19 Massive Number of Shadow-Casting Lights with Layered Rendering

Daniel Rákos

19.1 Introduction

Shadow map generation is one of the most time-consuming rendering tasks that today's graphics applications have to deal with due to the high number of dynamic light sources used. Deferred rendering techniques have provided a reasonable answer for handling a large number of dynamic light sources in linear time because they work independently of the scene complexity. However, shadow map generation is still an $O(nm)$ time complexity task where n is the number of light sources and m is the number of objects.

This chapter explores the possibility of taking advantage of some of the latest GPU technologies to decrease this time complexity by using *layered rendering* to render multiple *shadow maps* at once using the same input geometry. Layered rendering will enable us to decrease the vertex attribute–fetching bandwidth requirements and the vertex processing time to $O(m)$ time complexity.

While this approach will still require $O(nm)$ time complexity for rasterizing and fragment processing, this generally takes far less time in practice, as usually, the light volumes don't overlap completely, so most of the geometric primitives will be culled early by the rasterizer. We will also investigate whether the required rasterizer throughput can be further decreased by doing view-frustum culling in the *geometry shader* performing the layered rendering.

I will present a reference implementation for both traditional and *layered shadow map rendering* that enables us to provide performance comparison results for the two approaches in different scenes with various scene complexities and various numbers of light sources and types. In addition, I will provide measurements about how the shadow map resolution affects the performance of the traditional method and our new technique.

Performance measurements are executed both for client-side and server-side workloads because, besides decreasing the geometry processing requirements of shadow rendering, the technique also drastically decreases the number of necessary state changes and draw commands to render multiple shadow maps, thus providing an edge for CPU-bound applications.

While the technique primarily targets OpenGL 4.x–capable GPUs, we present how the same technique can be implemented with some caveats for GPUs with only OpenGL 3.x capabilities.

Further, I will present some special use cases of layered shadow map rendering that can be used to render *shadow cube maps* and *cascaded shadow maps* with a single draw call, and I will briefly present how the technique can be altered to accelerate the generation of reflection maps and reflection cube maps in a similar fashion.

Finally, I will discuss the limitations of the presented algorithm, explicitly mentioning those imposed by hardware limitations of GPU implementations.

19.2 Traditional Shadow Map Rendering in OpenGL

Shadow mapping or projective shadowing is an image-space rendering technique introduced by [Williams 78] that became the de facto standard for performing shadow rendering in real time and offline graphics applications.

The principle of shadow mapping is that if we view the scene from a light source's position, all the object points we can see from there appear in light and anything behind those is in shadow. Based on this, the algorithm works in the following way:

- We render the scene from the light source's point of view to a depth buffer using the well-known z-buffer visibility algorithm.
- When we render the scene from the camera's point of view, by comparing the distance of the light source to any point of the surface with the sampled value from the depth buffer corresponding to the point, we can decide whether the given point is in light or in shadow. See Figure 19.1.

Modern graphics processors provide hardware support for this technique in the form of two features:

- Providing a mechanism to store depth-buffer images in textures.
- Providing a mechanism to compare a reference depth value with a depth value stored in a depth texture.

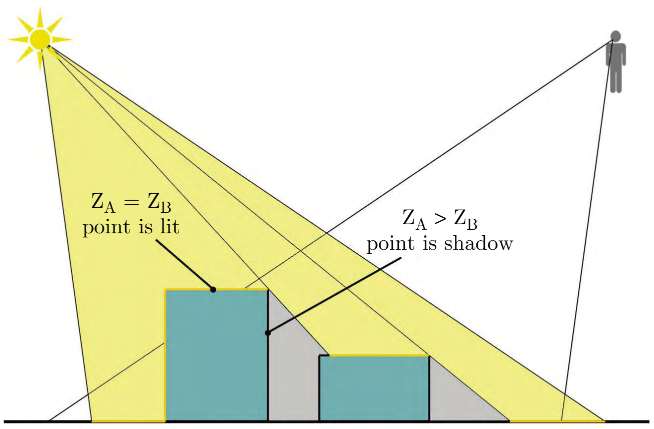

Figure 19.1. Illustration of the shadow mapping algorithm where Z_A is the distance of the fragment from the light source and Z_B is the value stored in the depth buffer generated in the first pass.

Both of these features are available as extensions [Paul 02] and have been part of the OpenGL specification since Version 1.4. These extensions provide a fixed-function mechanism that returns the boolean result of a comparison between the depth texture texel value and a reference value derived from the texture-coordinate set used for the fetch. Although in modern OpenGL these are all done with shaders, GLSL provides depth comparison texture lookup functions to perform the fixed-function comparison of the depth values.

Putting everything together, in order to implement shadow mapping for a single light source, we have to use a two-pass rendering algorithm. In the first pass, we set up the light's view-projection matrix for use in our shadow rendering *vertex shader*. Then we prepare the framebuffer for depth-texture rendering. Finally, we simply draw our scene without textures, materials, or any additional configuration, as we are interested only in the generated depth values (see Listing 19.1). There are no special requirements about the shaders used in the shadow map generation pass; we only need a single vertex shader, with no further shader stages, that performs the exact same transformations as it would do in case of regular scene rendering.

The framebuffer object used in Listing 19.1 is configured with only a single-depth attachment with no color attachments. The code needed to set up the depth texture and the framebuffer object used in Listing 19.1 is presented in Listing 19.2.

In the second pass, we use the depth texture generated in the first pass as a texture input, and we will also need the light's view-projection matrix to reconstruct each fragment's position in the light view's clip space to perform the texture lookup and the depth comparison (see Listing 19.3). Obviously, this pass differs based on whether a forward renderer or a deferred renderer is in use, as the rendered geometry is either

```
/* bind the framebuffer that has only a depth texture attachment */
glBindFramebuffer(GL_FRAMEBUFFER , depth_fbo);

/* we must ensure that depth testing and depth writes are enabled */
glEnable(GL_DEPTH_TEST);
glDepthMask(GL_TRUE);

/* clear the depth buffer before proceeding */
glClear(GL_DEPTH_BUFFER_BIT);

/* bind the shadow map rendering program which has only a vertex shader attached */
glUseProgram(shadow_po);

/* bind the uniform buffer containing the view-projection matrix of the light */
glBindBufferBase(GL_UNIFORM_BUFFER, 0, lightVP_ubo);

/* render the scene as usual */
..................
```

Listing 19.1. Traditional shadow map generation pass using OpenGL 3.3+.

the whole scene or a primitive that represents the light (a light volume or a full screen quad), respectively.

Usually, when we have multiple light sources, we need to execute both passes for each light source separately. Sometimes, the first pass needs to be executed multiple times even for a single light source, as in the case of omnidirectional light sources or if we would like to use cascaded shadow maps as presented by [Dimitrov 07] for directional light sources.

```
/* create the depth texture with a 16-bit depth internal format */
glGenTextures(1, &depth_texture);
glBindTexture(GL_TEXTURE_2D , depth_texture);
glTexImage2D(GL_TEXTURE_2D , 0, GL_DEPTH_COMPONENT16 , width , height , 0,
GL_DEPTH_COMPONENT, GL_FLOAT, NULL);

/* set up the appropriate filtering and wrapping modes */
glTexParameteri(GL_TEXTURE_2D , GL_TEXTURE_WRAP_S,     GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D , GL_TEXTURE_WRAP_T,     GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D , GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D , GL_TEXTURE_MIN_FILTER, GL_NEAREST);

/* create the framebuffer object and bind it */
glGenFramebuffers(1, &depth_fbo);
glBindFramebuffer(GL_FRAMEBUFFER , depth_fbo);

/* disable all color buffers */
glDrawBuffer(GL_NONE);

/* attach the depth texture as depth attachment */
glFramebufferTexture(GL_FRAMEBUFFER , GL_DEPTH_ATTACHMENT, depth_texture , 0);
```

Listing 19.2. Setting up a framebuffer object for shadow map rendering.

```
/* bind the target framebuffer that we want the lit scene to be rendered to */
glBindFramebuffer(GL_FRAMEBUFFER, final_fbo);
/* bind the shadow map as texture input */
glBindTexture(GL_TEXTURE_2D, depth_texture);
/* bind the light shader program with shadow mapping support */
glUseProgram(light_po);
/* bind the uniform buffer containing the view-projection matrix of the light */
glBindBufferBase(GL_UNIFORM_BUFFER, 0, lightVP_ubo);
/* render the scene or light as usual */
..................
```

Listing 19.3. Traditional shadow mapping pass using OpenGL 3.3+.

While shadow map generation is a lightweight rendering pass, especially for fragment processing as we don't have to compute per-fragment shading or other sophisticated effects, it still has a significant overhead for vertex and command processing. This is why we need a more streamlined algorithm for rendering shadow maps for multiple light sources.

19.3 Our Shadow Map Generation Algorithm

The latest generations of GPUs brought several hardware features that can be used to decrease the vertex processing and API overhead of the generation of multiple shadow maps. One of these features is the support for 1D and 2D *texture arrays* [Brown 06]. A texture array is actually an array of textures that have the same properties (internal format, size, etc.) and allows a programmable shader to access them through a single texture unit using a single coordinate vector. This means that we can access a 2D texture array with an (S, T, L) coordinate set where L selects the layer of the texture array and the (S, T) coordinate set is used to access that single layer of the array texture as if it were a regular 2D texture. This GPU generation does not allow us just to sample texture arrays but also to render to them.

From our point of view, the other important feature that was introduced with this hardware generation is the geometry shader [Brown and Lichtenbelt 08]. This new shader stage allows us to process OpenGL primitives as a whole, but it is capable of more: it can generate zero, one, or more output primitives based on the input primitive and also allows us to select the target texture layer to use for rasterization in the case of a layered rendering target. These features enable us to implement a more sophisticated shadow map generation algorithm.

In order to implement our shadow map generation algorithm, only small changes are required to be made to the traditional method. The first thing is to replace our 2D depth texture with a 2D depth texture array and to set up the framebuffer for layered rendering (this later step actually does not need any change compared to the code presented in Listing 19.2). The modified code is shown in Listing 19.4.

```
/* create the depth texture with a 16-bit depth internal format */
glGenTextures(1, &depth_texture);
glBindTexture(GL_TEXTURE_2D_ARRAY, depth_texture);
glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_DEPTH_COMPONENT16, width, height,
number_of_shadow_maps, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);

/* set up the appropriate filtering and wrapping modes */
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

/* create the framebuffer object and bind it */
glGenFramebuffers(1, &depth_fbo);
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);

/* disable all color buffers */
glDrawBuffer(GL_NONE);

/* attach the depth texture as depth attachment */
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depth_texture, 0);
```

Listing 19.4. Setting up a framebuffer object for layered shadow map rendering.

In order to emit all incoming geometric primitives to all the layers of the depth texture array, we have to inject a geometry shader into our shadow map rendering program that will do the task. We would like to use a separate view-projection matrix for each render target layer because they belong to different light sources; thus, the view-projection transformation has to be postponed to the geometry shader stage.

Here, we can take advantage of one more feature introduced with OpenGL 4–capable GPUs: *instanced geometry shaders*. With traditional OpenGL 3 geometry shaders, we could only emit multiple output primitives sequentially. As usual, sequential code is not really well suited for highly parallel processor architectures like the modern GPUs. Instanced geometry shaders allow us to execute multiple instances of the geometry shader on the same input primitive, thus making it possible to process and emit the output primitives in parallel. The implementation of such a geometry shader for emitting the incoming geometry to a total of 32 output layers is presented in Listing 19.5. This means that we can render with it to 32 depth textures at once. While this code contains a loop, the loop is very likely to be unrolled by the shader compiler, but more cautious developers can unroll the loop.

Let's discuss the effect the postponement of the view-projection transformation to the geometry shader can have on the usability of the presented algorithm. One may say that this can introduce performance issues because the transformation is executed multiple times on a single vertex; this can be especially problematic when skeletal animation or another sophisticated vertex transformation algorithm is used. While it is true that the light's view-projection may be executed multiple times on the same vertex, this is a fixed cost, and it may be done anyway multiple times even if we want to generate our shadow maps in the traditional way because of the limited storage

```
#version 420 core

layout(std140, binding = 0) uniform lightTransform {
    mat4 VPMatrix[32];
} LightTransform;

layout(triangles, invocations = 32) in;
layout(triangle_strip, max_vertices = 3) out;

layout(location = 0) in vec4 vertexPosition[];

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    for (int i=0; i<3; ++i) {
        gl_Position = LightTransform.VPMatrix[gl_InvocationID] * vertexPosition[i];
        gl_Layer = gl_InvocationID;
        EmitVertex();
    }
    EndPrimitive();
}
```

Listing 19.5. OpenGL 4.2 instanced geometry shader that renders input geometry to 32 output layers.

of the post-transform vertex cache. However, model transformations like skeletal animation don't have to be moved to the geometry shader, as they are independent of the view and the projection, so those should be kept in the vertex shader.

So what else do we need to change in our code to make the layered shadow map generation algorithm work? Nothing, except that when rendering the scene to the shadow maps, our culling algorithms have to be aware that we are rendering our geometry not only from a single view but from multiple views; thus, these algorithms should skip the rendering of a scene node only when it is not visible from any of these views.

19.4 Performance

We use a Radeon HD5770 and an Athlon X2 4000+ for performance testing. The basic scenario is to render the shadow of the Stanford dragon model, which has a total of 35,577 triangles in our case.

The scene is rendered to up to 32 shadow maps, including the depth buffer clear, making the whole scene visible in the depth texture with traditional and layered shadow map generation. The resulting shadow maps will look like the one shown in Figure 19.2. Also, we try multiple shadow map resolutions to see how they affect the rendering performance.

The vertex shader is shown in Listing 19.6; the preprocessor directive `LAYERED` is defined only in the case of our layered shadow map generation algorithm. The

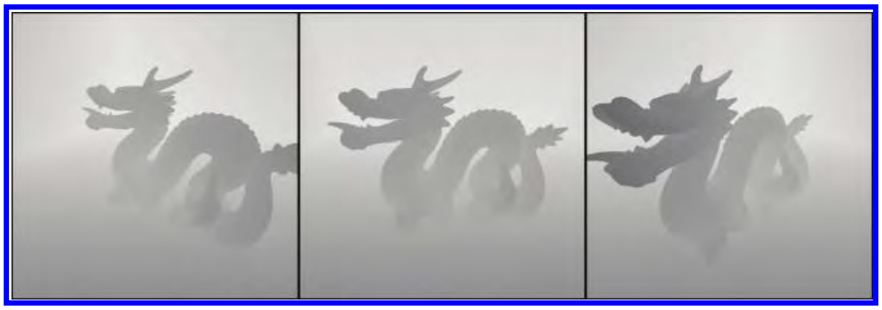

Figure 19.2. Sample depth maps of the Stanford dragon model consisting of 35,577 triangles from various light positions and directions.

geometry shader used is equivalent to the one presented earlier in Listing 19.5 with the number of geometry shader invocations set to the number of shadow maps that have to be rendered. The results of the GPU time needed to render the shadow maps for each particular resolution were measured using timer queries [Daniell 10] and can be seen in Figure 19.3.

```
#version 420 core

layout(location = 0) in vec3 inVertexPosition;

#ifdef LAYERED
layout(location = 0) out vec4 vertexPosition;
#endif

layout(std140, binding = 1) uniform transform {
    mat4 ModelMatrix;
} Transform;

#ifndef LAYERED
layout(std140, binding = 0) uniform lightTransform {
    mat4 VPMatrix;
} LightTransform;
out gl_PerVertex {
      vec4 gl_Position;
};
#endif

void main(void) {
#ifdef LAYERED
    vertexPosition = Transform.ModelMatrix * vec4(inVertexPosition, 1.f);
#else
    gl_Position = LightTransform.VPMatrix * (Transform.ModelMatrix * vec4(↩
         inVertexPosition, 1.f));
#endif
}
```

Listing 19.6. Shadow rendering vertex shader used for performance measurements.

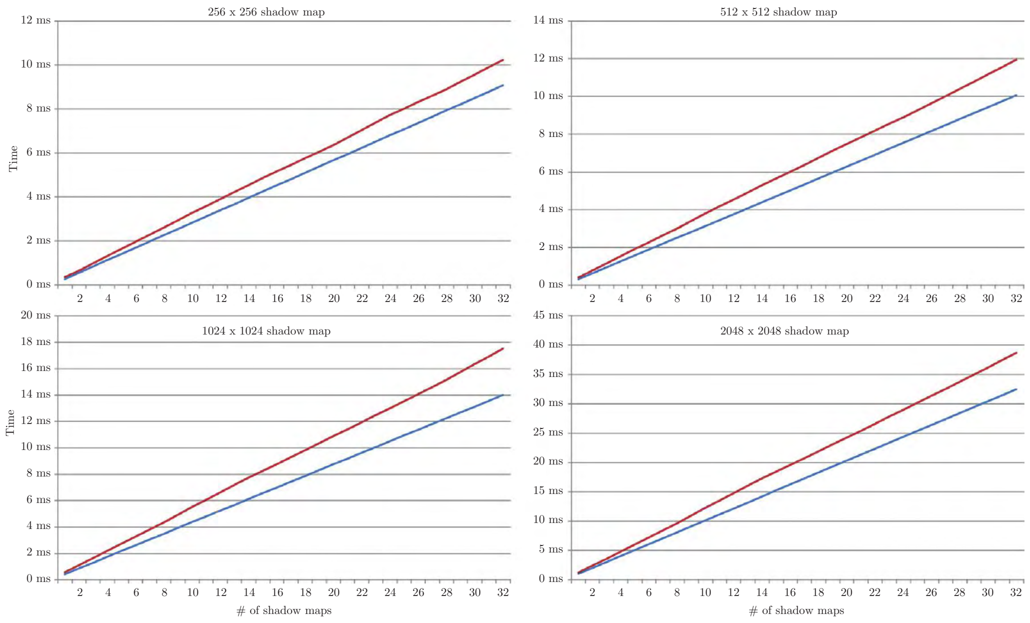

Figure 19.3. The GPU time required to render 1 to 32 shadow maps using traditional shadow map generation (blue) and our layered method (red). The resulting shadow maps are of size 256 × 256 (top left), 512 × 512 (top right), 1024 × 1024 (bottom left), and 2048 × 2048 (bottom right). Lower values are better.

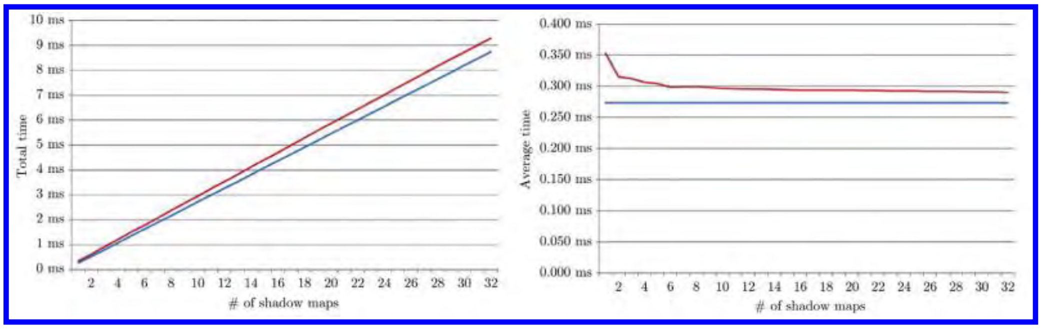

Figure 19.4. The total GPU time required to render 1 to 32 shadow maps (left) and the average GPU time required per shadow map (right) using traditional shadow map generation (blue) and our layered method (red). Lower values are better.

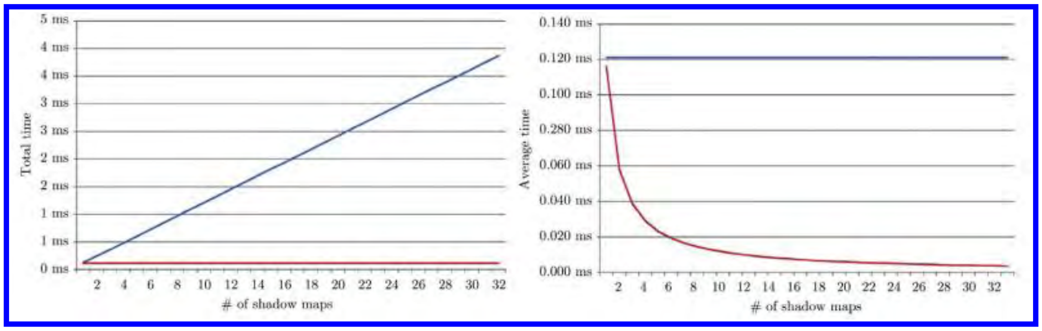

Figure 19.5. The total CPU time required to render 1 to 32 shadow maps (left) and the average CPU time required per shadow map (right) using traditional shadow map generation (blue) and our layered method (red). Lower values are better.

Figure 19.3 shows that when using such a simple vertex shader, the amount of GPU work saved by using layered rendering does not outweigh the overhead of the introduced geometry shader stage. Actually, it has roughly 10% lower performance than the traditional method. Also, the shadow map resolution has very little effect on the relative performance of the two techniques. The reason for this is that the fragment processing cost is equivalent in both cases. We will disable rasterization in our further measurements so that we can concentrate on the geometry processing time. This can be easily done by using the following command before our shadow map rendering:

```
glEnable(GL_RASTERIZER_DISCARD);
```

Figure 19.4 shows the performance results of the shadow map generation without rasterization. We also provide a separate chart to present the average GPU time required for generating a single shadow map.

When using such a simple vertex shader, the overhead of layered shadow map rendering makes the technique suboptimal from a GPU-resource point of view, but the time savings on the CPU side show the strength of layered rendering. As can be seen in Figure 19.5, the CPU time required to generate multiple shadow maps can go off the charts even though there is only a single draw command that renders the whole scene. Contrarily, our layered method has a constant CPU cost indifferent of the number of shadow maps generated.

19.4.1 Performance with Complex Vertex Shaders

The vertex processing requirements in our tests were too optimistic because we used a minimal vertex attribute setup of 12 bytes per vertex (3 floats for vertex position), and our shaders only perform two matrix-vertex multiplications, one for the model

```
#version 420 core

layout(location = 0) in vec3 inVertexPosition;
layout(location = 1) in ivec4 inBoneIndex;
layout(location = 2) in vec4 inBoneWeight;

#ifdef LAYERED
layout(location = 0) out vec4 vertexPosition;
#endif

layout(std140, binding = 1) uniform boneTransform {
    mat3 BoneMatrix[64];
} BoneTransform;

#ifndef LAYERED
layout(std140, binding = 0) uniform lightTransform {
    mat4 VPMatrix;
} LightTransform;
out gl_PerVertex {
      vec4 gl_Position;
};
#endif

void main(void) {
    vec3 vertex = (BoneTransform.BoneMatrix[inBoneIndex.x] * inVertexPosition) * ↩
         inBoneWeight.x;
    if (inBoneIndex.y != 0xFF) {
        vertex += (BoneTransform.BoneMatrix[inBoneIndex.y] * inVertexPosition) * ↩
             inBoneWeight.y;
        if (inBoneIndex.z != 0xFF) {
            vertex += (BoneTransform.BoneMatrix[inBoneIndex.z] * inVertexPosition) *↩
                 inBoneWeight.z;
            if (inBoneIndex.w != 0xFF) {
                vertex += (BoneTransform.BoneMatrix[inBoneIndex.w] * ↩
                     inVertexPosition) * inBoneWeight.w;
            }
        }
    }
#ifdef LAYERED
    vertexPosition = vec4(vertex, 1.f);
#else
    gl_Position = LightTransform.VPMatrix * vec4(vertex, 1.f));
#endif
}
```

Listing 19.7. Shadow rendering vertex shader that performs skeletal animation with up to four bones per vertex.

transformation and one for the view-projection transformation. Now, let's try a real-life scenario by implementing a vertex shader that performs simple *skeletal animation*. We will provide a maximum of 64 bone matrices and a vertex attribute setup of 32 bytes per vertex (3 floats for vertex position, 4 bytes for bone indices, and 4 floats for bone weights), which provides us up to four bone matrices per vertex with no model transformation. Based on this, our vertex shader will look like the one presented in Listing 19.7.

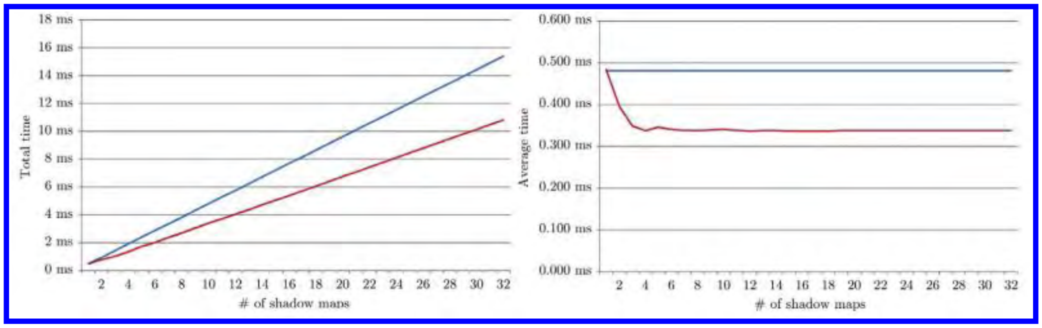

Figure 19.6. The total GPU time required to render 1 to 32 shadow maps (left) and the average GPU time required per shadow map (right) using traditional shadow map generation (blue) and our layered method (red) when using a vertex shader that performs skeletal animation. Lower values are better.

When we use the skeletal animation implementation instead of the simple vertex shader, layered shadow map generation outperforms the traditional method when rendering to more than two depth textures, saving about 30% of GPU time when we render three or more depth textures at once. It has diminishing performance difference compared to the traditional method even in the case of a single shadow map. See Figure 19.6.

Of course, one may say that skeletal animation is usually not applied to all the geometry rendered in a scene. But we also assume one more thing: the whole geometry is visible in the final rendering. While modern visibility-determination algorithms are very efficient, the CPU-based culling algorithms are usually rather coarse and conservative. This means that in many cases, a reasonable amount of geometry fed to the GPU does not contribute to the final image either due to occlusion or because the geometry falls out of the view's frustum.

19.4.2 View Frustum Culling Optimization

In the next test, we render the Stanford dragon model four times using *geometry instancing*, from which only one instance is visible from the light's perspective. With this, we can simulate a more realistic scenario when the input geometry is just partially visible. We won't do any sophisticated vertex transformation techniques like skeletal animation but simply use a per-instance model transformation matrix. The source code for the vertex shader is shown in Listing 19.8.

In addition to the geometry shader presented in Listing 19.5, used earlier for performing the layered rendering, we also perform tests with an alternative geometry shader shown in Listing 19.9 that performs conservative *view frustum culling* and emits the incoming triangles only when the triangle lies in the light's frustum.

```
#version 420 core

layout(location = 0) in vec3 inVertexPosition;

#ifdef LAYERED
layout(location = 0) out vec4 vertexPosition;
#endif

layout(std140, binding = 1) uniform transform {
    mat4 ModelMatrix[4];
} Transform;

#ifndef LAYERED
layout(std140, binding = 0) uniform lightTransform {
    mat4 VPMatrix;
} LightTransform;

out gl_PerVertex {
    vec4 gl_Position;
};
#endif

void main(void) {
#ifdef LAYERED
    vertexPosition = Transform.ModelMatrix[gl_InstanceID] * vec4(inVertexPosition, ↩
        1.f);
#else
    gl_Position = LightTransform.VPMatrix * (Transform.ModelMatrix[gl_InstanceID] * ↩
        vec4(inVertexPosition, 1.f));
#endif
}
```

Listing 19.8. Simplistic shadow rendering vertex shader with geometry instancing support.

As geometry shaders are usually output bound, we expect to get a reasonable performance increase from our new geometry shader. This optimization can reduce the

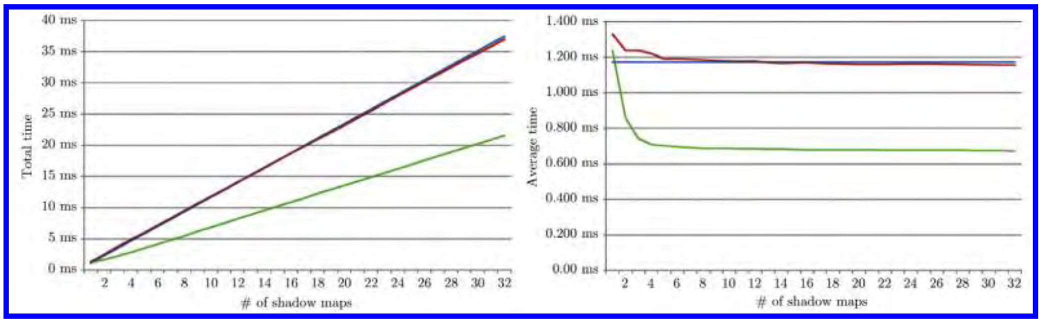

Figure 19.7. The total GPU time required to render 1 to 32 shadow maps (left) and the average GPU time required per shadow map (right) using traditional shadow map generation (blue), our layered method without view frustum culling (red) and our layered method with view frustum culling (green) using four instances of the scene. Lower values are better.

```
#version 420 core

layout(std140, binding = 0) uniform lightTransform {
    mat4 VPMatrix[32];
} LightTransform;

layout(triangles, invocations = 32) in;
layout(triangle_strip, max_vertices = 3) out;

layout(location = 0) in vec4 vertexPosition[];

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    vec4 vertex[3];
    int outOfBound[6] = int[6]{ 0, 0, 0, 0, 0, 0 };
    for (int i=0; i<3; ++i) {
        vertex[i] = LightTransform.VPMatrix[gl_InvocationID] * vertexPosition[i];
        if (vertex[i].x > +vertex[i].w) ++outOfBound[0];
        if (vertex[i].x < -vertex[i].w) ++outOfBound[1];
        if (vertex[i].y > +vertex[i].w) ++outOfBound[2];
        if (vertex[i].y < -vertex[i].w) ++outOfBound[3];
        if (vertex[i].z > +vertex[i].w) ++outOfBound[4];
        if (vertex[i].z < -vertex[i].w) ++outOfBound[5];
    }

    bool inFrustum = true;
    for (int i=0; i<6; ++i)
        if (outOfBound[i] == 3) inFrustum = false;

    if (inFrustum) {
        for (int i=0; i<3; ++i) {
            gl_Position = vertex[i];
            gl_Layer = gl_InvocationID;
            EmitVertex();
        }
        EndPrimitive();
    }
}
```

Listing 19.9. OpenGL 4.1 instanced geometry shader that performs view frustum culling to determine whether the incoming triangle has to be emitted to a particular layer.

number of vertices emitted by the geometry shader performing the layered rendering by a factor of four. The measurements from this combined performance are shown in Figure 19.7.

Even the naive layered shadow map rendering approach that does not perform view frustum culling reaches the performance of the traditional method when we render at least five to ten shadow maps simultaneously. This is most probably the result of performing geometry instancing in this case so our vertex shader accesses the `gl_InstanceID` built-in variable. This shows how easily vertex shader costs can increase when using even such simple techniques like instancing.

While our original layered rendering geometry shader already provides adequate performance in this use case, what is even more impressive is that performing view frustum culling in the geometry shader almost doubles the performance of our layered shadow map generation algorithm despite the additional cost of the actual culling algorithm.

We've seen that the proposed shadow map generation technique offers a clear performance advantage compared to the traditional method when complex vertex shaders are used or when the visibility determination algorithms performed on the CPU provide only conservative results, but what if neither applies to our case? We still have one more thing that we haven't taken advantage of: *back-face culling*.

19.4.3 Back-Face Culling Optimization

When rendering uniformly tessellated closed and opaque geometry, back-face culling usually halves the amount of geometry rasterized. While OpenGL supports fixed-function back-face culling, it is done after the geometry shader stage, and that is already too late, as our algorithm is usually geometry-shader output-bound. Still, back-face culling can be used for our advantage to further decrease the number of triangles emitted by the layered rendering geometry shader if we perform it manually within the shader itself. Thus, the last version of the performance tests uses the

```
#version 420 core

layout(std140, binding = 0) uniform lightTransform {
  mat4 VPMatrix[32];
  vec4 position[32];
} LightTransform;

layout(triangles, invocations = 32) in;
layout(triangle_strip, max_vertices = 3) out;

layout(location = 0) in vec4 vertexPosition[];

out gl_PerVertex {
  vec4 gl_Position;
};

void main() {
  vec3 normal = cross(vertexPosition[2].xyz - vertexPosition[0].xyz, vertexPosition↩
       [0].xyz - vertexPosition[1].xyz);
  vec3 view = LightTransform.position[gl_InvocationID].xyz - vertexPosition[0].xyz;

  if (dot(normal, view) > 0.f) {
    for (int i=0; i<3; ++i) {
      gl_Position = LightTransform.VPMatrix[gl_InvocationID] * vertexPosition[i];
      gl_Layer = gl_InvocationID;
      EmitVertex();
    }
    EndPrimitive();
  }
}
```

Listing 19.10. OpenGL 4.1 instanced geometry shader that performs back-face culling to determine whether the incoming triangle has to be emitted to a particular layer.

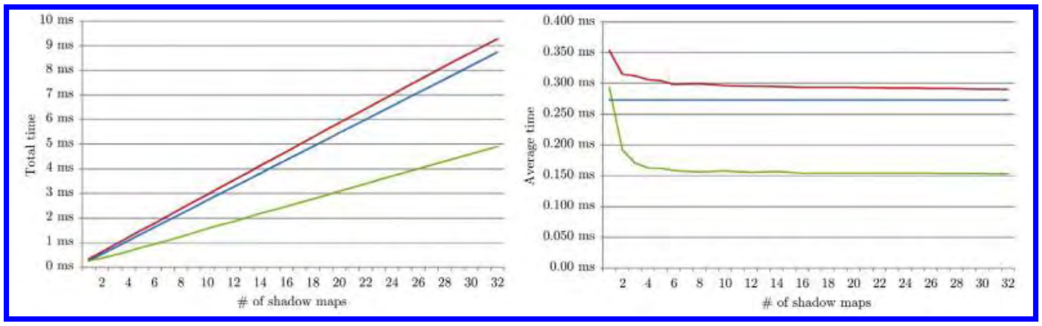

Figure 19.8. The total GPU time required to render 1 to 32 shadow maps (left) and the average GPU time required per shadow map (right) using traditional shadow map generation (blue), our layered method without back-face culling (red) and our layered method with back-face culling (green). Lower values are better.

geometry shader in Listing 19.10. The rest of the configuration is equivalent to the one used in the first test; we use the minimalistic vertex shader show in Listing 19.6 and only a single instance of the scene.

As shown in Figure 19.8, if we use back-face culling, the rendering time is almost halved, and the GPU time required by layered shadow map generation can be more than 40% lower than that of the traditional method, even in the case of a simple vertex shader and no other optimizations.

When layered shadow map rendering is properly implemented, it will outperform the traditional method despite the added overhead of using a geometry shader. The rule of thumb for the geometry shader applies to this technique: it's always worth spending even a lot of ALU instructions to decrease the number of output components emitted by the geometry shader.

In summary, the case of a slightly expensive vertex shader layered shadow map generation is about 30% faster; with a coarsely culled scene, view frustum culling brings roughly 40% speed advantage, and using back-face culling makes our layered technique more than 40% faster than the traditional method even for simple shaders. Obviously, these use cases may happen in combination; thus, we've executed a final test that renders four instances of the scene with skeletal animation done in the vertex shader, and with back-face culling and view frustum culling performed in the geometry shader in the case of layered shadow map generation. The results are shown on Figure 19.9.

The combined results show that our layered technique can take 60% less GPU time to render shadow maps if more than four depth textures are rendered at once. Further, even if in a special configuration, the presented technique does not provide any advantage over the traditional method from the GPU-time point of view, the amount of CPU time saved by batching the shadow map generation passes into a single pass can be a good enough reason to implement it.

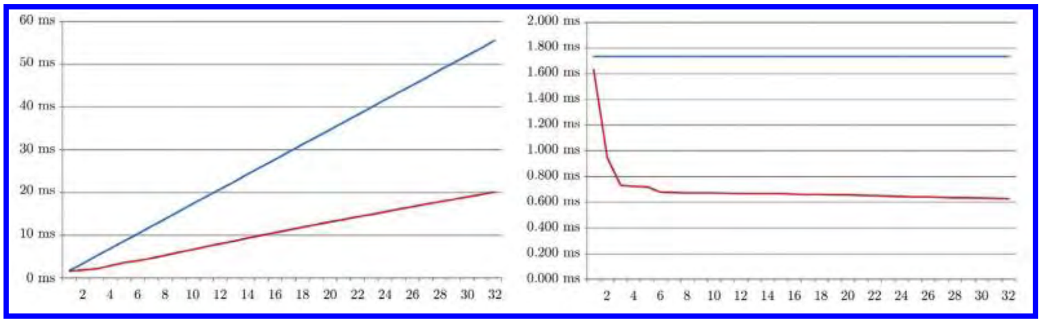

Figure 19.9. The total GPU time required to render 1 to 32 shadow maps (left) and the average GPU time required per shadow map (right) using traditional shadow map generation (blue) and our layered method (red) with back-face culling and view frustum culling. The scene is rendered four times using geometry instancing. Lower values are better.

19.5 Advanced Techniques

The shadow map generation technique we've presented can be directly used to render shadows for spot lights, but the technique can be applied in a similar manner for omnidirectional lights and directional light sources.

For omnidirectional lights, we can choose from several alternatives. We can split the omnidirectional light and use, for example, six 2D shadow maps from the shadow map array for rendering. Our technique naturally extends to omnidirectional light sources. We can also use cube shadow maps, in which case, we will use a cube-map texture array [Haemel 09] instead of a 2D one. Luckily, this does not affect the implementation of the layered shadow map generation algorithm, as each element of a cube-map texture array is actually seen as six 2D layers, and a single layer index can be used to address a specific side of a specific element in the cube-map texture array.

For directional light sources, layered rendering can be used to accelerate cascaded shadow map rendering. Usually, each layer of the cascaded shadow map has the same size because objects farther from the eye's position don't require that much resolution; all of the layers of the cascaded shadow map can be rendered in a single run using the very same algorithm. Actually, it is even possible to render multiple cascaded shadow maps in a single pass using our technique.

The presented algorithm also trivially handles other advanced shadow map generation techniques like the ones presented in [Martin and Tan 04, Stamminger and Drettakis 02, Wimmer et al. 04, Zhang et al. 06]. These techniques tackle the aliasing artifacts of shadow mapping resulting from undersampling by performing various transformations on the scene geometry before rendering it to the shadow map.

Layered rendering is not restricted to rendering shadow maps; it can be used to generate arbitrary textures. This enables us to use the presented rendering technique for the batched generation of reflection maps or reflection cube maps analogous to the shadow map generation for spot lights or omnidirectional light sources. The only difference is that we need to accompany our depth texture array with a color texture array that will be used as the color buffer of the framebuffer object and attach the appropriate fragment shaders to the pipeline.

Also, layered rendering can be advantageous for *stereoscopic rendering*. The only difference is that the view-projection matrices used will be those of the two eyes; the same geometry shaders presented could be used without any significant modification. This use case is a particularly good candidate to take advantage of the performance benefits of layered rendering, as the two eyes' position and orientation are roughly the same at any given time; thus, the list of triangles inside the view frustra are also more or less identical in the two cases.

Finally, layered rendering can be beneficial also for CAD software, which usually display multiple views of the same scene that can be rendered in a single pass this way.

19.6 Limitations

Though layered shadow map generation is very promising, it has limitations, many of them arising from certain hardware restrictions.

One of these limitations is that the layers of an array texture must have the same size. This means that multiple shadow maps, reflection maps, or other textures can be generated in a single pass using our technique only if they have the same size. We'll probably have to wait quite some time until hardware and APIs relax this restriction or provide other means to dispatch rendering to separate, different-sized texture images. Further, this also requires us to be able to select between different viewports besides target layers. While both viewport selection and layer selection is possible on the OpenGL 3.x hardware and API, support is limited to using only one of them at a time, not both. Until these issues are addressed by the hardware vendors, we need to use separate passes for each particular shadow map resolution. As an alternative, a possible solution on current hardware could be the use of a texture atlas that holds multiple depth images. In this case, the shaders should perform the appropriate viewport transformations and manually discard the fragments lying outside the intended viewport.

Another inherent limitation of the presented algorithm is the maximum number of parallel geometry shader invocations supported by hardware. At the time of this writing, this is capped at 32, although this may change in the near future. However, this is not a hard limit on the number of shadow maps our algorithm can handle, as more instances of an incoming primitive can be emitted to separate target layers even from within a single geometry shader invocation using a loop. While we lose some concurrency, we can somewhat increase the upper limit of the number of shadow maps that can be generated in a single run.

There is, however, also a performance perspective on the practical upper limit of the number of shadow maps that should be generated in a single pass. As we've seen in the performance-result charts, on a Radeon HD5770, our technique does not scale that nicely when rendering to more than six shadow maps. The average time per shadow map does still decrease somewhat above this limit, though we don't see as nice a slope as we see for lower number of layers. However, this may be different on other current or future hardware, including higher-end GPUs or hardware from other vendors.

There is also another practical limitation on the number of shadow maps that should be rendered in a single pass. Our technique performs best when there are many primitives that lie in more than one light source's frustum. This means that in the case where the light sources don't overlap well or, in other words, most of the triangles are visible only from a single light source's point of view, the added overhead of executing a geometry shader for every incoming primitive with an invocation count equal to the number of light sources will outweigh the eliminated vertex attribute fetching and vertex processing costs. Thus, care must be taken to group light sources in an appropriate way when their shadow maps are generated with the technique presented here.

19.7 Conclusion

I've presented a rendering technique that takes advantage of OpenGL's layered rendering capability to accelerate the generation of multiple shadow maps. I provided a reference implementation and performance measurements on OpenGL 4.x–capable hardware for both traditional shadow map generation and the layered technique. The layered technique outperforms traditional shadow map generation in most use cases on the GPU and can also reduce the CPU overhead of the generation process to constant time. While this technique primarily targets OpenGL 4.x–capable hardware, I've noted that an implementation is possible for OpenGL 3.x capable hardware.

Visibility determination algorithms like view frustum culling and back-face culling can be used to increase the efficiency of layered rendering and of geometry shaders in general. I've also presented how layered rendering can be used to increase the performance of other advanced rendering techniques like reflection map generation and stereoscopic rendering.

While layered rendering of shadow maps is not popular in the industry yet, I expect that it will soon gain attention, and developers will see more implementations taking advantage of it in the domains of video games, CAD software, and other computer graphics applications.

Bibliography

[Paul 02] Brian Paul. “ARB_depth_texture and ARB_shadow.” OpenGL extension specifications, 2002.

[Brown 06] Pat Brown. “EXT_texture_array.” OpenGL extension specification, 2006.

[Brown and Lichtenbelt 08] Pat Brown and Barthold Lichtenbelt. “ARB_geometry” “_shader4.” OpenGL extension specification, 2008.

[Daniell 10] Piers Daniell. “ARB_timer_query.” OpenGL extension specification, 2010.

[Dimitrov 07] Rouslan Dimitrov. “Cascaded Shadow Maps.” NVIDIA Corporation, 2007.

[Haemel 09] Nick Haemel. “ARB_texture_cube_map_array.” OpenGL extension specification, 2009.

[Martin and Tan 04] Tobias Martin and Tiow-Seng Tan. “Antialiasing and Continuity with Trapezoidal Shadow Maps.” School of Computing, National University of Singapore, 2004.

[Stamminger and Drettakis 02] Marc Stamminger and George Drettakis. “Perspective Shadow Maps.” REVES-INRIA, 2002.

[Williams 78] Lance Williams. “Casting Curved Shadows on Curved Surfaces.” Computer Graphics Lab, Old Westbury, New York: New York Institute of Technology, 1978.

[Wimmer et al. 04] Michael Wimmer, Daniel Scherzer and Werner Purgathofer. “Light Space Perspective Shadow Maps.” Eurographics Symposium on Rendering, 2004.

[Zhang et al. 06] Fan Zhang, Hanqiu Sun, Leilei Xu and Lee Kit Lun. “Parallel-Split Shadow Maps for Large-scale Virtual Environments.” Department of Computer Science and Engineering, The Chinese University of Hong Kong, 2006.