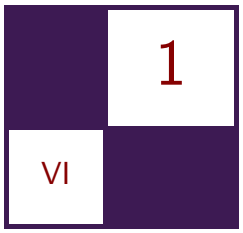


Compute-Based Tiled Culling

Jason Stewart

1.1 Introduction

Modern real-time rendering engines need to support many dynamic light sources in a scene. Meeting this requirement with traditional forward rendering is problematic. Typically, a forward-rendered engine culls lights on the CPU for each batch of scene geometry to be drawn, and changing the set of lights in use requires a separate draw call. Thus, there is an undesirable tradeoff between using smaller pieces of the scene for more efficient light culling versus using larger batches and more instancing for fewer total draw calls. The intersection tests required for light culling can also be a performance burden for the CPU.

Deferred rendering better supports large light counts because it decouples scene geometry rendering and material evaluation from lighting. First, the scene is rendered and geometric and material properties are stored into a geometry buffer or G-buffer [Saito and Takahashi 90]. Lighting is accumulated separately, using the G-buffer as input, by drawing light bounding volumes or screen-space quads. Removing lighting from the scene rendering pass eliminates the state switching for different light sets, allowing for better batching. In addition, CPU light culling is performed once against the view frustum instead of for each batch, reducing the performance cost. However, because each light is now accumulated separately, overlapping lights increase bandwidth consumption, which can decrease GPU performance [Lauritzen 10].

This chapter presents a better method for supporting large light counts: compute-based tiled culling. Modern GPUs, including those in Xbox One and Playstation 4, can execute general-purpose computation kernels. This capability allows light culling to be performed on the GPU. The technique can be used with both forward and deferred rendering. It eliminates light state switching and CPU culling, which helps forward rendering, and it calculates lighting in a single pass, which helps deferred rendering. This chapter presents the technique in detail, including code examples in HLSL and various optimizations. The companion code implements the technique for both forward and deferred rendering and includes a benchmark.

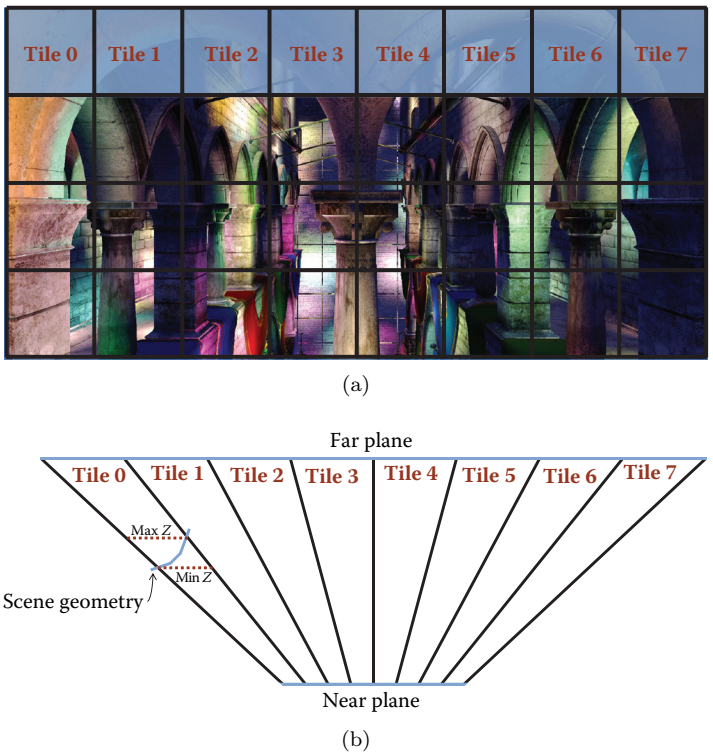


Figure 1.1. Partitioning the scene into tiles. (a) Example screen tiles. (b) Fitting view frustum partitions to the screen tiles. For clarity, the tiles shown in this figure are very large. They would typically be 16×16 pixels.

In addition to using the companion code to measure performance, results are presented using Unreal Engine 4, including a comparison of standard deferred rendering versus tiled deferred.

1.2 Overview

Compute-based tiled culling works by partitioning the screen into fixed-size tiles, as shown in Figure 1.1(a). For each tile, a compute shader[1] loops over all lights in the scene and determines which ones intersect that particular tile. Figure 1.1(b) gives a 2D, top-down example of how the tile bounding volume is constructed. Four planes are calculated to represent the left, right, top, and bottom of an

[1]This chapter uses Direct3D 11 terminology. In Direct3D 11, the general-purpose computation technology required for tiled culling is called DirectCompute 5.0, and the general-purpose kernel is called a compute shader.

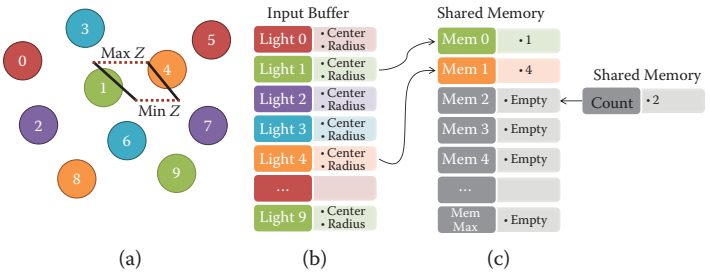


Figure 1.2. Tiled culling overview.

asymmetric partition of the view frustum that fits exactly around the tile. To allow for tighter culling, the minimum and maximum scene depths are calculated for the tile, as shown in Figure 1.1(b) for Tile 0. These depth values form the front and back of the frustum partition. This gives the six planes necessary for testing the intersection between light bounding volumes and the tile.

Figure 1.2 provides an overview of the algorithm. Figure 1.2(a) shows a 2D representation of a tile bounding volume, similar to that shown for Tile 0 in Figure 1.1(b). Several scene lights are also shown. Figure 1.2(b) shows the input buffer containing the scene light list. Each entry in the list contains the center and radius for that light's bounding sphere.

The compute shader is configured so that each thread group works on one tile. It loops over the lights in the input buffer and stores the indices of those that intersect the tile into shared memory.[2] Space is reserved for a per-tile maximum number of lights, and a counter tracks how many entries were actually written, as shown in Figure 1.2(c).

Algorithm 1.1 summarizes the technique.

Referring back to Figure 1.2 as a visual example of the loop in Algorithm 1.1, note from Figure 1.2(a) that two lights intersect the frustum partition: Light 1 and Light 4. The input buffer index (Figure 1.2(b)) of each intersecting light is written to shared memory (Figure 1.2(c)). To make this thread safe, so that lights can be culled in parallel, a counter is stored in shared memory and incremented using the atomic operations available in compute shaders.

1.3 Implementation

This section gives an implementation in HLSL of the compute-based tiled-culling algorithm discussed in the previous section. The three parts of Algorithm 1.1 will be presented in order: depth bounds calculation, frustum planes calculation, and intersection testing.

[2]Compute shader execution is organized into thread groups. Threads in the same thread group have access to shared memory.

```
Input: light list, scene depth
Output: per-tile list of intersecting lights

calculate depth bounds for the tile;
calculate frustum planes for the tile;

for i ← thread_index to num_lights do
    current_light ← light_list[i];
    test intersection against tile bounding volume;
    if intersection then
        thread-safe increment of list counter;
        write light index to per-tile list;
    end
    i ← i + num_threads_per_tile;
end
```

Algorithm 1.1. Basic tiled culling.

1.3.1 Depth Bounds Calculation

As mentioned previously (in Footnote 2), compute shader execution is organized into thread groups. You specify the exact organization as part of the compute shader. In HLSL, this is done with the `numthreads` attribute, as shown on line 15 of Listing 1.1. For tiled culling, the thread groups are organized to match the tile size. For example, `TILE_RES` is defined as 16 in Listing 1.1, and the 16×16-pixel tile size results in a 16×16-thread layout in the compute shader.

Compute shaders are executed with the `Dispatch` method, which specifies the number of thread groups to launch. For example, a 1920×1080 screen resolution with 16×16-pixel tiles requires 120×68 tiles to cover the screen. Thus, by calling `Dispatch(120,68,1)` for a compute shader with `[numthreads(16,16,1)]`, each thread maps to a particular screen pixel.

To calculate the depth bounds, each thread simply reads its pixel's depth value from the scene depth buffer and performs a thread-safe atomic minimum and maximum in shared memory. The depth buffer read happens on lines 20–21 of Listing 1.1. The `globalIdx` variable used to address the depth buffer is the `SV_DispatchThreadID` value (see line 16), one of the special system-value semantics available to compute shaders. Because the thread group layout from the `Dispatch` call matches the screen tiles and the thread layout from the `numthreads` attribute matches the tile size, the `SV_DispatchThreadID` value corresponds to a screen pixel address and can be used directly with the `Load` function.

One minor complication with the depth bounds calculation is that the scene depth value is floating point, but the atomic minimum and maximum functions (`InterlockedMin` and `InterlockedMax` on lines 43–44) only operate on integer types. Therefore, `asuint` is used to store the raw bits of the floating point depth value,

```
Texture2D<float> g_SceneDepthBuffer;

// Thread Group Shared Memory (aka local data share, or LDS)
groupshared uint ldsZMin;
groupshared uint ldsZMax;

// Convert a depth value from postprojection space
// into view space
float ConvertProjDepthToView(float z)
{
  return (1.f/(z*g_mProjectionInv._34 + g_mProjectionInv._44));
}

#define TILE_RES 16
[numthreads(TILE_RES,TILE_RES,1)]
void CullLightsCS(uint3 globalIdx : SV_DispatchThreadID,
                  uint3 localIdx  : SV_GroupThreadID,
                  uint3 groupIdx  : SV_GroupID)
{
  float depth = g_SceneDepthBuffer.Load(uint3(globalIdx.x,
                                        globalIdx.y,0)).x;
  float viewPosZ = ConvertProjDepthToView(depth);
  uint z = asuint(viewPosZ);

  uint threadNum = localIdx.x + localIdx.y*TILE_RES;

  // There is no way to initialize shared memory at
  // compile time, so thread zero does it at runtime
  if(threadNum == 0)
  {
    ldsZMin = 0x7f7fffff;  // FLT_MAX as a uint
    ldsZMax = 0;
  }
  GroupMemoryBarrierWithGroupSync();

  // Parts of the depth buffer that were never written
  // (e.g., the sky) will be zero (the companion code uses
  // inverted 32-bit float depth for better precision).
  if(depth != 0.f)
  {
    // Calculate the minimum and maximum depth for this tile
    // to form the front and back of the frustum
    InterlockedMin(ldsZMin,z);
    InterlockedMax(ldsZMax,z);
  }
  GroupMemoryBarrierWithGroupSync();

  float minZ = asfloat(ldsZMin);
  float maxZ = asfloat(ldsZMax);

  // Frustum planes and intersection code goes here
  ...
}
```

Listing 1.1. Depth bounds calculation.

and the minimum and maximum are performed against these unsigned bits. This works because the floating point depth is always positive, and the raw bits of a 32-bit floating point value increase monotonically in this case.

```
// Plane equation from three points, simplified
// for the case where the first point is the origin.
// N is normalized so that the plane equation can
// be used to compute signed distance.
float4 CreatePlaneEquation(float3 Q, float3 R)
{
  // N = normalize(cross(Q-P,R-P)),
  // except we know P is the origin
  float3 N = normalize(cross(Q,R));
  // D = -(N dot P), except we know P is the origin
  return float4(N,0);
}

// Convert a point from postprojection space into view space
float3 ConvertProjToView(float4 p)
{
  p = mul(p,g_mProjectionInv);
  return (p/p.w).xyz;
}

void CullLightsCS(uint3 globalIdx : SV_DispatchThreadID,
                  uint3 localIdx  : SV_GroupThreadID,
                  uint3 groupIdx  : SV_GroupID)
{
  // Depth bounds code goes here
  ...
  float4 frustumEqn[4];
  { // Construct frustum planes for this tile
    uint pxm = TILE_RES*groupIdx.x;
    uint pym = TILE_RES*groupIdx.y;
    uint pxp = TILE_RES*(groupIdx.x+1);
    uint pyp = TILE_RES*(groupIdx.y+1);
    uint width =  TILE_RES*GetNumTilesX();
    uint height = TILE_RES*GetNumTilesY();

    // Four corners of the tile, clockwise from top-left
    float3 p[4];
    p[0] = ConvertProjToView(float4(pxm/(float)width*2.f-1.f,
           (height-pym)/(float)height*2.f-1.f,1.f,1.f));
    p[1] = ConvertProjToView(float4(pxp/(float)width*2.f-1.f,
           (height-pym)/(float)height*2.f-1.f,1.f,1.f));
    p[2] = ConvertProjToView(float4(pxp/(float)width*2.f-1.f,
           (height-pyp)/(float)height*2.f-1.f,1.f,1.f));
    p[3] = ConvertProjToView(float4(pxm/(float)width*2.f-1.f,
           (height-pyp)/(float)height*2.f-1.f,1.f,1.f));

    // Create plane equations for the four sides, with
    // the positive half-space outside the frustum
    for(uint i=0; i<4; i++)
      frustumEqn[i] = CreatePlaneEquation(p[i], p[(i+1)&3]);
  }
  // Intersection code goes here
  ...
}
```

Listing 1.2. Frustum planes calculation.

1.3.2 Frustum Planes Calculation

The frustum planes code appears in Listing 1.2 and is straightforward. The four corners of the tile are constructed in postprojection space and converted to view space. These four corners are then used to calculate the planes for the four sides of the frustum partition. Two corners and the origin give the three points needed for each plane equation.

To calculate the pixel locations of the four corners, the `groupIdx` variable is used, which holds the `SV_GroupID` value (see line 23), another of the special system-value semantics available to compute shaders. Because the thread group layout from the `Dispatch` call matches the screen tiles, the `SV_GroupID` value corresponds to the tile number.

One subtlety happens on lines 33–34. Note that the screen size might not be evenly divisible by the tile size, so the screen width and height cannot be used directly in the four corners calculation. Instead, the code calculates the "whole tile" resolution, which is the closest greater-than (or equal-to) value that is evenly divisible by the tile size.

1.3.3 Intersection Testing

The depth bounds and the four plane equations form the six sides of the tile bounding volume. Light culling is accomplished by testing light bounding volumes for intersection against the tile bounding volume. This is shown in Listing 1.3.

In this example, the light bounding volumes are spheres (a natural fit for point lights), and a standard frustum versus sphere intersection test is performed. That is, the sphere is tested against the six planes of the frustum. If it passes, the index of the light in the input buffer is written to shared memory.

Note on line 28 that each thread starts the loop at a different index and increments the loop counter by `NUM_THREADS`, which is 256 for 16×16-pixel tiles. This allows 256 lights to be culled in parallel for each loop iteration. To make the parallel culling thread safe, `InterlockedAdd` is used on line 44 to increment the output list counter.

As mentioned in the introduction, compute-based tiled culling can be applied to forward rendering [Harada et al. 12] and deferred rendering [Andersson 09]. When used with forward rendering, it is commonly called Forward+ [Harada et al. 12]. When used with deferred rendering, it is called tile-based deferred [Lauritzen 10] or simply tiled deferred [Lauritzen 12]. For Forward+, the compute shader writes the per-tile list to an output buffer (i.e., `RWBuffer`). The forward pixel shader then calculates the tile to which it belongs and uses the list for that tile as input to calculate the lighting. For tiled deferred, the same compute shader that does the light culling can then do the lighting, using the list in shared

```
Buffer<float4> g_LightBufferCenterAndRadius;

#define MAX_NUM_LIGHTS_PER_TILE 256
groupshared uint ldsLightIdxCounter;
groupshared uint ldsLightIdx[MAX_NUM_LIGHTS_PER_TILE];

// Point-plane distance, simplified for the case where
// the plane passes through the origin
float GetSignedDistanceFromPlane(float3 p, float4 eqn)
{
  // dot(eqn.xyz, p) + eqn.w, except we know eqn.w is zero
  return dot(eqn.xyz, p);
}

#define NUM_THREADS (TILE_RES*TILE_RES)
void CullLightsCS(...)
{
  // Depth bounds and frustum planes code goes here
  ...
  if(threadNum == 0)
  {
    ldsLightIdxCounter = 0;
  }
  GroupMemoryBarrierWithGroupSync();

  // Loop over the lights and do a
  // sphere versus frustum intersection test
  for(uint i=threadNum; i<g_uNumLights; i+=NUM_THREADS)
  {
    float4 p = g_LightBufferCenterAndRadius[i];
    float  r = p.w;
    float3 c = mul(float4(p.xyz,1), g_mView).xyz;

    // Test if sphere is intersecting or inside frustum
    if((GetSignedDistanceFromPlane(c,frustumEqn[0]) < r) &&
       (GetSignedDistanceFromPlane(c,frustumEqn[1]) < r) &&
       (GetSignedDistanceFromPlane(c,frustumEqn[2]) < r) &&
       (GetSignedDistanceFromPlane(c,frustumEqn[3]) < r) &&
       (-c.z + minZ < r) && (c.z - maxZ < r))
    {
      // Do a thread-safe increment of the list counter
      // and put the index of this light into the list
      uint dstIdx = 0;
      InterlockedAdd(ldsLightIdxCounter,1,dstIdx);
      ldsLightIdx[dstIdx] = i;
    }
  }
  GroupMemoryBarrierWithGroupSync();
}
```

Listing 1.3. Intersection testing.

memory directly. Even if lights overlap, the G-buffer is only read once for each pixel, and the lighting results are accumulated into shader registers instead of blended into a render target, reducing bandwidth consumption.

1.4 Optimization

This section covers various optimizations to the compute-based tiled-culling technique. Common pitfalls to avoid are presented first, followed by several optimizations to the basic implementation from the previous section.

1.4.1 Common Pitfalls

Part of optimization is avoiding common pitfalls. Two such pitfalls for compute-based tiled culling are described in this section: forgetting to be cache friendly and choosing a suboptimal tile size. The pitfalls are illustrated by making two seemingly small changes to the code in Section 1.3 and showing that those changes hurt performance dramatically.

For the first change, note that line 1 in Listing 1.3 shows that the light bounding spheres (centers and radii) were stored in a buffer with no other data. However, for convenience and code clarity, developers might decide to include other light data in the same buffer, as shown below.

```
struct LightArrayData
{
  float4 v4CenterAndRadius;
  float4 v4Color;
};
StructuredBuffer<LightArrayData> g_LightBuffer;
```

For the second change, recall that line 14 in Listing 1.1 defines `TILE_RES` as 16, resulting in 16×16 threads per thread group, or 256 threads. For AMD GPUs, work is executed in 64-thread batches called *wavefronts*, while on NVIDIA GPUs, work is executed in 32-thread *warps*. Thus, efficient compute shader execution requires the number of threads in a thread group to be a multiple of 64 for AMD or 32 for NVIDIA. Since every multiple of 64 is a multiple of 32, standard performance advice is to configure the thread count to be a multiple of 64. Because 256 is a multiple of 64, setting `TILE_RES` to 16 follows this advice. Alternatively, setting `TILE_RES` to 8 (resulting in 8×8-pixel tiles) yields 64 threads per thread group, which is certainly also a multiple of 64, and the smaller tile size might result in tighter culling.

Although these two changes seem minor, both decrease performance, as shown in Figure 1.3. The "unoptimized" curve contains both changes (combined light data in a `StructuredBuffer` and 8×8 tiles). For the cache friendly curve, the

[3] All performance data in this chapter was gathered on an AMD Radeon R7 260X GPU. The R7 260X was chosen because its performance characteristics are roughly comparable to the Xbox One and Playstation 4.

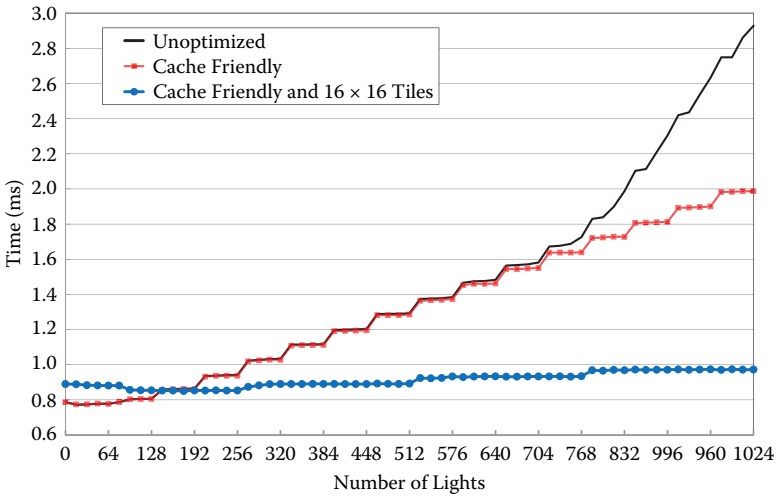


Figure 1.3. Basic optimizations.[3] Tiled-culling compute shader execution time versus number of lights for Forward+ rendering at 1920 × 1080 using the companion code for this chapter.

`StructuredBuffer` is replaced with the declaration shown in line 1 of Listing 1.3 containing only the data needed for culling. Note that, while performance is similar for much of the chart, performance improves by nearly 1 ms for 1024 lights. Specifically, compute shader execution time decreases from 2.93 ms to 1.99 ms, a 32% reduction.

The "cache friendly" label hints at why this configuration improves performance. Data not needed for culling pollutes the cache during compute shader execution, eventually becoming a bottleneck as light count increases. In general, a structure of arrays (in this case, separate arrays for culling data and light color) is often better for GPU execution than an array of structures, because it allows more cache-friendly memory access.

The "cache friendly and 16 × 16 tiles" curve keeps the cache-friendly light buffer and changes `TILE_RES` back to 16, resulting in the implementation given in Section 1.3. Because there are now 256 threads, many threads do not have any lights to cull at the lower end of the chart, resulting in a slight performance decrease initially. However, this version scales much better with increasing light counts. At 1024 lights, compute shader execution time is 0.97 ms, a 51% reduction from the previous version and a 67% reduction from the unoptimized version.

The 16 × 16 configuration is better because more threads per thread group results in more wavefronts/warps in flight per thread group. This allows GPU schedulers to better hide memory latency by switching execution to a new wavefront/warp when the current one hits a high-latency operation.

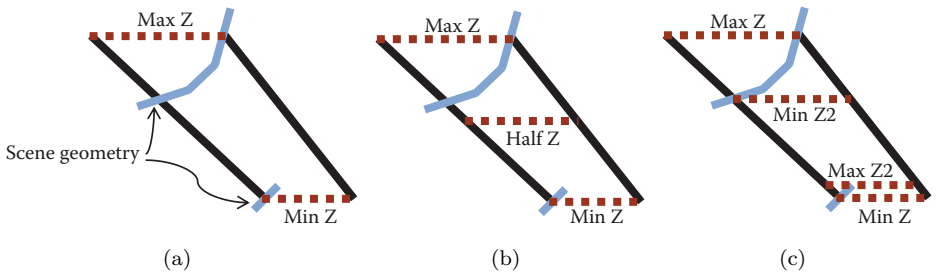


Figure 1.4. Depth discontinuity optimization strategies. (a) Scene depth discontinuities can cause a large depth range in the tile bounding volume. (b) The Half Z method splits the depth range in half and culls against the two ranges. (c) The Modified Half Z method calculates a second minimum and maximum, bounded by the Half Z value.

1.4.2 Depth Discontinuities

Having covered the basic optimizations already present in the code from Section 1.3, additional optimizations will now be presented, starting with those for discontinuities in scene depth.

Figure 1.4 shows 2D representations of a tile bounding volume, similar to that shown for Tile 0 in Figure 1.1(b). As demonstrated in Figure 1.4(a), a foreground object in front of a background object can lead to a large depth range in the tile bounding volume. Lights can intersect the empty space between foreground and background but not actually affect any pixels in the tile. That is, depth discontinuities can lead to an increase in false-positive intersections.

Half Z. Figure 1.4(b) shows a strategy to better handle depth discontinuities called the Half Z method. It simply divides the depth range in two at the midpoint and culls against two depth ranges: one from Min Z to Half Z, and one from Half Z to Max Z. A separate per-tile list is maintained for each depth range. This method requires only two additional plane tests and is a minor change to the code. Listing 1.4 shows the intersection test for this method.

Modified Half Z. Figure 1.4(c) shows a second strategy called the Modified Half Z method. It performs additional atomic operations to find a second maximum (Max Z2) between Min Z and Half Z and a second minimum (Min Z2) between Half Z and Max Z. This can result in tighter bounding volumes compared to the Half Z method, but calculating the additional minimum and maximum is more expensive than simply calculating Half Z, due to the additional atomic operations required.

Light count reduction results. Figure 1.5 shows the reduction in per-tile light count at depth discontinuities from the methods discussed in this section. Note the

```
// Test if sphere is intersecting or inside frustum
if((GetSignedDistanceFromPlane(c,frustumEqn[0]) < r) &&
   (GetSignedDistanceFromPlane(c,frustumEqn[1]) < r) &&
   (GetSignedDistanceFromPlane(c,frustumEqn[2]) < r) &&
   (GetSignedDistanceFromPlane(c,frustumEqn[3]) < r))
{
  if(-c.z + minZ < r && c.z - halfZ < r)
  {
    // Do a thread-safe increment of the list counter
    // and put the index of this light into the list
    uint dstIdx = 0;
    InterlockedAdd(ldsLightIdxCounterA,1,dstIdx);
    ldsLightIdxA[dstIdx] = i;
  }
  if(-c.z + halfZ < r && c.z - maxZ < r)
  {
    // Do a thread-safe increment of the list counter
    // and put the index of this light into the list
    uint dstIdx = 0;
    InterlockedAdd(ldsLightIdxCounterB,1,dstIdx);
    ldsLightIdxB[dstIdx] = i;
  }
}
```

Listing 1.4. Half Z method.

column in the foreground of the left side of the scene in Figure 1.5(a). This causes depth discontinuities for tiles along the column, resulting in the high light counts shown in red in Figure 1.5(c) for the baseline implementation in Section 1.3.

The results for the Half Z method are shown in Figure 1.5(d). Note that the light counts for tiles along the column have been reduced. Then, for the Modified Half Z method, note that light counts have been further reduced in Figure 1.5(e).

Performance results. Figure 1.6 shows the performance of these methods. Note that, while Figure 1.3 measured only the tiled-culling compute shader, Figure 1.6 measures both the compute shader and the forward pixel shader for Forward+ rendering. More time spent during culling can still be an overall performance win if enough time is saved during lighting, so it is important to measure both here.

The "Baseline" curve is from the implementation in Section 1.3. The "Half Z" curve shows this method at a slight performance disadvantage for lower light counts, because the savings during lighting do not yet outweigh the extra cost of testing two depth ranges and maintaining two lists. However, this method becomes faster at higher light counts. The "Modified Half Z" curve starts out with a bigger deficit, due to the higher cost of calculating the additional minimum and maximum with atomics. It eventually pulls ahead of the baseline method, but never catches Half Z. However, this method's smaller depth ranges can still be useful if additional optimizations are implemented, as shown next.

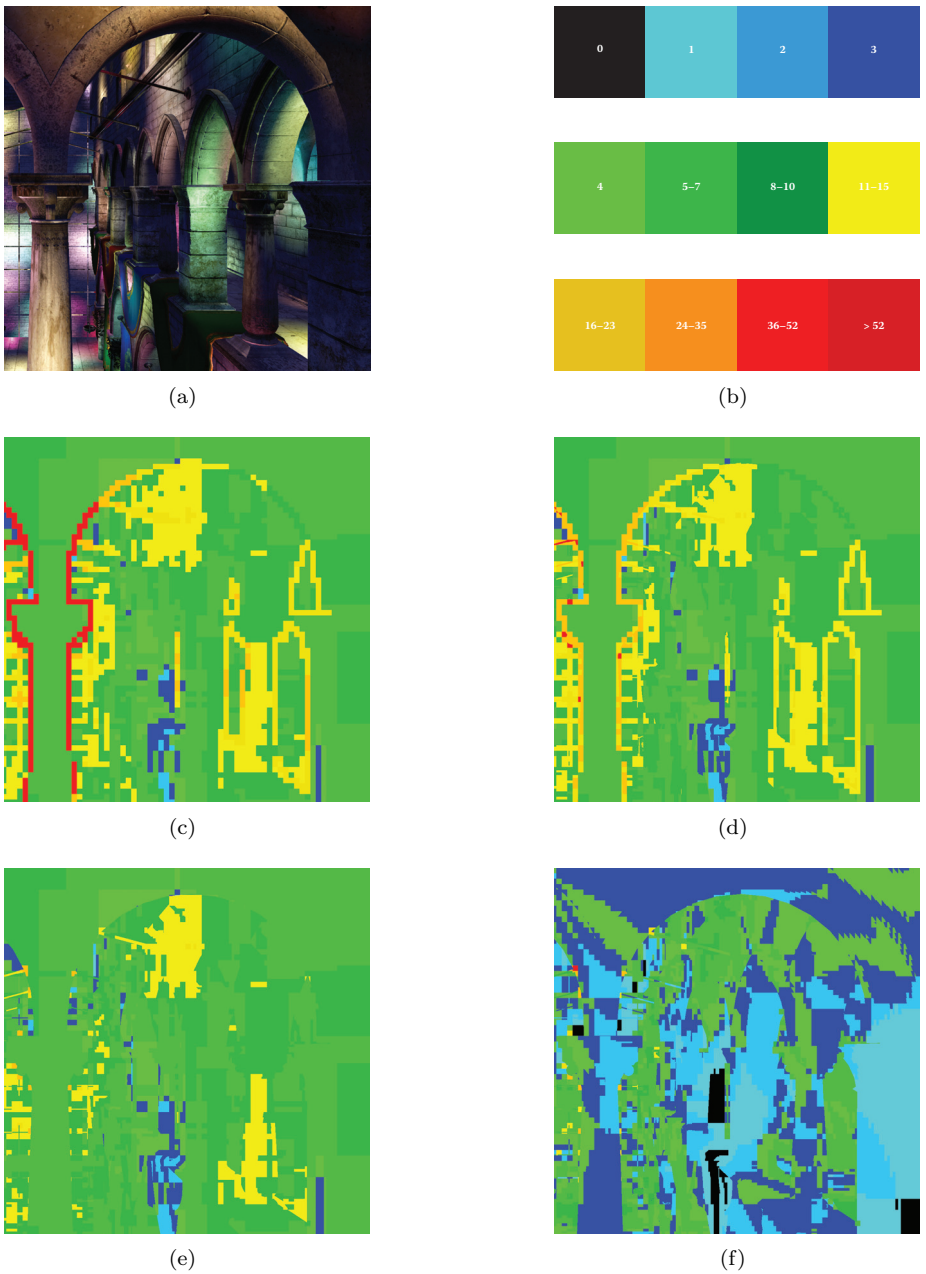


Figure 1.5. Tiled-culling optimization results using the companion code for this chapter. (a) Scene render. (b) Log scale lights-per-tile legend. (c) Baseline. (d) Half Z. (e) Modified Half Z. (f) Modified Half Z with AABBs.

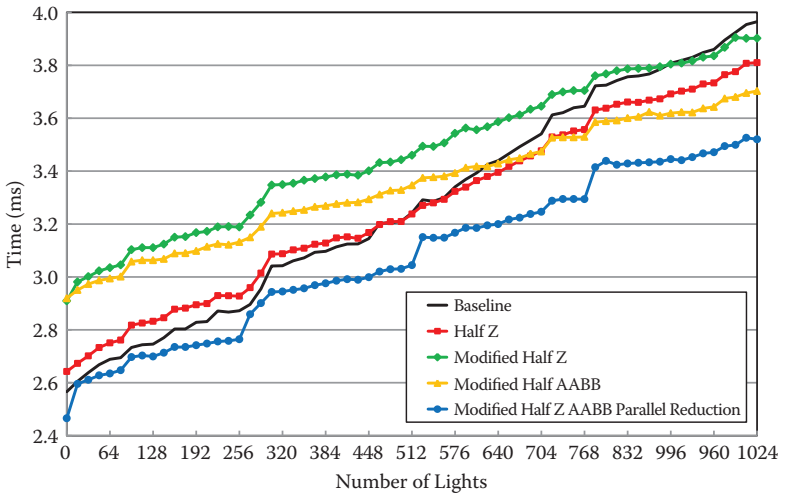


Figure 1.6. Tiled-culling optimizations. GPU execution time versus number of lights using the companion code for this chapter. The vertical axis represents the combined time for the tiled-culling compute shader and the forward pixel shader in Forward+ rendering at 1920×1080.

1.4.3 Frustum Planes versus AABBs

In our previous discussion of the results in Figure 1.5, one result was not mentioned. If view-space axis-aligned bounding boxes (AABBs) are used to bound the tile instead of frustum planes, per-tile light counts can be further reduced, as shown in Figure 1.5(f).

Testing intersection against a frustum using six planes is an approximation. As shown in Figure 1.7(a), the actual intersection volume has curved corners. Regions exist outside the curved corners that will still pass testing against the planes, resulting in false-positive intersections.

Fitting an AABB around the tile's frustum partition will also produce regions where false-positive intersections can occur, as illustrated in Figure 1.7(b). The key difference is that, as the depth range decreases (i.e., as Max Z gets closer to Min Z), these regions get smaller for AABBs, as shown in Figure 1.7(c).

Referring back to Figure 1.5(f), using AABBs with the smaller depth ranges of the Modified Half Z method results in a significant reduction in per-tile light counts. Whereas the previous results showed improvement primarily at depth discontinuities, this method shows an overall improvement. For small depth ranges, the AABB intersection volume nearly matches the true volume, resulting in tighter culling.

Referring back to Figure 1.6, the "Modified Half Z, AABB" curve still starts out at a deficit, due to the increased cost of finding the second minimum and max-

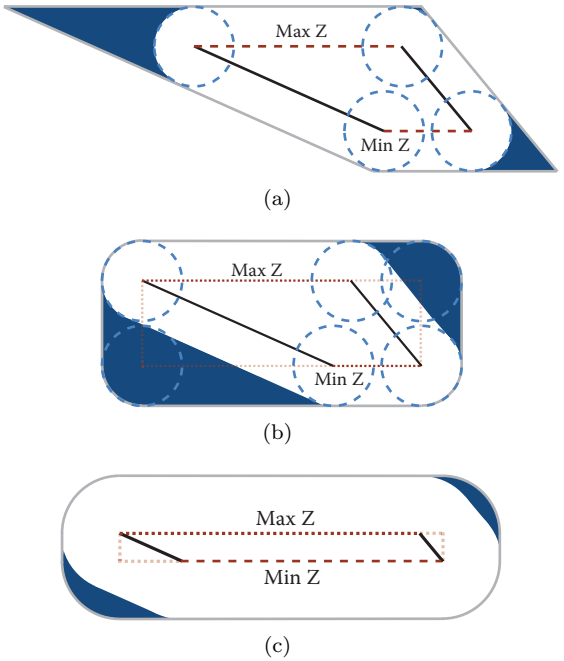


Figure 1.7. Frustum planes versus AABBs. False positive intersections will occur in the shaded regions. (a) Frustum intersection testing. (b) AABB intersection testing. (c) AABB intersection with a small depth range.

imum. However, it scales better as light count increases, eventually overtaking the Half Z method.

1.4.4 Parallel Reduction

Using AABBs with the smaller depth ranges of the Modified Half Z method produces good culling results, but the cost of the second minimum and maximum is significant. There is, however, another way to calculate the depth bounds: parallel reduction. Using the methods first outlined in [Harris 07], as well as the results from [Engel 14], an optimized parallel reduction implementation can be used to produce the smaller depth ranges of the Modified Half Z method, as shown in Listing 1.5.

```
1 Texture2D<float>     g_SceneDepthBuffer;
2 RWTexture2D<float4> g_DepthBounds;
3
4 #define TILE_RES 16
5 #define NUM_THREADS_1D (TILE_RES/2)
```

```
#define NUM_THREADS (NUM_THREADS_1D*NUM_THREADS_1D)

// Thread Group Shared Memory (aka local data share, or LDS)
groupshared float ldsZMin[NUM_THREADS];
groupshared float ldsZMax[NUM_THREADS];

// Convert a depth value from postprojection space
// into view space
float ConvertProjDepthToView( float z )
{
  return (1.f/(z*g_mProjectionInv._34 + g_mProjectionInv._44));
}

[numthreads(NUM_THREADS_1D,NUM_THREADS_1D,1)]
void DepthBoundsCS( uint3 globalIdx : SV_DispatchThreadID,
                    uint3 localIdx  : SV_GroupThreadID,
                    uint3 groupIdx  : SV_GroupID )
{
  uint2 sampleIdx = globalIdx.xy*2;

  // Load four depth samples
  float depth00 = g_SceneDepthBuffer.Load(uint3(sampleIdx.x,
                                                sampleIdx.y,0)).x;
  float depth01 = g_SceneDepthBuffer.Load(uint3(sampleIdx.x,
                                                sampleIdx.y+1,0)).x;
  float depth10 = g_SceneDepthBuffer.Load(uint3(sampleIdx.x+1,
                                                sampleIdx.y,0)).x;
  float depth11 = g_SceneDepthBuffer.Load(uint3(sampleIdx.x+1,
                                                sampleIdx.y+1,0)).x;

  float viewPosZ00 = ConvertProjDepthToView(depth00);
  float viewPosZ01 = ConvertProjDepthToView(depth01);
  float viewPosZ10 = ConvertProjDepthToView(depth10);
  float viewPosZ11 = ConvertProjDepthToView(depth11);

  uint threadNum = localIdx.x + localIdx.y*NUM_THREADS_1D;

  // Use parallel reduction to calculate the depth bounds
  {
    // Parts of the depth buffer that were never written
    // (e.g., the sky) will be zero (the companion code uses
    // inverted 32-bit float depth for better precision).
    float minZ00 = (depth00 != 0.f) ? viewPosZ00 : FLT_MAX;
    float minZ01 = (depth01 != 0.f) ? viewPosZ01 : FLT_MAX;
    float minZ10 = (depth10 != 0.f) ? viewPosZ10 : FLT_MAX;
    float minZ11 = (depth11 != 0.f) ? viewPosZ11 : FLT_MAX;

    float maxZ00 = (depth00 != 0.f) ? viewPosZ00 : 0.0f;
    float maxZ01 = (depth01 != 0.f) ? viewPosZ01 : 0.0f;
    float maxZ10 = (depth10 != 0.f) ? viewPosZ10 : 0.0f;
    float maxZ11 = (depth11 != 0.f) ? viewPosZ11 : 0.0f;

    // Initialize shared memory
    ldsZMin[threadNum] = min(minZ00,min(minZ01,
                                        min(minZ10,minZ11)));
    ldsZMax[threadNum] = max(maxZ00,max(maxZ01,
                                        max(maxZ10,maxZ11)));
    GroupMemoryBarrierWithGroupSync();

    // Minimum and maximum using parallel reduction, with the
    // loop manually unrolled for 8x8 thread groups (64 threads
    // per thread group)
    if (threadNum < 32)
    {
```

```
70        ldsZMin[threadNum] = min(ldsZMin[threadNum],
71                                 ldsZMin[threadNum+32]);
72        ldsZMax[threadNum] = max(ldsZMax[threadNum],
73                                 ldsZMax[threadNum+32]);
74        ldsZMin[threadNum] = min(ldsZMin[threadNum],
75                                 ldsZMin[threadNum+16]);
76        ldsZMax[threadNum] = max(ldsZMax[threadNum],
77                                 ldsZMax[threadNum+16]);
78        ldsZMin[threadNum] = min(ldsZMin[threadNum],
79                                 ldsZMin[threadNum+8]);
80        ldsZMax[threadNum] = max(ldsZMax[threadNum],
81                                 ldsZMax[threadNum+8]);
82        ldsZMin[threadNum] = min(ldsZMin[threadNum],
83                                 ldsZMin[threadNum+4]);
84        ldsZMax[threadNum] = max(ldsZMax[threadNum],
85                                 ldsZMax[threadNum+4]);
86        ldsZMin[threadNum] = min(ldsZMin[threadNum],
87                                 ldsZMin[threadNum+2]);
88        ldsZMax[threadNum] = max(ldsZMax[threadNum],
89                                 ldsZMax[threadNum+2]);
90        ldsZMin[threadNum] = min(ldsZMin[threadNum],
91                                 ldsZMin[threadNum+1]);
92        ldsZMax[threadNum] = max(ldsZMax[threadNum],
93                                 ldsZMax[threadNum+1]);
94      }
95    }
96    GroupMemoryBarrierWithGroupSync();
97
98    float minZ = ldsZMin[0];
99    float maxZ = ldsZMax[0];
100   float halfZ = 0.5f*(minZ + maxZ);
101
102   // Calculate a second set of depth values: the maximum
103   // on the near side of Half Z and the minimum on the far
104   // side of Half Z
105   {
106     // See the companion code for details
107     ...
108   }
109
110   // The first thread writes to the depth bounds texture
111   if(threadNum == 0)
112   {
113     float maxZ2 = ldsZMax[0];
114     float minZ2 = ldsZMin[0];
115     g_DepthBounds[groupIdx.xy] = float4(minZ,maxZ2,minZ2,maxZ);
116   }
117 }
```

Listing 1.5. Depth bounds using parallel reduction.

As noted in [Harris 07] and [Engel 14], an optimized parallel reduction implementation requires each thread to work on more than one source value. For the code in Listing 1.5, each thread loads four depth samples in a 2×2 grid instead of just a single sample. However, this requires the thread layout to be 8×8 for 16×16-pixel tiles. That is, the parallel reduction must be executed in a separate compute shader. However, even with the extra overhead of an additional pass, the four-samples-per-thread method is faster than keeping the parallel reduction in the culling compute shader but only loading a single sample per thread.

Referring back to Figure 1.6, the "Modified Half Z, AABB, Parallel Reduction" curve is the fastest method throughout. For 1024 lights, the baseline code executes in 3.97 ms, whereas this final optimized version takes 3.52 ms, a reduction of roughly half a millisecond. This represents an 11% decrease in execution time compared to the baseline.

1.5 Unreal Engine 4 Results

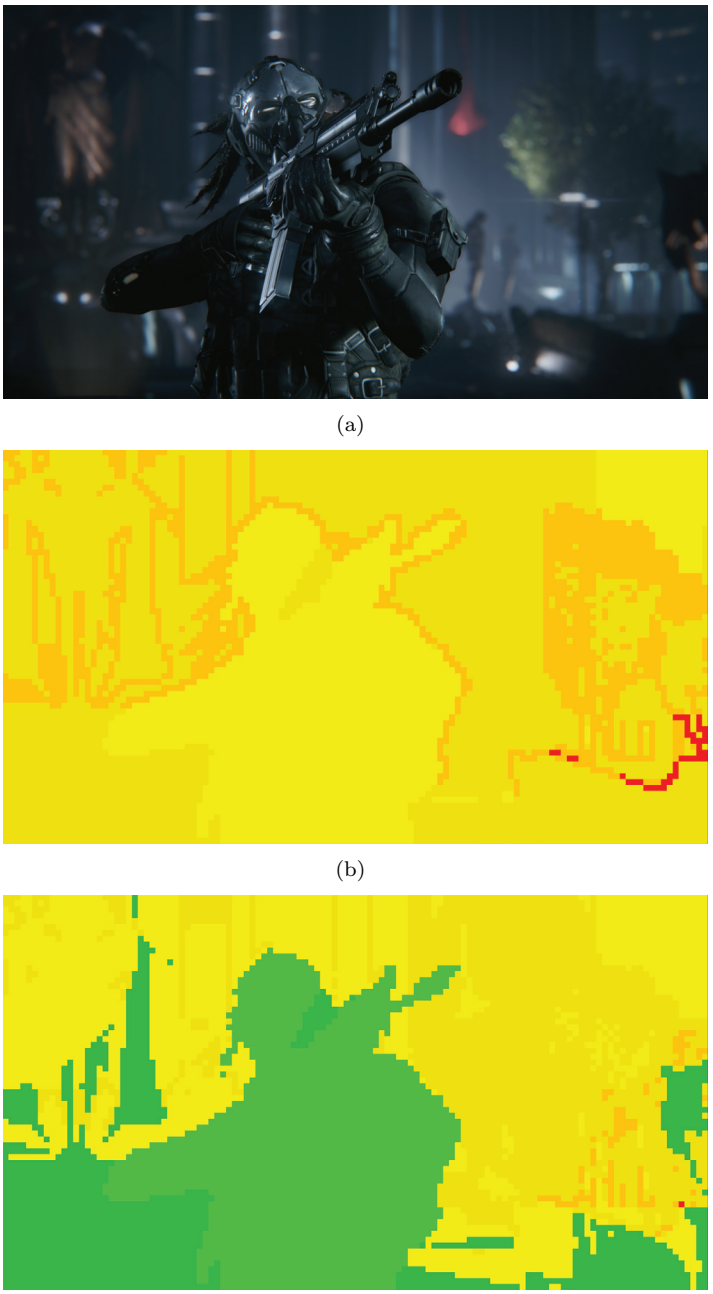
Results to this point have been gathered using the companion code for this chapter. This section presents results using the Unreal Engine 4 *Infiltrator* real-time demo. Unreal Engine 4 is a leading real-time rendering engine that implements the tiled-deferred technique. The *Infiltrator* demo allows results to be gathered using state-of-the-art visuals.

Figures 1.8 and 1.9 show two examples of the per-tile light count reduction achieved by using the Modified Half Z method with AABBs. Note the results for baseline tiled culling, which uses an implementation similar to Section 1.3. In each example, high-light-count areas appear along the silhouette of the infiltrator character, where the transition from foreground to background causes depth discontinuities. These areas are eliminated in the optimized version. In addition, the tighter tile bounding volumes from AABBs with small depth ranges reduce light counts overall.

Figure 1.10 shows the GPU execution time improvement of the optimized method (Modified Half Z with AABBs using parallel reduction for the depth ranges) compared to the baseline implementation similar to Section 1.3. For tiled deferred, the execution time includes the three parts of Algorithm 1.1 (depth bounds calculation, tile bounding volume construction, and intersection testing), as well as the lighting calculations. As shown in Figure 1.10, the optimized version is substantially faster over the entire *Infiltrator* demo. Average cost of the baseline implementation is 5.17 ms, whereas the optimized average cost is 3.74 ms, a reduction of 1.43 ms, or roughly 28% faster.

1.5.1 Standard Deferred versus Tiled Deferred

Unreal Engine 4 can apply lighting using either standard deferred or tiled deferred, offering the opportunity to compare the performance of the two methods. Figure 1.11 shows the GPU execution time improvement of the optimized tiled-deferred method compared to the standard-deferred method. Note that, while tiled deferred is usually faster in the demo, there are areas where standard deferred is faster (i.e., the negative values in the chart). Recall that the primary lighting performance concern with standard deferred is the extra bandwidth consumed when blending overlapping lights. In areas without much light overlap, the savings from tiled deferred's single-pass lighting might not outweigh the cost



(a)

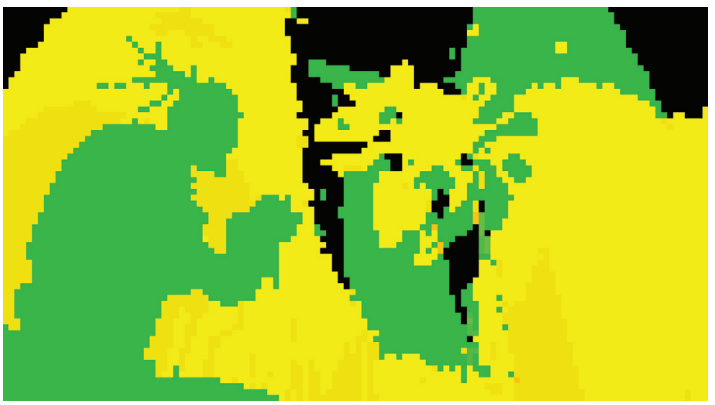
(b)

(c)

Figure 1.8. Unreal Engine 4 *Infiltrator* demo: Example 1. (a) Scene render. (b) Baseline tiled culling. (c) Modified Half Z with AABBs.

(a)

(b)



(c)

Figure 1.9. Unreal Engine 4 *Infiltrator* demo: Example 2. (a) Scene render. (b) Baseline tiled culling. (c) Modified Half Z with AABBs.

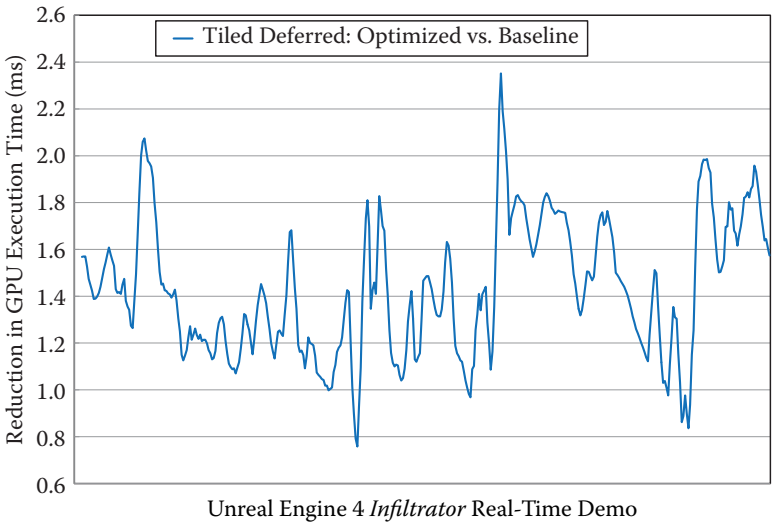


Figure 1.10. Unreal Engine 4 tiled-culling execution time improvement for the optimized version compared to the baseline implementation. Performance was measured over the entire *Infiltrator* demo at 1920×1080.

of calculating the depth bounds and performing the per-tile culling. However, averaged over the entire demo, tiled deferred is still faster overall. Specifically, the average cost of standard deferred is 4.28 ms, whereas the optimized tiled-deferred average cost is 3.74 ms, a reduction of 0.54 ms, or roughly 13% faster.

It is natural to wonder exactly how many lights are needed in a scene with "many lights" before tiled deferred is consistently faster than standard deferred. The answer will depend on several factors including the depth complexity of the scene and the amount of light overlap. For the *Infiltrator* demo, Figure 1.12 is a scatterplot of the data used to generate Figure 1.11 plotted against the number of lights processed during that particular frame. The demo uses a wide range of light counts, from a low of 7 to a high of 980. The average light count is 299 and the median is 218.

For high light counts (above 576), tiled deferred has either comparable or better performance, and is often significantly faster. For example, for counts above 640, tiled deferred is 1.65 ms faster on average. Conversely, for low light counts (below 64), standard deferred is faster. For light counts above 64 but below 576, the situation is less clear from just looking at the chart. Standard deferred values appear both above and below tiled deferred in this range. However, it is worth noting that tiled deferred comes out ahead on average over each interval on the "Number of Lights" axis (i.e., $[0, 64]$, $[64, 128]$, $[128, 192]$, etc.) except $[0, 64]$.

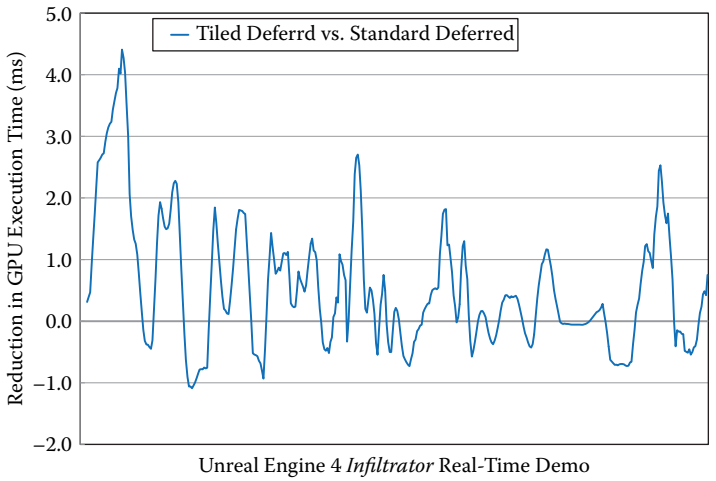


Figure 1.11. Unreal Engine 4 optimized tiled-deferred execution time improvement compared to standard deferred. Performance was measured over the entire *Infiltrator* demo using 1920 × 1080 screen resolution.

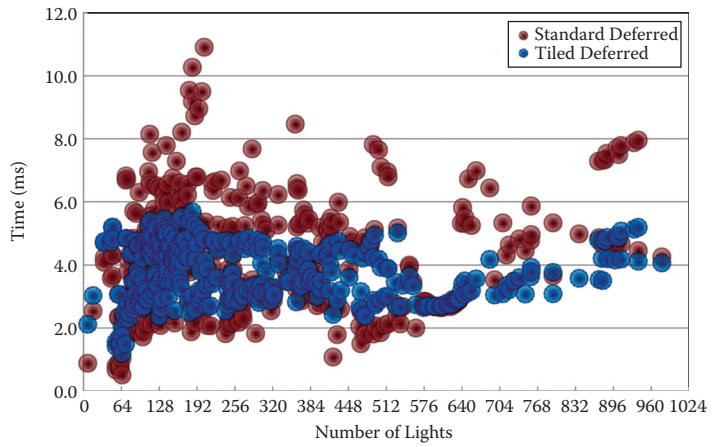


Figure 1.12. Unreal Engine 4 optimized tiled deferred versus standard deferred. GPU execution time versus number of lights. Performance was measured over the entire *Infiltrator* demo at 1920 × 1080.

To get a clearer picture of average performance, Figure 1.13 applies a moving average to the data in Figure 1.12. The data shows that, while standard deferred is 0.76 ms faster on average for light counts of 70 and below, tiled deferred is

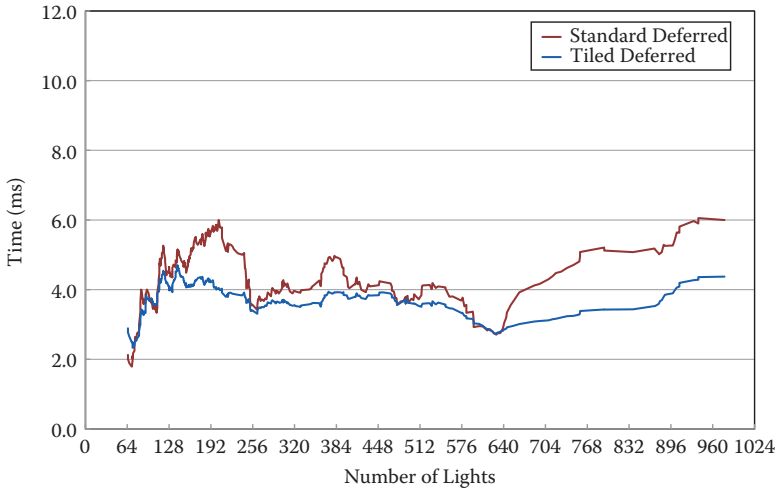


Figure 1.13. Unreal Engine 4 optimized tiled deferred versus standard deferred. GPU execution time versus number of lights. A moving average was applied to the data in Figure 1.12 to show overall trends.

on par with or faster than standard deferred for above 70 lights. Thus, for the particular case of the *Infiltrator* demo, 70 is the threshold for when tiled deferred is consistently faster than (or at least comparable to) standard deferred.

Referring back to Figure 1.12, another thing to note about the data is that the standard deviation is lower for tiled deferred. Specifically, the standard deviation is 1.79 ms for standard deferred and 0.90 ms for tiled deferred, a 50% reduction. Note that worst-case performance is also much better for tiled deferred, with no tiled deferred data point appearing above the 6.0 ms line. That is, in addition to getting faster performance on average, tiled deferred also offers more consistent performance, making it easier to achieve a smooth framerate.

1.6 Conclusion

This chapter presented an optimized compute-based tiled-culling implementation for scenes with many dynamic lights. The technique allows forward rendering to support such scenes with high performance. It also improves the performance of deferred rendering for these scenes by reducing the average cost to calculate lighting, as well as the worst-case cost and standard deviation. That is, it provides both faster performance (on average) and more consistent performance, avoiding the bandwidth bottleneck from blending overlapping lights. For more details, see the companion code.

1.7 Acknowledgments

Many thanks to the rendering engineers at Epic Games, specifically Brian Karis for the idea to use AABBs to bound the tiles and Martin Mittring for the initial implementation of AABBs and for the Modified Half Z method. Thanks also go out to Martin for providing feedback for this chapter. And thanks to the Epic rendering team and Epic Games in general for supporting this work.

Bibliography

[Andersson 09] Johan Andersson. "Parallel Graphics in Frostbite—Current and Future." Beyond Programmable Shading, SIGGRAPH Course, New Orleans, LA, August 3–7, 2009.

[Engel 14] Wolfgang Engel. "Compute Shader Optimizations for AMD GPUs: Parallel Reduction." *Diary of a Graphics Programmer*, http://diaryofagraphicsprogrammer.blogspot.com/2014/03/compute-shader-optimizations-for-amd.html, March 26, 2014.

[Harada et al. 12] Takahiro Harada, Jay McKee, and Jason C.Yang. "Forward+: Bringing Deferred Lighting to the Next Level." Paper presented at Eurographics, Cagliari, Italy, May 13–18, 2012.

[Harris 07] Mark Harris. "Optimizing Parallel Reduction in CUDA." NVIDIA, http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf, 2007.

[Lauritzen 10] Andrew Lauritzen. "Deferred Rendering for Current and Future Rendering Pipelines." Beyond Programmable Shading, SIGGRAPH Course, Los Angeles, CA, July 25–29, 2010.

[Lauritzen 12] Andrew Lauritzen. "Intersecting Lights with Pixels: Reasoning about Forward and Deferred Rendering." Beyond Programmable Shading, SIGGRAPH Course, Los Angeles, CA, August 5–9, 2012.

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