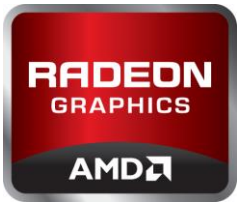

TiledLighting11

AMD Developer Relations

Overview

This sample provides an example implementation of two tile-based light culling methods: Forward+ (or Tiled Forward) and Tiled Deferred. Both methods support high numbers of dynamic lights while maintaining performance. They utilize a Direct3D 11 compute shader (DirectCompute 5.0) to divide the screen into tiles and quickly cull lights against those tiles. For Forward+, the resulting per-tile light lists are written out to a buffer for the forward pixel shader to use when lighting. For Tiled Deferred, lighting is done in the same compute shader that performs light culling.

In addition to non-shadow-casting point lights that are typically seen in tile-based light culling demos, this sample supports non-shadow-casting spot lights, as well as shadow-casting lights (point and spot). Moreover, it extends tiled light culling to work with alpha-blended geometry. It can also spawn virtual point lights (VPLs) to approximate one-bounce global illumination, as seen in AMD's Leo Demo [AMDLeoDemo12][McKee12].

Forward+ Implementation

The Forward+ implementation is based on [Harada12] and consists of three steps:

1. Depth pre-pass
2. Tiled light culling
3. Forward shading, using the per-tile light lists from Step 2

Depth Pre-Pass

A depth pre-pass is commonly used in forward rendering to avoid executing the pixel shader on a pixel that is later overwritten. This is especially important for Forward+, because the shader may be looping over many lights. For performance reasons, it is important to avoid paying that cost for pixels that will be invisible in the final image.

For Forward+, the depth pre-pass also serves to produce the data needed to calculate min and max depth per tile during light culling. We will discuss the use of the min and max depth in more detail in the Tiled Light Culling section.

When performing a depth pre-pass, be sure to enable depth-only rendering by setting a null color buffer with `OMSetRenderTargets`. Depth-only rendering is much faster than standard rendering. Also, set a null pixel shader for opaque geometry and a simple discard pixel shader for alpha test geometry.

Tiled Light Culling

The key to the Forward+ algorithm is the tiled light culling step. During this step, the screen is divided into fixed-size tiles, and a DirectCompute 5.0 compute shader is used to calculate a list of lights for each tile.

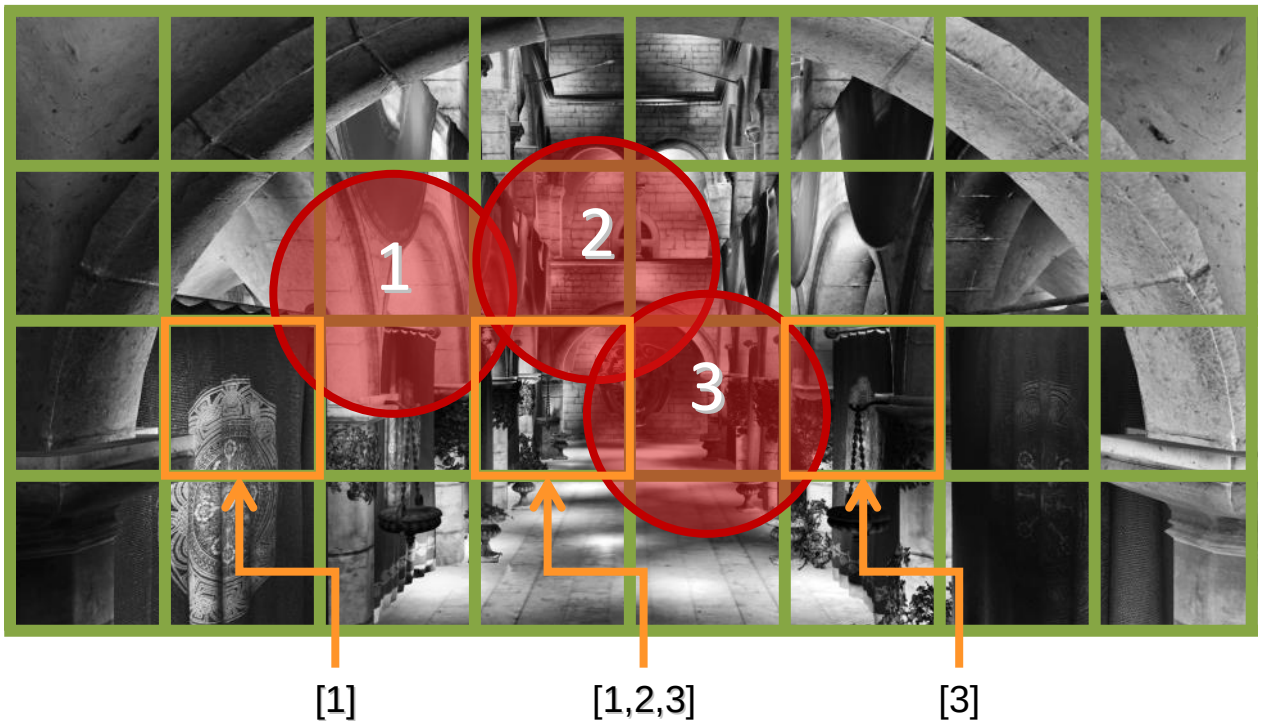

Figure 1: Simplified illustration of screen-space tiles

Figure 1 gives a simplified illustration of this process. To perform the culling, a compute shader thread group is executed per tile. For example, a 16x16 thread group configuration corresponds to a 16x16-pixel tile size. Each tile (i.e. each thread group) builds a per-tile list of lights that intersect that tile. The per-tile lists store light indices into the global light list.

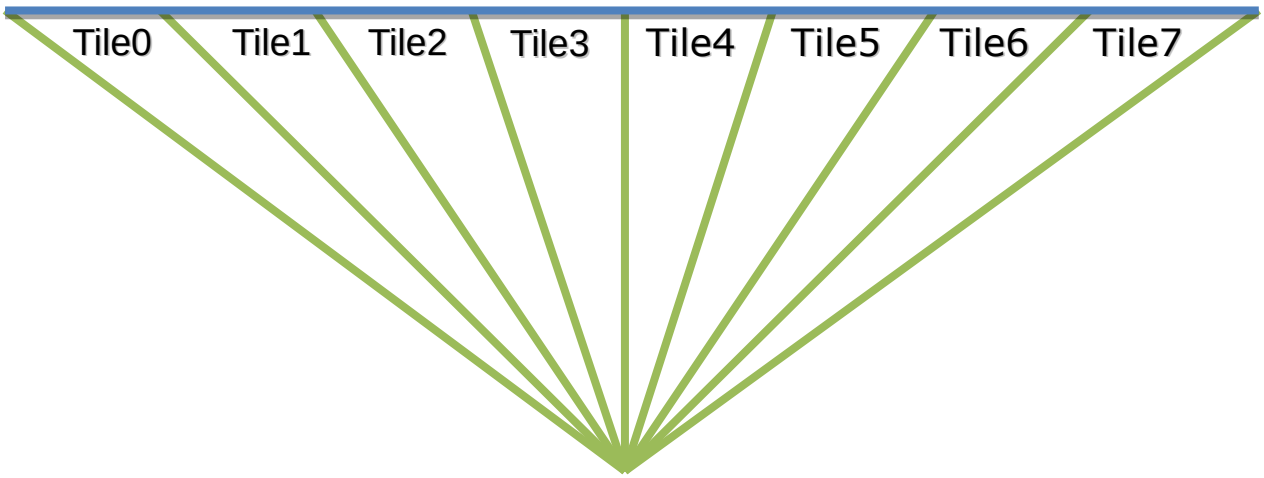

Figure 2: Top-down 2D illustration of screen-space tiles, showing how the view frustum is divided

Figure 2 shows how the view frustum is divided to perform the per-tile culling. Each compute shader thread group calculates an asymmetric frustum in view space that fits around the tile.

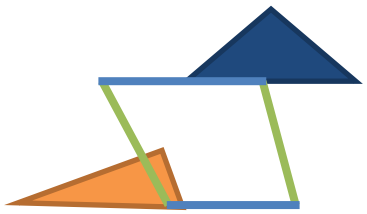

Figure 3: Bound the frustum with min and max depth

Then, to make the frustum fit more tightly around the scene geometry for the current frame, the min and max depth per tile is calculated using the depth buffer from the depth pre-pass. Figure 3 gives a simplified illustration of this process. Each thread in the thread group reads the depth for it's corresponding screen pixel, and atomic operations are used to calculate a thread-safe min and max. This smaller frustum is then used to build the per-tile light list by testing each light's bounding sphere against the frustum.

For each light that intersects the frustum, the compute shader performs a thread-safe write (using atomics) of the index of that light to the per-tile list in thread group shared memory. Once all threads in the thread group have finished culling lights, the list in thread group shared memory is written out to an Unordered Access View (UAV). Each tile is alloted a fixed area in the UAV, so that the start location of each list is known without any extra bookkeeping. The light count is stored at the beginning of each list.

Forward Shading

Forward shading for Forward+ is the same as classic forward rendering, except that the light list is now per tile from the compute shader, instead of per object in shader constants. Ease of implementation is one of the attractive qualities of the Forward+ algorithm.

Tiled Deferred Implementation

The Tiled Deferred algorithm consists of two steps:

1. Render to G-Buffer
2. Tiled light culling and shading

Render to G-Buffer

Building the G-Buffer for Tiled Deferred is the same as classic deferred rendering. Scene geometry is rendered and material properties are stored into the G-Buffer using multiple render targets (MRTs).

Tiled Light Culling and Shading

Tiled light culling for Tiled Deferred is identical to that used by the Forward+ algorithm. Refer to the Tiled Light Culling section above for more details. Unlike Forward+, however, the Tiled Deferred algorithm performs shading in the same compute shader after tiled light culling is complete. This means that the results of culling are not written out to a UAV, but rather the per-tile list in thread group shared memory is used directly

to iterate over the lights. Shading is done using the material properties stored in the G-Buffer.

The Tiled Deferred implementation in this sample is based on previous implementations described in [Lauritzen10] and [Andersson11]. In particular, MSAA support is implemented by building a list of edge pixels for the current tile in thread group shared memory. The threads are then redistributed to efficiently perform the per-sample shading only on those edge pixels. See [Lauritzen10] and [Andersson11] for more details.

The Tiled Deferred compute shader writes out to an oversized UAV when MSAA is enabled. That is, since MSAA textures cannot be bound as UAVs in Direct3D 11, the sample instead creates a larger, non-MSAA texture (times 2 in each dimension for 4x MSAA). The compute shader then calculates the correct write address for the UAV from the screen pixel coordinate and desired sub-sample [Pettineo12].

Shadow-Casting Lights

This sample supports shadow-casting point and spot lights.

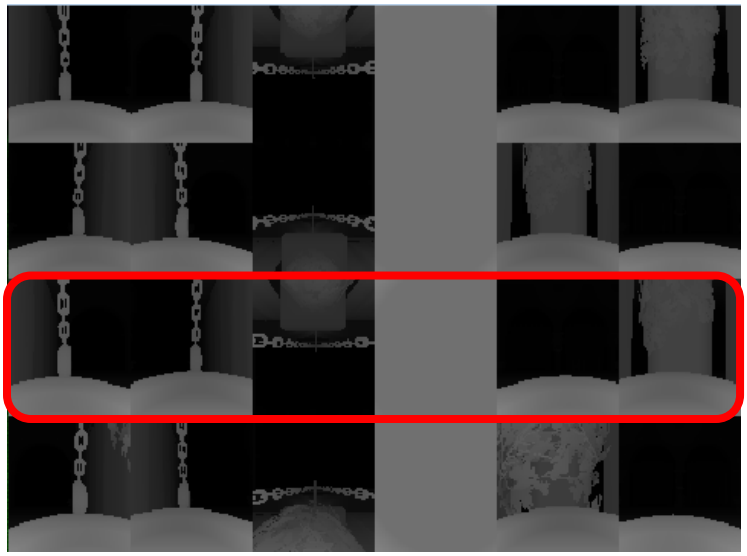

Figure 4: Texture atlas storing the shadow maps for multiple shadow-casting point lights. Each row of the atlas represents the cube map for a point light shadow map.

Shadow maps for multiple lights are stored in a 2D texture atlas. Figure 4 shows four shadow-casting point lights stored in a single texture atlas. Each row of the atlas stores the six faces of a cube map. The texture coordinate is scaled and biased to not read off the cube map face. The field of view (fov) of the cube map face render should also be adjusted, but you can typically get away with skipping this step.

The shadow maps for shadow-casting spot lights are also stored in a 2D texture atlas. The spot light case is simpler, however, because each spot light just requires a single shadow map in the atlas, not the six faces of a cube map.

Alpha-Blended Geometry

This sample supports forward rendering of alpha-blended geometry using tiled light culling.

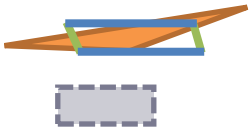

Figure 5: The frustum used for opaque cannot also be used for alpha-blended

The per-tile light lists for the opaque scene cannot be reused for alpha-blended geometry, because the culling frustum min and max was calculated from opaque scene depth only. For example, as seen in Figure 5, the frustum for the opaque scene might fail to capture lights that fall on the alpha-blended geometry represented by the gray box.

To get the necessary depth values, the sample performs depth-only rendering of alpha-blended geometry into a separate depth buffer. It then runs a slightly different version of the tiled culling compute shader that reads from both depth buffers (opaque and alpha-blended). The frustum min is taken from the alpha-blended depth buffer, while the frustum max is taken from the opaque depth buffer. This frustum in used to build a separate per-tile light list which is written out to a separate UAV. Forward shading with two-sided lighting is then used to render the alpha-blended geometry.

This technique is a natural extension to the Forward+ algorithm. For Tiled Deferred, the sample just uses this same forward rendering technique. That is, as is typical for deferred renderers, it takes the “just use forward rendering for transparency” approach.

Virtual Point Lights

The ability to efficiently render very large numbers of lights can be leveraged to approximate first bounce indirect light using a set of procedurally generated point lights (Virtual Point Lights or VPLs).

The first step in generating the VPLs is to render out a Reflective Shadow Map (RSM) [DS05]. Rendering is from the point of view of the direct light, much like a shadow map, only this time an MRT setup is used to store depth, normals, and albedo color.

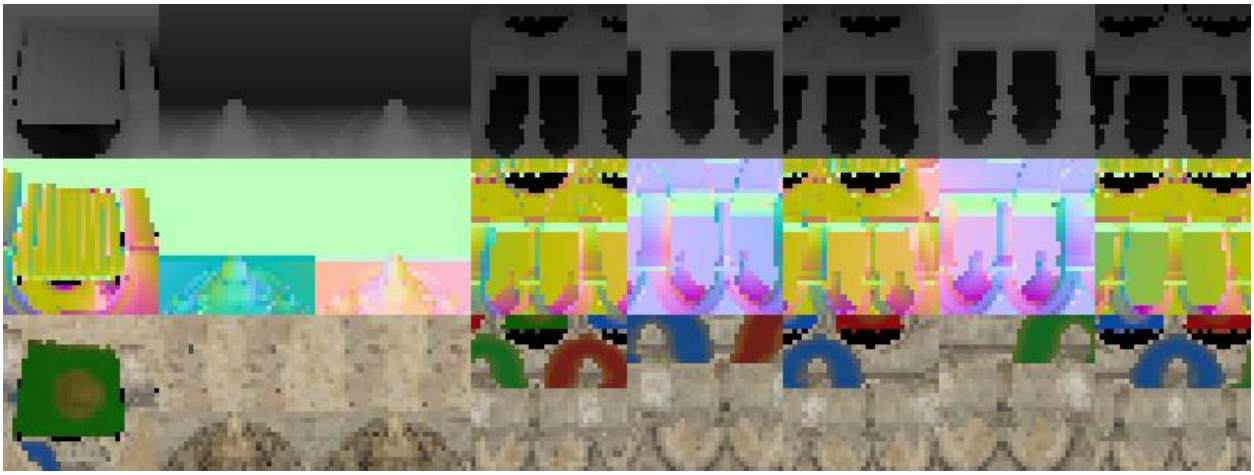

Figure 6: Each render target in the MRT setup is a texture atlas, to allow storing RSMs for several lights

Multiple lights are supported by using a texture atlas in each render target of the MRT setup. For example, in Figure 6, the first row contains an atlas of the depth values for eight separate spot lights. Similarly, the second row contains the normals and the third contains the albedo color.

Next, a compute shader generates VPLs per texel or per block of texels on the RSM. The heuristics to generate the VPLs are kept intentionally simple in this sample, but a commercial title should consider a more sophisticated algorithm for generating an effective set of VPLs. The VPLs are added to a list using a structured buffer and IncrementCounter().

The VPLs are subsequently treated as another light type for light culling and shading. That is, they get culled and put into a separate per-tile list in thread group shared memory. Similar to other light types, that list is written out to a UAV in Forward+ for use by the forward shading step, while the list is used directly in thread group shared memory for Tiled Deferred. Shading for VPLs is a specialized version of point lighting.

Optimizations

This section briefly covers some of the optimizations in the sample.

The first optimization improves performance when a tile contains a depth discontinuity.

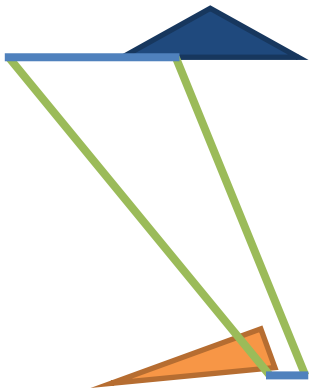

Figure 7: Depth discontinuities can cause large tile frusta

Figure 7 shows an example of a depth discontinuity resulting in a large frustum for the tile. This can lead to false positive intersections, where lights intersect the tile frustum but do not actually affect any scene geometry in that tile.

The sample implements the “half z” optimization to improve performance in these situations. The frustum is divided into two regions at the halfway point of the depth range (hence, “half z”). Then two light lists are created per tile, one for each region. During shading, only one list is chosen to loop over, based on the depth of the pixel being shaded.

Another optimization is simply to keep the light data required for culling (position and radius) in a separate buffer from other light data (color and spot light parameters, for example). Otherwise, when the light culling compute shader runs, it will be polluting the cache with unnecessary data. This reduction in cache efficiency can negatively impact performance, particularly at high light counts.

Performance Comparison

The answer to the question of which is the faster tile-based light culling method is, “it depends.”

Figure 8: Sliders for performance comparison

Forward+ performance depends on the geometry load during the depth pre-pass and forward shading. Since geometry is submitted twice for Forward+, high triangle density is more expensive in Forward+ than Tiled Deferred.

In the sample, in addition to the Sponza scene, extra grid objects are rendered to increase the geometry load. The sample contains two sliders to adjust this: Triangle Density and Active Grid Objects. The Triangle Density slider changes the number of triangles in each grid object: 484 for low, 1764 for medium, and 3600 for high. The Active Grid Objects slider changes the number of grid objects rendered in the scene. With a max of 280 grid objects, you can add 135,520 triangles at low, 493,920 at medium, and 1,008,000 at high. The Sponza scene contains roughly 250,000 triangles. Thus, you can have a maximum of approximately 1.25 million triangles.

Tiled Deferred performance depends on the size of the G-Buffer. As the shading model in the sample is fairly simple, the sample only needs to two 8888 render targets for its G-Buffer. Commercial titles that use deferred rendering often have 3 render targets, or possibly more. The sample creates extra dummy render targets to simulate the performance impact of larger G-Buffers. The “render to G-Buffer” step writes white out to the extra G-Buffers, and the shading step reads these extra G-Buffers and modulates the shading by the value (where multiplying a color by white leaves the original color unchanged). Use the Active G-Buffer RTs slider to change the G-Buffer size.

GPU memory bandwidth will determine the performance impact of larger G-Buffers. For example, the AMD Radeon™ HD 7970 GHz Edition has 288GB/s memory bandwidth and can absorb the “abuse” of larger G-Buffers better than mid-range and low-end GPUs. Consider testing on a mid-range card to better gauge the impact of different G-Buffer sizes.

Whether MSAA is enabled will also impact performance. Forward+ can use the built-in hardward MSAA features, where shading is done only once per pixel, with results written out to an MSAA render target based on triangle coverage. Tiled Deferred MSAA requires a software implementation (in the culling and shading compute shader), and it requires shading each subsample where edges are detected. Thus, MSAA is more expensive in Tiled Deferred than Forward+. Note that the sample has tuned its edge detection in an attempt to detect the minimum amount of edge pixels while still providing comparable quality to the Forward+ MSAA result.

Related to the above, the expense of the light loop will change the performance characteristics. A more expensive light loop (when shadows are enabled, for example)

may tend to impact Tiled Deferred more with MSAA enabled, since it has to shade subsamples.

And of course, it all depends on the number of lights in the scene.

Thus, final performance depends on a number of factors.

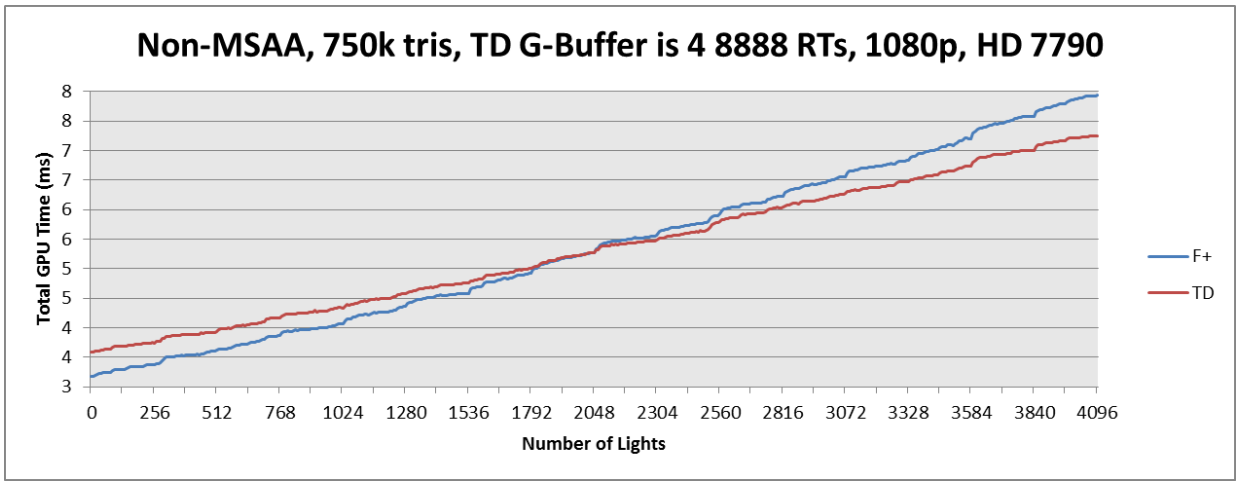

Figure 9: Performance comparison of Forward+ vs. Tiled Deferred, MSAA disabled, approximately 750k triangles, Tiled Deferred G-Buffer is 4 render targets, transparency, shadows, and VPLs are disabled

Figure 9 shows one example of the many scenarios you can compare with the sample. In this example, an AMD Radeon™ HD 7790 is used (note that's 7790, not 7970) to compare performance with MSAA disabled, medium triangle density, and 4 G-Buffer RTs. This uses random lights, with transparent objects, shadows, and VPLs disabled.

The chart shows that Forward+ is faster up to around 1792 lights, with Tiled Deferred eventually pulling ahead at higher light counts.

Other example scenarios include:

- With MSAA enabled on mid-range or lower cards, Forward+ wins more convincingly.
- Conversely, with MSAA disabled, high triangle density, and a smaller G-Buffer, Tiled Deferred comes out ahead.

See [ST13] for more performance comparisons between Forward+ and Tiled Deferred. And of course, use the sample to experiment.

Regarding Tiled Deferred MSAA performance, there is an optimization in the AMD Direct3D 11 driver that can improve performance when reading MSAA render targets. To ensure that the sample runs with this optimization enabled, install Catalyst 13.9 WHQL or 13.10 Beta (or later). If Tiled Deferred MSAA is slower than expected in your experiments on AMD hardware, outdated drivers is likely the reason. Note that the performance data in [ST13] was gathered with this optimization enabled.

References

- [AMDLeoDemo12] Leo Demo, http://developer.amd.com/resources/documentation-articles/samples-demos/gpu-demos/amd-radeon-hd-7900-series-graphics-real-time-demos/
- [Andersson11] Johan Andersson, “DirectX 11 Rendering in Battlefield 3”, GDC 2011
- [DS05] Carsten Dachsbacher & Marc Stamminger, “Reflective Shadow Maps”, SIGGRAPH 2005
- [Harada12] Takahiro Harada et al, “Forward+: Bringing Deferred Lighting to the Next Level”, Short Papers, Eurographics 2012
- [Lauritzen10] Andrew Lauritzen, “Deferred Rendering for Current and Future Rendering Pipelines”, Beyond Programmable Shading, SIGGRAPH 2010
- [McKee12] Jay McKee, “Technology Behind AMD’s Leo Demo”, GDC 2012
- [Pettineo12] Matt Pettineo, “Light Indexed Deferred Rendering”, http://mynameismjp.wordpress.com/2012/03/31/light-indexed-deferred-rendering/
- [ST13] Jason Stewart & Gareth Thomas, “Tiled Rendering Showdown: Forward++ vs. Deferred Rendering”, GDC 2013

Advanced Micro Devices
One AMD Place
P.O. Box 3453
Sunnyvale, CA 94088-3453

http://www.amd.com
http://developer.amd.com