University of Derby

School of Computing & Mathematics

A project completed as part of the requirements for the

BSc (Hons) Computer Games Programming

entitled

# Optimisations of the light culling algorithm in a Forward+ Rendering Pipeline

by

Luke Thatcher

luke.thatcher@live.co.uk

2013 – 2014

# 1 ABSTRACT

Use of modern graphics hardware capable of general purpose computation has been the focus of recent developments in the real-time computer graphics community. Various techniques have been proposed which improve upon more traditional rendering systems through the use of general purpose compute. Tile based systems, such as Forward+, utilise general purpose compute to dynamically build linked lists of lights entirely on the GPU, however the effectiveness of the pipeline is heavily dependent on the accuracy and performance of the light/tile intersection tests. This project focusses on the improvement of these intersection tests, providing a more efficient Forward+ rendering pipeline.

# 2 TABLE OF CONTENTS

# 3 TABLE OF FIGURES

# 4 INTRODUCTION

Since the earliest examples of three dimensional video games and computer graphics, there has been a continuous drive towards greater realism and visual fidelity, led by the desire to create larger, more immersive worlds with more compelling game play and narratives. Advances in the field of graphics programming have been realised through the combination of newer, more flexible and powerful graphics hardware, and innovative research focussing on the use of such hardware to solve the problem of displaying geometric meshes with a variety of surface and lighting properties on a raster-based display.

## 4.1 OFFLINE VERSES REAL-TIME GRAPHICS RENDERING

Computer graphics rendering involves the generation of images based on some source data representing a virtual world. It has an incredibly wide range of applications, including scientific visualisation, medical research, computer aided design, education, and entertainment such as movie visual effects and video games. Each of these applications demand certain characteristics from a computer based renderer, chief amongst which is the quality of the final image verses the level of interactivity required.

### 4.1.1 Offline Rendering

Higher quality and higher detail images require a greater amount of computational resources and time to achieve. In the field of movie visual effects, the emphasis is placed heavily on image quality. Since a movie is not interactive with the viewer, visual effects studios use rendering systems that may take several hours, days or weeks to render images, opting to spend a greater amount of computation time to achieve higher image quality and realism. The output video frames generated by the renderer can be stored, edited and post-processed, and are ultimately played back to the viewer sequentially in real-time. These rendering techniques are commonly referred to as “offline”, as they generate the required images in advance of them being presented to the viewer.

### 4.1.2 Real-time Rendering

This technique of offline rendering cannot be applied to video games, which are highly interactive forms of media. Video games must be able to display generated images at a rate fast enough to provide smooth motion, and in doing so, be able to respond to player input with minimal delay. Rendering techniques which take even a few seconds to compute an image are unusable for video games.

Assuming a display refresh rate of 30 Hz, a video game has just 33.33 milliseconds to render and display a single frame. For a 60 Hz game, this time is reduced to 16.66 milliseconds. If rendered images take longer than these times to produce, the player may notice stuttering of motion in the video output, and may experience an input delay, commonly known as “lag”, which adversely affects game play.

For this reason, well designed graphics rendering systems written for video games aim to achieve the best possible image quality, but are constrained to work within the short frame times to maintain interactivity.

## 4.2 Sprite Based Rendering

Due to the limited availability of memory resources and computational power, the earliest graphics hardware and video games consoles were only capable of two dimensional rendering, compositing a series of “sprites” (two dimensional images) onto the screen in layers to build the visual world. Image composition is not a particularly computationally expensive task, which suited the limited nature of the hardware in early games consoles and computers.

Sprite based systems, whilst they may achieve visually appealing results, are very limited in the realism they can provide. Despite this, sprite based games are still common in the video games industry today, most prevalent in the casual and mobile games market.

## 4.3 Three Dimensional Rendering

To add a greater depth of realism and immersion for the player, modern, higher budget console and computer games tend to use three dimensional rendering. Contrary to sprite based rendering, three dimensional rendering does not rely on images as its source data. Instead, artists define meshes composed of triangles in three dimensional space. These meshes are placed inside (and moved around within) a virtual three dimensional world by the game engine. The task of the renderer is then to solve what colour each pixel on the raster display should be set to, given the current positions and properties of the meshes within the virtual world, and the properties and position of a virtual camera.

Meshes within the virtual world may have various surface properties and materials associated with them, which define the way their surface reflects light. The renderer also models various light sources within the virtual world, and uses this information to determine the final colour of a given mesh’s surface.

Almost all three dimensional rendering techniques in use today can be placed within one of two groups: rasterization based techniques, and ray tracing based techniques. They both provide solutions for pixel visibility and colour from input geometry and camera properties, but they do so in very different ways.

### 4.3.1 Ray Tracing

Ray Tracing is a recursive technique for three dimensional rendering which, as the name implies, traces a ray for each pixel in the final image, into the scene from the viewer. The ray is defined mathematically with an origin and a direction vector. To trace a ray through the scene, the renderer performs a series of collision tests between the ray and each geometry object within the scene, keeping track of the closest collision found.

Once a collision has been resolved, the ray tracer can then evaluate the incoming light incident on that point on the object's surface. This is usually achieved by performing further ray tests against the light sources in the scene to determine if the current surface position is within shadow of a given light source. If not, the renderer determines how much light that source will contribute to the surface, using a computational model of the light source and surface material. Advanced light effects such as reflection, refraction, caustics, soft shadows and sub-surface scattering can be achieved by spawning further rays to sample the incoming light on the surface.

The renderer proceeds to recursively cast rays until the current ray exits the world, or a given recursion limit is reached. The result of a ray test is the total amount of light reflected from the surface point, in the direction of the ray's origin. The recursive results of these ray tests are passed back to the parent ray, accumulating light intensity according to surface reflectance properties, until the final ray has returned the result for the pixel. This final value then becomes the amount of light reflected in the direction of the viewer for that pixel, and ultimately determines the colour for that pixel.

Ray tracing renderers are capable of generating exceptionally realistic images, as they are able to emulate the physical properties of light transport, albeit in the reverse direction to how light behaves in the physical world. Complex light behaviour is trivial to achieve, simply by casting additional rays to compute a given physical lighting effect. The realism ray tracing can achieve is highly favoured with the visual effects industry and their use of offline rendering.

As a contrast to its apparent simplicity, ray tracing is very computationally expensive. Millions of ray to geometry object collision tests may be required to compute an entire image. This count only increases with additional geometry and lights, and higher resolutions. To cut down on the number of ray tests required, ray tracers often employ spatial partitioning structures which filter geometry objects based on

their world space position and bounds. This accelerates the ray-geometry tests by quickly identifying only those objects which are candidates for collision, rather than needing to test every ray against every object.

However, even with these acceleration structures, the computational requirements of the ray tracing algorithm exceed the performance constraints demanded by real-time interactive applications on all but the most powerful graphics hardware. Ray tracing is currently considered not feasible for mainstream interactive video game development.

### 4.3.2 Rasterization

Rasterization is an alternative rendering technique which has become the de facto standard in interactive video games rendering, due to its speed, simplicity, scalability and relatively small memory footprint. Modern graphics processing units contain dedicated hardware to compute the rasterization algorithm.

Whilst ray-tracing solves pixels by casting rays into the world, rasterization instead finds the mapping of geometry to screen positions, allowing it to fill pixels directly. As a result, a rasterization pipeline does not need to hold the entire scene in memory. On the simplest level, it is possible to render one triangle at a time.

Vector based source geometry is fed into the rasterization pipeline. The pipeline then performs a series of matrix transformations to map the vertices of each triangle to the render target. Once the pixel coordinates of the three points of a triangle have been obtained, the pixels covered by the triangle can be filled using an algorithm such as scanline conversion. The colour for each pixel is determined by a function of the interpolated values from the three vertices of the triangle. Depending on the pipeline design, various stages of the rasterizer may be programmable, allowing custom effects and techniques to be achieved.

Figure 1 outlines the structure of a simple shader-driven rasterization pipeline. A similar pipeline design is implemented within Microsoft's Direct3D 9 graphics API.

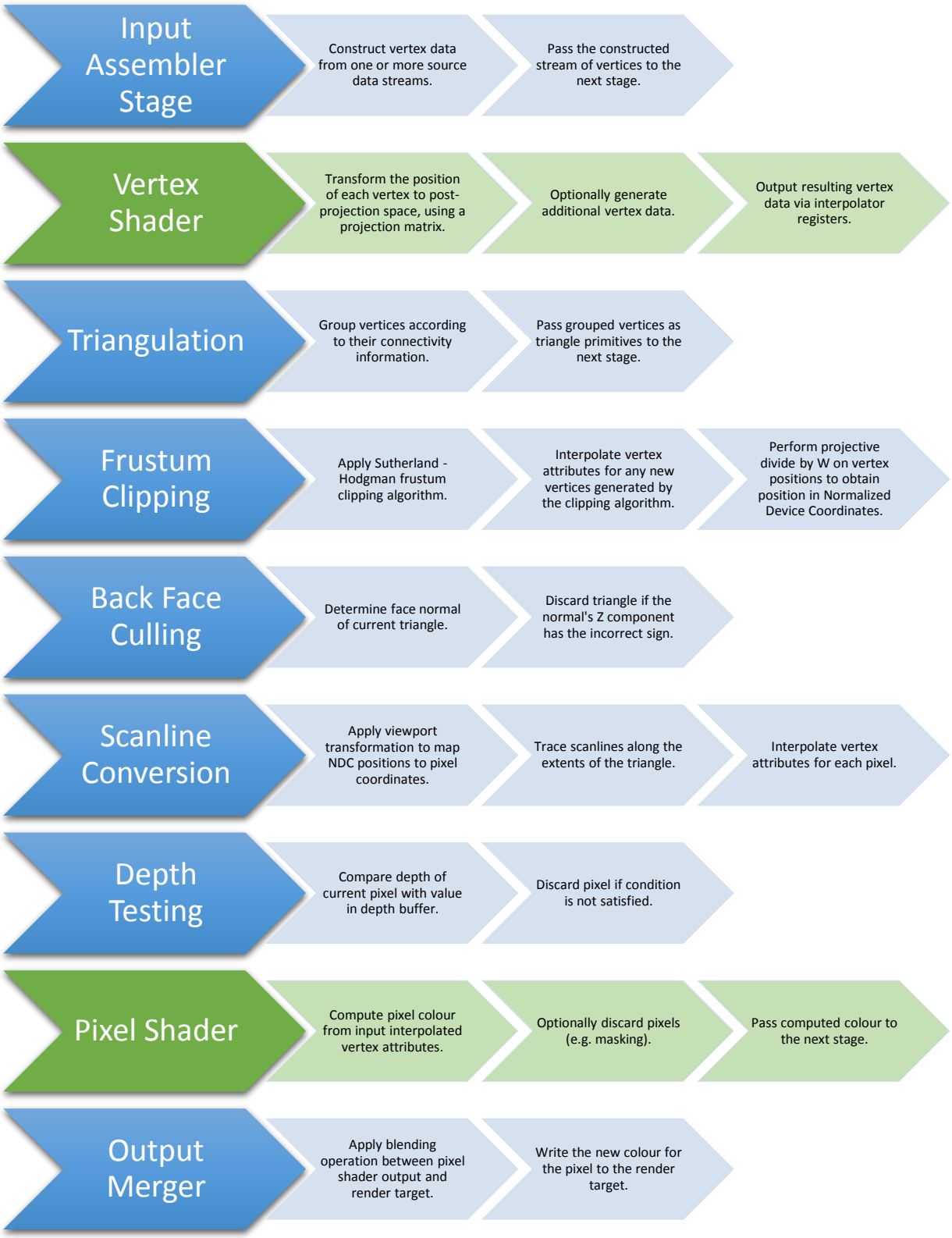


*Figure 1 – Simplified Rasterization Pipeline*

Within modern graphics APIs, the pipeline stages highlighted in green are fully programmable. This provides the pipeline with a great deal of flexibility. Most of the other pipeline stages, whilst their operation is fixed, are often configurable with render state information. For example, the depth testing stage is driven by a user-specified comparison function. Manipulating this function provides control over how pixels are discarded by the depth test. Subsequent versions of Direct3D and OpenGL have introduced additional pipeline stages to handle features such as dynamic geometry creation (Geometry Shaders), and tessellation and surface subdivision (Hull and Domain Shaders).

In contrast to ray-tracing, the speed of rasterization is highly suitable for real-time rendering applications such as video games. An optimised pipeline running on modern graphics hardware can render many hundreds of thousands of triangle primitives within the allotted frame time. Unfortunately whilst rasterization is much faster than ray-tracing, it suffers from the inability to accurately compute physical lighting effects in an intuitive way. Rasterization renders single objects at a time; the entire scene's geometry is not all in memory at once. As such, pixel shader functions cannot directly account for effects such as indirect lighting, shadowing, and reflections, as these effects depend on knowledge of the world outside of the current surface being rendered.

To achieve these effects, various techniques must be applied to capture certain elements of the scene, to provide that data to the pixel shader function for a surface. This often involves using an additional rasterization rendering pass to draw the scene from a different camera's perspective to an off-screen texture held in video memory. This texture collects information about the scene, relevant to some effect, which can then be read back in the pixel shader function to approximate that effect. An example of this is shadow mapping, a commonly used technique to approximate shadows cast by objects in the scene, as outlined in Algorithm 1:

---

```
set render target to shadow map
FOR each mesh in the scene
        transform mesh vertices using light's view-projection matrix
        rasterize mesh, apply depth testing to keep only the closest pixels
        write closest depth value to shadow map
ENDFOR
set render target to the back buffer
FOR each mesh in the scene
        transform mesh vertices using camera's view-projection matrix
        rasterize mesh, apply depth testing to keep only the closest pixels
        LET colour = compute light contribution
        LET distance = compute distance from surface to light
        LET mapValue = read corresponding value from shadow map
        IF mapValue < distance THEN
```

```
            darken colour
        ENDIF
        write colour to render target
ENDFOR
```

*Algorithm 1 – Single Light Shadow Mapping*

The renderer first begins by rendering the scene from the light's point of view. Rather than colour, the renderer stores the depth of each pixel from the light in an off-screen depth buffer. Once the shadow map has been completed, the renderer switches to drawing the world to the screen from the camera's point of view. At each pixel, the corresponding location in the shadow map is determined using the light's view-projection matrix. The value from the shadow map is read back and compared to the computed distance between the surface and the light for the current pixel.

If the values are approximately equal, then the current pixel is considered to be in direct line of sight with the light, so is not in shadow. Otherwise, the value stored in the shadow map will be less than the value for the current pixel. This implies there was some additional geometry between the light and the surface, and so the current pixel is in shadow; the colour of the pixel is dimmed to approximate this.

Rasterization based pipelines heavily rely on such techniques to achieve advanced lighting effects, which ray-tracing can achieve with relatively little effort. The flexibility of rasterization pipelines has driven research to find better techniques for approximating advanced lighting effects, aiming to improve the realism and detail within rasterized images. Such research has led to the development of a number of renderer designs, each aiming to improve image quality and complexity, whilst maintaining real-time performance.

## 4.4 Forward Rendering

The simplest rasterizing renderer design maps very closely to the rasterization pipeline described in Figure 1. This design employs a technique commonly known as "Forward Rendering" to render the effect of each light on the scene's meshes, additively accumulating the contribution of each light.

```
FOR each light in the scene
    FOR each mesh in the scene within the light's bounds
        transform mesh vertex data
        compute lighting from current light at each pixel
        sum the light result into back buffer using additive blending
    ENDFOR
ENDFOR
```

*Algorithm 2 – Multiple Pass Forward Rendering Pipeline*

The term “Forward Rendering” originates from the idea that all geometry and lighting data flows in one direction through the pipeline. Forward rendering suffers from an **O(L*M)** complexity, and therefore does not scale well when additional lights are added to the scene. The pipeline is also rather inefficient, as during each rendering pass the vertex data must be processed. When more than one light is in the scene, this creates repeated work, wasting time that could be utilised on something more useful in rendering the final image.

Early graphics hardware was particularly inflexible, and typically employed fixed function rendering pipelines. Historically, both the OpenGL and Direct3D graphics APIs exposed such pipelines which relied heavily on the Forward Rendering technique.

The introduction of programmable shader pipelines allowed graphics programmers to implement optimised versions of the standard forward renderer. One such optimisation involves batching lights together into single draw operations, avoiding the repetition of vertex calculations at least between the lights in the current batch.

```
create light batches from lights in the scene
FOR each light batch
        FOR each mesh in the scene within the batch’s bounds
                transform mesh vertex data
                LET sumLights = 0
                FOR each light in current batch
                        sumLights += computed lighting from current light
                ENDFOR
                write sumLights to back buffer using additive blending
        ENDFOR
ENDFOR
```

*Algorithm 3 – Single Pass Forward Rendering Pipeline*

Whilst this optimisation avoids the reprocessing of vertex data between lights in a given batch, there is still a requirement to reprocess vertex data between batches of lights. Furthermore, the shaders required to implement the algorithm for each light type, batch size, and material can lead to a combinatorial explosion, creating a graphics pipeline dependent on an excessive number of compiled shader programs. This results in a great deal of overhead, which adversely affects the pipeline’s performance and ease of use during development.

## 4.5 Focus and Objectives

For many years, various techniques have been developed to provide a solution to the light-mesh coupling, complexity and combinatorial explosion issues found in forward renderers. Whilst these techniques provide much more efficient pipelines, they are far from perfect and suffer from various quality issues and limitations.

Recent advances in graphics hardware have introduced the ability to perform general purpose computing in a highly parallel manner, outside of the standard rasterization pipeline. This has allowed the implementation of more optimal forward rendering pipelines, involving tile based culling of lights running entirely on graphics hardware, such as “Forward+” presented by (Harada, et al., 2012), which shall be the focus of this project.

An investigation will be made into possible improvements to the pipeline discussed by (Harada, et al., 2012), with an emphasis on the efficiency of the light culling algorithm. This will be implemented within a graphics framework that takes advantage of the most recent platform features, focussing on the flexibility of shader management.

After an analysis of various academic sources and industry discussion on graphics rendering techniques, the following objectives have been outlined as the targeted field of research for this project.

- Implement a Forward+ based rendering pipeline.
    - Target native Direct3D 11.1 running on the Windows 8.1 platform.

- Investigate the performance of, and implement performance improvements to, the light culling phase of Forward+ rendering.

- Focus on the flexibility of the rendering pipeline, implementing the following features:
    - Dynamic shader recompilation.
    - Support for materials.
    - Asset management.

# 5 LITERATURE REVIEW

## 5.1 ALTERNATIVE RENDERING PIPELINES

The design of a graphics pipeline which avoids the coupling of mesh data and lights has been the focus of research for many years. These techniques usually involve a "deferred" phase, opting to store some amount of intermediate data in viewport sized textures held in video memory, rather than passing data through the entire pipeline at once.

### 5.1.1 Deferred Rendering

A common alternative to traditional Forward renderers is "Deferred Rendering", a technique that has been employed in a number of video game titles across a range of programmable graphics hardware (Valient, 2007) (Klint, 2008) (Shishkovtsov, 2005).

Rather than computing the surface properties and applying the influence of each light in a single draw operation, deferred rendering utilises a series of intermediate render targets, named the "G-Buffer", to store surface properties. Lighting is then applied as a second, independent pass, reading back the data stored in the G-Buffer to compute the effect of lighting on visible surfaces. In doing so, Deferred Rendering achieves an **O(L+M)** complexity, since geometry data is processed once, rather than once per light. Applying lighting to the scene is then a process that operates entirely on the data stored in the G-Buffer.

To increase performance, it is desirable to compute lighting functions only on the pixels within the sphere of influence of a given light. Various techniques exist, such as Carmack's Reverse (Carmack & Kilgard, 2009), also known as Z-Fail stencilling, which utilise the hardware's stencil buffer to quickly filter the candidate pixels affected by a given light.

```
FOR each mesh in the scene
        transform mesh vertex data
        write interpolated surface properties to G-Buffer textures
ENDFOR
FOR each light in the scene
        apply stencilling to mask pixels within the current light's bounds
        perform lighting calculations at masked pixel positions, using G-Buffer surface data
        write computed lighting to back buffer, using additive blending
ENDFOR
```

*Algorithm 4 – Deferred Rendering Pipeline*

The use of stencilling within Deferred Rendering is an effective means of culling pixels from lighting calculations, ensuring that no more work than is strictly necessary is done. The result is a rendering solution which scales very well with the number of lights or detail of meshes used.

Whilst Deferred Rendering is an attractive solution to many modern video games and interactive graphics applications, there are a number of inherent limitations.

Firstly, the amount of space available in the G-Buffer is very limited. On modern hardware, the G-Buffer is implemented as a series of two dimensional textures, which are written to during the initial geometry pass using “Multiple Render Targets” (MRT). A restriction when using MRT is that all textures bound as output for the current pass must have the same pixel width in bits, for example they must all be 32-bits per pixel.

The result is that careful packing of data within the G-Buffer is necessary, often sacrificing floating point precision in favour of smaller width fields. The layout of the G-Buffer is also particularly inflexible, which makes the use of custom material BRDFs (Bi-Directional Reflectance Distribution Functions) very difficult. Unreal Engine 4 (Epic Games, 2014) attempts to solve this by encoding a two bit value representing the “material type”. Various fields within the G-Buffer are redefined based on the value stored as the material type. This allows a choice of up to four unique material BRDF models to be used during lighting computation, however the input to these BRDFs must still be arranged to fit within the G-Buffer. There is also an additional ALU cost involved in the encoding and decoding of any data packed in the G-Buffer textures.

Of course if there are no more fields available within the G-Buffer to store additional data, an entire additional texture may be used to increase the available space. However, this presents the second limitation of Deferred Rendering. There is a large bandwidth cost that arises from the read/write operations on the G-Buffer, a cost which is further exacerbated with the use of additional render targets to support complex material BRDFs. Accessing texture memory is inherently slower than the local memory used by a shader program, so often Deferred Rendering is limited by the available memory bandwidth on the graphics hardware.

Deferred rendering also suffers from several other limitations. Since G-Buffer data is stored per-pixel, there is no support for hardware Multi-Sample Anti-Aliasing (MSAA) unless G-Buffer data is stored per-sample, and the lighting for each sample is computed manually during the lighting phase. To achieve anti-aliasing within a deferred pipeline requires the use of full screen post processing passes after the scene image has been rendered to blur the edges of polygons. Various techniques exist, such as Fast

Approximate Anti-Aliasing (FXAA), and Temporal Anti-Aliasing (TAA), which exhibit various quality issues often in the form of blurred detail in textures.

Deferred rendering also has no way of handling transparent objects correctly. The G-Buffer is only capable of storing surface information of the nearest opaque pixel to the camera. To achieve transparency effects, a separate forward based pipeline must exist which is applied after the opaque deferred pass. This requires the developer to maintain two unique rendering systems, increasing development effort.

### 5.1.2 Light Pre-Pass Rendering

An alternative technique to full Deferred Rendering is Light Pre-Pass Rendering (Lee, 2009). This technique exists almost as a hybrid approach between full deferred and more traditional forward renderers. It aims to maintain the decoupling of lights and geometry, but reduce the size of the G-Buffer to help alleviate the bandwidth and memory usage issues associated with the use of multiple textures.

```
FOR each mesh in the scene
        transform mesh vertex data
        output depth and surface normal to intermediate textures
ENDFOR
FOR each light in the scene
        apply stencilling to mask pixels within the current light's bounds
        perform lighting calculations at masked pixel positions
        write diffuse lighting result to diffuse buffer using additive blending
        write specular lighting result to specular buffer using additive blending
ENDFOR
FOR each mesh in the scene
        transform mesh vertex data
        modulate lighting from diffuse/specular buffers with surface material properties
        write computed lighting to back buffer
ENDFOR
```

*Algorithm 5 – Light Pre-Pass Rendering*

While light pre-pass rendering addresses several issues related to a full deferred pipeline, a number of limitations still exist.

The use of a diffuse and specular lighting buffer common to all light types and materials eliminates the possibility of using specialised material BRDFs. Since the lighting properties of each light are computed in a common light pre-pass shader, the material of each mesh may only modulate the result, rather than provide an entirely different lighting model where required, for example in the rendering of hair (which exhibits highly anisotropic properties), or foliage (which exhibits subsurface scattering effects and light transmission).

Similarly to deferred rendering, light pre-pass rendering also does not provide a solution for correctly handling translucent materials. A secondary, entirely forward based pipeline is still required to render such materials.

## 5.2 FORWARD+ (FORWARD "PLUS") RENDERING

In recent years, graphics hardware has become vastly more flexible with the introduction of vendor specific General Purpose Graphics Processing Unit (GPGPU) support, such as NVIDIA's CUDA, and AMD's Stream, and the subsequent extension of cross-vendor graphics APIs to support GPGPU operations using Compute Shaders (under Direct3D 10/11), and OpenCL (the GPGPU equivalent to OpenGL).

Modern graphics hardware is now capable of performing massively parallel general computation, outside of the traditional raster pipeline. This has allowed new rendering techniques to be implemented which were not possible on older generation hardware.

One such technique is Forward+ Rendering, which aims to improve the traditional forward pipeline by solving the coupling of lights and mesh data using compute shaders, providing a fully forward based alternative to the wider used deferred and semi-deferred pipelines.

First proposed by AMD, Forward+ utilises a compute shader to generate a linked list of light indices for each tile in a uniform grid placed across the viewport. The properties of each light are stored in a buffer in video memory. During the final rendering pass, the pixel shader traverses the linked list for the tile the current pixel belongs to, and sums the influence of each light.

---

```
FOR each mesh in scene
        transform mesh vertex data
        output depth to depth buffer
ENDFOR
FOR each tile in viewport
        find minimum and maximum depth values within the tile
        compute frustum of the tile using depth values
        FOR each light in scene
                IF light's bounding sphere intersects tile frustum THEN
                        atomically insert light index into linked list for tile
                ENDIF
        ENDFOR
ENDFOR
FOR each mesh in scene
        transform mesh vertex data
        LET sumLights = 0
        FOR each light in linked list for pixel's current tile
                sumLights += computed lighting from current light
```

**ENDFOR**
**output sumLights to back buffer**
**ENDFOR**

*Algorithm 6 – Forward+ Rendering Pipeline*

With this per-tile linked list, the contribution of every light affecting a given mesh can be computed in a single pass, eliminating the repeated vertex data processing found in traditional single and multi-pass forward rendering. Forward+ is also much more flexible in terms of material BRDF use when compared to Deferred Rendering or Light Pre-Pass Rendering, as any shader can be used in the final rendering pass for a given mesh, allowing different BRDFs to be selected where appropriate.

Forward+ also supports native hardware MSAA, since meshes are rendered directly to the back buffer rather than an intermediate texture, allowing the pixel shader to operate on sample or pixel granularity automatically. Forward+ is also capable of handling translucent materials. An additional light culling pass may be used to compute tile linked lists against any translucent geometry, reusing the same pipeline with only minor blend state changes. This provides an elegant solution for handling correctly lit translucent geometry.

Whilst Forward+ solves many of the issues with traditional forward renderers, there are still a few pitfalls when compared to deferred solutions. All lighting data, including any shadow maps etc., must be resident in graphics memory and accessible by the final pass shaders, whereas under a deferred pipeline, only the data associated with the current light being applied need be resident in graphics memory.

### 5.2.1 Inefficiencies in Tile Culling

Key to the performance of Forward+ is the tile culling compute shader stage. Since each light added to a tile's linked list directly affects the performance of the final rendering pass, it is beneficial to cull as many lights as possible from these lists. In their paper, (Harada, et al.) do not specifically outline how they achieve this culling beyond simply stating that a sphere-frustum test is performed between each light and tile. However, example source code provided by (Advanced Micro Devices, Inc, 2013) unveils they perform a series of sphere-plane tests against the six planes of the frustum.

These tests are very efficient to compute on graphics hardware, as once the plane equations have been computed, the sphere-plane test can be reduced to a single dot product and comparison for the outer four planes. However the simple use of planes does not lead to the accurate intersection test between spheres and frusta. There exists a number of cases where a sphere which does not directly intersect a frustum, does at least partially lie on the positive half-space of all six planes, resulting in a false positive intersection result.

Figure 2 details this issue in two dimensions. False positives may arise when the circle's centre is in the Voronoi region of one of the trapezoid's vertices. When extended to three dimensions, false positives may be found when the sphere's centre is within the Voronoi region of any of the frustum's vertices or edges. These false positives are accumulated in the tile linked lists creating additional work for the final rendering pass.

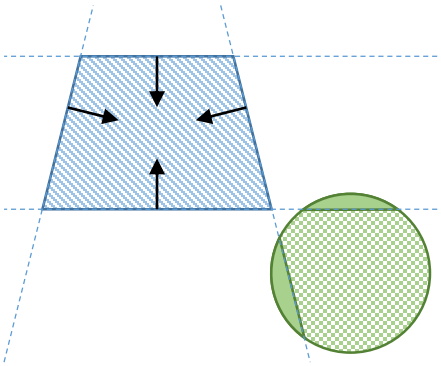

*Figure 2 – Inefficiencies in plane-sphere based tile frustum tests*

*The green circle does not directly intersect the blue trapezoid, but does lie at least partially on the positive half-space of every plane.*

Whilst many CPU based frustum culling implementations used for visibility testing in game engines rely on these sphere-plane tests and provide acceptable results, these are mostly cases with a single very large frustum (the frustum of the camera's projection) and comparatively smaller object bounding spheres.

The likelihood of a false positive result from these tests is exacerbated when the size of the frusta decrease in relation to the size of the bounding spheres, as is the case with Forward+ light culling. The tile frusta are generally very narrow, thin, and have potentially a long extension in depth. This results in very inefficient culling between tiles and light bounding spheres.

Generally, the tile culling phase of the Forward+ pipeline is much less costly than the final rendering pass where the lighting, shadowing and material properties are apply by traversing the linked lists. It is therefore desirable to reduce the amount of work required in the final rendering pass, even if that means increasing the total compute time for the light culling phase. The cost of a slightly more expensive, but more accurate culling algorithm is likely to be vastly outweighed by the performance saving found during the final scene rendering pass.

# 6 METHODOLOGY

## 6.1 FORWARD+ IMPLEMENTATION

The primary focus of this project is the efficient implementation of a Forward+ rendering pipeline. The development of the pipeline relied heavily on the underlying engine design, discussed in section 6.2.

The Forward+ pipeline has two main sections. The tile culling of lights is handled by the *ForwardPlusTileCulling.fx* compute shader, and the final rendering of the scene where the lighting is calculated is handled by the *ForwardPlusMainMaterial.fx* material shader.

The pipeline supports directional, point and spot light sources, although only point lights were used for testing and gathering of performance results. Directional lights do not participate in tile culling, as they apply to every pixel, and so are added to the linked lists for every tile. Spot lights use the same culling technique as point lights, the only difference being the addition of a spot light attenuation factor in the material shader.

Various settings have been hardcoded through the use of pre-processor macros, such as the total number of lights supported, and the tile width and height in pixels. To support a given number of lights, the pipeline assumes worst case memory usage and allocates linked list buffer textures to hold one entry per light, per tile. The actual memory usage for a real-world game would be considerably less than this, but for the purposes of performance analysis, it is simpler to allocate for the worst case. In a production game, the size of the linked list buffers should be tailored to fit the game's use of the rendering pipeline.

### 6.1.1 Culling Algorithm Investigation

A primary goal of this project was the investigation of more optimal culling algorithms for use in the *ForwardPlusTileCulling.fx* compute shader. This led research into various techniques for computing sphere-frustum intersections, and evaluation of their suitability for the compute shader.

The selected algorithm must maximise the number of light-tile interactions culled, whilst minimising the required work done within the compute shader. Careful balancing is required to ensure that the extra compute time spent in the culling phase is less than the time that extra culling work saves during the scene rendering phase. As such, an algorithm which allows a trade-off between culling accuracy and compute time is favourable.

#### 6.1.1.1 *Gilbert–Johnson–Keerthi (GJK) Distance Algorithm*

One possibility of an improved sphere-frustum intersection test is the GJK algorithm (Gilbert, et al., 1988). This algorithm solves intersection tests between two arbitrary convex hulls in three dimensional space by computing the Minkowski difference of the two hulls.

The algorithm operates iteratively until either the origin is found to be enclosed by the Minkowski difference, or the next iteration moves further away from the origin, in which case there is found to be no intersection.

Whilst the GJK algorithm will provide an exact solution for the intersection between tile frusta and light bounding spheres, it is not an appropriate choice for the light culling compute shader. The algorithm must complete fully before a result can be returned. There is no scope to balance the work done in the culling algorithm and the accuracy of the results.

#### 6.1.1.2 *Separating Axis Theorem*

The Separating Axis Theorem (SAT) states that two convex hulls do not intersect if there exists a hyperplane which separates them. To test this, both shapes are projected onto an arbitrary axis. The resulting projections are tested for overlap. If no overlap exists, then the two objects can be separated by a hyperplane perpendicular to this axis, and therefore do not intersect.

To fully test two given convex shapes, the SAT test is repeated for each possible separating axis between the two shapes until either one of the tests finds no overlap in the projections, or all axes have been exhausted. Between a trapezoid and a circle in two dimensions, there are seven possible axes of separation: one axis perpendicular to each edge of the trapezoid (excluding the duplicate axis, since the top and bottom edges of the trapezoid are parallel), and an axis from each vertex in the trapezoid to the centre of the circle.

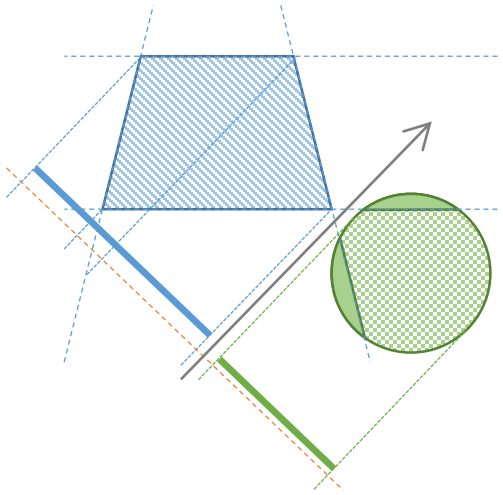

*Figure 3 – Separating Axis Theorem*

*The two shapes have been projected onto the axis parallel to the vector from the lower right vertex of the trapezoid to the centre of the circle.*

*The projections (denoted by the thick lines) do not overlap, therefore there exists a hyperplane that separates the two shapes, perpendicular to the axis of projection (black arrow).*

Whilst there are seven possible axes to test, it is not necessary to test every vertex-to-circle-centre. Only the vertex closest to the centre of the circle need be considered. More specifically, only the axis between

the circle centre and the nearest feature of the trapezoid need be considered. The nearest feature is given by the feature associated with the Voronoi region containing the circle's centre.

When applied to the example previously given in Figure 2, the SAT test correctly finds that the two shapes do not intersect, as shown in Figure 3.

Considering a frustum and a sphere in three dimensions, there are 25 total unique possible separating axes (five axes from the face normals of the frustum, eight axes from each vertex to the sphere centre, and twelve axes to test the edges of the frustum).

Again, only the axes given by the Voronoi region in which the sphere's centre lies must be tested. Selection of the correct Voronoi region is trivial in the case of orthographic projection, as the frustum becomes a cuboid. However determining which region to select when using perspective projection is not trivial, particularly given the asymmetric nature of the tile frustums, and involves several point to feature distance tests.

As a result, it is not possible to quickly determine the correct axis of separation to test, given an arbitrary frustum and sphere. An alternative approach is required for reducing the work done in the collision test.

The SAT algorithm begins with the assumption that the two objects intersect, and then searches for a separating plane to prove that they do not. If a number of separating axes are skipped, the algorithm will likely overestimate the number of positive intersection tests. This is desirable when applied to tile culling, as underestimations in the intersection test will result in light disappearing from portions of the scene.

Due to the shape of the frustums involved in tile culling, it was hypothesised that a given bounding sphere of a light within a typical game world is likely to be closest to one of the frustum's vertices than any of the frustum's other features. Therefore, the vertex-sphere axis tests are likely to be the test which gain the greatest culling benefit for the least work done in the culling compute shader.

### 6.1.2 Chosen Culling Solutions

Initially, two culling solutions were selected for performance analysis: an implementation of plane based frustum culling, as found in the example source code provided by (Advanced Micro Devices, Inc, 2013) and discussed in section 5.2.1, and a full Separating Axis Theorem (SAT) test between the frustum and the bounding sphere. The SAT test was grouped by frustum features, i.e. faces, edges and vertices. Each group can be enabled or disabled via a pre-processor macro at the top of the shader file. This allowed the performance results of each feature group to be investigated, aiming to confirm the hypothesis that vertex tests are the most beneficial.

### 6.1.3 Testing of Culling Solutions

Each combination of edges, faces and vertices for the SAT algorithm were tested. Their culling accuracy was compared against the best-case culling found by the full “brute-force” SAT test involving every feature, and the AMD planes culling algorithm.

From the information obtained by these tests, two further culling algorithms were devised. “Nearest Vertex” which involves a single SAT test against the axis given by the vertex nearest to the sphere’s centre, and “Hybrid” which combines the original AMD planes algorithm with the “Nearest Vertex” algorithm.

Analysis of each method was driven by timing information obtained through real-time performance counters discussed in section 6.2.6, and a heat-map visualisation to show the concentration of lights in each tile.

For testing purposes, each culling algorithm was implemented within the *ForwardPlusTileCulling.fx* compute shader and selectively enabled or disabled using pre-processor macros. From the support provided by the shader and material system in the engine discussed in section 6.2.5, it is then possible to switch between culling modes at runtime by modifying the controlling macro in the source file directly, followed by issuing a “**recompileshaders**” command at the in-engine console.

## 6.2 Engine Design

Many engine design decisions in this project have been heavily influenced by Unreal Engine 4 (Epic Games, 2014). The early development process focussed on the creation of a platform abstraction layer, a shader management system, and a render hardware interface module. The framework built from these systems aided in the efficient implementation of the Forward+ pipeline. The rendering systems focus on the abstraction of Direct3D functionality, and provide various utility classes and optimisations to ensure Direct3D is used in an efficient manner. They also provided the basis to implement the performance analysis system, enabling the comparison between the Forward+ light culling techniques.

The emphasis on engine design intended to focus the pipeline on a more realistic and useful implementation than that typically found in most technology demonstrations, which tend to focus purely on the algorithm being presented. An engine style framework helps ground the pipeline, and prove its viability as a technique for advanced game graphics rendering.

### 6.2.1 Engine Modularity

The engine takes a modular approach within its source code structure, as outlined in Figure 4. Modules are compiled into Dynamic Link Libraries (DLLs), with a well-defined public interface through the use of DLL import/export macros, such as `CORE_API`, or `SHADERCORE_API`. The compilation of each module, and inclusion of appropriate public module headers, is handled through the use of MSBuild property files contained within each module's source directory.

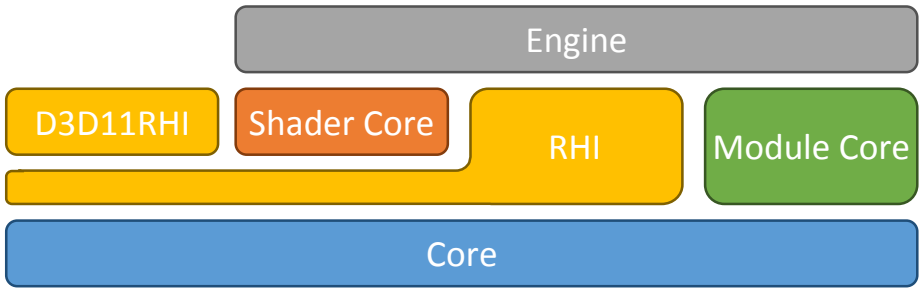


*Figure 4 – Modular Engine Structure*

### 6.2.2 Platform Abstraction Layer

An integral part of the "Core" module is the platform abstraction layer. The layer provides the engine with a common interface for file, window and thread operations. This allows the core module to hide the implementation details of the Win32 API, wrapping its functionality in various C++ classes to be more "engine friendly".

A platform abstraction layer also simplifies the task of porting the engine to a different platform, as the underlying operating system functionality the engine relies on is focused in a single module, which can be replaced with a layer targeting a different operating system as required.

### 6.2.3 Render Hardware Interface

The Render Hardware Interface (RHI) is a concept used within Unreal Engine 4 (Epic Games, 2014). It is similar to the platform abstraction layer, but is responsible for directly communicating with the underlying graphics API, which in this instance is Direct3D 11.1. Like a platform abstraction layer, the RHI provides a common interface to the graphics API, allowing the engine to be retargeted to a different graphics API by providing an alternative implementation of the RHI.

Whilst the implementation within Unreal Engine focuses on providing a minimal interface to Direct3D 11, the RHI created for this project features optimisations such as device context state shadowing. This helps to eliminate any unnecessary API calls which can lead to CPU side performance issues. For example, an attempt to re-set the same rasterization state to the RHI that is already in effect will be discarded, preventing a duplicate call to IDirect3DDeviceContext::RSSetState. This state shadowing technique is applied to almost all state objects and pipeline stages held by the device context.

The RHI also provides similar utility classes as Unreal Engine, to simplify the creation and use of various pipeline state objects, such as rasterizer, blend, and depth-stencil states. The properties of each state object are defined as template arguments in a static class. Direct3D state objects are then only created by the RHI when a particular combination of template arguments have never been used before. Instances with the same arguments reuse the same, shared state object.

```
	// Set the default rasterizer state and disable color writes.
	GDynamicRHI->SetRasterizerState(TStaticRHIRasterizerState<>::GetRHI());
	GDynamicRHI->SetBlendState(TStaticRHIBlendState<false, false,
		ERHIColorWriteMask::None>::GetRHI(), FVector4(), INDEX_NONE);
```

*Code Sample 1 – Use of RHI State Objects (Engine.cpp: Lines 1340-1342)*

### 6.2.4 Efficient Buffer Management

Key to an optimal rendering pipeline is the prevention of pipeline stalls. Stalls arise when the CPU requests read or write access to a resource currently in use by the GPU. An example is during the update of a constant buffer:

1. The CPU creates the buffer and specifies the buffer's initial data.
2. The CPU then issues a draw call that makes use of that buffer.

3. The GPU begins reading data from the constant buffer as part of a draw operation.
4. The CPU attempts to lock the buffer for write access before the GPU has completed the preceding draw operation.

In this scenario, there are two possible outcomes. The graphics driver may allow the CPU to write the new constant buffer data to an area of temporary system memory. The driver then commits the new data to the buffer once the buffer is no longer in use by the GPU, and then frees the temporary memory.

Alternatively, the graphics driver will stall the CPU, preventing the calling application from continuing until the GPU has finished accessing the buffer. When the graphics driver resumes the calling application, it grants direct write access to the buffer.

Both of these outcomes are not desirable from a performance perspective. In the first instance, an additional memory copy is required, and in the second the entire pipeline is flushed and synchronised.

To avoid this situation, careful buffer management is required. Direct3D 11 introduced the `D3D11_MAP_WRITE_NO_OVERWRITE` flag, used when locking a buffer for write access. This flag allows the graphics driver to grant the CPU direct buffer write access, on the condition that the CPU does not write to areas of the buffer the GPU is currently using. The use of this flag prevents any pipeline stalls or additional copies to temporary buffers. It then becomes the engine's responsibility to ensure it does not write to areas of the buffer in use by the GPU, otherwise undefined behaviour may occur.

This engine features a ring buffer system for dynamic vertex, index and constant buffers, similar in implementation to the "Discard-Free Temporary Buffers" discussed by (McDonald, 2012). The Direct3D RHI module tracks the GPU's progress by issuing an event query at the end of each frame. Allocations made by the CPU within a ring buffer are associated with the current frame number. These areas remain allocated and are assumed to be in use by the GPU until the event query for the frame a given area was allocated in returns true, signalling the GPU has completed the frame. The allocated areas are then freed, allowing the CPU to reallocate them as required.

The ring buffer system handles several edge cases so that from the engine's perspective, ring buffers behave exactly like regular buffers. The system "steals" an allocation from a previous frame if the buffer is reused in a subsequent frame without the CPU first making a new allocation. The buffers are also demand grown, doubling in size if not enough space is available to make an allocation. After a few frames have been rendered, the buffers have automatically grown to a sufficient size to handle subsequent frames, and no further reallocations are made, providing the complexity of the frames remain the same.

### 6.2.5 Shader and Material System

A focus of this project was the implementation of a flexible shader and material system. Such a system is capable of easing development effort, as shaders may be recompiled dynamically whilst the engine is running, allowing iterative development of shaders without the overhead of having to restart the engine with each iteration. The system also allows data-driven shader compilation, a key requirement for supporting an artist driven material system such as the one found in Unreal Engine 4 (Epic Games, 2014).

The primary components of the shader system are implemented within the "ShaderCore" module. Each shader defined in a source file must be attached to the engine through the creation of a binding class, as shown in Code Sample 2.

```
class FDepthOnlyVS : public TShader<ERHIShaderFrequency::Vertex>
{
	DEFINE_GLOBAL_SHADER(FDepthOnlyVS, Vertex, STR("DepthOnlyVertexShader.fx"), "VSMain")
	FDepthOnlyVS(const FShaderInitializer& InInitializer)
		: TShader(InInitializer)
	{}
};
```

*Code Sample 2 – Vertex Shader Binding Class (Engine.cpp: Lines 49-55)*

The source shader file and entry point are given by the last two arguments of the `DEFINE_GLOBAL_SHADER` macro. Each binding class automatically registers a new shader type with the shader system on engine start-up. The shader system maintains a list of all the shader types present in the engine, allowing the system to detect when a given shader requires recompilation.

#### *6.2.5.1 Shader Types*

There are two main kinds of shader type: global shaders, and material shaders. Global shaders are stand-alone shaders which have a single compiled instance, and are accessible by the renderer as global singletons. The *ForwardPlusTileCulling.fx* compute shader is an example of a global shader.

Material shaders are shaders which are controlled by the material system and used to render meshes in the world. One compiled instance of a material shader exists for each material asset in the engine. The compiled shader instances themselves are only accessible to the renderer via a material asset instance. If the renderer does not have a valid reference to a material (for example, a given mesh's material field is null), the material system provides a default material instance as a replacement. The *ForwardPlusMainMaterial.fx* pixel shader is an example of a material shader.

#### *6.2.5.2 Shader Modules*

Shader types contain a number of shader modules. Shader modules provide the mechanism to hook up shader compilation to various other graphics systems in the engine. Global and material shaders contain

an HLSL shader module, and a constant buffer shader module. Material shaders contain an additional material shader module.

During compilation, the shader system receives a callback from the D3D11 shader compiler for each included file. The shader system then iterates through the shader modules for the current shader being compiled, and allows the modules to resolve the given filename into HLSL source code, which is passed back to the D3D11 shader compiler. Using this mechanism, the material and constant buffer systems can inject HLSL source code from various different sources, including HLSL code generated at runtime, to produce different shader instances.

#### *6.2.5.3 Recompilation Detection*

Once shaders have been compiled successfully, they are saved to a file along with various metadata, such as the list of files included, and a raw byte array per shader module. This metadata is used to determine if a given shader instance is out of date due to source or asset changes somewhere in the engine. The engine uses file timestamps to detect changes in shader source files. For changes in runtime generated data, the shader system relies on the raw byte array produced by each shader module. The data contained in the array is specific to each module, and allows the module to detect when a change has occurred, for example, a material's type has been changed from "default" to "masked".

#### *6.2.5.4 Compilation Error Recovery*

To maintain productivity during shader development, the shader system must be resilient to compilation failures due to programmer error. To achieve this, the shader system does not unload previously loaded shader instances until a compilation has completed successfully. The newly compiled shader is saved to disk, and then reloaded from disk to replace the shader instance currently in use. By doing this, the shader system can avoid causing an engine crash even if a compilation error has been introduced that has damaged all shaders in the engine. Unreal Engine 4 (Epic Games, 2014) does not achieve this resilience, resulting in shader compilation errors that can cause full engine crashes, requiring the engine to be restarted.

When compilation has failed, a message is displayed on screen to indicate this, and the older shader available from disk is used instead. The engine can also survive restarts with bad shaders. The changes which caused a failed shader compilation will trigger a shader recompilation on engine start-up. Unless the issue has been resolved, the compilation will fail again. The engine will then resort to using the older shader from disk, and display the shader compilation error message. The only time an engine crash will occur is when a global shader has a compilation error, and no older shader is available on disk.

### 6.2.6 Real-time Performance Analysis

To obtain performance results of different light culling techniques, a system for gathering the elapsed time of each draw operation on the graphics hardware is required. Direct3D exposes several event query objects which provide a timestamp of an internal clock for accurate measurement of elapsed time durations on graphics hardware.

The RHI provides the PushEvent and PopEvent functions which enclose draw and dispatch calls in these timestamp queries to establish how long those calls took to complete on the GPU. The RHI builds a hierarchy of these events, which is returned when the GPU has finished processing that frame (which may be several frames later from the CPU's perspective). The tick counts in each event entry within the hierarchy are converted to milliseconds and drawn to the screen. The RHI also performs CPU performance monitoring within the same hierarchy structure.

### 6.2.7 Asset Management

Each asset file begins with the asset header. This contains the asset's Globally Unique Identifier (GUID), and type enumeration. On engine start-up, the asset manager scans the "Content" directory to find all valid asset files, and using the data from their asset headers, builds a map of assets keyed by their GUIDs for fast lookup.

#### *6.2.7.1 Serialization*

Each asset class is responsible for serializing its members on request. References to other assets are serialized using the target asset's GUID. To load an asset, the asset manager first looks up the asset's record in the asset map. If found, the asset's type enumeration value is used to create an instance of the correct class, which then loads the data it requires from the asset's file stream. Inter-asset references are resolved recursively as they are encountered.

#### *6.2.7.2 Asset Identity*

An asset's identity is formed exclusively by its GUID. The GUID is allocated when the asset is created and remains the permanent identity of the asset throughout the engine. All inter-asset references are serialized using the GUID of the target asset referred to.

Assets may also be referred to by their path name, however the path name of an asset is not permanent. It is determined at engine start-up during the directory scan. This allows asset files to be freely moved within the content directory. Since the asset's identity is embedded within the file, references to that asset remain intact, regardless of its position in the content directory.

# 7 RESULTS

## 7.1 DATA GATHERING

Millisecond event times have been collected from the real-time performance analysis system discussed in section 6.2.6. Whilst the data is not averaged across several frames, the values themselves were very stable, fluctuating by only a few fractions of a millisecond between frames.

### 7.1.1 Instability of Offline Performance Analysis

Both Intel GPA (Intel Corporation, 2014), and NVIDIA nSight (NVIDIA Corporation, 2014) were also used to obtain performance results. The advantage of offline performance analysis is that it eliminates any bottlenecks caused by the CPU, which lead to incorrect results gathered from the GPU timestamp event queries. There are cases where the GPU may be left idle during a frame if the CPU does not maintain a constant feed of dispatch/draw commands, but the timestamp system is unable to remove GPU idle time from the results for each event. This leads to timing values that are falsely inflated compared to the actual time spent doing work by the GPU. In these cases, the engine is actually CPU bound.

Offline performance analysis ensures that the GPU is not idle during testing, since the entire command buffer was captured in advance. It is therefore desirable to perform offline analysis when concerned specifically with GPU performance. However, NVIDIA nSight proved to be highly unstable, and crashed the development system before results could be obtained. Similarly, whilst Intel GPA was more stable, the results it provided had an unacceptably high variance, making the data very unreliable. This instability may be related to hardware or graphics driver issues in the test system. As a result, offline analysis was abandoned.

### 7.1.2 Consideration of CPU Performance Bottlenecks

As offline analysis was not possible, results were gathered from the real-time performance analysis built into the engine. Whilst this creates the possibility of discrepancies due to CPU bottlenecks, the CPU workload was constant between tests, with only the culling method changing, an entirely GPU side change which has no effect on the work done by the CPU.

To further ensure CPU bottlenecks are not causing incorrect results, the CPU performance values were monitored alongside those from the GPU. In the case of a CPU bottleneck, the reported time for completion on the CPU will be roughly equal to that of the GPU. Throughout testing, this situation was not found. This is a strong indication that the engine is not CPU bound.

### 7.1.3 Fixed Camera Positions

For consistent results, pre-defined camera positions were included, to ensure the same scene is rendered between tests, or restarting of the engine. These positions were chosen manually after performing preliminary fly-through tests to gauge the general performance of different areas of the scene.

Tests for all culling techniques were repeated across three main camera positions, specifically positions 0, 1, and 5, each of which highlight different characteristics of the chosen culling techniques. The repositioning of the camera can be performed using the "**setcam index**" command, where index is an integer between 0 and 6 inclusive.

### 7.1.4 "Countlights" command

The length of each tile linked list generated by the tile culling compute shader is a good overall indication of how accurate a given culling algorithm is across the entire frame, particularly when compared to the best case culling generated by the brute force full SAT test. To obtain these results, a compute shader is used to sum the lengths of each tile linked list for the current frame into a shared output buffer. The graphics pipeline is then stalled and the CPU awaits the results, which it prints to the engine's console. This operation is performed when the command "**countlights**" is entered at the in-engine console.

The result of the "**countlights**" operation was also particularly useful in confirming the correctness of the culling algorithm. If the value returned by the operation was less than the value returned under the brute force full SAT test, then the algorithm has underestimated the light/tile interactions, and there will be visual discrepancies in the rendered scene.

### 7.1.5 Target System

The data presented here was obtained from a system with the following specifications:

- NVIDIA GeForce GTX 690 Graphics Card
- Intel Core 2 Quad Q9300 CPU @ 2.50 GHz
- 4GB Dual Channel DDR2 RAM @ 800 MHz
- Intel DP35DP Motherboard, with Intel P35 Chipset
- Microsoft Windows 8.1

## 7.2 Culling Accuracy

The following two tables contain the total number of light/tile interactions at two different camera positions for each culling method and varying number of lights. The values are the sum of the lengths of each tile linked list, gathered using the "**countlights**" command as outlined in section 7.1.4.

The SAT based culling methods list the combination of features which were tested, where "E" is edges, "V" is vertices, and "F" is faces.

| | *Number of Light/Tile Interactions* | | | | |
|---|---|---|---|---|---|
| *Culling Method* | *64 Lights* | *128 Lights* | *256 Lights* | *512 Lights* | *1024 Lights* |
| *None* | 921,600 | 1,843,200 | 3,686,400 | 7,372,800 | 14,745,600 |
| *AMD Planes* | 49,106 | 83,415 | 187,200 | 368,046 | 940,339 |
| *SAT (E)* | 50,406 | 89,106 | 191,121 | 361,423 | 950,090 |
| *SAT (V)* | 28,443 | 49,880 | 105,445 | 183,937 | 453,789 |
| *SAT (F)* | 49,030 | 83,302 | 186,885 | 366,915 | 937,421 |
| *SAT (E + V)* | 27,993 | 49,162 | 103,904 | 181,262 | 448,642 |
| *SAT (E + F)* | 39,053 | 67,018 | 140,571 | 258,860 | 654,539 |
| *SAT (V + F)* | 27,726 | 48,633 | 102,769 | 179,009 | 442,891 |
| *SAT (E + V + F)* | 27,681 | 48,559 | 102,627 | 178,710 | 442,258 |
| *Nearest Vertex* | 28,499 | 49,971 | 105,639 | 184,254 | 454,379 |
| *Hybrid* | 27,729 | 48,639 | 102,789 | 179,045 | 443,011 |

*Table 1 – Culling accuracy measurements for camera position 0, at a resolution of 1280x720*

| | *Number of Light/Tile Interactions* | | | | |
|---|---|---|---|---|---|
| *Culling Method* | *64 Lights* | *128 Lights* | *256 Lights* | *512 Lights* | *1024 Lights* |
| *None* | 921,600 | 1,843,200 | 3,686,400 | 7,372,800 | 14,745,600 |
| *AMD Planes* | 99,199 | 186,105 | 303,138 | 507,752 | 839,107 |
| *SAT (E)* | 89,340 | 174,198 | 290,225 | 493,689 | 847,912 |
| *SAT (V)* | 64,770 | 110,618 | 181,715 | 308,129 | 519,727 |
| *SAT (F)* | 98,942 | 185,535 | 302,310 | 506,125 | 836,028 |
| *SAT (E + V)* | 62,959 | 107,152 | 175,252 | 292,178 | 483,699 |
| *SAT (E + F)* | 76,797 | 139,401 | 227,457 | 380,007 | 627,147 |
| *SAT (V + F)* | 61,617 | 104,051 | 168,521 | 278,077 | 452,547 |
| *SAT (E + V + F)* | 61,598 | 104,007 | 168,418 | 277,877 | 452,037 |
| *Nearest Vertex* | 64,822 | 110,727 | 181,948 | 308,586 | 520,655 |
| *Hybrid* | 61,622 | 104,060 | 168,540 | 278,126 | 452,679 |

*Table 2 – Culling accuracy measurements for camera position 1, at a resolution of 1280x720*

During the initial investigation to culling accuracy, the SAT tests were compared to the AMD Planes implementation. In almost all cases it was found that any combination of SAT feature testing provides more accurate culling than the use of planes alone.

More interestingly, the vertices SAT test provides the greatest gain in culling accuracy, as outlined in Figure 5 and Figure 6. All culling algorithms involving the vertex SAT test have considerably tighter culling than those methods without the vertex test. The variation between the algorithms using the vertex test is very minor, and is less than 3% between the best and worst cases.

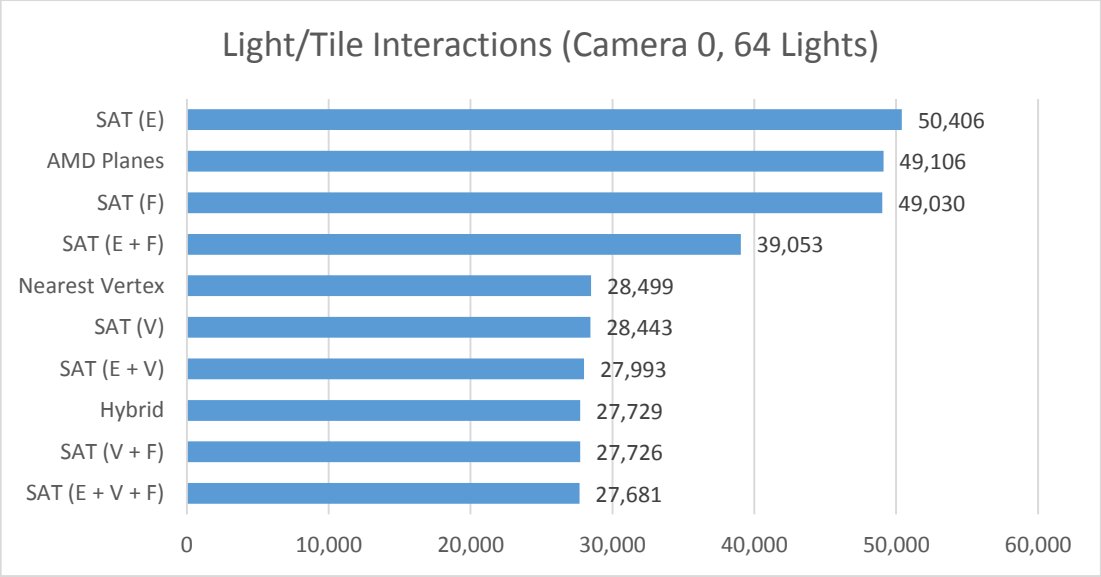


*Figure 5 – Number of light/tile interactions at Camera 0, 64 lights, and 1280x720 resolution*

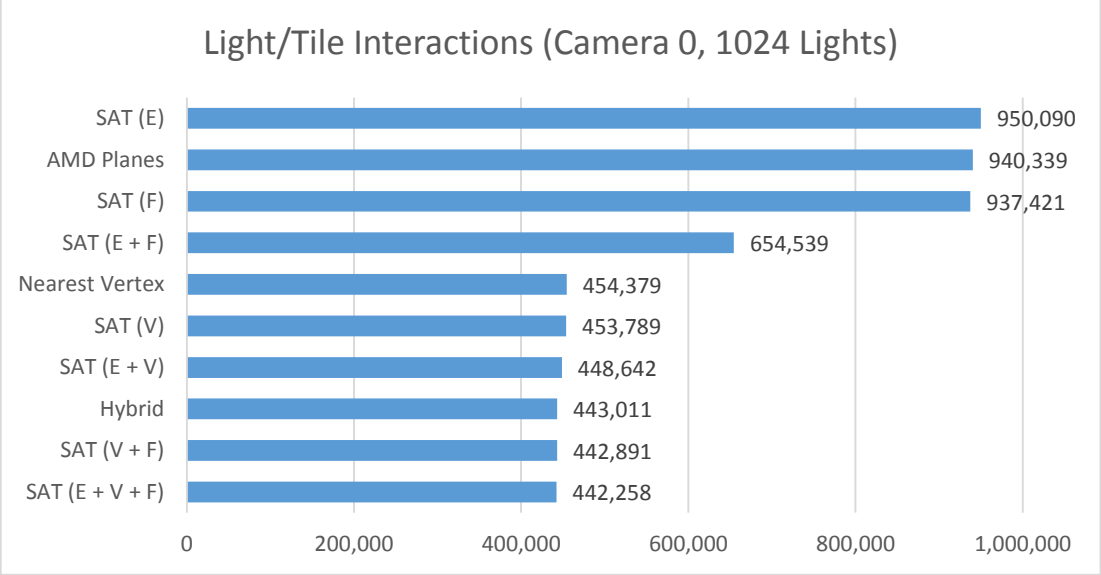


*Figure 6 – Number of light/tile interactions at Camera 0, 1024 lights, and 1280x720 resolution*

Similar results are found at camera position 1, and at higher resolutions such as 1920x1080.

## 7.3 Real-Time Performance Results

From this lack of variation and the large effect the vertex SAT tests have on culling accuracy, the vertex SAT test along with the AMD Planes algorithm were taken forward for direct performance testing, measuring the actual compute time required on the target hardware to render the scene with these given culling algorithms.

From these observations, a third culling algorithm was also devised, and named “Nearest Vertex”, as outlined in section 6.1.3, aiming to reduce the complexity of performing multiple SAT vertex tests to only a single test with the vertex closest to the sphere’s centre.

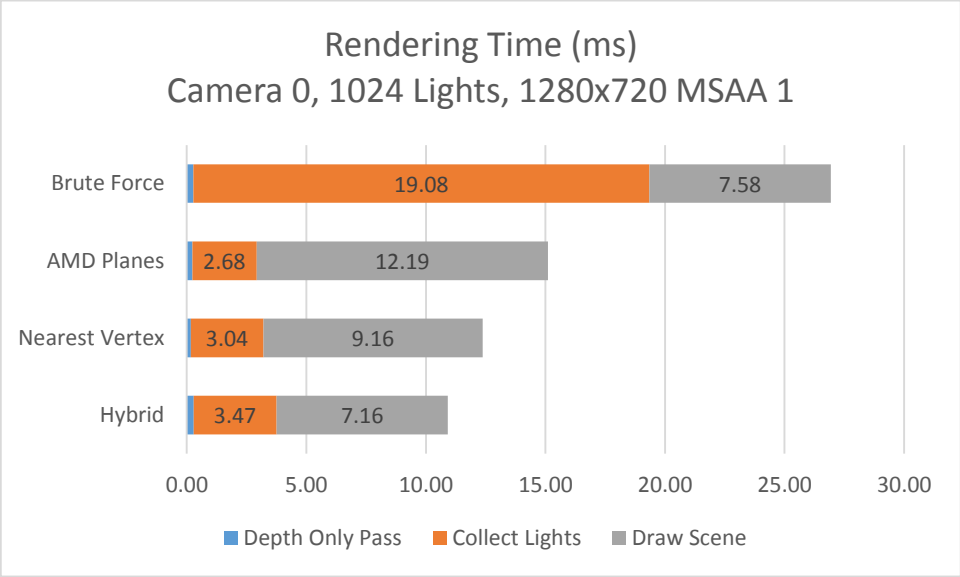


*Figure 7 – Rendering time in milliseconds, at Camera 0, 1280x720 resolution, and no MSAA*

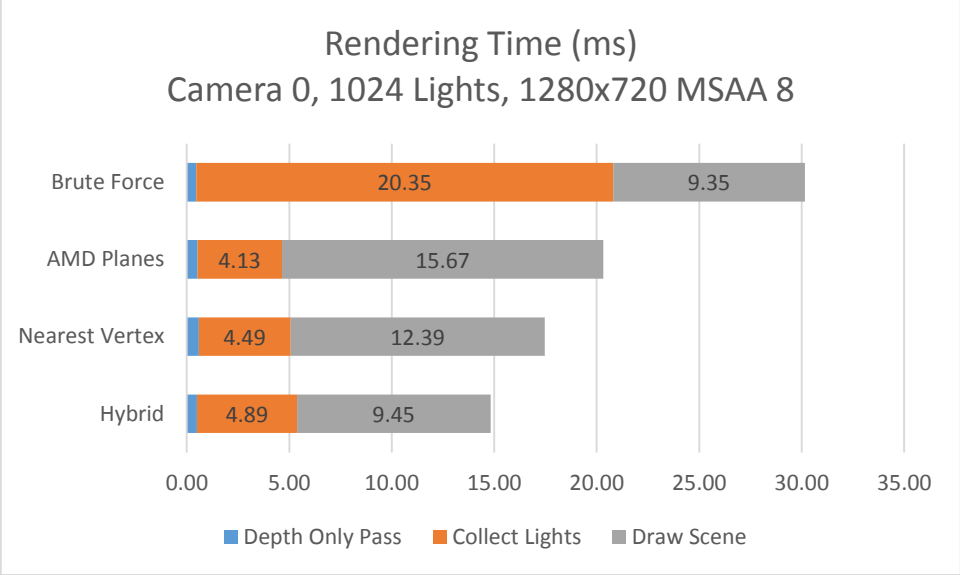


*Figure 8 – Rendering time in milliseconds, at Camera 0, 1280x720 resolution, and 8x MSAA*

At camera position 0, the values obtained from the Brute Force, AMD Planes and Nearest Vertex algorithms were largely as expected, as shown in Figure 7 and Figure 8. Whilst the Brute Force algorithm has the greatest cost in the Collect Lights phase, it has the lowest cost in the final Draw Scene phase.

Likewise, whilst the AMD Planes algorithm has the least cost in the Collect Lights phase (due to the relative simplicity of the algorithm), it has a larger final Draw Scene phase cost due to the inefficient culling.

The effect of Multi-Sample Antialiasing is noticed most greatly in the cost of the Draw Scene phase, as is expected. The only additional work required in the Collect Lights phase to allow MSAA is to compute the minimum and maximum depth values per tile from each sample, rather than each pixel.

If considering only camera position 0, then the Nearest Vertex algorithm would be the most efficient of the three algorithms tested. However, similar results were not found at alternative camera positions.

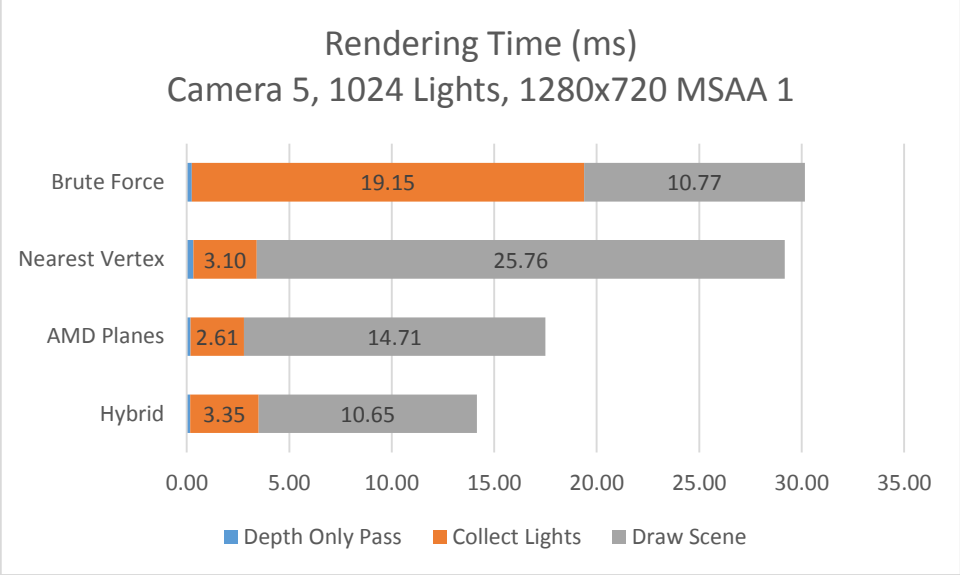


*Figure 9 – Rendering time in milliseconds, at Camera 5, 1280x720 resolution, and no MSAA*

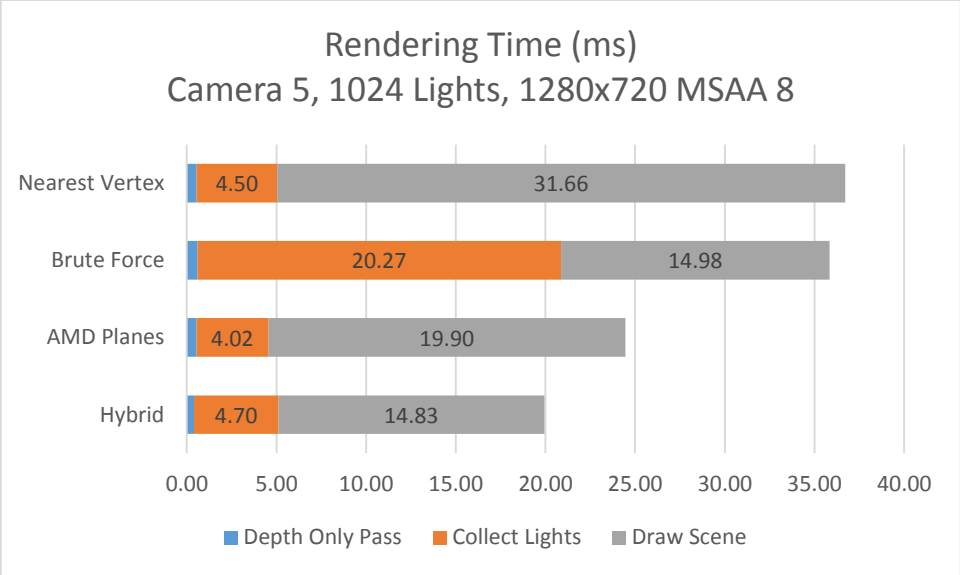


*Figure 10 – Rendering time in milliseconds, at Camera 5, 1280x720 resolution, and 8x MSAA*

Figure 9 and Figure 10 show that camera position 5 exhibits drastically different results to camera position 0 for the Nearest Vertex algorithm. Whilst the Collect Lights phase has remained at approximately the same proportion to the other techniques, the Draw Scene cost is vastly inflated, even surpassing the total cost of the Brute Force algorithm with 8x MSAA. This despite the Nearest Vertex algorithm achieving a lower number of light/tile interactions than the AMD Planes algorithm, according to the "**countlights**" test, as outlined in Table 3.

| *Camera Position* | *Culling Mode* | | | |
|---|---|---|---|---|
| | *AMD Planes* | *Brute Force* | *Nearest Vertex* | *Hybrid* |
| *0* | 940,339 | 442,258 | 454,379 | 443,011 |
| *5* | 1,059,524 | 544,656 | 662,369 | 546,454 |

*Table 3 – Number of light tile interactions at Cameras 0 and 5, at 1280x720*

## 7.4 Heat Map Analysis

It was determined that the likely cause of such variance in the rendering cost for the Nearest Vertex algorithm is linked to the distribution of light/tile interactions in relation to depth disparities in the scene.

To investigate this further, a "heat map" visualisation was created. The heat map displays a colour for each tile chosen according to the total number of lights linked to that tile, determined by the culling algorithm. The colour assigned to each tile is chosen according to the following scale:

| Colour | Number of Lights |
|---|---|
| **Black** | 0 |
| **Blue** | 1 – 7 |
| **Cyan** | 8 – 15 |
| **Green** | 16 – 31 |
| **Yellow** | 32 – 63 |
| **Red** | 64 – 127 |
| **White** | 128 + |

*Table 4 – Heat Map Colour Scale*

It is clear from comparing the heat maps at Camera 5 for AMD Planes (Figure 17) and Nearest Vertex (Figure 18) culling, that whilst Nearest Vertex provides better overall culling, there are a larger number of white tiles around areas with large depth disparities, such as the flag poles and edges of the columns. These particular tiles have very poor culling compared to the best case solution given by Brute Force culling (Figure 20).

The additional work required to render the scene under these tiles is becoming the bottleneck during the Draw Scene phase, creating the large increase in overall frame rendering time as shown in Figure 9 and Figure 10, particularly when MSAA is involved.

Analysis of the heat maps also explains how the Nearest Vertex culling algorithm produces better results than AMD Planes at Camera 0. At this position, there are less tiles with large depth disparities, resulting in the number of white tiles in Figure 13 being approximately the same as Figure 12. In this configuration, the Draw Scene phase benefits from the tighter culling granted by the Nearest Vertex algorithm.


*Figure 11 – Final rendered scene at Camera 0*
*Rendered at 1280x720 resolution, with Hybrid Culling, 8x MSAA, and 1024 lights.*

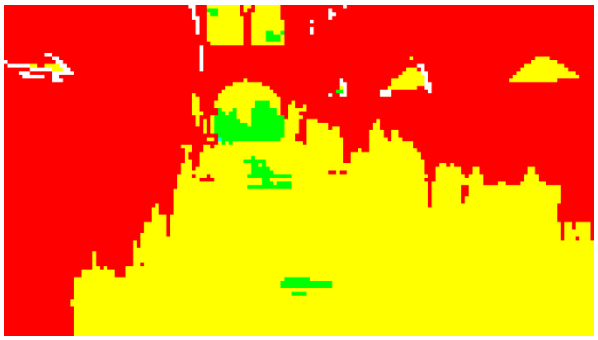

*Figure 12 – Heat map at Camera 0, for AMD Planes Culling*

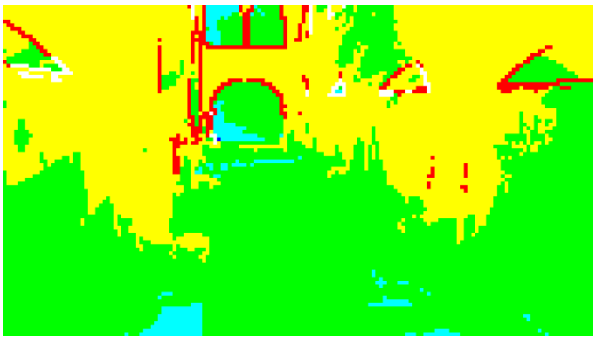

*Figure 13 – Heat map at Camera 0, for Nearest Vertex Culling*

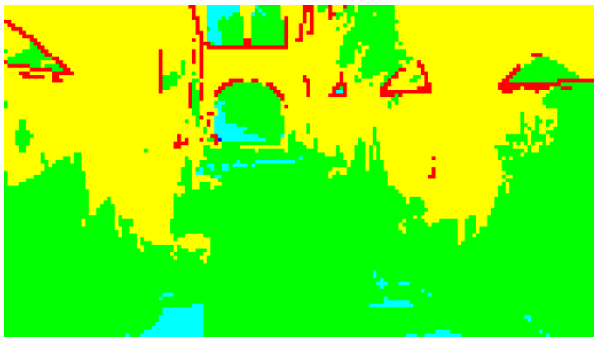

*Figure 14 – Heat map at Camera 0, for Hybrid Culling*

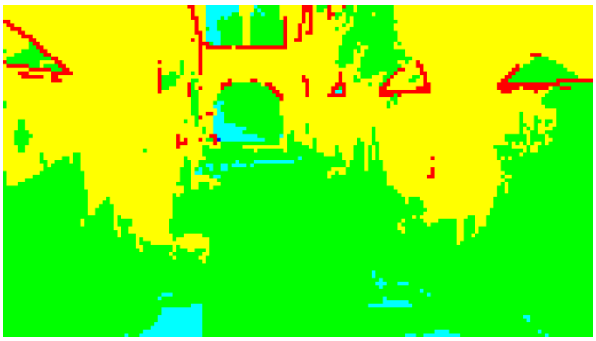

*Figure 15 – Heat map at Camera 0, for Brute Force Culling*


*Figure 16 – Final rendered scene at Camera 5*
*Rendered at 1280x720 resolution, with Hybrid Culling, 8x MSAA, and 1024 lights.*

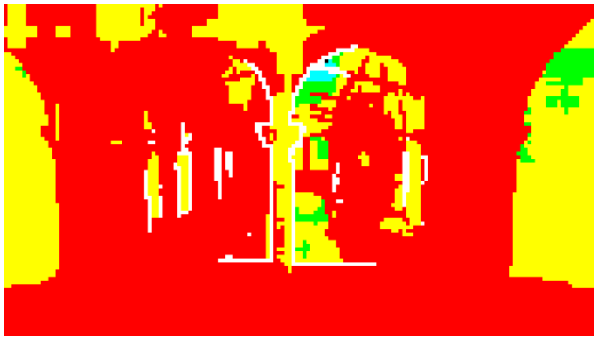

*Figure 17 – Heat map at Camera 5, for AMD Planes Culling*

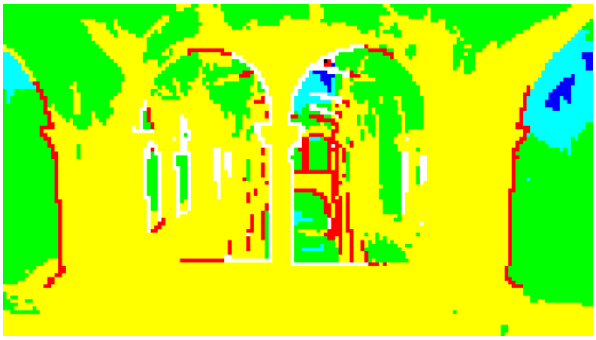

*Figure 18 – Heat map at Camera 5, for Nearest Vertex Culling*

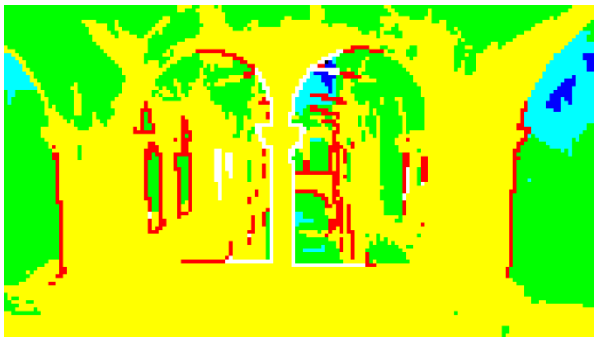

*Figure 19 – Heat map at Camera 5, for Hybrid Culling*

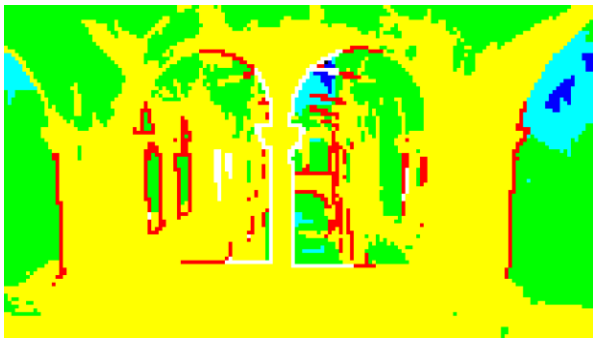

*Figure 20 – Heat map at Camera 5, for Brute Force Culling*

## 7.5 HYBRID CULLING

The Nearest Vertex algorithm is not suitable for use as it is too sensitive to changes in depth across the pixels of a tile. The results discussed in the previous sections drove the development of an improved culling algorithm to avoid the issues present in both the Nearest Vertex and AMD Planes algorithms.

The Hybrid Culling algorithm combines the AMD Planes and Nearest Vertex tests into a single algorithm, as detailed in Algorithm 7. The plane tests are computed first, as their computational simplicity provides an "early-out" case for the majority of lights which fail the plane tests, avoiding the computation of the nearest vertex or projection of the frustum. The Hybrid algorithm replaces the second **FOR** loop in Algorithm 6, with the rest of the Forward+ pipeline remaining unmodified.

```
FOR each tile in the viewport
        find minimum and maximum depth values within the tile
        compute frustum of the tile using depth values
        FOR each light in the scene
                LET intersection = TRUE
                FOR each plane in frustum
                        IF light's bounding sphere lies on the negative half space of the plane THEN
                                intersection = FALSE
                                BREAK
                        ENDIF
                ENDFOR
                IF intersection is TRUE THEN
                        find nearest frustum vertex to light's position
                        project frustum and sphere against axis from vertex to light position
                        IF projection of frustum overlaps projection of sphere
                                atomically insert light index into linked list for tile
                        ENDIF
                ENDIF
        ENDFOR
ENDFOR
```

*Algorithm 7 – Hybrid Light Tile Culling Algorithm*

The results for the Hybrid algorithm are provided alongside the results for the other algorithms in the above figures. Specifically, the number of light/tile interactions given for the Hybrid algorithm in Table 1 and Table 2 show only a 0.17% increase over the best case, Brute Force algorithm.

Analysis of the heat maps generated by the Hybrid algorithm show a similarly tight culling, with very little difference between Figure 14 and Figure 15, and between Figure 19 and Figure 20.

The Hybrid algorithm provided the best case performance results in all real-time performance tests conducted, not only at the pre-defined test camera positions, but throughout the scene when directly controlling the camera.

The accuracy of the culling is reflected in the real-time performance results gathered for the Draw Scene phase, listed in Figure 7, Figure 8, Figure 9, and Figure 10. The Draw Scene phase for the Hybrid algorithm is approximately equal to the same phase for the Brute Force algorithm in these cases. The benefit of the Hybrid solution is that it achieves very tight light culling, on par with the best case Brute Force algorithm, but at a substantially reduced computational cost to the Brute Force algorithm.

Whilst the Hybrid algorithm is the most expensive of the three optimised algorithms, the computational saving it makes in the Draw Scene phase greatly outweighs this cost, making it the most optimal of the three solutions.

# 8 CONCLUSION AND EVALUATION

Referring to section 4.5, the primary focus of this project was the investigation and improvement of the light culling algorithm employed within a Forward+ rendering pipeline. The improvement of light culling in such a pipeline will improve the overall efficiency of the entire pipeline. The greater the efficiency of the rendering pipeline, the more scope there is for added scene complexity, and visual fidelity whilst maintaining an interactive frame rate.

As outlined in section 7.5, a new algorithm for solving light bounding sphere verses tile frusta collision tests has been created. In all test cases explored, the Hybrid culling algorithm exceeds the performance of all other algorithms tested. It has therefore improved on the efficiency of the standard Forward+ pipeline by reducing computation time required in the final Draw Scene phase.

A secondary focus of this project was the implementation of the Forward+ pipeline within a modular engine framework capable of supporting a robust shader and material system, and asset management. Throughout the development of the Forward+ pipeline, the underlying shader system proved invaluable in the fast iteration and debugging of shaders.

## 8.1 LIMITATIONS

### 8.1.1 Support for Orthographic Projections

The current implementation of the AMD Planes algorithm, which forms the first half of the Hybrid algorithm, does not support orthographic projections. To simplify the computation of the planes for the top, bottom, left and right sides of each tile, an assumption is made that the plane intersects the view-space origin. In doing so, the plane may be constructed using only the two view-space vertices from an edge on the near plane of a tile frustum, as a simple cross product.

Whilst this would likely not be an issue in a game scenario which tend to exclusively use perspective projections, it will cause issues for any tools or editor software created for the engine which use orthographic viewports. In these cases, the light culling algorithm will break, leading to incorrect lighting results.

To correct this, the plane computation needs to be extended to handle orthographic projection cases where the planes do not converge on the view-space origin, by computing the plane given the two near plane vertices and one vertex at the far plane. This is a very small change, and would likely result in only a minor performance hit in the compute shader execution time.

### 8.1.2 Resolution Issues

The current implementation requires that the resolution of the viewport be an exact multiple of the tile size, currently set as 8x8 pixels. This ensures that pixels on the right and bottom edges of the viewport lie exactly within a tile. Issues arise when resolutions such as 1366x768 or 1680x1050 are used, which result in the tiles on the far right and bottom edges being clipped by the bounds of the viewport.

The current tile culling compute shader does not account for this, leading to incorrect results if these resolutions are used. This would be a particular limitation for end users on desktop and laptop systems, as many standard monitor resolutions are not completely divisible by eight. This would be less of an issue on modern console platforms, as their supported output resolution is usually restricted to either 720p (1280x720), or 1080p (1920x1080), in accordance with high definition television standards.

To correct this, the computation of tile bounds must be modified to account for these additional pixels. A special case must also be made when reading depth values from pixels in these clipped tiles, to avoid reading any values from outside the bounds of the depth buffer.

## 8.2 Future Work

### 8.2.1 Culling Improvements

There still remains a number of areas for investigation in the optimality of the light/tile culling algorithm.

#### *8.2.1.1 Screen Space Light Acceleration Structure*

Under the current Hybrid algorithm, all tiles are tested against all lights. The AMD Planes section of the Hybrid algorithm tests each plane in turn against a given light sphere, regardless of the relative positions of the tile and the light sphere in screen space. Tiles on the extreme left side of the screen will still be testing lights on the extreme right, which are likely not to be intersecting that tile's frustum.

A better approach may be to first coarsely pre-filter the lights into a screen space structure, such as a quad-tree. Using such a structure, the frustum-sphere tests will only be made between lights and tiles in close proximity in screen space. This has the potential to reduce the cost of the Collect Lights compute shader phase, whilst maintaining the culling accuracy given by the Hybrid algorithm.

#### *8.2.1.2 Improved Culling for Tiles with Large Depth Disparities*

Since tiles are constructed using the minimum and maximum depth value obtain from all pixels within the tile, it is possible for tiles to have frustums which stretch very far into the scene. Whilst the Hybrid algorithm provides very accurate culling, these long, thin frusta still results in a high number of false

positives for these tiles. These are particularly visible in Figure 19 at the edges of the foreground column. The frusta for these tiles extend from the foreground column, through the entire open space at the centre of the scene, and end at the far wall. As such, they intersect a great number of lights, most of which will have no effect on any of the pixels within those tiles.

This effect is a key limitation of the standard Forward+ pipeline, and leads to suboptimal performance in areas with large depth disparities within tiles. This may cause performance issues when rendering outdoor/forest scenes with a large amount of masked foliage geometry. The high frequency detail in the leaves of the foliage is likely to create large numbers of tiles with extended frusta.

An improvement can be made to the standard Forward+ pipeline by introducing a "2.5D clustering" scheme, as described by (Harada, et al., 2013). The depth extent of each tile frustum is split into a number of sub-frustums. The occupancy of scene geometry is determined in each sub-frustum and marked in a bitmask for the tile. During the Collect Lights compute shader phase, lights are only linked to a given tile if the light intersects at least one of the sub-frustums marked as occupied.

This extension would fit into the existing engine with minimal changes, and would still benefit from the tighter sphere-frustum culling given by the Hybrid algorithm. Once implemented, it will provide much more stable performance results in scenes with a larger number of tile depth disparities, such as those caused by foliage and masked geometry.

#### *8.2.1.3 Spot Light Cones*

Spot lights were not a focus of the performance investigation. They are currently implemented using the same sphere-based culling scheme as point lights. The field of influence of a spot light is cone shaped, so sphere based culling is very wasteful. An alternative culling technique should be investigated to provide tighter cone-based culling to spot lights. This would likely involve direct SAT tests between the frustum and cone shapes, or involve an alternative approximation of the cone.

### 8.2.2 Graphical Improvements

The Forward+ technique is fully compatible with a number of other advanced lighting, shading and post-processing techniques. When dealing with a scene more representative of a production game, the optimisations obtained within this project provide additional performance scope in which these techniques may be implemented, without sacrificing frame rate. Such features would include Shadow Mapping, High Dynamic Range lighting, Screen Space Ambient Occlusion, and various post-processing effects to achieve higher image quality, alongside more complex materials and BRDFs to take advantage of the inherent flexibility in a forward based rendering pipeline.

### 8.2.3 Asset Management Improvements

The asset system employed by the engine, whilst suitable for supporting the implementation and refinement of the Forward+ pipeline, is lacking a number of features such as versioning support, required by more advanced game engines. Versioning support allows changes to be made to the underlying data structures, without breaking compatibility with older assets generated by the engine. In a production environment, such changes can happen quite regularly, and without versioning support in asset serialization, all assets affected by a change would have to be upgraded manually. This process would be particularly cumbersome and error-prone.

Unreal Engine 4 (Epic Games, 2014) primarily handles asset versioning through a C++ reflection system which relies on a build tool to parse type information from specific C++ header files. The resulting reflection data is used not only for asset serialization to disk, but also for data replication over the network in multiplayer game scenarios, and dynamic construction of classes via their “Blueprint” system.

### 8.2.4 Shader System Improvements

Binding a vertex shader to the RHI is a rather manual process, requiring the input vertex format to be specified at the creation of the bound shader state object. Feeding materials with vertex data is also very manual, restricting the flexibility of the material system. The engine is currently unable to support alternative vertex sources, such as particle systems, skinned characters, and instanced foliage, without manual implementation of vertex shaders for each specific material.

To rectify this, the material and shader system should be extended to support the concept of “vertex factories”, such as those implemented by (Epic Games, 2014) in Unreal Engine 4. A vertex factory provides a unified interface to the material system, allowing a given material asset to be applied to a number of different geometry types, regardless of how the vertex data for that geometry is generated.

Within the context of this engine, vertex factories should be implemented as an additional shader module, allowing the new vertex factory system to inject vertex shader definitions as HLSL source code. Vertex input formats would then be tied to a specific vertex factory type, removing the need to specify the input format when binding the vertex shader to the RHI.

## 8.3 Closing Statement

The algorithm presented by this project has successfully improved upon the standard plane-sphere based light culling employed within Forward+ rendering, as confirmed by the results presented in section 7. The performance savings found in this project should not be taken in isolation, as there are still many avenues of performance improvements to explore, as detailed in section 8.2.

The dynamic nature of the graphics industry has never been more apparent than now, with the introduction of general purpose compute, and the release of new video games consoles and PC platforms. Microsoft's recent announcement of DirectX 12 (Sandy, 2014) at the Games Developers Conference 2014 in San Francisco, has the potential for a revolution in game engine design. The removal of the graphics driver abstraction overhead that has plagued PC based games development for years is a great step forwards, providing console-style low-level access to graphics hardware, allowing developers to get the most out of the available hardware.

Whilst a relatively new technique within the graphics community, Forward+ rendering has the potential to provide the foundation of next generation graphics in games. Of course, as with all elements of programming, the costs of such a system should be evaluated alongside the costs of others, and the best solution chosen for a given game production. With this in mind, Forward+ still remains a strong candidate for providing the next generation of graphics fidelity in future production game engines and interactive rendering systems.

# 9 REFERENCES


Advanced Micro Devices, Inc, 2013. *Radeon SDK - TiledLighting11 v1.1.* [Online]
Available at: http://developer.amd.com/tools-and-sdks/graphics-development/amd-radeon-sdk/
[Accessed 20 March 2014].

Carmack, J. & Kilgard, M., 2009. *Practical and Robust Shadow Volumes - John Carmack on Shadow Volumes.* [Online]
Available at: http://web.archive.org/web/20090127020935/http://developer.nvidia.com/attach/6832
[Accessed 20 March 2014].

Epic Games, 2014. *Unreal Engine 4.* [Online]
Available at: https://www.unrealengine.com/
[Accessed 21 March 2014].

Gilbert, E., Johnson, D. & Keerthi, S., 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation,* 4(2), pp. 193-203.

Harada, T., McKee, J. & Yang, J. C., 2012. Forward+: Bringing Deferred Lighting to the Next Level. *EUROGRAPHICS 2012.*

Harada, T., McKee, J. & Yang, J. C., 2013. Forward+: A Step Toward Film-Style Shading in Real Time. In: W. Engel, ed. *GPU Pro 4: Advanced Rendering Techniques, Volume 4.* s.l.:s.n., pp. 115-136.

Intel Corporation, 2014. *Intel Graphics Performance Analyzers.* [Online]
Available at: http://software.intel.com/en-us/vcsource/tools/intel-gpa
[Accessed 21 March 2014].

Klint, J., 2008. *Deferred Rendering in Leadwerks Engine.* [Online]
Available at: http://www.leadwerks.com/files/Deferred_Rendering_in_Leadwerks_Engine.pdf
[Accessed 20 March 2014].

Lee, M., 2009. *Pre-lighting in Resistance 2.* [Online]
Available at: http://www.insomniacgames.com/tech/articles/0409/files/GDC09_Lee_Prelighting.pdf
[Accessed 23 March 2014].

McDonald, J., 2012. *Don't Throw it all Away: Efficient Buffer Management.* [Online]
Available at:
https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/gdc12/Efficient_Buffer_Management_McDonald.pdf
[Accessed 23 March 2014].

NVIDIA Corporation, 2014. *CUDA Parallel Computing.* [Online]
Available at: http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html

NVIDIA Corporation, 2014. *NVIDIA Nsight Visual Studio Edition.* [Online]
Available at: https://developer.nvidia.com/nvidia-nsight-visual-studio-edition
[Accessed 21 March 2014].

Sandy, M., 2014. *DirectX 12 - DirectX Developer Blog.* [Online]
Available at: http://blogs.msdn.com/b/directx/archive/2014/03/20/directx-12.aspx
[Accessed 03 April 2014].

Shishkovtsov, O., 2005. *GPU Gems 2. Chaper 9 - Deferred Rendering in S.T.A.L.K.E.R..* [Online]
Available at: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html
[Accessed 20 March 2014].

Valient, M., 2007. *Deferred Rendering in Killzone 2.* [Online]
Available at: http://www.guerrilla-games.com/presentations/Develop07_Valient_DeferredRenderingInKillzone2.pdf
[Accessed 20 March 2014].

# 10 APPENDICES

## 10.1 ENGINE DIRECTORY STRUCTURE AND WORKING DIRECTORY

This section details the directory structure used by the engine to organise source code, assets and binaries. In order for the engine to find the content and shader directories, it must be run from the root directory, not the binaries output directory.

The paths in the following table are relative to the working directory.

| Folder | Description |
|---|---|
| **Assets\Engine** | Engine specific source texture and material assets, and import .ebat scripts. |
| **Assets\Sponza** | All source assets specific to the Sponza atrium scene, and import .ebat scripts. |
| **Binaries** | Output build directory organised by target platform (Win32/x64). |
| **Build\Intermediate** | Compiler Intermediate build directory. Contains object files and build logs. |
| **Build\Properties** | Contains MSBuild property files common to all modules. |
| **Content** | Imported .asset files created by the engine, and searched on engine start-up. |
| **DataCache\Shaders** | Output compilation directory for shaders, used by the shader system. |
| **Source\Engine** | All source code for the main engine modules as defined in section 6.2.1. |
| **Source\Shaders** | HLSL shader source code for all shaders in the engine. |
| **Source\ThirdParty** | Third party source code and static libraries used by the engine. |
| **Source\Tools** | Content processing tools used during the development of the engine. |

*Table 5 – Engine Directory Structure*

## 10.2 ENGINE FIRST RUN AND CONTENT IMPORTING

Before the engine can run the Sponza atrium demo, the content files must be imported from the Assets directory. The in-engine console is displayed immediately after engine start-up. On the first run issue the following command:

**batch Assets\ImportAll.ebat**

This will execute a batch script to create and import all engine and Sponza assets, and then save the imported assets to disk under the **Content** directory.

Once the content has been imported, issue the **demo** command to load the Sponza demo scene.

## 10.3 SUPPORTED CONSOLE COMMANDS

| Command | Description |
|---|---|
| **asset** | Performs actions on the specified asset. See section 10.4. |
| **batch** | Executed a series of commands as defined by the specified batch file. |
| **clear** | Clears the console log. |
| **close** | Closes the Sponza demo scene, if running. |
| **countlights** | If the Sponza demo is running, executes a compute shader at the end of the next frame to count the total number of light/tile interactions. Reports the count in the console log. |
| **create** | Creates a new asset of the specified name and type. |
| **demo** | Loads the Sponza demo scene. |
| **dumpcampos** | Prints the current camera position and rotation as vectors to the console log. |
| **exit** | Shuts down the engine, and terminates the application. |
| **info** | Prints information specific to the specified asset. |
| **msaa** | Sets the Multi-Sample Anti-Aliasing count. Valid counts are 1 (MSAA disabled), 2, 4 and 8. |
| **numlights** | Sets the number of lights visible in the Sponza scene. Valid numbers are 0 to 1024 inclusive. |
| **recompileshaders** | Scans for any out of date shaders, and if any are found, recompiles them. For all shaders that were recompiled successfully, they are dynamically loaded into the engine, replacing any previously loaded instance.<br>Compile failures are printed to the console log, and a fixed error message is displayed in the centre of the screen when running the Sponza demo. |
| **save** | Saves all modified/newly created assets to disk. |
| **setcam** | Sets the camera position in the Sponza demo to one of 7 predefined positions. Valid position indices are 0 to 6 inclusive. |
| **setres** | Sets the resolution and full screen mode of the engine using the following format:<br>[width]x[height][w\|f]<br>e.g.<br>**setres 1280x720w**<br>**setres 1920x1080f** |
| **stats** | Toggles the display of real-time performance counters in the Sponza scene. |
| **visdebug** | Toggles the display of a 3-axis measuring grid in the Sponza scene. Grid lines are spaced one world unit apart. |
| **vislightvolumes** | Toggles the display of a series of axis aligned bounding boxes, which are used when generating the list of randomly positioned lights. |
| **visnumlights** | Toggles the display of the heat-map visualization overlay in the Sponza scene. |
| **visorbs** | Toggles the display of small semi-transparent orbs at the position of each point light. |
| **visres** | Toggles the back buffer resolution text in the upper left corner of the window. |

*Table 6 – List of supported console commands*

## 10.4 Supported Asset Commands

The asset command allows the manipulation of asset properties, and performs tasks such as importing textures and meshes, and setting material parameters. Asset commands begin with the asset's name and a command action specific to that asset type, in the following format:

**asset [asset name] [action] [options]**

The options field is dependent on the asset type and action chosen. Asset names are given as the file path relative to the Content directory.

The following is an example of an asset command to set the "**DiffuseTexture**" material parameter on the "**MI_Bricks**" material instance to the 2D texture asset "**Bricks_A_Diffuse**":

**asset Materials\Sponza\MI_Bricks set DiffuseTexture**
**texture2d Textures\Sponza\Bricks_A_Diffuse**

### 10.4.1 Texture 2D Asset Commands

| Command | Description |
|---|---|
| **import [source image] {options}** | Imports the image data from the specified source file. Options are given as key=value pairs.<br><br>The following options are supported:<br>• Compress<br>Specifies the compression format to apply to the image data. Valid values are 0 (uncompressed) to 7 inclusive.<br><br>Most engine textures use compress=3 for BC3 encoding. |

*Table 7 – Supported Texture Asset Commands*

### 10.4.2 Material Asset Commands

| Command | Description |
|---|---|
| **import [HLSL file]** | Links the material with the specified source HLSL file, then triggers a shader compilation for that material. |
| **option [key=value]** | Sets the specified option on the material to the given value.<br><br>Only one option is supported:<br>• mode=[default\|masked]<br>Sets the material rendering mode to default (solid) or masked. |

*Table 8 – Supported Material Asset Commands*

### 10.4.3 Material Instance Asset Commands

| Command | Description |
|---|---|
| **parent [parent asset name]** | Sets the parent of the material instance to the specified material or material instance assert. |
| **set [variable name] [type] [value]** | Sets the value and type of the specified variable on the material instance. Variables set on a material instance override the value provided by the parent, if specified.<br><br>Type is one of the following:<br>• scalar<br>• float2<br>• float3<br>• float4<br>• texture2d<br>Components of vector type values are separated by a comma character, for example:<br>1.0, -3.5, 2<br><br>Texture 2D assets are specified using their asset path name. |
| **unset [variable name]** | Removes the specified variable from the material instance. This returns the variable to its default (inherited) value. |
| **list** | Lists all the variables defined in this material instance and their values. |

*Table 9 – Supported Material Instance Asset Commands*

### 10.4.4 Static Mesh Asset Commands

| Command | Description |
|---|---|
| **import [static mesh file] {scale=value} {center} {discard=meshname}** | Imports mesh data from the specified source file.<br><br>Additional options:<br>• **scale=value**<br>Scales the source mesh by the specified value.<br>• **center**<br>Centres the source mesh in object space.<br>• **discard**<br>Does not import meshes with the specified name. Multiple discard meshes may be specified.<br><br>Once the source data has been imported, all mesh parts have the engine default material assigned. |
| **material [bone name] [mesh name] [material asset name]** | Applies the specified material or material instance asset to the mesh part with the given bone and mesh name. |

*Table 10 – Supported Static Mesh Asset Commands*

## 10.5 Engine Input and Controls

| Keyboard | Xbox 360 Controller | Description |
|---|---|---|
| **Escape** | | Close the console, or exit the engine. |
| **Tab** | | Display the small console. |
| **` (Back quote)** | | Display the full screen console. |
| **Page Up/Down** | | Scroll the full screen console log. |
| **Up/Down Key** | | Previous/Next command in console history. |
| **Enter** | | Execute command in console. |
| **W** | Left Stick Up | Move Camera Forwards |
| **S** | Left Stick Down | Move Camera Backwards |
| **A** | Left Stick Left | Move Camera Left |
| **D** | Left Stick Right | Move Camera Right |
| **Q** | Right Trigger | Move Camera Upwards |
| **E** | Left Trigger | Move Camera Downwards |
| **Z** | Left Shoulder | Decrease Camera Speed |
| **X** | Right Shoulder | Increase Camera Speed |

*Table 11 – Engine Keyboard and Xbox 360 Controller Input Mappings*