Contents

ShadowProxies

Jochem van der Spek

1.1 Introduction

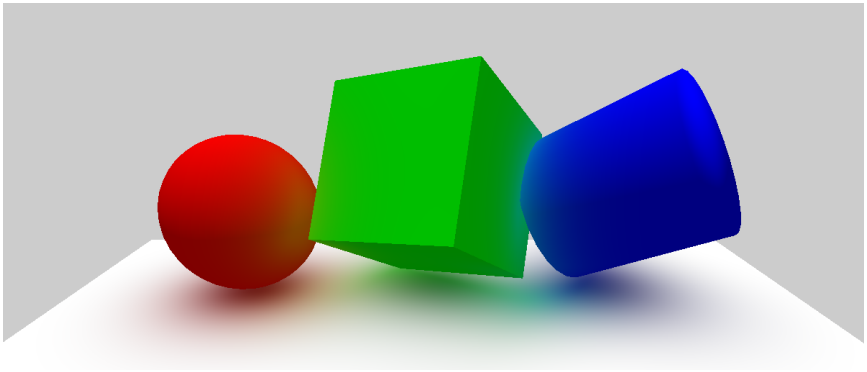

Figure 1.1. Still from the demo movie.

For real-time rendering of the virtual painting machines that I regularly show in exhibitions, I needed a shadowing technique capable of rendering soft shadows without any rendering artifacts such as banding or edge jitter no matter how close the the camera came to the penumbra. I call such shadows *infinitely soft*. In addition, I wanted a method to render color bleeding so that the color and shadow of one object could reflect onto others. Searching through the existing real-time soft shadow techniques [Hasenfratz et al. 03], I found that most were either too complex to implement in the relatively short time available, or they were simply not accurate enough, especially when it came to getting the camera infinitely close to the penumbra. Most techniques for rendering the color bleeding required setting up of some form of real-time radiosity rendering that would be prohibitively complex and expensive in terms of computing power.

The solution came in the form of a reversed argument: if we can not globally model the way the light influences the objects, why not locally model the way the objects influence the light? Given that in a diffusely lit environment shadows and reflections have limited spatial influence, some sort of *halo* around the model could function as a *light subtraction* volume

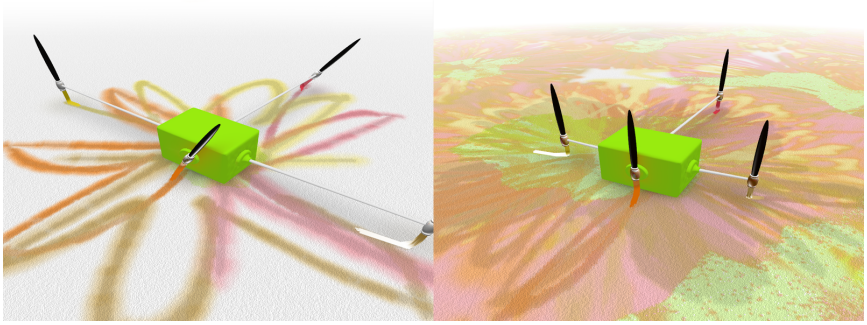

Figure 1.2. Stills from a virtual painting machine.

- see figures 1.3 and 1.5. In order to model directional lighting the shadow volume could be expanded in the direction away from the light source and contracted to zero in the opposite direction. The color bleeding volume could be expanded toward the light in the same manner. We call these volumes *shadow proxies*[1] as they serve as a stand-in for the actual geometry. The volume of a proxy covers the maximum spatial extent of the shadow and color bleeding of the geometry that the proxy represents. The technique is therefore limited to finite shadow volumes, and is mostly useful for diffusely lit environments. This is similar to the *Ambient Occlusion Fields* technique by [Kontkanen and Laine 05] although with *ShadowProxies*, modeling and modulating the shadow volumes is done on the fly rather than precalculating the light accessibility of the geometry into a cubemap.

1.2 Anatomy of a shadow proxy

In the current implementation, each shadow proxy can only represent a simple geometrical shape like a sphere, box or cylinder allowing quick proximity calculations in the fragment shader that eventually renders the shadows.

An implementation that used *super-ellipsoids* [Barr 81] was also attempted. Even though the surface lookup was fast enough to be used in real-time the method presented problems with the requirement of perfectly smooth penumbra. This was because the search for the boundary of an implicit surface results typically in an approximation and zooming in onto

[1]The ShadowProxies technique is implemented in the OpenGL- based cross-platform scenegraph library called RenderTools, available under the GNU Public License (GPL) which ensures open source distribution. RenderTools is available on Sourceforge at http://sourceforge.net/projects/rendertools and through the OpenGL Insights website, http://www.openglinsights.com

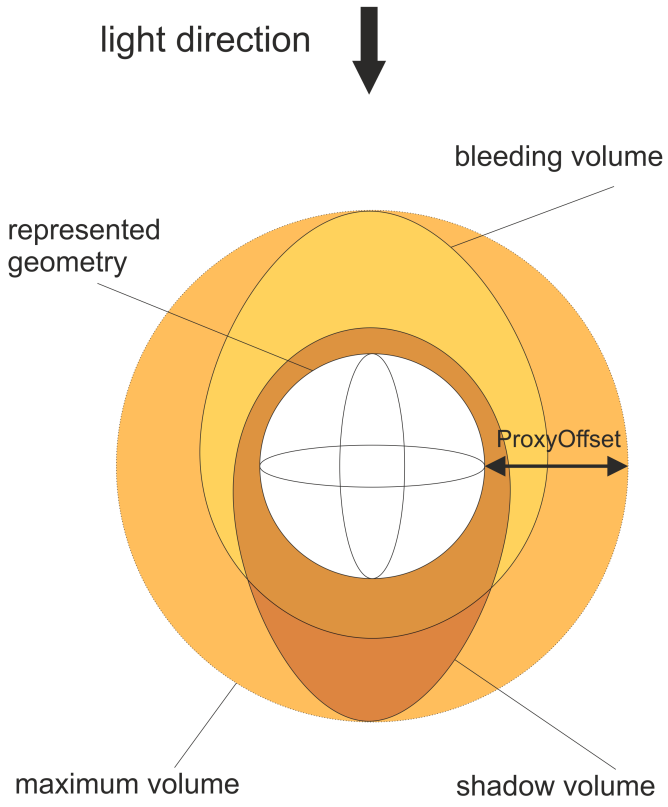

Figure 1.3. The volume regions of a shadow proxy.

the penumbra area means zooming in onto the error of the approximation which quickly became visible in the form of banding. An adaptive algorithm where accuracy was dependent on camera proximity was not attempted. Furthermore, a version was implemented where the sharp edges of the shapes were replaced by arcs. A parameter allowed the dynamic modification of the radius of the arc, allowing for quite a host of different shapes. In practice this method turned out to be ineffective for the relative high computational cost.

Surprisingly, the extremely simple, almost trivial surface-determination algorithm now implemented outperforms both previous approaches in terms of efficiency, simplicity and quality.

Aside from shape information, position, orientation and size, each shadow

```
vec3 closestPoint( int shape, mat4 proxy, vec3 fragment ){

  vec3 local = ( inverse( proxy ) * fragment ).xyz;
  vec3 localSgn = sign( local );
  vec3 localAbs = abs( local );

  if( shape == SPHERE ){
    localAbs = normalize( localAbs );
  }
  else if( shape == BOX ){
    localAbs = min( localAbs, 1.0 );
  }
  else if( shape == CYLINDER ){
    if( length( localAbs.xy ) > 1.0 ){
      localAbs.xy = normalize( localAbs.xy );
    }
    localAbs.z = min( localAbs.z, 1.0 );
  }
  return( proxy * ( localSgn * localAbs ) );
}
```

Listing 1.1. The nearest point on the surface of a shadow proxy from the worldposition of a fragment.

proxy holds information about the material it represents such as the diffuse and reflective colors of the geometry. Instead of specifying the exact extent of each proxy volume, a fixed offset distance is added to the size of the geometry as the maximum extent of the volume that the proxy represents. This single-valued *ProxyOffset* parameter is represented in the shader as a uniform float. Other global parameters include the falloff of the shadows as the exponent to the attenuation function, a cutoff value to allow for an offset between the surface and the start of the shadow falloff, the amount of shadow contribution, the amount of color bleeding, etc. The complete list can be found in the ShadowProxyTest example in the RenderTools library.

1.3 Setting up the pipeline

The flow of information 'from the scene to the screen' is as follows:

1. collect the objects that cast shadows or reflect their color and collect their shadow proxy objects.

2. clip the proxies against the viewing clip planes

3. pass the information about the shadow proxies such as size, color, position, etc. that are in view to the uniforms in the *ShadowProxy-*

enabled fragment shader

4. render the geometry that receives shadows

Because a scene could have hundreds of different shadow proxies inside the view frustum, the last step in the process became a bottleneck as each fragment needed to be tested against each shadow proxy. In order to reduce the number of pairwise comparisons, a spatial subdivision scheme was needed so that each fragment was only tested against proxies that were nearby. This was done on by subdividing the viewport into an orthogonal grid, and then testing the overlap of the bounding box of each shadow proxy projected onto the near plane of the frustum with each cell in the grid. This overlap calculation is quite straight forward: the corners of the non-axis-aligned bounding box of each proxy are projected onto the near plane of the frustum, and then the minima and maxima are calculated in terms of grid-indices. The index of that proxy is then added to the rectangle of cells within those minima and maxima. The proxy index is simply the index of the proxy in the list of proxies that are in view. Each grid's cell should be able to hold several shadow proxies, but not very many. In fact in my experience, situations with more than three proxies overlapping the same cell were rare.

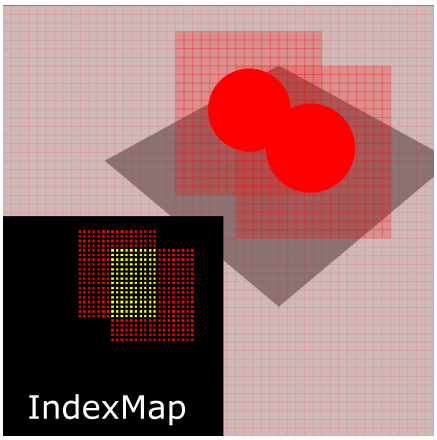

Figure 1.4. Index storage - a red pixel in the IndexMap means that the first index of that cell is set with the index of a proxy ranging from 0-255, yellow means the first two indices are set.

The information of each grid's cell is encoded in a texture called the IndexMap, using a fixed number of pixels to store the indices of the shadow proxies. For portability, we chose to use the GLubyte data type, limiting

the number of unique proxy indices to 255 as each color component of a texel holds one single proxy index, but this limitation can be overcome by using less portable floating point textures or by using more than one component for an index. Thus, in order to encode a grid of 64 x 64 cells with each cell capable of holding 16 indices, an RGBA texture of 128x128 pixels suffices. The information for each proxy neatly fits into a 4x4 floating point matrix by using one float for the type, three for size, four for position and orientation and one float each for the colors, packing the RGBA values into a single float. I wrote a simple packing function for this, only to find out later that GLSL 4 introduces some handy pack/unpack functions. Using this encoding scheme, the proxies can be sent to the shader as an array of uniform mat4. The array is ordered as indexed by the clipping algorithm, so that each index in the IndexMap corresponds directly to the index in the array.

1.4 The ShadowProxy-enabled fragment shader

The algorithm for determining whether a fragment needs shadowing or additional coloring because of color bleeding is summarized as follows:

1. calculate the index of the grid's cell that contains the current fragment
2. fetch the indices of the proxies that that grid's cell overlaps
3. for each shadow proxy index, retrieve the corresponding mat4 uniform that contains all the positional, type and color data and construct a 4x4 transformation matrix for that proxy.
4. using the proxies' transformation matrices, test if the fragment overlaps any of the proxies' influence volume.

First we obtain a list of shadow proxies that are potentially influencing the color of the fragment. For each proxy in that list, we test if the fragment is contained within its influence volume. This containment test is performed by comparing the distance from the fragment to the closest point on the surface of the shadow proxy. If the distance is smaller than the ProxyOffset parameter, the fragment is deemed inside the volume. To calculate this closest point, we recognize that all three shapes under consideration are symmetrical in the three planes xy, xz, and yz. This allows us to take the absolute value of the local fragment coordinate relative to the shadow proxy's reference frame, and clamp that vector to the positive boundaries for each axis so that we consider just the positive quadrant of

```
// find the cell index for this fragment
vec2 index = floor( ( gl_FragCoord.xy / viewport ) * proxyGridSize );
// find out the uv- coordinate of the center of the pixel
vec2 uv = index * vec2( cellSizeX, cellSizeY ) + vec2( 0.5, 0.5 );

// loop over each pixel of the cell that this fragment is in
for( int j = 0; j < cellSizeX; j++ ){
  for( int i = 0; i < cellSizeY; i++ ){
    // get 4 proxy indices from this pixel (scaled to 255)
    vec4 proxies = texture2D( IndexMap, ( ( uv + vec2( i, j ) ) / ↩
        IndexMapSize2 ) );
    for( int k = 0; k < 4; k++ ){
      // if this index == 0, the algorithm ends
      if( proxies[ k ] == 0.0 ){
        return( returnStruct );
      }
      // retrieve the index of the proxy from the texel
      int currentIndex = int( proxies[ k ] * 255.0 ) - 1;
      if( currentIndex == ( proxyIndex - 1 ) ){
        // ignore self-shadows
        continue;
      }
      // we have a valid index, so find the associated parameters
      mat4 params = proxyParams[ currentIndex ];

      //... calculate and accumulate shadow and bleed values
    }
  }
}
```

Listing 1.2. GLSL code to retrieve the proxy index and data

the shape. Finally we obtain the true point on the surface by multiplying the result with the original sign of the local fragment coordinate and the shadow proxy's reference frame.

```
// shadow is 1.0 at the geometry surface, and goes to 0.0 at the edge ↩
    of the proxy surface
float shadow = 1.0 - clamp( min( length( vertexToSurface ) / ↩
    proxyOffset, 1.0 ), 0.0, 1.0 );
// shadowFactor and shadowFalloffExponent are uniform float paramaters
shadow *= pow( shadowFactor, shadowFalloffExponent );
```

Listing 1.3. calculating the shadow and/or reflection factor.

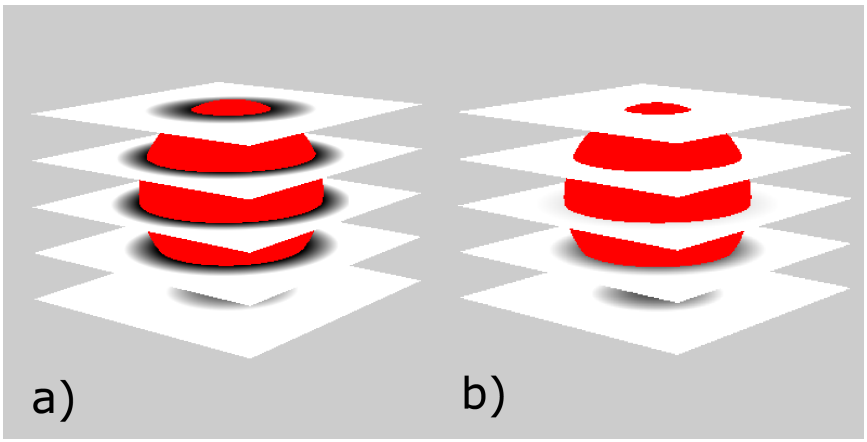

Figure 1.5. The shadow volume of a proxy. a) shows the volume without modulation, b) the volume multiplied by the dot product of the surface normal and light direction.

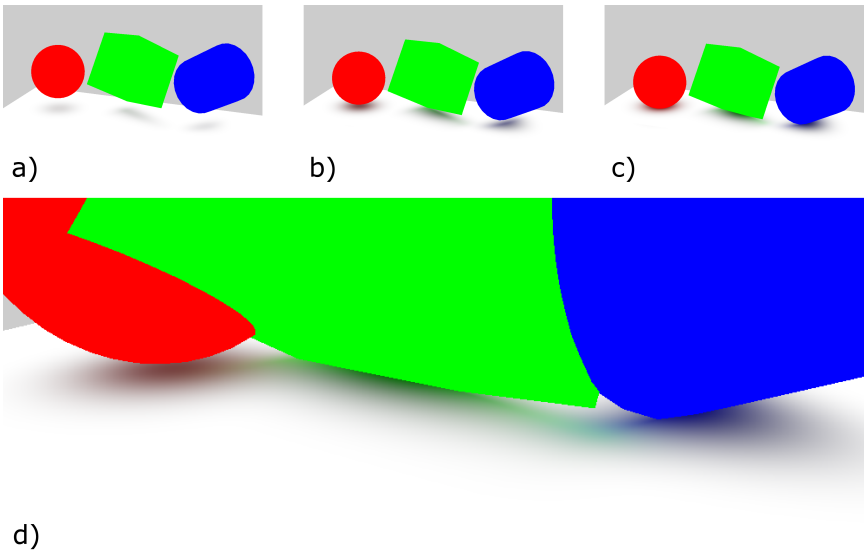

Figure 1.6. Shadow of the different shapes. Notice how the shadow is sharper where the distance to the geometrical surface is smaller.

1.5 Modulating the shadow volume

When we want to model a directional light, the shadow and bleeding volumes need to be modulated to an egg-shaped volume that snugly fits the

geometry. This is done by multiplying the dot product of the normal at the proxy surface with the normalized vector from that point toward the light. Exactly the same calculation but with reversed normal gives the volume of the color bleeding in the opposite direction. The direction in which the shadow or color bleeding is cast is taken to be the normalized directional vector from the fragment world coordinate to the closest point on the surface of the maximum extended volume.

```
// modulate the shadow to an egg-shaped volume around the geometry
shadow *= clamp( dot( lightDirection, surface.normal ), ↩
    shadowCutoffValue, 1.0 );
```

Listing 1.4. Modulating the shadow volume. The surface normal is the normal at the closest point on the proxy surface.

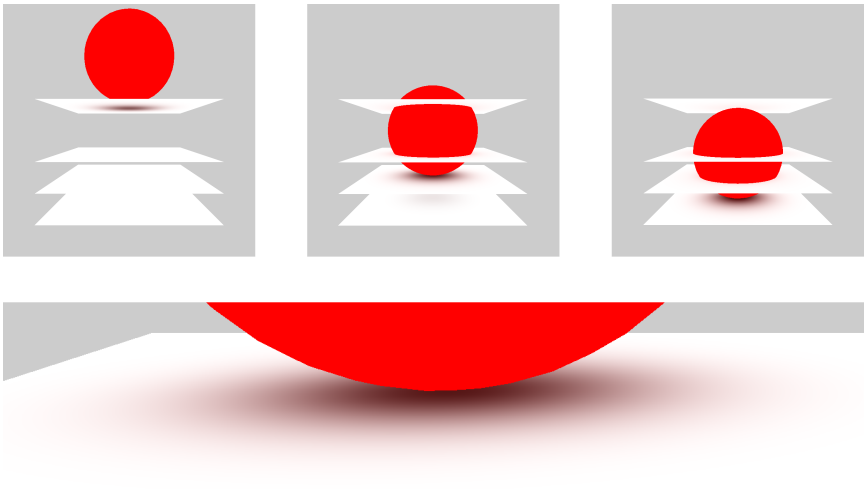

Figure 1.7. The combined effect of shadow and color bleeding.

1.6 Performance

Performance is mostly influenced by the *ProxyOffset* parameter that determines the size of the shadow volumes. When the parameter is small

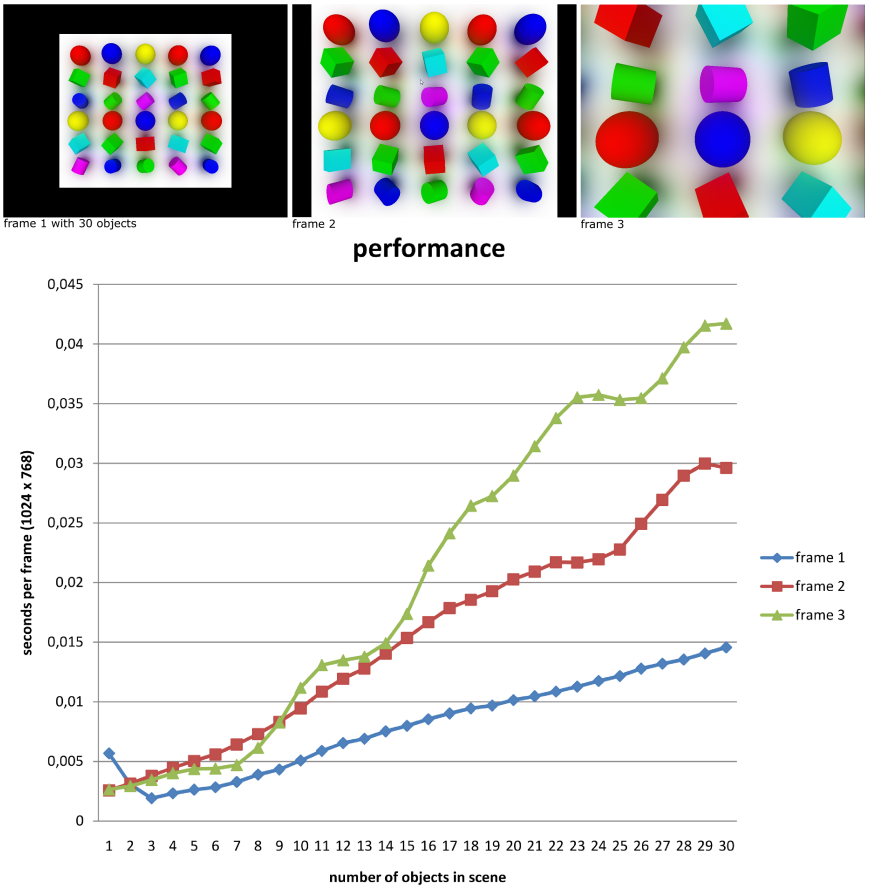

Figure 1.8. Duration of each frame was measured using glQueryCounter at the beginning and end of the render calls. The graph shows the performance of rendering a single frame at three distances from the camera with increasing numbers of objects in the scene. The test was run on a MacBook Pro 2,4GHz with an NVidia GeForce GT330M.

compared to the geometry, volume overlaps occur less often and shader performance scales with the number of overlaps. However, due to the limited size of the volumes, scaling is linear as can be seen in Figure 1.8.

1.7 Conclusion and future work

The *ShadowProxies* technique was developed for a specific purpose and the quality of the result is sufficient for the project at hand, but the technique is admittedly limited. However, the technique has proven to be very useful in small games and other projects like the painting machines, and can be particularly effective in situations with relatively simple geometrical shapes and scenery. Because the technique is so easy to implement and provides an original rendering style, we believe many games that otherwise can not afford soft shadows much less color bleeding, could benefit a great deal by using it.

A simple but useful extension to the algorithm can introduce multiple colored light sources with similarly colored overlapping shadows. This can be achieved by iterating over the available light sources when doing the shadow calculations. Another feature that is almost trivial to add is light emission by the proxy or, a bit less trivial, modeling caustics like those caused by a semi-transparent marble. A more sophisticated light transport model can be envisioned where the orientation of the receiving surface and the direction of the incoming shadow or color reflection plays a much greater role than is currently the case. Finally, representation of the geometrical shapes could be extended by implementing some form of Constructive Solid Geometry, or could be replaced altogether by reconstructing the represented geometry from its spherical harmonics representation [Mousa et al. 07].

Bibliography

[Barr 81] A. Barr. "Superquadrics and angle-preserving transformations." *IEEE Computer Graphics and Applications* 1:1 (1981), 11–23. http://vis.cs.brown.edu/results/bibtex/Barr-1981-SAP.bib(bibtex: Barr-1981-SAP).

[Hasenfratz et al. 03] J.-M. Hasenfratz, M. Lapierre, N. Holzschuch, and F.X. Sillion. "A Survey of Real-time Soft Shadows Algorithms.", 2003.

[Kontkanen and Laine 05] Janne Kontkanen and Samuli Laine. "Ambient Occlusion Fields." In *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*, pp. 41–48. ACM Press, 2005.

[Mousa et al. 07] Mohamed Mousa, Raphalle Chaine, Samir Akkouche, and Eric Galin. "Efficient spherical harmonics representation of 3D

objects." In *15 th Pacific Graphics*, pp. 248–257, 2007. Available online (http://liris.cnrs.fr/publis/?id=2972).