Tiled Shading - Preprint

To appear in JGT 2011

Ola Olsson and Ulf Assarsson

Chalmers University of Technology

Abstract

In this article we describe and investigate *Tiled Shading*. The tiled techniques, though simple, enable substantial improvements to both deferred and forward shading. Tiled Shading has previously been talked about only in terms of deferred shading (*Tiled Deferred Shading*). We contribute a more detailed description of the technique, introduce *Tiled Forward Shading* (a generalization of Tiled Deferred Shading to also apply to forward shading), and a thorough performance evaluation.

Tiled Forward Shading has many of the advantages of deferred shading, e.g. scene management and light management is decoupled. At the same time, unlike traditional deferred and tiled deferred shading, *Full Screen Anti Aliasing* and transparency is trivially supported.

We also contribute a thorough comparison of the performance of Tiled Deferred, Tiled Forward and traditional deferred shading. Our evaluation shows that Tiled Deferred Shading has the least variable worst case performance, and scales the best with faster GPUs. Tiled Deferred Shading is especially suitable when there are many light sources. Tiled Forward Shading is shown to be competitive for scenes with fewer lights, while being much simpler than traditional forward shading techniques.

Tiled shading also enables simple transitioning between deferred and forward shading. We demonstrate how this can be used to handle transparent geometry, frequently a problem when using deferred shading.

1 Introduction

Tiled Shading works by bucketing lights into screen space tiles. Each tile contains a list of (potentially) affecting lights. The tiles can then be processed independently to compute the lighting. We will describe how this technique can be used to great advantage, when implementing both deferred and forward shading.

Tiled Shading has similarities to *Tiled Rendering* [FPE+89], an old technique where the tiling is applied to geometry primitives, instead of lights. Tiled Rendering has been unable to scale to the millions of primitives used today. Lights, in contrast, number at most in the thousands, for even the most demanding real-time applications. For Tiled Shading, this enables quick bucketing, and real-time performance.

Tiled Deferred Shading is not a new technique, but has been described in several recent talks [BE08, And09, Swo09, Lau10]. In this article we describe the technique

in detail, and show how it can be generalized to apply to forward shading as well. We also contribute an in-depth comparison of the performance, comparing Tiled Forward Shading, Tiled Deferred Shading and traditional Deferred Shading.

1.1 Definitions

Lights, in this article, are *point lights* with a finite *range*, beyond which they do not contribute any lighting. Thus, when we refer to the *light volume*, this is the spherical volume defined by the light position and influence radius. This is obviously not physically correct, but represents a very common type of lights in real-time applications. The techniques presented can be applied to arbitrary kinds of lights, but this is beyond the scope of this article.

In the literature it is not always clear what is meant by *Deferred Shading*. In this article we mean the technique whereby all required geometry attributes are rendered into Geometry Buffers (*G-Buffers* [ST90]), in a single geometry pass. The G-Buffers contain attributes such as position, normal and material properties for each pixel. This is followed by a lighting pass, during which the lights are applied one at a time by rasterizing the light volumes. Note that this is different from *Deferred Lighting* [AA03], which only performs light computations in the deferred pass and adds a separate geometry pass to compute final shading. This technique is also referred to as *Light Pre-pass Rendering* [Eng09]. As deferred lighting has the same basic characteristics as deferred shading, we do not evaluate this technique in this article.

We use the term *Forward Shading* to mean when lighting is computed in the fragment (or sometimes, vertex) shader as part of the rasterization of the scene geometry. This technique is probably still the most common in real-time applications, such as games.

2 Tiled Deferred Shading

Recently, a technique called *Tiled Deferred Shading* has been discussed in the game development community [BE08, And09, Swo09, Lau10]. The primary goal is to solve the bandwidth problem, which is plaguing deferred shading. The problem is that each time a fragment is affected by a light, the geometry information must be read from the G-Buffers. Tiled deferred shading also brings other improvements over traditional deferred shading, listed below:

- G-Buffers are read exactly once for each lit fragment.
- Common terms in the rendering equation can be factored out.
- The frame buffer is written exactly once.
- Light accumulation is done in register, at full floating point precision.
- Fragments (in the same tile) coherently process the same lights.

The first point also benefits computations, as texture fetch and data unpacking need only be performed once. An example of the common terms that can be factored out is the material diffuse and specular colors, as these are the same for all lights. As can be seen, there are major improvements to both bandwidth and compute. The Tiled Deferred Shading algorithm is summarized in the following steps.

1. Render the (opaque) geometry into the G-Buffers.
2. Construct a screen space grid, covering the frame buffer, with some fixed tile size, $t = (x, y)$, e.g. 32×32 pixels.
3. For each light: find the screen space extents of the light volume and append the light ID to each affected grid cell.
4. For each fragment in the frame buffer, with location $f = (x, y)$.
   (a) sample the G-Buffers at f.
   (b) accumulate light contributions from all lights in tile at $\lfloor f/t \rfloor$
   (c) output total light contributions to frame buffer at f.

The first step is an ordinary deferred geometry pass. The second and third steps are part of the light grid construction process, which can be implemented in several ways (see Section 6 and 7).Listing 1 and Listing 2 shows a pseudo (GLSL) implementation of the shading process.

```
vec3 computeLight(vec3 position, vec3 normal, vec3 albedo,
                  vec3 specular, vec3 viewDir, float shininess,
                  ivec2 fragPos)
{
  ivec2 l = ivec2(fragPos.x / LIGHT_GRID_CELL_DIM_X,
                  fragPos.y / LIGHT_GRID_CELL_DIM_Y);

  int count = lightGrid[l.x + l.y * gridDim.x].x;
  int offset = lightGrid[l.x + l.y * gridDim.x].y;

  vec3 shading = vec3(0.0);

  for (int i = 0; i < count; ++i)
  {
    ivec2 dataInd = ivec2((offset + i) % TILE_DATA_TEX_WIDTH,
                          (offset + i) / TILE_DATA_TEX_WIDTH);
    int lightId = texelFetch(tileDataTex, dataInd, 0).x;
    shading += applyLight(position, normal, albedo, specular,
                     shininess, viewDir, lightId);
  }

  return shading;
}
```

Listing 1: Function for light accumulation, using tiled lighting. The grid is assumed to be stored the uniform `lightGrid`, which is an array of ivec2s, and the tile lists are

stored in a texture `tileDataTex`. Note how the function does not depend on anything else; It can thus be called from either a deferred or forward fragment shader, or indeed a compute shader. In fact, there is nothing preventing it being used from a vertex shader either.

In this example the shading is done in a deferred fragment shader (see Listing 2). This can be performed by drawing a full screen quad. It can equally well, or better, be implemented using compute shaders, CUDA, Open CL or SPUs, as platform or desire dictates.

```
void main()
{
  ivec2 fragPos = ivec2(gl_FragCoord.xy);
  vec3 albedo = texelFetch(albedoTex, fragPos).xyz;
  vec4 specShine = texelFetch(specularShininessTex, fragPos);
  vec3 position = unProject(gl_FragCoord.xy,
                    texelFetch(depthTex, fragPos));
  vec3 normal = texelFetch(normalTex, fragPos).xyz;
  vec3 viewDir = -normalize(position);

  gl_fragColor = computeLight(position, normal, albedo,
                  specShine.xyz, viewDir, specShine.w,
                  fragPos)
}
```

Listing 2: Tiled deferred fragment shader example. Simply loads G-Buffer contents, and invokes the function from Listing 1. Note that the position is reconstructed from the depth, this is done only once for all lights.

2.1 Limitations

Deferred shading has some notable weaknesses, some of which are shared by the tiled approach. The most important, perhaps, is when combined with *Full Screen Anti Aliasing* (FSAA). The primary issue is simply the required frame buffer size. At 1080p (1920x1080), and with 16 times *Multi Sample Anti Aliasing* (MSAA), a 32-bit color render requires almost 256Mb to store depth and color samples. For most current graphics hardware, adding several other G-Buffers on top of this is simply not possible. Performing the deferred shading post-resolve (i.e. after AA averaging) re-creates the aliasing, wherever shading changes quickly. It is, however, possible to compute the shading pre-resolve, and to do so only where edges are present [Swo09, Lau10] – provided, that is, G-buffers can be made to fit in memory.

Neither Deferred shading, nor Tiled Deferred Shading, provides a way to handle transparency. We are instead left with techniques that are approximate [KL09, ESSL10], or expensive [Eve01]. In Section 4, we present a way to conveniently support transparency when using tiled shading.

Another issue, common to both forward shading and tiled deferred shading, is that all lights (that cast shadows) must have their shadow maps built before the shading pass. This is because all lights are in flight simultaneously. Storing shadow maps

for hundreds of lights can be prohibitive in terms of memory. In contrast, traditional deferred shading computes all light contributions from each light, one at a time. This means a single shadow map can be re-used for all lights.

3 Tiled Forward Shading

Tiled shading is not limited to performing deferred shading. We can also apply the technique to traditional forward shading. We simply access the grid in the fragment shader, and apply the relevant lights. We illustrate this in Listing This approach has the following advantageous properties:

- Light management is decoupled from geometry.
- Light data can be uploaded to the GPU once per scene.
- FSAA works as expected.
- Common terms in the rendering equation can be factored out.
- Light accumulation is done in register, at full floating point precision.
- Same shading function as Tiled Deferred.

The traditional approach in forward shading is to find and upload a minimal set of lights for each batch of rendered geometry. This is time consuming and imposes an unfortunate conflict between optimal batch sizes and minimizing the number of lights. With the tiled approach, geometry batching can be optimized separately from lighting, and light data can be uploaded once for the entire scene.

As shading is done in its traditional place in the pipeline, there is no problem applying any FSAA scheme. Also there are no G-Buffers to worry about. Integrating tiled shading into an existing forward shading pipeline is straightforward, owing to the self contained nature of Tiled Shading.

The last property is also worth highlighting. As we use the same data structure for lights, we can use the same shader functions for both deferred and forward shading (illustrated in Listing 3). This enables much easier transition between the two techniques. We also exploit this to support transparency, described further in Section 4.

```
in vec3 normal;
in vec3 position;
in vec2 texCoord;
uniform vec3 specular;
uniform float shininess;
uniform float diffuse;

void main()
{
  ivec2 fragPos = ivec2(gl_FragCoord.xy);
  vec3 albedo = texture2D(albedoTex, texCoord).xyz * diffuse;
  vec3 viewDir = -normalize(position);

  gl_fragColor = computeLight(position, normal, albedo,
                  specular, viewDir, shininess,
                  fragPos);
}
```

Listing 3: Tiled forward fragment shader example. Simply invokes the function from Listing 1 with interpolated and uniform attributes.

3.1 Limitations

One drawback, compared to deferred shading techniques, is that each fragment may be shaded more than once. When overdraw occurs, the same fragment is influenced by several primitives, and consequently shaded for each. This overdraw problem can be addressed by using a pre-z pass, which introduces an extra geometry pass to prime the hierarchical/early Z buffer. This is already common practice to avoid re-processing complex shaders, and is ultimately a trade-off between the cost of an extra geometry pass vs the cost of the redundant shading work. For example, if there are few lights or low scene depth complexity, it may be quicker to skip the pre-z pass.

A related problem occurs when FSAA is enabled. Along primitive edges, fragments can also be shaded several times, up to once for each sample. Where there are shading discontinuities, this is what yields a nice softened edge. However, if the edge is an interior edge in a continuous mesh, this creates redundant work, whereby each sample is shaded to a very similar (or identical) tone. This is in part a general problem for GPUs, as triangles are becoming smaller (perhaps especially since the introduction of tessellation units). Deferred techniques, in contrast, can analyze the samples in the G-Buffers and compute shading only once, unless a discontinuity is found. In principle then, deferred shading has an advantage, though efficiently implementing it is not trivial.

4 Transparency

In real-time rendering, transparent geometry is usually handled in a separate pass. The transparent geometry is submitted in (rough) back to front order, with alpha blending enabled. This approach is impossible with deferred shading, as only one layer is

represented in the G-Buffers. A separate forward pipeline must thus be maintained, complete with light management, which, as mentioned earlier, is both complex and costly, and increasingly so as the number of lights grow.

However, using Tiled Shading we can reuse the grid built for the deferred pass, and apply a second Tiled Forward Shading pass with sorting and blending. Due to the fact that all light information is stored in a single global structure (the grid), the rendering pipeline can be mostly the same for both passes. This makes it vastly simpler to support transparency, while using deferred shading for the bulk of the geometry.

```
in vec3 position;
in vec3 normal;

// uniforms: textures, material attributes etc...

void main()
{
    // compute the needed attributes as desired
#if ENABLE_DEFERRED_OUTPUT
    outputToGBuffers(position, normal, diffuse, specular ...);
#else
    outputShading(position, normal, diffuse, specular ...);
#endif
}
```

Listing 4: The selection between forward and deferred in a shader can be controlled with a single flag, choosing between output shader functions.

5 Algorithm Comparison

In Table 1, we summarize the the key differences and properties of the algorithms, for easy comparison of important features of the algorithms. Note that the properties are not independent: many depend on other, more fundamental features. They are listed in this way nonetheless, in order to highlight important practical differences.

The most prominent property is the innermost loop structure. If this loop is over the pixels, then lights are accessed in a sequential manner. This makes it possible to re-use shadow maps. Conversely, when the innermost loop is over the lights, the frame buffer data is sequentially accessed. This is what enables fetching the G-Buffer data once, providing the large bandwidth savings of tiled deferred shading.

6 Building Tiles

When constructing the grid, our first task is to choose tile size. This choice involves many trade offs between memory and computation. For example, smaller tiles means more storage and bandwidth use for the grid, but less wasted computation at the light boundaries. It is therefore not likely that there exists a single best tile size for all scene configurations (or even for all views of the same scene).

Table 1: Comparison of properties of the algorithms.

	Deferred	Tiled Deferred	Tiled Forward
Innermost loop	Pixels	Lights	Lights
Light data access pattern	Sequential	Random	Random
Pixel data access pattern	Random	Sequential	Sequential
Re-use Shadow Maps	Yes	No	No
Shading Pass	Deferred	Deferred	Geometry
G-Buffers	Yes	Yes	No
Overdraw of shading	No	No	Yes
Transparency	Difficult	Simple[a]	Simple
Supporting FSAA	Difficult	Difficult	Trivial
Bandwidth Usage	High	Low	Low
Light volume intersection	Per Pixel	Per Tile	Per Tile

[a]That is, simple to implement by applying a Tiled Forward pass reusing the light grid (described in Section 4).

For this article we use a tile size of 32×32, as this gives three orders of magnitude fewer grid cells than pixels. Consequently, the time and memory spent on managing tiles should not create a bottleneck. We did not experimentally evaluate varying the tile size. However, 16×16 has also been reported to work well [And09, Lau10].

6.1 Light Insertion

Next, we must insert the lights into the grid. The simplest way is to find the screen space extents of the light bounding sphere [Len02, SWBG06], and then insert the light into the covered grid cells. This process is simple enough to have a relatively small cost even for hundreds of lights, if implemented on the CPU. For thousands of lights, with high overdraw, it can be implemented on the GPU. The CPU approach is also suitable for older hardware and APIs, which do not support compute shaders.

An interesting way to implement this, which we have not tested, might be to use rasterization to build the grid. This would allow for easy handling of arbitrarily shaped light volumes, e.g. spot light cones. To ensure all lights are included we must use conservative rasterization [HAMO05], as ordinary rasterization only samples fragment centers; and an A-Buffer, for example using per-fragment linked lists [TG10]. It is, however, unclear how well the GPU would perform with these very small render targets, e.g. 60×34 when using tiles of size 32×32 at 1080p. A quick estimate, obtained by performing standard deferred shading to a render target of this size, indicate that it would be at least around five times slower, compared to the screen space bounding sphere approach.

6.2 Data Structure

To facilitate look up from shaders, we must store the data structure in a suitable format. We have chosen to use three arrays, as depicted in Figure 1. The *Light Grid* contains an offset to and size of the light list for each tile. The *Tile Light Index Lists* contains light indices, referring to the lights in the *Global Light List*. This data structure can

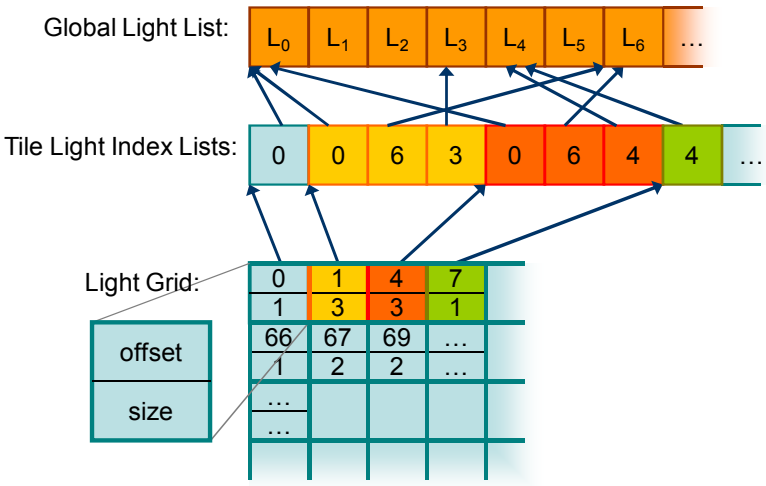

Figure 1: Grid data structure.

easily be stored on the GPU as constant buffers or textures. The index list lengths vary with light overdraw, and can become relatively large. It is thus suitable for storing in a texture.

6.3 Depth Range Optimization

Standard deferred shading implementations often use the *Stencil Optimization* [AA03]. This technique uses an approach analogous to shadow volumes, but with light volumes, to create a stencil mask. The mask lets through only fragments that are actually inside the light volume. This can offer substantial performance improvements where the light is large on screen, but only affects a few fragments (for example a light in the middle of an empty corridor).

Tiled shading can make use of a similar technique, using the depth buffer to compute a min and max Z value for each grid tile. These bounds are then used when adding lights to the grid, to exclude lights that cannot affect the geometry in the tile. Note that tiles that span a depth discontinuity can have a quite large difference between min and max depth. The tiled techniques will therefore always cull somewhat less work than the stencil optimization (this is shown in Figure 3).

Computing the tile min/max depth requires access to the Z buffer before the grid is constructed. This is not a problem if a pre-z or deferred pass is performed. The min/-max operation is suitable for implementation on the GPU, using a parallel reduction

per tile. The result can be read back to the host for CPU grid building, or the grid can be constructed entirely on the GPU. Note that if the CPU is used, the process cannot be completely pipelined with most current APIs.

When rendering transparent geometry (using the tiled forward approach), this geometry is not represented in the depth buffer. We can thus only use the farthest Z value, to reject lights that are hidden by the opaque geometry. We could, if needed, render the transparent geometry to a separate depth buffer with reversed depth test, to find the nearest depth value. This would allow us to reject any light completely in front of all transparent geometry. In scenes where the transparent geometry lies close to the opaque, for example shallow water, this could be a useful optimization.

7 Single Kernel Tiled Deferred Shading

In past presentations on tiled deferred shading [And09, Lau10], the whole process of grid building and lighting application is performed in a single DirectX 11 compute shader. This works by launching one thread group per tile, with one thread per fragment in the tile. Each thread group then independently tests all lights. The six frustum planes for the tile are constructed and then tested against each light bounding sphere. The light list for the tile is built in local, on chip, memory, and the threads involved then switch to lighting the fragments in the tile.

Treating each tile independently leads to a lot of redundant calculations. For example, all tiles in a column or row share planes. Also, consider a tile which has at least two opposite neighbors that are affected by a light. This tile must also be affected by the light, without needing any plane tests at all. The approach also requires atomic operations and thread group synchronization, making it unsuitable for older hardware.

An advantage for the single kernel approach is that the process is self contained, and hence simple to implement and integrate. Storing the light index list in shared memory saves some bandwidth, but is not a large improvement as these lists are relatively small.

8 Performance Evaluation

Performance was measured on a PC with an Intel Core 2 Quad at 2.5GHz, using either an NVIDIA GTX 280 or GTX 480 GPU (as indicated). The frames were rendered at full HD resolution, 1920x1080. The following variations were implemented:

1. **TiledForward** - Tiled Forward Shading, using the light grid from the pixel shader. Uses the CPU to build the grid, and OpenGL for everything else.

2. **TiledForward-PreZ** - As above, with pre-z pass and depth range optimization (min-max reduction implemented in CUDA).

3. **Deferred** - Standard deferred shading.

4. **Deferred-Stencil** - As above, with stencil optimization to cull unaffected fragments within light volumes.

5. **TiledDeferred** - Tiled Deferred Shading, using CUDA to build the grid, compute depth range and lighting.

We evaluated several flavors of Tiled Deferred Shading, but we only report results for the best performing version. This was an implementation of the process outlined in Section 2 and 6, using multiple CUDA kernels.

We tried using a full screen quad in OpenGL, computing the lighting in the fragment shader. However, this approach was substantially slower. For comparison we also ported the single kernel method used in [Lau10] to CUDA. We optimized their depth min/max reduction by using a warp-parallel SIMD reduction (similar to `warpReduce` in [Har08]) within each warp and only atomic operations between warps. This sped up the reduction by a factor of six for a 16×16 tile on our GTX280. After this optimization, performance is very close to our chosen implementation, with a small advantage for the reported version.

The light model is a fairly ordinary Blinn-Phong model with diffuse and specular reflections. To facilitate this model, the G-Buffers are: *Depth*, *Normal*, *Diffuse Color*, *Specular Color and Shininess* and *Emissive and Ambient*.

Each buffer, except depth, store four 16-bit floating point values per fragment. However we also tested using 32 bits, to investigate the impact of G-Buffer size on performance (see Tables 2 and 3). The depth buffer always stores a scalar 32 bit floating point value.

We implemented the 32-bit G-Buffers to explore how varying the balance between compute and bandwidth affect the outcome. The expected behavior is that more bandwidth use will favor tiled deferred, whereas increasing compute demand favors traditional deferred. Changing the light model or packing the G-Buffers would have a similar impact.

As test scene we choose the Robots scene from the BART suite [LAM01]. We choose this scene rather than a, perhaps better looking, game scene because it will allow our experiments to be repeated. The BART suite is freely available, whereas most game data is not.

We augmented the scene with point lights, and created two variations, with differing light distributions (for reference, the main street is 29.1 units wide):

- **Many Static Lights** - 924 lights evenly spaced along the animation paths of the robots, with random sideways offsets. The lights have a range of 12.5 units.
- **Few Dynamic Lights** - One light attached to each of the 11 robots. The lights have a range of 40 units.

8.1 Tiled Deferred Shading Performance

Overall, the results confirm that Tiled Deferred Shading is much less variable, with smaller differences between best and worst case performance. This is seen in Figure 2 and Tables 2 and 3.

Traditional deferred shading is usually bandwidth limited. Thus we expect frame times to scale linearly with G-Buffer size. This was confirmed in our experiments, as can be seen in Tables 2 and 3. The tiled deferred variants are much less affected.

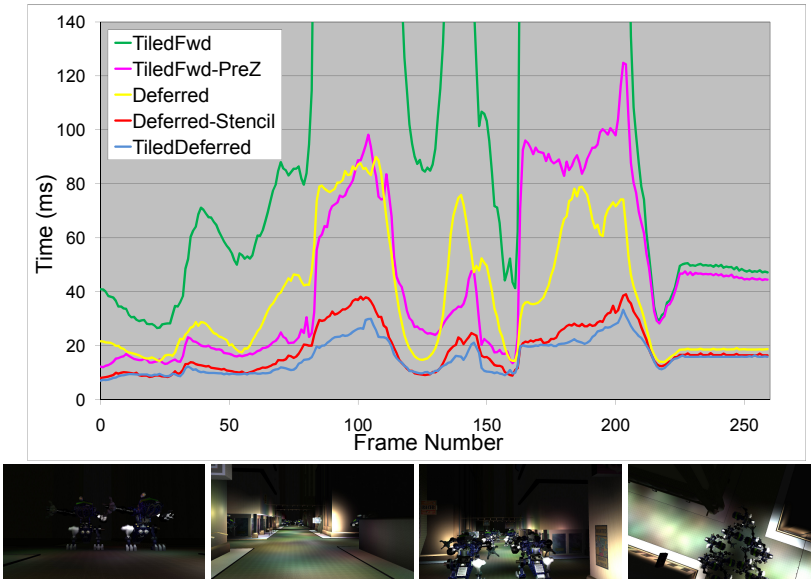

Figure 2: Frame times over the animation sequence in the scene with many static lights, measured on an NVIDIA GTX280 GPU. The animation is sampled at 5 fps. The peaks correspond to times when the camera is looking through a great many lights. Note that the tiled forward version is clipped, in order to make the presentation clearer. It plateaus at around 400ms, with a similar shape to the others. The thumbnails below the graph show frames 0, 86, 186 and 259.

Tables 2 and 3 also shows results from a GTX480. This new GPU roughly doubles the (attainable) compute capabilities, whereas bandwidth only grows by about 30%. This also favors the tiled techniques, almost doubling their performance on the new architecture. Traditional deferred only improves by about the expected 30%.

Notice that traditional deferred with stencil optimization is fairly competitive, when we use 16-bit G-Buffers. However, with fatter G-Buffers, or increasing compute/bandwidth ratio in the GPU, performance quickly falls behind the tiled techniques. This may not be enough to outweigh the advantages, such as being able to share shadow map storage between lights.

In Figure 3, we show how the number of lighting computations (i.e. number of lighting function invocations) performed each frame for the different techniques. As expected, the stencil optimization is the most efficient at culling work, as it is performed at a per-fragment level. TiledFwd is worst, but is substantially improved by the addition of pre-z pass and depth range optimization. Notice the clear similarity to the frame time curves shown in Figure 2. This implies that (perhaps unsurprisingly) there is a reasonably fixed cost per lighting operation, albeit with quite different scales.

Table 2: Average frame times in milliseconds for the scene with many lights. Min and max frame times are also shown. Results for 16-bit and 32-bit G-Buffers are shown, for both the GTX280 and GTX480 GPUs. Note that G-Buffers are not used for forward shading; thus only one value is needed for these techniques.

GPU	GTX280		GTX480	
G-Buffer Depth	16-bit	32-bit	16-bit	32-bit
TiledDeferred	15.7	17.2	7.87	10.4
min / max	7.05 / 33.3	8.82 / 34.8	4.19 / 15.9	6.91 / 18.2
TiledFwd	148		43.0	
min / max	26.5 / 410		11.0 / 107	
TiledFwd-PreZ	45.5		30.1	
min / max	12.1 / 125		9.31 / 72.4	
Deferred	38.3	82.1	26.8	51.3
min / max	14.0 / 90.0	27.7 / 197	9.36 / 64.3	18.2 / 121
DeferredStencil	18.4	28.3	12.2	20.2
min / max	8.1 / 39.1	12.4 / 64.8	4.98 / 26.6	9.55 / 43.8

Table 3: Average frame times in milliseconds for the scene with few lights. Min and max frame times are also shown. Results for 16-bit and 32-bit G-Buffers are shown, for both the GTX280 and GTX480 GPUs. Note that G-Buffers are not used for forward shading; thus only one value is needed for these techniques.

GPU	GTX280		GTX480	
G-Buffer Depth	16-bit	32-bit	16-bit	32-bit
TiledDeferred	5.55	7.06	3.34	6.34
min / max	3.00 / 7.13	5.20 / 9.44	2.36 / 4.48	4.68 / 8.98
TiledFwd	9.08		4.03	
min / max	3.39 / 21.1		1.39 / 9.95	
TiledFwd-PreZ	8.39		5.46	
min / max	5.53 / 14.4		3.22 / 9.82	
Deferred	6.55	12.8	4.17	9.36
min / max	3.86 / 10.6	6.65 / 22.7	2.12 / 7.41	5.29 / 15.5
DeferredStencil	6.20	10.6	3.65	8.05
min / max	3.90 / 9.30	6.03 / 16.9	1.65 / 5.81	4.07 / 12.4

8.2 Tiled Forward Shading Performance

Tiled Forward scales much worse with increasing light overdraw, but the performance curves have the same overall shape (see Figure 3). In fact, TiledFwd-PreZ is close to a factor four, and a constant offset, slower than TiledDeferred. Since they ought to perform the same number of lighting computations, the hardware is not utilized as efficiently. One factor may be fragments belonging to different tiles being packed into the same warp (SIMD unit), causing divergence. Also, along the edges of triangles, there can be many pixel quads that are not full, i.e. up to three of the four fragments are outside the triangle (a 2x2 pixel quad is the basic unit handled by fragment shaders), wasting up to $3/4$ of computational resources. Furthermore, early Z cull is conservative. Thus, there will be fragments that are shaded, but finally discarded by the Z test, pre-z pass notwithstanding.

On the GTX480, the TiledFwd technique improves by up to four times for the worst cases. This brings performance closer to expectations, given the high number of lighting computations. The TiledFwd-PreZ only shows modest improvement. It is unclear why it does not improve as much as TiledFwd.

One scenario where tiled forward shading should work well is when scene depth complexity is low, and most light volumes are overlapping to the geometry. The is because lights overlapping the geometry nullifies the effect of depth range optimizations, as these optimizations are designed to cull lights that do *not* overlap the geometry. This description would match a Real Time Strategy (RTS) game pretty well, assuming a top down view and lots of action on, or near, the terrain.

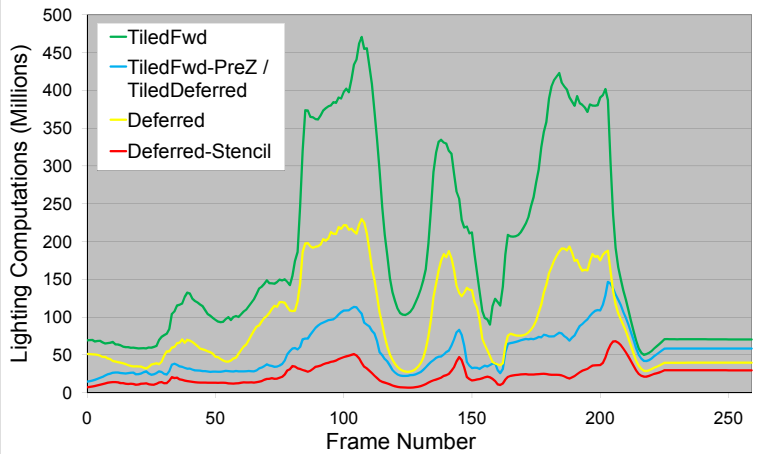

Figure 3: Lighting computations each frame, that is the number of times the lighting function is executed, over the animation sequence in the scene with many static lights. This value corresponds to light volume overdraw for deferred shading, measured using occlusion queries. For TiledFwd-PreZ and TiledDeferred, this is simply the tile size multiplied by the total length of the Tile Light Lists. TiledFwd suffers from geometry overdraw; we measured this using a simple shader, outputting the light counts for each pixel with additive blending enabled.

8.3 Fewer Lights

When rendering fewer lights (i.e. the scene with 11 dynamic lights), the situation is quite different, see Table 3. One important factor is that the total frame time is smaller, and consequently, a larger proportion is spent rendering the model into the G-Buffers. This favors the forward shading approach, especially on the GTX480 using 32-bit G-Buffers, when compared to the deferred techniques.

However, tiled deferred is still the technique which scales best across platforms and G-Buffer depth. It has the fastest worst case performance in all tests performed. The worst case performance is arguably the most important metric for real-time applications, as a stable frame rate is very important for the perceived quality.

At the same time, it is clear that tiled forward shading offers very competitive performance on the GTX 480. Remember that forward shading supports both AA and transparency, and may therefore be a good choice if not so many lights are used.

References

[AA03] Jukka Arvo and Timo Aila. Optimized shadow mapping using the stencil buffer. *journal of graphics, gpu, and game tools*, 8(3):23–32, 2003.

[And09] Johan Andersson. Parallel graphics in frostbite - current & future. SIGGRAPH Course: Beyond Programmable Shading, 2009.

[BE08] Christophe Balestra and Pål-Kristian Engstad. The technology of uncharted: Drake's fortune. Game Developer Conference, 2008.

[Eng09] Wolfgang Engel. The light pre-pass renderer: Renderer design for efficient support of multiple lights. SIGGRAPH Course: Advances in real-time rendering in 3D graphics and games, 2009.

[ESSL10] Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. Stochastic transparency. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 157–164, New York, NY, USA, 2010. ACM.

[Eve01] Cass Everitt. Interactive order-independent transparency. NVIDIA White Paper, 2001.

[FPE+89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 79–88, New York, NY, USA, 1989. ACM.

[HAMO05] J Hasselgren, T Akenine-Möller, and L Ohlsson. Conservative rasterization. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2*, chapter 42, pages 677–690. Addison-Wesley Professional, Reading, MA, 2005.

[Har08] Mark Harris. Optimizing parallel reduction in cuda. NVIDIA CUDA Sample, 2008.

[KL09] Scott Kircher and Alan Lawrance. Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects. In *Sandbox '09: Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, pages 39–45, New York, NY, USA, 2009. ACM.

[LAM01] Jonas Lext, Ulf Assarsson, and Tomas Möller. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications*, 21:22–31, 2001.

[Lau10] Andrew Lauritzen. Deferred rendering for current and future rendering pipelines. SIGGRAPH Course: Beyond Programmable Shading, 2010.

[Len02] Eric Lengyel. The mechanics of robust stencil shadows. Gamasutra, 2002.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, 1990.

[SWBG06] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus Gross. Gpu-based ray casting of quadratic surfaces. In *Proceedings of Eurographics Symposium on Point-Based Graphics*, 2006.

[Swo09] Matt Swoboda. Deferred lighting and post processing on playstation 3. Game Developer Conference, 2009.

[TG10] Nick Thibieroz and Holger Grün. Oit and gi using dx11 linked lists. Game Developer Conference, 2010.