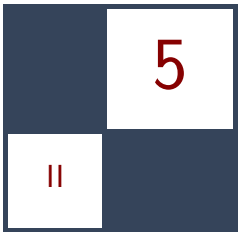


Forward+: A Step Toward Film-Style Shading in Real Time

Takahiro Harada, Jay McKee, and Jason C. Yang

5.1 Introduction

Modern GPU hardware along with the feature set provided by the DirectX 11 API provides developers more flexibility to choose among a variety of rendering pipelines. In order to exploit the performance of modern GPUs, we believe it is important to choose a pipeline that takes advantage of GPU hardware features, scales well, and provides flexibility for artists, tech artists, and programmers to achieve high-quality rendering with unique visuals. The ability to differentiate a game's visual look from today's games, which modern computer-generated (CG) films are extremely good at doing, likely will be a key for game graphics in the future. However, the ability to produce high-quality renderings that approach the styling in CG films will require great flexibility to support arbitrary data formats and shaders for more sophisticated rendering of surface materials and special effects.

Our goal was to find a rendering pipeline that would best meet these objectives. We boiled things down to a few specific requirements:

- Materials may need to be both physically and nonphysically based. Tech artists will want to build large trees of materials made of arbitrary complexity. Material types will likely be similar to those found in offline renderers such as RenderMan, mental ray, and Maxwell Render shading systems.
- Artists want complete freedom regarding the number of lights that can be placed in a scene at once.
- Rendering data should be decoupled from the underlying rendering engine. Artists and programmers should be able to write shaders and new materials freely at runtime for quick turnaround—going from concept to seeing results

should be fast and easy. The architecture should be simple and not get in the way of creative expression.

We have devised a rendering pipeline that we believe meets these objectives well and is a good match for modern GPU hardware going into the foreseeable future. We refer to it as the Forward+ rendering pipeline [Harada et al. 11].

5.2 Forward+

The Forward+ rendering pipeline requires three stages:

- Z prepass. Z prepass is an option for forward rendering, but it is essential for Forward+ to reduce the pixel overdraws of the final shading step. This is especially expensive for Forward+ due to the possible traversal of many lights per pixel, which we will detail later in this section.

- Light culling. Light culling is a stage that calculates the list of lights affecting a pixel.

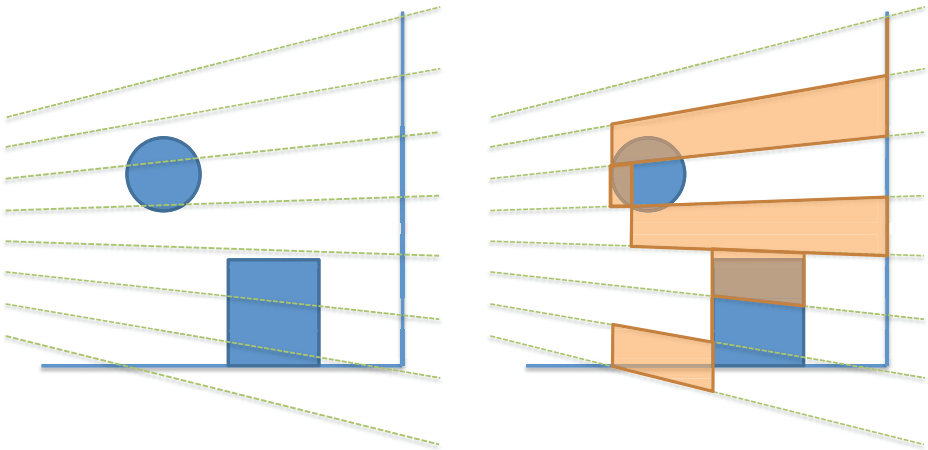
- Final shading. Final shading, which is an extension to the shading pass in forward rendering, shades the entire surface. A required change is the way to pass lights to shaders. In Forward+, any lights in a scene have to be accessible from shaders rather than binding some subset of lights for each objects as is typical of traditional forward rendering.

5.2.1 Light Culling

The light-culling stage is similar to the light-accumulation step of deferred lighting. Instead of calculating lighting components, light culling calculates a list of light indices overlapping a pixel. The list of lights can be calculated for each pixel, which is a better choice for final shading.

However, storing a per-pixel light list requires a large memory footprint and significant computation at the light-culling stage. Instead, the screen is split into tiles and light indices are calculated on a per-tile basis (Figure 5.1). Although tiling can add false positives to the list for a pixel in a tile, it reduces the overall memory footprint and computation time necessary for generating the light lists. Thus we are making a tradeoff between light-index buffer memory and final shader efficiency.

By utilizing the computing capability of modern GPUs, light culling can be implemented entirely on the GPU as detailed in Section 5.3. Therefore, the whole lighting pipeline can be executed entirely on the GPU.



(a) (b)

Figure 5.1. Illustration of light culling in 2D. (a) A camera is placed on the left, and green lines indicate tile borders. (b) Light culling creates a frustum for each tile bounded by minimum and maximum depth of pixels in a tile.

5.2.2 Shading

Whereas light culling creates the list of lights overlapping each pixel, final shading loops through the list of lights and evaluates materials using material parameters describing the surface properties of the rendered object along with information stored for each light. With unordered access view (UAV) support, per-material instance information can be stored and accessed in linear structured buffers passed to material shaders. Therefore, at least in theory, the full render equation can be satisfied without limitation because light accumulation and shading happen simultaneously in one place with complete material and light information.

Use of complex materials and more accurate lighting models to improve visual quality is not constrained other than by the GPU computational cost, which is largely determined by the average number of overlapping lights on each pixel multiplied by the average cost for material calculation.

With this method, high pixel overdraw can kill performance; therefore, a Z prepass is critical to minimize the cost of final shading.

5.3 Implementation and Optimization

A standard forward rendering pipeline can be converted to a Forward+ rendering pipeline by adding the light-culling stage and modifying existing pixel shaders to make them implement Forward+'s final shading stage as described in Section 5.2.

No modification is necessary for the Z prepass, so we do not describe its implementation. The light-culling stage can be implemented in several ways thanks to the flexibility of current GPUs. Specifically, direct compute and read-writable structure data buffers or UAVs are the key features to utilizing Forward+. In this section, we first describe which features of DirectX 11 are essential to making Forward+ work well on modern GPUs. Then we explain a light-culling implementation that works well for a scene with thousands of lights. If there are more lights, we might be better off considering other implementations such as those described in [Harada et al. 11]. This section concludes by describing modifications for final shading.

5.3.1 Gather-Based Light Culling

During light culling, the computation is done on a by-tile basis. Therefore, it is natural to execute a thread group for a tile. A thread group can share data using thread group shared memory (called shared memory from now on), which can reduce a lot of redundant computation in a thread group. The computation is identical for each tile; therefore, we explain the computation for a single tile.

The compute shader for light culling is executed as a two-dimensional (2D) work group. A thread group is assigned a unique 2D index, and a thread in a thread group is assigned a unique 2D index in the group.

In the pseudocode in this subsection, the following macros are used for these variables:

- `GET_GROUP_IDX`: thread group index in X direction (`SV_GroupID`);
- `GET_GROUP_IDY`: thread group index in Y direction (`SV_GroupID`);
- `GET_GLOBAL_IDX`: global thread index in X direction (`SV_DispatchThreadID`);
- `GET_GLOBAL_IDY`: global thread index in Y direction (`SV_DispatchThreadID`);
- `GET_LOCAL_IDX`: local thread index in X direction (`SV_GroupThreadID`);
- `GET_LOCAL_IDY`: local thread index in Y direction (`SV_GroupThreadID`).

The first step is computation of the frustum of a tile in view space. To reconstruct four side faces, we need to calculate the view-space coordinates of the four corner points of the tile. With these four points and the origin, four side planes can be constructed.

```
float4 frustum[4];
{  //  construct frustum
  float4 v[4];
  v[0]=projToView(8*GET_GROUP_IDX, 8*GET_GROUP_IDY,1.f) );
  v[1]=projToView(8*(GET_GROUP_IDX+1), 8*GET_GROUP_IDY,1.f) );
  v[2]=projToView(8*(GET_GROUP_IDX+1),8*(GET_GROUP_IDY+1),1.f));
```

```
  v[3]=projToView(8*GET_GROUP_IDX, 8*(GET_GROUP_IDY+1),1.f) );
  float4 o = make_float4(0.f,0.f,0.f,0.f);
  for(int i=0; i<4; i++)
    frustum[i] = createEquation( o, v[i], v[(i+1)&3] );
}
```

`projToView()` is a function that takes screen-space pixel indices and depth value and returns coordinates in view space. `createEquation()` creates a plane equation from three vertex positions.

The frustum at this point has infinite length in the depth direction; however, we can clip the frustum by using the maximum and minimum depth values of the pixels in the tile. To obtain the depth extent, a thread first reads the depth value of the assigned pixel from the depth buffer, which is created in the depth prepass. Then it is converted to the coordinate in view space. To select the maximum and minimum values among threads in a group, we used atomic operations to shared memory. We cannot use this feature if we do not launch a thread group for computation of a tile.

```
float depth = depthIn.Load(
uint3(GET_GLOBAL_IDX,GET_GLOBAL_IDY,0) );

float4 viewPos = projToView(GET_GLOBAL_IDX, GET_GLOBAL_IDY,
depth);

int lIdx = GET_LOCAL_IDX + GET_LOCAL_IDY*8;
{// calculate bound
  if( lIdx == 0 )// initialize
  {
    ldsZMax = 0;
    ldsZMin = 0xffffffff;
  }
  GroupMemoryBarrierWithGroupSync();
  u32 z = asuint( viewPos.z );
  if( depth != 1.f )
  {
    AtomMax( ldsZMax, z );
    AtomMin( ldsZMin, z );
  }
  GroupMemoryBarrierWithGroupSync();
  maxZ = asfloat( ldsZMax );
  minZ = asfloat( ldsZMin );
}
```

`ldsZMax` and `ldsZMin` store maximum and minimum z coordinates, which are bounds of a frustum in the z direction, in shared memory. Once a frustum is constructed, we are ready to go through all the lights in the scene. Because there are several threads executed per tile, we can cull several lights at the same time. We used 8×8 for the size of a thread group; thus, 64 lights are processed in parallel. The code for the test is as follows:

```
for(int i=0; i<nBodies; i+=64)
{
  int il = lIdx + i;
  if( il < nBodies )
  {
    if(overlaps(frustum, gLightGeometry[il]))
    {
      appendLightToList(il);
    }
   }
}
```

In `overlaps()`, a light-geometry overlap is checked against a frustum using the separating axis theorem [Ericson 04]. If a light is overlapping the frustum, the light index is stored to the list of the overlapping lights in `appendLightToList()`. There are several data structures we can use to store the light list. The obvious way would be to build a linked list using a few atomic operations [Yang et al. 10].

However, this approach is relatively expensive: we need to use a few global atomic operations to insert a light, and a global memory write is necessary whenever an overlapping light is found. Therefore, we took another approach in which a memory write is performed in two steps. A tile is computed by a thread group, and so we can use shared memory for the first level storage. Light index storage and counter for the storage is allocated as follows:

```
groupshared u32 ldsLightIdx[LIGHT_CAPACITY];
groupshared u32 ldsLightIdxCounter;
```

In our implementation, we set `LIGHT_CAPACITY` to 256. The `appendLightToList()` is implemented as follows:

```
void appendLightToList( int i )
{
  u32 dstIdx = 0;
  InterlockedAdd( ldsLightIdxCounter, 1, dstIdx );
  if( dstIdx < LIGHT_CAPACITY )
    ldsLightIdx[dstIdx] = i;
}
```

With this implementation, no global memory write is necessary until all the lights are tested.

After testing all the lights against a frustum, indices of lights overlapping that frustum are collected in the shared memory. The last step of the kernel is to write these to the global memory.

For the storage of light indices in the global memory, we allocated two buffers: `gLightIdx`, which is a memory pool for the indices, and `gLightIdxCounter`, which

is a memory counter for the memory pool. Memory sections for light indices for a tile are not allocated in advance. Thus, we first need to reserve memory in `gLightIdx`. This is done by an atomic operation to `gLightIdxCounter` using a thread in the thread group.

Once a memory offset is obtained, we just fill the light indices to the assigned contiguous memory of `gLightIdx` using all the threads in a thread group. The code for doing this memory write is as follows:

```
{  // write back
  u32 startOffset = 0;
  if( lIdx == 0 )
  {// reserve memory
    if( ldsLightIdxCounter != 0 )
      InterlockedAdd( gLightIdxCounter , ldsLightIdxCounter,
startOffset );

    ptLowerBound[tileIdx] = startOffset;
    ldsLightIdxStart = startOffset;
  }
  GroupMemoryBarrierWithGroupSync();
  startOffset = ldsLightIdxStart;

  for(int i=lIdx; i<ldsLightIdxCounter; i+=64)
  {
    gLightIdx[startOffset+i] = ldsLightIdx[i];
  }
}
```

This light-culling kernel reads light geometry (for spherical lights, that includes the location of the light and its radius). There are several options for the structure of the light buffer. Of course, we can pack light geometry and lighting properties, such as intensity and falloff, to a single structure. However, this is not a good idea for our light-culling approach because all the necessary data for the light culling is padded with light properties, which are not used in the light culling. A GPU usually reads data by page. Therefore, it is likely to transfer lighting properties as well as light geometry although they are not read by the kernel when this data structure is employed for the lights.

A better choice for the data structure is to separate the light geometry and lighting properties into two separate buffers. The light-culling kernel only touches the light geometry buffer, increasing the performance because we do not have to read unnecessary data.

5.3.2 Final Shading

For final shading, all objects in the camera frustum are rendered with their authored materials. This is different than forward rendering because we need to iterate through the lights overlapping each tile.

To write a pixel shader, we created "building blocks" of common operations for different shaders. This design makes it easy to write shaders, as we will show now. The most important building blocks are the following two, implemented as macros:

```
#define LIGHT_LOOP_BEGIN
  int tileIndex = GetTileIndex(screenPos);
  uint startIndex, endIndex;
  GetTileOffsets( tileIndex, startIndex, endIndex );


  for( uint lightListIdx = startIdx;
       lightListIdx < endIdx;
       lightListIdx++ )
  {
     int lightIdx = LightIndexBuffer[lightListIdx];
     LightParams directLight;
     LightParams indirectLight;

     if( isIndirectLight( lightIdx ) )
     {
       FetchIndirectLight(lightIdx , indirectLight);
     }
     else
     {
       FetchDirectLight( lightIndex, directLight );
     }
#define LIGHT_LOOP_END
  }
```

`LIGHT_LOOP_BEGIN` first calculates the tile index of the pixel using its screen-space position. Then it opens a loop to iterate all the lights overlapping the tile and fills light parameters for direct and indirect light. `LIGHT_LOOP_END` is a macro to close the loop.

By using these building blocks, an implementation of a pixel shader is simple and looks almost the same as a pixel shader used in forward rendering. For example, a shader for a microfacet surface is implemented as follows:

```
float4 PS ( PSInput i ) : SV_TARGET
{
  float3 colorOut = 0;
#LIGHT_LOOP_BEGIN
  colorOut += EvaluateMicrofacet ( directLight, indirectLight );
#LIGHT_LOOP_END
  return float4(colorOut, 1.f );
}
```

Other shaders can be implemented by just changing the lines between the two macros. This building block also allows us to change the implementation easily

based on performance needs. For instance, we can change `LIGHT_LOOP_BEGIN` to iterate a few lights on a slower platform.

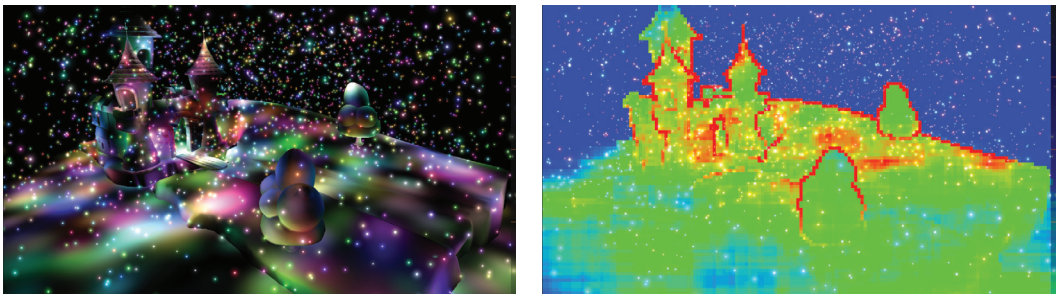
An optimization we can do for the host side is to sort all render draw calls by material type and render all triangles that belong to each unique material at the same time. This reduces GPU state change and makes good use of the cache because all pixels needing the same data will be rendered together.

5.4 Results

We implemented Forward+ using DirectX 11 and benchmarked using the scene shown in Figure 5.2 to compare the performance of Forward+ to compute-based deferred lighting [Andersson 11].

In short, Forward+ was faster on both the AMD Radeon HD 6970 and HD 7970 (Figure 5.3). Once we compare the memory transfer size and the amount of computing, it makes sense. Three timers are placed in a frame of the benchmark to measure time for prepass, light processing, and final shading. In Forward+, these three are depth prepass, light culling, and final shading. In compute-based deferred, they are geometry pass (or G-pass), which exports geometry information to full screen buffers, light culling, screen-space light accumulation, and final shading.

Prepass. Forward+ writes a screen-sized depth buffer while deferred writes a depth buffer and another `float4` buffer that packs the normal vector of the visible pixel. The specular coefficient can be stored in the W component of the buffer, too. Therefore, Forward+ writes less than deferred and is faster on prepass.



(a) (b)

Figure 5.2. A scene with 3,072 dynamic lights rendered in 1,280 × 720 resolution. (a) Using diffuse lighting. (b) Visualization of number of lights overlapping each tile. Blue, green and red tiles have 0, 25, and 50 lights, respectively. The numbers in between are shown as interpolated colors. The maximum number is clamped to 50.

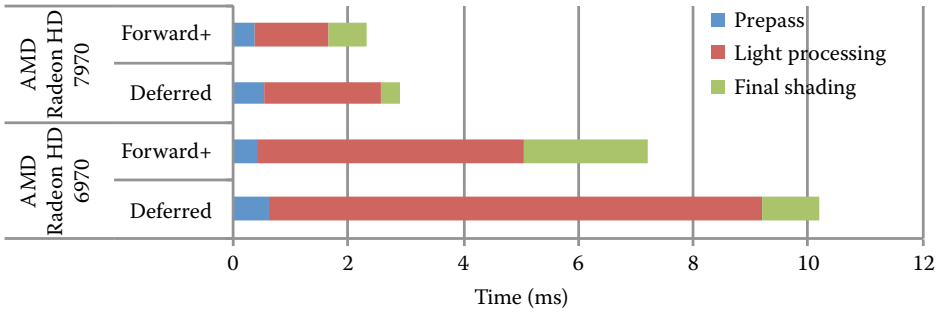


Figure 5.3. Breakdown of the computation time for three stages of Forward+ and deferred on an AMD Radeon HD 6970 GPU and an AMD Radeon HD 7970 GPU.

Light processing. Forward+ reads the depth and light geometry buffers. Deferred also reads them, but the `float4` buffer storing normal vectors and lighting properties has to be read as well because lighting is done at this stage. Therefore, Forward+ has less memory read compared to deferred.

As for the amount of the computation, Forward+ culls lights. On the other hand, deferred not only culls lights but also performs lighting computation. Forward+ has less computation.

For the memory write, Forward+ writes light indices, the sizes of which depend on the scene and tile size. If 8×8 tiles are used, deferred has to write $8 \times 8 \times 4$ bytes if a `float4` data is written for each pixel. With this data size, Forward+ can write 256 ($8 \times 8 \times 4$) light indices for a tile; if the number of lights is less than 256 per tile, Forward+ writes less. In our test scene, there was no tile overlapped with more than 256 lights.

To summarize this stage, Forward+ is reading, computing, and writing less than deferred. This is why Forward+ is so fast at this stage.

Final shading. It is obvious that Forward+ takes more time compared to deferred at shading because it has to iterate through all the lights in the pixel shader. This is a disadvantage in terms of the performance, but it is designed this way to get more freedom.

5.5 Forward+ in the AMD Leo Demo

We created the AMD Leo demo to show an implementation of Forward+ in real-time in a real-world setting. A screenshot from the demo is shown in Figure 5.4. We chose scene geometry on the order of what can be found in current PC-based video games (one to two million polygons). We also had the objective of rendering with a unique stylized look that could be characterized as "CGish" in that it uses material types that resemble those found in an offline renderer. There are more

Figure 5.4. A screenshot from the AMD Leo Demo.

than 150 lights in the scenes. Artists created about 50 lights by hand. Other lights are dynamically spawned at runtime for one-bounce indirect illumination lighting using the technique described in this section. Although Forward+ is capable of using thousands of dynamic lights, a few hundred lights were more than enough for our artists to achieve their lighting goals, especially for a single-room indoor scene.

We use a material system in which a material consists of N layers where each layer can have M weighted BRDF models along with other physically based constants like those involving transmission, absorption, refraction, and reflections of incoming light.

Material parameters for a single layer include physical properties for lighting such as coefficients for a microfacet surface and a refractive index as well as many modifiers for standard lighting parameters. We deliberately allow numeric ranges to go beyond the "physically correct" values to give artists freedom to bend the rules for a given desired effect.

For lighting, artists can dynamically create and place any number of omnidirectional lights and spotlights into a scene. The light data structure contains a material index mask. This variable is used to filter lights to only effect specific material types. While not physically correct, this greatly helps artists fine-tune lighting without unwanted side effects.

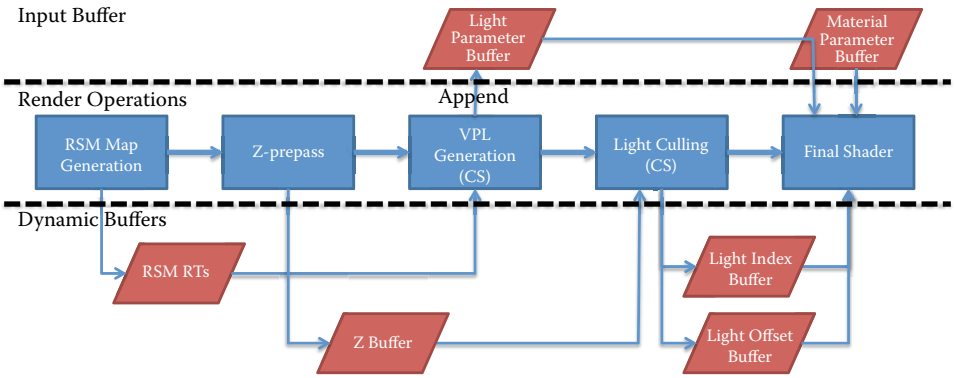


Figure 5.5. Forward+ render passes and GPU buffers in the AMD Leo Demo.

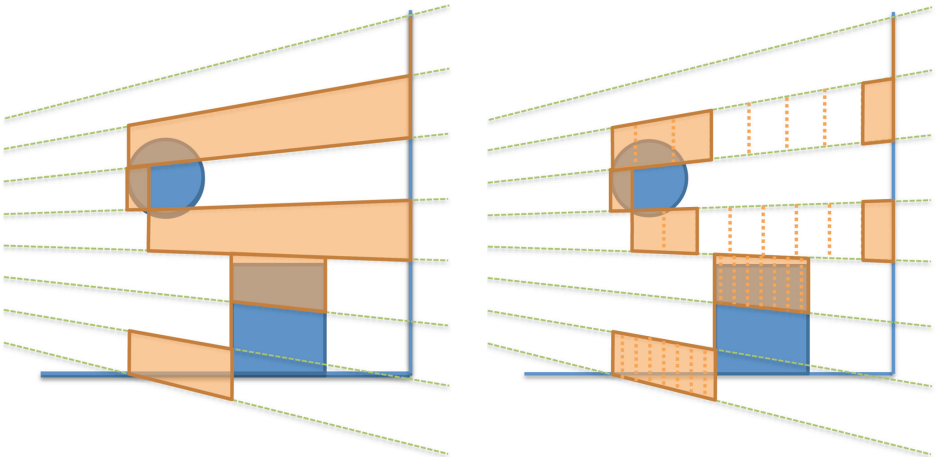
5.5.1 One-Bounce Indirect Illumination

As a unique extension of the light-culling system, lights can be used as what we call an indirect light to generate one-bounce indirect illumination in the scene. If a given light is tagged to be an indirect light, the following will occur for that light before any rendering passes at runtime:

- Generate a reflective shadow map (RSM) of the scene from the point of view of the light [Dachsbacher and Stamminger 05]. Normal buffer, color buffer, and world-space buffers are generated.

- A compute shader is executed to create spotlights at the location captured in the RSM. The generated spotlights are appended to the main light list. The direction of the spotlight will be the reflection of the vector from the world position to the original indirect light around the normal. Set other parameters for the new spotlight that conforms to the settings for the indirect light. We added art-driven parameters to control the effect of indirect lighting.

This new "indirect" light type is used by artists to spawn virtual spotlights that represent one-bounce lighting from the environment. This method seems to give artists good control over all aspects of lighting without requiring them to hand-place thousands or millions of lights or prebake lightmaps. Each indirect light can spawn $N \times N$ virtual spotlights, so it takes only a handful to create a nice indirect lighting effect. Once virtual lights are spawned in the compute shader, they go through the same light-culling process as all the other lights in the system. Thus, we could keep the entire rendering pipeline simple. Figure 5.5 illustrates the rendering pipeline used in the AMD Leo demo.



(a) (b)

Figure 5.6. Illustration of 2.5D culling. (a) Frustum culling creates a long frustum for a tile with foreground and background. (b) 2.5D culling—splitting depth into eight cells—does not capture lights falling between foreground and background.

5.6 Extensions

5.6.1 2.5D Culling

At the light-culling stage of Forward+, light geometries are tested against a frustum of each tile that is clipped by the maximum and minimum depth values of a tile. This light culling works well if there is little variance in the depth in a tile. Otherwise, it can create a long frustum for a tile. This results in capturing a lot of lights for a tile, as we can see at the edge of geometries in Figure 5.3(b), although some lights have no influence on any of the pixels in a tile if they fall at the void space in the frustum.

As the number of lights reported for a tile increases, the computational cost of final shading increases. This is critical especially when the shading computation for a light is expensive. This is often the case because one of the motivations of employing Forward+ is its ability to use sophisticated BRDFs for shading.

One obvious way to improve the efficiency of culling is to cull the lights using a 3D grid. However, this increases the computation as well as the size of the data to be exported. It is possible to develop sophisticated and expensive culling, but it shouldn't be overkill. Our proposed 2.5D culling constructs a nonuniform 3D grid without adding a lot of computation or stressing the memory bandwidth.

The idea is illustrated in Figure 5.6. This approach first constructs a frustum for a tile in the same way as the screen-space culling described in Section 5.3.

Then the extent of a frustum is split into cells; for each pixel in a tile, we flag a cell to which the pixel belongs. We call the data we construct for a tile a frustum and an array of occupancy flags a depth mask.

To check overlap of light geometry on the tile, the light geometry first is checked against the frustum. If the light overlaps, a depth mask is created for the light. This is done by calculating the extent of the light geometry in the depth direction of the frustum and flagging the cells to that extent. By comparing the depth mask for a light to the depth mask for the tile, we can cull the light in the depth direction. Overlap of the light is reported only if there is at least one cell flagged by both depth masks.

If a tile has a foreground and background, the 2.5D culling can detect and drop lights that fall between these two surfaces, thus reducing the number of lights to be processed at the final shading.

Implementation. The 2.5D culling splits a frustum into 32 cells, and so the occupancy information is stored in a 32-bit value. This cell data is allocated in shared memory to make it available to all threads in a group. The first modification to the light-culling kernel is the construction of an occupancy mask of the surface. This is performed after calculating the frustum extent in the depth direction. The pitch of a cell is calculated from the extent.

Once the pitch and the minimum depth value are obtained, any depth value can be converted to a cell index. To create the depth mask for a tile, we iterate through all the pixels in the tile and calculate a cell index for each pixel. Then a flag for the occupied cell is created by a bit shift, which is used to mark the depth mask in shared memory using an atomic logical-or operation.

Once we find a light overlapping the frustum, a depth mask is created for the light. The minimum and maximum depth values of the geometry are calculated and converted to cell indices. Once the cell indices are calculated, two bit-shift operations and a bit-and operation are necessary to create the depth mask for the light. If the light and surface occupy the same cell, both have the same flag at the cell. Thus taking logical and operation between these two masks is enough to check the overlap.

Results. We took several scenes and counted the number of lights per tile with the original Forward+ and Forward+ with our proposed 2.5D culling. The first benchmark is performed against the scene in Figure 5.7(a), which has a large variance in the depth. Figures 5.7(b) and 5.7(c) visualize the number of lights overlapping each tile using Forward+ with frustum culling and the proposed 2.5D culling.

Figure 5.7(b) makes clear that tiles that contain an object's edge capture a large number of lights. The number of overlapping lights is reduced dramatically when 2.5D culling is used (Figure 5.7(c)). We also counted the number of lights overlapping each tile and quantitatively compared these two culling methods

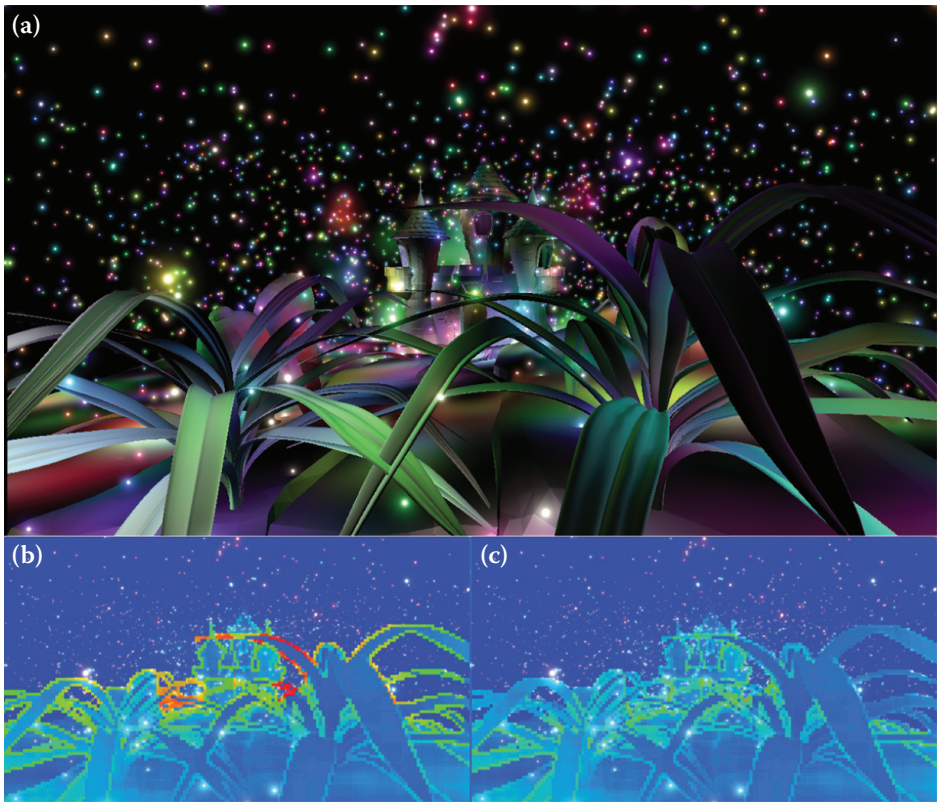


Figure 5.7. (a) A scene with a large depth variance that the original Forward+ could not process efficiently. (b) Visualization of the number of lights per tile using frustum culled with maximum and minimum depth values in a tile. (c) Visualization of the number of lights per tile using the proposed 2.5D culling.

(Figure 5.9(a)). Without the proposed method, there are a lot of tiles with more than 200 lights overlapping. However, by using the 2.5D culling, a tile has at most 120 overlapping lights. The benefit we can get from final shading depends on the implementation of shader, but culling eliminates a lot of unnecessary memory reads and computation for the final shader.

We also performed a test on the scene shown in Figure 5.8(a), which does not have as much depth variance as the scene in Figure 5.7(a). Because the depth difference is not large in these scenes, the number of lights overlapping a tile, including an edge of an object, is less than in the previous scene. However, color temperature is low when the 2.5D culling is used. A quantitative comparison is shown in Figure 5.9(b). Although the improvement is not as large as the

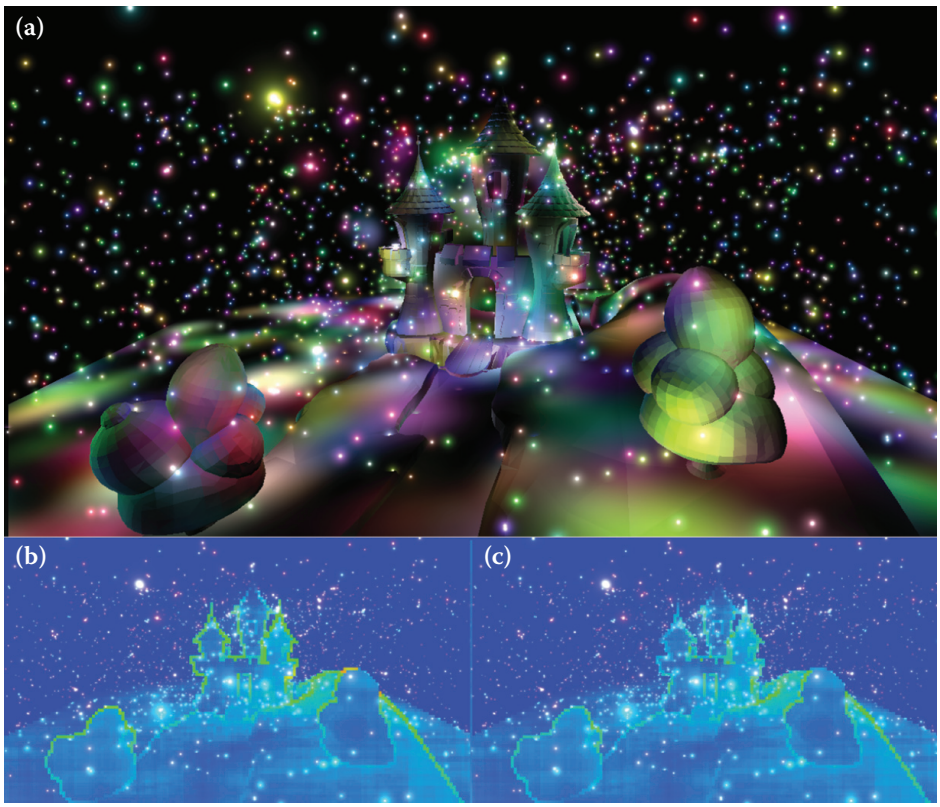
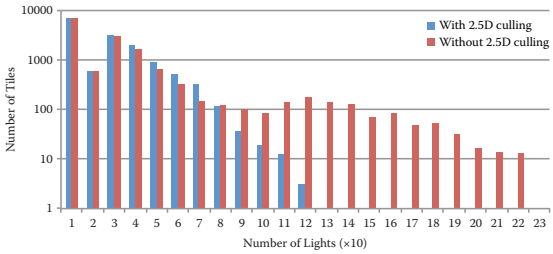
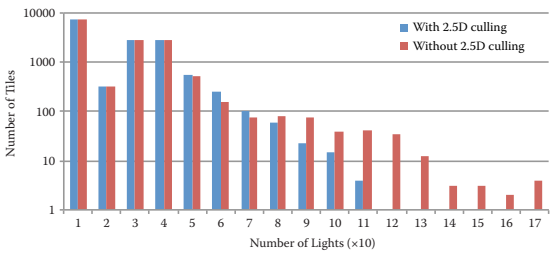


Figure 5.8. (a) A scene without a large depth variance. (b) Visualization of the number of lights per tile using frustum culled with maximum and minimum depth values in a tile. (c) Visualization of the number of lights per tile using the proposed 2.5D culling.



(a)



(b)

Figure 5.9. The count of tiles in terms of the number of lights for the scenes shown in Figures 5.7(a) and 5.8(a), respectively.

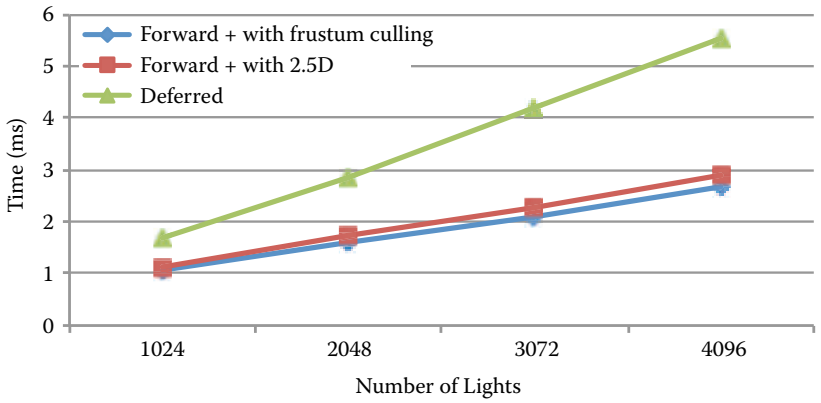


Figure 5.10. Comparison of computation time for the light-culling stage of Forward+ using frustum culling only and frustum culling plus 2.5D culling by changing the number of lights in the scene shown in Figure 5.7(a). Computation time for the light accumulation in compute-based deferred lighting is also shown.

previous scene, the proposed method could reduce the number of overlapping lights on tiles.

Figure 5.10 compares the computation time for the light-culling stage for the scene of Figure 5.1(a) as measured on an AMD Radeon HD 7970 GPU. This comparison indicates the overhead of additional computation in 2.5D culling is less than 10% of the time without the culling; when there are 1,024 lights, the overhead is about 5%. The 2.5D culling is effective regardless of the number of the light in the scene.

Figure 5.10 also contains the light accumulation time of the compute-based deferred lighting. We can see that the light-culling stage with the 2.5D culling in Forward+ is much faster than its counterpart in deferred lighting.

5.6.2 Shadowing from Many Lights

Shadows from a light can be calculated by a shadow map, with which we can get occlusion from the light in the pixel shader when the forward-rendering pipeline is used. We can calculate shadows in the same way for Forward+.

Because Forward+ is capable of using hundreds of lights for lighting, it is natural to wonder how we can use all of those lights in the scene as shadow-casting lights. One option is use a shadow map for each light. This solution is not practical because shadow map creation—the cost of which is linear to scene complexity—can be prohibitively expensive.

We can reduce the shadow map resolution, but this results in low-quality shadows.

Another option relies on rasterization and also borrows an idea from ray tracing. To check the visibility to a light, we can cast a ray to the light. If the light is local, the length of the ray is short. This means we do not have to traverse much in the scene; the cost is not as high as the cost of ray casting a long ray in full ray tracing.

In this subsection, we describe how ray casting can be integrated in Forward+ to add shadows from hundreds of lights and show that a perfect shadow from hundreds of lights can be obtained in real time. After adding this feature, Forward+ is not just an extension of forward-rendering pipeline but a hybrid of forward, deferred-rendering pipelines and ray tracing.

Implementation. To ray cast against the scene, we need the position and normal vector of a primary ray hit and the acceleration data structure for ray casting. The position of a primary ray hit can be reconstructed from the depth buffer by applying inverse projection. The normal vector of the entire visible surface, which is used to avoid casting rays to a light that is at the back of the surface and to offset the ray origin, can be written at the depth prepass. The prepass is no longer writing only the depth value, and so it is essentially identical to a G-pass in the deferred-rendering pipeline. The acceleration structure has to be updated every frame for a dynamic scene; however, this is a more extensive research topic and we do not explore it in this chapter. Instead, we just assume that the data structure is built already.

After the prepass, implementing a ray-cast shadow is straightforward. In a pixel shader, we have access to all the information about lights, which includes light position. A shadow ray can be created by the light position and surface location. Then we can cast the ray against the acceleration structure for an intersection test. If the ray is intersecting, contribution from the light is masked.

Although this naive implementation is easy to implement, it is far from practical in terms of performance. The issue is a legacy of the forward-rendering pipeline. The number of rays to be cast for each pixel is not constant, which means the computational load or time can vary considerably among pixels even if they belong to the same surface. This results in a poor utilization of the GPU.

An alternative is to separate ray casting from pixel shading for better performance. After separating ray casting from pixel shading, the pipeline looks like this:

- G-pass,
- light culling,
- ray-cast job creation,
- ray casting,
- final shading.

After indices of lights overlapping each tile are calculated in the light-culling stage, ray-cast jobs are created and accumulated in a job buffer by iterating through all the screen pixels. This is a screen-space computation in which a thread is executed for a pixel and goes through the list of lights. If a pixel overlaps a light, a ray-cast job is created. To create a ray in the ray-casting stage, we need a pixel index to obtain surface position and normal, and a light index against which the ray is cast. These two indices are packed into a 32-bit value and stored in the job buffer.

After creating all the ray-cast jobs in a buffer, we dispatch a thread for each ray-cast job. Then it does not have the issue of uneven load balancing we experience when rays are cast in a pixel shader. Each thread is casting a ray. After identifying whether a shadow ray is blocked, the information has to be stored somewhere to pass to a pixel shader. We focused only on a hard shadow, which means the output from a ray cast is a binary value. Therefore, we have packed results from 32 rays into one 32-bit value.

But in a scene with hundreds of lights, storing a mask for all of them takes too much space even after the compression. We took advantage of the fact that we have a list of lights per tile; masks for lights in the list of a tile are only stored. We limit the number of rays to be cast per pixel to 128, which means the mask can be encoded as an int4 value. At the ray-casting stage, the result is written to the mask of the pixel using an atomic OR operation to flip the assigned bit.

After separating ray casting from pixel shading, we can keep the final shading almost the same in Forward+. We only need to read the shadow mask for each pixel; whenever a light is processed, the mask is read to get the occlusion.

Results. Figure 5.11 is a screenshot of a scene with 512 shadow-casting lights. We can see legs of chairs are casting shadows from many dynamic lights in the scene. The screen resolution was 1,280 × 720. The number of rays cast for this scene was more than 7 million. A frame computation time is about 32 ms on an AMD Radeon HD 7970 GPU. G-pass and light culling took negligible time compared to ray-cast job creation and ray casting, each of which took 11.57 ms and 19.91 ms for this frame. This is another example of hybrid ray-traced and rasterized graphics.

5.7 Conclusion

We have presented Forward+, a rendering pipeline that adds a GPU compute-based light-culling stage to the traditional forward-rendering pipeline to handle many lights while keeping the flexibility for material usage. We also presented the implementation detail of Forward+ using DirectX 11, and its performance. We described how the Forward+ rendering pipeline is extended to use an indirect illumination technique in the AMD Leo Demo.

Figure 5.11. Dynamic shadowing from 512 lights in a scene with 282,755 triangles.

Because of its simplicity and flexibility, there are many avenues to extend Forward+. We have described two extensions in this chapter: a 2.5D culling, which improves the light-culling efficiency, and dynamic shadowing from many lights.

5.8 Acknowledgments

We would like to thank to members of AMD GPU Tech initiatives and other people who worked on the AMD Leo demo.

Bibliography

[Andersson 11] J. Andersson. "DirectX 11 Rendering in Battlefield 3." Presentation, Game Developers Conference, San Francisco, CA, 2011.

[Dachsbacher and Stamminger 05] C. Dachsbacher and M. Stamminger. "Reflective Shadow Maps." In *Symposium on Interactive 3D Graphics and Games (I3D)*, pp. 203–231. New York: ACM, 2005.

[Ericson 04] C. Ericson. *Real-Time Collision Detection*. San Francisco: Morgan Kaufmann, 2004.

[Harada et al. 11] T. Harada, J. McKee, and J. C. Yang. "Forward+: Bringing Deferred Lighting to the Next Level," Eurographics Short Paper, Cagliari, Italy, May 15, 2012.

[Yang et al. 10] J. C. Yang, J. Hensley, H. Grun, and N. Thibieroz. "Real-Time Concurrent Linked List Construction on the GPU." *Computer Graphics Forum* 29:4 (2010), 1297–1304.