

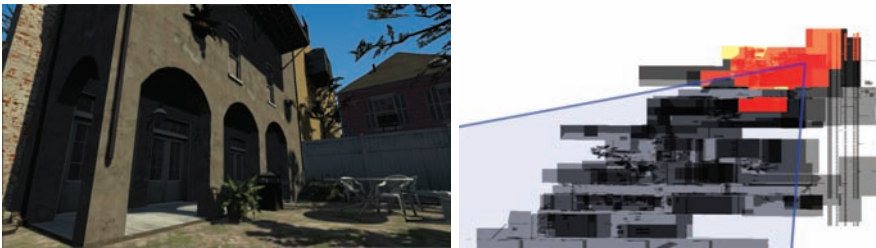
# Efficient Online Visibility for Shadow Maps

Oliver Mattausch, Jiri Bittner, Ari Silvennoinen,  
Daniel Scherzer, and Michael Wimmer

## 1.1 Introduction

Standard online occlusion culling is able to vastly improve the rasterization performance of walkthrough applications by identifying large parts of the scene as invisible from the camera and rendering only the visible geometry. However, it is of little use for the acceleration of shadow-map generation (i.e., rasterizing the scene from the light view [Williams 78]), so that typically a high percentage of the geometry will be visible when rendering shadow maps. For example, in outdoor scenes typical viewpoints are near the ground and therefore have significant occlusion, while light viewpoints are higher up and see most of the geometry.

Our algorithm remedies this situation by quickly detecting and culling the geometry that does not contribute to the shadow in the final image. Note that from the geometry visible from the light, only a small fraction will remain (for



**Figure 1.1.** The shadow-map geometry rendered for a particular *Left 4 Dead* view (left) and the corresponding light-view visualization (right), where the rendered shadow casters are shown in red.

example, the red parts in Figure 1.1). The main idea is to use camera-view visibility information to create a mask of potential shadow receivers in the light view, which restricts the areas where shadow casters have to be rendered. This algorithm makes shadow-map rendering efficient by providing the important property of *output sensitivity* (i.e., the complexity depends only on what is visible from the camera and not the size of the scene).

The method is easy to integrate into an existing rendering engine that already performs occlusion culling for rasterization. It is orthogonal to the particular occlusion-culling algorithm being used. We used the CHC++ algorithm [Matusch et al. 08] in our implementation, but in principle any state-of-the-art occlusion-culling algorithm can benefit from our method. Likewise, our method does not pose any restriction on the shadow-mapping algorithm being used. It was tested successfully with different algorithms like uniform shadows, LiSPSM, or cascaded shadow maps.

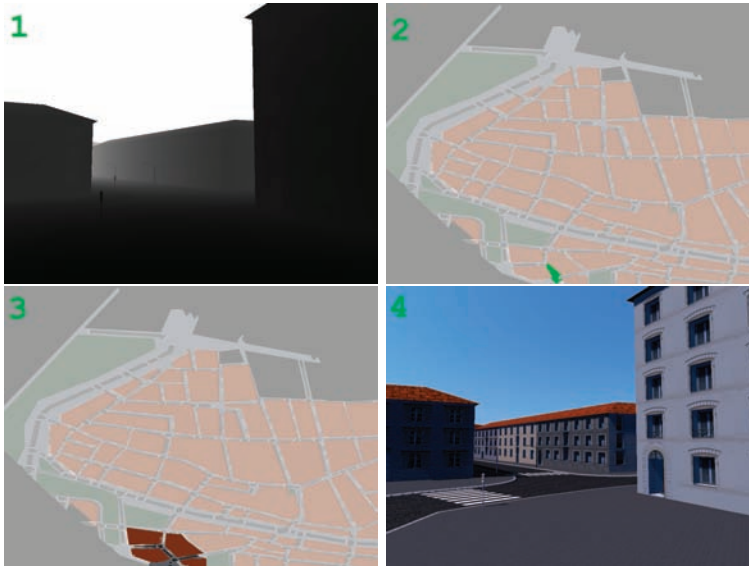
Our method is particularly useful for shadow mapping in large-scale outdoor scenes. In terms of overall render time (i.e., the whole pipeline until the final shaded image is rendered), the algorithm achieves a speedup of up to ten in real-world city scenes compared to the naïve use of occlusion culling. It also brings a significant speedup of up to two in real game scenes (e.g., a *Left 4 Dead* level as shown in Figure 1.1).

## 1.2 Algorithm Overview

The algorithm consists of the following four main steps, as also shown in Figure 1.2. Steps 1 and 4 constitute the standard approach for deferred shading (including shadow mapping); the main contributions of our algorithm are Steps 2 and 3.

*Step 1: Determine shadow receivers.* First we use occlusion culling to render the scene from the camera. This gives us the visible geometry, which corresponds to the potential shadow receiver geometry. Such an initial depth pass is a common practice in rendering engines. Our implementation uses a deferred shading approach, where other attributes like the geometry normals are stored in separate render targets for subsequent shading together with the depth buffer. We use a bounding volume hierarchy over the geometry as input to our occlusion-culling algorithm, and the potential shadow receivers correspond to the leaves of this hierarchy. Note that this step provides a conservative estimate of the visible geometry.

*Step 2: Create a mask of shadow receivers.* Next we render the potential receivers from the light view to generate a so-called receiver mask. During shadow-map rendering, shadow map updates are restricted to this mask. We can further tighten the receiver mask and restrict shadow-map rendering to only those shadow map texels that correspond to visible receiver



**Figure 1.2.** Steps of our algorithm. (1) Determine the potential shadow receivers and compute the depth buffer. (2) In light view, create a mask from the potential receivers containing only those fragments that contribute to a pixel in camera view (shown in green). (3) Determine visible subset of shadow casters using occlusion queries against the mask, and rasterize them into a shadow map. (4) Shade image using the depth buffer and the shadow map.

pixels. For this purpose, we make an additional lookup into the camera depth buffer when rendering into the shadow map (this step can be seen as a *reverse shadow test*).

*Step 3: Render shadow casters using the mask for culling.* After using the potential receivers for mask creation, we rasterize the rest of the scene geometry in order to complete the shadow map, i.e., the *potential shadow casters*. Our receiver mask allows us to quickly reject geometry that does not contribute to the final image, and significantly reduces the number of rendered shadow casters. This is done using hierarchical occlusion culling from the light view, issuing fast hardware occlusion queries to test if a node affects any masked pixels. Note that the speedup of our method is achieved in this step.

*Step 4: Compute shading.* Finally, we use the shadow map generated in the previous step to perform shadow mapping to find those pixels that are visible from the light source, and shade them accordingly. Note that this step is not altered by our approach.

## 1.3 Detailed Description

### 1.3.1 Determine Shadow Receivers

The first step of our algorithm consists of rendering the scene using an online occlusion-culling algorithm, which efficiently detects and culls the geometry that is not visible from the camera. Occlusion queries are issued on cheap proxy geometry (e.g., a bounding box) in order to determine the visibility of the contained complex geometry. Luckily a hardware implementation of occlusion queries is available [Cunniff et al. 07], which returns the number of visible fragments after a small latency.

Occlusion-culling algorithms usually employ front-to-back rendering, and make heavy use of temporal coherence for efficiency and minimizing the query overhead. Culling becomes particularly efficient if it employs a spatial hierarchy to quickly cull large groups of shadow casters, e.g., a bounding volume hierarchy in the case of the CHC++ algorithm. Occlusion culling is less effective for extreme bird's eye views, as much of the scene is visible from such views. Hence, for a typical light view (consider shadows cast from the sun) it is not feasible to use naïve occlusion culling.

Besides determining the depth buffer, such an occlusion pass also gives a good estimate of the geometry visible in the current frame and hence the potential shadow receiver geometry (as of course we are only interested in shadowing the visible geometry). We will use both depth buffer and visible geometry as input to our receiver-masking algorithm.

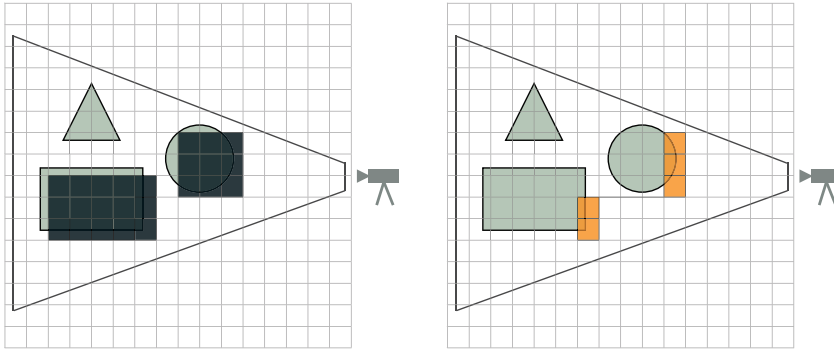
### 1.3.2 Create a Mask of Shadow Receivers

The general idea of our method is to create a mask of visible shadow receivers, i.e., those objects determined as visible in the first camera-rendering pass (Step 1). In order to create the mask, we rasterize the actual geometry of the visible shadow receivers into a render target. The creation of the mask happens in a separate pass before rendering the shadow map, but already generates parts of the shadow map itself as well, simplifying the subsequent shadow-map rendering pass. Note that this method alone would already create a valid receiver mask (shown in Figure 1.3 (left)).

However, we can do even better and create a tighter mask by considering the fact that not all of the potential receiver fragments are visible from the camera view. This is because visibility in the camera view is only determined on a per-object basis, while for some objects, only a few pixels are actually visible. The invisible pixels can be detected and discarded using a *reverse shadow test* (see Listing 1.1<sup>1</sup>), leaving only the fragments actually visible in the camera view (depicted in orange in Figure 1.3 (left)). The shader tests whether the current

---

<sup>1</sup>Note that all code segments are given in the Cg shading language.



**Figure 1.3.** The mask is created by rasterizing the potential receiver geometry from the light view (dark-blue fragments, left). With an additional depth buffer lookup, we can discard all potential receiver fragments which are not visible from the camera (leaving only the orange fragments in the mask, right).

fragment lies within the screen-space boundaries and passes the depth test with respect to the camera view. It outputs the test result to a color channel.

```
Fragment ReverseShadowTest(fragin IN,
                          uniform sampler2D depthBuffer)
{
    Fragment OUT;
    // post-projection screen-space position of current fragment
    float4 screenSpacePos = IN.screenSpacePos;
    screenSpacePos /= screenSpacePos.w;

    // the depth of this fragment from the camera
    float fragmentDepth = screenSpacePos.z;
    // the depth of the current pixel from the camera
    float4 depth = tex2D(depthBuffer, screenSpacePos.xy).x;

    // depth comparison: is current fragment visible?
    bool visible = fragmentDepth <= depth + 1e-4f;
    // is fragment inside screen boundaries?
    bool inside = all(saturate(screenSpacePos) == screenSpacePos);

    // if fragment contributes to shading, add to mask
    OUT.color.x = (visible && inside) ? 1.0f : .0f;
    return OUT;
}
```

**Listing 1.1.** The depth buffer of the camera is used to test the visibility of shadow-map fragments. This can be seen as a reverse shadow test that reverses the role of camera and light view.

```
Fragment OcclusionQuery(fragin IN, uniform sampler2D fragMask)
{
    Fragment OUT;
    // post-projection position in receiver mask
    float2 texCoord = IN.maskPos.xy / maskPos.w;
    // lookup corresponding fragment mask value
    float maskVal = tex2D(fragMask, texCoord).x;

    // discard if current fragment not masked
    if (maskVal < .5f) discard;
    return OUT;
}
```

**Listing 1.2.** Fragment shader for the receiver-mask lookup of an occlusion query.

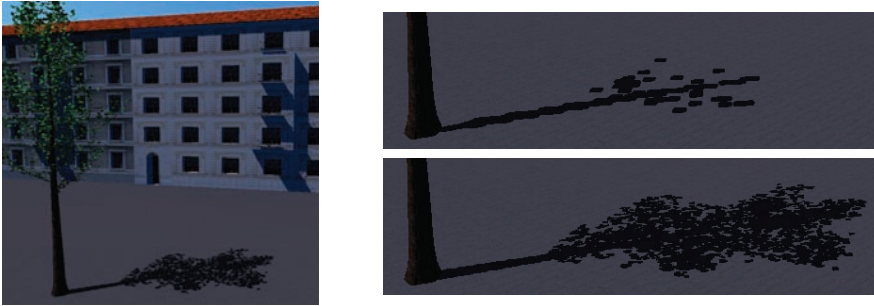
### 1.3.3 Render Shadow Casters Using the Mask for Culling

In the shadow-map rendering pass, we rasterize the rest of the geometry as potential shadow casters. The mask is used in this pass to cull those potential shadow casters that do not contribute to the visible shadows. When issuing a hardware occlusion query, we use the fragment mask as a lookup texture and discard all fragments lying outside the mask (as shown in Listing 1.2).

Note that the lookup into the mask creates a minor overhead as compared to a standard occlusion query, which could be avoided if it were possible to directly write to the stencil buffer within the fragment shader in order to create a stencil mask. However, a suitable OpenGL extension (`GL_ARB_shader_stencil_export`) is already available and hopefully this feature will be better supported in the future.

## 1.4 Optimization: Shadow-Map Focusing

For very large scenes, the shadow-map resolution can become critical for maintaining a reasonable shadow quality. Shadow-map warping algorithms like LiSPSM [Wimmer et al. 04] and shadow-map partitioning algorithms like cascaded shadow maps [Engel 06, Zhang et al. 06] provide a better distribution of shadow map texels between near and far geometry, and in addition focus the shadow map in order not to waste shadow-map space on geometry not within the view frustum. However, if the distance to the far plane is very large compared to the near plane, the shadow quality can still become unacceptable due to lack of resolution. While cascaded shadow maps improve shadow quality significantly by slicing the view frustum and computing a shadow map for each slice individually, the problem still remains that a lot of shadow resolution can be wasted for areas that cannot be seen from the camera [Lauritzen et al. 11].




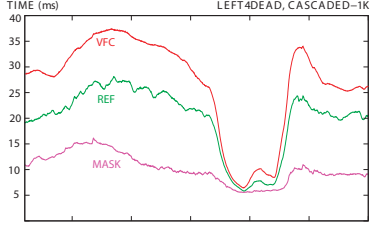

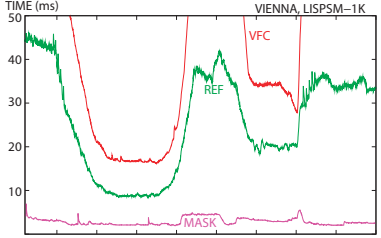
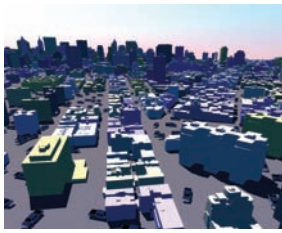
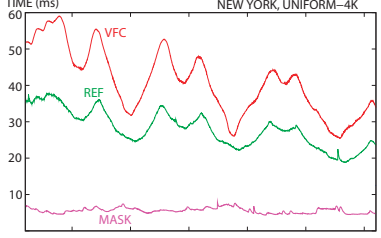
**Figure 1.4.** The shadow quality of an unfocused shadow map (top) can be greatly improved by focusing the shadow map on the visible geometry (bottom).

Since we collect the geometry visible from the camera in the first step of our algorithm anyway, we get all the information necessary for effectively focusing the shadow map for free. In particular, instead of the usual approach of intersecting the view frustum with the scene boundaries and focusing the shadow map on the resulting polytope, we intersect the view frustum with the union of the bounding boxes of all visible objects.

This algorithm can significantly improve the shadow quality in cases where there is sufficient occlusion (Figure 1.4). Focusing can also improve shadow-map rendering times due to more accurate shadow-frustum culling. However, the increase in the effective shadow-map resolution and the more accurate shadow-frustum culling gained from focusing are both temporally unstable. This temporal instability is potentially manifested by flickering shadow artifacts due to varying shadow-map resolution and incoherent shadow-map rendering times. The former is hard to control, although one could use temporal coherence and careful level design to mitigate the effect. The latter is an example of a non-output-sensitive process and at worst, a single visibility event can make the focused culling dependent on the whole scene. In contrast to simple shadow-frustum culling, our receiver-masking technique is output sensitive and hence the shadow-map rendering times are always predictable. Focusing as a stand-alone algorithm is not a reliable acceleration technique and should always be used in combination with receiver-masking.

## 1.5 Results

We implemented the presented algorithm in OpenGL and C++ and evaluated it using a GeForce 480 GTX GPU and a single Intel Core i7 CPU 920 with 2.67 GHz. For the camera-view render pass, we used an  $800 \times 600$  32-bit RGBA render target (to store color and depth) and a 16-bit RGB render target (to store the normals). For the light-view render pass, we used a 32-bit depth texture.

<p><i>Left 4 Dead</i></p> 	<p>1,442,775 vertices</p> 
<p>Vienna</p> 	<p>12,685,693 vertices</p> 
<p>Manhattan</p> 	<p>104,960,856 vertices</p> 

**Table 1.1.** Test scenes and example frame times with varying shadow parameters.

We compared the proposed receiver-mask algorithm (MASK) to view frustum culling (VFC) and a reference method (REF), which uses our unmodified occlusion culling algorithm for both light and camera views, and we plotted the total frame-rendering times for different test scenes in Table 1.1.

As can be observed from the timings, our algorithm works particularly well in the two city environments: a model of Manhattan and the town of Vienna, which were populated with various scene objects. This is because they are large, containing open scenes with a high cost for shadow mapping. We get a lower speedup of 1.4–2 in the *Left 4 Dead* game scene (rendered at 720p resolution and using four cascaded shadow maps with 1 K resolution each), partially explained by the overall lower geometric complexity. However, keep in mind that a two times speedup in a scene otherwise highly optimized for fast rendering is very worthwhile. The dependence of the algorithm performance on the shadow-map



SM type	LiSPSM			UNIFORM		
Shadow size	1K	2K	4K	1K	2K	4K
Scene	Vienna					
Reference	21.6	22.3	22.4	28.7	28.7	29.2
Our method	2.9	3.5	6.1	2.9	3.4	5.9
Scene	Manhattan					
Reference	36.6	35.9	35.1	44.2	43.9	41.8
Our method	4.5	5.4	9.0	4.5	5.3	8.6

**Table 1.2.** Average frame times for two of the tested scenes (in ms).

resolution (1 K–4 K) and the used shadow-mapping algorithm (LiSPSM [Wimmer et al. 04] and uniform shadow maps) can be seen in Table 1.2.

## 1.6 Conclusion

We presented an algorithm for fast, output-sensitive shadow mapping in complex scenes. The proposed method generalizes trivially over a wide class of occlusion-culling algorithms as long as they are compatible with receiver masking, a property that holds for all rasterization-based algorithms. The basic principle is easy to integrate into existing game engines, especially if the engine is already using occlusion culling for the main view. We demonstrated the benefits of the algorithm using a reference implementation in the context of large directional light sources and note that the small overhead of generating the receiver mask is easily compensated by the performance gains during shadow-map generation.

## 1.7 Acknowledgments

We would like to thank Jiri Dusek for an early implementation of shadow map culling ideas; Jason Mitchell for the *Left 4 Dead 2* model; Stephen Hill and Petri Häkkinen for feedback. This work has been supported by the Austrian Science Fund (FWF) contract no. P21130-N13; the Ministry of Education, Youth, and Sports of the Czech Republic under research program LC-06008 (Center for Computer Graphics); and the Grant Agency of the Czech Republic under research program P202/11/1883.

## Bibliography

[Cunniff et al. 07] Ross Cunniff, Matt Craighead, Daniel Ginsburg, Kevin Lefebvre, Bill Licea-Kane, and Nick Triantos. “ARB\_occlusion\_query.” *OpenGL Registry*. Available online ([http://www.opengl.org/registry/specs/ARB/occlusion\\_query.txt](http://www.opengl.org/registry/specs/ARB/occlusion_query.txt)).

- [Engel 06] Wolfgang Engel. “Cascaded Shadow Maps.” In *ShaderX<sup>5</sup>: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 197–206. Hingham, MA: Charles River Media, 2006.
- [Lauritzen et al. 11] Andrew Lauritzen, Marco Salvi, and Aaron Lefohn. “Sample Distribution Shadow Maps.” In *Symposium on Interactive 3D Graphics and Games, I3D ’11*, pp. 97–102. New York: ACM Press, 2011.
- [Mattausch et al. 08] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. “CHC++: Coherent Hierarchical Culling Revisited.” *Computer Graphics Forum (Proceedings of Eurographics 2008)* 27:2 (2008), 221–230.
- [Williams 78] Lance Williams. “Casting Curved Shadows on Curved Surfaces.” *Computer Graphics (SIGGRAPH ’78 Proceedings)* 12:3 (1978), 270–274.
- [Wimmer et al. 04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. “Light Space Perspective Shadow Maps.” In *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, pp. 143–151. Aire-la-Ville, Switzerland: Eurographics Association, 2004.
- [Zhang et al. 06] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. “Parallel-Split Shadow Maps for Large-Scale Virtual Environments.” In *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and its Applications*, pp. 311–318. New York: ACM Press, 2006.