Shadow Mapping for Hemispherical and Omnidirectional Light Sources

Stefan Brabec Thomas Annen Hans-Peter Seidel
Computer Graphics Group
Max-Planck-Institut für Infomatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
{brabec, tannen, hpseidel}@ mpi-sb.mpg.de

Abstract

In this paper we present a shadow mapping technique for hemispherical and omnidirectional light sources using dual-paraboloid mapping. In contrast to the traditional perspective projection this parameterization has the benefit that only a minimal number of rendering passes is needed during generation of the shadow maps, making the method suitable for dynamic environments and real time applications. By utilizing programmable features available on state-of-the-art graphics cards we show how the algorithm can be efficiently mapped to hardware.

Keywords: Shadow Algorithms, Shadow Mapping, Dual-Paraboloid Mapping, Graphics Hardware.

1 Introduction and Background

One of the most challenging effect in the field of real time rendering is the accurate and fast calculation of shadows. For most applications, e.g. in computer games, shadows are still problematic due to their *global* nature. In order to decide if a given surface pixel can be seen by a light source one has to check a large number of potential occluders that may block the path from light to surface.

Although there are a large number of shadow algorithms, only a small subset of algorithms are really appropriate for real time rendering. Nearly all techniques in this subset can further be categorized into two main *branches* of algorithms: Those which are based on Crow's *shadow volumes* approach [3], and those that are variants of Williams' *shadow mapping* technique [13].

Both categories have their pros and cons: The shadow volume approach is capable of producing very accurate shadows due to the object space calculation. For each (possible) blocker object, a semi-infinite shadow volume is constructed which defines the volume in which all pixels inside cannot be seen by the light source. Although the actual shadow volume *in-out* test can be accelerated by using graphics hardware, the CPU still has to perform the time-consuming task of constructing the actual shadow volumes.

The shadow mapping technique on the other hand does not need any CPU time at all since all steps can by directly implemented on the graphics hardware itself (which will be explained in more detail later). Shadow mapping works on a sampled depth map of the scene, taken from the view of the light source. The actual shadow test is then a very simple operation that compares a surface pixel's depth value with the appropriate entry in the depth map. In contrast to the shadow volume approach, shadow mapping does not directly depend on the scene's complexity. All objects that can be rendered can also be used for shadow mapping, which makes this algorithm suitable for a wide range of applications. The disadvantage is of course the quality of shadows, which is directly related to the sampling of the scene. As pointed out by Reeves et. al. [9], sampling artifacts occur during the generation of the shadow map as well when performing the shadow test. To eliminate these artifacts [9] proposed a filtering scheme that softens shadow boundaries.

One main argument that often leads people to prefer shadow volumes over shadow mapping is the limited *field-of-view* when generating a suitable shadow map for a given point light source. In cases where a point light source radiates over the complete hemisphere or even omnidirectional, traditional shadow mapping (using only a single depth map) fails.

The only solution to this problem so far is to use more than one shadow map. In the worst case, up to six shadow maps must be used to cover the complete environment [4]. Although these maps can be represented as a cube map (resulting in a single texture lookup when performing the shadow test), this approach still requires up to six rendering passes when generating the shadow maps, making it unsuitable for interactive or real time applications.

In this paper we present a method to perform shadow mapping for hemispherical and omnidirectional point light sources using only one (hemispherical) or two (omnidirectional) rendering passes for the generation of the shadow map and one final rendering pass to perform the shadow test. Similar to the traditional shadow mapping technique, all steps of the algorithm can be implemented using graphics hardware.

2 Shadow Mapping

As described by [13] the traditional shadow mapping algorithm consists of two steps. In the first pass, a shadow map is generated by rendering the scene as seen by the light source and depth values of the front most pixels are stored away in this map. In the final step the scene is rendered once again, this time from the camera's point of view. During this pass the coordinates of visible surface points are transformed to the light source coordinate system. To check whether a given pixel can be seen by the light source one has to compare the depth value of the transformed pixel with the depth value stored in the corresponding shadow map entry. If the shadow map has a depth value which is less than the actual one, the pixel is blocked by an occluder in between. Otherwise the pixel is lit by the light source.

A hardware implementation for this algorithm was proposed by [10]. Using automatic texture coordinate generation it is possible to have homogeneous texture coordinates (s, t, r, q) that are derived from the eye-space coordinate system. In conjunction with a texture matrix (which is a 4×4 matrix applied to these texture coordinate) one can implement the necessary transformation to the light source coordinate system including the perspective projection.

In order to perform the shadow test, a dedicated texture mapping mode is needed that compares the entry at $(s/q, t/q)$ in the shadow map with the computed depth value (r/q). The result of this operation is coded as color $(0, 0, 0, 0)$ or color $(1, 1, 1, 1)$. In addition to this shadow test mode, the graphics hardware has also to support a way of storing depth values as textures.

In OpenGL [11] all of these capabilities are supported by two extensions [8]: The *depth_texture* extensions introduces new internal texture formats for 16, 24, or 32 bit depth values. The *shadow* extensions defines two operations ($\leq$ and $\geq$) that compare (r/q) with the value stored at $(s/q, t/q)$.

For a long time these extensions were only supported on high-end graphics workstations. Recently also consumer-class graphics cards (e.g. NVIDIA GeForce3) provide this functionality.

3 Shadow Mapping for Hemispherical and Omnidirectional Light Sources

As stated in the previous sections shadows generated with the traditional shadow mapping algorithm are limited to the view frustum used when generating the shadow map, as depicted in Figure 1(a).

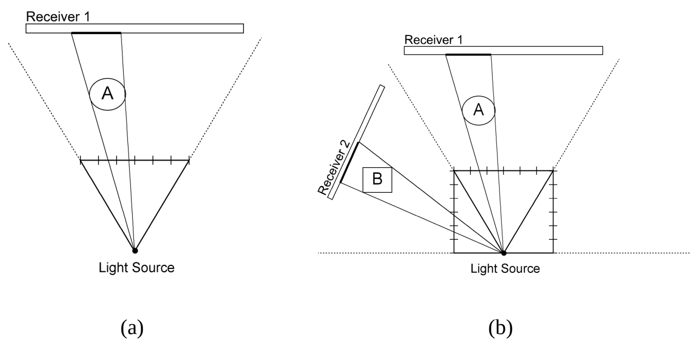

Fig. 1. (a): Traditional shadow mapping frustum. (b): Multiple shadow maps for field-of-view of 180° (hemispherical).

Since all relevant occluder geometry is inside the frustum shadows are generated as expected. This setup is useful if the light source has a spotlight characteristic (only objects inside the spotlight cone are illuminated) but fails if the light source is hemispherical or omnidirectional (*field-of-view* $>= 180°$). The trivial solution to handle such cases would be to use a number of shadow maps that together cover the whole environment seen by the light source.

Figure 1(b) shows such a setup for a light source with a viewing angle of $180°$(hemispherical). In order to cover Object A as well as Object B the environment needs to be subdivided in a cube like manner where a separate shadow map is used for each side of the cube[1]. While this setup can be efficiently rendered if the graphics hardware supports cube map textures (as explained in [4]) the generation of the (up to six) shadow maps is still too expensive.

The solution to this problem is quite simple: In order to minimize the cost of generating shadow maps we have to find a way of computing a shadow map that covers the whole field-of-view of the given light source. Since the sampling rate should be somehow constant to avoid changes in shadow quality we have to choose a parameterization that fulfills this criterion and that is also easy to compute (hardware-accelerated rendering).

[1]Since only one hemisphere will be lit, only five sides of the cube are needed.

We can find such parameterizations in the field of environment mapping, which approximates global illumination by pre-computing a so called *environment map* [1] which is later used to determine the incoming light for a given direction (reflection vector). Although there exist a vast number of 3D-to-2D mappings (used to *flatten* a panoramic environment into a 2D texture map), only a small subset are really appropriate for hardware accelerated rendering:

Spherical Mapping. For a long time this was the *standard* parameterization used to represent environment maps [5]. This parameterization can be simple explained by imagining a perfectly reflecting mirror ball centered around the object of interest. Although this parameterization only has one point of singularity, the sampling rate since pixels get extremely distorted towards the perimeter of the flattened sphere.

Blinn/Newell Mapping. A different parameterization of the sphere was proposed by [1]. Here 2D coordinates (u, v) are computed using a longitude-latitude mapping of the direction vector. Although this approach does not introduce as much distortion as the previous sphere mapping technique, it is not commonly used due to the expensive longitude-latitude mapping (which involves computing $arctan$ and $arcsin$).

Cube Mapping. As already mentioned in previous section, the cube map parameterization [12] is very popular since it does not require any re-warping to obtain images for the cube faces. Recent graphics hardware (e.g. NVIDIA, ATI) directly support texture fetching using cube maps. Again the main disadvantage are the number of rendering passes during the generation phase, making this method nearly unusable for dynamic environments.

(Dual-) Paraboloid Mapping. Another parameterization was proposed by [6]. Here the analogy is the image obtained by an orthographic camera viewing a perfectly reflecting paraboloid. When compared to sphere or cube mapping, this parameterization introduces less artifacts because the sampling rate only varies by a factor of 4 over the complete hemisphere. For 360°views, two parabolic maps can be attached back to back.

When comparing these different environment mapping techniques, it becomes obvious that the parabolic parameterization would be among the best choices for hemispherical and omnidirectional shadow maps due to the following properties:

- Sampling ratio varies only by a factor of 4.
- One map covers one hemisphere (number of generation passes ?)
- Easy to implement (described later)

In the next sections we will describe the concept of paraboloid mapping in detail, first using the mathematical interpretation and later with respect to graphics hardware (implementation).

3.1 Theory of Paraboloid Mapping

As described by Heidrich et al. [6], the image seen by an orthographic camera facing a reflecting paraboloid

$$f(x, y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2) \qquad , x^2 + y^2 \leq 1, \tag{1}$$

contains all information about the hemisphere centered at $(0, 0, 0)$ and oriented towards the camera $(0, 0, 1)$. This function is plotted in Figure 2(a). Since the paraboloid acts like a lens, all reflected rays originate from the focal point $(0, 0, 0)$ of the paraboloid.

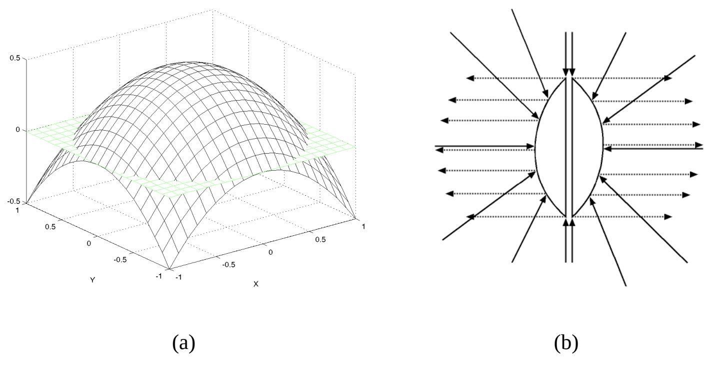

Fig. 2. (a): Paraboloid $f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2)$. (b): Using two paraboloids to capture the complete environment.

In order to capture the complete environment (360 °), two paraboloids attached back-to-back can be used, as sketched in Figure 2(b). Each paraboloid captures rays from one hemisphere and reflects it to one of the two main directions.

To use the paraboloid as a 3D-to-2D mapping scheme all we have to do is finding the point $P = (x, y, z)$ on the paraboloid that reflects a given direction $\vec{v}$ towards the direction $d_0 = (0, 0, 1)$ (or $d_1 = (0, 0, -1)$ for the opposite hemisphere). Using Equation 1 we find the normal vector at P to be

$$\vec{n} = \frac{1}{z} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad . \tag{2}$$

Since the paraboloid is perfectly reflecting we simply calculate the halfway vector $\vec{h}$ which is equal to $\vec{n}$ up to some scaling factor. Using $\vec{h}$ and Equation 2 we can now formulate the 2D mapping of $\vec{v}$:

$$\vec{h} = \vec{d_0} + \vec{v} = k \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad v_z \geq 0. \tag{3}$$

For $v_z < 0$ d_0 gets replaced by d_1 which corresponds to the other hemisphere (paraboloid).

3.2 Dual-Paraboloid Shadow Mapping

As shown in Equation 3 the paraboloid mapping can be used to parameterize one hemisphere using 2D coordinates (x, y). This 2D coordinates can of course also be used to perform a shadow map lookup. Instead of computing a perspective projection $(x/z, y/z)$ all we have to do is to use the new mapping instead.

It is clear that this won't work alone because we still need some scalar value representing the depth of a pixel for the actual shadow test. Since the dual approach already divides the environment into positive and negative z regions, we can get along by just using the distance between the given surface point and the center of the paraboloid $(0, 0, 0)$. This way we extended the original paraboloid mapping to be a 3D-to-3D mapping which retains all information relevant for shadow mapping.

Hemispherical Point Light. In the case of a point light source with a field-of-view of 180° one paraboloid map is capable of storing all relevant depth information that can be seen by the light source. The shadow map is generated by first calculating the transformation that translates the light's position to $(0, 0, 0)$ and rotates the light direction (main axis) into either d_0 or d_1. For all surface points in front of the light source we compute the 2D coordinates (x and y of halfway vector scaled to $z = 1$) and store the point's distance to the origin (light source) at that shadow map position.

During the actual shadow test surface points get first transformed to the new light source coordinate system. If the transformed point is in front of the light source we calculate the paraboloid coordinates and compare the stored depth value with the actual distance of the point to the origin. The point is now in shadow if the stored depth value is less than the actual computed one. Otherwise the point is lit.

Omnidirectional Point Light. For light sources that illuminate the complete environment (360°) we have to use the dual-paraboloid approach since one paraboloid map only covers one hemisphere. To generate these two maps we compute the transformation that brings the point light to the origin $(0, 0, 0)$. For all surface points we first have to check which map is responsible for storing the information. Based on the sign of the z component we choose either the *front-facing* [2] paraboloid with $d_0 = (0, 0, 1)$ or the *back-facing* one with $d_1 = (0, 0, -1)$. After this test the shadow map position and entry is computed as before (using either d_0 or d_1 to compute the halfway vector) and stored in the selected shadow map.

When performing the shadow test all we have to do is to transform the surface point to the light source coordinate system, choose the corresponding shadow map and $d_{0/1}$ based on the sign of the z component and do the shadow map test as in the hemispherical case.

4 Implementation

Heidrich et. al. [6] showed that dual-paraboloid environment mapping can be implemented on standard graphics hardware (e.g. using OpenGL [11]). Since they used a pre-processing step to compute the environment map it is not obvious how paraboloid maps could be generated for fully dynamic environments. One, although very time consuming way would be to first generate a cube map (six rendering passes) and re-sample those images as described in [2]. Although this could be implemented using graphics hardware it would still be too slow for real time applications.

To speed up this generation phase the obvious way would be to render an image using the paraboloid mapping instead of the perspective projection normally used. Rasterization hardware renders triangles by perspective correct interpolation (homogeneous coordinates). Since this part of the hardware is fixed, we cannot directly map pixels to new positions as the paraboloid mapping would require.

However we can accept this linear interpolation if we assume that the scene geometry is tessellated fine enough. In this case we can simply transform the vertices of triangles according to the paraboloid mapping since the interpolated pixels won't differ too much from the exact solution due to the fine tessellation.

Mapping vertices according to the extended paraboloid mapping (2D coordinates and depth value) can easily be implemented using the so called programmable vertex engines

[2]The terms *front-* and *back-facing* are non significant and only used to divide the environment into two regions.

[7] available on state-of-the-art graphics cards (e.g. NVIDIA GeForce, ATI Radeon). These vertex programs[3] operate on a stream of vertices and replace the former fixed vertex pipeline (transformation, lighting, texture coordinate generation etc.) by a user-defined program. These programs are usually directly evaluated on the graphics hardware resulting in high speed and high flexibility.

4.1 Generation of Paraboloid Shadow Maps

Implementing the generation phase for one paraboloid shadow map using vertex programming is straight forward since the programming model supports all operations needed.

However we have to be careful about numerical stability. Since we want to include only those pixels that a really part of the hemisphere belonging to the chosen z axis (d_0 or d_1) we have to find a way of culling away unwanted pixels. Theoretically this test could be based solely on the range of valid coordinates $x^2 + y^2 \leq 1$ (Equation 1). For the implementation this fails due to numerical problems: When using $d_0 = (0, 0, 1)$ the normalization for points with $z \rightarrow -1$ the would break down due to the singularity at this point. If we would hand down these *undefined* x and y coordinates to the rasterization engine it would result in an undefined polygon being rasterized.

The solution to this is a per-pixel culling test. During generation of the shadow map we calculate an alpha value based on the z coordinate of the transformed vertex. This alpha value is mapped to unsigned value $[0; 1]$ by an offset of 0.5. Using the alpha test we can now cull away pixels based on the sign of the z coordinate which corresponds to either the front- or back-facing hemisphere being rendered.

With this scheme we can now implement a vertex program to generate a paraboloid shadow map for one hemisphere d_0 according to the following pseudo-code[4]:

```
$P' = M_{light} \cdot M_{model} \cdot P$
$P' = P'/P'_w$
$\texttt{output alpha} = 0.5 + \frac{P'_z}{z_{scale}}$    /* for alpha test */
$\texttt{len}_{P'} = ||P'||$
$P' = \frac{P'}{\texttt{len}_{P'}}$
$P' = P' + d_0$    /* halfway vector */
$P'_x = \frac{P'_x}{P'_z}$    /* $x$ paraboloid coordinate */
$P'_y = \frac{P'_y}{P'_z}$    /* $y$ paraboloid coordinate */
$P'_z = \frac{\texttt{len}_{P'} - z_{near}}{z_{far} - z_{near}} + z_{bias}$    /* distance */
$P'_w = 1$
$\texttt{output position} = P'$
```

First the incoming vertex is transformed and normalized by its w component. Next, an alpha value $\in [0; 1]$ is computed by scaling the z component by some user-defined constant (e.g. light's far plane z_{far}) and biasing it by 0.5 to conserve the sign of z. Using an alpha test function $\alpha \geq 0.5$ pixel values belonging to the opposite hemisphere d_1 are culled away.

Finally, we compute the paraboloid coordinates P'_x and P'_y as described previously. For P'_z we assign the scaled and biased distance from P' to $(0, 0, 0)$. Due to the precision

[3]Actually there are two names for the same functionality: OpenGL *vertex programs* and DirectX *vertex shaders*.

[4]Instead of the assembler code we choose a more readable form here.

of the depth buffer we have to use appropriate scaling and biasing factors here, similar to the selection of near and far clipping plane when using a perspective projection[5]. To avoid self shadowing artifacts we also have to move the z component slightly away from the light source using a bias value z_{bias} (similar as in [9]).

Generating a shadow map for the opposite hemisphere d_1 is trivial since we only have to flip the sign of P'_z after the transformation step and use the parameterization of d_0 as before:

```
$P' = M_{light} \cdot M_{model} \cdot P$
$P'_z = -P'_z$
...
```

4.2 Shadow Mapping with Paraboloid Shadow Maps

Implementing the shadow test using paraboloid mapping is as trivial as the generation step. In the case of an omnidirectional point light we compute the mappings for P'_0 (direction d_0) and P'_1 (direction d_1) and assign these as texture coordinates for texture unit 0 and 1. This step requires an additional scale and bias operation since texture coordinates need to be in the range of $[0;1]$.

In addition to this we also have to compute a value for selecting the right paraboloid map. By computing an alpha value based on the sign of the z component of P'_0 we can later select the appropriate texture map by checking $\alpha \geq 0.5$.

Finally the *normal* OpenGL vertex operations (camera transformation, perspective projection, lighting, additional texture coordinates etc.) are applied.

In pseudo-code the vertex program for rendering from the camera view using two paraboloid shadow maps will look like this:

```
$P'_0 = M_{light} \cdot M_{model} \cdot P$
...
output alpha $= 0.5 + \frac{P'_{0,z}}{z_{scale}}$
...
$P'_0 = P' + d_0$   /* hemisphere $(0,0,1)$ */
...
texcoords$_0 = 0.5 + 0.5 \cdot P'_0$   /* texcoords $\in [0;1]$ */
$P'_1 = M_{light} \cdot M_{model} \cdot P$
...
$P'_1 = P' + d_1$   /* hemisphere $(0,0,-1)$ */
...
texcoords$_1 = 0.5 + 0.5 \cdot P'_1$   /* texcoords $\in [0;1]$ */
...
normal OpenGL calculation for
lighting and vertex position
```

During the texturing stage we select the appropriate shadow test result depending on the computed alpha value:

[5]Computing the distance to the origin means that we have near and far clipping *spheres* instead of planes.

```
if$(\alpha \geq 0.5)$then
  $res = tex_0$   /* 1 for lit, 0 for shadow */
else
  $res = tex_1$   /* 1 for lit, 0 for shadow */
endif
$output_{RGB} = res \cdot full\ illumination\ +$
    $(1 - res) \cdot ambient\ illumination$
```

This can be implemented using programmable texture blending (e.g. NVIDIA's register combiners [8]) which is available on all state-of-the-art graphics cards.

For hemispherical point lights only one texture unit is used for shadow mapping and the result of one *if*-clause sets *res* $= 0$ (one hemisphere always in shadow).

5 Results

We implemented and tested our shadow mapping technique on an AMD Athlon Linux PC equipped with a NVIDIA GeForce3 64Mb graphics card using OpenGL as a graphics API. This cards supports vertex programs, programmable texture blending and also multitexturing with up to four texture units. This way we can include the shadow test when rendering the final scene while still having two (or three) texture units left for the scene's textures.

As stated in previous sections, the number of rendering passes is two for hemispherical point lights (one shadow map generation and one final rendering pass) versus three for a omnidirectional light source (two for generating two hemispherical shadow maps and one final rendering pass using two texture units for the shadow test).

Figure 3 shows a scene illuminated by an omnidirectional point light source located in the middle of the room. This scene was directly imported from 3D Studio Max and only the wall polygons have been tesselated further (each wall subdivided into 64 quads) to avoid the interpolation problems addressed in Section 4. On a GeForce3 this scene can be rendered in real time (> 30 frames per second) including dynamic update of shadow maps at each frame (three rendering passes in total).

Figure 4 shows the results for a hemispherical light source located at the top of the room. As in Figure 3 only the surrounding walls and the floor polygon had been further subdivided. In this example an additional texturing unit is used to include the scene's textures in the final pass. This scene can also be rendered at real time rates (one shadow map generation pass and one final pass).

Both examples had been rendered using a resolution of 512 by 512 pixels for both, shadow maps and final image. Shadow quality can be further improved by generating high resolution maps using offscreen buffers.

6 Conclusion

In this paper we have shown that the dual paraboloid approach presented by [6] can easily be adopted for shadow mapping. This approach is well suited for hemispherical and omnidirectional light sources. By utilizing advanced graphics cards features the algorithm runs completely in hardware and uses only a minimal amount of rendering passes.

Although the algorithm fails for very large polygons due to the linear interpolation performed during rasterization this is in most cases no problem: For games and other interactive applications most geometry is already very high-detailed. The only exception to this are large polygons representing walls, floor, ceiling etc. These polygons could either be further tesselated or simply ignored during the shadow map generation phase. Later is valid in cases where the visible part of the scene is bounded by such polygons, meaning those parts of the scene won't cast visible shadows anyway.

Since our method differs from the traditional shadow mapping approach only in terms of parameterization, we are able to apply all known quality improvements (e.g. per centage closer filtering [9]) without any efforts.

7 Acknowledgements

We would like to thank Prof. Wolfgang Heidrich of the University of British Columbia, Canada, for valuable discussions on this topic.

References

[1] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542—546, 1976.

[2] David Blythe. Advanced graphics programming techniques using opengl. SIGGRAPH Course, 1999.

[3] Franklin C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 242–248, July 1977.

[4] Sim Dietrich. *Shadow Techniques*. NVIDIA Corporation, 2001. PowerPoint Presentation available from http://www.nvidia.com.

[5] Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. *Fourth Eurographics Workshop on Rendering*, pages 259–266, June 1993. Held in held in Paris, France, 14-16 June 1993.

[6] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 39–46, August 1998. Held in Lisbon, Portugal.

[7] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. *Proceedings of SIGGRAPH 2001*, pages 149–158, August 2001. ISBN 1-58113-292-1.

[8] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, October 1999. Available from http://www.nvidia.com.

[9] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 283–291, July 1987.

[10] Marc Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadow and lighting effects using texture mapping. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 249–252, July 1992.

[11] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2)*, 1998.

[12] Douglas Voorhies and Jim Foran. Reflection vector shading hardware. *Proceedings of SIGGRAPH 94*, pages 163–166, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[13] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, August 1978.

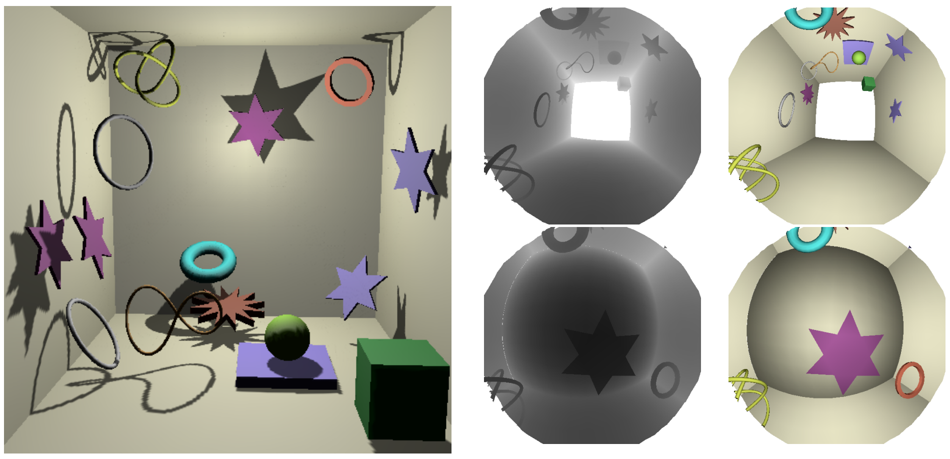

Fig. 3. Left: Scene illuminated by an omnidirectional point light (located in the middle of the room). Middle column: The two paraboloid shadow maps used (shown as grey scale images). Right column: The two hemispheres as seen by the light source.

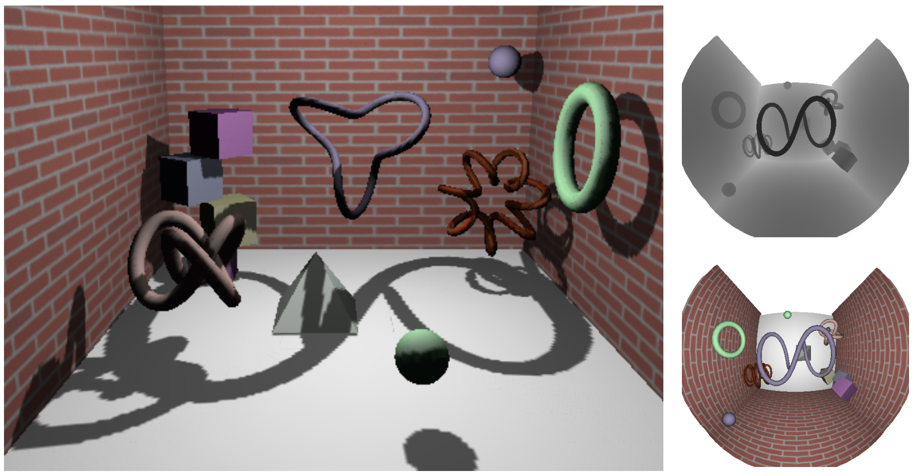

Fig. 4. Left: Scene illuminated by a hemispherical point light. Right column: Shadow map and light source view (paraboloid mapping).