Efficient Tile-Based Deferred Shading Pipeline

BY

Denis Ishmukhametov
Bachelor of Science, Computer Science
Ufa State Aviation Technical University, June 2011

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the graduate studies program
of DigiPen Institute Of Technology
Redmond, Washington
United States of America

Spring
2013

Thesis Advisor: Dr. Gary Herron

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDY PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS OF DENIS ISHMUKHAMETOV HAS BEEN SUCCESSFULLY COMPLETED ON _____

TITLE OF THESIS: EFFICIENT TILE-BASED DEFERRED SHADING PIPELINE

MAJOR FILED OF STUDY: COMPUTER SCIENCE.

COMMITTEE:

Gary Herron, Chair	Xin Li
Pushpak Karnick	Antonie Boerkoel

APPROVED :

date	date
Graduate Program Director	Associate Dean
date	date
Department of Computer Science	Dean

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

INSTITUTE OF DIGIPEN INSTITUTE OF TECHNOLOGY

PROGRAM OF MASTER'S DEGREE

THESIS APPROVAL

DATE: _____DATE_____

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE THESIS PREPARED BY

Denis Ishmukhametov

ENTITLED

Efficient Tile-Based Deferred Shading Pipeline

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE FROM THE PROGRAM OF MASTER'S DEGREE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

_____

Thesis Advisory Committee Chair

_____

Director of Graduate Study Program

_____

Dean of Faculty

Contents

Abstract

One of the main goals of computer graphics has always been the ability to generate realistic images. An essential element of a realistic image is illumination. The real-life situations that we are trying to simulate may contain multiple light sources, sometimes even thousands and more e.g. a view of a city in the night; lots of street lamps, windows, car headlights etc. Although, non real-time CG usually cares more about the quality rather than time, in real-time rendering, especially in games, frame rendering time is strictly limited to 60 or 30 frames per second (16 or 33 ms). Thus rendering an image with multiple lights becomes a serious and challenging task.

This thesis discusses Deferred Shading – a group of algorithms that allow efficient dynamic rendering of multiple lights. We examine the existing algorithms and their optimizations and we also present a new improved version of Tile-Based Deferred Shading that shows excellent visual results and competitive performance.

Chapter 1 – Introduction

Rendering multiple dynamic lights is challenging problem of real-time computer graphics. Recently deferred shading algorithms became very popular. Deferred shading algorithms solve the problem of multiple dynamic lights by decoupling rendering of objects and their lighting. However, before we dive into a large variety of deferred shading algorithms and their complexities, let's consider how lighting was handled before deferred shading.

1.1 Forward Rendering

Using conventional forward rendering to render a set of lit objects we will pick each object in the scene in no particular order and calculate its surface color based on its material and all the lights that affect that object. The pseudo code for that will look like this:

```
For each object
      For each light
            Color += Lighting (object, light)
```

Shader implementation for that approach will have a loop in the pixel shader to iterate over all light sources, compute and sum the contribution of each light source. This will give us the complexity O(N * L), where N is the number of all scene objects' surface pixels and L is the number of lights.

Forward Shading issues:

1. *High shading complexity O (N * L)*

   Shading performance is dependent on the number of objects' surface pixels. This happens because objects that share same pixels on the screen will be shaded. Therefore, performance could be lost due to unnecessary shading of screen pixels that will be replaced, because of depth test.
2. *Shader combination problem*

   Since lighting calculation is tightly coupled with geometry type being rendered, it becomes problematic to handle all types of geometry (static,

skinned) and all types of lights (directional, point, spot) in a shader program.

1.2 Single Pass and Multi-Pass Rendering

There are two ways to solve the second problem of forward rendering: single pass and multi pass rendering. In single pass we will create a large shader (Über shader) that handles any geometry-material-light combination. This could be done by using dynamic branching in a shader of by using pre-processor defines to compile different versions of the shader.

In multi pass rendering we will create a small shader for each geometry-material-light combination and then render each object multiple times using additive blending. The pseudo code for multi-pass rendering will look like this:

```
For each light
      For each object
            Framebuffer += Lighting (object, light)
```

The problem with multi-pass rendering is that repetitive vertex transformations are performed for each light source.

1.3 Deferred Shading

History

Deferred shading was first introduced in a hardware design in 1988 [Deering88], with a more generalized method using G-buffers (Geometry buffers) in 1990 [Saito90]. Multiple Render Targets (MRT) feature was introduced in OpenGL 2.0 and Direct3D 9. That feature allowed the programmable rendering pipeline to render images to multiple render target textures at once. Thus, Deferred Shading [Hargreaves04] in its modern form became possible on existing hardware.

Idea

The idea of deferred shading is to decouple lighting calculation and geometry rendering, making it simpler to manage large number of lights. Instead of calculating lighting immediately for each geometry surface pixel and writing it to the frame buffer, we extract any geometry data that we want to use for shading, store it into multiple screen-space buffers (G-buffer) using MRT. Then we

traverse each screen-space pixel and shade it based on the extracted properties in screen-space buffers and the scene light sources. This gives us a complexity O(N + L), where N is the number of screen pixels and L is the number of lights. The pseudo code for deferred shading will look like this:

```
For each object
      Render to multiple render targets
For each light
      Apply light as a 2D postprocess
```

Deferred Shading solves both main problems of Forward Rendering. Deferred Shading advantages are the following:

1. *Low Shading Complexity O(N + L)*
   In Deferred Shading geometry and lighting are decoupled. As a result each geometry triangle is rendered once and each visible geometry surface pixel is shaded only once. Many small lights are just as cheap as a few big ones.
2. *Easy to add new light sources*
   Since geometry and lighting are decoupled, lights do not depend on geometry and it is easy to add various post process shaders for different light types (point, spot, directional etc.)
3. *Easy to add post processing effects*
   It is easy to add various post processing effects such as motion blur, heat haze etc. to deferred shading pipeline, because geometry data is already available in G-buffer.

Deferred Shading has following disadvantages:

1. *Transparency is not supported*
   G-buffer stores information only about the closest geometry. Thus, transparent object cannot be rendered using Deferred Shading.
2. *No hardware support for MSAA (Multi Sample Anti-Aliasing)*
   Hardware assisted anti-aliasing cannot be performed in screen-space after all lighting is calculated. It can only be used when geometry is rendered to G-buffer, which will produce incorrect results.

3. *Multiple-materials problem*

   There is a difficulty using multiple materials. Since lighting is applied as a post process using only data stored in G-buffer, G-buffer needs to store more information if many various materials required.
4. *High Memory Usage and Bandwidth*

   G-buffer that consists of multiple buffers can easily decrease performance when it is being generated and passed to lighting stage and used in other post-processing shaders.
5. *Old hardware do not support MRT*

   There are some old GPUs, consoles that do not support MRT feature.

Deferred Shading algorithm

To implement deferred shading algorithm we will need the following:

1. Create several render targets. The number and size of render targets depends on our needs: specifically it depends on various effects or lighting features that we are planning to implement. The obvious render target layout goes something like:

Data	Format
Position	A32B32G32R32F
Normal	A16B16G16R16F
Diffuse color	A8R8G8B8
Material parameters	A8R8G8B8

2. Ability to render full screen quad. To do that we just create 4 vertices, where each vertex has screen corner coordinates in NDC space: (-1, 1), (-1, -1), (1, 1), (1, -1). And later render it with a desired shader.
3. Low-poly meshes as convex hulls for lights: sphere mesh for point lights, cone mesh for spot lights.

Deferred shading (version of 2004) algorithm is the following:

1. **Setup stage**. Clear all render targets that are used as MRT and depth buffer.

2. **G-buffer creation.** Set MRTs for rendering. Render our geometry; extract per-pixel geometry properties such as Normal, Position in world space, color from diffuse textures, other material parameters (if needed) and write to corresponding render targets.
3. **Lighting stage.** Set back buffer for rendering and enable additive blending. Render ambient light as full screen quad. If we have any directional or lights render them as a full screen quad. For each pixel read position, normal, and diffuse color from G-buffer and calculate lighting, output lit pixel to back buffer. If we have any point or spot lights render them as sphere or cone and calculate lighting per pixel in the same way as for directional light, but with attenuation and other special features depending on the light type. This is done so that pixel processor is only fired inside the screen space convex light volume projection.

By using additive blending for lighting we perform the following operation on lights:

$$Pixel\ color = A \times D_m + \sum_{i=0}^{n} (Att_i \times (N \cdot L_i) \times D_m \times D_i + BRDF \times S_i \times S_m)$$

Where:

A is ambient color

N is surface's normal,

L_i is light direction of i-th light,

D_i is diffuse light color of i-th light,

S_i is specular light color of i-th light,

S_m is material specular color,

D_m is material diffuse color

Att_i is distance attenuation from i-th light,

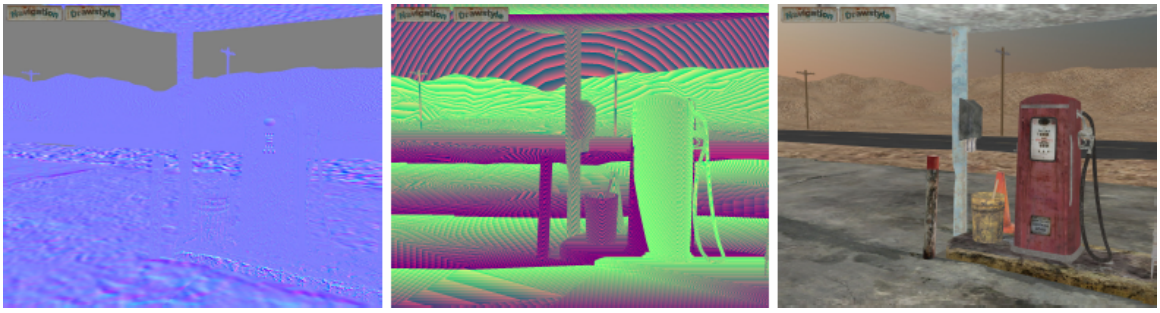
n is the number of lights

Since material diffuse color is constant we can change that calculation to:

$$Pixel\ color\ =$$

$$= D_m \times \left(A + \sum_{i=0}^{n} Att_i \times (N \cdot L_i) \times D_i \right) + \sum_{i=0}^{n} Att_i \times BRDF \times S_i \times S_m$$

Using that fact we can slightly change lighting calculation which will make is faster. Instead of multiplying by the material diffuse color for every light we will first calculate the sum of all light contributions and then multiply by diffuse color. We will need additional render target – **Light accumulation buffer**. Step 3 of previously described algorithm will change to the following two steps:



G-buffer (left to right): Normals, Position, Diffuse color **[Hargreaves04]**

3. **Light accumulation stage.** Set light accumulation buffer as active render target. Set blend state to additive blending. Render all lights the same way as previously described in step 3, but output diffuse light and specular light to the render target separately. It can be either diffuse light to RGB, specular intensity (no specular color) to alpha channel or diffuse to RGB and specular color and intensity to the second render target if we want to use colored specular.

4. **Composition.** Set back buffer as render target. Draw full screen quad. For each pixel combine material diffuse from G-buffer and light diffuse and specular colors:

framebuffer = diffuse * G-buff.diffuse + specular

Chapter 2: Survey of current Deferred Shading algorithms

In this chapter we will examine in detail the existing variations of Deferred Shading. The algorithms that we will discuss can be categorized in the following way:

1. Deferred Lighting (3 pass algorithms)
    a. Light Pre-Pass [Engel08]
    b. Prelighting [Lee08]
    c. Inferred Lighting [Kircher09] [Kircher12]
2. Tile-Based Deferred Shading
    a. PlayStation 3 Implementation [Balestra08] [Swoboda09] [Coffin11]
    b. Implementation with Compute Shader [Andersson09] [Andersson11] [Lauritzen10] [Olsson11]
    c. Clustered Deferred and Forward Shading [Olsson12]
3. “Forward” shading
    a. Light Indexed Deferred Lighting [Trebilco08] [Pettineo12]
    b. Tile-Based Forward Shading
        i. Implementation with Compute Shader [Harada12] [McKee12]
        ii. DX11 implementation with UAVs [Lewis12]

The first group of algorithms, Deferred Lighting, uses 3 passes: G-buffer pass, lighting pass and material pass. In the material pass the scene is rendered for the second time.

Tile-Based Deferred Shading algorithms use screen space tiles or spatial tiles (clusters) for light culling and as a result saving lighting computations.

The algorithms from the third category render the geometry twice and they only store depth in the G-buffer. They combine features of deferred and forward shading.

2.1 Light Pre-Pass/ Pre-Lighting

Some of the main problems of Deferred Shading are G-buffer size and material problem. Light Pre-Pass (also known as Deferred Lighting) [Engel08] tries to solve those problems. This algorithm was presented as an internet article by Wolfgang Engel in 2008. Light Pre-Pass has been widely used in commercial video games such as Crysis 2, StarCraft 2, Blur etc. There is even an implementation on IPhone [Yeung12].

There are 3 main steps in the algorithm:

1. **Geometry pass:** Render geometry and write normals and depth values to the corresponding buffers.
2. **Lighting pass:** Calculate lighting using normals and depth values, store lighting information in light buffer. This pass is similar to lighting pass in the Deferred Shading. Lights are rendered as volumes. The only difference is that there is no information about specular intensity and diffuse color. The following data is stored in the light buffer:
   First 3 channels store

$$\sum_{i=0}^{n} (N \cdot L_i) \times D_i \times Att_i$$

   for 3 RGB components of the diffuse light color. Alpha channel stores luminance of the sum of all specular contributions

$$\sum_{i=0}^{n} lum\ \left((N \cdot L_i) \times (N \cdot H_i)^n \times Att_i\right)$$

3. **Second geometry pass:** Render geometry for the second time, apply different material terms and calculate final color using the light buffer.

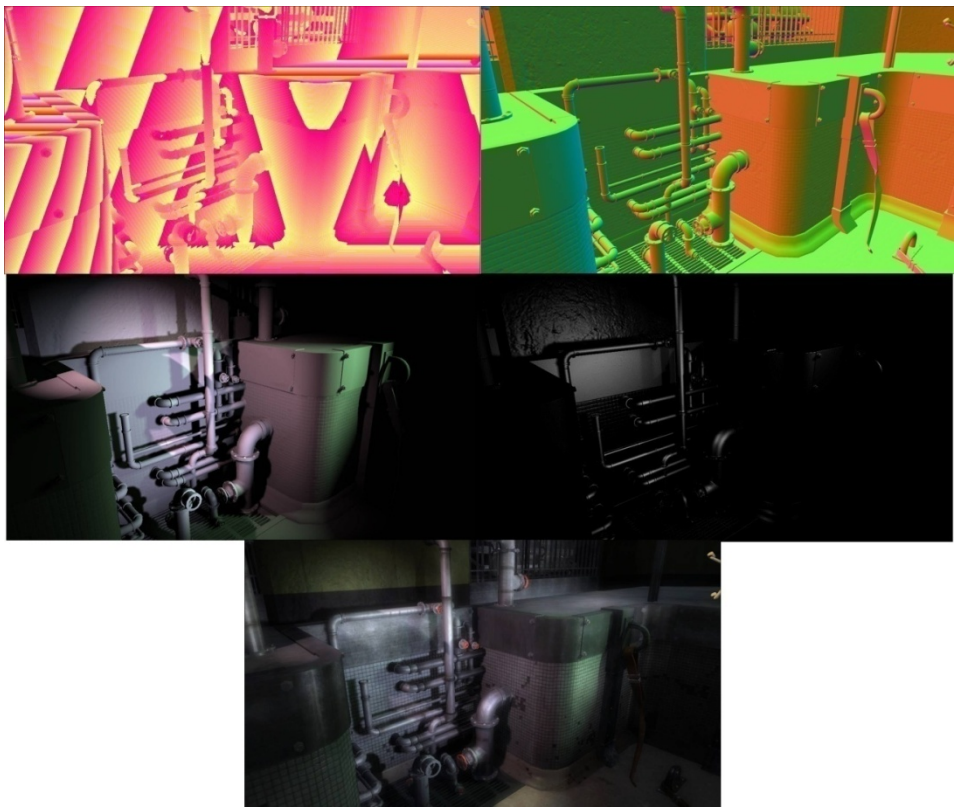
Light Pre-Pass has some advantages over Deferred Shading:

1. Less memory and bandwidth than Deferred Shading
2. Works better with multiple materials
3. Works better with hardware MSAA in DirectX 9
4. Can be implemented without MRTs

The disadvantages are the following:

1. Scene needs to be rendered 2 times (redundant vertex transformations)
2. Specular contributions are blended (implementations are usually limited to monochromatic specular)
3. Forward rendering required for translucent objects

There are some variations of light pre-pass algorithm. One of the variations is called Pre-Lighting [Lee08]. It was used in a video game called Resistance 2. The main difference of the algorithm is that it uses 2 buffers for diffuse and specular light so that color from specular term is correctly accumulated. Pre-Lighting also uses screen-space quads for rendering lights instead of spheres.

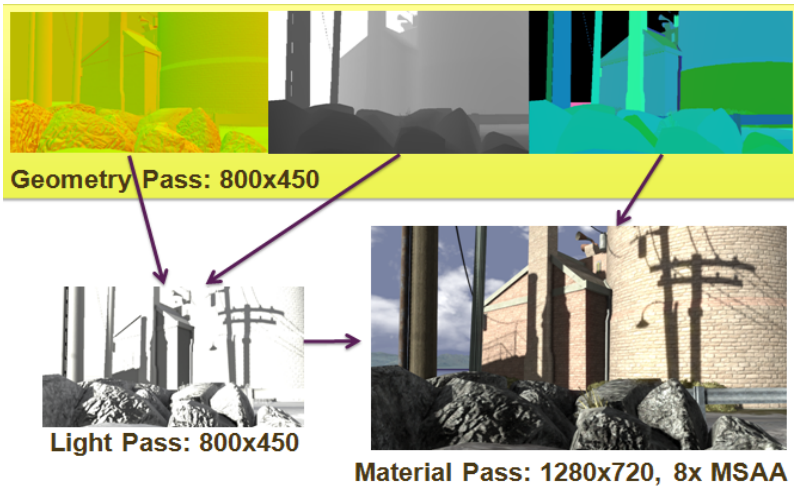


Pre-Lighting in Resistance 2: Depth buffer (Top Left), Normal buffer (Top Right), Diffuse light buffer (Middle Left), Specular Light Buffer (Middle Right), Final result (Bottom) [Lee08]

2.2 Inferred Lighting

Inferred Lighting is an algorithm similar to Light Pre-Pass. However, it was developed independently at Volition, Inc. in 2009 by Scott Kircher and Alan Lawrence. This technique was used in several released commercial video games such as Red Faction: Armageddon and Saints Row: The third.

Inferred Lighting is trying to solve some problems of traditional Deferred Shading: transparency, material variety and MSAA anti-aliasing. In traditional Deferred Shading transparency is usually handled in a separate forward pass, while Inferred Lighting offers a unified pipeline for handling opaque and transparent objects. The main features of the technique are mixed resolution rendering and Discontinuity Sensitive Filtering.



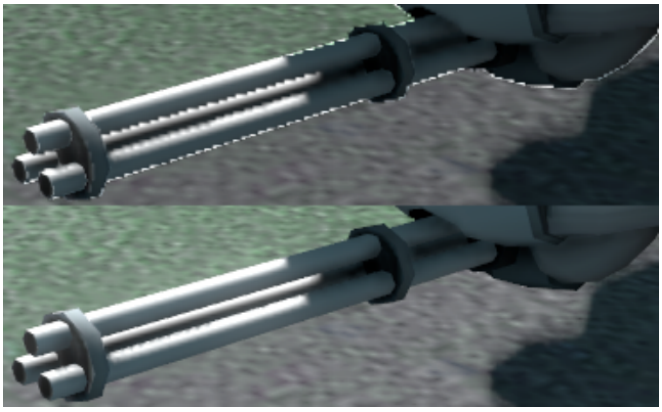
Normals (Top Left), Depth (Top Center), DSF data (Top Right), L-buffer (Bottom Left), Final output (Bottom Right) [Kircher09]

The algorithm consists of 3 steps:

1. **Geometry pass.** In the first pass geometry of the scene is rendered to G-buffer. G-buffer uses lower resolution than frame buffer. The minimal G-buffer information required for Inferred Lighting is normals, depth and discontinuity sensitive filter data (DSF). Normals stored as two 16 bit values, DSF data is stored as two 16 bit values, linear depth and ID value.
2. **Light pass.** Contribution of ambient light and dynamic lights is calculated in screen space and stored in L-buffer (4x16 bit channels, diffuse lighting in RGB channels and specular light is stored as accumulated intensity in the alpha channel). L-buffer has the same low resolution as G-buffer.
3. **Material pass.** Scene geometry is rendered again with material shaders that read data from L-buffer. Transparent objects are sorted and rendered after opaque objects and lit in the same manner. Material pass is rendered

at frame buffer resolution, which requires to up-sample the L-buffer and perform special filtering – Discontinuity Sensitive Filtering.

Discontinuity sensitive filtering is performed in the pixel shader in material pass. The goal is to get rid of light bleeding artifacts that occur due to lighting being calculated at lower resolution. In geometry pass 16 bit of DSF data is being stored. It consists of 8 bits for Object ID and 8 bits for normal-group ID. Using that data in the material pass DSF samples 4 pixels from L-buffer and compares their depth and ID with currently rendered object. It applies custom bilinear filtering and discards samples that do not belong to the current surface.



Bad lighting aliasing is noticeable if DSF is not used (Top), DSF solves this problem (Bottom) [Kircher09]

DSF enables a way to light transparent surfaces. Transparent surfaces are rendered during geometry pass using a stipple pattern so that their G-buffer samples are interleaved with opaque polygon samples. The light pass will automatically light those stippled pixels. No special case processing during the light pass is necessary, as long as the lighting operations are one to one (i.e., do not involve blurring of the L-buffer). In the material pass the DSF for opaque polygons will automatically reject stippled transparent pixels, and transparent polygons are handled by finding the four closest L-buffer samples in the same stipple pattern, again using DSF to make sure the samples were not overwritten by some other geometry.

Interleaved transparent and opaque samples (Left), Lit result (Right) [Kircher09]

Inferred Lighting has the following advantages over deferred shading:

1. Greater material flexibility than deferred shading.
2. Compatible with MSAA.
3. Unified pipeline for processing transparent objects.
4. Reduced memory bandwidth and pixel shading cost.

There are some disadvantages too:

1. Transparent objects are being lit at even lower resolution.
2. Only 3 layers of transparency are supported.
3. Normal maps quality is low since some information about normals is lost due to low resolution G-buffer.
4. Up-sampling can be costly.

2.3 Light Indexed Deferred Lighting

Light Indexed Deferred Lighting was introduced by Damian Trebilco in 2008. The idea of the algorithm is to assign each light a unique index and store this index at each pixel the light hits. These indices can then be used in a pixel shader to lookup the light properties from the global light table to perform lighting. The algorithm consists of 3 steps:

1. **Z pre-pass.** Render depth only.
2. **Light volumes rendering.** Render light volumes into a light index texture with depth writing disabled.

3. **Geometry rendering.** Render geometry using standard forward rendering. Perform lighting using the light index texture to access lighting properties in pixel shader.

There is a problem with multiple lights overlap. If no lights overlap then step two can simply write the light index to the texture and it can be directly accessed in step three. The paper proposes 3 light index packing schemes for multiple overlapping lights. All methods assume that light indices are 8 bit and light index texture is RGBA8.

1. **CPU sorting.**
    a. On the CPU, create four arrays to hold light volume data. Then for each scene light, find the light data array it can be added to without intersecting any of the existing lights in the array. (eg. Attempt to add to array one, then attempt to add to array two etc.) If a light cannot be added, it will have to be discarded or stored to be processed in a second pass.
    b. Clear light index color buffer to zero.
    c. Enable writing to the Red channel only and render light volumes from light data array one.
    d. Enable writing to the Green channel only and render light volumes from light data array two.
    e. Enable writing to the Blue channel only and render light volumes from light data array three.
    f. Enable writing to the Alpha channel only and render light volumes from light data array four.
2. **GPU multi-pass max blend equation.**
    a. Clear color and stencil buffers to zero.
    b. Set blend equation mode to MAX
    c. Mask out writes to Blue and Alpha channels

d. Set stencil to increment on stencil pass and set the stencil compare value to only pass on values < 2. (only allow a max of two writes per fragment)
e. Render the light volumes and output (index, 1.0-index) in the red and green channels.
f. Mask out writes to Red and Green channels and enable Blue and Alpha channels
g. Set stencil to decrement on stencil failure and set the stencil compare value to only pass on values equal to 0.
h. Render the light volumes and output (index, 1.0-index) in the blue and alpha channels.

3. **GPU bit shifting.** This method requires video card with bit logic support (Shader Model 4)
    a. Clear the color buffer to zero.
    b. Set the blend mode to ONE, CONSTANT_COLOR where the constant color is set to 0.25.
    c. This shifts existing color bits down two places (>> 2 = * 0.25) and adds the two new bits to the top of the number.
    d. Render the light volumes and break the 8bit index value into four 2bit values and output each 2 bit value into RGBA channels as high bits. eg. Red channel = (index & 0x3) << 6.
    e. This index splitting can be done offline and simply supplied as an output color to the light volume pass.

Light Indexed Deferred Lighting has following advantages:

1. Efficient middle ground between forward and deferred
2. Forward renderers can be easily modified to use this technique
3. No problems with multiple materials

The disadvantages are the following:

1. Total lights number and total number of overlapping lights are limited

2. Difficult to add shadows
3. Multiple light types are problematic

The idea of the described algorithm is interesting, but the strict limitation on the number of overlapping lights may outweigh all the advantages.

2.4 Tile-Based Shading

The main idea of Tile-Based Shading (also known as Screen-space tile classification) group of algorithms is to divide the screen up into 2D tiles and determine how many and which light sources intersect each tile. Then calculate lighting only for visible light sources for each pixel in each tile.

2.4.1 Tile-Based Deferred Shading

Tile-Based Deferred Shading replaces the lighting stage of traditional Deferred Shading with light culling stage, where light sources are culled against screen space tile frusta. There are several implementations which use the same idea, but implemented on different hardware.

PlayStaion 3 implementation

The PS3 uses the Cell microprocessor, which is made up of one 3.2 GHz PowerPC-based "Power Processing Unit" (PPU) and 6 accessible Synergistic Processing Units (SPUs). On top of that there is a NVIDIA G70 Graphics Processing Unit. Since SPUs are designed for 128 bit SIMD vector operations that make them suitable for processing of various rendering tasks. SPU and GPU can execute in parallel.

Tile-Based Deferred Shading first appeared on PlayStation 3 hardware because it allowed efficient usage of hardware parallelism. It was first mentioned in 2008 at GDC presentation entitled “The technology of Uncharted: Drake Fortune”[Balestra08]. There is not

From “The Technology of Uncharted: Drake’s Fortune” [Balestra08]

much detail about their particular implementation available. However, according to their presentation the algorithm they used is the following:

1. Render opaque dynamically lit geometry: world normal + specular exponent in screen space
2. Divide the screen into a grid
3. Find which lights intersect each cell
4. Render quads over each cell calculating up to 8 lights per pass: store results in a light accumulation buffer

Another PS3 implementation was presented in 2009 in Matt Swoboda from PhyreEngine team. Their rendering pipeline heavily used the advantages of SPUs. The high-level algorithm that they presented is the following:

1. Calculate affecting lights per tile
    a. Build a frustum around that tile using min and max depth values in that tile
    b. Perform frustum check with each light's bounding volume
    c. Compare light direction with tile average normal value
2. Choose fast paths based on tile contents
    a. If no lights affect the tile use fast path
    b. Check material values to see if any pixels are marked as lit
3. Choose whether to process MSAA per tile

Implementation with compute shaders

Compute shaders thread groups can operate in screen-space tiles similar to SPUs on PlayStation 3. Therefore, ideas of algorithms from PS3 can be applied on PC. Tile-based shading implementation with compute shaders was first presented at SIGGRAPH in 2009 by Johan Andersson in his talk "Parallel Graphics in

Per-tile visible light count (black = 0 lights, white = 40) [Andersson09]

Frostbite – Current & Future". Later different implementations appeared: by Andrew Lauritzen in 2010 [Lauritzen10] and Ola Olsson in 2011 [Olsson11].

Johan Andersson proposed a new hybrid graphics/compute shading pipeline:

1. Graphics pipeline rasterizes G-buffer for opaque surfaces
2. Compute pipeline uses G-buffer, culls light sources, computes lighting and combines with shading

Input data for compute shader stage is G-buffer, global list of lights. Output is fully composited and lit HDR texture. Each thread processes each pixel. Thread groups (tiles) are 16x16 pixels.

Compute shader algorithm:

1. Load G-buffer
2. Calculate min and max Z in thread group (tile)
    a. Use InterlockedMin / Max. Since atomics work only on integers, cast float to int. Z is either always positive or negative depending if left-handed or right handed system is used.
3. Determine visible light source for each tile. Output for each tile is the number of visible light sources and index list of visible light sources.

	Lights	Indices
Global list	1000+	0 1 2 3 4 5 6 7 8 ..
Tile visible list	~0-40+	0 2 5 6 8 ..

Example input and output [Andersson09]

    a. Each thread switches to process light sources instead of pixels. 256 light sources in parallel per tile. Multiple iterations if number of lights is greater than 256.
    b. Intersect light source and tile using tile frustum with min and max Z values
    c. For visible lights append light index using atomic InterlockedAdd
    d. Synchronize group and switch back to processing pixels
4. For each pixel, accumulate lighting from visible light sources in a for loop. Read from tile visible light index list in thread group shared memory.

5. Combine lighting and material albedos

Advantages of tile-based deferred shading:

1. Constant and absolute minimal bandwidth. G-buffer data is read once per pixel.
2. No need for intermediate light accumulation buffer
3. Scales up to huge amount of big overlapping light sources

Disadvantages of tile-based deferred shading:

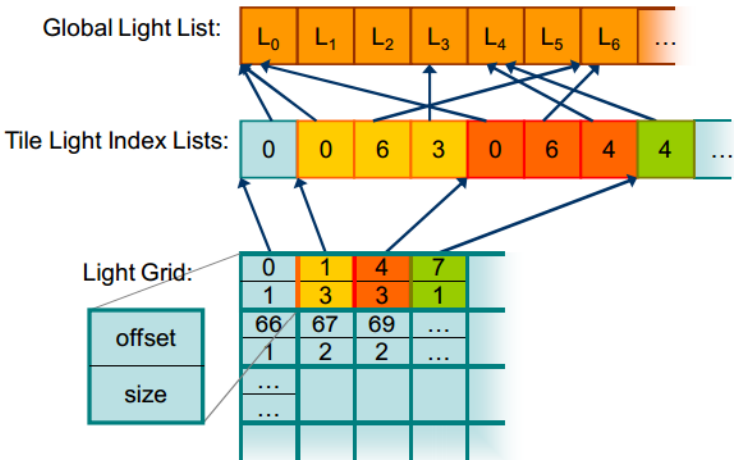
1. DirectX 11 hardware is required (Compute Shader 5.0)
2. Same problems with MSAA as in traditional deferred shading
3. Same problems with transparent objects

2.4.2 Tile-Based Forward Shading

Tiled shading is not limited to deferred shading. Recently tile-based forward shading implementations have appeared, “Tile Shading” [Olsson11], “Forward+” [Harada12] [McKee12], “DirectX 11 implementation with UAVs” [Lewis12]. These implementations are similar to Light Indexed Deferred Shading.

Implementation with compute shaders

The idea of light culling is very similar to tile-based deferred shading. Light sources are being culled against screen space tile frusta. But instead of performing shading right after culling visible light lists are stored and then later used in pixel shader for lighting when geometry is



Grid data structure [Olsson11]

rendered. Proper data structures should be used to save information about visible light for every tile. Ola Olsson proposes the method of using 3 arrays:

1. **Global Light List** contains light properties.
2. **The light index list** contains light indices to the global light list
3. **Light grid** contains an offset and size of the light list for each tile

Advantages of tile-based forward shading:

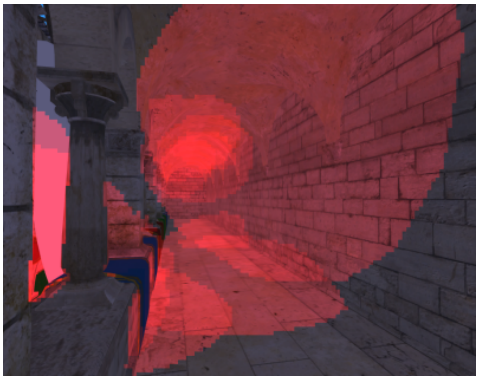
1. Light management is decoupled from geometry
2. Transparent objects can be shaded the same way as opaque
3. No problems with multiple materials and lighting models

Disadvantages of tile-based forward shading:

1. Each pixel may be shaded more than once
2. Problems with shadows
3. Scales worse with increasing light overdraw than tiled deferred shading

DirectX 11 implementation with UAVs

This implementation was developed independently by Peter J. B. Lewis and presented on his website in the article entitled “Tile-Based Forward Rendering”. The algorithm developed by Peter Lewis is somewhat similar to “Light Indexed Deferred Lighting” [Treblico08]. In his original implementation he did not use Compute Shader, but instead he used UAVs. UAV stands for Unordered Access View (buffer, texture or texture array). It is a new resource type in DirectX 11 that allows temporally unordered read/write access from multiple threads. This means that this resource type can be read / written simultaneously by multiple threads without generating memory conflicts through



Tiles being affected by lights [Lewis12]

the use of Atomic Functions. For example, UAVs can be used in pixel shader or compute shader. The presented algorithm is the following:

1. **Depth pre pass.** Depth is rendered to a texture. Then it is down-sampled in additional pass so that there is one depth value per tile. Tiles are 8x8 pixels in screen space. Shader finds the maximum depth of all the pixels in each tile and writes it to a smaller depth buffer.
2. **Building the Per-Tile Linked Lists.** Bounding volumes of light sources are rendered here the same way as in traditional deferred shading. Instead of shading each light sources ID is added to a linked list for each tile. Down-sampled buffer from previous stage is used to quickly reject pixels that have no lights affecting them. The linked list generation is based on AMD's Order Independent Transparency presentation [Hensley10]. Two Unordered Access Views are used: one stores the linked list elements (the LinkBuffer), and the other stores the offset into the LinkBuffer that marks the start of the list for that tile (the HeadBuffer).

```
struct LightLink
{
    uint LightID;
    uint Next;
};
```

3. **Rendering the scene.** Scene is rendered using the LinkBuffer, HeadBuffer and buffer containing all light sources data.

```
struct PointLight
{
    float3 Position;
    float Radius;
    float3 Colour;
};
```

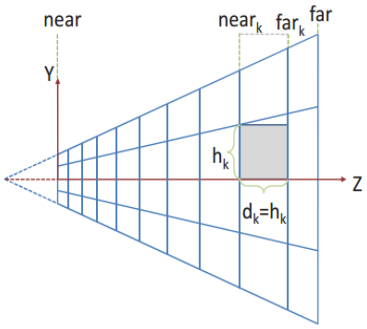
Lighting calculation is performed in pixel shader by looping through the linked list of lights that affect the tile current pixel is in.

```
uint next = HeadBuffer[head_index];
while (next != 0xFFFFFFFF)
{
    LightLink link = LinkBuffer[next];
    PointLight pointLight = PointLightBuffer[link.LightID];
```

```
    lighting += Shade(worldPos, worldNormal, eyeVec, specPower,
pointLight);
    next = link.Next;
}
```

2.5 Clustered Deferred and Forward Shading

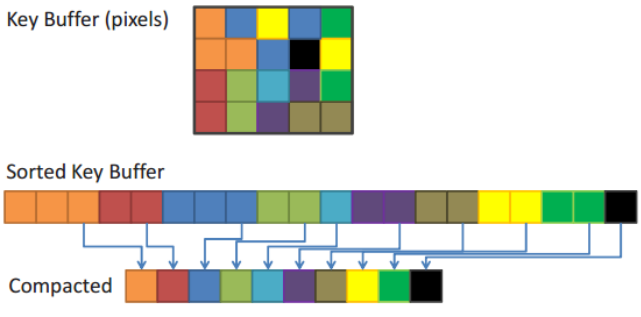
The latest available paper on tile-based shading is "Clustered Deferred and Forward Shading" by Ola Olsson, Markus Billeter, and Ulf Assarsson. In Clustered Shading, view samples with similar properties (e.g. 3D-position and/or normal) are grouped into clusters. This is comparable to tiled shading, where view samples are grouped into tiles based on 2D-position only. According to the paper clustered Shading enables real-time scenes with two to three orders of magnitudes more lights than previously feasible (up to around one million light sources).



Exponential spacing in view space [Olsson12]

The Clustered Deferred Shading algorithm has the following steps:

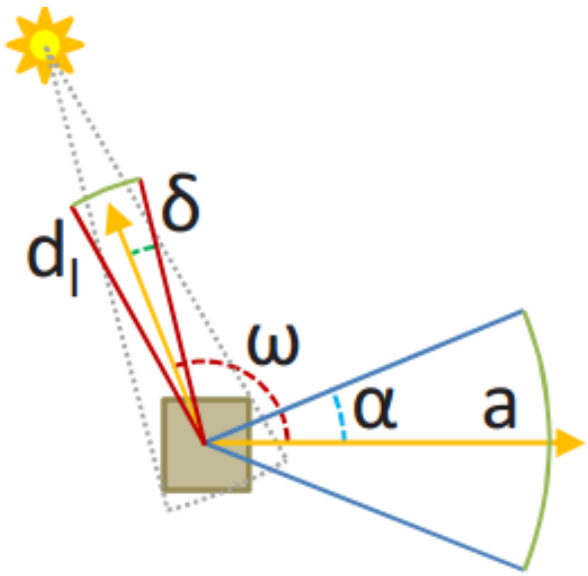
1. **Render scene to G-buffer.** This is performed in the same way as in traditional deferred shading.
2. **Cluster assignment.** The goal of the cluster assignment is to compute an integer cluster key for a given view sample from the information available in the G-Buffer. Position and, optionally, the normal are used. Subdivisions are performed in view space, by spacing the divisions exponentially to achieve self-similar subdivisions. Cluster key tuple (i, j, k) is computed from screen-space coordinates and the view-space depth.
3. **Find unique clusters.** The cluster keys in the key buffer are sorted and then compacted, to find the list of unique clusters. The sorting is, for instance, based on the



Sorting and compacting the key buffer to find unique clusters [Olsson12]

view sample's depth and normal direction. The paper presents two methods for finding unique clusters: local sorting and page tables (similar to page table approach used by virtual textures .

4. **Assign lights to clusters.** Each frame, a bounding volume hierarchy (BVH)of lights is constructed by first sorting the lights according to the Z-order (Morton Code) based on the discretized center position of each light. For each cluster, BVH is traversed using depth-first traversal. At each level, the bounding box of the cluster (either explicitly computed from the cluster's contents or implicitly derived from the cluster's key) is tested against the bounding volumes of the child nodes. For the leaf nodes, the sphere bounding the light source is used; other nodes store an AABB enclosing the node. If a normal cone is available for a cluster, it is used to further reject lights that will not affect any samples in the cluster.

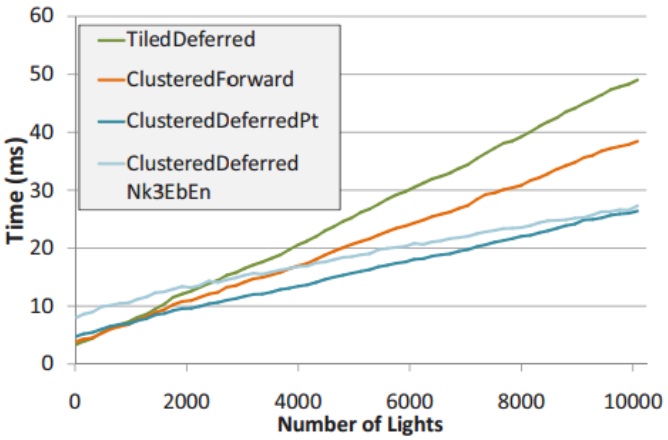


Back-face culling of lights against clusters. If the angle between the incoming light and the axis of the normal cone ω is greater than $\frac{\pi}{2}+\alpha+\delta$, the light faces the back of all samples in the cluster, and can therefore be ignored. [Olsson12]

5. **Shade samples.** Shading is different from Tile-based deferred shading. In the sorting approach, index into the list of uniqe clusters is explicitly stored

for each pixel. This is achieved by tracking references back to the originating pixel, and, when the unique cluster list is established, storing the index to the correct pixel in a full screen buffer. When using page tables, after the unique clusters are found, we store the cluster index back to the physical memory location used to store the cluster key earlier (using the same page table as before). This means that a virtual lookup for the cluster key will yield the cluster index. Thus, each
sample can look up the cluster index using the cluster key computed earlier (or re-computed).

Accroding to the paper the algorithm can support up to 1 million light sources, but there is an overhead of assigning clusters and finding unique clusters that make algorithm slower and ineffective for smaller number of lights. In one of the graphs presented in the paper we can see that Tiled Deferred Shading (green) is much faster than any variant of Clustered Shading when number of lights is smaller than around 2000.



Algorithm time comparison graph [Olsson12]

Chapter 3: Deferred Shading optimization techniques

Deferred Shading has been used in commercial video games since 2005. Multiple developers have implemented some form of deferred shading and came

up with various optimization techniques. There are two main categories of optimizations:

1. Bandwidth optimizations. Bandwidth cost is reduced by reducing the size of the G-buffer
2. Light volume rendering optimizations.

First category is the most important one since multiple MRTs is one of the most significant bottlenecks of the Deferred Shading. We will examine G-buffer layouts and optimizations used in released commercial video games.

3.1 S.T.A.L.K.E.R (2005)

A video game S.T.A.L.K.E.R. that was released in 2005 used 3 render targets for their G-buffer. It was the first game to use Deferred Shading. [Shishkovtsov05]

	Data	Format
RT0	3D Position + Material ID	RGBA16F
RT1	Normal + Ambient Occlusion	RGBA16F
RT2	Color + Gloss (Specular Exponent)	RGBA8

Additional parameters stored in G-buffer were material ID and ambient occlusion (AO). Ambient occlusion was precomputed and saved into special textures for each surface. During G-buffer creation these textures were sampled and AO values were saved for later use in the lighting stage. Material ID is widely used in Deferred Renderers to overcome the material variety problem. When geometry is rendered into G-buffer material ID is assigned for every object or surface. This ID is later used in the lighting stage to lookup and apply specific lighting model for the given pixel.

This particular implementation of the G-buffer is one of the earliest one. It uses 160 bits per pixel, which may not seem like a lot of memory, but. for example, for the screen resolution 1024x768 it results in 15 MB of bandwidth every frame.

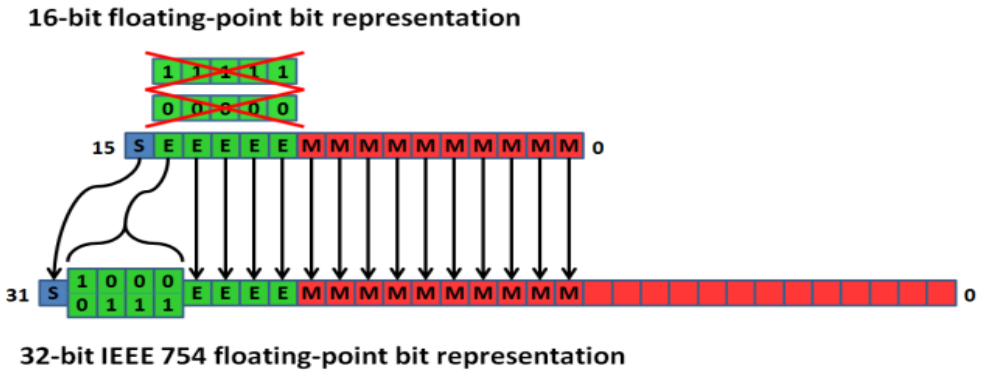
3.2 S.T.A.L.K.E.R: Clear Sky (2009)

A video game S.T.A.L.K.E.R: Clear Sky was released 4 years after the release of the original S.T.A.L.K.E.R. It featured greatly improved Deferred Shading pipeline [Lobanchikov09]. The major improvement is that developers managed to squeeze all data into 2 render targets:

	Data	Format
RT0	Normal + Depth + Material ID + Ambient Occlusion	RGBA16F
RT1	Color + Gloss (Specular exponent)	RGBA8

New G-buffer is using 96 bits per pixel vs 160 bits per pixel before. Some packing math is involved that increases the total number of arithmetic operations, but the number of texture operations and bandwidth are reduced. Main optimizations used here are the following:

1. 3D position is stored as linear depth in one 16 bit float. Position is reconstructed from depth in the pixel shader. (See next subsection for details)
2. Normal vector is stored as 2 values, 16 bit float each. Simple packing scheme is used where X and Y are stored and Z is reconstructed. (See next subsection for details)
3. Shader bitwise operations are used to pack Material ID and Ambient Occlusion in one 16 bit float channel. The packing algorithm consists of following steps:
    a. Pack data into a 32bit uint as a bit pattern that is a valid 16bit fp number
    b. Cast the uint to float using asfloat()
    c. Cast back for unpacking using asuint()
    d. Extract bits



Bit correspondence between 32 bit and 16 bit IEEE float [Lobanchikov09]

3.3 Killzone 2 (2007)

Killzone 2 is a video game developed exclusively for PlayStation 3 and released in 2007. The game features Deferred Shading [Valient07] and uses G-buffer with four RGBA8 render targets that contain lots of additional parameters:

R8	G8	B8	A8
Depth 24bpp			Stencil
Lighting Accumulation RGB			Intensity
Normal X (FP16)		Normal Y (FP16)	
Motion Vectors XY		Spec-Power	Spec-Intensity
Diffuse Albedo RGB			Sun-Occlusion

World position is reconstructed from 24 bit depth. Normal vector is stored as X and Y (See next subsection for details). Additional data that is stored in the G-buffer includes:

1. **Motion Vectors.** They are used later in the pipeline for screen space per-pixel motion blur post process.
2. **Sun-Occlusion.** This is basically static sun shadows that were precomputed in offline renderer and saved into textures.

This G-buffer layout uses 128 bits per pixel.

3.4 Battlefield 3 (2011)

Battlefield 3 is a video game built with an engine called Frostbite 2, which uses Deferred Shading and also advanced features such as real-time global illumination [Coffin11][Andersson11]. G-buffer that is used in Frostbite 2 consists of four 32 bit render targets and one depth buffer with 24 bit and 8 bit stencil:

	R8	G8	B8	A8
GB0	Normal .xyz			Spec. Smoothness
GB1	Diffuse albedo .rgb			Specular albedo
GB2	Sky visibility	Custom envmap ID	Material Param.	Material ID
GB3	Irradiance (dynamic radiosity)			

G-buffer uses 128 bits per pixel. Some interesting pieces of data that are stored are the following: Sky visibility, environment map ID, material ID and other material parameters. One of the render targets is completely reserved for Irradiance which is generated by dynamic radiosity - real-time global illumination algorithm.

3.5 Light Volume Rendering optimizations

This type of optimization is used to speed up the lighting stage of the Deferred Shading. In Deferred Shading lights are rendered as light volumes that correspond to the light range: full screen quad for directional light, sphere for point light, cone for spot light etc.

The following optimizations are used to improve the rendering efficiency of lights:

1. When the camera is outside of the light volume front faces of the light volumes should be rendered. When it is outside – back faces are rendered.
2. Geometry instancing can be used to render all light volumes of one light type in one draw call. This feature is only available in DirectX 10/11 level hardware.

3. As an alternative to rendering light volumes, camera facing quads can be rendered for each light. Quad screen coordinates need to cover the extents of the light volume. Rendered geometry is simpler, as a result less vertex shader operations.
4. Texture read minimization. When G-buffer is fetched point-sampling should be used in pixel shader to avoid the cost of unnecessary bilinear filtering, because there is one to one correspondence between pixels in G-buffer and the back-buffer if same resolution is used.
5. Blending cost minimization. Additively blending lights into the light accumulation buffer is not free. If light contribution is zero for the given pixel that pixel shader output should be discarded and not written into the render target.

3.6 Reconstruction of Position From Depth

There are 2 ways to reconstruct position from depth: using inverse transformation and using ray from the camera and view frustum.

Inverse transformation approach

If depth is non-linear and stored as z/w (e.g. in hardware depth buffer) we can use inverse transformation matrices to get world or view space position. The following algorithm reconstructs position from depth:

1. Convert UV texture coordinates in the range [0,1] to ND coordinates [-1,1]

```
float x = texCoord.x * 2 - 1;
float y = (1 - texCoord.y) * 2 - 1;
```

2. Read depth from depth buffer texture

```
float z = tex2D(DepthSampler, texCoord);
```

3. Transform by inverse projection matrix to get position in view space or by inverse view projection matrix to get position in world space

```
float4 projectedPos = float4(x, y, z, 1.0f);
float4 positionVS = mul(projectedPos, InvProjection);
```

4. Divide by w to get final view or world space position

```
positionVS = positionVS.xyz / positionVS.w;
```

Ray and View Frustum approach

If hardware depth is not available (e.g. in DirectX 9) and depth is stored as linear normalized z value we can reconstruct position by scaling linear depth by the ray pointing from the camera to the far plane. To get the position in world space we need to perform the following steps;

1. In the vertex shader of the quad, calculate the direction vector from the camera position to the vertex (view ray).
2. In the pixel shader, normalize the view ray vector
3. Sample the depth texture to get the distance from the camera to the G-Buffer surface
4. Multiply the sampled distance with the view ray
5. Add the camera position

3.6 Compact normal storage in G-buffer

There are multiple ways to store normals compactly in G-buffer. Majority of approaches try to use some compression algorithm to store normals as 2 values instead of 3 to save memory and space in G-buffer. This section is based on Aras Pranckevičius's article "Compact Normal Storage for Small G-Buffers".

X&Y, Z reconstruction

If a normal is normalized (i.e. unit length) then we can store X and Y and the reconstruct Z value from X and Y. This method was used in Killzone 2 [Valient07]. The encoding for the normal is very simple. Normal's X and Y are mapped from [-1, 1] range to [0, 1]

```
float2 normalOut = normalIn.xy * 0.5 + 0.5;
```

And decoding is the following:

```
float3 normal
normal.xy = normalOut.xy * 2 – 1;
normal.z = sqrt(1 - normalOut.x*normalOut.x - normalOut.y*normalOut.y);
```

This method has a big disadvantage. If the normal is pointing away from the camera then Z will be negative, but the sign information will be lost.

Spherical coordinates (based on Wolfgang Engel's blog)

It is possible to use spherical coordinates to encode the normal. Since we know its unit length, we can just store the two angles. This method works with normals both in view space and world space. But there are a lot of arithmetic operations involved for encoding and decoding.

Encoding:

```
float2 atanYX = atan2(normalIn.y,normalIn.x);
float2 normalOut = float2(atanYX / PI, normalIn.z);
normalOut = (normalOut + 1.0) + 0.5;
```

Decoding:

```
float2 angles = normalOut * 2.0 - 1.0;
float2 theta;

sincos( angles.x * PI, theta.x, theta.y );

float2 phi = float2( sqrt( 1.0 - angles.y * angles.y ), angles.y );

flaot3 normal = float3( theta.y * phi.x, theta.x * phi.x, phi.y );
```

Sphere map transform

Spherical environment mapping (indirectly) maps reflection vector to a texture coordinate in [0,1] range. The reflection vector can point away from the camera, just like our view space normals. [Mittring09] This method was used in CryEngine 3. The encoding and decoding is very cheap on arithmetic operations.

Encoding:

```
normalOut = normalize(normalIn.xy) * sqrt(normalIn.z * 0.5 + 0.5);
```

Decoding:

```
normal.z = length2(normalOut.xy) * 2 - 1;
normal.xy = normalize(normalOut.xy) * sqrt(1 - normal.z * normal.z);
```

Stereographic Projection

Stereographic projection maps a unit sphere to circle of infinite size. We can use it and the rescale it so that “practically visible” range of normals maps into unit circle. Scaling factor depends on field of view and on the desired precision for normals that point away from the camera.

Encoding:

```
float scale = 1.7777;
float2 normalOut = normalIn.xy / (normalIn.z + 1);
normalOut /= scale;
normalOut = normalOut * 0.5 + 0.5;
```

Decoding:

```
float scale = 1.7777;
float3 a = normalOut.xyz * float3(2 * scale, 2 * scale, 0)
a += float3(-scale, -scale, 1);
float b = 2.0 / dot(a.xyz, a.xyz);

float3 normal;
normal.xy = b * a.xy;
normal.z = b - 1;
```

Per-pixel View Space

If we compute view space per-pixel, then Z component of a normal can never be negative. Then just store X&Y, and compute Z.

Per-pixel view matrix creation:

```
float3 x,y,z;
z = -viewVector;
x = normalize (float3(z.z, 0, -z.x));
y = cross (z,x);
float3x3 viewMatrixPerPixel = float3x3 (x,y,z);
```

Encoding:

```
float2 normalOut = mul(viewMatrixPerPixel, normalIn) * 0.5 + 0.5;
```

Decoding:

```
float3 normal;
normal.xy = normalOut * 2 - 1;
normal.z = sqrt(1 + dot(normal.xy,-normal.xy));
normal = mul(normal, viewMatrixPerPixel );
```

Best fit for normals

Best fit for normals is a method to store a normal as 3 values with 8 bits per channel and utilize all available values in 8 bit per channel representation. [Kaplanyan10]. If we store normalized normals as 3 values in 8 bit per channel render targets we get various banding artifacts in the final shaded image. The main mistake is that we store normalized normals. 24 bit range effectively gives us 3D grid of 256x256x256 = 16777216 values. But by normalizing normals we reduce the possible range to the surface of the unit sphere in that 3D grid. As a result we use only around 2% of all available values of 24 bit space.

To utilize all values best-fit method for normals was proposed. For the given normal direction we calculate quantization error for each cell the normal intersects. That effectively gives us the error if we store the normal in that cell. However, this task is too computationally expensive for real-time thus it was proposed to prebake the results of the search into a huge cubemap of directions. Each texel of the cubemap stores the distance to the best cell for the

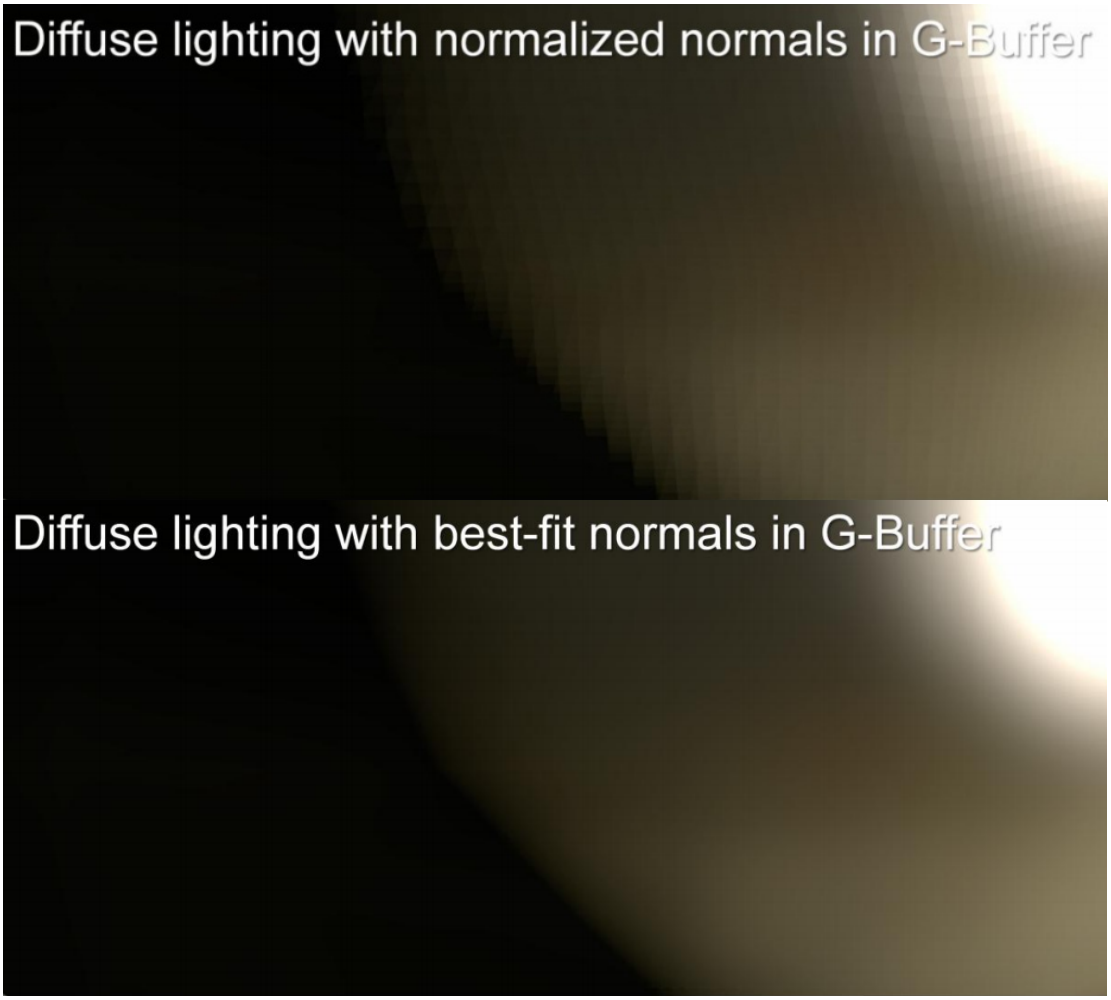
normal with this direction. Since the cubemap has a lot of symmetry inside it is possible to save only one 2D side as 512x512 texture.

The algorithm of outputting the normal into the G-Buffer is the following:

1. Prepare the texture coordinates for 2D lookup and look up the distance to the best cell from the precomputed 2D texture
2. Scale the normalized normal by this value in order to fit it into the precomputed best cell
3. Output the scaled normal into the G-Buffer

The reconstruction of the normal is just reading the value from G-buffer and normalization.

Best-fit for normals allows to utilize 98.2% possible values of 24 bit render target.



Best-fit for normals [Kaplanyan10]

Chapter 4: Original contribution

In the previous chapters we have examined all existing variations of deferred shading and optimization techniques. The most interesting ones are two recent approaches: Tile-Based Deferred Rendering and Clustered Deferred Rendering. Both methods use latest hardware features such us compute shader and unordered access view. Tile-Based Deferred Rendering performs fast and efficient light culling. There are multiple reasons why this algorithm is faster than traditional Deferred Shading:

1. Light volumes are not drawn anymore, no vertex shader computation needed
2. G-buffer is accessed once for every pixel for all lights instead of accessing it for every light for every pixel
3. Additional light accumulation pass is not needed

Clustered Deferred Rendering allows using even more lights than Tile-Based Deferred Rendering (up to 1 million lights), but it performs worse on smaller number of lights (several thousand), because cluster assignment and unique cluster search is rather expensive and not justified for smaller number of lights. Considering the fact that a typical videogame scene usually does not require million of lights Tile-Based Deferred Rendering is more appealing. However, both methods present interesting ideas for improving efficiency of light culling. So we can combine the advantages of both methods to get an intermediate method that will work efficiently on average number of lights sources. Tile-Based technique uses screen-space tiles as clusters, this is faster and more efficient than spatial clusters in clustered method. Clustered technique on the other hand uses spatial hierarchical data structures for lights and it takes into account normals for clusters and that allows culling more lights and, as a result, performing fewer number of lighting calculations.

If we take Tile-Based Deferred Rendering with compute shader as a basis there are 2 options to improve it:

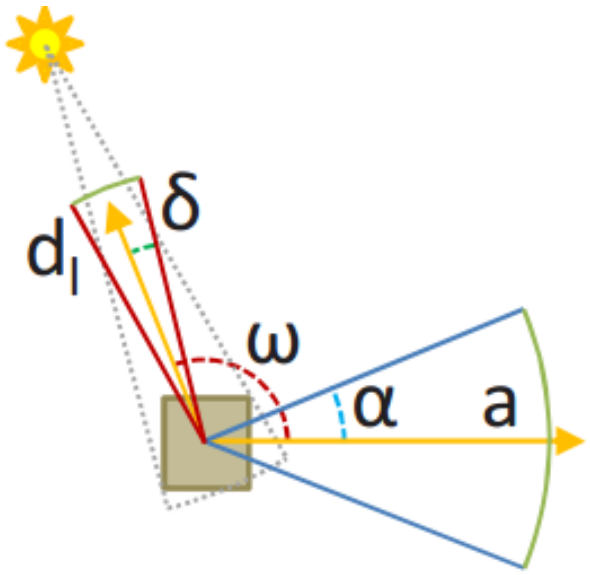
1. Use some hierarchical spatial data structure for lights
2. Take into account normals for tiles (Back-face culling for lights)

We will consider the second option. Clustered Deferred Shading mentions that normals were taken into account to perform back face culling for lights. But specific details of the technique were not described.

The idea of the algorithm on the high level is that for lights that intersect the tile frustum, we can reject any light whose direction is opposite all normals in the tile. By rejecting these lights we avoid lighting computations for all pixels in that tile,

because their contribution will be zero anyway. For example, if we have a 16x16 pixel tile and we reject a light we will avoid lighting calculation for 256 pixels and that's where the performance increase comes from. We will gain more performance improvement compared to the original Tile-Based Deferred Shading algorithm as the number of back-face culled lights increase. On the other hand we will not get any improvement if no lights are culled.

Based on the illustration presented in the paper we have designed the algorithm to perform back face culling for lights. To perform back-face culling we need to construct a normal cone and a light cone for tile.



Back-face culling of lights against clusters. If the angle between the incoming light and the axis of the normal cone ω is greater than $\frac{\pi}{2} + \alpha + \delta$, the light faces the back of all samples in the cluster, and can therefore be ignored. [Olsson12]

Our algorithm is the following:

1. If the light source's bounding volume intersects with the tile frustum, perform back face culling with normals

2. For each tile, calculate the central normal direction for the tile normal cone by adding all normals in the tile and normalizing the resulting average normal.
3. Find the angular extents of the tile normal cone by performing dot products between the central normal direction and all normals in the tile. The final maximal normal cone angle α will be the arccosine of the minimal dot product.
4. Calculate 8 corners of the tile frustum in view space.
5. For each light calculate light cone. The central direction of the light cone will be the vector between center of the tile and the light source position in view space.
6. Calculate dot products between central light cone direction and 8 vectors between light and frustum corners. Find the maximal angle δ, which will be the arccosine of the minimal dot product.
7. Find the angle ω by calculating dot product between the given light direction and central normal direction of the tile. Reject the given light source if the angle ω is greater than $\frac{\pi}{2} + \alpha + \delta$

We will measure the efficiency of the improved Tile-Based Deferred Lighting algorithm by comparing the frame time with traditional Deferred Shading, original Tile-Based Deferred Lighting and comparing the visual accuracy (e.g. absence of visual artifacts).

4.1 Implementation details

A demonstration application was implemented to test the new improved algorithm. Three algorithms were implemented and one major optimization technique:

1. Traditional Deferred Shading
2. Tile-Based Deferred Rendering
3. Tile-Based Deferred Rendering with back-face culling (new method)

4. View frustum culling for lights (CPU)

All work was done using DirectX 11 API. ATI Radeon HD5730 GPU was used for development. All images were rendered at 1280x720 HD resolution.

Deferred Shading

Traditional Deferred Shading implementation is based on [Hargreaves04]. Three passes are implemented: G-buffer pass, light pass and combine pass. The G-buffer structure is the following:

	Data	Format
RT0	Albedo color + Specular Intensity	RGBA8
RT1	Normal + Specular exponent	RGBA16F
	Depth Stencil (Hardware)	D24S8

Two render targets were used + hardware depth buffer. Multiple earlier mentioned optimizations were used to speed up the algorithm: depth was used instead of 3d position, light volumes have low number of triangles, instancing was used for light volume rendering, only back faces of light volumes are rendered, if light contribution is zero for a given pixel, that sample is discarded to save blending operation.

One directional sun light was used and multiple small point lights. Shadowing technique from the directional light was also implemented.

Tile-Based Deferred Rendering

Tile-Based Deferred Rendering with compute shaders was implemented. Implementation was based on [Lauritzen10] and [Andersson09]. 16x16 pixel tiles were used.The most important parts of the implementation performance wise are the creation of the frustum per tile and the sphere-frustum intersection.

The following code was used for the creation of the frustum per-tile:

```
float2 tileScale = textureSize.xy * float(1 / (2 * BLOCKSIZE));
float2 tileBias  = tileScale - float2(groupId.xy);
```

```
float4 c1 = float4(Projection._11 * tileScale.x, 0.0f, -tileBias.x,
0.0f);
float4 c2 = float4(0.0f, -Projection._22 * tileScale.y, -tileBias.y,
0.0f);
float4 c4 = float4(0.0f, 0.0f, -1.0f, 0.0f);

Tile currentTile;
currentTile.frustumPlanes[0]= c4 - c1; // right
currentTile.frustumPlanes[1]= c4 + c1; // left
currentTile.frustumPlanes[2]= c4 - c2;
currentTile.frustumPlanes[3]= c4 + c2;
currentTile.frustumPlanes[4]= float4(0.0f, 0.0f, 1.0f, maxGroupDepth);
// far
currentTile.frustumPlanes[5]= float4(0.0f, 0.0f, -1.0f,-minGroupDepth);
// near
```

Sphere frustum intersection function:

```
bool Intersects(PointLight light, Tile tile)
{
      bool inFrustum = true;
      [unroll] for (uint i = 0; i < 6; ++i) {
            float d = dot(tile.frustumPlanes[i],
            float4(light.positionView, 1.0f));
            inFrustum = inFrustum && (d >= -light.radius);
      }
      return inFrustum;
}
```

Tile-Based Deferred Rendering with back-face culling (new method)

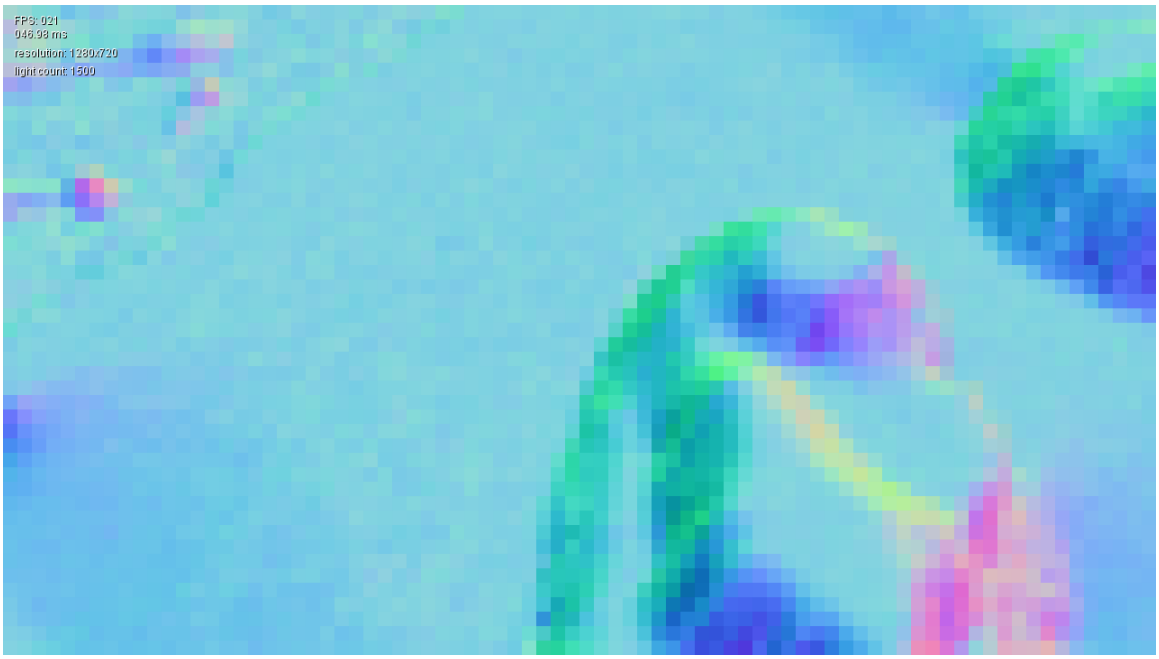
For the back-face culling for normals the algorithm described earlier was used. The most computationally intensive step of the algorithm is the first one, where central normal for the tile has to be calculated. First attempt to do that was to use atomic functions. Atomics in compute shader 5.0 work only on integers so normals were quantized and converted to integer, then added and converted back to float.

```
int3 normalInt = int3(normal * 256.0f); // [-1, 1] to [-256, 256]

GroupMemoryBarrierWithGroupSync();
InterlockedAdd(centerNormalInt.x, normalInt.x);
InterlockedAdd(centerNormalInt.y, normalInt.y);
InterlockedAdd(centerNormalInt.z, normalInt.z);
GroupMemoryBarrierWithGroupSync();

float3 centerNormal = float3(centerNormalInt) * INV_THREAD_COUNT; // [-
256x, 256x] to [-x, x]
centerNormal = normalize(centerNormal); // [-x, x] to [-1, 1]
```

These calculations turned out to be slow, because of multiple synchronization barriers. An alternative solution was found. Averaging the normals for a particular tile is the same operation as calculating mip level for the normal render target. If our thread group size is 16x16 pixels that means that need log16 mip level, which is 4. Mip level needs to be read with point sampling to avoid unnecessary interpolation of normals between tiles.

```
float3 centerNormal = normalize(2.0f *
NormalTexture.SampleLevel(SamplerPointClamp, texCoord, 4).xyz - 1.0f);
```

This method is significantly faster, since hardware functions are used generate mip levels. Moreover, generated mip levels for the normal buffer can be reused in various post-effects.



Central normals for 16x16 pixel tiles.

View frustum culling for lights (CPU)

View frustum culling for lights was implemented on the CPU side to reject the lights that are outside of the screen. Camera view frustum consisting of 6 planes was constructed. Each point light source in the scene is tested against the camera view frustum every frame using sphere-frustum intersection calculation.

A corresponding boolean flag is set for every light. Only lights that have that flag set to true are rendered in case of the traditional deferred shading or sent to the compute shader in case of tile-based deferred shading.

4.2 Results

For real-time interactive applications the most important measurement is frame time in milliseconds or the number of frames per second. We used both metrics to compare the efficiency of traditional Deferred Shading, original Tile-Based Deferred Rendering and new improved tile-based deferred rendering algorithm. Another metric that was used is the visual correctness of the algorithms: correct lighting and absence of visual artifacts.

For the comparison of the three algorithms, tests were run on a large scene with multiple lights, while measuring the frame-rate. Four tests were carefully constructed varying the number of lights cullable by our algorithm from 0 to 100, 500, and 1000 respectively. For each of the tests, the three algorithms were each run twice, once with CPU frustum culling enabled, and once without, and the resulting six measurements are displayed in a table, one per test.

Test 1

A total number of lights in the scene is 1500. Enabling View Frustum Culling (CPU) leaves only 55 lights. No lights are culled by Tile-Based Deferred Shading with back-face culling algorithm.

	View Frustum Culling (CPU) Enabled	View Frustum Culling (CPU) Disabled
	Time (ms)	**Time (ms)**
Traditional Deferred	29.35	33.07
Tile-Based	24.90	29.45
Tile-Based with back-face culling	26.23	31.48

Test 2

A total number of lights in the scene is 1600. Enabling View Frustum Culling (CPU) leaves only 183 lights. 100 lights are culled by Tile-Based Deferred Shading with back-face culling algorithm.

	View Frustum Culling (CPU) Enabled	View Frustum Culling (CPU) Disabled
	Time (ms)	**Time (ms)**
Traditional Deferred	66.56	68.49
Tile-Based	41.67	44.25
Tile-Based with back-face culling	35.54	38.57

Test 3

A total number of lights in the scene is 2000. Enabling View Frustum Culling (CPU) leaves only 583 lights.500 lights are culled by Tile-Based Deferred Shading with back-face culling algorithm.

	View Frustum Culling (CPU) Enabled	View Frustum Culling (CPU) Disabled
	Time (ms)	**Time (ms)**
Traditional Deferred	209.95	211.98
Tile-Based	96.56	99.11
Tile-Based with back-face culling	64.45	67.35

Test 4

A total number of lights in the scene is 2500. Enabling View Frustum Culling (CPU) leaves only 1083 lights.1000 lights are culled by Tile-Based Deferred Shading with back-face culling algorithm.

	View Frustum Culling (CPU) Enabled	View Frustum Culling (CPU) Disabled
	Time (ms)	**Time (ms)**
Traditional Deferred	390.49	391.47
Tile-Based	165.79	168.58
Tile-Based with back-face culling	99.98	103.05

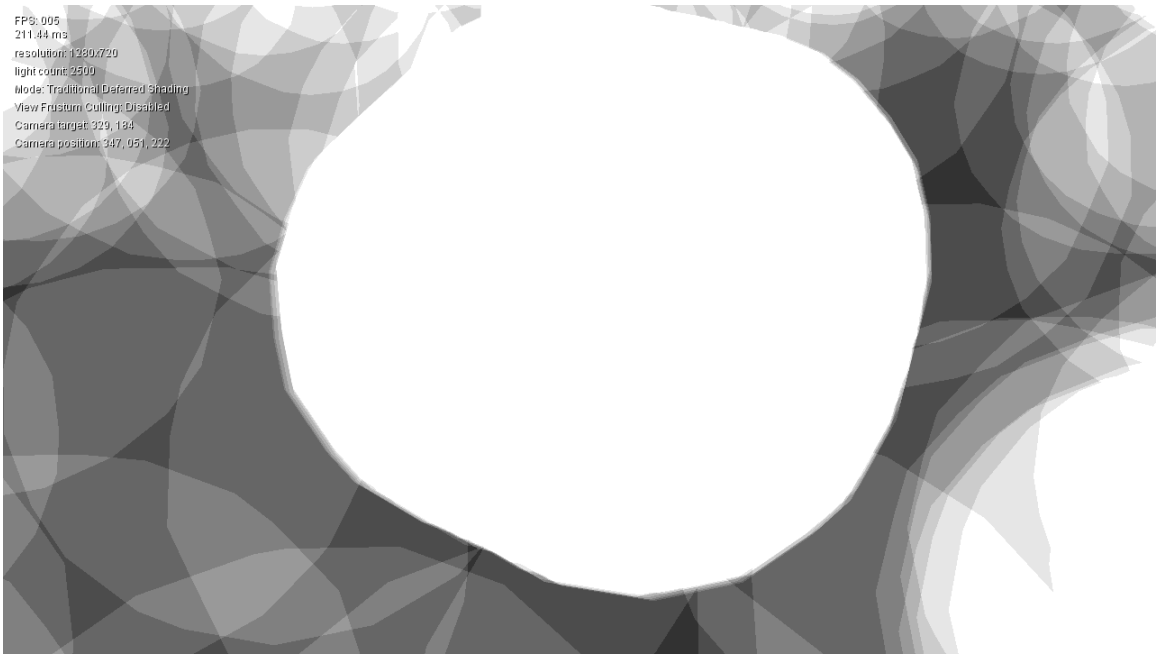
As we can see from the measurements traditional Deferred Shading is the slowest technique. Tile-Based Deferred Shading performs significantly better than Deferred Shading. And our new technique is faster than just Tile-Based Deferred Shading in cases were multiple lights are culled. Additionally, it is scaled well when the number of culled lights increase. According to the measured results we see that we get 15% time improvement over Tile-Based Deferred Shading when 100 lights are culled, 33% when 500 lights are culled and 40% when 1000 lights are culled. However, the back-face culling computation is still not very cheap. Thus there is no speed improvement when no lights are culled.

We can also see that all methods gain a slight speed improvement from using View-Frustum Culling on the CPU side.

Final fully lit result of a rendered scene using traditional Deferred Shading.

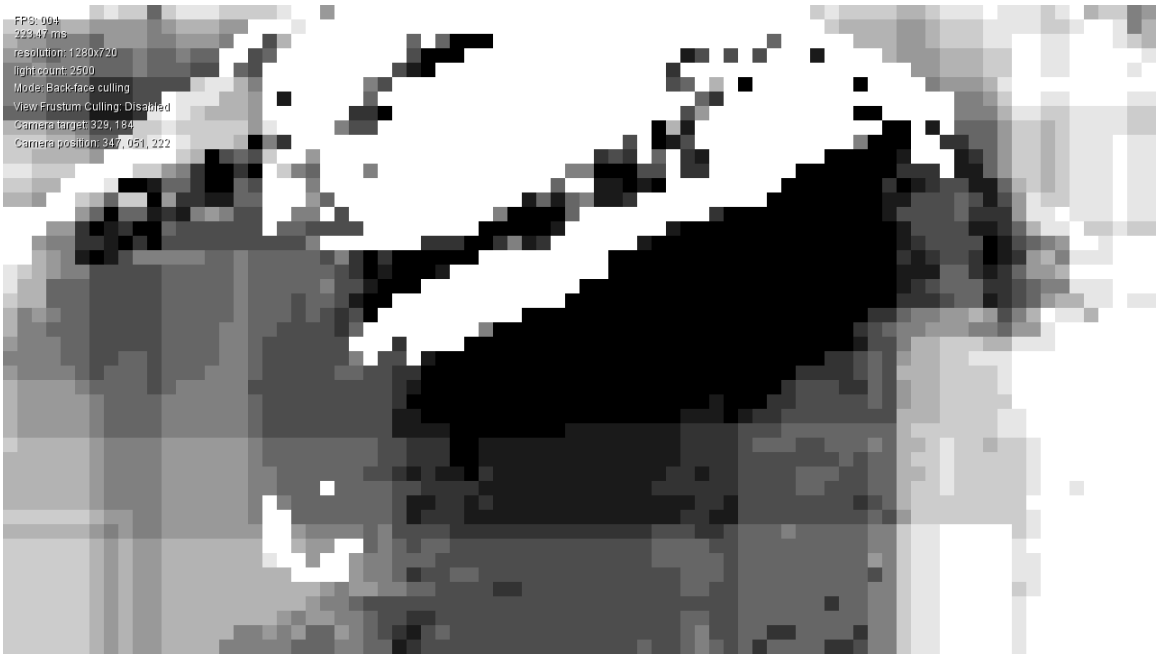
Final fully lit result of a rendered scene using Tile-Based Deferred Shading.

Final fully lit result of a rendered scene using Tile-Based Deferred Shading with back-face culling.



Number of lights for traditional Deferred Shading (white = more than 10 lights, black= 0 lights)

Number of lights for Tile-Based Deferred Shading (white = more than 10 lights, black= 0 lights)



Number of lights for Tile-Based Deferred Shading with back-face culling (white = more than 10 lights, black= 0 lights)

Visual comparison images show that there is no visual difference in all tested algorithms. In the light number images we can see that large amount of lights

were rejected in the back-face culling approach compared to just tile-based approach and that saved a lot of lighting calculations.

Conclusion

This thesis examined optimizations and the possibility of improvement for the Deferred Shading algorithm. It was shown that Tile-Based Deferred Shading is more efficient than traditional Deferred Shading. New back-face culling approach was proposed that combined the strengths of Tile-Based Deferred Shading and Clustered Deferred Shading. The new approach gives an advantage in scenes with large surfaces with normals pointing away from light sources. For example, if a scene has a wall and there are multiple lights sources behind that wall, the new algorithm will reject these light sources and as a result save many lighting calculations. However, in scenes where no lights are rejected we will not gain any speed improvement. So this algorithm should be used wisely. It is possible to use the back-face culling conditionally depending on the scene setup.

There are a lot of variations of deferred shading and different optimizations. It is still an ongoing research. So we will see better and faster algorithms in the future as new hardware features become available.

References

1. [Deering88] Michael Deering, et al. “The triangle processor and normal vector shader: a VLSI system for high performance graphics”, SIGGRAPH 1988 http://dl.acm.org/citation.cfm?id=378468
2. [Saito90] Takafumi Saito, Tokiichiro Takahashi, “Comprehensible rendering of 3-D shapes”, SIGGRAPH 1990 http://dl.acm.org/citation.cfm?id=97901
3. [Calver03] Dean Calver, “Photo-realistic Deferred Lighting” http://www.beyond3d.com/content/articles/19/
4. [Hargreaves04] Shawn Hargreaves, “Deferred Shading”, GDC 2004 http://www.shawnhargreaves.com/DeferredShading.pdf

5. [Hargreaves04] Shawn Hargreaves, Matt Harris, “Deferred Shading: 6800 Leagues Under The Sea” http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf
6. [Geldreich04] Rich Geldreich, Matt Pritchard, John Brooks, “Deferred Lighting and Shading”, GDC 2004 http://www.tenacioussoftware.com/pritchard_matt.ppt
7. [Thibieroz04] Nick Thibieroz, “Deferred Shading with Multiple-Render-Targets,” pp. 251 – 269, ShaderX2 – Shader Programming Tips & Tricks with DirectX9
8. [Policarpo05] Fabio Policarpo, Francisco Fonseca, “Deferred Shading Tutorial” http://www710.univ-lyon1.fr/~jciehl/Public/educ/GAMA/2007/Deferred_Shading_Tutorial_SBGAMES2005.pdf
9. [Shishkovtsov05] Oles Shishkovtsov (GSC Game World), “Deferred Shading in S.T.A.L.K.E.R.”, GPU Gems 2 Chapter 9 http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html
10. [Valient07] Michal Valient (Guerrilla), “Deferred Rendering in Killzone 2” http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf
11. [Lee08] Mark Lee (Insomniac Games), “Pre-lighting” http://www.insomniacgames.com/tech/articles/0209/files/prelighting.pdf
12. [Lee08] Mark Lee (Insomniac Games), “Pre-lighting in Resistance 2” http://www.insomniacgames.com/gdc09-resistance-2-prelighting/
13. [Engel09] Wolfgang Engel, “Light Pre-Pass -Deferred Lighting: Latest Development”, SIGGRAPH 2009 http://www.bungie.net/images/Inside/publications/siggraph/Engel/LightPrePass.ppt
14. [Lobanchikov09] Igor Lobanchikov, “GSC Game World‘s S.T.A.L.K.E.R : Clear Sky – a showcase for Direct3D 10.0/1”, GDC 2009 http://developer.amd.com/gpu_assets/01gdc09ad3ddstalkerclearsky210309.ppt

15. [Mittring09] Martin Mittring (Crytek), “A bit more deferred – CryEngine3”, Triangle Game Conference 2009 http://www.crytek.com/cryengine/presentations&page=2
16. [Tovey10] Steven Tovey, Stephen McAuley, “Parallelized Light Pre-Pass Rendering with the Cell Broadband Engine™“ in “GPU Pro: Advanced Rendering Techniques,” May 2010
17. [Kircher09] Scott Kircher (Volition), Alan Lawrance (Volition), “Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects”, SIGGRAPH 2009 http://www.volition-inc.com/publications/inferred-lighting-fast-dynamic-lighting-and-shadows-for-opaque-and-translucent-objects/
18. [Kircher12] Scott Kircher (Volition), “Lighting and Simplifying Saints Row: The Third”, GDC 2012 http://www.volition-inc.com/publications/lighting-and-simplifying-saints-row-the-third/
19. [Trebilco08] Damian Trebilco, “Light Indexed Deferred Lighting” http://code.google.com/p/lightindexed-deferredrender/
20. [Pettineo12] Matt Pettineo, “Light Indexed Deferred Rendering” http://mynameismjp.wordpress.com/2012/03/31/light-indexed-deferred-rendering/
21. [Balestra08] Christophe Balestra (Naughty Dog), Pal-Kristian Engstad (Naughty Dog), “The Technology of Uncharted: Drake’s Fortune”, GDC 2008 http://www.naughtydog.com/docs/Naughty-Dog-GDC08-UNCHARTED-Tech.pdf
22. [Swoboda09] Matt Swoboda (SCEE), “Deferred Lighting and Post Processing on PLAYSTATION®3” http://www.technology.scee.net/files/presentations/gdc2009/DeferredLightingandPostProcessingonPS3.ppt
23. [Coffin11] Christina Coffin (DICE) “SPU-based Deferred Shading for Battlefield 3 on Playstation 3”, GDC 2011 http://publications.dice.se/attachments/Christina_Coffin_Programming_SPU_Based_Deferred.pdf

24. [Andersson09] Johan Andersson (DICE), “Parallel Graphics in Frostbite – Current & Future”, SIGGRAPH 2009 http://s09.idav.ucdavis.edu/talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf
25. [Andersson11] Johan Andersson (DICE), “DirectX 11 Rendering in Battlefield 3”, GDC 2011 http://www.slideshare.net/DICEStudio/directx-11-rendering-in-battlefield-3
26. [Lauritzen10] Andrew Lauritzen (Intel), “Deferred Rendering for Current and Future Rendering Pipelines” http://visual-computing.intel-research.net/art/publications/deferred_rendering/
27. [Olsson11] Ola Olsson, Ulf Assarsson, “Tiled Shading” http://www.cse.chalmers.se/~olaolss/papers/tiled_shading_preprint.pdf
28. [Olsson12] Ola Olsson, Markus Billeter, Ulf Assarsson “Clustered Deferred and Forward Shading” http://www.cse.chalmers.se/~olaolss/papers/clustered_shading_preprint.pdf
29. [Harada12] Takahiro Harada (AMD), Jay McKee (AMD), Jason C.Yang (AMD), “Forward+: Bringing Deferred Lighting to the Next Level”, Eurographics 2012 https://sites.google.com/site/takahiroharada/
30. [McKee12] Jay McKee (AMD), “Technology Behind AMD’s Leo Demo”, GDC 2012 http://www.slideserve.com/ojal/technology-behind-amd-s-leo-demo-jay-mckee-mts-engineer-amd
31. [Lewis12] Peter J. B. Lewis, “Tile-Based Forward Rendering” http://www.pjblewis.com/articles/tile-based-forward-rendering/
32. [Liktor12] Gabor Liktor, Carsten Dachsbacher, “Decoupled Deferred Shading for Hardware Rasterization” http://cg.ibds.kit.edu/publications/p2012/shadingreuse/shadingreuse_preprint.pdf
33. [Segovia06] B. Segovia et al, ”Non-interleaved Deferred Shading of Interleaved Sample Patterns” http://liris.cnrs.fr/Documents/Liris-2476.pdf

34. [Yeung12] Simon Yeung, “Light Pre Pass Renderer on iPhone” http://www.altdevblogaday.com/2012/03/01/light-pre-pass-renderer-on-iphone/
35. [Kaplanyan10] Anton Kaplanyan, “CryEngine 3: Reaching the speed of light” http://advances.realtimerendering.com/s2010/Kaplanyan-CryEngine3(SIGGRAPH%202010%20Advanced%20RealTime%20Rendering%20Course).pdf
36. [Hensley10] Justin Hensley “Order-Independent Transparency in DirectX 11” https://graphics.stanford.edu/wikis/cs448s-10/FrontPage?action=AttachFile&do=get&target=CS448s-10-11-oit.pdf

Appendix A: Compute Shader - Overview

A compute shader is a programmable shader stage that expands Microsoft Direct3D 11 beyond graphics programming. The compute shader technology is also known as the DirectCompute technology. Like other programmable shaders (vertex and geometry shaders for example), a compute shader is designed and implemented with HLSL. A compute shader provides high-speed general purpose computing and takes advantage of the large numbers of parallel processors on the graphics processing unit (GPU). The compute shader provides memory sharing and thread synchronization features to allow more effective parallel programming methods.

A thread is a basic Compute Shader (CS) processing element. CS declares the number of threads to operate on (the “thread group”).

```
[numthreads(X, Y, Z)]
void CS(uint3 groupID:          SV_GroupID,
        uint3 groupThreadID:    SV_GroupThreadID,
        uint3 dispatchThreadID: SV_DispatchThreadID,
        uint  groupIndex:       SV_GroupIndex)
        {…}
```

To start CS execution the following command is called:

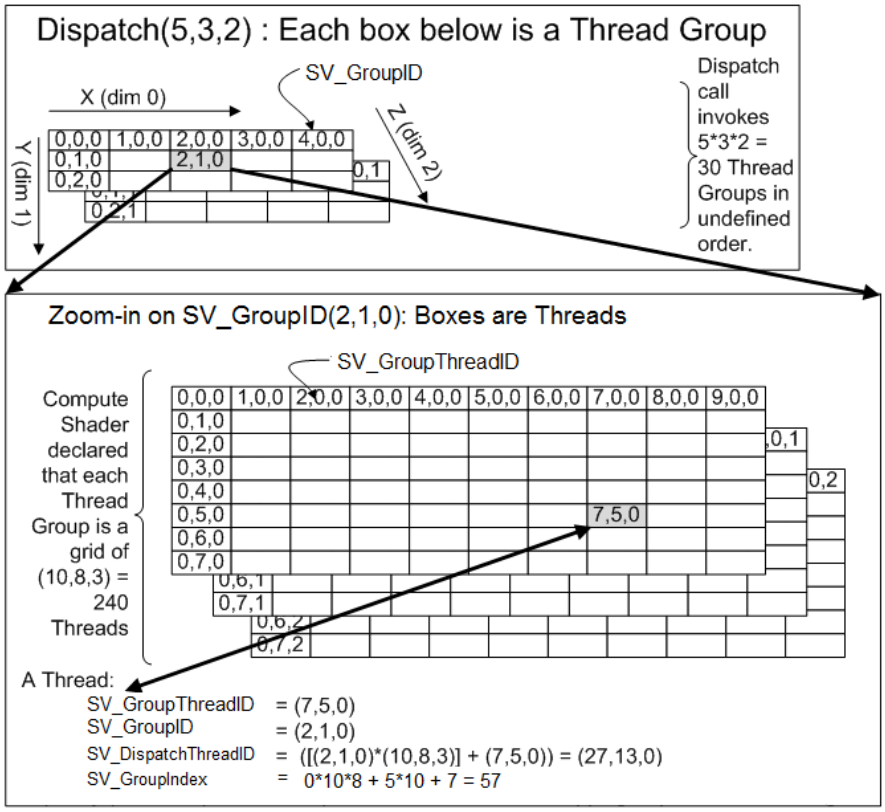
```
ID3D11DeviceContext::Dispatch(nx, ny, nz);
```

Where nx, ny, nz are the number of thread groups to execute. The total number of executed threads is x * y * z * nx * ny * nz.

In Compute Shader 5.0 number of threads in thread group is limited to 1024x1024x64. Group shared memory is limited to 32 KB per group.

Compute Shader also supports atomic operations. They are used when multiple threads are trying to modify Unordered Access View or group shared memory. Atomic operations can optionally return original value. The following atomic operations are supported:

- InterlockedAdd
- InterlockedAnd/InterlockedOr/InterlockedXor
- InterlockedCompareExchange
- InterlockedCompareStore
- InterlockedExchange
- InterlockedMax/InterlockedMin



Compute Shader thread execution