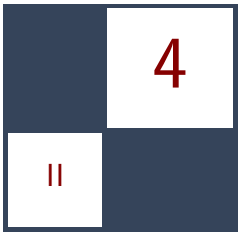


Tiled Forward Shading

Markus Billeter, Ola Olsson, and Ulf Assarsson

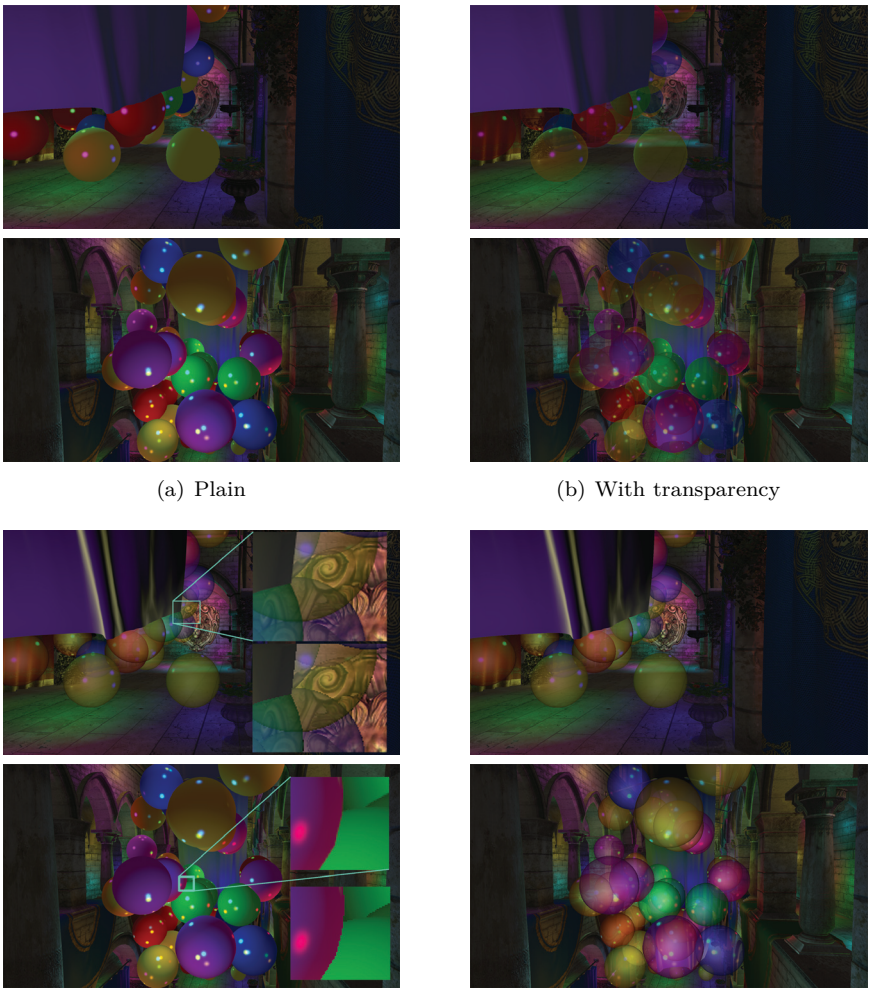
4.1 Introduction

We will explore the *tiled forward shading* algorithm in this chapter. Tiled forward shading is an extension or modification of *tiled deferred shading* [Balestra and Engstad 08, Swoboda 09, Andersson 09, Lauritzen 10, Olsson and Assarsson 11], which itself improves upon traditional deferred shading methods [Hargreaves and Harris 04, Engel 09].

Deferred shading has two main features: decoupling of lighting and shading from geometry management and minimization of the number of lighting computations performed [Hargreaves and Harris 04]. The former allows for more efficient geometry submission and management [Shishkovtsov 05] and simplifies shaders and management of shader resources. However the latter is becoming less of an issue on modern GPUs, which allow complex flow control in shaders, and support uniform buffers and more complex data structures in GPU memory.

Traditional forward pipelines typically render objects one by one and consider each light for each rasterized fragment. In deferred shading, one would instead render a representation of the geometry into a screen-sized G-buffer [Saito and Takahashi 90], which contains shading data for each pixel, such as normal and depth/position. Then, in a separate pass, the lighting and shading is computed by, for example, rendering light sources one by one (where each light source is represented by a bounding volume enclosing the light's influence region). For each generated fragment during this pass, data for the corresponding pixel is fetched from the G-buffer, shading is computed, and the results are blended into an output buffer. The number of lighting computations performed comes very close to the optimum of one per light per visible sample (somewhat depending on the bounding volumes used to represent light sources).

Deferred shading thereby succeeds in reducing the amount of computations needed for lighting, but at the cost of increased memory requirements (the G-buffer is much larger than a color buffer) and much higher memory bandwidth usage. Tiled deferred shading fixes the latter (Section 4.2), but still requires large G-buffers.



(a) Plain

(b) With transparency

(c) MSAA

(d) Custom shaders

Figure 4.1. We explore tiled forward shading in this article. (a) While tiled deferred shading outperforms tiled forward shading in the plain (no transparency, no multisample antialiasing (MSAA)) case by approximately 25%, (b) tiled forward shading enables use of transparency. Additionally, (c) we can use hardware supported MSAA, which, when emulated in deferred shading requires large amounts of memory. Furthermore, at 4× MSAA, tiled forward shading outperforms deferred with equal quality by 1.5 to 2 times. The image shows 8× MSAA, which we were unable to emulate for deferred rendering due to memory constraints. (d) Finally, we discuss custom shaders. As with standard forward rendering, shaders can be attached to geometry chunks. The scene contains 1,024 randomly placed lights, and the demo is run on an NVIDIA GTX480 GPU.

Tiled forward shading attempts to combine one of the main advantages of (tiled) deferred rendering, i.e., the reduced amount of lighting computations done, with the advantages of forward rendering. Besides reduced memory requirements (forward rendering does not need a large G-buffer), it also enables transparency [Kircher and Lawrance 09,Enderton et al. 10] (Section 4.5), enables multisampling schemes [Swoboda 09, Lauritzen 10] (Section 4.6), and does not force the use of übershaders if different shading models must be supported (Section 4.7). See the images in Figure 4.1 for a demonstration of these different aspects.

4.2 Recap: Forward, Deferred, and Tiled Shading

The terms *forward*, *deferred,* and *tiled shading* will be appearing quite frequently in this chapter. Therefore, let us define what we mean, since usage of these terms sometimes varies slightly in the community. The definitions we show here are identical to the ones used by [Olsson and Assarsson 11].

With *forward shading*, we refer to the process of rendering where lighting and shading computations are performed in the same pass as the geometry is rasterized. This corresponds to the standard setup consisting of a vertex shader that transforms geometry and a fragment shader that computes a resulting color for each rasterized fragment.

Deferred shading splits this process into two passes. First, geometry is rasterized, but, in contrast to forward shading, geometry attributes are output into a set of geometry buffers (G-buffers). After all geometry has been processed this way, an additional pass that computes the lighting or full shading is performed using the data stored in the G-buffers.

In its very simplest form, the second pass (the lighting pass) may look something like following:

```
for each G-buffer sample {
  sample_attr = load attributes from G-buffer

  for each light {
    color += shade(sample_attr, light)
  }

  output pixel color;
}
```

Sometimes, the order of the loops is reversed. The deferred algorithm described in Section 4.1 is an example of this.

The light pass shown above requires $\mathcal{O}(N_{\text{lights}} \cdot N_{\text{samples}})$ lighting computations. If we somehow know which lights were affecting what samples, we could reduce this number significantly [Trebilco 09].

Tiled deferred shading does this by dividing samples into tiles of $N \times N$ samples. (We have had particularly good successes with $N = 32$, but this should be somewhat hardware and application dependent.) Lights are then assigned to these tiles. We may optionally compute the minimum and maximum Z-bounds of each tile, which allows us to further reduce the number of lights affecting each tile (more discussion on this in Section 4.4).

Benefits of tiled deferred shading [Olsson and Assarsson 11] are the following:

- The G-buffers are read only once for each lit sample.
- The framebuffer is written to once.
- Common terms of the rendering equation can be factored out and computed once instead of recomputing them for each light.
- The work becomes coherent within each tile; i.e., each sample in a tile requires the same amount of work (iterates over the same lights), which allows for efficient implementation on SIMD-like architectures (unless, of course, the shader code contains many branches).

For tiled deferred shading (and most deferred techniques) to be worthwhile, most lights must have a limited range. If all lights potentially affect all of the scene, there is obviously no benefit to the tiling (Figure 4.2(a)).

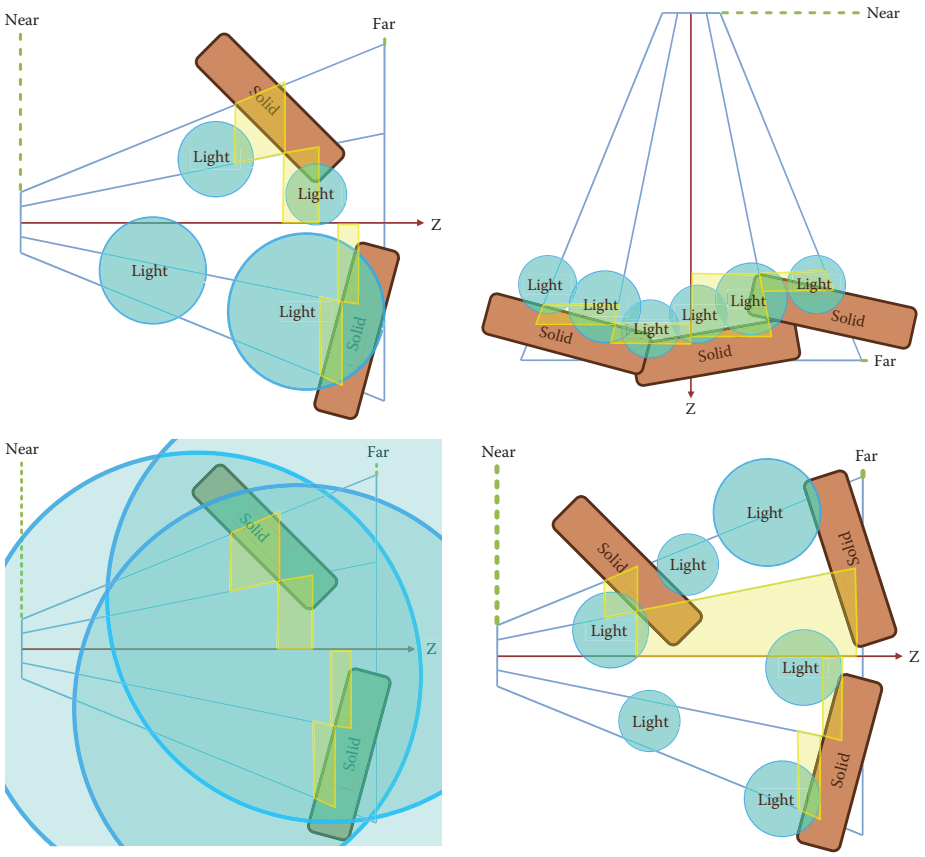
Tiled deferred shading can be generalized into *Tiled Shading*, which includes both the deferred and forward variants. The basic tiled shading algorithm looks like the following:

1. Subdivide screen into tiles.
2. Optional: find minimum and maximum Z-bounds for each tile.
3. Assign lights to each tile.
4. For each sample: process all lights affecting the current sample's tile.

Step 1 is basically free; if we use regular $N \times N$ tiles, the subdivision is implicit. Finding minimum and maximum Z-bounds for each tile is optional (Step 2). For instance, a top-down view on a scene with low depth complexity may not allow for additional culling of lights in the Z-direction. Other cases, however, can benefit from tighter tile Z-bounds, since fewer lights are found to influence that tile (Figure 4.2(b)).

In tiled *deferred* shading, the samples in Step 4 are fetched from the G-buffers. In tiled *forward* shading, the samples are generated during rasterization. We will explore the latter in the rest of the article.

We recently presented an extension to tiled shading, called *clustered shading* [Olsson et al. 12b]. Clustered shading is an extension of tiled shading that



(a) Small and large lights

(b) Top-down and first person views

Figure 4.2. (a) The effect of having lights that are too large (bottom image): there is no gain from the tiling, as all light sources affect all tiles (drawn in yellow), compared to the top image, where there is one light per tile on average. (b) Comparison of a top-down view and a first-person view. In the top-down view (top), all lights are close to the ground, which has only small variations in the Z-direction. In this case, not much is gained from computing minimum and maximum Z-bounds. In the first-person view (bottom), the bounds help (three lights in the image affect no tiles at all).

handles complex light and geometry configurations more robustly with respect to performance. However, tiled forward shading is significantly simpler to implement, and works on a much broader range of hardware. We will discuss the clustered shading extension and how it interacts with the tiled forward shading presented here in Section 4.8.

4.3 Tiled Forward Shading: Why?

The main strength of deferred techniques, including tiled deferred shading, is that over-shading due to over-draw is eliminated. However, most deferred techniques suffer from the following weaknesses when compared to forward rendering:

- Transparency/blending is tricky, since traditional G-buffers only allow storage of a single sample at each position in the buffer.
- The memory storage and bandwidth requirements are higher and become even worse with MSAA and related techniques (Section 4.6).

Forward rendering, on the other hand, has good support for

- transparency via alpha blending,
- MSAA and related techniques through hardware features (much less memory storage is required).

In addition, forward rendering trivially supports different shaders and materials for different geometries. Deferred techniques would generally need to fall back to übershaders (or perform multiple shading passes).

A special advantage for tiled forward shading is its low requirements on GPU hardware. It is possible to implement a tiled forward renderer without compute shaders and other (relatively) recent hardware features. In fact, it is possible to implement a tiled forward renderer on any hardware that supports dependent texture lookups in the fragment shader. On the other hand, if compute shaders are available, we can take advantage of this during, say, light assignment (Section 4.4).

In the following sections, we first present a tiled forward shading renderer to which we add support for transparency, MSAA and finally experiment with having a few different shaders for different objects. We compare performance and resource consumption to a reference tiled deferred shading renderer and show where the tiled forward renderer wins.

4.4 Basic Tiled Forward Shading

We listed the basic algorithm for all tiled shading variants in Section 4.2. For clarity, it is repeated here including any specifics for the forward variant.

1. Subdivide screen into tiles
2. Optional: pre-Z pass—render geometry and store depth values for each sample in the standard Z-buffer.
3. Optional: find minimum and/or maximum Z-bounds for each tile.

4. Assign lights to each tile.

5. Render geometry and compute shading for each generated fragment.

Subdivision of screen. We use regular $N \times N$ pixel tiles (e.g., $N = 32$). Having very large tiles creates a worse light assignment; each tile will be affected by more light sources that affect a smaller subset of samples in the tile. Creating very small tiles makes the light assignment more expensive and increases the required memory storage—especially when the tiles are small enough that many adjacent tiles are found to be affected by the same light sources.

Optional pre-Z pass. An optional pre-Z pass can help in two ways. First, it is required if we wish to find the Z-bounds for each tile in the next step. Secondly, in the final rendering pass it can reduce the number of samples that need to be shaded through early-Z tests and similar hardware features.

The pre-Z pass should, of course, only include opaque geometry. Transparent geometry is discussed in Section 4.5.

Though a pre-Z pass is scene and view dependent, in our tests we have found that adding it improves performance significantly. For instance, for the images in Figure 4.1(a), rendering time is reduced from 22.4 ms (upper view) and 37.9 ms (lower view) to 15.6 ms and 18.7 ms, respectively.

Optional minimum or maximum Z-bounds. If a depth buffer exists, e.g., from the pre-Z pass described above, we can use this information to find (reduce) the extents of each tile in the Z-direction (depth). This yields smaller per-tile bounding volumes, reducing the number of lights that affect a tile during light assignment.

Depending on the application, finding only either the minimum or the maximum bounds can be sufficient (if bounds are required at all). Again, transparency (Section 4.5) interacts with this, as do various multisampling schemes (Section 4.6).

In conjunction with the pre-Z test above, the minimum or maximum reduction yields a further significant improvement for the views in Figure 4.1(a). Rendering time with both pre-Z and minimum or maximum reduction is 10.9 ms (upper) and 13.8 ms (lower), respectively—which is quite comparable to the performance of tiled deferred shading (8.5 ms and 10.9 ms). The reduction itself is implemented using a loop in a fragment shader (for simplicity) and currently takes about 0.75 ms (for $1{,}920 \times 1{,}080$ resolution).

Light assignment. Next, we must assign lights to tiles. Basically, we want to efficiently find which lights affect samples in which tiles. This requires a few choices and considerations.

In tiled shading, where the number of tiles is relatively small (for instance, a resolution of $1{,}920 \times 1{,}080$ with 32×32 tiles yields just about 2,040 tiles), it can be feasible to do the assignment on the CPU. This is especially true if the

number of lights is relatively small (e.g., a few thousand). On the CPU, a simple implementation is to find the screen-space axis-aligned bounding boxes (AABBs) for each light source and loop over all the tiles that are contained in the 2D region of the AABB. If we have computed the minimum and maximum depths for each tile, we need to perform an additional test to discard lights that are outside of the tile in the Z-direction.

On the GPU, a simple brute-force variant works for moderate amounts of lights (up to around 10,000 lights). In the brute-force variant, each tile is checked against all light sources. If each tile gets its own thread group, the implementation is fairly simple and performs relatively well. Obviously, the brute-force algorithm does not scale very well. In our clustered shading implementation [Olsson et al. 12b], we build a simple light hierarchy (a BVH) each frame and test the tiles (clusters) against this hierarchy. We show that this approach can scale up to at least one million lights in real time. The same approach is applicable for tiled shading as well.

Rendering and shading. The final step is to render all geometry. The pipeline for this looks almost like a standard forward rendering pipeline; different shaders and related resources may be attached to different chunks of geometry. There are no changes to the stages other than the fragment shader.

The fragment shader will, for each generated sample, look up which lights affect that sample by checking what lights are assigned to the sample's tile (Listing 4.1).

4.5 Supporting Transparency

As mentioned in the beginning of this article, deferred techniques have some difficulty dealing with transparency since traditional G-buffers only can store attributes from a single sample at each buffer location [Thibieroz and Grün 10]. However, with forward rendering, we never need to store attributes for samples. Instead we can simply blend the resulting colors using standard alpha-blending.

Note that we are not solving the order-dependent transparency problem. Rather, we support, unlike many deferred techniques, standard alpha-blending where each layer is lit correctly. The application must, however, ensure that transparent objects are drawn in the correct back-to-front order.

We need to make the following changes, compared to the basic tiled forward shading algorithm (Section 4.4).

Optional minimum or maximum Z-bounds. We need to consider transparent geometry here, as nonoccluded transparent objects will affect a tile's bounds inasmuch that it moves a tile's minimum Z-bound ("near plane") closer to the camera.

We ended up using two different sets of tiles for opaque and transparent geometries, rather than extending a single set of tiles to include both opaque and

```
// 1D texture holding per-tile light lists
uniform isampleBuffer tex_tileLightLists;

// uniform buffer holding each tile's light count and
// start offset of the tile's light list (in
// tex_tileLightIndices)
uniform TileLightListRanges
{
  ivec2 u_lightListRange[MAX_NUM_TILES];
}

void shading_function( inout FragmentData aFragData )
{
  // ...

  // find fragment's tile using gl_FragCoord
  ivec2 tileCoord = ivec2(gl_FragCoord.xy)
    / ivec2(TILE_SIZE_X, TILE_SIZE_Y);
  int tileIdx = tileCoord.x
    + tileCoord.y * LIGHT_GRID_SIZE_X;

  // fetch tile's light data start offset (.y) and
  // number of lights (.x)
  ivec2 lightListRange = u_lightListRange[tileIdx].xy;

  // iterate over lights affecting this tile
  for( int i = 0; i < lightListRange.x; ++i )
  {
    int lightIndex = lightListRange.y + i;

    // fetch global light ID
    int globalLightId = texelFetch(
      tex_tileLightLists, lightIndex ).x;

    // get the light's data (position, colors, ...)
    LightData lightData;
    light_get_data( lightData, globalLightId );

    // compute shading from the light
    shade( aFragData, lightData );
  }

  // ...
}
```

Listing 4.1. GLSL pseudocode that demonstrates how lights affecting a given sample are fetched. First, we find the fragment's associated tile (`tileIdx`) based on its position in the framebuffer. For each tile we store two integers (`u_lightListRange` array), one indicating the number of lights affecting the tile, and the other describes the offset into the global per-tile light list buffer (`tex_tileLightLists`). The light list buffer stores a number of integers per tile, each integer identifying a globally unique light that is affecting the tile.

transparent geometries. The Z-bounds for tiles used with opaque geometry are computed as described in Section 4.4, which gives a good light assignment for the opaque geometry (Figure 4.3(a)).

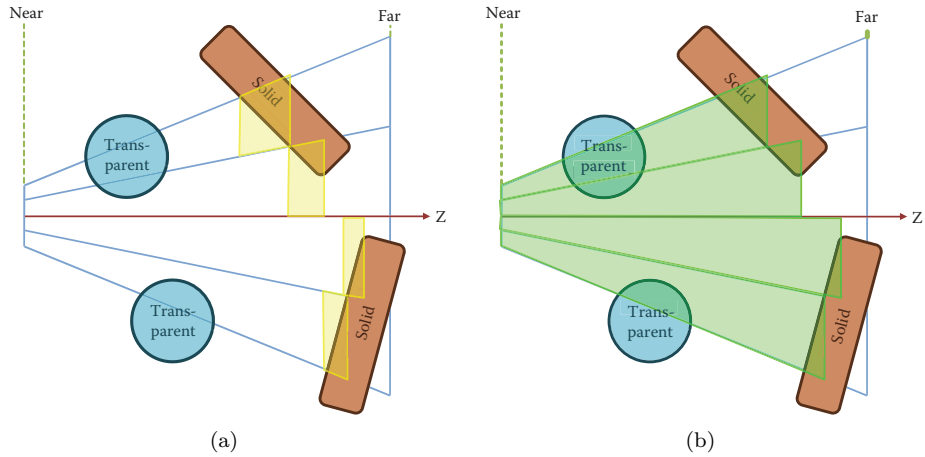


Figure 4.3. Z-bounds used for (a) opaque and (b) transparent geometries.

For transparent geometry, we would like to find the transparent objects' minimum Z-value and the minimum of the transparent objects' and opaque objects' respective maximum Z-values. However, this is somewhat cumbersome, requiring several passes over the transparent geometry; therefore, we simply use the maximum Z-value from the opaque geometry to cap the tiles in the far direction. This discards lights that are hidden by opaque geometry. In the near direction, we extend the tiles to the camera's near plane, as shown in Figure 4.3(b).

Using separate bounds turned out to be slightly faster than using the same tile bounds for both opaque and transparent geometry; in Figure 4.1(b), when using separate bounds, rendering takes 15.1 ms (upper) and 21.9 ms (lower), compared to 16.1 ms and 23.5 ms when using the extended bounds for both opaque and transparent geometries.

We would like to note that this is, again, scene dependent. Regardless of whether we use the approximate variant or the exact one, we can still use the depth buffer from the opaque geometry during the final render in order to enable early-Z and similar optimizations. If we do not use the minimum or maximum reduction to learn a tile's actual bounds, no modifications are required to support transparency.

Light assignment. If separate sets of tiles are used, light assignment must be done twice. In our case, a special optimization is possible: we can first assign lights in two dimensions and then discard lights that lie behind opaque geometry (use the maximum Z-bound from the tiles only). This yields the light lists for transparent geometry. For opaque geometry, we additionally discard lights based on the minimum Z-bound information (Listing 4.2).

```
// assign lights to 2D tiles
tiles2D = build_2d_tiles();
lightLists2D = assign_lights_to_2d_tiles( tiles2D );

// draw opaque geometry in pre-Z pass and find tiles'
// extents in the Z-direction
depthBuffer = render_preZ_pass();
tileZBounds = reduce_z_bounds( tiles2D, depthBuffer );

// for transparent geometry, prune lights against maximum Z-direction
lightListsTrans
  = prune_lights_max( lightLists2D, tileZBounds );

// for opaque geometry additionally prune lights against
// minimum Z-direction
lightListsOpaque
  = prune_lights_min( lightListsTrans, tileZBounds );

// ...

// later: rendering
draw( opaque geometry, lightListsOpaque );
draw( trasparent geometry, lightListsTrans );
```

Listing 4.2. Pseudocode describing the rendering algorithm used to support transparency, as shown in Figure 4.1(b). We perform the additional optimization where we first prune lights based on the maximum Z-direction, which gives the light assignment for transparent geometry. Then, we prune lights in the minimum Z-direction, which gives us light lists for opaque geometry.

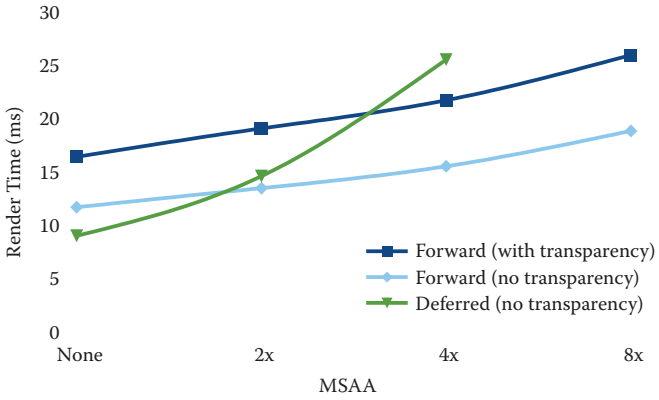
Rendering and shading. No special modifications are required here, other than using the appropriate set of light lists for opaque and transparent geometries, respectively. First, all opaque geometry should be rendered. Then the transparent geometry is rendered back to front.[1]

4.6 Support for MSAA

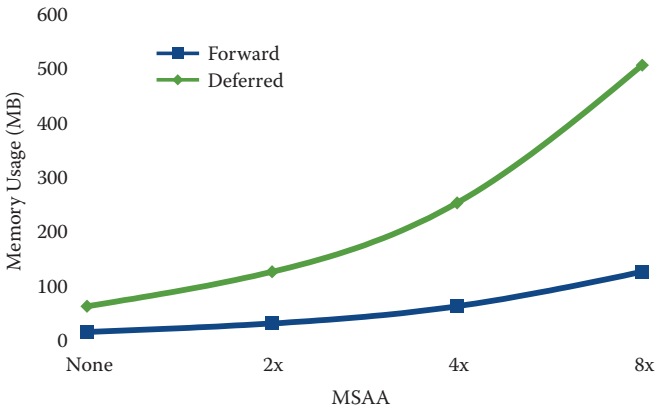
Supporting MSAA and similar schemes is very simple with tiled forward shading. We mainly need to ensure that all render targets are created with MSAA enabled. Additionally, we need to consider all (multi)samples during the optional minimum or maximum Z-reduction step.

We show the effect of MSAA on render time in Figure 4.4. As we compare to tiled deferred shading, which does not support transparency, Figure 4.4 includes timings for tiled forward both with (Figure 4.1(b)) and without (Figure 4.1(a)) transparency. Additionally, we compare memory usage between our forward and deferred implementations.

[1] In our demo, we sort on a per-object basis, which obviously causes some artifacts when transparent objects overlap. This is not a limitation in the technique but rather one in our implementation.



(a) Render time



(b) Memory usage

Figure 4.4. (a) Render time and (b) memory usage for tiled forward and tiled deferred shading with varying MSAA settings. We were unable to allocate the 8× MSAA framebuffer for deferred, which is why no timing results are available for that configuration. Memory usage estimates are based on a G-buffer with 32-bit depth, 32-bit ambient, and 64-bit normal, diffuse, and specular components (using the `RGBA16F` format).

One interesting note is that our unoptimized Z-reduction scales almost linearly with the number of samples: from 0.75 ms when using one sample to 5.1 ms with 8× MSAA. At that point, the contribution of the Z-reduction is quite significant with respect to the total frame time. However, it still provides a speedup in our tests. It is also likely possible to optimize the Z-reduction step further, for instance, by using compute shaders instead of a fragment shader.

4.7 Supporting Different Shaders

Like all forward rendering, we can attach different shaders and resources (textures, uniforms, etc.) to different chunks of geometry. Of course, if desired, we can still use the übershader approach in the forward rendering.

We have implemented three different shader types to test this, as seen in Figure 4.1(d): a default diffuse-specular shader, a shader emulating two-color car paint (see transparent bubbles and lion), and a rim-light shader (see large fabric in the middle of the scene).

The forward renderer uses the different shaders, compiled as different shader programs, with different chunks of geometry. For comparison, we implemented this as an übershader for deferred rendering. An integer identifying which shader should be used is stored in the G-buffer for each sample. (There were some unused bits available in the G-buffer, so we did not have to allocate additional storage.) The deferred shading code selects the appropriate shader at runtime using runtime branches in GLSL.

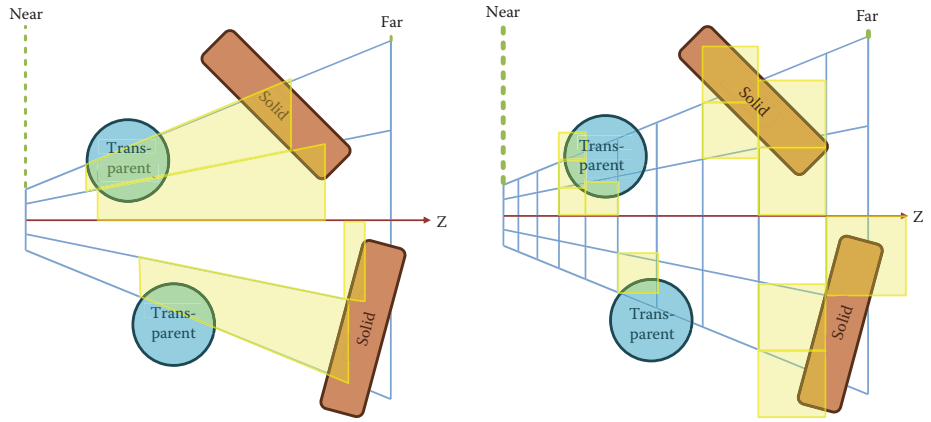
Performance degradation for using different shaders seems to be slightly smaller for the forward renderer; switching from diffuse-specular shading only to using the different shaders described above caused performance to drop by 1.4 ms on average. For the deferred shader, the drop was around 2.2 ms. However, the variations in rendering time for different views are in the same order of magnitude.

4.8 Conclusion and Further Improvements

We have explored *tiled forward shading* in this chapter. Tiled forward shading combines advantages from both tiled deferred shading and standard forward rendering. It is quite adaptable to different conditions, by, for instance, omitting steps in the algorithm made unnecessary by application-specific knowledge. An example is the optional computation of minimum and/or maximum Z-bounds for top-down views.

An extension that we have been exploring recently is *clustered shading*. Tiled shading (both forward and deferred) mainly considers 2D groupings of samples, which, while simple, cause performance and robustness issues in some scene and view configurations. One example of this is in scenes with first-person-like cameras where many discontinuities occur in the Z-direction (Figure 4.5). In clustered shading, we instead consider 3D groupings of samples, which handle this case much more gracefully.

Clustered shading's main advantage is a much lower view dependency, delivering more predictable performance in scenes with high or unpredictable complexity in the Z-direction. The disadvantages are increased complexity, requirements on hardware (we rely heavily on compute shaders/CUDA), and several new constant costs. For instance, with tiled shading, the subdivision of the screen into tiles is basically free. In clustered shading, this step becomes much more expensive—in



(a) Tiled shading (b) Clustered shading

Figure 4.5. Comparison between volumes created by (a) the tiling explored in this article and (b) clustering, as described in [Olsson et al. 12b]. Finding the tiled volumes is relatively simple and can be done in standard fragment shaders. Clustering is implemented with compute shaders, as is the light assignment to clusters.

fact, in some cases it offsets time won in the shading from the better light-to-sample mapping offered by clustering (Figure 4.6). We are also further exploring clustered forward shading [Olsson et al. 12a], which shows good promise on modern high-end GPUs with compute shader capabilities. Tiled forward shading, on the other hand, is implementable on a much wider range of hardware.

(a) (b)

Figure 4.6. Comparison between tiled forward shading and clustered forward shading. (a) In the top-down view, tiled forward outperforms our current clustered forward implementation (6.6 ms versus 9.3 ms). (b) In the first-person-like view, tiled forward becomes slightly slower (9.4 ms versus 9.1 ms). While somewhat slower in the first view, one of the main features of clustered shading is its robust performance. There are 1,024 randomly placed light sources.

Bibliography

[Andersson 09] Johan Andersson. "Parallel Graphics in Frostbite - Current & Future." SIGGRAPH Course: Beyond Programmable Shading, New Orleans, LA, August 6, 2009. (Available at http://s09.idav.ucdavis.edu/talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf.)

[Balestra and Engstad 08] Christophe Balestra and Pål-Kristian Engstad. "The Technology of Uncharted: Drake's Fortune." Presentation, Game Developer Conference, San Francisco, CA, 2008. (Available at http://www.naughtydog.com/docs/Naughty-Dog-GDC08-UNCHARTED-Tech.pdf.)

[Enderton et al. 10] 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*, pp. 157–164. New York: ACM, 2010.

[Engel 09] 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, New Orleans, LA, August 3, 2009. (Available at http://www.bungie.net/News/content.aspx?type=topnews&link=Siggraph_09.)

[Hargreaves and Harris 04] Shawn Hargreaves and Mark Harris. "Deferred Shading." Presentation, NVIDIA Developer Conference: 6800 Leagues Under the Sea, London, UK, June 29, 2004. (Available at http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf.)

[Kircher and Lawrance 09] 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*, pp. 39–45. New York: ACM, 2009.

[Lauritzen 10] Andrew Lauritzen. "Deferred Rendering for Current and Future Rendering Pipelines." SIGGRAPH Course: Beyond Programmable Shading, Los Angeles, CA, July 29, 2010. (Available at http://bps10.idav.ucdavis.edu/talks/12-lauritzen_DeferredShading_BPS_SIGGRAPH2010.pdf.)

[Olsson and Assarsson 11] Ola Olsson and Ulf Assarsson. "Tiled Shading." *Journal of Graphics, GPU, and Game Tools* 15:4 (2011), 235–251. (Available at http://www.tandfonline.com/doi/abs/10.1080/2151237X.2011.621761.)

[Olsson et al. 12a] Ola Olsson, Markus Billeter, and Ulf Assarsson. "Clustered and Tiled Forward Shading: Supporting Transparency and MSAA." In *SIGGRAPH '12: ACM SIGGRAPH 2012 Talks*, article no. 37. New York: ACM, 2012.

[Olsson et al. 12b] Ola Olsson, Markus Billeter, and Ulf Assarsson. "Clustered Deferred and Forward Shading." In *HPG '12: Proceedings of the Fourth ACD SIGGRPAH/Eurographics Conference on High Performance Graphics*, pp. 87–96. Aire-la-Ville, Switzerland: Eurogaphics, 2012.

[Saito and Takahashi 90] Takafumi Saito and Tokiichiro Takahashi. "Comprehensible Rendering of 3D Shapes." *SIGGRAPH Comput. Graph.* 24:4 (1990), 197–206.

[Shishkovtsov 05] Oles Shishkovtsov. "Deferred Shading in S.T.A.L.K.E.R." In *GPU Gems 2*, edited by Matt Pharr and Randima Fernando, pp. 143–166. Reading, MA: Addison-Wesley, 2005.

[Swoboda 09] Matt Swoboda. "Deferred Lighting and Post Processing on PLAYSTATION 3." Presentation, Game Developer Conference, San Francisco, 2009. (Available at http://www.technology.scee.net/files/presentations/gdc2009/DeferredLightingandPostProcessingonPS3.ppt.)

[Thibieroz and Grün 10] Nick Thibieroz and Holger Grün. "OIT and GI Using DX11 Linked Lists." Presentation, Game Developer Conference, San Francisco, CA, 2010. (Available at http://developer.amd.com/gpu_assets/OIT%20and%20Indirect%20Illumination%20using%20DX11%20Linked%20Lists_forweb.ppsx.)

[Trebilco 09] Damian Trebilco. "Light Indexed Deferred Rendering." In *ShaderX7: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 243–256. Hingham, MA: Charles River Media, 2009.