to use that implementation to initiate the development of multi-domain simulation, which would allow for real-time tracking and visualization of multiple assets from land, sea, air and space. This required much bigger scales than what we were accustomed to working with in games. The 32-bit floating point math most games use produces a vertex distortion effect when the camera moves more than a few kilometers away from the origin. The use of 64-bit floats can prevent this effect by increasing the precision of calculations (see Figure 5).

Figure 5. Vertex distortion at large distances with 32-bit floats (left) is resolved by using 64-bit floats (right).

A naive approach to increasing precision would be to simply replace all 32-bit float variables with 64-bit double values in our C++ source code. However, there are two problems with this. First, 64-bit floating point math is slower than 32-bit floating point math (DeLoura, 2001) and many of our routines do not benefit from increased precision. For example, material colors are stored as 32-bit Vec4 values and do not suffer from any precision issues. The second problem is that graphics hardware does not actually fully support 64-bit floating point values. The maximum depth buffer resolution supported by Vulkan at the time of this writing is a 32-bit floating point value, and the vertex shader output value *gl_VertexPosition* in GLSL uses 32-bit precision. Consequently, we will always be operating in a mixed-precision environment. The only question is what precision we will use for each data type, and at each stage in the rendering pipeline.

We started by implementing double-float versions of our 3D math structures. It was determined during development that double-float structures would include a constructor that would automatically convert from their 32-bit analog, but that the conversion should not work automatically in reverse

For object 4x4 matrices we use a *dMat4* structure in the double float build, which is a 4x4 matrix made up of four *dVec4* structures. To reduce the amount of data sent to the GPU each frame we stored these values persistently in a GPU memory buffer, and only send a single 32-bit unsigned integer ID indicating which instance of an object to draw. This integer value is an offset in the buffer to the position where the object's matrix is stored. A uvec4 array is used in the vertex shader for more efficient packing.

Vertices

Mesh vertex positions still use 32-bit floating points for each axis, for three reasons. First, a single discrete model of a spacecraft or building is likely to have vertex positions relatively close to the object's local origin. Second, our primary model format glTF does not support double float vertex positions, so increasing the precision here would not preserve any data from any of our existing 3D models. Finally, a large model of a planet or other heavenly body is more likely to be displayed using a geometry streaming system rather than a single model loaded into memory at once.

Animation

For animated skeleton data, we discovered we could get away with using 16-bit half-precision floats, because a realistic animated skinned character will not be not much bigger than a human being. Half-floats are not a number type the CPU recognizes, but it does provide a way to compress these values before sending them over the PCI bridge to the GPU. Our implementation stores this data in a texture using the *VK_FORMAT_R16G16B16A16_SFLOAT* format. (Vulkan 1.1 has since added support for half-float data stored in GPU memory buffers.) Decompression to a 32-bit vec4 values occurs when the texture lookup is performed in the vertex shader. We also changed our bone data format structure to make it smaller. Our previous renderer used a 16-element 4x4 matrix, but our new structure packs the same data into just eight elements made up of a position, quaternion, and uniform scale value:

```
struct BoneOrientation {
       short pos_x, pos_y, pos_z, scale;
       short rot_x, rot_y, rot_z, rot_w;
};
```

We did not create a *half* data type or corresponding *hVec3* or *hQuat* classes because conversion from 32-bit to 16-bit floats in our code is one-way only, no math is performed at 16-bit resolution, and half-floats are rarely used.

Although the quaternion to 4x4 matrix conversion performed in the vertex shader is relatively expensive, this is ideal because animated skeleton data is updated frequently, making it a prime candidate for optimization. Two textures are swapped each frame. The current animation data is sent to one, and the other holds the orientation from the previous frame. The vertex shader interpolates between the two orientations, for each bone. This makes the vertex shader more expensive, but reduces the amount of data sent over the PCI bridge. Minimizing our data update frequency and the use of half-floats reduced our memory bandwidth usage by nearly ten times when rendering to high-frequency displays (see Table 1). This design takes advantage of our observation that we could afford more computation on both the CPU and GPU side, if it reduced the amount of memory bandwidth usage.

Table 1. Animation Memory Bandwidth Usage

	Skeletons	Bones	Bytes per bone	Frequency	Data (Mb/s)
Original	1000	256	64	144	2250
Compressed	1000	256	16	60	234

Our mixed-precision approach allows us to simulate the entire Earth and moon to-scale, with sub-millimeter precision, while compressing data that does not require high precision (see Table 2). We opted to create two builds of our 3D engine. The single-precision build provides optimum performance for games and small simulations. The double-precision build provides support for very large areas but has slower performance in vertex-limited scenes.

Table 2. Data Types

Data	Type	Bytes per element	Update Frequency
4x4 Matrices	double	8	occasional
Vertex positions	float	4	rare
Bone orientations	half	2	frequent

Lighting

Deferred lighting is a popular method of rendering lighting in video games (Pharr, 2005). The technique works by drawing the scene diffuse, normal, depth, specular, and other attributes into a set of textures sometimes referred to as a *gbuffer*. Lights are then rendered in a post-processing step using the contents of the gbuffer for input values (see Figure 6). This allows deferred rendering to display a large number of lights with enough flexibility to show different light types, something that early forward renderers struggled with.

Figure 6. Deferred rendering involves drawing data to several images including albedo (top-left), world-space normals (top-right), and depth (bottom-left), which are combined into the final lit image (bottom-right).

However, the technique comes with two limitations worth considering. Transparent surfaces can be a challenge with deferred rendering, since only one value is stored in the depth buffer. One solution is to discard half the pixels on transparent surfaces in a checkerboard pattern. When combined with multisample antialiasing (MSAA) the resulting image approximates a surface that is 50% transparent, with correct lighting.

The other problem with deferred rendering is that rendering to large textures introduces a large initial hit in performance, although deferred rendering scales well relative to a naive forward renderer as more lights are added to the scene. Although initial implementations of deferred rendering did not support hardware MSAA, the advent of multisample textures in OpenGL with the *GL_ARB_texture_multisample* extension solved this issue. However, multisampled framebuffers increase memory bandwidth usage even more.

We saw good initial results with a technique called *clustered forward rendering* (Olsson, Billeter and Assarsson, 2012). This is a forward renderer that divides the view frustum up into a 3D grid cells (see Figure 7). Each cell stores information to access all the relevant lights that intersect that volume. The lighting information is only sent to the GPU with each new visibility set, and may be reused to render several frames.

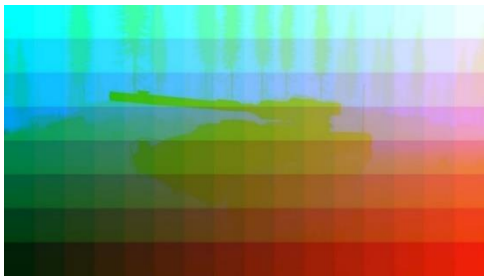

Figure 7. Clustered forward lighting divides the frustum into a 3D grid. Each cell references a buffer offset where intersecting light indexes are stored.

This technique promised to relieve the renderer from the high bandwidth usage of deferred rendering. Multiple layers of shaded transparent surfaces can be easily displayed with no special code. Because the lighting calculations are relatively expensive, we perform an early Z-pass before the final render, drawing only to the depth buffer. This ensures that each pixel only has lighting calculated once, except for transparent surfaces which are skipped in the early Z pass and then rendered back-to-front in the final pass. In effect we are trading a high memory bandwidth usage technique with relatively simple shader logic for a low memory bandwidth usage technique that uses more complex shaders.

Migration from OpenGL to Vulkan

Our rendering architecture was first prototyped using the OpenGL graphics API. OpenGL was more familiar to us, and we expected the verbosity of the Vulkan API would make it difficult to quickly experiment with different ideas. Once our core design was solidified, we migrated our renderer over to the newer Vulkan graphics API, as the promise of lower driver overhead seemed to align with our goals. Vulkan also offered reliable support for multi-view rendering with the *VK_KHR_multiview* device extension, with guaranteed minimum support for rendering six views at once. This can also be used for single-pass stereo rendering for VR. It is also useful for drawing to six faces of a cube map in a single pass, for cube shadow maps.

RESULTS

We devised four benchmarks to test the performance of our new Vulkan renderer (referred to below as “Ultra Engine”) against the older OpenGL renderer we started with (Leadwerks 4.6). We also recreated the benchmarks in Unity (2020.3.15f2). Each test is designed to evaluate scalability under different scenarios that may occur during normal usage. All measurements were recorded using a PC with a quad-core Intel Core i7-7700K CPU @4.20 GHz and an Nvidia GeForce 1080 GTX GPU with driver 471.41 installed, on Windows 10.

Instanced Geometry Test

In this test, a 3D grid of 32,768 instanced cubes is created (see Figure 8). The purpose of this test is to see how the renderer handles a large number of objects with CPU frustum culling enabled. Frustum culling on the CPU is important because it allows us to discard large numbers of objects from the camera visibility set, but if all objects are visible it actually becomes a liability, since the test has a computational cost but does not actually prevent any objects from being drawn. We expected our new renderer to perform faster due to the more advanced threading architecture and reduced data transfer.

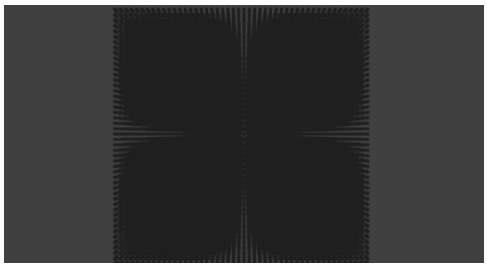

Figure 8. Instanced geometry test to evaluate object management and frustum culling efficiency.

This test produced a similar framerate in both Leadwerks and Unity, with both engines demonstrating a rendering bottleneck on the CPU. Our new Vulkan renderer ran with 95% GPU utilization and a framerate more than 20 times higher (see Table 3).

Table 3. Instanced Geometry Test Results

	GPU Utilization	Framerate
Leadwerks (OpenGL)	8%	52
Unity (DX11)	4%	40
Unity (Vulkan)	4%	40
Ultra Engine (Vulkan)	95%	1206

Animation Test

In this test, 1024 skinned character models are animated and displayed (see Figure 9). Each model maintains its own skeletal system, and there is no duplicated skeleton data shared across multiple instances. This test demonstrates the overhead the animation system incurs, the cost of sending animation data to the GPU, and the cost of weighting vertices with bone data in the vertex shader. We expected our new renderer to perform faster due to the reduced data transfer and the animation threading design, even though our vertex shader had become more complex.

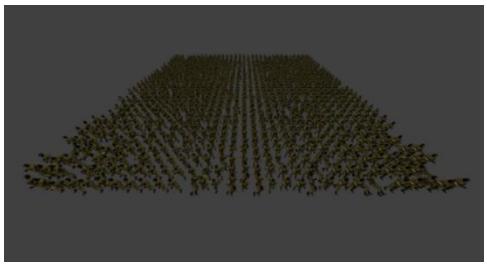

Figure 9. Animation test evaluates animation efficiency during high frequency rendering.

When this test was run in Leadwerks we observed just 1% GPU utilization and a framerate of five frames per second. The test ran in Unity with 45% GPU utilization and a framerate around 63. Our new Vulkan renderer ran with 96% GPU utilization and a framerate more than ten times higher than other engines tested (see Table 4).

Table 4. Animation Test Results

	GPU Utilization	Framerate
Leadwerks (OpenGL)	1%	5
Unity (DX11)	45%	62
Unity (Vulkan)	45%	64
Ultra Engine (Vulkan)	96%	1179

Lighting Test

This test provides a measure of the efficiency of our new clustered forward renderer versus the old deferred renderer. A grid of 25 point lights was arranged over a scene consisting of 968 instanced boxes (see Figure 10). Because the clustered forward renderer uses less memory bandwidth, we expected the new renderer to again demonstrate much faster performance.

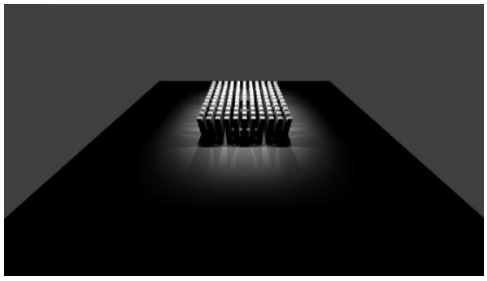

Figure 10. The lighting test evaluates efficiency when drawing large numbers of point lights.

When this test was run in Leadwerks we observed 49% GPU utilization and a framerate of 704. The same test ran in Unity with around 32% GPU utilization and a slightly higher framerate in DirectX11 than in Vulkan. Our new Vulkan renderer ran with 96% GPU utilization at 1456 FPS (see Table 5).

Table 5. Lighting Test Results

	GPU Utilization	Framerate
Leadwerks (OpenGL)	49%	704
Unity (DX11)	33%	90
Unity (Vulkan)	30%	74
Ultra Engine (Vulkan)	96%	1456

Unique Geometry Test

In this test, we created a grid of 4096 models that are copies, not instances, of a box shape (see Figure 11). Each box has identical geometry but is a unique collection of vertices and indices in memory. This test most closely simulates the conditions we would experience when drawing a complex scene with many unique objects. We expected our new renderer to outperform others due to the low overhead of our design.

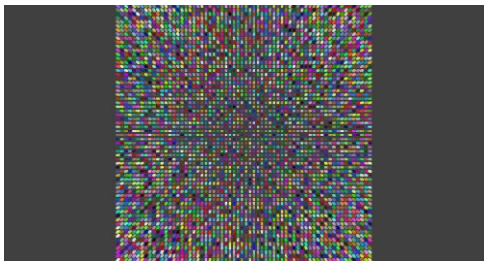

Figure 11. The draw call test simulates conditions for a complex scene made up of many unique objects.

When this test was run in the Leadwerks we observed 10% GPU utilization and a framerate of 49. In Unity we observed 4-5% GPU utilization and two frames per second, in both the DirectX11 and Vulkan renderer. Our new Vulkan renderer ran with 63% GPU utilization at 6213 frames per second (see Table 6).

Table 6. Unique Geometry Test Results

	GPU Utilization	FPS
Leadwerks (OpenGL)	10%	49
Unity (DX11)	5%	2
Unity (Vulkan)	4%	2
Ultra Engine (Vulkan)	63%	6213

CONCLUSIONS

We utilized a design framework based on an analysis of hardware to guide our renderer development. A series of performance benchmarks confirms that our resulting rendering architecture makes much more efficient use of the processing power available in graphics hardware, under a variety of conditions. These techniques can be used to develop multi-domain simulations (see Figure 12) and VR applications with faster framerates and lower system requirements.

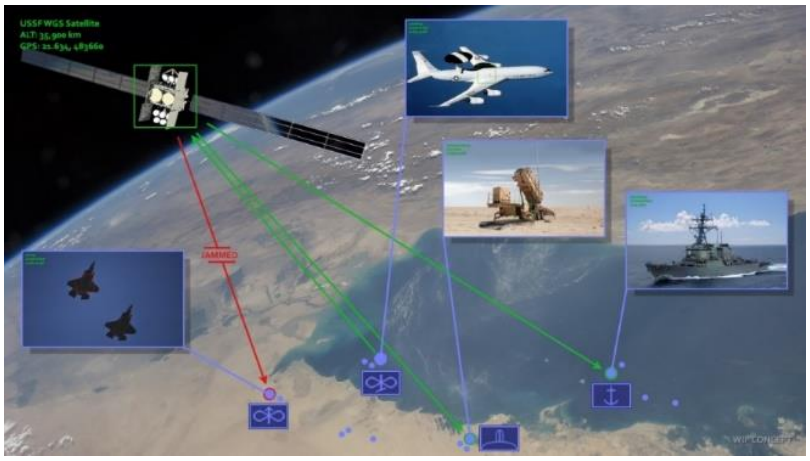

Figure 12. Multi-Domain Simulation, with insets of selected assets.

ACKNOWLEDGEMENTS

We would like to extend our gratitude to Goddard Space Flight Center, ATA Aerospace, Valve Corporation, Frank Klippenberg, and Eric Lengyel.

REFERENCES

Akenine-Möller T., Haines E., Hoffman N., 2018, *Real-Time Rendering (4th ed.),* A. K. Peters, Ltd., USA., pp. 127

DeLoura M. (2001) *Game Programming Gems 2*, Hingham, Massachusetts: Charles River Media, pp.169

Knarr D., (2019) *NVidia Graphics Card Specification Chart.* Retrieved May 15, 2020, from https://www.studio1productions.com/Articles/NVidia-GPU-Chart.htm

Olsson O., Billeter M., and Assarsson, U. (2012). Clustered deferred and forward shading. High-Performance Graphics 2012, HPG 2012 - ACM SIGGRAPH / Eurographics Symposium Proceedings. 87-96. 10.2312/EGGH/HPG12/087-096.

Pharr M. et al. (2005) *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, Upper Saddle River, NJ: Addison-Wesley Professional, pp.143

Smith M., Desnoyers-Stewart J., Kratzig G., (2019) *Designing Virtual Reality Tools: making simulated interventions feel and act like their real counterparts*, Interservice/Industry Training, Simulation, and Education Conference, pp.2, 2019

Whitehead N., Fit-Florea A., (2011) *Precision & performance: floating point and IEEE 754 compliance for NVIDIA GPUs.* Computer Science, pp.4, 2011