PERFORMANCE TECHNIQUES FOR RENDERING PROCEDURALLY GENERATED TERRAIN IN VIRTUAL REALITY USING RAY MARCHING

BY

MICHAEL RICHARD BROWN

A thesis submitted to the Department of Computer Science

in partial fulfillment of the requirements

for the degree

BACHELOR OF SCIENCE

Subject: Computer Science

NEW MEXICO STATE UNIVERSITY

LAS CRUCES, NEW MEXICO

DECEMBER 2017

VITA

1994	Graduated from Oregon High School Oregon, Wisconsin
1994-1995	Lead Software Engineer BOSS International, Madison, Wisconsin
1995-2000	Director of Information Systems BOSS International, Madison, Wisconsin (Acquired by Autodesk, Inc.)
2007-2014	Senior Software Engineer, NASA TDRSS Honeywell TSI, NENS Contract Computer Sciences Corporation, SCNS Contract NASA White Sands Complex, Las Cruces, New Mexico

Awards

2007, 2008, 2010	Bronze Bravo Award Honeywell TSI
2012	North American Public Sector President's Award Computer Sciences Corporation
2013	NASA Space Flight Awareness Honoree Award

ABSTRACT

PERFORMANCE TECHNIQUES FOR RENDERING PROCEDURALLY GENERATED TERRAIN IN VIRTUAL REALITY USING RAY MARCHING

BY

MICHAEL RICHARD BROWN

NEW MEXICO STATE UNIVERSITY

LAS CRUCES, NEW MEXICO

DECEMBER 2017

Rendering three-dimensional environments for virtual reality headsets presents significant performance challenges over the traditional rendering done for standard two-dimensional displays. Stereoscopic rendering requires that scenes be rendered from two different perspectives and potentially twice the number of combine pixels when compared to standard displays. Additionally, a minimum frame rate of 90 frames per second must be maintained in virtual reality to prevent disorientation and nausea caused when the brain detects conflicts between head movement and visual perception. This thesis demonstrates techniques which can be used to render world-scale procedurally generated terrain in virtual reality while achieving the performance requirements necessary to maintain comfort.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

INTRODUCTION

Virtual reality has had a recent resurgence with the availability of the Oculus Rift and HTC Vive stereoscopic head mounted displays. These headsets have brought the costs of experiencing immersive three-dimensional environments down to the level of average consumers and gaming enthusiasts. However, developing software for them presents new challenges over traditional monoscopic displays.

Studies have shown that virtual reality can induce simulator sickness. Simulator sickness is disorientation, nausea, headaches, and other negative side-effects when insufficient frame rates cause flicker and visual lag, inducing a conflict between the visual and vestibular senses. To prevent these problems, a frame rate equal to the refresh rate of the headset display must be consistently maintained (Kolasinski, 1995). The Oculus Rift and HTC Vive have displays with 90 Hz refresh rates, so software must render scenes at a minimum of 90 fps (frames per second) to match their refresh rate.

When rendering software is not able to maintain 90 fps, the runtime software for the Oculus Rift and HTC Vive can employ techniques such as interleaved reprojection, asynchronous reprojection (called asynchronous timewarp by Oculus), or asynchronous spacewarp to attempt to maintain the appearance of 90 fps by projecting old frames using updated head tracking information. However, these mitigation strategies come with their own visual artifacts, which can also conflict with vestibular senses and lead to uncomfortable experiences. It is recommended that they be used to fill in the occasionally skipped frame, and not as a primary method to achieve consistent frame rates (Antonov, 2015). And to ensure a comfortable user experience for their customers, the Oculus Store software distribution policies requires that software displays graphics in the headset at 90 fps, with no significant frame rate drops (Oculus VR, 2017b).

In addition to high frame rates, the Oculus Rift and HTC Vive also have displays with higher rendering resolutions than the 1080p display used on the average gaming PC (Valve Corporation, 2017). Due to the combination of higher rendering resolution and 90 fps requirements, virtual reality software may need to render between 6 and 8 times more pixels per second than monoscopic software when rendering the same scene (Table 1). Software developers must make trade-offs in detail by simplifying geometry and shaders to maintain this pixel fill rate. These compromises and limitations result in virtual reality environments which are simplistic, cartoonish, or rely on pre-rendered content (such as 360-degree video).

Table 1: Comparison of display resolutions and minimum framerates

Display Method	Rendering Resolution	Pixels	Minimum Framerate	Pixels per Second
720p	1280 x 720	921,600	30 fps (generally acceptable)	27,648,000
1080p	1920 x 1080	2,073,600		62,208,000
Oculus Rift	2700 x 1600	4,320,000	90 fps	388,800,000
HTC Vive	3026 x 1680	5,083,680		457,531,200

Rendering complex world-scale terrain is especially difficult to achieve in real-time because it must have a high level of detail close to the observer, but be visible from very long distances. In traditional rasterization, a tessellated mesh must be created with sufficient resolution to provide detail at both distances. This mesh must be continuously modified or generated as the observer moves through the terrain to ensure that the appropriate level of detail is maintained at each visible distance (Cervin, 2013).

During the tessellation process, large numbers of triangles are generated to match the variation of terrain at sufficient level of detail. Many of the triangles that are generated are not rendered because they are on opposite sides of the hills and facing away from the camera and therefore back-face culled. Pixels may be overdrawn multiple times as nearer terrain obscures terrain drawn farther in the distance. The performance of traditional rasterization of terrain depends on the number of triangles generated during the process of tessellation, how many triangles are culled and clipped by screen space, how many times rendered pixels are overdrawn by overlapping triangles, and the computational expense of rendering each pixel. Rasterizing world-scale terrain is within the processing ability of modern GPUs when rendering to monoscopic displays, but not at the consistent fill rates demanded by virtual reality.

Ray marching is similar to ray tracing in that a ray is cast from the observer through a pixel to find the surface to be rendered in a scene. It is used when intersections with the surfaces cannot be mathematically determined by an equation but must be empirically determined through a series of steps. At each step along the ray, a test is done to determine whether the ray has intersected a surface. When an intersection has been found, the surface is rendered to the pixel (Hart, 1989). Ray marching is rarely used in interactive graphics software because it is typically more computationally expensive than rasterization of polygons, especially on commercial GPU hardware designed specifically to rasterize texture mapped triangles (Macedo, 2015).

However, if some optimization techniques are employed, ray marching has unique advantages over rasterization for rendering procedurally generated terrain over large distances in virtual reality. Noisy surfaces such as rock faces, vegetation, and water waves do not require high resolution meshes to be generated. The number of computations needed per frame is very closely related to the number of pixels drawn, and the number of marched steps which must be taken per pixel to find the line-of-sight intersection with the terrain (Jiarathanakul, 2012). Unlike rasterization, ray marching can limit rendering to the pixels on the polygonal surface used to display it, and there is no overdraw in ray marching; every pixel is only rendered once.

This thesis will demonstrate how ray marching, combined with two optimization methods to reduce the number of pixels rendered and to reduce the number of steps necessary to render each pixel, allows rendering of world-scale procedurally generated terrain in virtual reality while achieving the frame rate requirements necessary to avoid simulator sickness.

METHODS

Iris

A common feature of virtual reality headsets is their limited elliptical field of view. The Oculus Rift has an estimated 94° horizontal by 93° vertical field of view. The HTC Vive has an estimated 100° by 113° field of view (Kreylos, 2016). Beyond this elliptical field of view, rendered pixels in the remainder of the rectangular display are not visible, which means a large portion of pixels in the scene could be eliminated from rendering.

With traditional rasterized rendering, the entire rectangular clip space of the display is rendered. In many cases, ray marching is also performed from pixels rendered on the surface of a quad (two triangles) which fills the entire display. However, to prevent rendering pixels outside the field of view in virtual reality headsets, ray marching can be performed on the surface of a suitably elliptical shape which only encompasses the field of view.

For the purposes of simplicity, an octagonal iris comprised of six triangles was implemented to demonstrate the utility of this optimization (Figure 1). A more sophisticated implementation could detect the use of particular headsets and use a surface which more closely matches the field of view and geometry of the detected headset to further eliminate additional pixels from rendering.

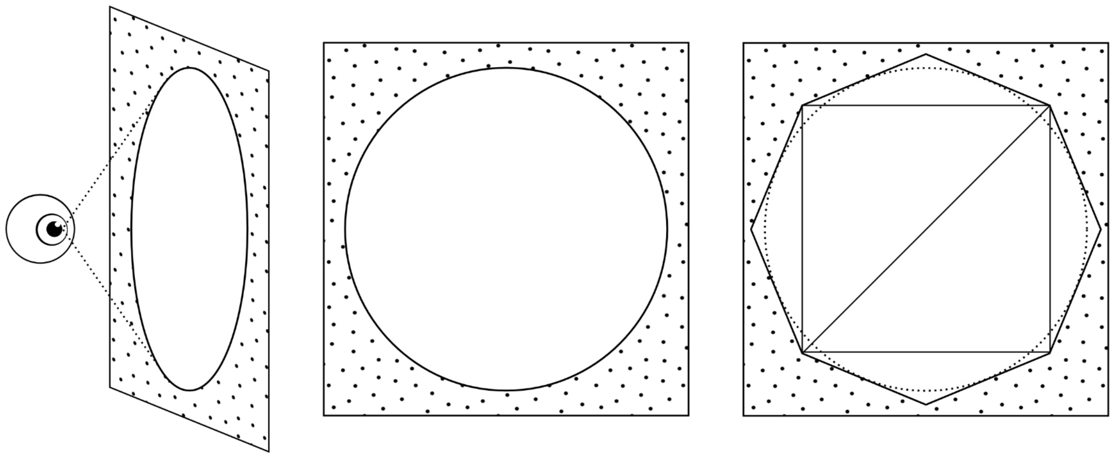

Figure 1: An octagonal iris can limit rendering of the scene to the limited field of view seen in a virtual reality headset. The dotted area is not rendered.

Distance Hints

At each step of the ray marching process, an adaptive determination of the next step size is made, based the estimated distance to the surface at the current step. Step size determination is essential to the efficiency and quality of the process (Jiarathanakul, 2012). Step sizes which are too large will miss or penetrate the surface too deeply and produce distortions. Step sizes which are to small will result in unnecessary steps and slow the process.

Height of the current step above the terrain's surface is the fundamental to the determination of step sizes. Other factors such as terrain slope, ray distance from the observer, minimum or maximum step sizes, and cached maximum height zones may also be used (at some computational cost) to make improvements to the estimation, but height above terrain remains the primary factor (Tevs, 2008).

Rays that march close to the surface of terrain for long distances will take a larger number of small steps than rays which remain high above the terrain they follow (Figure 2). If a maximum limit on the number of steps is enforced to ensure consistent rendering performance, then a ray will terminate unsuccessfully and leave empty artifacts when the limit is reached (Figure 3 and 4). If no limit exists, then rendering times can increase dramatically in scenes when the observer is looking horizontally over large flat areas, which can be problematic in virtual reality.

Limiting the number of steps taken per pixel can guarantee a maximum rendering time per frame, but because it causes artifacts, reducing the number of steps needed to find the terrain intersection is preferable. The optimization technique for reducing the number of ray marching steps per pixel investigated in this thesis involves generating and storing distance estimates as hints before the ray marching process begins, and then using those hints as starting points. With distance hints, the ray marching process can begin at a location which is much nearer to the actual intersection with terrain, rather than at the observer.

To generate the distance hints, a low-resolution mesh of the terrain is rendered using the camera's point of view to a texture using a shader that stores distances instead of colors (Algorithm 1). The hint mesh is offset above the terrain (and any features like vegetation or trees) to prevent hints which might cause the ray marching to miss or penetrate the terrain too deeply. The resulting distance hint texture is similar to a depth buffer, but it is created before terrain rendering, and the distances are actual, not scaled between the near and far clipping planes.

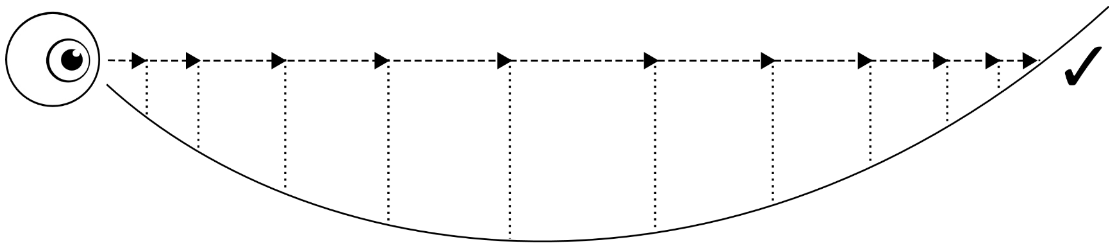

Figure 2: Ray marching above terrain most efficient when the height above the terrain is large enough to allow long step deltas.

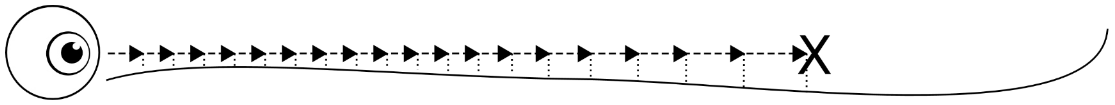

Figure 3: Ray marching above terrain may terminate unsuccessfully when the ray follows the terrain closely for long distances and small step deltas are taken.

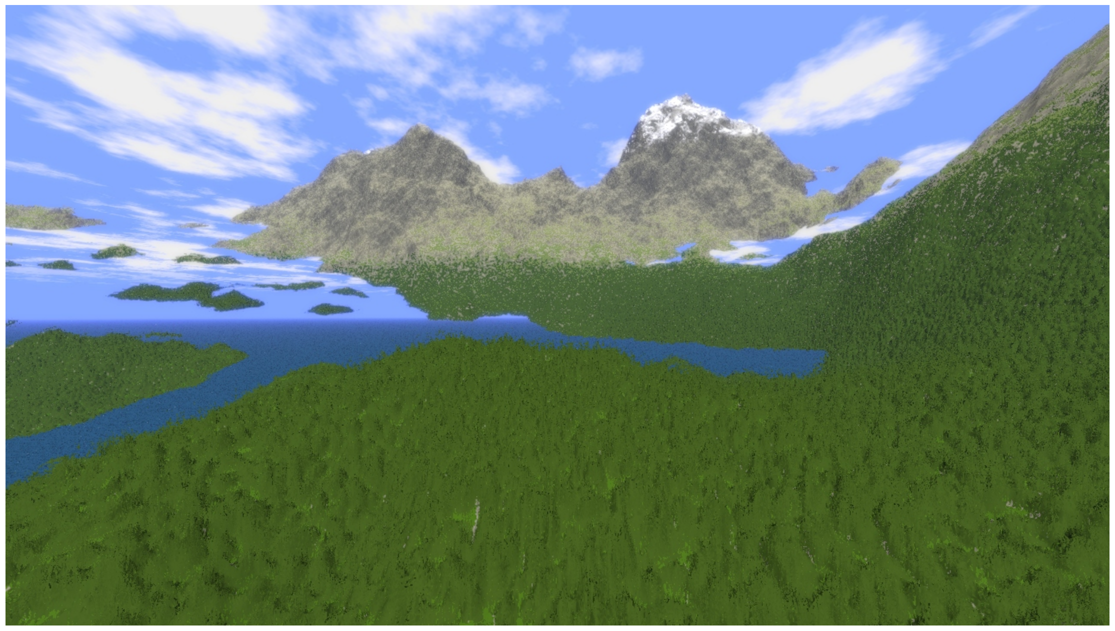
Figure 4: Artifacts similar to mirages or floating islands may appear when ray marching terminates unsuccessfully before an intersection can be found.

Algorithm 1: Shader to deform a flat mesh to match the geometry of terrain, and then project and render distance hints from that mesh to a render texture.

```
// Received positions of model vertexes
struct appdata
{
  float4 vertex : POSITION;
};

// Vertex to fragment structure
struct v2f
{
  float4 pos : SV_POSITION;
  float distance : TANGENT;
};

v2f vert(appdata v)
{
  // Apply model matrix to get the world position of each vertex
  // in the mesh.
  float4 world_pos = mul(unity_ObjectToWorld, v.vertex);

  // Set the height of each vertex in the mesh according to its
  // location on the terrain. Apply a height offset to allow for
  // surface attributes (like vegetation) to protrude above the
  // terrain.
  world_pos.y = getTerrainHeight(world_pos.xz) + hintHeightOffset;

  // Get the distance from the vertex to the camera
  float distance = length(_WorldSpaceCameraPos.xyz - world_pos.xyz);

  v2f o;
  // Apply the view projection matrix to pass on the projection
  // space position of the vertex.
  o.pos = mul(UNITY_MATRIX_VP, world_pos);
  // Pass on the the distance to the camera.
  o.distance = distance;
  return o;
}

float4 frag(v2f i) : SV_Target
{
  // If the pixel is drawn to the render texture, the red channel
  // will indicate the distance to the terrain at that location in
  // screen space. The green, blue, and alpha channels are set to
  // one to indicate the hint is valid. Otherwise, the camera clears
  // them to zero.
  return float4(i.distance, 1.0f, 1.0f, 1.0f);
}
```

When the ray marching algorithm begins at each pixel, it first checks the distance hint texture using the pixel's screen coordinates to see if a valid hint is available (Algorithm 2). If a valid hint is found, then the ray marching can begin at the hint distance instead of the observer and skip the intermediate steps (Figure 5). If no hint is available, the ray marching can still start from the observer.

When there is no hint, the ray will probably not intersect terrain, and the sky will be rendered, but in some cases, the low-resolution hint mesh does not adequately represent the terrain. In these cases, the normal ray marching process will find an intersection with the terrain.

In some cases, the hints will be premature, because the hint mesh is slightly higher than the terrain it represents, and so distance hints just above the edge of some terrain will contain distances to the edge rather than the terrain behind it (Figure 6).

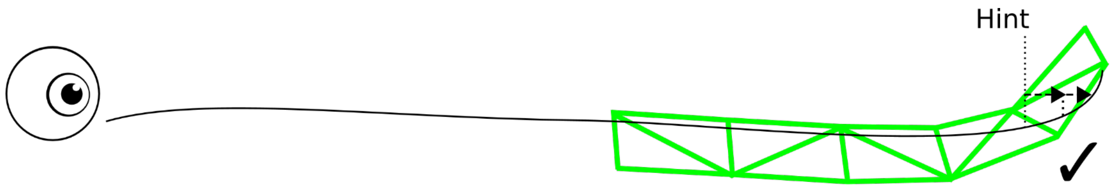

Figure 5: Distance hints from a mesh rendered to a texture as distances can be used to reduce the number of ray marching steps required to find an intersection with terrain.

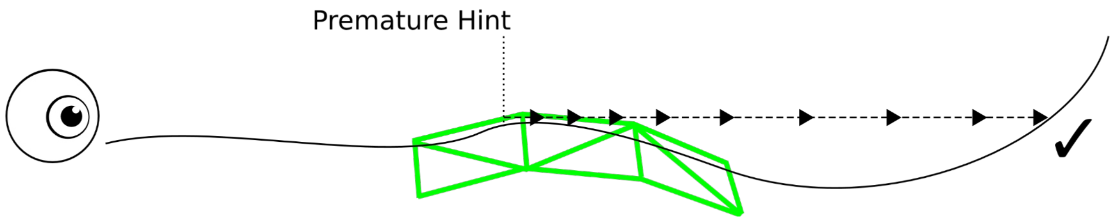

Figure 6: Premature hints can cause ray marching to start too soon, but this still reduces the number of ray marching steps.

Algorithm 2: Ray marching hit detection algorithm using distance hints retrieved from a texture by screen coordinates

```
rayMarchTerrain()
{
  hit = false;

  stepDistance = getTerrainDistanceHint(screenCoordinates);

  for (terrainMaxSteps)
  {
    stepPosition = cameraPosition + (stepDistance * rayDirection);

    height = getHeightAboveTerrain(stepPosition);

    if (height < terrainHitEpsilon)
    {
      hit = true;
      break;
    }

    // Delta to next step is based on the current step's height
    // above terrain and distance from camera. When the ray skims
    // close to the terrain, the steps are small and the number of
    // allowed steps may be exhausted.
    heightDelta = height * heightDeltaScale;
    distanceDelta = stepDistance * distanceDeltaScale;
    stepDelta = max(stepDeltaMin, heightDelta) + distanceDelta;
    stepDistance += stepDelta;
  }

  return (hit, stepDistance);
}
```

TESTS

Test software was created using the Unity game engine to render procedurally generated terrain using ray marching and record the performance statistics with and without the optimizations enabled to determine the effectiveness of these techniques. Each test traverses the same 50 km of terrain at 90 meters per second, rendering 15 meters above ground level with a horizontal view. The traversal subjects the rendering and optimizations to a variety of different terrain topologies. This test is representative of the worst-case rendering times because ray marching that follows closely and horizontally over terrain requires the most steps to find intersecting surfaces.

All tests were conducted using a Windows 10 PC with an AMD A10-6700 3.7 GHz CPU, 8 GB main memory, NVIDIA GeForce GTX 970 GPU, and 4 GB display memory. This CPU performs below the minimum specifications that are recommended for virtual reality by Oculus and HTC. The GPU and memory does meet the Oculus and HTC recommended specifications for virtual reality (Oculus VR, 2017a) (HTC, 2017). Since the tests and optimization techniques were designed to use the GPU and test rendering performance, the test software included minimal scripting and no physics which would utilize the CPU. Unity profiling tools indicated that the CPU was at less than 30% usage at all times during the tests, and that the rendering times were bound to GPU (not CPU) usage.

Monoscopic tests were performed at full screen 1080p to demonstrate the effectiveness of using distance hints without the overhead of virtual reality runtimes, and without the frame rate capped at 90 fps by the 90 Hz refresh rates of virtual reality headsets. The rendering time between frames was recorded in milliseconds to a log file.

Tests were also performed with an HTC Vive as the rendering target to capture the effectiveness of both the iris and distance hints optimization in virtual reality. This test recorded the amount of time spent rendering on the GPU in milliseconds (excluding the time waiting for the 90 Hz display refresh to complete), the number of frames that were reprojected by the virtual reality runtime when a new frame was not available in time, and the number of frames that were dropped because a reprojection was needed but could not be performed in time.

The test uses a distance hint mesh with 63,600 vertices. The distance hint texture is a square 1024 x 1024 pixel texture sampled without filtering. This texture is smaller than both 1080p and HTC Vive rendering resolutions. According to Unity profiling, it took 0.2 milliseconds to render the distance hint mesh to the distance hint texture. For virtual reality tests, the hint texture was generated twice, once for each eye, taking a total of 0.4 milliseconds to render.

RESULTS

The distance hint optimization produced a significant performance improvement in the mean, median, and maximum rendering times on the 1080p monoscopic display (Figure 7). The mean and median rendering times were both reduced by 2.1 milliseconds, for a 37% and 35% performance increase, respectively.

The maximum rendering time was reduced by 8.8 milliseconds, for a 49% performance increase. The worst framerate drops that had rendered at 37 fps (27 milliseconds per frame) without the distance hints could render at 55 fps (18.2 milliseconds per frame) with distance hints.

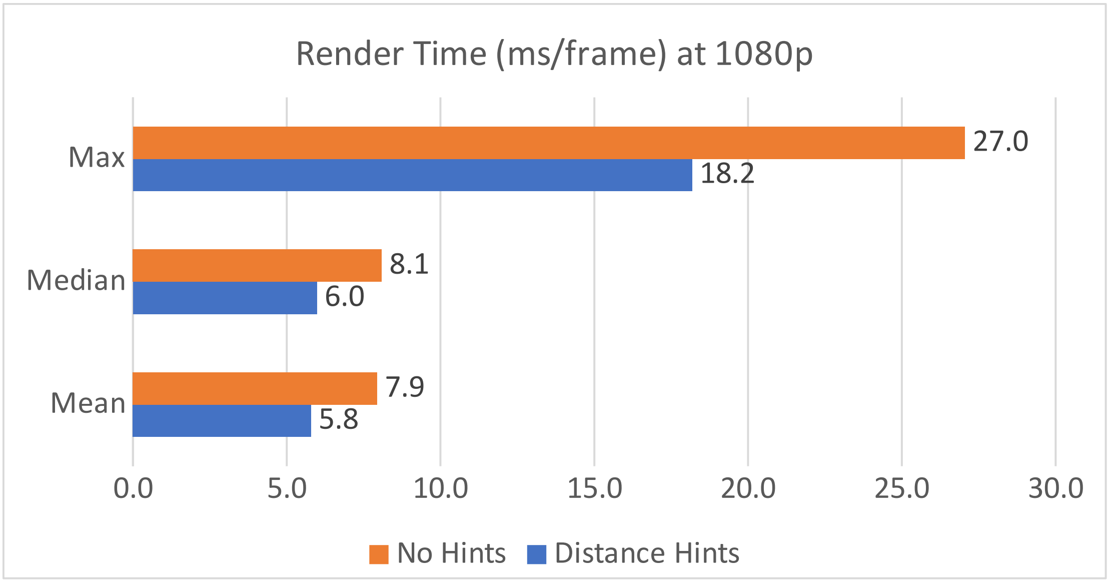

Figure 7: Rendering times for ray marching terrain at 1080p with and without distance hints using an NVIDIA GeForce GTX 970 GPU.

The distance hint optimization also produced a significant performance improvement on the HTC Vive, although not quite as much as on the 1080p monoscopic display (Figure 8). The mean and maximum rendering times were both reduced by 7.9 and 18.4 milliseconds, for a 32% and 38% performance increase, respectively.

Individually, the iris optimization produced a much more impressive performance improvement than the distance hint optimization on the HTC Vive (Figure 8). The mean and maximum rendering times were both reduced by 22.5 and 51.4 milliseconds, for a 220% and 327% performance increase, respectively.

Combined, the iris and distance hint optimizations were able to reduce the mean and maximum rendering times on the HTC Vive by 24.6 and 54.6 milliseconds, for a 302% and 435% performance increase, respectively. This brought the majority of rendering times below the 90 fps threshold of 11.1 milliseconds per frame. As a result, the number of dropped and reprojected frames were substantially reduced when both optimizations were used together (Figure 9).

In general, no reprojections were necessary if the GPU rendering time was below 10.6 milliseconds, because this provided the HTC Vive virtual reality runtime sufficient time for its processing overhead. Dropped frames tended to occur when the GPU rendering time exceeded 28 milliseconds or more.

CONCLUSION

Limiting rendering to an iris encompassing the virtual reality headset field of view had a very dramatic performance benefit with almost no perceptible difference in user experience. This optimization is generally limited to use with virtual reality headsets, but it is not limited to ray marching terrain. It is likely to be a useful strategy for optimizing ray marching other types of scenes, or with other similar rendering schemes like ray tracing.

Using distance hints to optimize ray marching terrain had no visible negative side effects. Moreover, the artifacts (mirages and floating islands) which were occasionally seen when the ray marching reached a maximum limit of steps without distance hints were eliminated when the distance hint optimization was employed. The cost of rendering the distance hint texture before rendering the terrain is clearly worthwhile when ray marching procedurally generated terrain. It may also have some utility when ray marching other types of surfaces or objects which can be represented or enclosed in low-resolution meshes.

With both the iris and distance hint optimizations combined, procedurally generated world-scale terrain can be rendered using ray marching on modern, commercially available virtual reality hardware while achieving the frame rate requirements necessary to avoid simulator sickness.

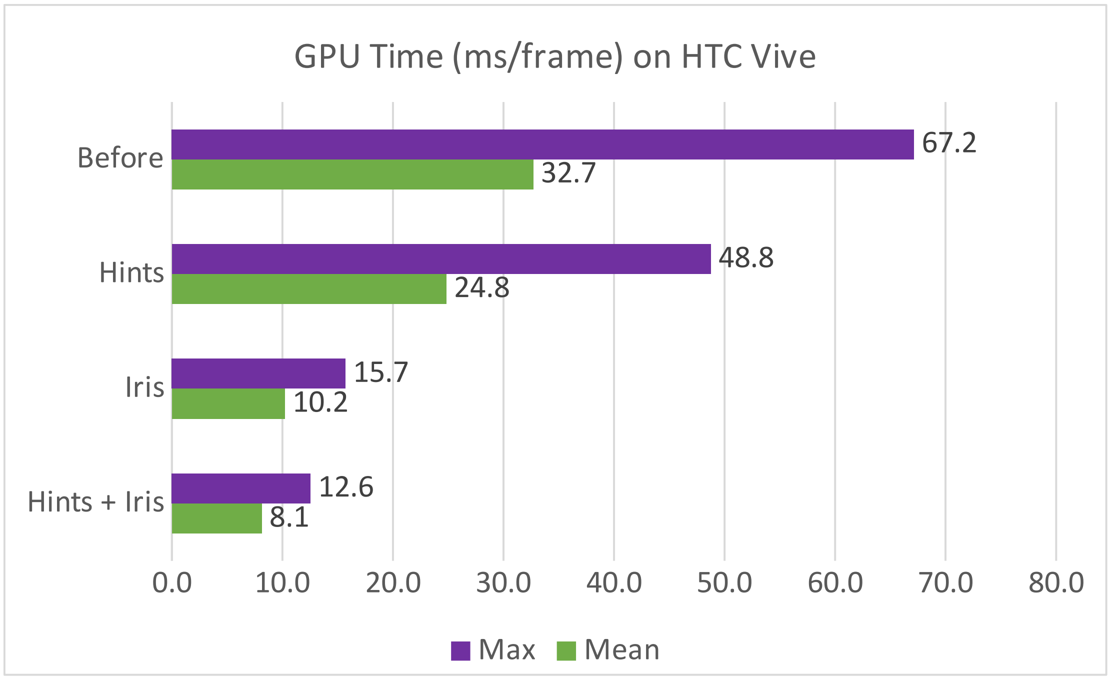

Figure 8: GPU times before and after applying the iris and distance hint optimization techniques to ray marching terrain on the HTC Vive.

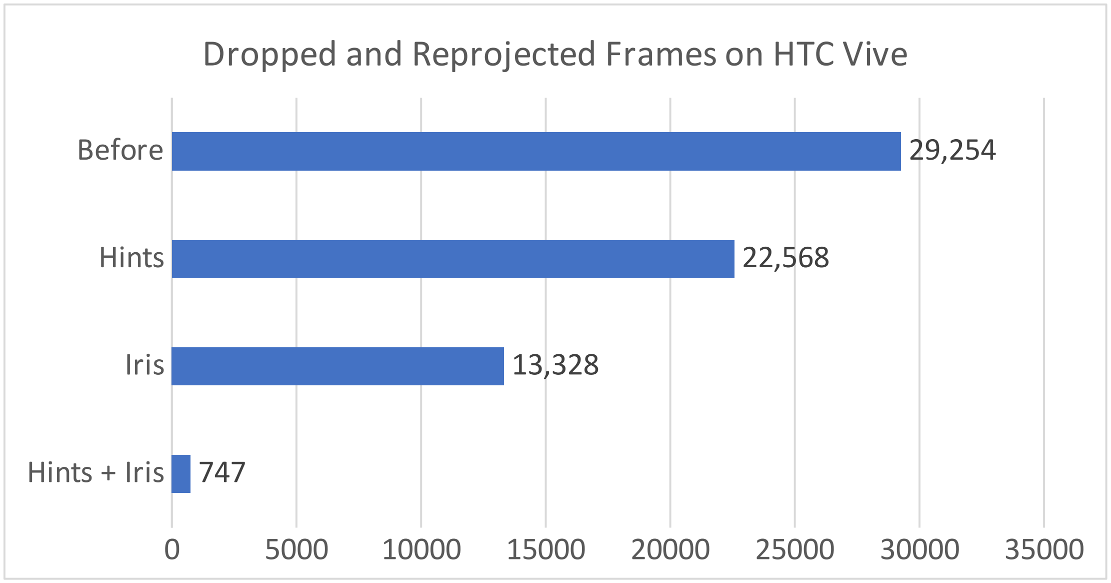

Figure 9: The number of frames reprojected or dropped before and after applying the iris and distance hint optimization techniques to ray marching terrain on the HTC Vive.

REFERENCES

Antonov, M. (2015, March 2). *Asynchronous Timewarp Examined.* Retrieved from Oculus Developer Center: https://developer.oculus.com/blog/asynchronous-timewarp-examined/

Cervin, A. (2013). *Adaptive Hardware-accelerated Terrain Tessellation.* Norrköping: Linköping University.

Hart, J. C. (1989). Ray Tracing Deterministic 3-D Fractals. *SIGGRAPH '89 Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (pp. 289-296). New York: ACM.

HTC. (2017). *Vive Ready Computers*. Retrieved from VIVE: https://www.vive.com/us/ready/

Jiarathanakul, P. (2012). *Ray Marching Distance Fields in Real-time on WebGL.* Philadelphia: University of Pennsylvania.

Kolasinski, E. (1995). *Simulator Sickness in Virtual Environments.* Alexandria: Army Research Institute for the Behavioral and Social Sciences.

Kreylos, O. (2016, April 1). *Optical Properties of Current VR HMDs*. Retrieved from http://doc-ok.org/?p=1414

Macedo, D. R. (2015, November 11). Realistic Rendering in 3D Walkthroughs with High Quality Fast Reflections. *SBC - Proceedings of SBGames*, pp. 9-15.

Oculus VR. (2017a). *Recommended Specs*. Retrieved from Oculus Support Center: https://support.oculus.com/170128916778795/

Oculus VR. (2017b). *Rift Performance VRCs.* Retrieved from Oculus Developer Center: https://developer.oculus.com/distribute/latest/concepts/publish-rift-app-submission-performance/

Tevs, A. I. (2008). Maximum Mipmaps for Fast, Accurate, and Scalable Dynamic Height Field Rendering. *I3D '08 Proceedings of the 2008 symposium on Interactive 3D graphics and games* (pp. 183-190). New York: ACM.

Valve Corporation. (2017, November). *Steam Hardware & Software Survey*. Retrieved from http://store.steampowered.com/hwsurvey