https://doi.org/10.2352/EI.2023.35.14.COIMG-169

BowTie Rasterization for Extreme Multi-view Light-field Displays

Thomas Burnett, Justin Halter, Justin Jensen

Abstract

Radiance image rasterization is the process by which a 3D scene is rendered into a light-field radiance image. Every pixel in the light-field radiance image represents a unique ray passing through a 3D volume of space. Radiance image rasterization is an example of extreme multi-view rendering as the 3D scene must be rendered from many points of view (1000s to millions) into small viewports per update of the light-field display. However, GPUs and their accompanying APIs (OpenGL, DirectX, Vulkan) generally expect to render a scene from one point of view to a single large viewport/framebuffer. Therefore, light-field radiance image rendering is extremely time consuming and a compute intensive serial rendering process. This paper reviews the full-parallax, BowTie Radiance Image Rasterization process and builds upon the concept of a Multi-view Processing Unit (MvPU) embedded within the light-field display architecture in order to accelerate light-field radiance image rendering.

Introduction

Emerging light-field display (LfD) technology provides a glasses-free 3D aerial image with the depth cues expected by the human visual system; therefore, the visual experience for the observer is natural and without nausea-inducing artifacts. A light-field can be described as a set of rays that pass through a 3D volume of space. The light-field radiance image (example radiance images can be seen in Figure 9) is a raster description of a light-field where every pixel in the image represents a unique ray within that 3D volume. The light-field display radiance image can be projected through an array of *Hogel (*Holographic Element) micro-lenses to reconstruct a perspective-correct 3D aerial image visible for all viewers within the display's projection frustum. Within this paper, the term *Hogel* will be used to represent both a micro-lens and the accompanying micro-image. The micro-image consists of all the perspective rays that pass through that point spot on the light-field (hogel) image plane; the micro-lenses are used to angularly distribute the light over a projection field-of-view (FoV).

Light-field radiance image rendering is an example of extreme multi-view rendering where a scene must be rendered from many (thousands to millions) viewpoints per update of the display. While a GPU can be used to generate a light-field radiance image, the traditional GPU raster pipeline expects to render a scene from a single viewpoint per dispatch of the scene geometry. Therefore, the burden of radiance image rendering falls to the host 3D application, which must understand the exact nature of the LfD's projection system and render all the appropriate views sequentially. For every view rendered, the host application sets the camera view/projection matrix, the viewport to render to, and redispatches the scene's render commands. As a result, radiance image rendering can require exceedingly long computation times whereby the LfD is unresponsive in the meantime. Therefore, the update rate of the LfD and thus the power required to render animated content is a factor of the render algorithm, scene complexity and the number of scene dispatches (renders/views) that a computation engine incurs to update a display at the desired framerate.

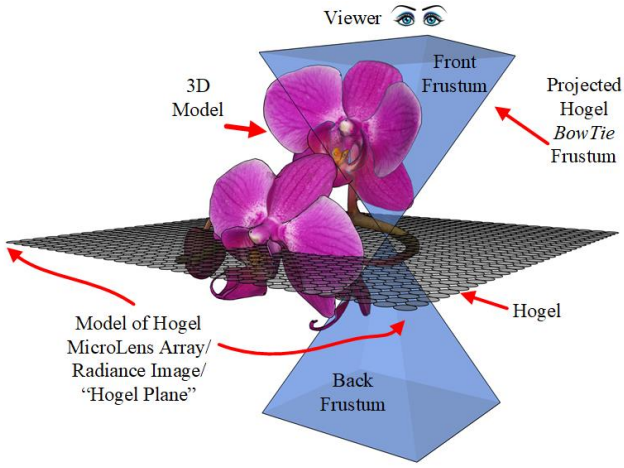

FIGURE 1. HOGEL IMAGE PLANE WITH 3D MODEL AND BOWTIE FRUSTUM

Related Work

In [1] on rasterizing synthetic radiance images, both the *Double Frustum* [3] and *Oblique Slice and Dice* full-parallax, light-field radiance image rasterization algorithms were reviewed. The major difference between the two algorithms is the order in which the 4D light-field is decomposed into 2D render passes. The *Double Frustum* algorithm renders hogel micro-images using two independent back and front perspective frustums. The *Oblique Slice and Dice* algorithm renders directions using sheared orthographic projections; after which, every oblique pixel must be transformed/swizzled and/or sampled into hogel micro-images.

It was noted in those papers that a custom rasterization pipeline specifically designed for extreme multi-view rendering could improve full-parallax, radiance image rendering performance. This paper reviews a custom *BowTie* renderer designed to take advantage of performance optimizations inherent in extreme multi-view, full-parallax, synthetic radiance image rendering. The *BowTie* radiance image renderer uses a single bowtie/pinhole projection matrix and inverts the triangle/view rendering priority. Therefore, this paper builds upon the concept of a *Multi-view Processing Unit* (MvPU) (introduced in [1]), a GPU like device embedded in the LfD architecture and tailored specifically to the rendering needs of a large format 3D light-field display.

Double Frustum Epsilon Region

The *Double Frustum* algorithm has one notable drawback, at least when implemented in OpenGL with the traditional perspective camera matrix. In OpenGL, the perspective camera matrix cannot be defined with a near plane on or behind the camera origin. Rather, the near plane is defined at a positive offset, with expectation that the viewport is mapped to the near plane.

As shown in Figure 2, if the front and back frustum definitions share the same origin, then there exists a small region between the two frusta near planes that is not seen by either camera: the *Epsilon* region. Portions of triangles that pass through the *Epsilon* region are not rendered, resulting in un-rasterized portions of the hogel micro-image. The noticeable solution is to back offset both cameras so that the near planes are co-planar and keep the near plane offset small [3]. This does alleviate most of the hogel corruption but is still not a perfect solution.

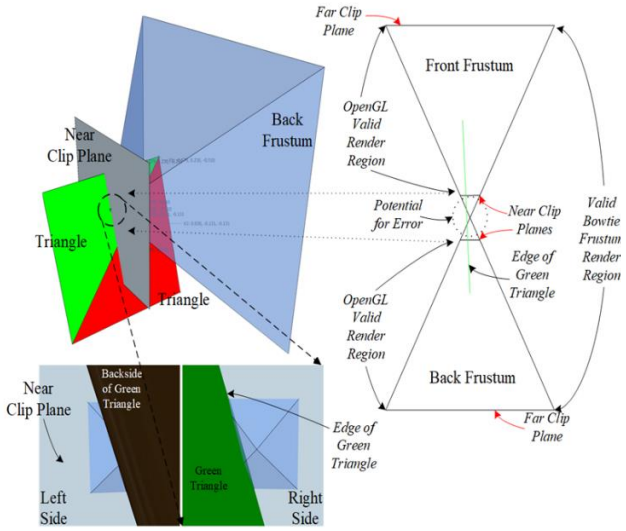

FIGURE 2. THE DOUBLE FRUSTUM EPSILON REGION; A SOURCE OF HOGEL CORRUPTION

Figure 3A sets up a model where triangles pass through the coplanar near planes of a *Double Frustum* camera pair which results in the un-rasterized corner portions of the 5x5 hogel radiance image shown in Figure 3B. Figure 3B was rendered using the *Double Frustum* algorithm, with coplanar near planes on a GPU in OpenGL. Figure 3C was rendered with the *Double Frustum* algorithm in FoVI[3D]'s MvPU simulator using the same camera definitions. For clarity, the MvPU simulator clears the background to an uncommon shade of pink to better highlight hogel corruption and/or un-rasterized/un-shaded pixels. Figure 3D was rendered using the *BowTie* algorithm within the MvPU simulator and shows the corner hogels correctly rendered as the *BowTie* projection has no *Epsilon* region.

BowTie Radiance Image Rasterization

The *BowTie* hogel camera uses a perspective projection, which can be defined with a positive far plane in front, and a negative near plane behind. However, the *BowTie* frustum can also be defined without either a near or far plane as shown in Figure 4 and which also implies that the four remaining clip planes define an infinite fore and aft hourglass or "bowtie" frustum. As such, the *BowTie* frustum is essentially an invertible pinhole projection, bisected by the hogel image plane (Figure 1).

Since the *BowTie* frustum planes have different normals above and below the image plane, the plane equations used for triangle culling/clipping operations are different for the front and back halves of the *BowTie* frustum. This can be accounted for in code, or by use of two sets of plane equations, a set for the front frustum and another set for the back frustum. The hogel image plane is itself a plane whose plane equation can determine whether clipping should occur by use of the front, back or both sets of clipping planes. This test can be done once per triangle (per object) per render cycle and the result cached for subsequent *BowTie* triangle/frustum clipping operations.

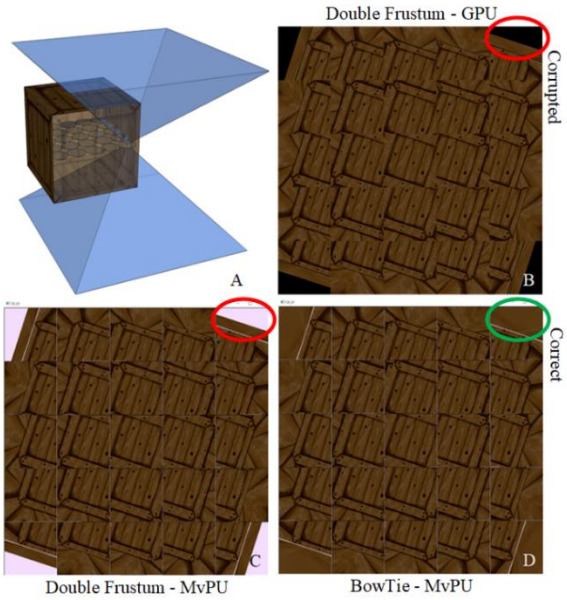

FIGURE 3. AN EXAMPLE OF DOUBLE FRUSTUM HOGEL CORRUPTION; THE BOWTIE RASTERIZATION RENDERS THE RADIANCE IMAGE CORRECTLY

FoVI[3D] defines the hogel camera (model) matrices facing up along the positive y-axis with the corresponding up-vector along the negative z-axis; the right-vector lies along the positive x-axis. Therefore, the hogel plane is defined on the x-z plane, normalized, and centered between (-0.5, -0.5) and (0.5, 0.5). Hogels are numbered from left to right, top to bottom, an also assigned a center coordinate in normalized viewport space. The 2D array of hogel camera matrices and accompanying viewport centers that define the radiance image rendering view definitions is referred to as the *Hogel Plane Definition* (HPD); the HPD is unique to the physical LfD requirements, i.e. number of hogels, number of rays per hogel, etc.

e = 1 / tan(fov/2)

e	0	0	0
0	e	0	0
0	0	0	-1
0	0	-1	0

FIGURE 4. BOWTIE PROJECTION MATRIX

The *View Volume Transform* (VVT) is a 4x4 transform matrix that defines the 3D cuboid volume in world space to be rendered. In other words, the VVT defines the 3D cuboid volume within a scene that a volumetric, light-field, or holographic display projects. Multiplying the hogel camera matrices defined within the HPD by the VVT transforms the hogel cameras into world space. Figure 9

shows the VVTs rendered as a transparent green cuboid, bisected by the hogel plane rendered in red.

Order of Operations: View vs. Triangle Major Rendering

The traditional render system sets the viewport and then renders all the triangles from all the scene objects onto it. This is view-major rendering. If there are many objects in a scene (as there often are) with unique vertex lists, textures, materials, and so forth, then there is the potential that the same object data is being constantly swapped in and out of the processor cache, possibly on a per-view or per-hogel basis.

As the view-triangle relationship is essentially a 2D array of operations, then processing the triangles against views may make better use of the cache and increase rendering efficiency. Therefore, theoretically, an object's definition is loaded once per render cycle into processor cache and an object's triangles are rendered in turn against all the views. This is triangle-major rendering.

The order of the operation for the *Double Frustum*, *Oblique Slice and Dice* and *BowTie* radiance rendering algorithms is outlined in Figure 5. In addition, each line in the outline is annotated with a processing unit label (C: CPU, G: GPU and M: MvPU) in which the processing typically occurs.

Double Frustum

```
C: For Every Hogel Micro-Image
C:   For Every Object
G:     For Every Triangle
G:       Render Back Frustum
G:     For Every Triangle
G:       Render Front Frustum
```

Oblique Slice and Dice

```
C: For Every Oblique View
C:   For Every Object
G:     For Every Triangle
G:       Render Oblique Frustum

C:   Pixel Swizzle into Hogel Micro Images
```

BowTie

```
M: For Intersecting Objects
M:   For Every Triangle
M:     For Intersecting Hogel Micro-Images
M:       Render BowTie Frustum
```

FIGURE 5. ALGORITHM ORDER OF OPERATION

For instance, "For Every Object" refers to the dispatching of render commands from a CPU, while "For Every Triangle" refers to the processing of a vertex list by a GPU. It should be noted that while "Pixel Swizzling" can occur on a GPU, large radiance images may require hundreds of gigabytes of RAM in which case pixel swizzling might be better served by a CPU. Also "Pixel Swizzling" may be a completely post-render process in which all the oblique views are rendered first, then a massive pixel swizzling or sampling operation ensues. Lastly, the application of 2D calibration coefficients that account for LfD manufacturing and assembly tolerances can only be executed against fully rendered/assembled hogel micro-images and are unique to each display. Therefore, 2D calibration corrections are best applied in the LfD's micro-image display drivers and not when the hogels are initially rendered.

Object Culling

Object culling is traditionally executed by the host application against the application's camera frustum to alleviate issuing unnecessary graphics commands for non-visible geometry. As the host application camera is not the same as those defined within the HPD, the host application cannot cull objects against its own camera. In addition, the host application cannot cull objects against the VVT either as the projection frustums derived from the VVT can also extend outside the VVT's cuboid definition as is the case with *Double* or *BowTie* camera frustums of the HPD.

As the hogel camera frustums are defined within the HPD, the first real processing stage of a radiance image rendering pipeline is to determine which hogel *BowTie* frustums intersect each object. Again, traditionally this is done by comparing an object's bounding volume for intersection with a camera frustum. However, when the HPD may contain millions of camera frustums, this can be an expensive task. It is more efficient to transform the object's maximum and minimum bounding volume extents into the HPD space and then reverse cast the *Bowtie* frustum edges from the transformed extents onto the normalized hogel image plane (Figure 6). The resulting intersections encompass the subset of hogel frustums that intersect the object's bounding volume. Limiting the processing of objects within that narrower subset of hogels speeds up rendering significantly.

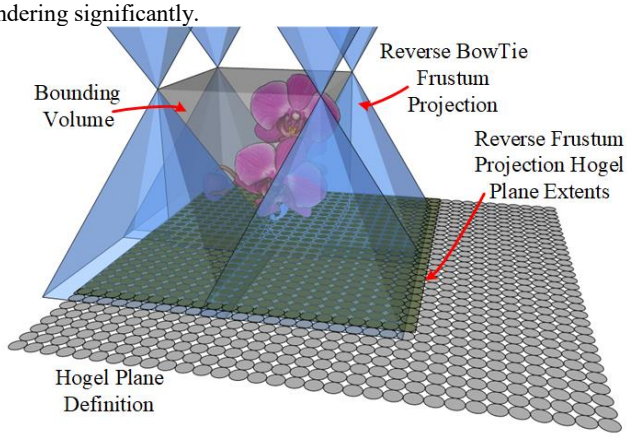

FIGURE 6. BOUNDING VOLUME/HPD INTERSECTION TEST USING REVERSE FRUSTUM PROJECTION

BowTie Clipping

The subset of hogels that intersect a particular triangle can also be calculated in a similar manner. Here, the vertices of the triangle are transformed into HPD space and the frustum edges cast from the vertices onto the hogel image plane, isolating the subset of hogels whose frustums intersect that triangle. This is shown in Figure 7.

Model Space Triangle Clipping

The traditional rendering pipeline expect a triangle's vertices to be multiplied by the model-view-projection (MVP) matrix before being submitted to the rasterizer for clipping/culling in unity clip space. However, this implies at least 3 [4x4] matrix by [4x1] vertex multiples (~48 multiplies) per triangle per hogel within the identified hogel plane sub-region. Note: Using the *Double Frustum* algorithm is twice as bad; the front and back frustums are defined separately and must be evaluated separately, which can be a significant number of multiplications (and additions) just to see if a triangle is visible to an individual hogel frustum. Hogel frustum definitions can be narrow, resulting in many culled or clipped triangles. Therefore, the *BowTie* renderer clips in model space to avoid many unnecessary triangle vertex transforms.

As noted in the excellent *Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix* paper [6], deriving clipping planes from the MVP matrix allows for clipping in model space. This implies, though, that on a per-object basis, the intersecting HPD subregion of hogels would need to be transformed into the object's model space to determine the necessary clipping planes.

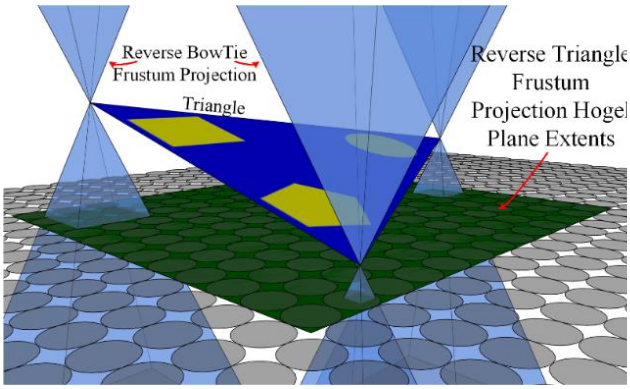

FIGURE 7. TRIANGLE/HPD INTERSECTION TEST USING REVERSE FRUSTUM CAST

Shift Clipping

However, as all the hogels are on a plane and have the same orientation, only one hogel needs to be transformed into the object's model space; the remaining hogel frustum planes can be calculated merely by shifting one set of transformed hogel frustum planes with a few scaled additions. Therefore, as part of the HPD, one set of hogel frustum planes can be defined at the origin and transformed into model space when a new object enters the pipeline. Subsequent hogel-specific frustum planes are then derived through inexpensive addition operations and cached when a hogel frustum requires an intersection test with the first triangle of an object. In this manner, hogel frustum planes are efficiently calculated once per object render and only when necessary.

Smart Clipping

Clipping algorithms such as *Sutherland-Hodgman Clipping* [7] use the "distance point to plane" dot product calculation to determine whether a point is behind, on or in front of a plane. Therefore, during the *Sutherland-Hodgman* edge clip operation the cardinal direction of where the points lie relative to the frustum can be recorded and used to prevent future hogel frustum intersection tests.

Figure 8 highlights this concept where a triangle is far to the left of the hogel plane center. If the center-most hogel was tested against this triangle, the result would be a fully culled triangle and a flag/bit mask indicating that all the triangle vertices were to the left (or west). Therefore, no hogel frustum in the same up/down (North/South) column or any frustum to the right (east) would require testing.

Or, instead of indexing through the HPD using indices calculated by the reverse frustum cast, the clip direction could be used to binary search through the HPD to find a valid triangle intersection.

The order of frustum plane testing might have an impact on performance and some examples of *Sutherland-Hodgman* show clipping to the left, then right, then up, down, and so on. It is more beneficial to clip left, up, right, and then down as to more quickly derive the appropriate direction to search next.

Lastly, if the hogel frustum plane equations are normalized, then the direction vector magnitude is the distance to that vertex from the frustum center. This information can be used to directly index the next hogel in the HPD, negating a search entirely.

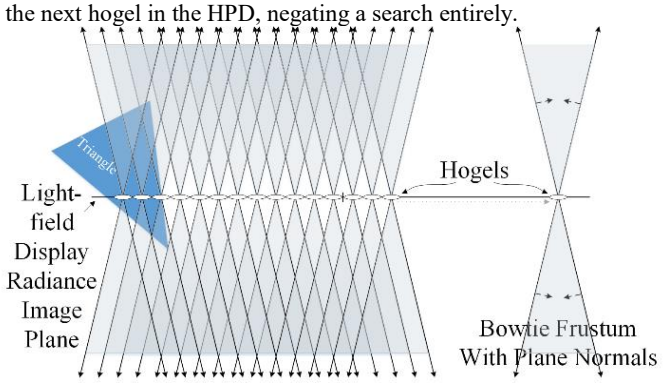

FIGURE 8. BOWTIE SMART CLIPPING

The Multi-view Processing Unit (MvPU)

Unlike a traditional GPU which is tightly bound to a host CPU and application, the conceptual MvPU is intended to reside within the 3D LfD hardware, relieving the host application of render responsibility for unique display architectures. The MvPU renders all the views in parallel from a single dispatch of the geometry from the host application and writes directly to the display back-buffers within the projection subsystem of the LfD. This allows the host application to be implemented without regard of the LfD hardware or the LfDs unique projection requirements.

MvPU Simulator

FoVI^{3D}'s *Multi-view Processing Unit* (MvPU) simulator is a C/C++ raster pipeline simulator developed to research means to render light-field radiance images effectively. C/C++ was chosen to implement the research pipeline merely to ensure maximum flexibility over the render pipeline architecture and for the ease of debugging. In addition, since the *BowTie* algorithm cannot be implemented in any traditional graphics API (OpenGL, DirectX, etc), the only fair comparison in algorithm performance was to implement algorithms such as the *Double Frustum* and *BowTie* in a neutral language such as C/C++ where the variations of the algorithms could share as many stages, routines and classes as possible. As such, any resulting gains are strictly the result of optimizations inherit to the algorithm. It should also be noted that since the previously described *BowTie* radiance image rendering is more concerned with setting up view/frustum/triangle relationship and the effect of shared optimizations across the HPD, more modern shader stages such as the geometry or tessellation shaders were not considered for this rendering evaluation.

The first stage of the MvPU raster pipeline consists of a scene/object dispatcher which iterates over objects within the scene and culls objects against the HPD using the reverse *BowTie* frustum projection test. Therefore, the inputs to the MvPU pipeline are the scene description, a VVT and a HPD. For every object that intersects the HPD, a subset of HPD hogel indices is forwarded to the second pipeline stage.

The second pipeline stage is an object triangle dispatcher where triangle vertices are culled by the reverse *BowTie* frustum projection/HPD test. The resulting list of HDP indices are the subset of hogel frustums that intersect the triangle. The next stage is to clip the triangle against the subset of hogel frustums. Surviving

triangles/polygons are then passed to the final stage, the fragment shader.

Algorithms and Optimizations Implemented within the MvPU C/C++ Simulator

The two primary algorithms implemented and tested were the *Double Frustum* and *BowTie* radiance image renderers. Three flavors of the *BowTie* renderer were implemented to test the previously described culling/clipping optimizations. The four tested algorithm implementations are described below:

Double Frustum

Within the MvPU simulator, the *Double Frustum* algorithm was implemented as it would be implemented within an OpenGL render pipeline and is therefore a view-major process where the front frustum and the back frustums are processed separately into the same viewport. There are no common camera/view optimizations implemented as each hogel frustum must be tested individually without shared knowledge from sibling hogels. Therefore, all scene triangles are tested against all *Double Frustum* hogels.

Note: One huge and notable *Double Frustum* implementation exception is that as a completely C/C++ simulator, the MvPU simulator does not simulate the cost of dispatching rendering commands from the host CPU to a GPU on a per hogel basis. Ideally, this would not be ignored as cost of dispatching rendering commands per view is not insignificant.

Basic BowTie

The *Basic BowTie* algorithm was implemented as a triangle-major process and clipping is done in model space. One *BowTie* frustum is defined per hogel, which reduces the number of triangles tests required by the *Double Frustum* algorithm. Each triangle is first tested against the hogel image plane to determine which sides of the bowtie need to be evaluated. This information is used to preselect which halves of the *BowTie* require processing for the entire HPD. No additional optimizations were implemented to directly compare the order/magnitude of operations with the *Double Frustum* algorithm; therefore, all scene triangles are tested against all *BowTie* hogel frustums.

Binary Search/Fill BowTie

The *Binary Search/Fill BowTie* builds upon the *Basic BowTie* algorithm by using a binary search to quickly find a first triangle intersecting frustum. The search starts with the center most hogel in the HPD. When an intersection is discovered, the binary search is halted and neighboring hogels are scheduled on a stack for future processing; this is repeated until there are no more hogels to schedule and the triangle has been processed against all intersecting *BowTie* hogel frustums.

Intersection Map BowTie

The *Intersection Map BowTie* calculates the HPD sub-region for triangle processing by reverse frustum projection from the triangle vertices onto the hogel image plane in HPD space.

Test Models and Radiance Image Rendering Testing

All four algorithms for this evaluation had *Object Culling* disabled and all use the exact same triangle rasterizer and fragment shader. Therefore, this review is exclusively about the order and cost of the triangle vertex processing. The models shown in Figure 9 and described below were used for this radiance image rendering evaluation:

- ***Utah Teapot – 15,704 triangles***
  The *Utah Teapot* is a standard polygonal model used for rendering and graphics evaluations. It is a low-complexity (low triangle count) model whose vertices were generated through parametric equations and is well structured.
- ***Gears – 101,586 triangles***
  Gears is a moderate-complexity model (6x triangles of *Utah Teapot*) and is representive of a model created by a 3D graphics artist within a 3D modeling package.
- ***DyingGaul – 379,526 triangles***
  DyingGaul is a high complexity model (>24x triangles of Utah Teapot) polygonized from a high-resolution scan model.

Four 30x30 hogel HPDs were constructed using 30°, 45°, 60° and 90° FoV frustums to highlight the effect of FoV on algorithm performance. The hogel micro-image resolution was 60x60 pixels on a 64x64 pixel center-to-center pitch, resulting in 1920x1920 pixel radiance images. Bear in mind, this is a very small radiance image and only constructed for this evaluation.

Testing and Test Results

The MvPU simulator and radiance image rendering evaluations were conducted on a single thread of an Intel i7-4790 4.00 GHz CPU to directly compare algorithm performance. The radiance images generated by the *Double Frustum* algorithm were considered the reference images, and subsequent rendering tests were compared to these images for validation by taking the absolute difference between the rendered images.

Upfront, there are two analyses to consider in this evaluation. The first is the performance of the *Double Frustum* versus *Basic BowTie* radiance image renderers. The second is the effect of globally culling and clipping triangles within the *BowTie* renderer either by searching for valid hogel frustum intersections or by calculating a triangle's bounding frustum intersection region.

Double Frustum Vs. BowTie Dispatch & Vertex Processing Time Summary

Figure 9 summarizes the total time spent within the dispatch and vertex processing stages of the *Double Frustum* and *Basic BowTie* radiance image rasterizers for all the tests and models, and highlights the benefits of rasterizing a single, 4-plane bowtie frustum with triangle major rendering over two independent 6-plane defined frustums with view major rendering. The largest *BowTie* rendering gain was noticed within the 30° Gears test which was 8.2x faster than the *Double Frustum* renderer. The smallest *BowTie* rendering gain was 4.7x within the 90° Dying Gaul test.

Basic BowTie Vs. BowTie Binary Search/Fill Vs. BowTie Intersection Map Triangle/Frustum Culling and Clipping Summary

Figure 9 shows the performance of the various *BowTie* triangle/frustum culling and clipper processors that were evaluated. As expected, the search/fill and intersection maps enabled the *BowTie* algorithm to accelerate triangle/frustum culling and clipping anywhere from 2x to 28x faster than the *Basic BowTie* renderer. In all the tests, the intersection map implementation outperformed the binary search/fill implementation. The amount of performance gain was relative to the FoV, with the largest gains occurring with narrow FoV where more triangles are discarded upfront.

Conclusion

The traditional render pipeline is intended to render from a single viewpoint onto a single, large viewport. GPUs and the accompanying APIs are well suited for this purpose. Light-field radiance image rendering requires rendering from many viewpoints onto small viewports, shifting the rendering processing burden from the back-end rasterizer and fragment shader to the front-end vertex processor. In addition, radiance image rendering burdens the host

application with managing all the viewpoints/viewports specific to an LfD architecture, binding the host application to that LfD architecture.

In the *Heterogeneous Display Environment* (HDE), the display is responsible for rendering the views it requires for its unique projection requirements. While it can be argued that an array of PCs can be allotted for this purpose, the *Size, Weight, Power and Cost* (SWaP-C) of a PC cluster is extremely high for a specific and singular rendering purpose.

As demonstrated in this evaluation, a custom multi-view render pipeline for an LfD can expect significant performance gains by reordering triangle dispatch and exploiting triangle optimizations common to a 2D array of *Bowtie* frustums. In addition, by moving radiance image rendering into the LfD, the enormous number of pixels required for a deep 3D visual experience can be written directly to the LfD micro-display back buffers, forgoing all the cabling, protocols, space, power, and complexity of an external cluster.

Next Steps

The C/C++ MvPU simulator has been a useful tool for the research and development of the *BowTie* algorithm for rendering radiance images more efficiently. The next phase of research is to implement the *BowTie* renderer for a GPU in CUDA and exploit the extreme parallelism capabilities of a modern GPU.

References

[1] Thomas Burnett. 2017. Light-field Display Architecture and the Challenge of Synthetic Radiance Image Rendering. SID Symposium Digest of Technical Papers.

[2] Thomas Burnett. 2017. Light-field Displays and Extreme Multiview Rendering. Information Display.

[3] Michael W. Halle, Adam B. Kropp. 1997. Fast Computer Graphics Rendering for Full Parallax Spatial Displays. Proc. SPIE. Vol. 3011.

[4] Shaohui Jiao, Xiaoguang Wang, Mingcai Zhou, Weiming Li, Tao Hong, Dongkyung Nam, Jin-Ho Lee, Enhua Wu, Haitao Wang, and Ji-Yeun Kim, "Multiple ray cluster rendering for interactive integral imaging system," Opt. Express 21, 10070-10086 (2013)

[5] Yanxin Guan, Xinzhu Sang, Shujun Xing, Yingying Chen, Yuanhang Li, Duo Chen, Xunbo Yu, and Binbin Yan. 2020. Parallel multi-view polygon rasterization for 3D light field display. Optics Express 28, no. 23.

[6] Gil Gribb, Klaus Hartmann. 2001. Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix.

[7] Randy Gaul. 2013. Understanding Sutherland-Hodgman Clipping for Physics Engines. Retrieved from: https://gamedevelopment.tutsplus.com/tutorials/understanding-sutherland-hodgman-clipping-for-physics-engines--gamedev-11917.

Author Biography

Thomas Burnett has been developing static and dynamic light-field displays since 2003. In 2015, Thomas co-founded FoVI^{3D} to research and develop light-field display technology and solutions.

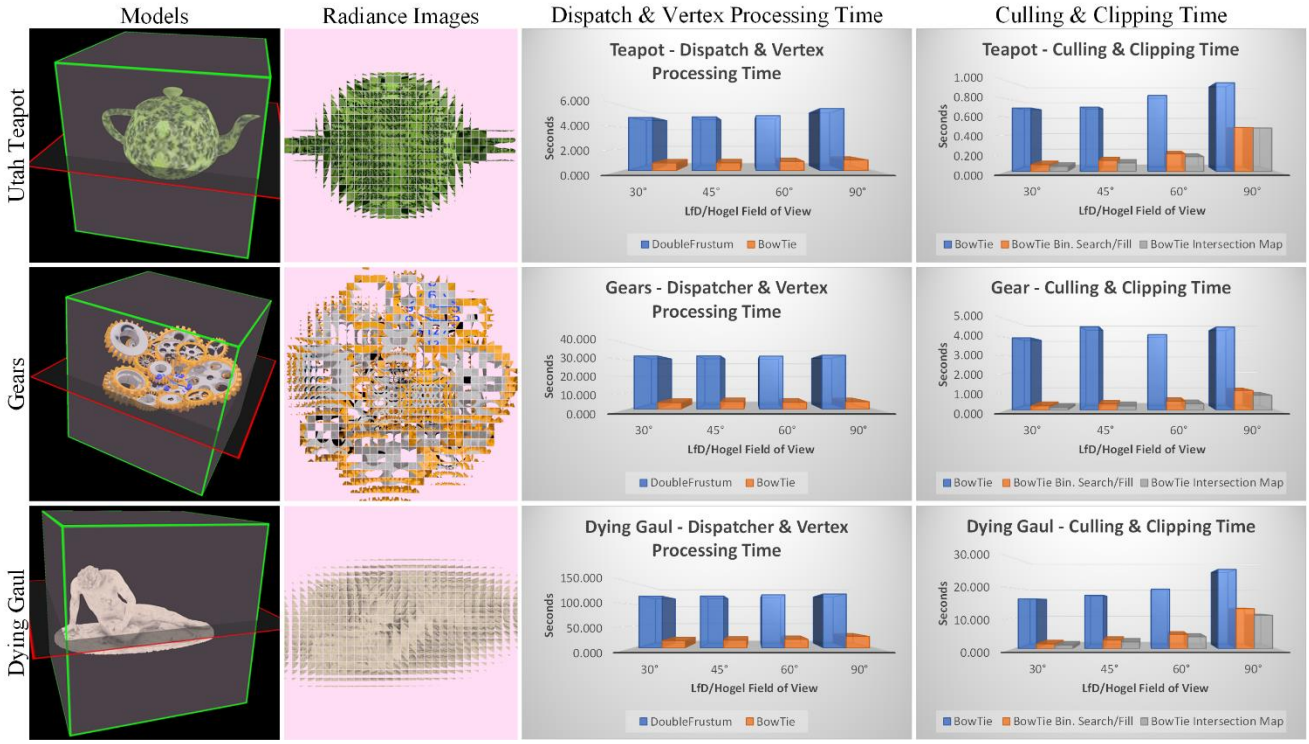

FIGURE 9. TEST MODELS, RADIANCE IMAGES AND PROCESSING RESULTS