View-independent Environment Maps

Wolfgang Heidrich and Hans-Peter Seidel

Computer Graphics Group
University of Erlangen
{heidrich,seidel}@informatik.uni-erlangen.de

Abstract

Environment maps are widely used for approximating reflections in hardware-accelerated rendering applications. Unfortunately, the parameterizations for environment maps used in today's graphics hardware severely undersample certain directions, and can thus not be used from multiple viewing directions. Other parameterizations exist, but require operations that would be too expensive for hardware implementations.

In this paper we introduce an inexpensive new parameterization for environment maps that allows us to reuse the environment map for any given viewing direction. We describe how, under certain restrictions, these maps can be used today in standard OpenGL implementations. Furthermore, we explore how OpenGL could be extended to support this kind of environment map more directly.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and framebuffer operations; I.3.6 [Computer Graphics]: Methodology and Techniques—Standards I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, Shadowing and Texture I.4.1 [Image Processing and Computer Vision]: Digitization and Image Capture—Sampling

Keywords: Environment mapping, OpenGL

1 Introduction

The basic idea of environment maps is striking [1]: if a reflecting object is small compared to its distance from the environment, the incoming illumination on the surface really only depends on the direction of the reflected ray. Its origin, that is the actual position on the surface, can be neglected. Therefore, the incoming illumination at the object can be precomputed and stored in a 2-dimensional texture map.

If the parameterization for this texture map is cleverly chosen, the illumination for reflections off the surface can be looked up very efficiently. Of course, the assumption of a small object compared to the environment often does not hold, but environment maps are a good compromise between rendering quality, and the need to store the full, 4-dimensional radiance field on the surface.

Both offline [3, 9] and interactive, hardware-based renderers [8] have used this implementation of reflections, often with amazing results.

Given the above description of environment maps, one would think that it should be possible to use a single map for all viewing positions and directions. After all, the environment map is supposed to contain information about illumination from *all* directions. Thus, it should be possible to modify the lookup process in order to extract the correct information for all possible points of view.

In reality, however, this is not quite true. The parameterization used in most of today's graphics hardware exhibits a singularity as well as areas of extremely poor sampling. As a consequence, this form of environment map cannot be used for any viewing direction except the one for which it was originally generated.

This leaves us with a situation, where the environment map has to be re-generated for each frame even in simple applications such as walkthroughs. Other parameterizations exist (see Section 2), but require operations that would be too expensive for hardware implementations. In this paper we introduce a parameterization for environment maps that uses only simple operations (additions, multiplications and matrix operations), but provides a good enough sampling so that *one* map can be used for *all* viewing directions.

In the following, we first discuss existing parameterizations for environment maps and their deficiencies. Then, in Section 3, we introduce our new parameterization, and describe how it can be used on contemporary graphics hardware under certain restrictions (Section 5). Finally, we propose a simple extension to the texture coordinate generation of OpenGL that would add direct support for our method (Section 6), and we present results of our implementation in (Section 7).

2 Previous Work

The parameterization used most commonly in computer graphics hardware today, are *spherical environment maps* [8]. It is based on the simple analogy of a small, perfectly mirroring ball centered around the object. The image that an orthographic camera sees when looking at this ball from a certain viewing direction is the environment map. An example environment map from the center of a colored cube is shown in Figure 1.

The sampling rate of this map is maximal for directions opposing the viewing direction (that is, objects behind the viewer), and goes towards zero for directions close to the viewing direction. Moreover, there is a singularity the in viewing direction, since all points where the viewing vector is tangential to the sphere show the same point of the

1998 Workshop on Graphics Hardware Lisbon Portugal

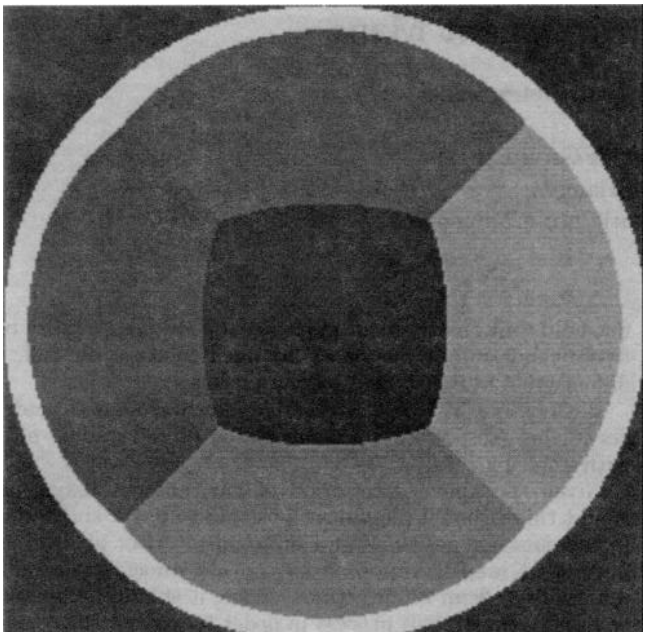

Figure 1: A spherical environment map from the center of a colored cube. Note the bad sampling of the cube face directly in front of the observer (light gray).

environment.

With these properties, it is clear that this parameterization is not suitable for viewing directions other than the original one. The major reason why it is used anyway, is that the lookup can be computed efficiently with simple operations in hardware. The parameterization proposed in this paper solves the sampling problems while at the same time maintaining the simplicity of the lookup process.

Another parameterization are *latitude-longitude maps* [9]. Here, the s, and t parameters of the texture map are interpreted as the latitude and longitude, respectively, with respect to a certain viewing direction. Apart from the fact that these maps are severely oversampled around the poles, the lookup process involves the computation of inverse trigonometric functions, which is inappropriate for hardware implementations.

Finally, *cubical environment maps* [2, 10] consist of 6 independent perspective images from the center of a cube through each of its faces. The sampling of these maps is fairly good, as the sampling rates for the directions differ by a factor of $3\sqrt{3} \approx 5.2$. Also, the lookup process within each of the 6 images is inexpensive. However, the difficulty is to decide which of the six images to use for the lookup. This requires several conditional jumps, and interpolation of texture coordinates is difficult for polygons containing vertices in more than one image. Because of these problems cubical environment maps are difficult and expensive to implement in hardware, although they are quite widespread in software renderers (e.g. [9]).

Many interactive systems initially obtain the illumination as a cubical environment map, and then resample this information into a spherical environment map. There are two ways this can be done. The first is to re-render the cubical map for every frame, so that the cube is always aligned with the current viewing direction. Of course this is slow if the environment contains complex geometry. The other method is to generate the cubical map only once, and then re-compute the mapping from the cubical to the spherical map for each frame. This, however, makes the resampling step more expensive, and can lead to numerical problems around the singularity.

In both cases, the resampling can be performed as a multipass algorithm in hardware, using morphing and texture mapping. Yet, the bandwidth imposed by this method onto the graphics system is quite large: the six textures from the cubical representation have to be loaded into texture memory, and the resulting image has to be transferred from the framebuffer into texture RAM or even into main memory.

3 A New Parameterization

The parameterization we use is based on an analogy similar to the one used to describe spherical environment maps. Assume that the reflecting object lies in the origin, and that the viewing direction is along the negative z axis. The image seen by an orthographic camera when looking at the paraboloid

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

contains the information about the hemisphere facing towards the viewer. The complete environment is stored in two separate textures, each containing the information of one hemisphere. The geometry is depicted in Figure 2.

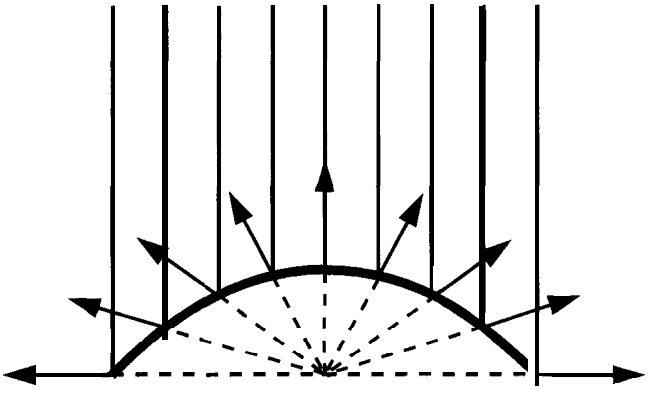

Figure 2: The rays of an orthographic camera reflected on a paraboloid sample a complete hemisphere of directions.

It should be noted that this parameterization has recently been introduced by Nayar [6, 5] in a different context. Nayar actually built a lens and camera system that is capable of capturing this sort of image from the real world. Besides raytracing and resampling of cubical environment maps, this is actually one way of acquiring maps in the proposed format. Since two of these cameras can be attached back to back [5], it is possible to create full 360° images of real world scenes.

The geometry described above has some interesting properties. Firstly, the reflection rays in each point of the paraboloid all intersect in a single point, the origin (see dashed lines in Figure 2). This means that the resulting image can indeed be used as an environment map for an object in the origin.

Secondly, the sampling rate varies by a factor of 4 over the complete image, as depicted in Figure 3. Pixels in the outer regions of the map cover only 1/4 of the solid angle covered by center pixels. This means that directions perpendicular

to the viewing direction are sampled at a higher rate than directions parallel to the viewing direction. Depending on how we select mipmap levels, the factor of 4 in the sampling rate corresponds to one or two levels difference, which is quite acceptable. In particular this is better than the sampling of cubical environment maps.

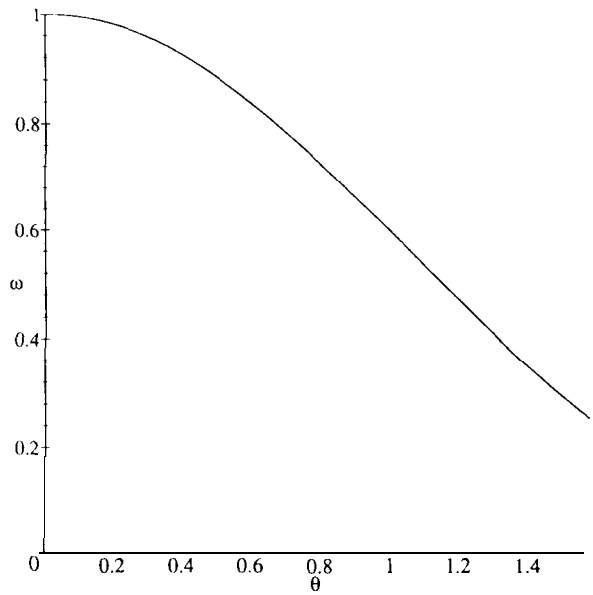

Figure 3: The change of solid angle ω covered by a single pixel versus the angle θ between the viewing direction and the reflected ray. The sampling rate varies by a factor of 4 over the environment map. Directions perpendicular to the viewing direction are sampled at a higher rate than directions parallel to the viewing direction.

Figure 4 shows the two images comprising an environment map for the simple scene used in Figure 1. The top image represents the hemisphere facing towards the camera, while the bottom image represents the hemisphere facing away from it.

4 Lookups from Arbitrary Viewing Positions

In the following, we describe the math behind the lookup of a reflection value from an arbitrary viewing position. We assume that environment maps are specified relative to a coordinate system, where the reflecting object lies in the origin, and the map is generated for a viewing direction of $\mathbf{d}_o = (0, 0, -1)^T$. It is not necessary that this coordinate system represents the object space of the reflecting object, although this would be an obvious choice. However, it is important that the transformation between this space and eye space is a rigid body transformation, as this means that vectors do not have to be normalized after transformation. To simplify the notation, we will in the following use the term "object space" for this space.

In the following, $\mathbf{v}_e$ denotes the normalized vector from the eye point to the point on the surface, while the vector $\mathbf{n}_e = (n_{e,x}, n_{e,y}, n_{e,z})^T$ is the normal of the surface point in eye space. Furthermore, the (affine) model-view matrix is

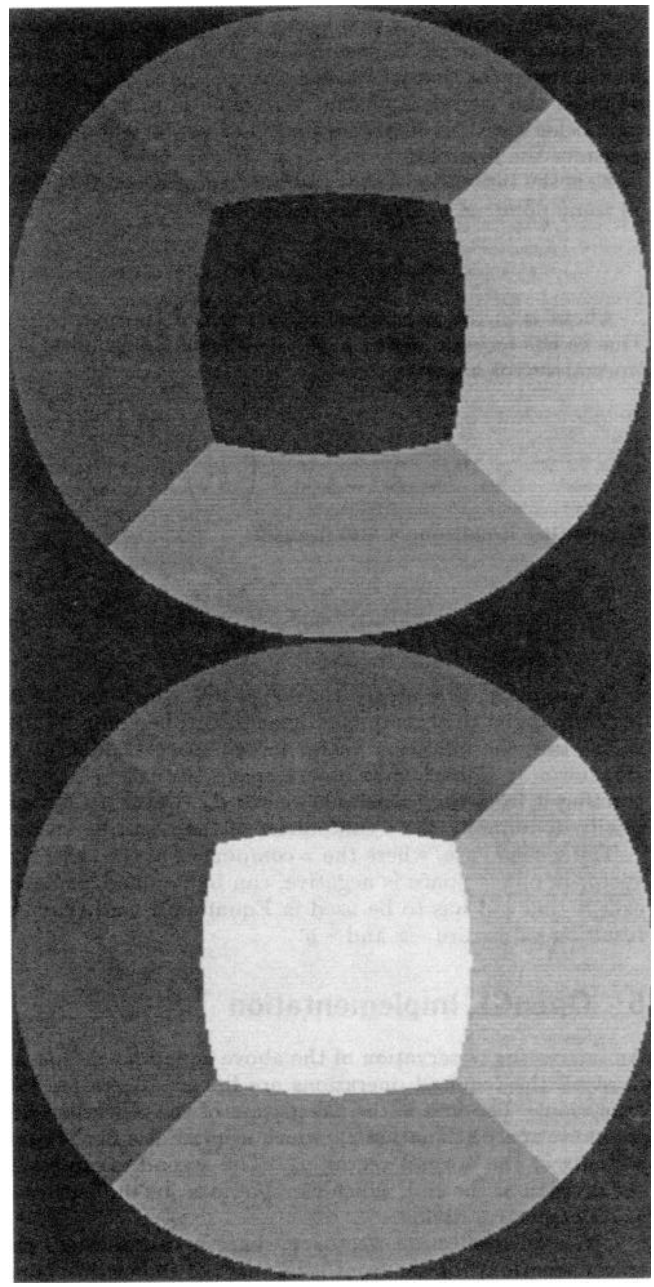

Figure 4: The two textures comprising an environment map for an object in the center of a colored cube.

given as $\mathbf{M}$. This means, that the normal vector in eye space $\mathbf{n}_e$ is really the transformation $\mathbf{M}^{-T}\ \mathbf{n}_o$ of some normal vector in object space. If $\mathbf{M}$ is a rigid body transformation, and $\mathbf{n}_o$ was normalized, then so is $\mathbf{n}_e$. The reflection vector in eye space is then given as

$$\mathbf{r}_e = \mathbf{v}_e + 2\,\langle \mathbf{n}_e, -\mathbf{v}_e \rangle \cdot \mathbf{n}_e. \tag{2}$$

Transforming this vector with the inverse of $\mathbf{M}$ yields the reflection vector in object space:

$$\mathbf{r}_o = \mathbf{M}^{-1}\,\mathbf{r}_e. \tag{3}$$

The illumination for this vector in object space is stored somewhere in one of the two images. More specifically, if the z component of this vector is positive, the vector is facing towards the viewer, and thus the value is in the first image, otherwise it is in the second. Let us, for the moment, consider the first case.

$\mathbf{r}_o$ is the reflection of the constant vector $\mathbf{d}_o = (0, 0, -1)^T$ in some point (x, y, z) on the paraboloid:

$$\mathbf{r}_o = \mathbf{d}_o + 2\langle \mathbf{n}, -\mathbf{d}_o \rangle \cdot \mathbf{n}, \tag{4}$$

where $\mathbf{n}$ is the normal at that point of the paraboloid. Due to the formula of the paraboloid from Equation 1, this normal vector happens to be

$$\mathbf{n} = \frac{1}{\sqrt{x^2 + y^2 + 1}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{5}$$

Combining Equations 4 and 5 yields

$$\mathbf{d}_o - \mathbf{r}_o = 2\langle \mathbf{n}, \mathbf{v} \rangle \mathbf{n} = \begin{pmatrix} k \cdot x \\ k \cdot y \\ k \end{pmatrix}. \tag{6}$$

In summary, this means that x and y, which can be directly mapped to texture coordinates, can be computed by calculating the reflection vector in eye space (Equation 2), transforming it back into object space (Equation 3), subtracting it from the (constant) vector $\mathbf{d}_o$ (Equation 6), and finally dividing by the z component of the resulting vector.

The second case, where the z component of the reflection vector in object space is negative, can be handled similarly, except that $-\mathbf{d}$ has to be used in Equation 6, and that the resulting values are $-x$ and $-y$.

5 OpenGL Implementation

An interesting observation of the above equations is that almost all the required operations are linear. There are two exceptions. The first is the calculation of the reflection vector in eye space (Equation 2), which is quadratic in the components of the normal vector $\mathbf{n}_e$. The second exception is the division at the end, which can, however, be implemented as a perspective divide.

Given the reflection vector $\mathbf{r}_e$ in eye coordinates, the transformations for the frontfacing part of the environment can be written in homogeneous coordinates as follows:

$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} = \mathbf{P} \cdot \mathbf{S} \cdot (\mathbf{M}_l)^{-1} \cdot \begin{bmatrix} r_{e,x} \\ r_{e,y} \\ r_{e,z} \\ 1 \end{bmatrix}, \tag{7}$$

where

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

is a projective transformation that divides by the z component,

$$\mathbf{S} = \begin{bmatrix} -1 & 0 & 0 & d_{o,x} \\ 0 & -1 & 0 & d_{o,y} \\ 0 & 0 & 1 & d_{o,z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

computes $\mathbf{d}_o - \mathbf{r}_o$, and $\mathbf{M}_l$ is the linear part of the affine transformation $\mathbf{M}$. Another matrix is required for mapping x and y into the interval $[0, 1]$ for the use as texture coordinates:

$$\begin{bmatrix} s \\ t \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix}$$

Similar transformations can be derived for the backfacing parts of the environment. These matrices can be used as texture matrices, if $\mathbf{r}_e$ is specified as the texture coordinate for the vertex. Note that $\mathbf{r}_e$ changes from vertex to vertex, while the matrices remain constant.

Due to the non-linearity of the reflection vector, $\mathbf{r}_e$ has to be computed in software. This is the step that corresponds to the automatic generation of texture coordinates for spherical environment maps in OpenGL (`glTexGen`). Actually, this process can be further simplified by assuming that the vector $\mathbf{v}$ from the eye to the object point is constant. This is true, if the object is far away from the camera, compared to its size, or if the camera is orthographic. Otherwise, the assumption breaks down, which is particularly noticeable on flat objects.

What remains to be done is to combine frontfacing and backfacing regions of the environment into a single image. To this end we use OpenGL's alpha test feature. In the two texture maps, we mark those pixels inside the circle $x^2 + y^2 \leq 1$ with an alpha value of 1, the pixels outside the circle with an alpha value of 0. Then the algorithm works as follows:

```
glAlphaFunc( GL_EQUAL, 1.0 );
glEnable( GL_ALPHA_TEST );
glMatrixMode( GL_TEXTURE );

glBindTexture( GL_TEXTURE_2D,
               frontFacingMap );
glLoadMatrix( frontFacingMatrix );
draw object with r_o as texture coord.

glBindTexture( GL_TEXTURE_2D,
               backFacingMap );
glLoadMatrix( backFacingMatrix );
draw object with r_o as texture coord.
```

The important point here is that backfacing vectors $\mathbf{r}_o$ will result in texture coordinates $x^2 + y^2 > 1$ while the matrix for the frontfacing part is active, and will thus not be rendered. Similarly frontfacing vectors will not be rendered while the matrix for the backfacing part is active.

6 Extending OpenGL

While the method described in Section 5 works, and is also quite fast (see Section 7), it is not the best one could hope for. Firstly, since the texture coordinates have to be generated in software, it is not possible to use display lists[1]. Secondly, many vectors required to compute the reflected vector

[1] Actually, since the texture coordinates are identical for both passes of our method, display lists can be used to render the two passes within one frame, but they cannot be reused for other frames.

$\mathbf{r}_e$ are already available further down the pipeline (that is, when OpenGL texture coordinate generation takes place), but are not easily available in software.

For example, the normal vector in a vertex is typically only known in object space. In order to compute $\mathbf{n}_e$, this vector has to be transformed by hand, although the transformed normal is later being transformed by the hardware anyway (for lighting calculations).

Interestingly, the texture coordinates generated for spherical environment maps are the x and y components of the halfway vector between the reflection vector $\mathbf{r}_e$ and the negative viewing direction $(0, 0, 1)^T$ [8]. Thus, current OpenGL implementations essentially already require the computation of $\mathbf{r}_e$ for environment mapping.

We would like to emphasize that this computation of the reflection vector is really necessary, even with the standard OpenGL spherical environment maps. On first sight, one could think that for this parameterization, it would be possible to directly use the x and y components of the surface normal as texture coordinates. This, however, would only yield the desired result for orthographic views. For perspective views it would lead to sever artifacts, such as flat mirroring surfaces being colored in a single, uniform color.

Because OpenGL implementations already compute the reflection vector, the changes necessary to fully support our parameterization are minimal. All that is required, is a new mode for texture coordinate generation that directly stores $\mathbf{r}_e$ into the s, t, and r texture coordinates, and sets q to 1.

Furthermore, and independent of this proposal, it would be possible get rid the second rendering pass by allowing for multiple textures to be bound at the same time. OpenGL extensions for this purpose are currently being discussed by the OpenGL architecture review board.

7 Results

We have implemented the software-based method described in Section 5, and tested it with several scenes. Figure 5 shows two images making up one environment map. These images where generated using ray-casting. The circles indicate regions with $x^2 + y^2 \leq 1$. The regions outside the circles are do not really belong to the map, but have been generated during the ray-casting step by extending the paraboloid to the domain $-1 \leq x, y \leq 1$. We found it useful to have an additional ring of one or two pixels available outside the actual circle, in order to avoid seams in regions where front- and backfacing regions touch.

Figure 6 shows a sphere to which this environment map has been applied. The images have been taken from different viewpoints, but with the same environment map. This image can be rendered in full screen (1280 × 1024) at about 15 frames per second on an SGI O2 and at > 20 frames per second on an SGI RealityEngine2. The tessellation used for this sphere was 72 × 72 quadrilaterals.

A closeup of the seams between frontfacing and backfacing regions of the environment map can be seen in Figure 7. In the left image, the curve indicates this seam, which hard to detect in the right image.

Finally, Figure 8 shows a torus with the environment map applied. Here we used a tessellation of 144 × 72, and the timings where 13 frames per second on the O2, and > 20 frames per second on the RealityEngine2, again at full screen resolution

Based on the discussion in Section 6, we are confident, that these times could be further improved, if some minor extensions were made to the texture coordinate generation mechanism of OpenGL.

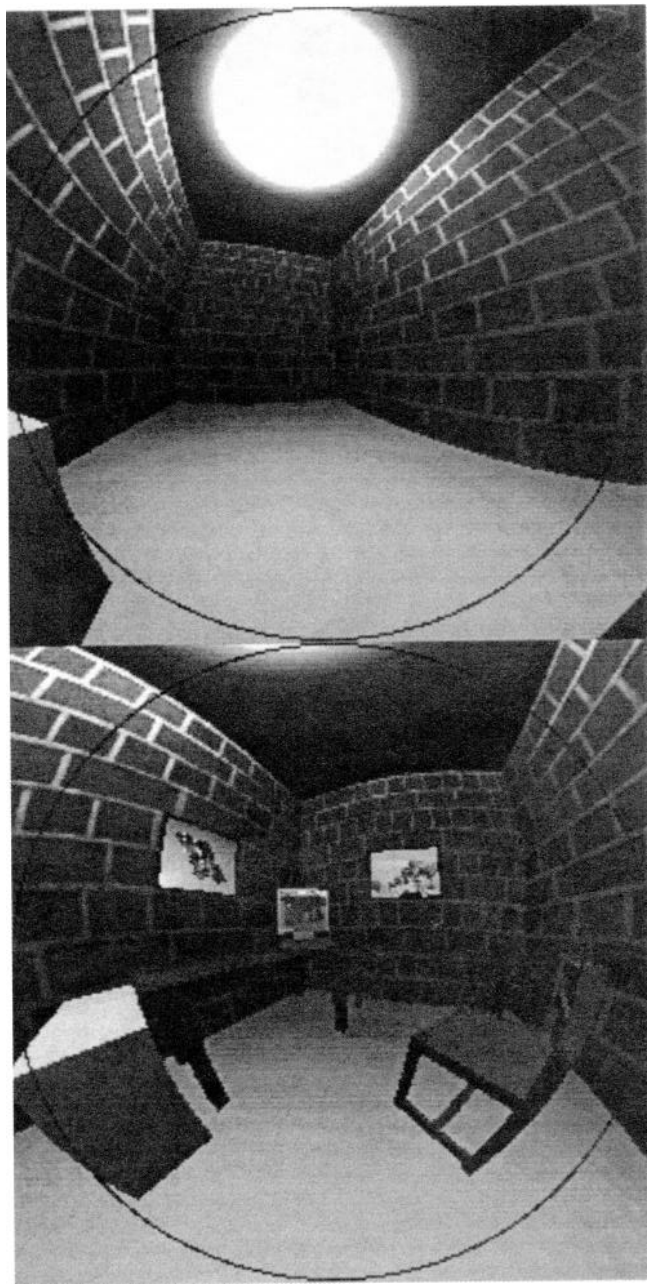

Figure 5: The two textures of the environment map for an object in the center of an office scene.

8 Conclusions

In this paper, we have introduced a novel parameterization for environment maps, which allows us to reuse them from multiple viewpoints. This allows us to generate walkthroughs of scenes with reflecting objects without the need to recompute environment maps for each frame.

We have shown how the new parameterization can be used today in standard OpenGL implementations. Although this method is partly based on a software algorithm, we have

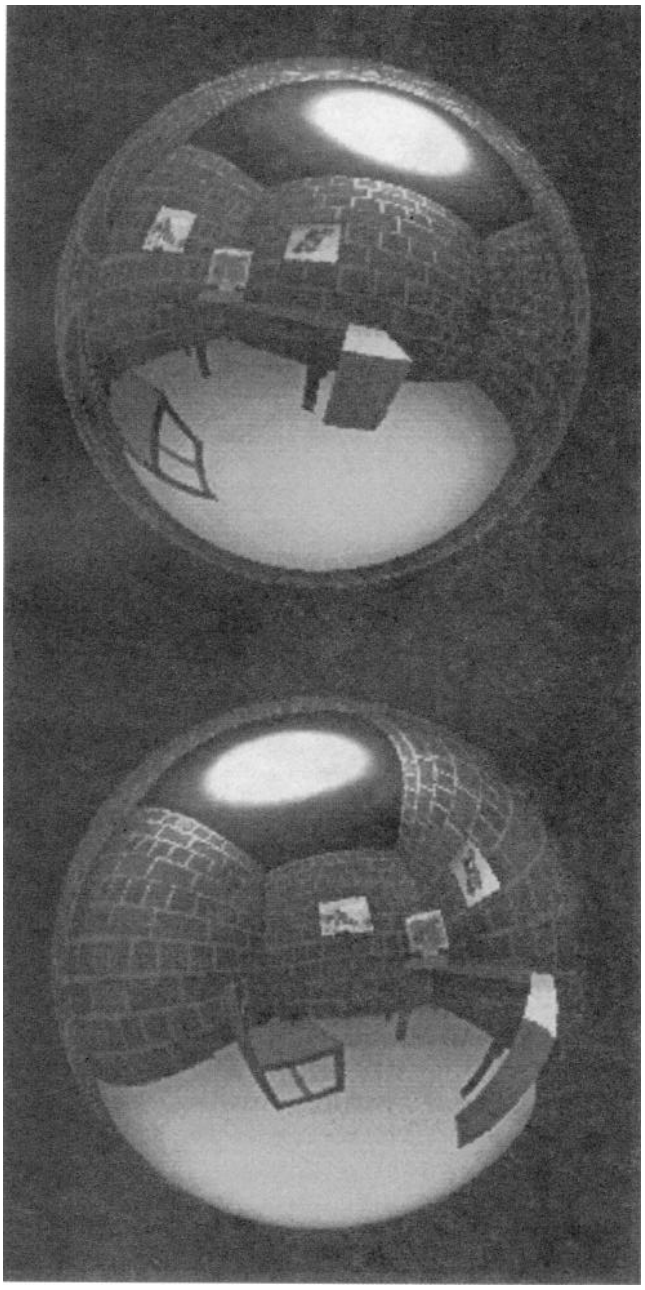

Figure 6: The environment map from Figure 5 applied to a sphere seen from two different viewpoints.

demonstrated it to be efficient enough for many practical purposes.

Further performance improvements are possible by adding some direct support for our parameterization into the OpenGL API. Only minimal changes would be necessary for this support, and they would be backwards compatible to the current interface.

References

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

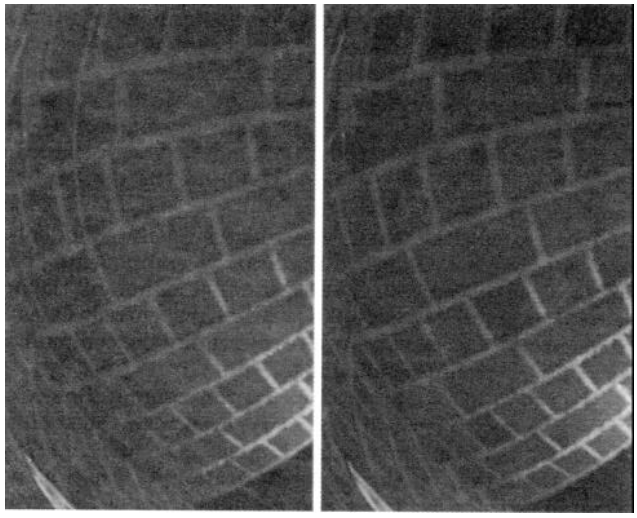

Figure 7: A closeup of the seams between the frontfacing and the backfacing regions of the environment map. The curve in the left image indicates the location of the seam, which is hard to detect in the right image.

[2] Ned Greene. Applications of world projections. In M. Green, editor, *Proceedings of Graphics Interface '86*, pages 108–114, May 1986.

[3] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculation. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):289–298, August 1990.

[4] Shree Nayar. Omnicamera home page. Available from http://www.cs.columbia.edu/CAVE/omnicam, 1997.

[5] Shree Nayar. Omnidirectional image sensing. Invited Talk at the 1998 Workshop on Image-Based Modeling and Rendering, March 1998.

[6] Shree K. Nayar. Catadioptric omnidirectional camera. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 482–488, June 1997.

[7] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison Wesley, 1993.

[8] OpenGL ARB. *OpenGL Specification, Version 1.1*, 1995.

[9] Pixar. *The RenderMan Interface, Version 3.1*. Pixar, San Rafael, CA, September 1989.

[10] Douglas Voorhies and Jim Foran. Reflection vector shading hardware. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 163–166, July 1994.

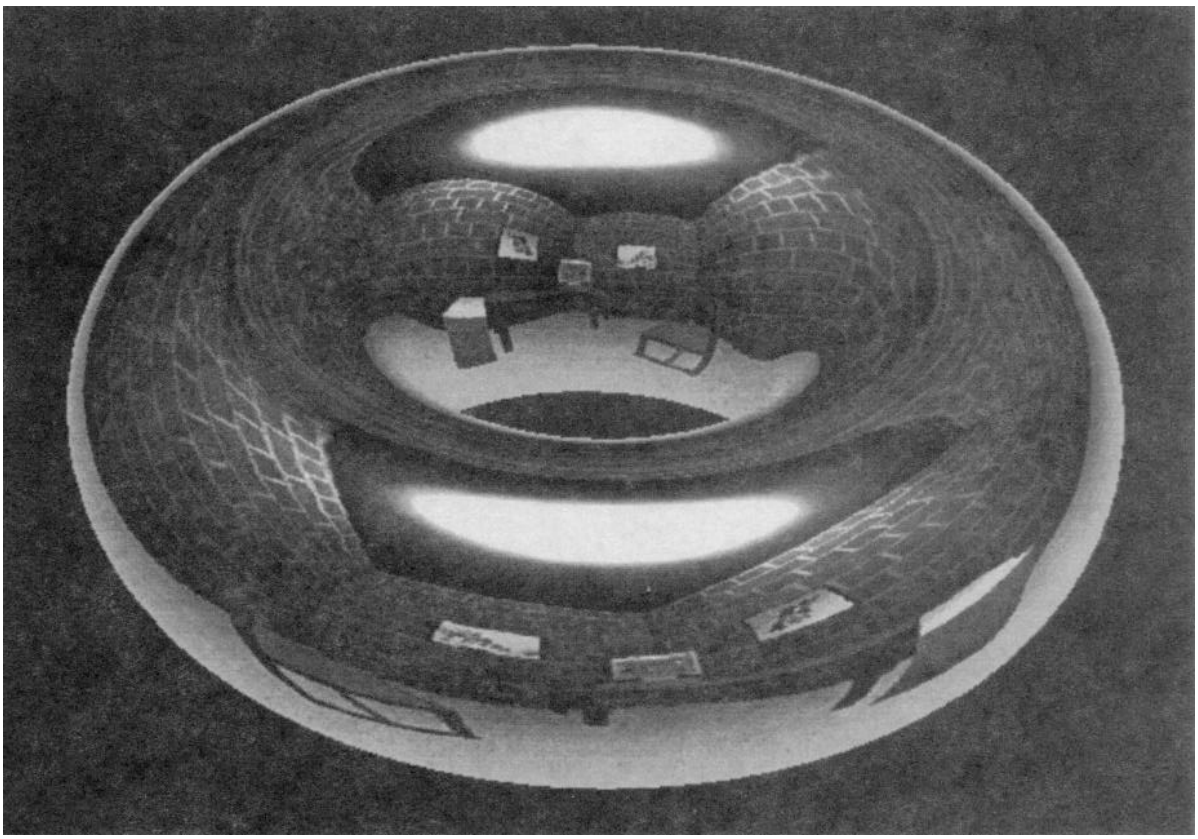

Figure 8: A torus with view-independent environment mapping, rendered using OpenGL