# COMP238: Raster Graphics

Final Project:

*Spherical, Cubic, and Parabolic Environment Mappings*

by Paul Zimmons

December 10, 1999

This project shows the mechanics of how each of three types of environment mappings work, features of each mapping, problems associated with each type of mapping, and simulations of their indexing schemes.

## Introduction

### *"What is environment mapping?"*

Environment mapping (also called *reflection mapping*) is a method of adding realism to a scene by using special texture indexing methods. In general, the object appears to mirror the environment around it. A normal texture map applies an image onto an object based on hard-coded texture coordinates specified at each vertex. A single triangle making up part of a textured surface will reference 3 points in the texture map. These points are identified in the texture's s-t (or also called u-v) coordinate system. Texture coordinates range from (0,0) to (1,1) for an image unless some method of wrapping arbitrary texture coordinates around to (0,0) and (1,1) is implemented. Generally, s is represented as being along the texture's x axis and t is along the texture's y axis. For example, a texture might look like Figure 1.

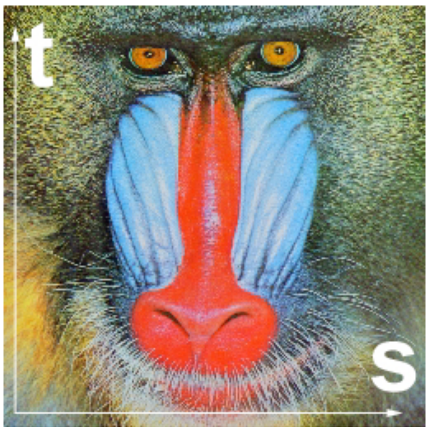


*Figure 1: A texture with the S and T texture axes labeled.*

When a texture is mapped onto a triangle that composes a surface, the triangle's (s,t) coordinates at each corner define a region in the texture to apply to the polygon, much like a flexible sticker. This is illustrated in Figure 2.

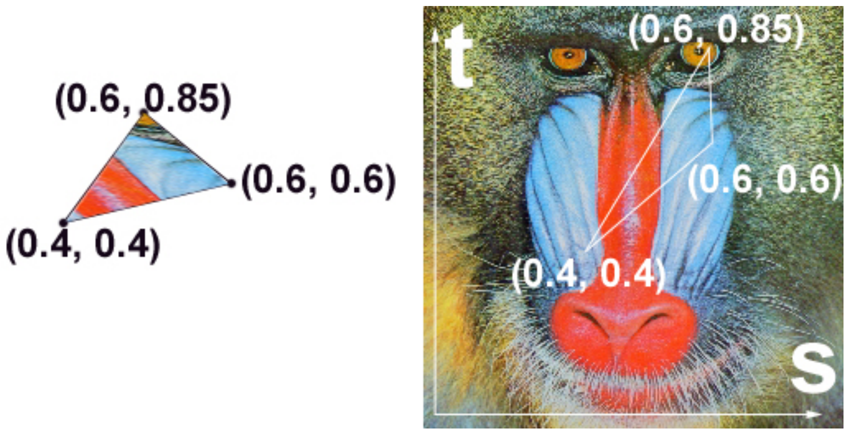


*Figure 2: The application of the mandril image to a polygon by using each vertex's (s,t) texture coordinates.*

Instead of manually generating texture coordinates for each vertex, environment mapping uses properties of the surface to automatically generate them. By performing this special texture indexing, the object will appear to reflect the environment around it. In order for the environment mapping to be convincing, the texture used should be derived from the environment the object is in. For example, an environment mapping teapot in a room should use a texture representing the teapots point of view in the room. Although a static environment map may look convincing, if the environment mapped object moves or the scene changes, the environment map may need to be recomputed on the fly in order to reflect the current scene.

All the environment mapping techniques discussed in this project operate by calculating a *reflected vector* off the surface. This reflected vector is in a mirror direction from the viewer's eye position (and sometimes orientation) as seen in Figure 3. To calculate the mirror direction, we flip the incoming vector from the eye about a vector normal (perpendicular) to the surface.

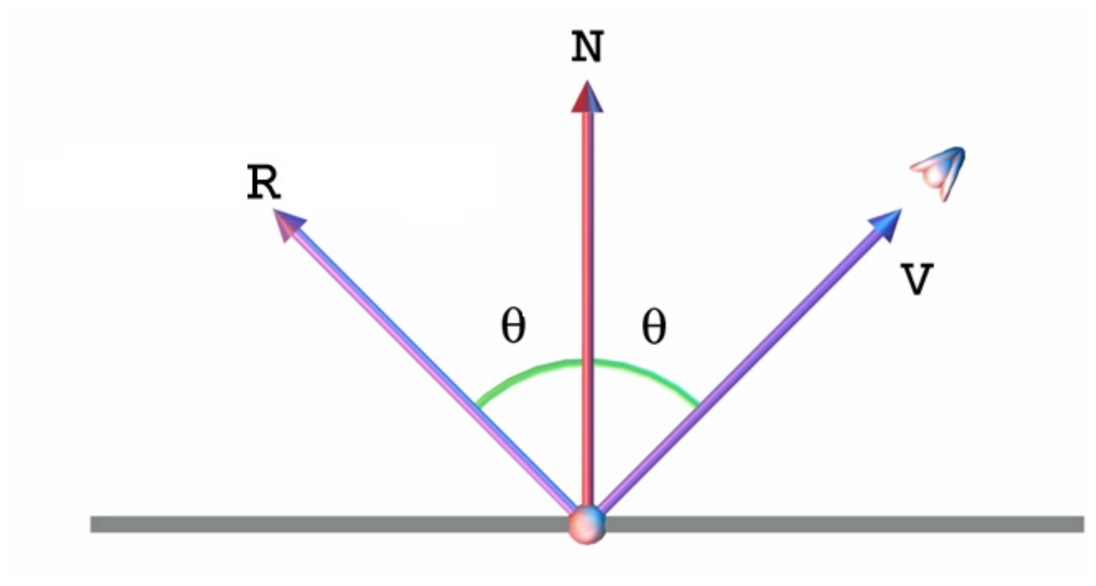
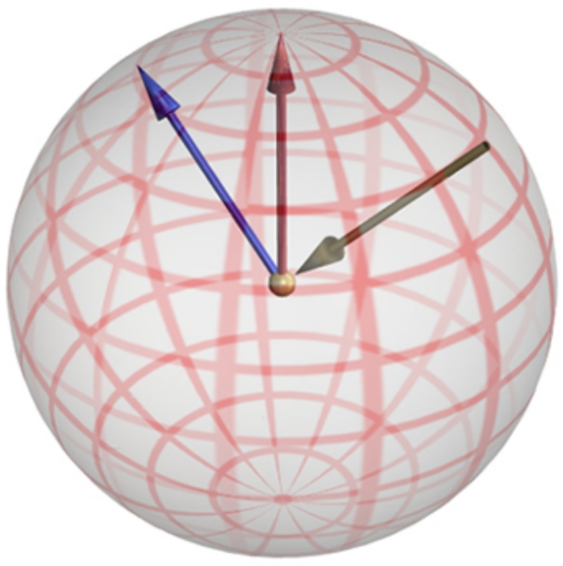
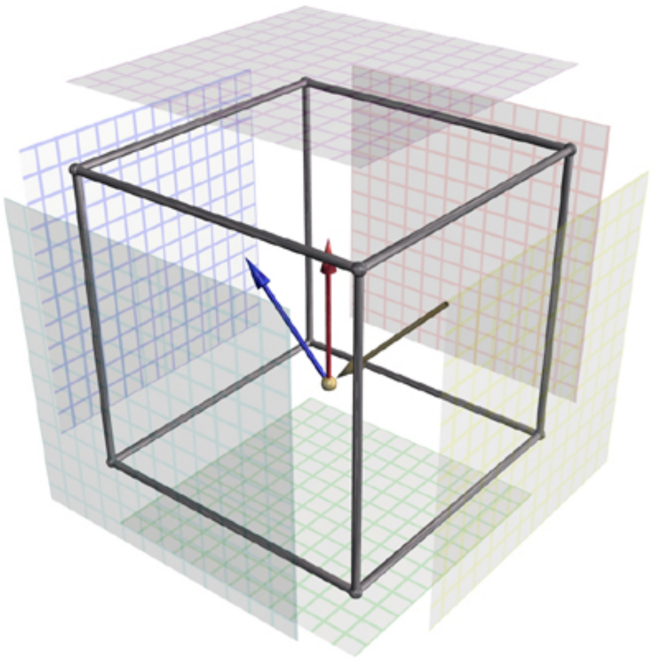
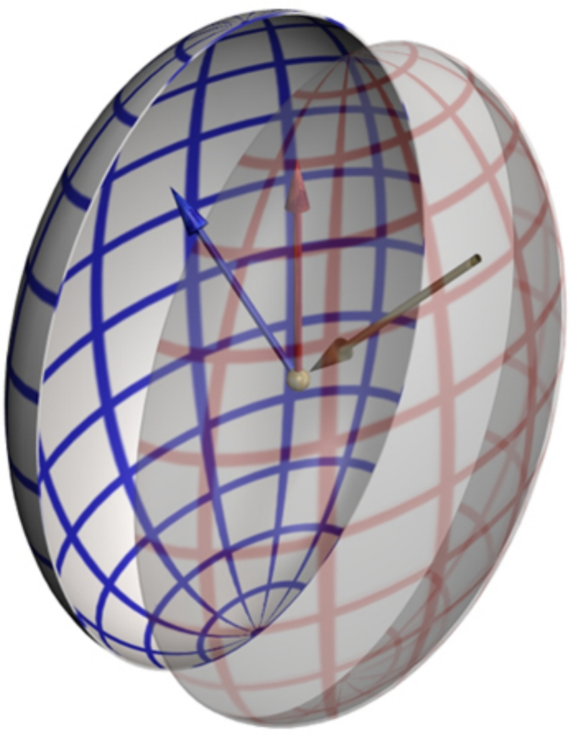


*Figure 3: The reflected vector, R, derived from a vector to the viewer, V, and the normal, N, to the surface at a vertex.*

The formula for R for this configuration of vectors is: **R** = **V** - 2(**V** dot **N**)**N**. (A derivation of this formula can be found in page 19 of *3D Computer Graphics* by Alan Watt.) The environment mapping techniques differ in where this reflected vector **R** is calcaluted (whether it is calcualted relative to the screen or the world, etc.) and how it is used to derived the texture coordinates for a vertex. *Intuitively*, we can think of spherical, cubic, and paraboloid mapping as using R to intersect a sphere, cube, or a pair of parabolas in order to get the texture coordinate. However, these images are not literally what goes on in the mappings. Figures 4, 5, and 6 illustrate the idea behind the mappings.







*Figures 4, 5 and 6. The intuitive notions behind spherical, cubic, and dual paraboloid mappings. The Yellow vector is coming from the viewer's eye, the Red vector is a normal to the dot at the center, and the Blue vector is a reflected vector. These calculations occur in different spaces depending on which mapping is being performed.*

As you can see, the different environment mapping techniques use different geometric assumptions as well as different numbers of texture maps. Sphere mapping utilizes 1 texture map, cubic mapping 6, and dual paraboloid mapping 2. The number of texture maps also affects how much an environment mapping technique distorts its environment. For example, 1 map for spherical environment mapping usually contains less texels than the 6 maps required by cubic mapping. Therefore a spherical map has less information to work with when representing the environment. Beyond texel count, the assumed geometric configuration of the indexing scheme can also cause distortion. Spherical and paraboloid maps both have significantly less texture information at their poles. Cubic mapping does not have a specific pole or favored alignment and so has the least distortion of all the mappings discussed here.

Now that the general idea and concepts related to environment mapping have been introduced, the three environment mapping techniques will be discussed in more detail. For each type of mapping, the indexing method, the features and problems, and images from a simulator will be presented.

---

## Spherical Environment Mapping

Spherical environment mapping is currently the most popular form of environment mapping implemented in hardware. It uses a single texture map to represent the entire environment. Since it is applying a sphere onto a plane, there is some distortion involved in representing the environment map. A typical spherical environment map can be seen in Figure 7.

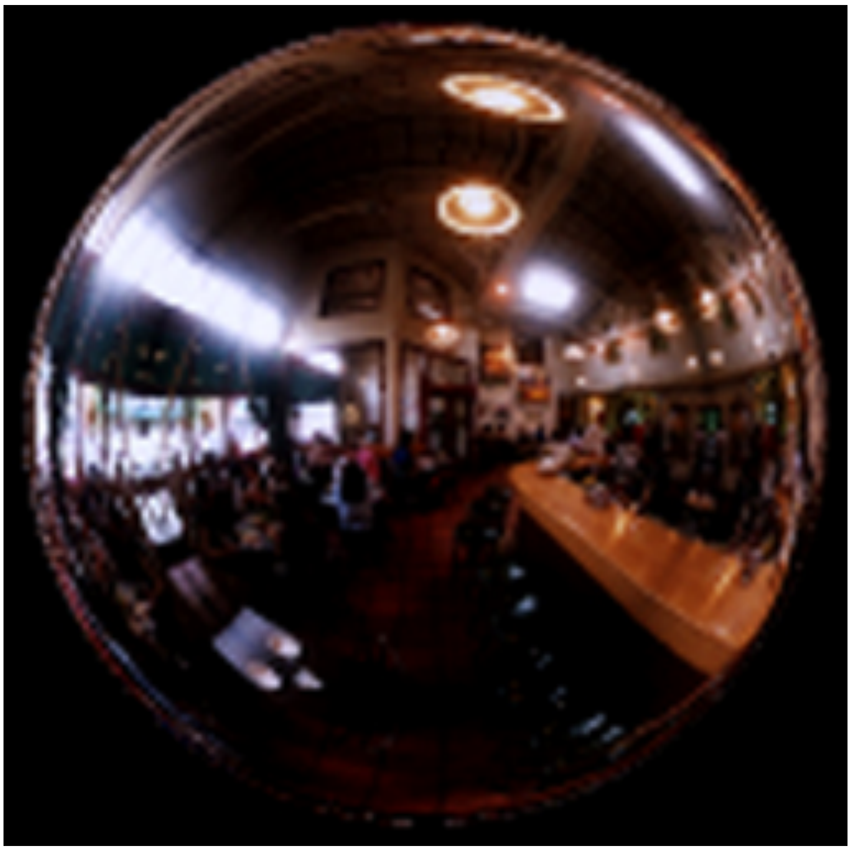


*Figure 7: A typical spherical environment map.*

The map is similar to the image on a Christmas ornament in a room, and this is the intended result when using sphere mapping. (In fact, one of the earliest environmemt maps was created using a Christmas ornament.) However, a sphere map considers the sphere to be infinitely small so that the image of the reflected rays has an identical starting point. This allows indexing based on the reflected ray to be consistent no matter the direction of the reflected ray. The viewer is also assumed to be at infinity when creating the map so that all the incoming rays will be parallel. Hence, the contents of the sphere map are based on a particular view (and are fixed once the texture is generated). Figure 8 is helpful in understanding how the sphere map is derived.

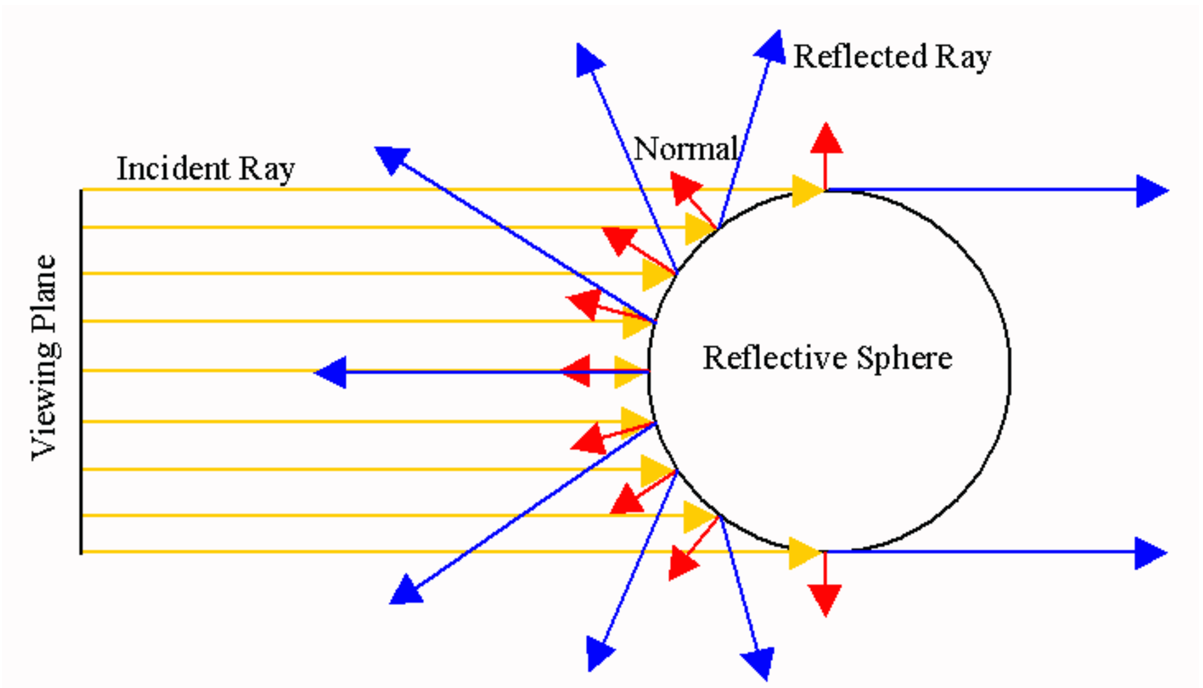
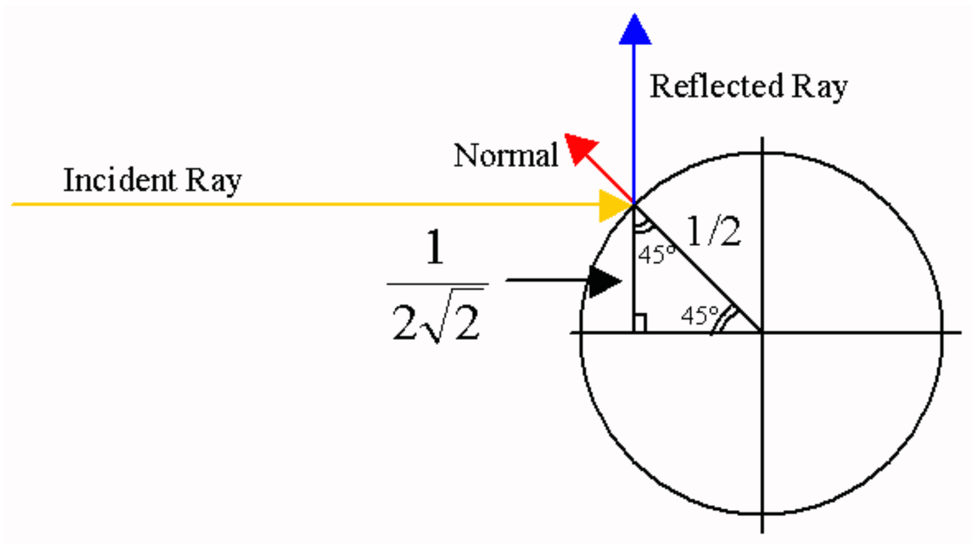


*Figure 8: The geometric configuration of a sphere map. Red represents normals, blue represents the reflected ray, and yellow represents incident rays. The sphere map stores the color seen by the reflected rays at the point on the sphere. (Adapted from the Siggraph 98 Advanced OpenGL Rendering Course Notes, p119)*

The reflective sphere in Figure 8 is meant to be extremely small (negligible radius) so that a sphere of directions is represented (starting at the same center) in the sphere map. The edge of the sphere map shows values for reflected vectors which are all in the same direction about the same point. Therefore, the pixels in the outermost ring should receive the same color.

## Properties of Sphere Maps

The sphere map contains information about the front and back of the environment about a point. Calculating where this crossover occurs in helpful in understand how much information is stored in which parts of the map, and therefore gives an idea of what resolution map is needed to represent detailed environments. The cross over point occurs when the reflected vector is perpendicular to the viewing plane as can be seen in Figure 9. Since the texture coordinates representing the sphere map range from (0,0) to (1,1), the sphere will have an assumed radius of 1/2.



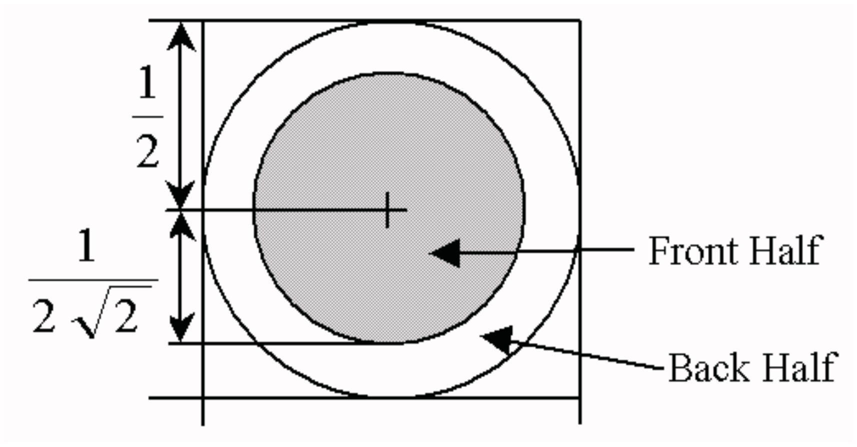


*Figure 9 and 10: The point of perpedicular reflection occurs at 1/(2*Ö2) = 0.35355339059327 on the environment map. (Adapted from here.)*

Using the previous calculations, some facts abou the sphere map can be derived. Since the sphere map is a circle embedded in a square, not all of the map contains useful information. In fact, 1*1 - p*$(1/2)^2$ = 1-p/4 = 21.46% of the map is wasted. p*$(1/(2*Ö2))^2$ = p/8 = 39.2699% of the map is devoted to the forward facing half of the environment while 1-((1-p/4) + p/8) = p/8 = 39.2699% is devoted to the back half of the environment. So the environment is represented equally front to back. However, this hides the fact that a pixel step on the front facing portion of the map subtends a much smaller portion of the world than a pixel near the edge. To make this concept more concrete, let us look at the specific example of a 512 pixels by 512 pixel map. Each pixel is 1/512 by 1/512 in width and height in texture space. To make the example a little easier to understand (and to calculate), the z coordinate will only be analyzed. A full analysis would calculate the projected pyramid from the origin of the sphere to the pixel image in all three dimensions. The area subtended by a pixel near the center and one near the edge will be examined. Figure 11 represents the situation with an 8 pixel map and clearly shows the difference in area represented by the center-most pixel and a pixel near the edge.

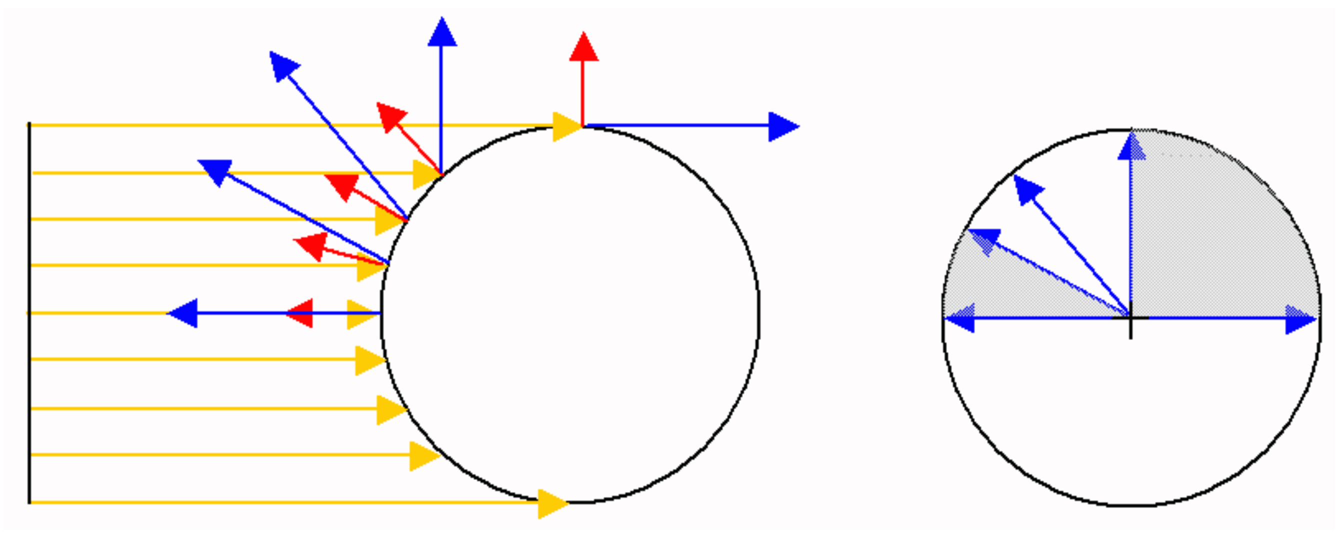
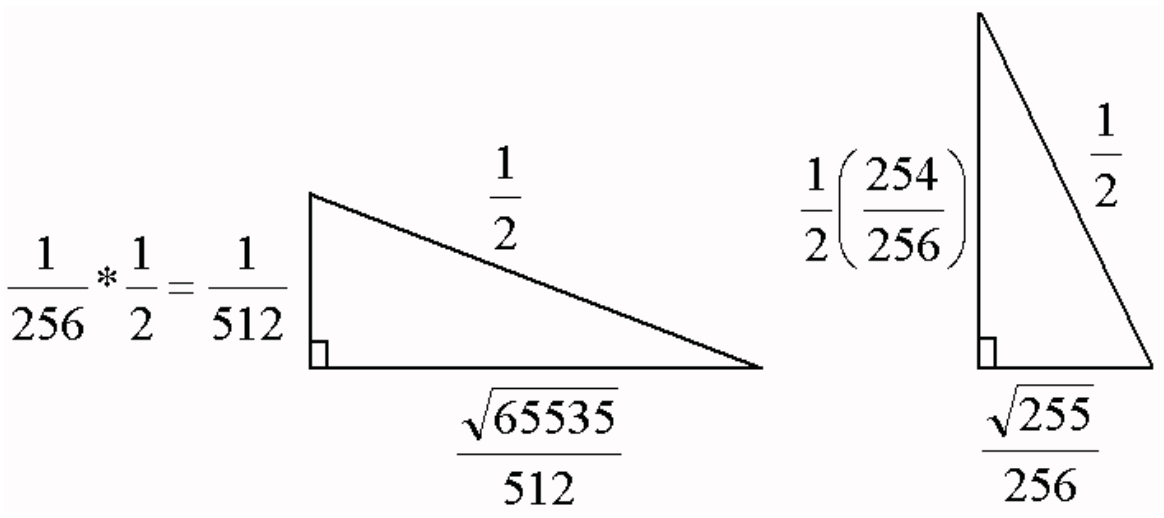


*Figure 11: With 8 pixels, there is a large difference between the area subtended by the inner and outer pixels in the map.*

For 512 x 512 pixels, three pixels will be examined, the pixel from 0 (considered to be the center) to +1 and the pixel from +254 to +255 and the pixel from +255 to +256 (the outermost). The set up is basic trigonometry and solving of lengths and angles. Figure 12 shows some of the triangles involved.



$\frac{1}{256} * \frac{1}{2} = \frac{1}{512}$
$\frac{1}{2}$
$\frac{\sqrt{65535}}{512}$
$\frac{1}{2}\left(\frac{254}{256}\right)$
$\frac{1}{2}$
$\frac{\sqrt{255}}{256}$

*Figure 12: Each pixel takes up 1/512 of the image in a single direction, in the positive direction one pixel has a coordinate of 1/256. In texture coordinates that ranges from +1/2 to -1/2 from the center, 1/256 has a coordinate of (1/2)(1/256) = +1/512 in texture coordiantes if the center of the map is considered (0,0).*

In short, there are 3 angles to consider, the angle between the reflected vector betwen the 0 and 1, the angle between the reflected vector from 254 to 255, the angle between the reflected vectors for the 255 to 256. The incoming ray will be called **u** = (-1,0,0) for all the calculations. For 0, the reflected ray is right back at **u**, $\mathbf{r_0}$ = (1,0,0). For 1, (the top of the 1st pixel), normalized $\mathbf{n_1}$ = (65537*Sqrt(65539)/16777984, Sqrt(4295098365)/16777984, 0), so $\mathbf{r_1}$ = **u** - 2(**u** dot **n1**) **n1** = (0.99996948, 0.0078122, 0). Taking the angle between the vectors is $\mathbf{a_1} = \cos^{-1}(\mathbf{r_0}$ dot $\mathbf{r_1})$ = 0.44761°. For the angle between 254 and 255 (bottom and top of pixel 255), the normalized normal for 254 is $\mathbf{n_2}$ = (Sqrt(255)/128, 127/128, 0). Therefore normalized $\mathbf{r_2}$ = **u** - 2(**u** dot $\mathbf{n_2}$) $\mathbf{n_2}$ = (-0.968872, 0.2475619, 0). For 255, normalized $\mathbf{n_3}$ = (Sqrt(511)/512, 255/256, 0) then $\mathbf{r_3}$ = **u** - 2(**u** dot $\mathbf{n_3}$) $\mathbf{n_3}$ = (-0.984405, 0.1759141, 0). The angle for pixel 255 is $\mathbf{a_3} = \cos^{-1}(\mathbf{r_2}$ dot $\mathbf{r_3})$ = 4.2014°. For the top most pixel, the reflected vector, as mentioned before, is known to be $\mathbf{r_4}$ = (-1,0,0). $\mathbf{a_3} = \cos^{-1}(\mathbf{r_3}$ dot $\mathbf{r_4})$ = 10.13186°. The angle subtended by a pixel ranges by between 4.2/0.45 = 9.3 times to 10.13/0.45 = 22.5 times. The variation only get larger for more pixels because the center covers a smaller angle while the edge covers a larger one.

In general, for an p by p map and texture coordinate sphere of radius s, for a starting point y (<p/2) at the bottom of a pixel, the value is incremented by 1. To convert the point y into texture coordinates, the following formula is used (as per the example above), texture coordinate = $tc_1$ = s * (y/(p/2)). This means that the radius of the sphere in texture coordinates, usually 1/2 but called s1 here for generality, times the fraction from 0 to p/2 (assuming the center of the map is 0). The lower normal, n, for the pixel is just:

$$n_1 = \begin{bmatrix} \sqrt{s^2 - \left(s\frac{y}{p/2}\right)^2} \\ s\frac{y}{p/2} \\ 0 \end{bmatrix}$$

u is taken to be (-1, 0, 0) again. $r_1$ = u-2(u dot $n_1$)$n_1$ which evaluates to:

$$r_1 = \begin{bmatrix} 2\left(s^2 - \left(\frac{s\,y}{p/2}\right)^2\right) - 1 \\ \frac{4s\,y\sqrt{s^2 - \left(\frac{s\,y}{p/2}\right)^2}}{p} \\ 0 \end{bmatrix}$$

The formulas for the top of the pixel are very similar except that they use s (y+1) / (p/2). Now normalizing n1 and n2 as well as the result r1 and r2 and then plotting ArcCos[r1 dot r2] for a given s (=1/2), p (say 512), the angle subtended by each pixel can be graphed. The plot can be seen in Figure 13.

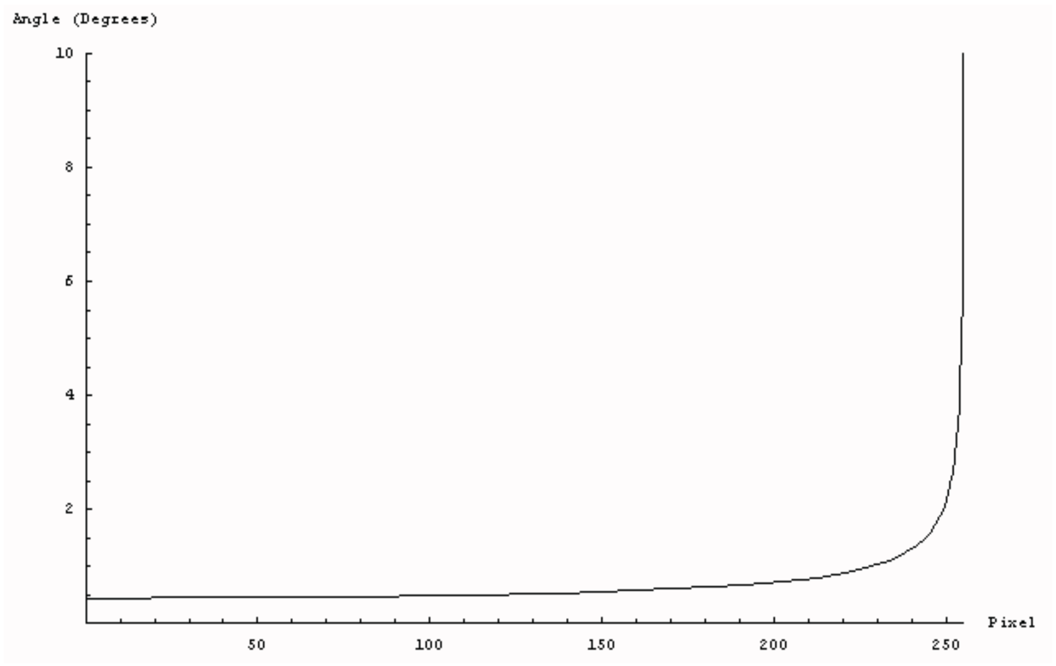


*Figure 13: A plot of the angle subtended by a pixel in a sphere map vs. the pixel number in the map.*

Figure 13 shows the singularity at the edge of the sphere map. It is important to keep in mind that this analysis only handled *one dimension* of the sphere map. As opposed to calculations radially from the center (middle) of the map), calculations across the arcs of the map (clockwise say) are also important. Just as the map has a singularity radially, there is a singularity about the center of the map when sampling along an arc. Since the arc length for a given angle increases away from the radius, the distortion is lower in that direction. A full

3D analysis with projected pyramids would provide a fuller picture of the errors in both dimensions for the sphere map. Figure 14 gives an intuitive feeling for the error along the arc of the sphere map.

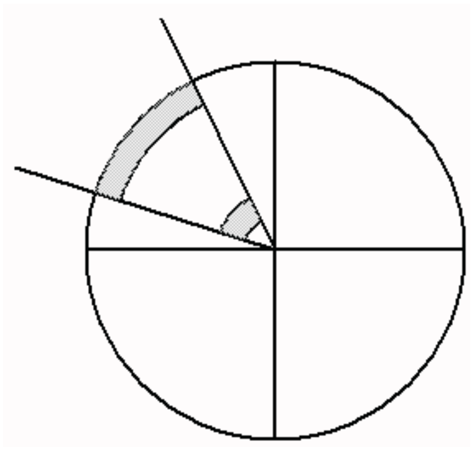


*Figure 14: For an angle subtending an arc along the map, there are less pixels toward the center and more toward the edge representing the environment.*

The overall lesson in sphere mapping is that when one tries to sample a nonlinear object (like a sphere) with a regular grid (like a texture map image) then there will be sampling problems. More practical considerations will now be discussed.

**<u>More Practical Considerations</u>**

Figure 10 does not label the origin of the sphere map and there is a bit of confusion about where the actual origin is for sphere mapping. The display of the sphere map behaves normally with the origin in the lower left corner. However, in OpenGL, applying the texture matrix to the sphere map (to rotate it say) works correctly if the origin is seemingly in the upper right. Figure 15 makes this explicit.

*Figure 15: Origin for sphere mapping in OpenGL vs. origin as seen by the texture matrix in OpenGL*

Another very important consideration in using sphere maps is the graphics hardware that processes the map. A sphere map is a *nonlinear* mapping. Two points in a sphere map are connected by an arc (except if then coincide radially) but graphics hardware uses a linear interpolation to connect texture coordinates. Therefore, towards the edges of the map, there can be a large misrepresentation of the environment beyond that inherent in the mapping. Figure 16 shows the difference between linear and nonlinear mappings on the sphere map.

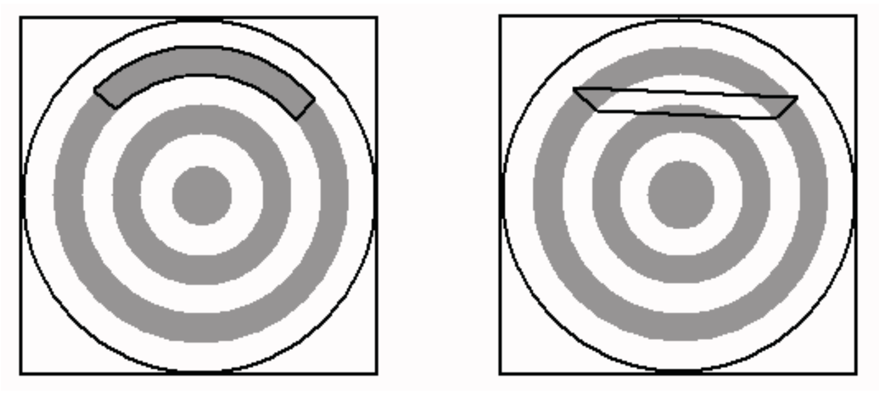


*Figure 16: The linear interpolation of texture coordiantes in graphics hardware provides incorrect texture information on a spherical environment map.*

Perhaps the spline rendering hardware already present in graphics systems could be used to determine points along the arcs for texture evaluation much using dynamic subdivision. In fact, since the arcs will always be of constant curvature, a dedicated piece of arc evaluation hardware could resolve the inaccuracies related to these types of nonlinear mappings. Alternatively, a linear table lookup into an array storing nonlinear values could also provide a means of getting more accurate texture evaluation.

## Indexing Scheme

A good deal of effort has been spent in trying to explain the properties of sphere mapping. However, the indexing method used by sphere mapping is an important aspect of its usefulness (or lack thereof depending on your point of view).

In nearly all cases, except perhaps NURBS modeling tools, a sphere map is applied to a triangular or quadritlateral mesh composed of vertices. In addition, the surface has normals stored at the vertices (and sometimes at the triangle centers but that is not considered here). Then the reflected vector is calculated from the camera's (or eye's) point of view to get the reflected vector coordinates. The configuration is illustrated in Figure 17.

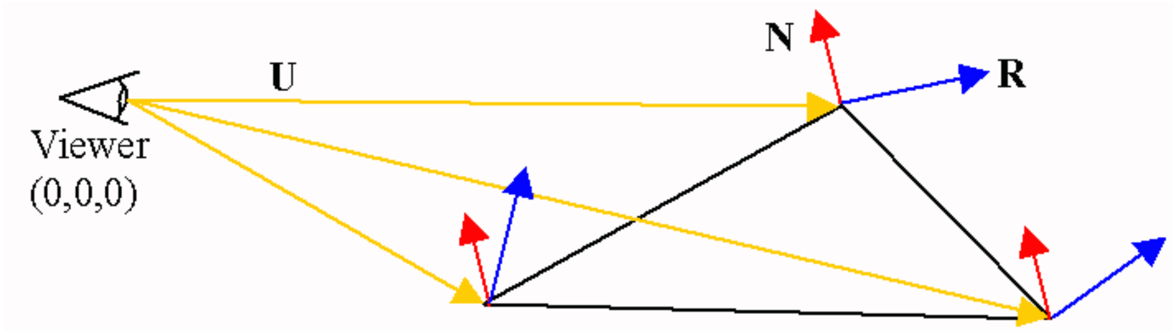


*Figure 17: Reflected Vectors are calculated in Eye Space. (Adapted from the SIGGRAPH '98 Advanced OpenGL Rendering Course Notes, p120)*

The polygon has coordinates in world space with the origin at the corner of a room (say). The indexing approach used by sphere mapping converts the normals and vertices from world space into *eye space*. Eye space puts the camera (or viewer) at (0,0,0) with the -z coordinate pointing from the eye and y defined to be up from the eye and x to the right. Eye space is still a 3D space though. With these new numbers defining the position vertices and orientation of the normals, the formula R = 2(N dot U)N - U is used to calculate the reflected vector in eye space. The formula for R is reversed from the previous formulas because the U vector is coming from the eye instead of away from the point of reflection (reversing the sense of the reflected vector equation). R has x,y, and z components denoted as $R_x$, $R_y$, and $R_z$. The (s,t) texture coordinates are computed with the following formulas.

$$\begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} \dfrac{r_x}{2\sqrt{r_x^2 + r_y^2 + (r_z+1)^2}} + \dfrac{1}{2} \\ \dfrac{r_y}{2\sqrt{r_x^2 + r_y^2 + (r_z+1)^2}} + \dfrac{1}{2} \end{pmatrix}$$

Please see this tech report describing the derivation of these formulas.

The most important aspect of this indexing is that it is dependent on the viewer's position relative to the object. In other words, it is *view dependent*. The eye is always considered the origin in the calculations and therefore all values are calculated relative to it. This means that the map moves *with the viewer*. A reflected vector pointing towards the viewer will always index the middle of the map (1/2, 1/2) even as the viewer moves around the object.

A good way to understand the difference between a view dependent and view independent map is to consider a sphere mapped sphere in a computer generated room. Consider the sphere map to be generated at one particular viewpoint of the room (much like the cafe sphere map at the beginning of this section). With a view dependent mapping (like a sphere map), as the user moves around the ball, the same image will present itself to the user at all times since it is mapping relative to the viewer's eye. With a view independent mapping, the ball will behave like a Christmas ornament, showing different parts of the environment in the ball with different viewing locations. Figure 18 will hopefully make this clearer.

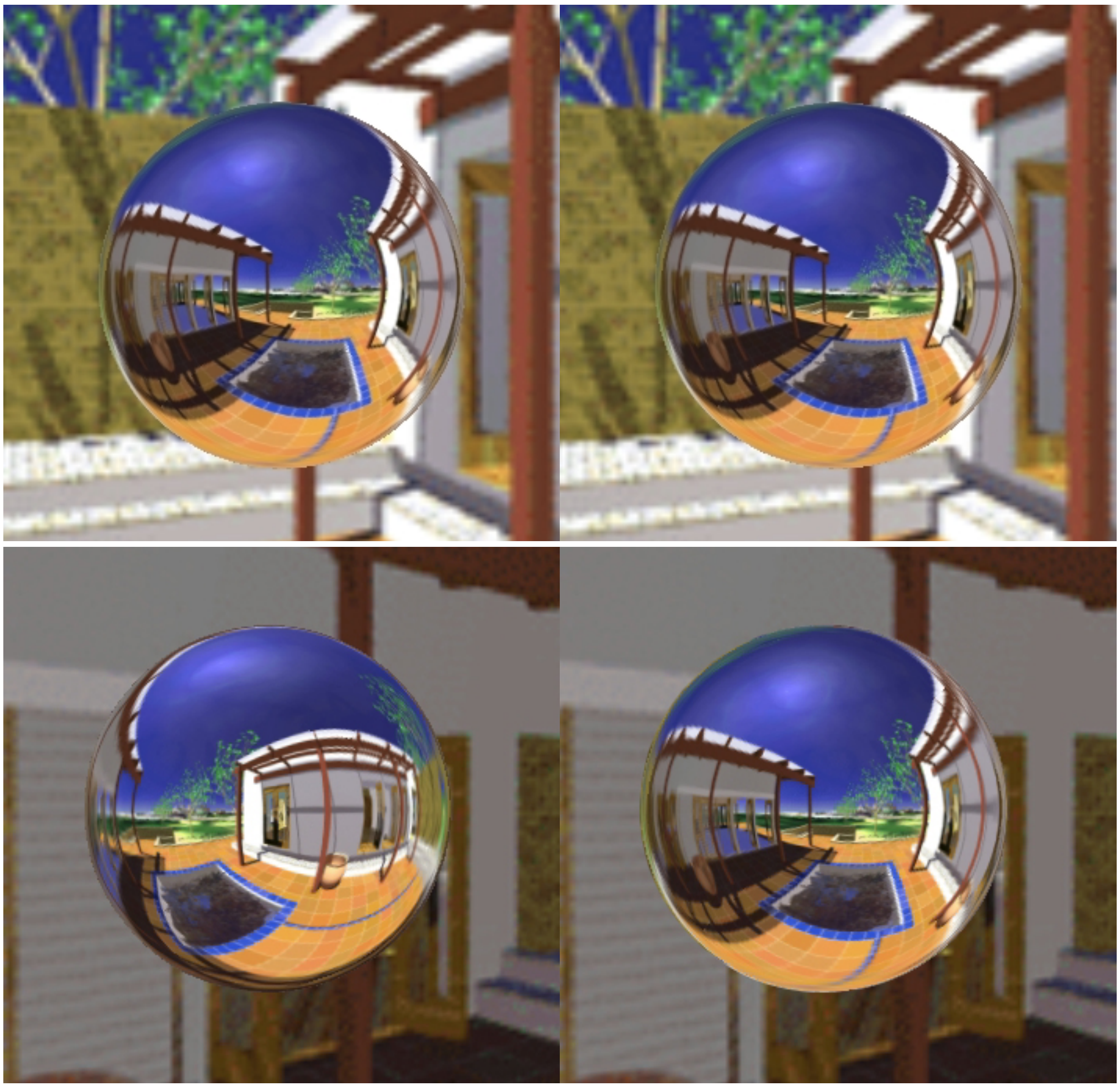


*Figure 18: A view independent map is applied to the ball on the left while a view dependent map is applied to the ball on the right. The maps agree at one particular view point in the top row (the view point at which the view dependent sphere map was created) but disagree as the viewer moves away from the point in the bottom row. The view dependent map will show the same map as long as the viewer is looking at the center of the ball. (Images adapted from nVidia's Ball demonstration program)*

At first, view dependence may seem like a major draw back to using sphere mapping. However, there are many effects seen in the world that a based purely on the viewer's viewpoint and sphere mapping is the only efficient method to simulate them. Effects such as colorgrams, CDs, Fresnel effects, laquer color shifts, and lighting highlights are all based relative to the viewer's position and viewing direction. One of the more common applications of sphere maps in this context is in the creation of Phong highlights on hardware that only supports sphere mapped environment mapping. The pixels in the map are looped over and for a given pixel, the corresponding reflected vector can be derived. With knowledge of the viewer's location (fixed in spherical environment maps) and the reflected vector, the normal can be derived. The normal and the viewing vector are fed into the Phong equation. The Phong equation returns an intensity which is stored in that particular place in the map. When the object is display, the map is being treated like a look up table into the Phong intensities. The value is multiplied by the objects inherent color in order to create a highlight. The drawback of using this technique though is that the highlight is relative to the viewer and so moves with the viewer as if the light were attached to the viewer. Figure 19 shows an example of a Phong environment map.

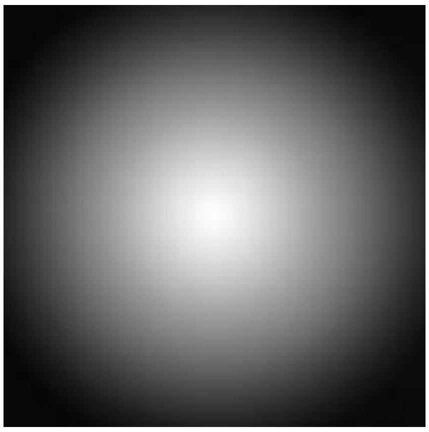


*Figure 19: A spherical environment mapped with intensity values derived from the Phong lighting model.*

This section touched on some of the more practical issues related to sphere mapping. The next section describes some of the assumptions of sphere mapping (an environment mapping in general) as well as describing the benefits and costs of using sphere mapping.

**Assumptions of Sphere Mapping (and environment mapping in general)**

It is somewhat difficult to glue all the assumptions into one paragraph. Therefore, the assumptions will be presented as a list with a small description afterwards.

● The environment is infinitely far away from the object being mapped. Since the environment map uses a single image to represent the environment, there is no parallax between elements as the viewer changes position or viewing orientation. This is the same effect as the environment being infinitely far away from the object (or far enough to cause no perspective change larger than a single pixel when moving around).

● The object is concave. Environment mapping only captures single rays from the image of a sphere. It records only one level of rays. Self reflection required at least one more level of rays to be cast to generate another sphere map (perhaps several for each level of reflection) from a convex portion of an object. It is possible to do this, but it is much slower and more cumbersome than normal environment mapping.

● There are no other environment mapped objects (or reflective surfaces) near the current one. This is similar to the one above. Interreflections are generally not represented in environment maps. Several passes and dynamic creation of environment maps are required to enable this effect.

● Sincle environment mapped polygons do not take up a large portion of the screen. Since the reflected vectors are used to index, an environment mapped polygon near the screen will have large differences in the direction of its reflected vectors. These reflected vector will have lare differences in their texture coordinates. Since linear interpolation is applied to the texture coordinates, in accurate indexing will be exaggerated by being so close.

● The object is finely tesselated. This allows the linear approximation of the nonlinear indexing for the environment map to be ameliorated. Tesselating the object finely is similar to making a fine linear piecewise approximation to a curve; the finer the tesselation, the better the approximation to real nonlinear texture indexing.

● The object has some degree of curvature. An environment map's distortion will become apparent if the environment map is applied to planar objects such as a mirror. An object with curved surfaces will alleviate the distortion inherent in the environment map image.

● The object is a perfect mirror. The reflected vector calculations for environment mapping all use the mirror direction formula for calculations.

**Problems with Sphere Mapping**

● There is a singularity in the mapping at the outer edge. And another one in the direction of the arcs toward the center.

● Relies on nonlinear indexing for proper texture representation.

● Distortion and sampling problems at grazing angles. Sampling rate is not equal in both dimensions towards the edge of the map. This affects filtering the texture as well.

● Does not use all the texture in the texture map. There is a waste of texture memory.

● View dependent. Environment mapped object should accurately reflect the outside world even if the viewer's changes position and orientation. Sphere mapping is based solely on the viewer position and orientation.

**Benefits of Sphere Mapping**

● There is a single texture image to use. This means that look ups are fast and the environment is represented compactly (at least the part of the texture that is used).

● Common to graphics hardware. Nearly all graphics cards implement this type of environment mapping.

● View dependent. This is listed in problems but certain effects make this a desirable mapping quality.

● Can be made view independent. Using OpenGL graphics extensions, the mapping can be made to sit still on an object while the viewer moves.

● Filtering occurs within a single map which is faster (in the worst case) than multiple texture environment maps.

**Sphere Map Simulator**

These are some screen shots of a test application used to apply environment mapping (sphere mapping in this case) to the simulation of an out door environment. Environment mapping is a good candidate to replace outside geometry with a single image since there is a large amount of view dependence based on head orientation and position when looking outside a window. Unfortunately, normal sphere mapping may not be enough to provide convincing interaction since the mapping is view dependent whereas a real window is view dependent based only on the viewer's position (the world does not move outside when a person rotates the head near a window or tilt when the viewer tilts his/her head). In lieu of these issues, these screen shots should give an idea of the odd texture coordinates that can be generated with sphere mapping. The application is applying a sphere map to a single quad. The normal is represented as a green line and the reflected vector as a red line.

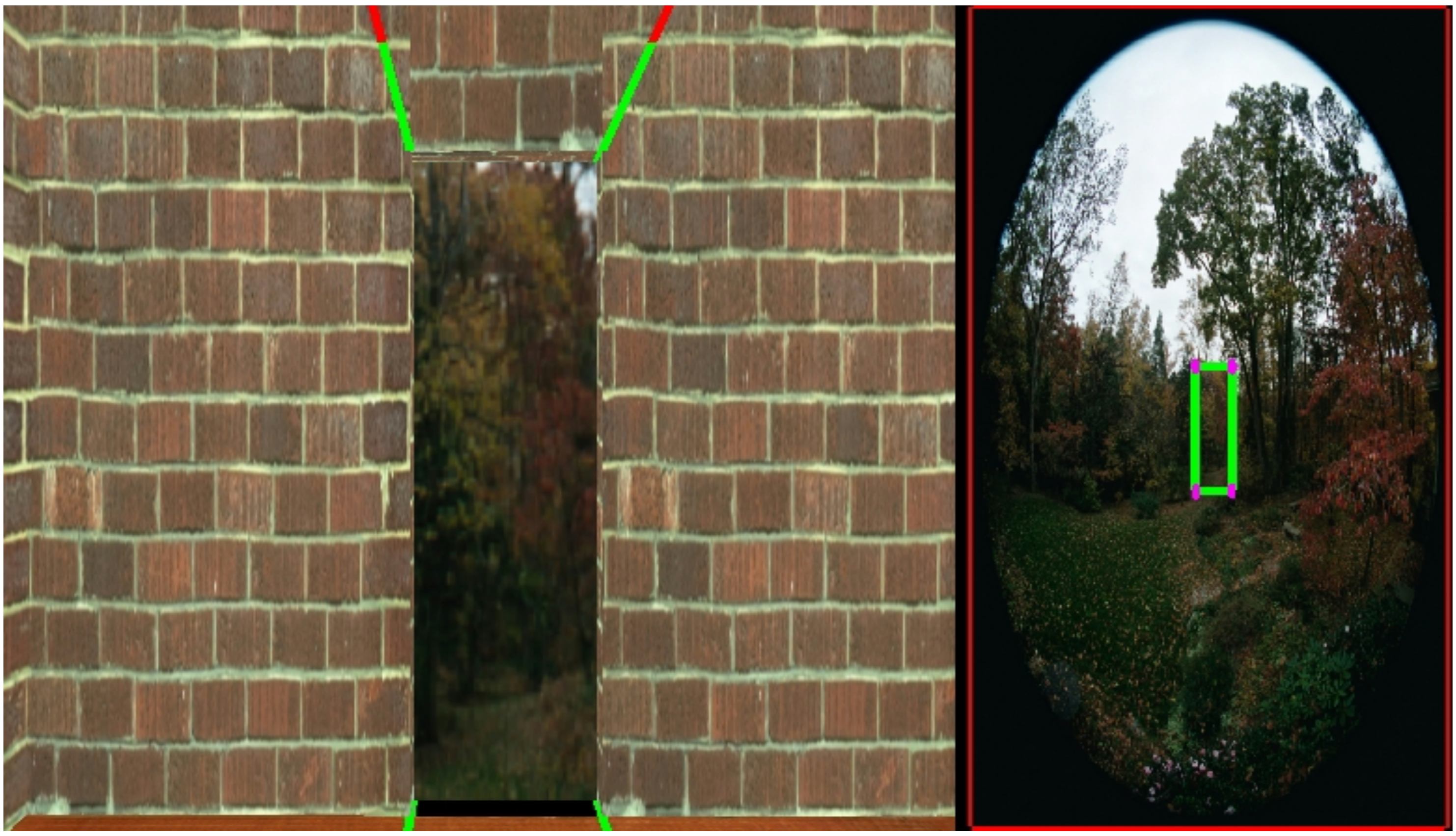


*Figure 20: This is a view looking straight at the door. Since the map indexes based on the reflected vector and the reflected vector is pointing towards the viewer, the map indexes the middle of the texture.*

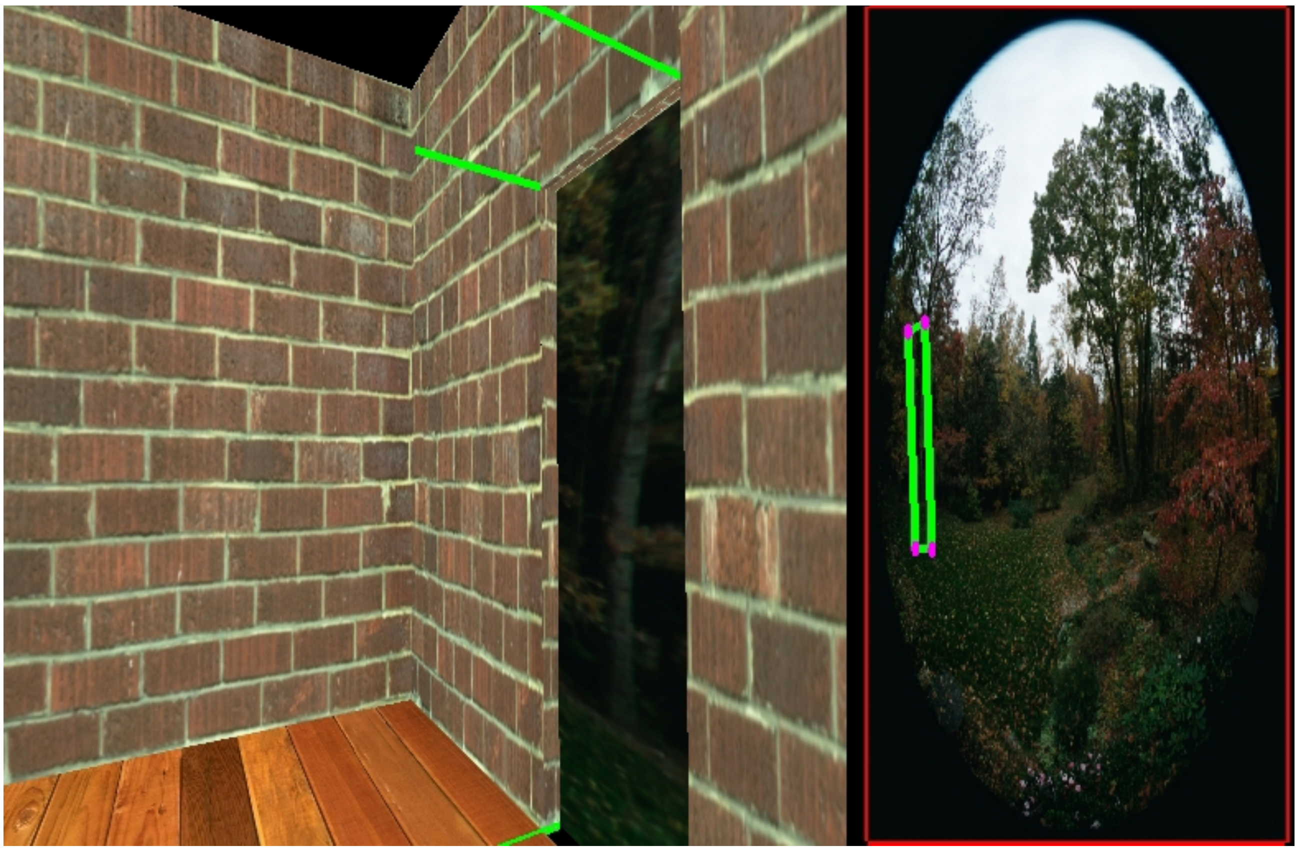


*Figure 21: At grazing angles, the linear interpolation between texture coordinates becomes apparent as can be seen by the bending tree as seen through the door. Ideally this tree should be shown at straight up and down and the indexing should be arcing along the map.*

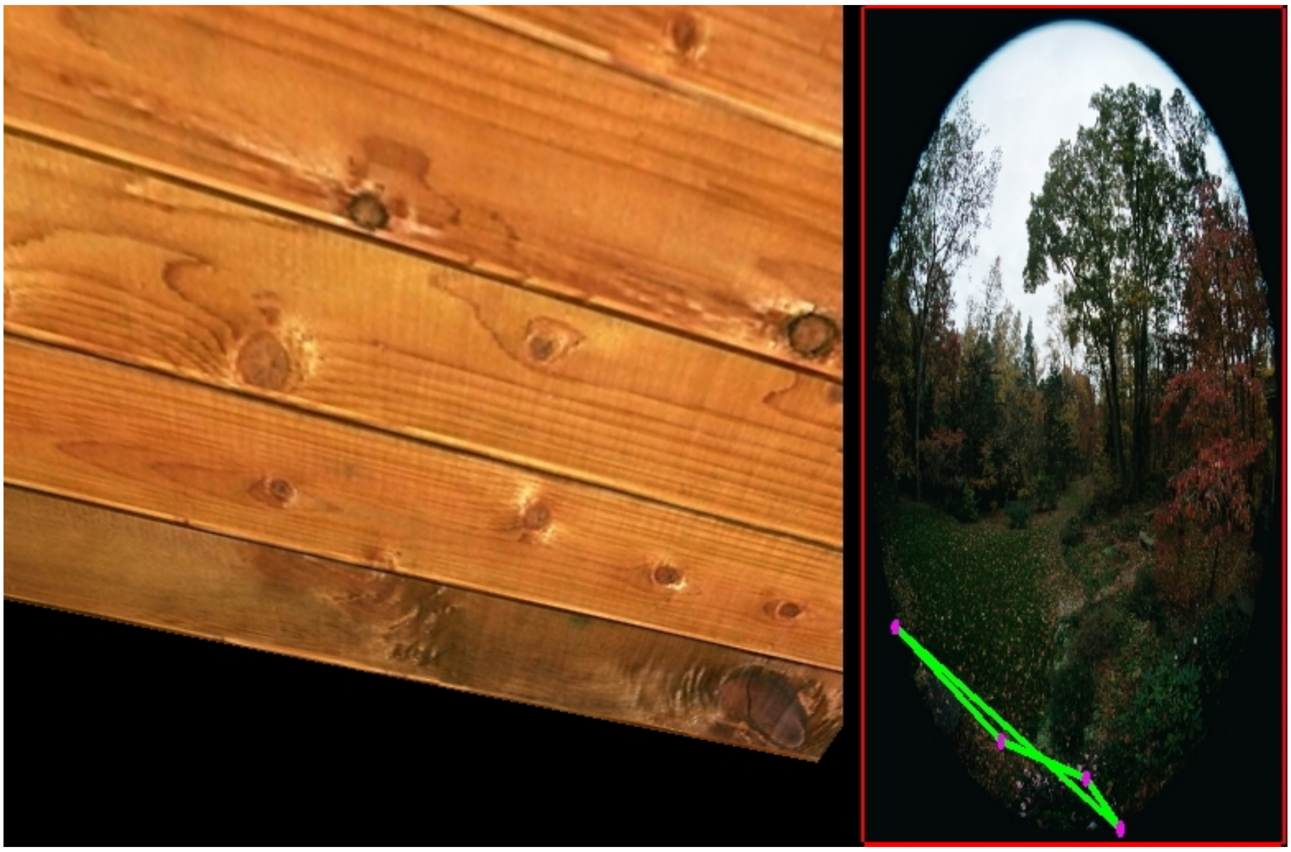


*Figure 22: This is a view looking down and away from the door. Although the door is not visible in this view, it is interesting to observe the indexing. In this case, sphere mapping the normals has resulted in a 'bow tie' effect on the texture coordinates. Since the view that generated this does not include the door, there is no harm done to the quality of the rendering.*

The sphere map simulator can be found here. It contains the Windows executable and 3 RGB images. The simulator requires OpenGL to run. The complete source code is not available but the (s,t) calculation can be found here.

**Conclusion**

This section has provided information on the properties of sphere maps and their use in enhancing visual quality in computer generated imagery.

## Cubic Environment Mapping

Cubic environment mapping is the oldest and most accurate of the environment mapping techniques covered here. Cube mapping was invented by Ned Greene in 1986. Although many graphics applications have made use of cube mapping for environment mapping (the most notable of which has been Renderman), the only graphics hardware that currently supports cubic environment mapping is nVidia's GeForce chip set. The cube map stores the environment as 6 textures. Each texture represents the view out of one face of the cube as seen from the center of the cube. A typical cube map can be seen in Figure 23. It is easy to imagine how a picture of the world in any direction can be made by folding the map into a cube.

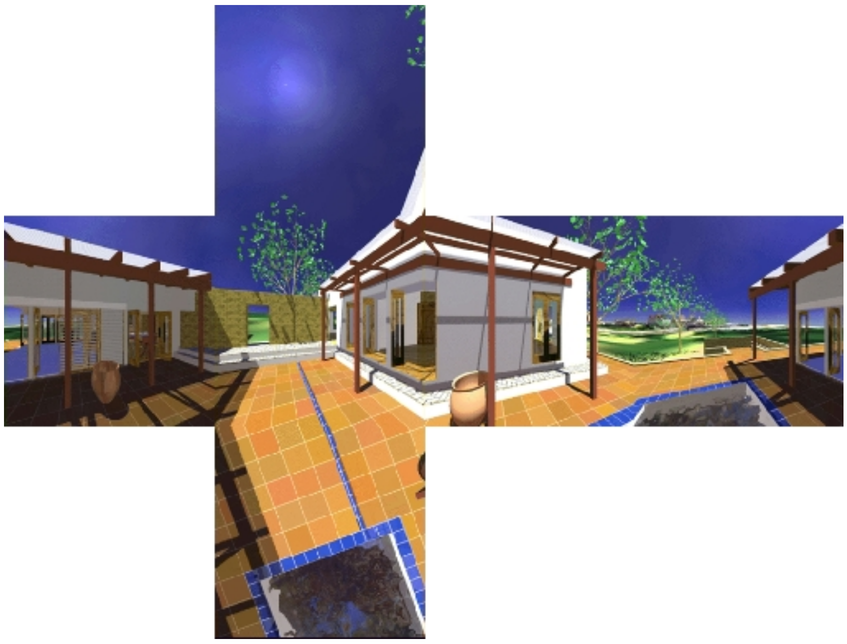


*Figure 23: A cube map environment texture. This texture is part of the Bubble demonstration program from nVidia.*

Although the map is represented as a single set of contiguous pixels, applications of cube mapping usually use six separate textures, one for each face. Similar to the sphere map and paraboloid map, cube maps are images at a single depth and so the environment is considered to be infinitely far away from the center of cube. The cube is also considered to be infinitely small so that reflected vectors used for indexing all use the same origin. The general set up for the cube map is depicted in Figure 24 and 25. Alternatively, Renderman and the hardware implementation of cube mapping from nVidia use a different cube set up.

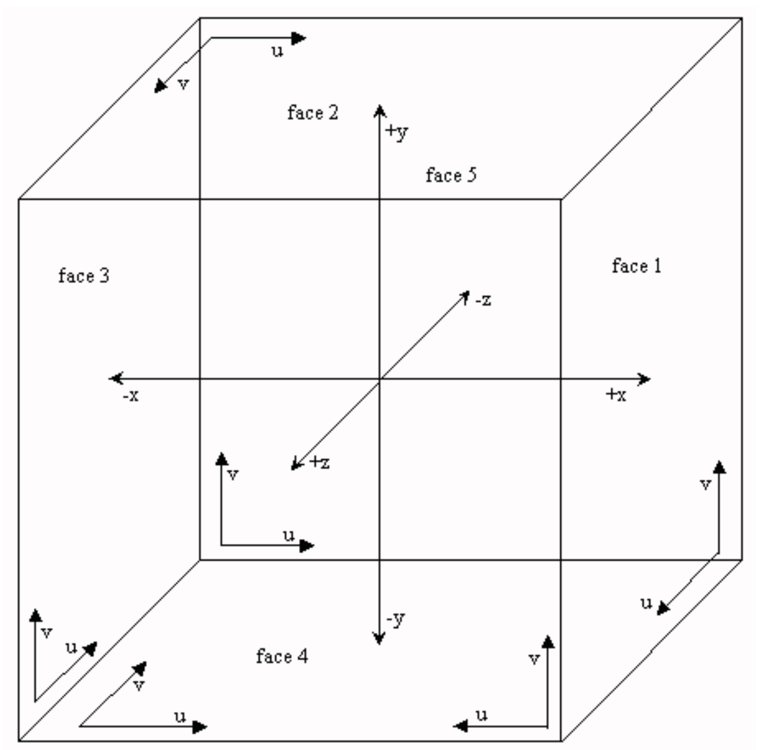


*Figure 24: The coordinate system for a cube map according to Alan Watt (Adapted from Advanced Rendering and Animation Techniques by Alan Watt. p 192).*

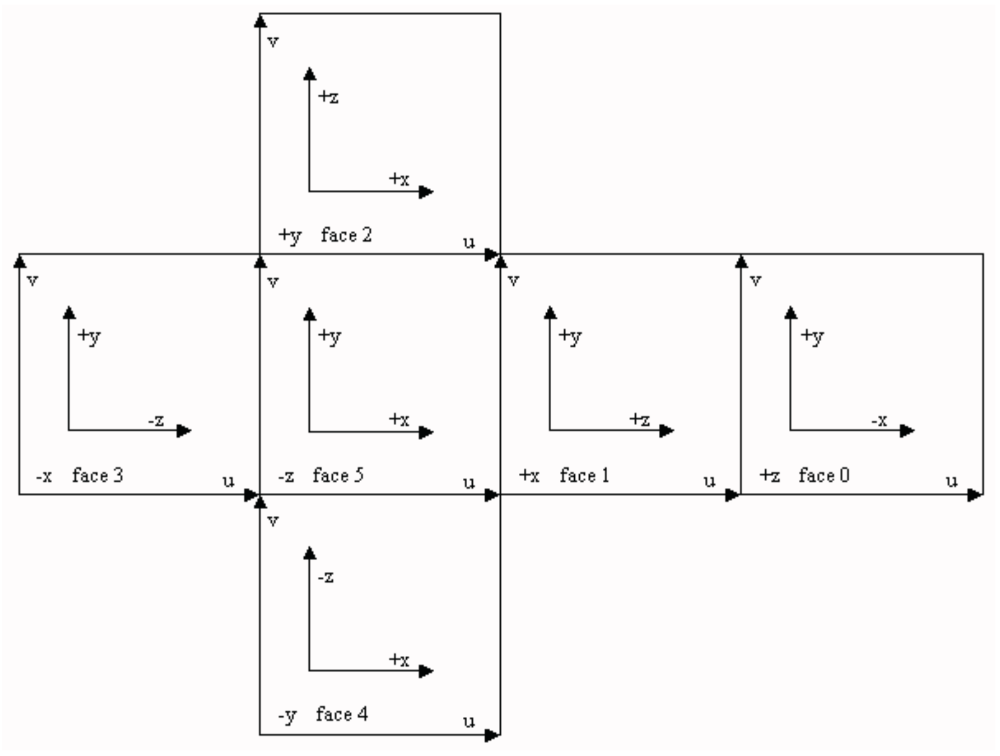


*Figure 25: The corresponding cube map.*

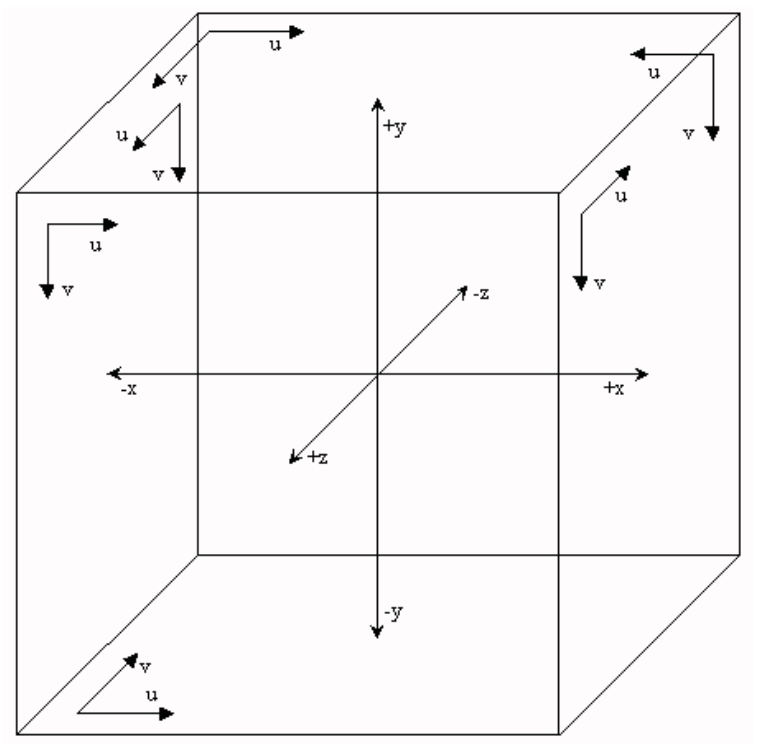


*Figure 26: The coordinate system for a cube map according to the Renderman standard.*

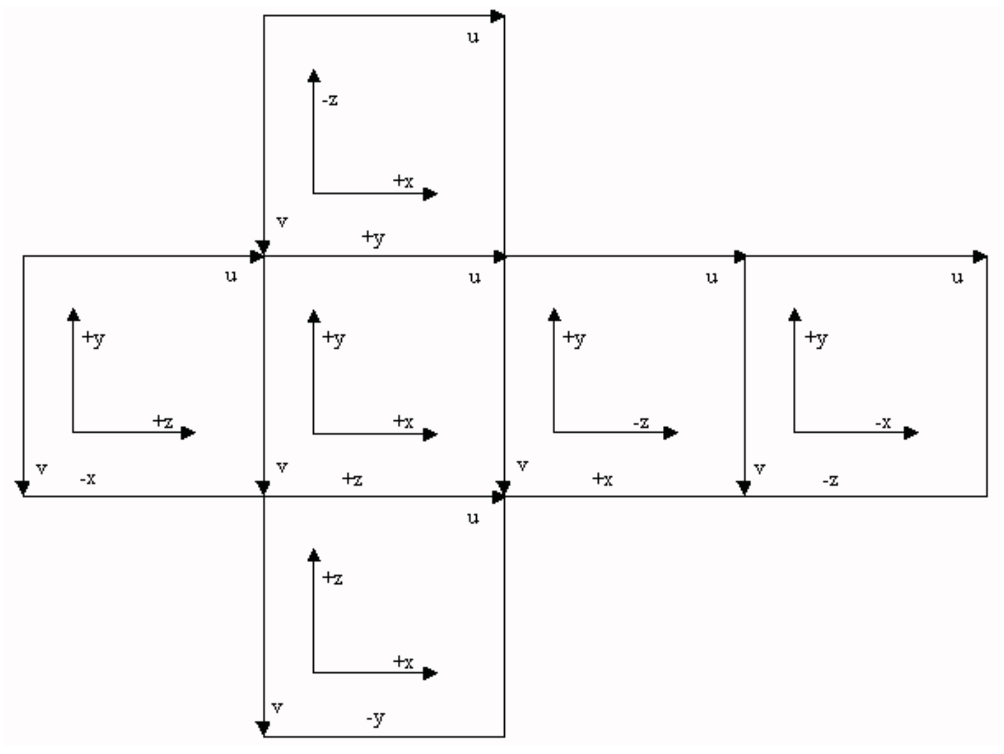


*Figure 27: The unfolded cube map for Figure 26.*

For Figure 26, the coordinate system (outside of the box) is a right handed coordinate system but inside the cube, the coordinate system is left handed. This choice was makes the unfolded maps look sensible as an image, allowing the middle four faces to form a panorama. The names of the faces are usually referred to as 'px' for +x = face 1 or 'nx' for -x = face 3 instead of the face numbers; the same goes for y and z.

**Properties of Cube Maps**

The cube map contains information about the front and back of the environment about a point. Since the map is cubic, analysis is a little simpler. The cross over between the front and back of the map occurs halfway in the cube. 50% is for the front of the environment and 50% is for the back. 100% of the pixels are utilized since all the environment textures are square and contain images of the environment. Since the texture coordinates representing the faces of the cube range from (0,0) to (1,1), the cube has a dimensions of 1/2 by 1/2 by 1/2.

Since the mapping from a sphere to a cube is not perfect, there is still some distortion in the representing the environment as a cube. Comparing the angle subtended by a pixel in the cube map can provide a means of comparing (roughly) the distortion within a cube map itself and the distortion between sphere mapping and cube mapping. Similar to the sphere map, different pixels in the environment map texture subtend different angles in a cube map. Figure 28 represents the situation with an 8 pixel map and shows the difference in area represented by the center-most pixel and a pixel near the corner.

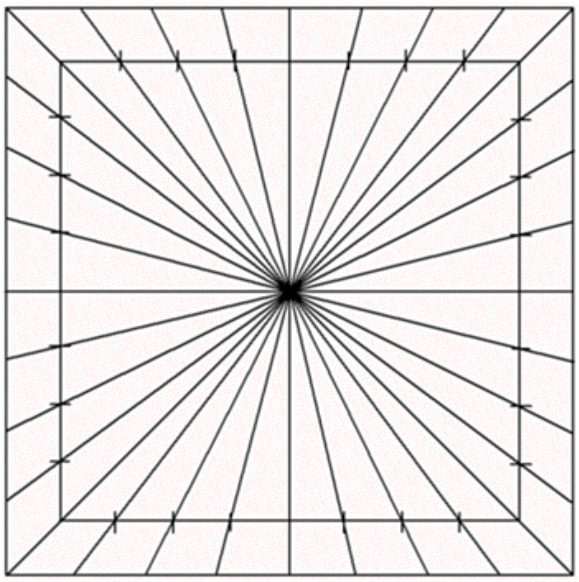


*Figure 28: The angles subtended per pixel for a slice of an 8x8x8 pixel cube map.*

For a 512x512 cube map, the angles are easier to calculate than with the sphere map. The following triangles in Figure 29 show the angle calculations involved.

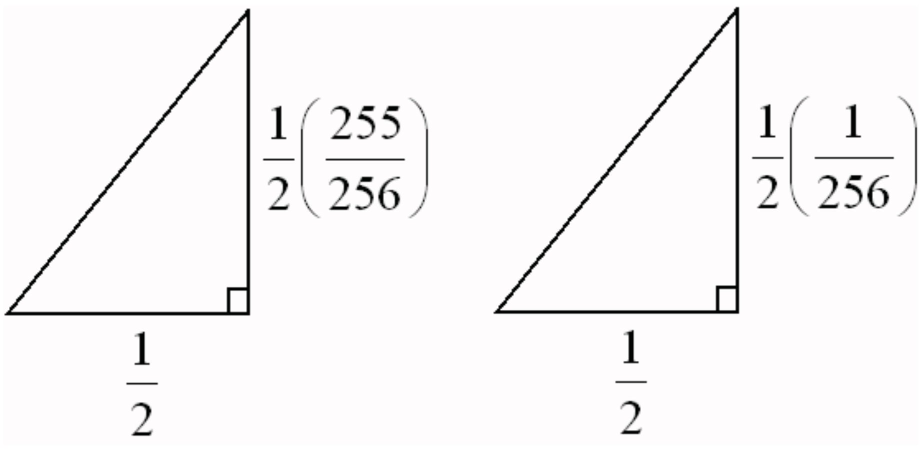

$\frac{1}{2}\left(\frac{255}{256}\right)$
$\frac{1}{2}$
$\frac{1}{2}\left(\frac{1}{256}\right)$
$\frac{1}{2}$

*Figure 29: The triangles used for angle calculations for pixels in the a cube map.*

Then angle from the bottom of pixel 1 to the top of pixel 1 = ArcTan((1/2)*(1/256)/(1/2)) = 0.2238105°. The outermost pixel has a top vector of 45 while the vector at the bottom of the pixel has and angle = ArcTan((1/2)*(255/256)/(1/2)) = 0.111212°. Therefore, the representation of the environment changes by at most 0.2238105°/0.111212° = 1.996 in this case. The origin was assume to be in the middle of the map for this analysis.

For a nxnxn map indexing a pixel y (£ n/2), the change in angle can be represented by the following formula ArcTan(y/n) - ArcTan((y-1)/n). Plotting this for a face of the cube results in the graph in Figure 30 shows this relation.

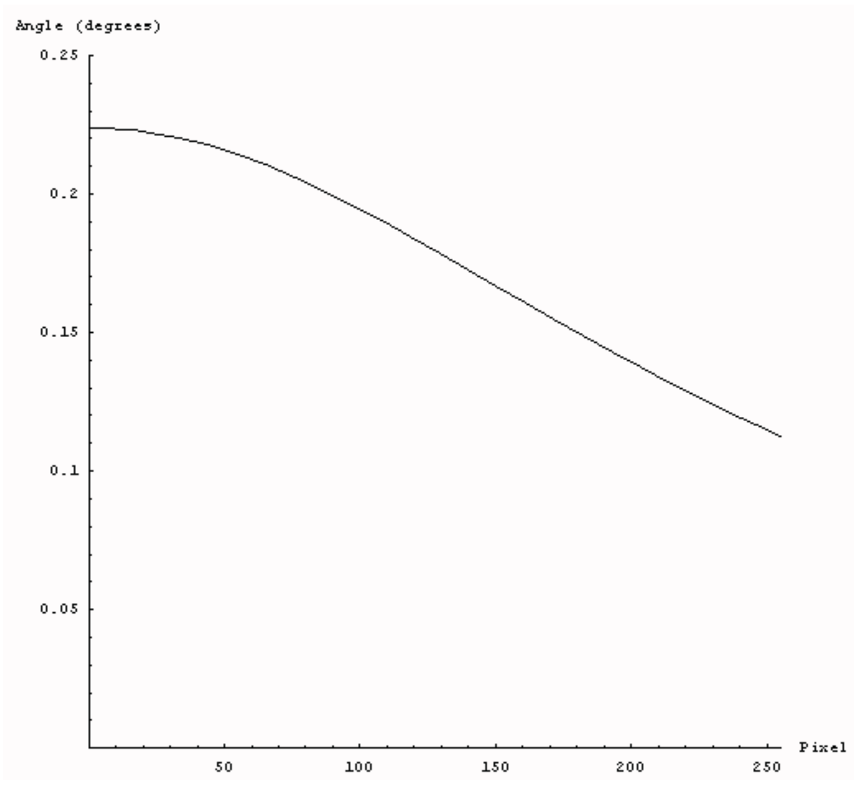
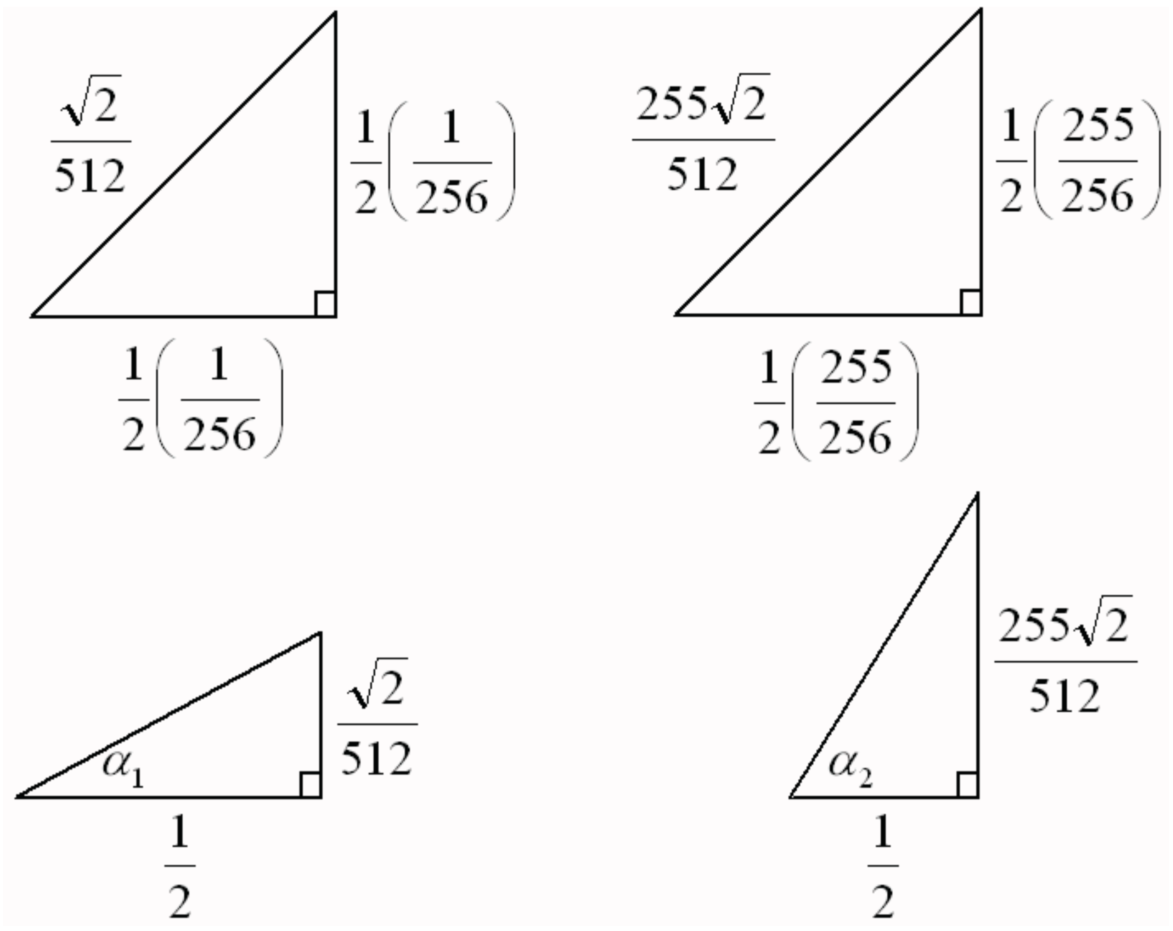


*Figure 30: The angle subtended per pixel in the face of a cube map decreases near the edges.*

The graph in Figure 30 is completely different from the graph for the sphere map (Figure 13). The difference in error is due to how the map is derived. Sphere mapping derived it's large growth of angle subtended per pixel because of the geometry of the method driving the derivation of the map. The curvature of the sphere when combined with the movement of the reflected ray and projection of the sphere onto a single map all combine against good angle characteristics. Cube mapping, however, uses the opposite approach (and no curvature) to reduce the effects. Since the ratio of the minimum angle to the maximum angle is one way to characterise the distortion in the mapping, perhaps there is another method which will try to make the minimum to maximum path even smoother and lower. The dual paraboloid map, discussed in the last section of this project, fills in part of the gap between cubic and spherical mappings.

Although the analysis above seems rather straightforward, there is still an issue of pixel size. Pixels are usually square (and are assumed to be here), so the angle subtended by looking along an axis will be different than looking along a diagonal. The math is not so different in this case, merely a Ö2 factor. The sides of the pixels are 1/(n/2) since the origin of the map is considered to be in the center of the map. For a 512x512 map, the triangles are as in Figure 31.



$\frac{\sqrt{2}}{512}$
$\frac{1}{2}\left(\frac{1}{256}\right)$
$\frac{1}{2}\left(\frac{1}{256}\right)$
$\frac{255\sqrt{2}}{512}$
$\frac{1}{2}\left(\frac{255}{256}\right)$
$\frac{1}{2}\left(\frac{255}{256}\right)$
$\alpha_1$
$\frac{\sqrt{2}}{512}$
$\frac{1}{2}$
$\alpha_2$
$\frac{255\sqrt{2}}{512}$
$\frac{1}{2}$

*Figure 31: Some of the triangles used in calculating the angles subtended by pixels along the diagonal of a cube map.*

Each side of a pixel is (1/2)(1/256) so the diagonal is Ö2/512. The cube has a distance of 1/2 from the center of the cube to the center of a face. Therefore, combining the size of the cube with the length of the diagonal of a pixel, $a_1$ = ArcTan((Ö2/512)/(1/2)) = 0.316514235°. For the last pixel in the diagonal, the coordinates (1/2)(255/256) are used, making the diagonal 255Ö2/512. Since the cube is still unit, $a_2$ = ArcTan((255Ö2/512)/(1/2)) = 54.629829147°. By similar reasoning, the very edge of the diagonal (the corner) has an angle of 54.7356103° making the last pixel subtend 0.105781170222°. The ratio is then 0.3165/0.10578 = 3.3243291 which indeed is worse than along the diagonal. The reason for this is that the diagonals are longer than along an axis and the pixel diagonals are larger than the pixels along an axis. Another analysis of cube mapping (mentioned in the Heidrich and Siedel paper View Independent Environment Mapping) measures the distortion at 3Ö3 = 5.2 but this was calculated in two dimensions simultaneously instead of the single dimension described here.

Earlier the cube map was compared to the sphere map. However, the comparison is not necessarily fair. Since the ratio of the angle subtended by a pixel depends on the number of pixels in the map, it is not the same to measure a 512x512 sphere map against a 512x512 by six map cube map. If the comparisons are make equal in space as a 512x512 sphere map, then the cube map face size reduces to 209.023 => 209 pixels per side of the map. Making these changes to the cube map calculations along the diagonal above, the ratio then becomes 0.3876901/0.1296452 = 2.9904, still less than the sphere map. Even if the equivalent is change to include only the useful pixels in the sphere map (only ~78% of the pixels in a sphere map contain environment information), the ratio for the cube map is 0.4379831/0.1465249 = 2.98914 which is still better than sphere mapping by about a factor of 3.

## Indexing Scheme

The indexing scheme in cube mapping is simple compared to sphere and dual paraboloid mapping. Since sphere mapping has been discussed previously, the cube map indexing scheme will be compared with the sphere mapping scheme. The cube mapping method is view dependent much like sphere mapping. As the view changes, the contents displayed on the environment mapped object changes along with it but the calculations are always relative to the viewer's eye. A major difference between sphere mapping and cube mapping is that cube mapping is performed per fragment (or per pixel) as opposed to per vertex. A fragment, as defined by the *OpenGL Programming Guide*, is "a grid square along with its associated values of color, z (depth), and texture coordinates." The grid square ends up being a pixel on the screen. In sphere mapping, the reflected vector is calculated per vertex. Each reflected vector per polygon (usually 3 or 4) results in a texture coordinate on the sphere map. The coordinates are then linearly interpolated and that section of the sphere map is applied to the polygon. Cube mapping, on the other hand, performs a look up into the cube map per fragment of a environment mapped object. Each pixel of the environment mapped object has a corresponding reflected vector. This reflected vector is direction is then intersected with the cube map in 3D and the corresponding texture map and texel are calculated.

There are two different methods for indexing in cube mapping that will be described here. The first is implemented in cube mapping hardware while the second is based on projected pyramids through pixels.

Cube mapping in hardware begins the indexing process much like sphere mapping. The reflected vector is calculated per vertex of a polygon and is calculated relative to the eye. The reflected vector calculated in this way is in eye space and has components ($r_x$, $r_y$, $r_z$). This handles the corners of the polygon but leaves the middle undefined. The middle of the polygon is filled in by performing a linear interpolation of the reflected vector across the polygon in increments of pixel widths. The interpolation can be performed incrementally from one fragment to the next. The inerpolated reflected vectors can be seen in Figure 32.

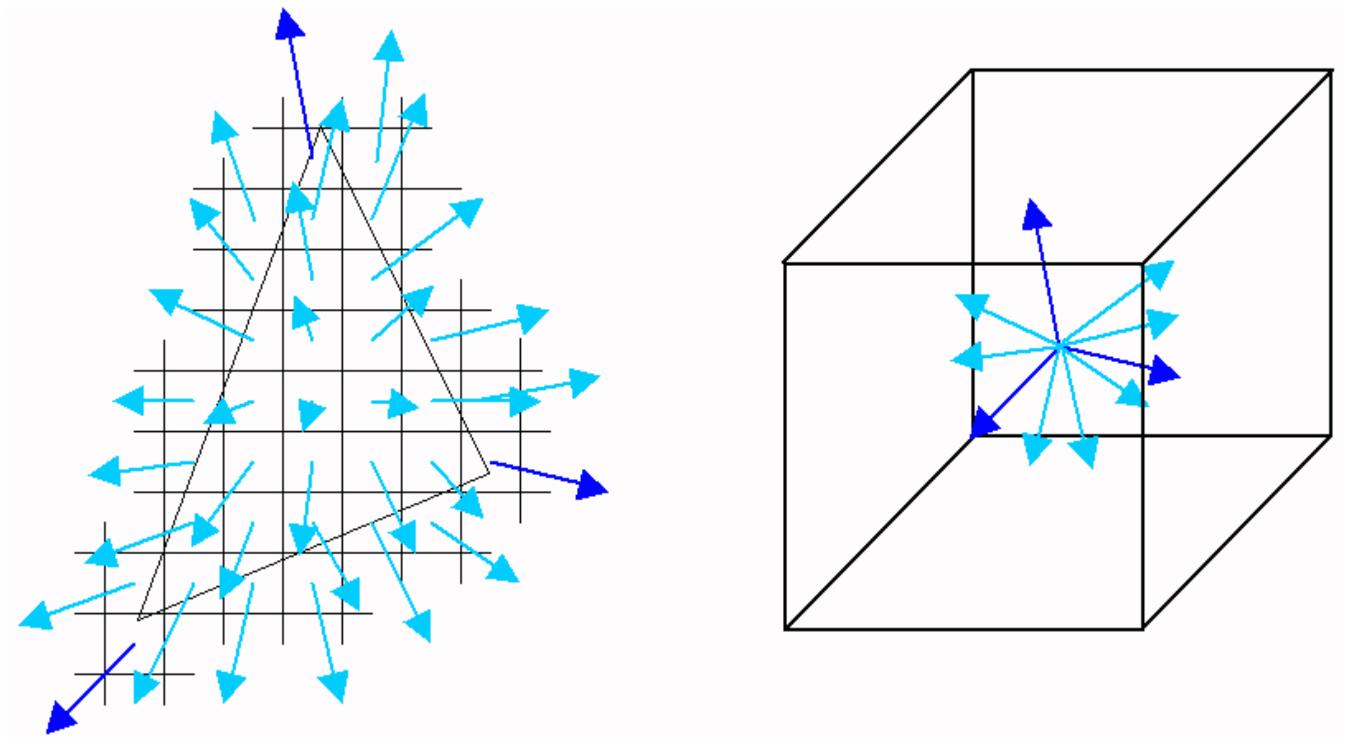


*Figure 32: Fragments of a polygon along with their reflected vectors are on the left. The representation of those reflected vectors inside the cube map. (Dark blue are calculated, light blue interpolated).*

Since cube mapping is three dimensional, the map uses 3 texture coordinates acting as a reflected vector. That is ($r_x$, $r_y$, $r_z$) become (s,t,r) texture coordinates. Once a fragment has been assigned a reflected vector, the texel in the cube map must be retrieved to give the fragment its color (or part of its color if it is blended with another texture or shaded). The first step of the indexing examines the reflected vector to see which component is largest, the x, y, or z component. This dictates which face is examined. Then once the face has been determined, some triangle calculation can be used to determine where on the face the direction hits.

The standard method of doing this, based on a ($r_x$, $r_y$, $r_z$) vector is to first look up some values in a table. These values are used for the (s,t) texture calulation. Figure 33 lists the table.

| major axis direction | sc | tc | ma |
|---|---|---|---|
| +rx | -rz | -ry | rx |
| -rx | +rz | -ry | rx |
| +ry | +rx | +rz | ry |
| -ry | +rx | -rz | ry |
| +rz | +rx | -ry | rz |
| -rz | -rx | -ry | rz |

*Figure 33: The assigning of variables in cube map indexing based upon the largest component of the reflected vector. (From the texture_cube_map OpenGL extension spec listed here)*

One `sc`, `tc`, and `ma` has been assigned values, (s,t) coordinates for that face can be calculated with the following formulas.

$$s = \frac{\frac{sc}{|ma|} + 1}{2}$$

$$t = \frac{\frac{tc}{|ma|} + 1}{2}$$

To understand how the table and formulas work together, the (s,t) calculation of the reflected vector (0.25, 0.25, 0.5) will be considered. Since the indexing is based on relative size of each coordinate, the vector (2,2,4) would also produce the same (s,t) coordinates. In other words, the reflected vector does not need to be normalized, only its direction matters. Since 0.5 is the largest coordinate, the table indicates the +rz row should be used. In this row, the s coordinate uses +rx. This is because the growth of positive x agrees with the growth of positive u on the +rz face (as can be seen in Figure 26). The growth of y corresponds to a decrease in v on the +rz face. Therefore, -ry is used to change the sense of growth of y in the +rz face. This explains the signs in the `sc` and `tc` columns. The indexing within the face is based on the relative growth of each axis compared to the major axis. If they are the same, then the vector has a slope of 1 (or -1). The slope can range from -1 to 1 on a face. However, (s,t) indices range from 0 to 1. Therefore, the formula for s and t maps the relative growth of each coordinate against the major coordinate from -1 to 1 to 0 to 1 on a face. This results in the +1 in the numerator and division by 2 in the denominator.

A small piece of code to calculate all possible candidates for (s,t) coordinates is provided here.

There are still two issues related to the indexing method. The first is degeneracies in the indexing. The reflected vector may become (0,0,0) by interpolating between reflected vectors with opposite direction at the vertices of a polygon or perhaps due to degenerate normals. In either case, this can cause `ma` to become 0 which will leave s and t undefined. Hopefully this will only occur at a single fragment on a polygon and not affect the environment mapped object heavily. The second issue is the linear interpolation of the reflected vector. By linearly interpolating the reflected vector instead of recalculating it based on an eye vector and interpolated normal, errors are introduced. To obtain an intuitive understanding of the nature of this error, the amount of disagreement between the calculated and interpolated reflected vectors were calculated for a line ranging from -10 to +10 on the x axis. The viewer was placed at various positions 1 unit above the line and a dot product between the vectors generated with interpolation as opposed to calculation was derived. Figure 34 describes the example set up.

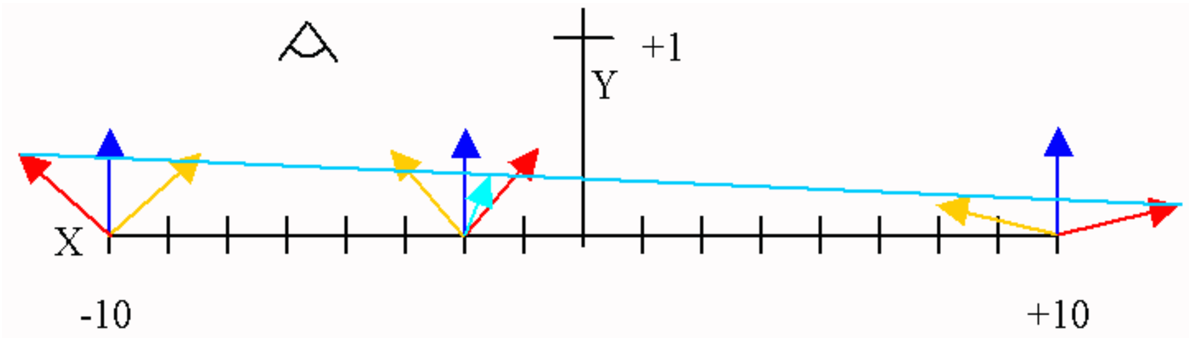


*Figure 34: The use of an interpolated reflected vecotr (light blue) as versus a calculated one (red). The error depends on the viewer's location.*

For a particular viewer location, the angle between the calculate reflected vector and the interpolated reflected vector can be calculated along the range -10 to 10. One such graph of angle difference can be seen in Figure 35. In Figure 35, the viewer has a location of (-8.3, 1, 0).

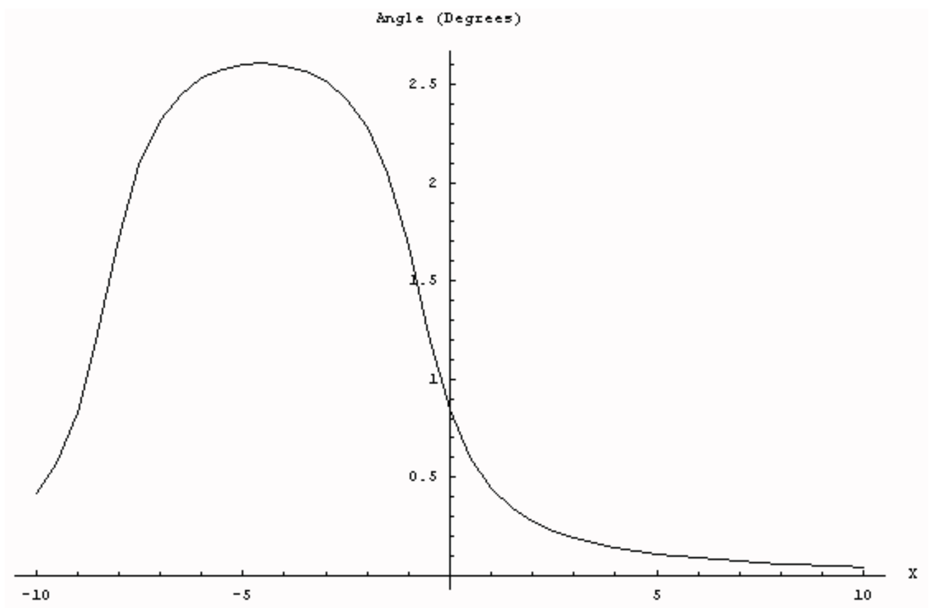


*Figure 35: The angle difference between a caculated reflected vector and an interpolated reflected vector. The maximum difference (~2.61°) occurs around X=-4.5. The angle difference is symmetric so X=+4.5 will produced a horizontally flipped version of the graph.*

Figure 35 examines the angular difference for a particular point. It is natural to ask what is the maximum error as the viewer changes location along the X axis. Figure 36 shows the result of graphing maximum error at a particular X location.

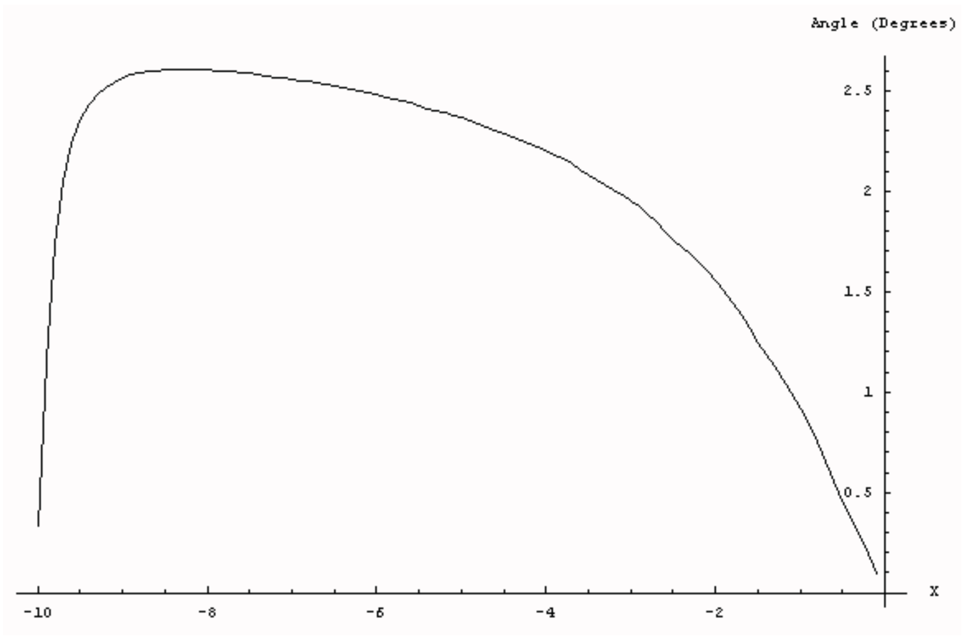


*Figure 36: Maximum error based on position along the X axis for a span from -10 to 10 with the viewpoint +1 unit in the Y axis. The graph is symmetric so only -10 to 0 is displayed.*

According to Figure 36, the maximum error at a particular view point occurs at X=+/-8.3 and the angular difference is 2.60751° which as described in Figure 35 occurs at X= +/-4.5. The *Mathematica* script which generated this plot can be found here.

This section has addressed one method of cube map indexing, the method used by current hardware. The next portion of the indexing section will discuss (more briefly) the indexing method used by Alan Watt in his description of cube mapping in the book *Advanced Rendering and Animation Techniques*.

## Alternate Indexing Method

The method described by Alan Watt is derived from Ned Greene's description of cube mapping in the original paper. In the original description, the rays from the viewer's eyepoint intersect the corners of screen pixels. These four vectors are then reflected off the surface. The four reflected vectors subtend an area of the cube map which is filtered and averaged to derive a single color value. Figure 37 shows the relative positions of the viewer, screen, and object. Greene's method is more accurate than the hardware method described previously since it uses pixel areas rather than treating the pixel like a single point. Unfortunately, the indexing can become much more complicated than the hardware method.

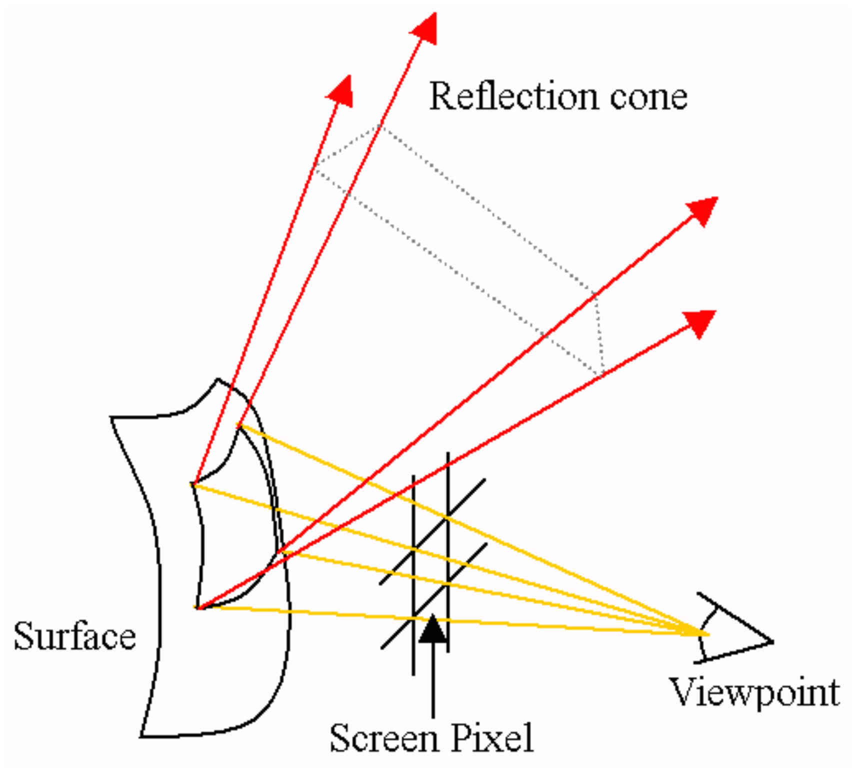


*Figure 37: The configuration of the viewer, screen, surface, and reflected vectors in cube mapping*.

Under this indexing scheme, each pixel generates four reflected rays which subtend an area of the cube as illustrated in Figure 38.

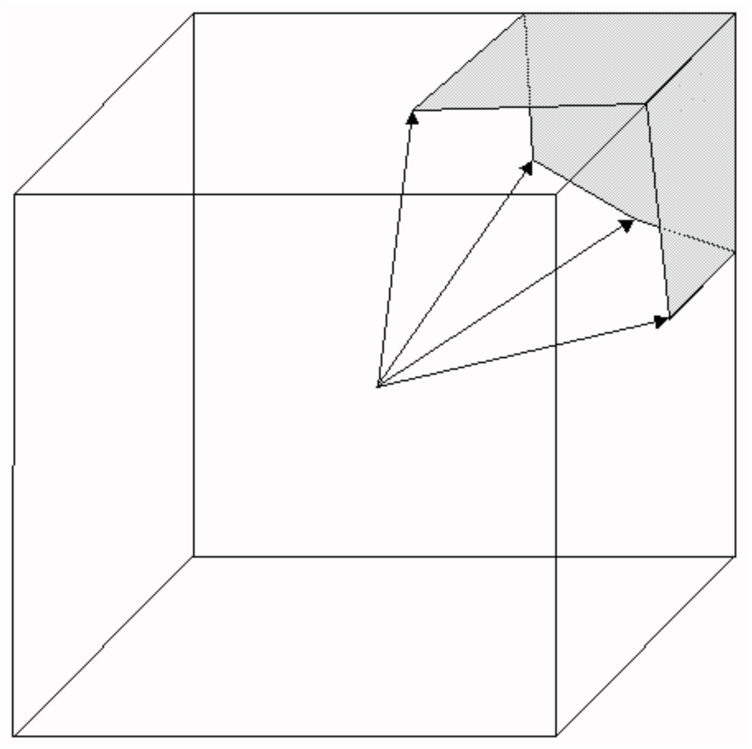


*Figure 38: The area subtended by rays generated per pixel*. *(Adapted from Watt.)*

When four vectors can be used simultaneously, the problem of determining a single color for the area is much more complicated. The main difficulty is projecting the square created by the vectors onto the cube. The projection can span many faces and retrieving the proper texels on a face involves performing edge intersections with the faces of the cube. The number of texels examined can be up to half of the entire cube meaning that 5 of the 6 faces would have to be processed for a single pixel's color value. The complex texel retrieval combined with texel filtering makes this form of cube mapping expensive. Large differences in reflected vector direction can cause extra cases where faces not touched by reflected vectors still contributed to the pixel color. Figures 39 and 40 illustrates this point.

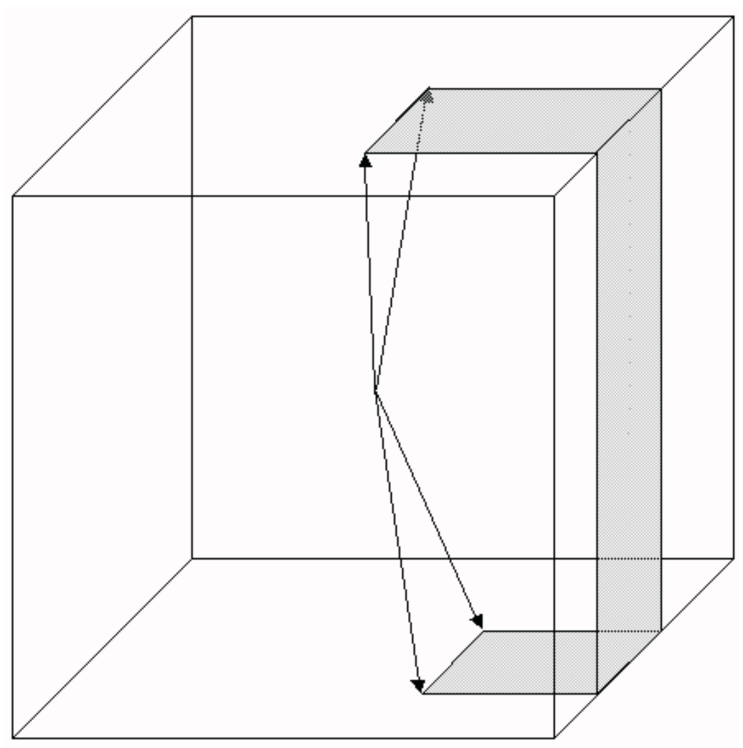


*Figure 39: Large differences in the reflected vector can skip faces which must be correctly incorporated into the final texel set.*

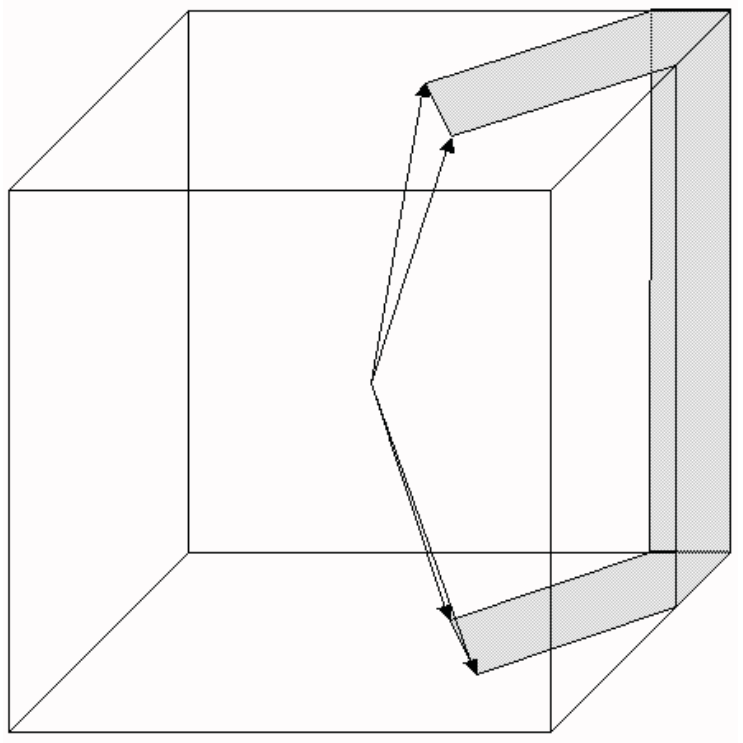


*Figure 40: A more complicated indexing situation than Figure 39. Now two extra faces must be projected against.*

Using four vectors can also create a singularity in a cube map: the four vectors can be coplanar. This is illustrated in Figure 41.

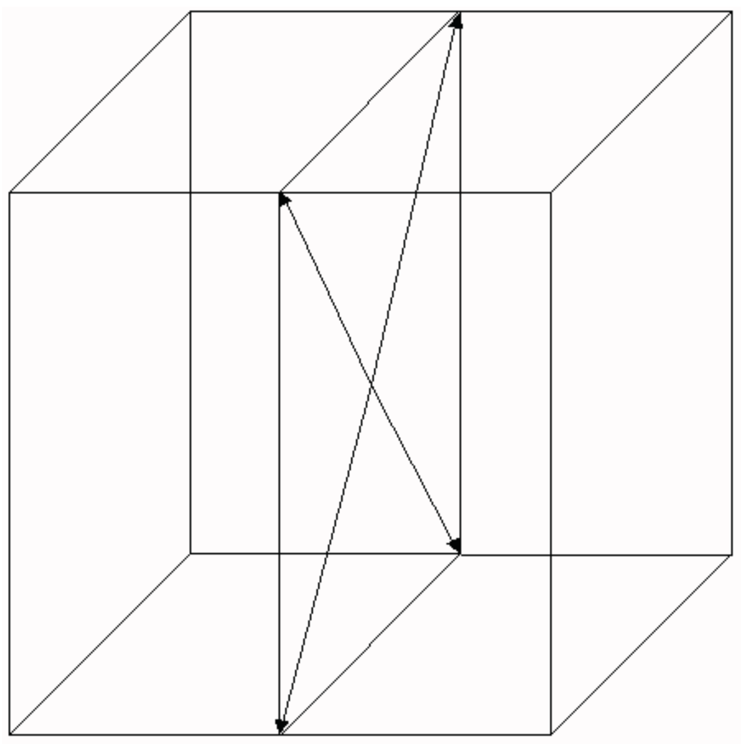


*Figure 41: A singularity produced by using four reflected vectors for indexing.*

The situation portrayed in Figure 41 can occur if the surface has high curvature. If the curvature is high enough, a single pixel may have reflected vectors in a plane. For polygonal models, this can also occur if the pixel subtends an area covered by four polygons meeting at a right pyramid. Each corner of the pixel would reflected off each polygon in a different direction, but all the reflected vectors together would be in the same plane.

Alan Watt's book provides code for generating the (s,t) coordinates per face. It is the only implementation of this form of cube mapping that is well known. The code handles situations in which the reflected vectors are on faces that share edges. Therefore situations represented in Figure 39, 40, and 41 are not handled by the code but Figure 38 is handled.

For each reflected vector, the intersection with a face of the cube is computed. These intersection points are then projected onto each face of the cube in order to define a quadrilateral area within a face containing useful texel data. For each face's quadrilateral, the texels are averaged until a final color is achieved for the pixel being cube mapped.

The simulator for cube mapping uses the hardware method for mapping the quadrilateral (mapped with a test image) but uses Watt's cube map indexing code (altered to calculate the indices of just the corners of the square based on world space reflected vectors). This is discussed in much more detail later.

Now that the indexing methods involved in cube mapping have been explored, some assumptions, problems, and benefits of cube mapping will be discussed.

**Assumptions in Cube Mapping (in addition to the assumptions mentioned in sphere mapping)**

● The cube map is axis aligned. All the indexing calculations rely on the cube map to be axis aligned for simpler equations and hence faster calculations.

**Problems with Cube Mapping**

● The mapping is discontinuous. The discontinuity is introduced by the edges of the map and result in seams along the object. Although there are methods to lessen this effect, they cannot be eliminted.

● Requires indexing through several textures to compose the final image. The book *3D Computer Graphics* by Alan Watt notes that cubic environment mapping can cause thrashing of textures and poor performance in some cases. This, however, is based on the projection of a pyramid rather than using a single point.

● Reflected vectors are interpolated linearly in hardware cube mapping. A better (and of course more expensive) method would be to use the viewer's eye vector to the vertex and an interpolated normal across the polygon to calculate the reflected vector for a particular fragment.

● Filtering occurs across texture map boundaries causing several texture maps to be indexed for a single filtered pixel.

- Multiple textures units are needed to perform cubic mapping in a single clock cycle.

- Edge pixels from adjacent faces may have color discontinuities due to rotating the camera into the different views.

**Benefits of Cube Mapping**

- Easier to generate than a sphere map. Usually a sphere map is made by projecting the faces of a cube mapping onto a sphere and then flattening the sphere to make a sphere map. With cube mapping, there is no conversion necessary.

- View dependent. Greene's method is based on intersection the projection of the pixel onto the surface of an object to generate reflected vectors. This method is view dependent since the same reflected vectors will be generated regardless of the location of the viewer. Only the orientation of the viewer with respect to the screen is important.

- Can be made view independent. Using OpenGL graphics extensions, the mapping can be made to sit still on an object while the viewer moves.

- Less distortion that other mappings. The distortion involved in cube mapping is around 10x lower than sphere mapping.

**Cube Map Simulator**

These are some screen shots of a test application used to apply environment mapping (cube mapping in this case) to a test map. The application is applying a cube map to a single quad. The test application uses hardware cube mapping for the door/portal/window while the index coordinates are displayed using the code in Alan Watt's book. The map used on the right hand side was modified to correspond to what the hardware indexing retrieves from the cube map face textures. There is also some disagreement between the image displayed on the quad and the texels referenced (the pink points). It should still provide an intuitive notion of the values references by a cubic mapping. OpenGL texture indexing extensions have been used to make the map view independent so the contents of the quad are dependent only on the viewers location rather than the orientation of the viewer. Watt's code has bee likewise modified. Figures 42, 43 and 44 show some screen shots of the test application.

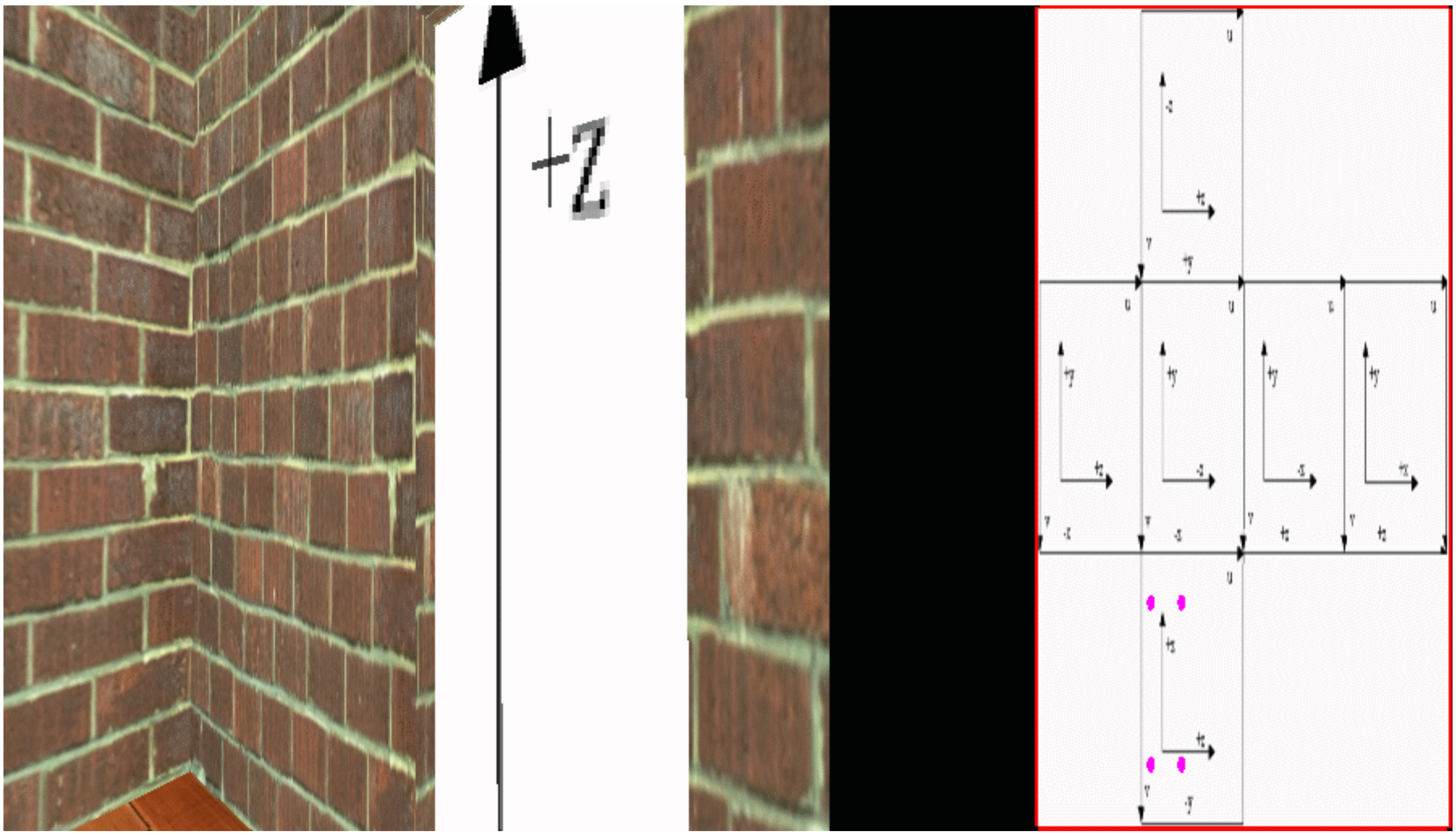


*Figure 42: The cube map simulator applied to a single quad. Since the viewer has reflected vectors within a single face, the indexing is simple.*

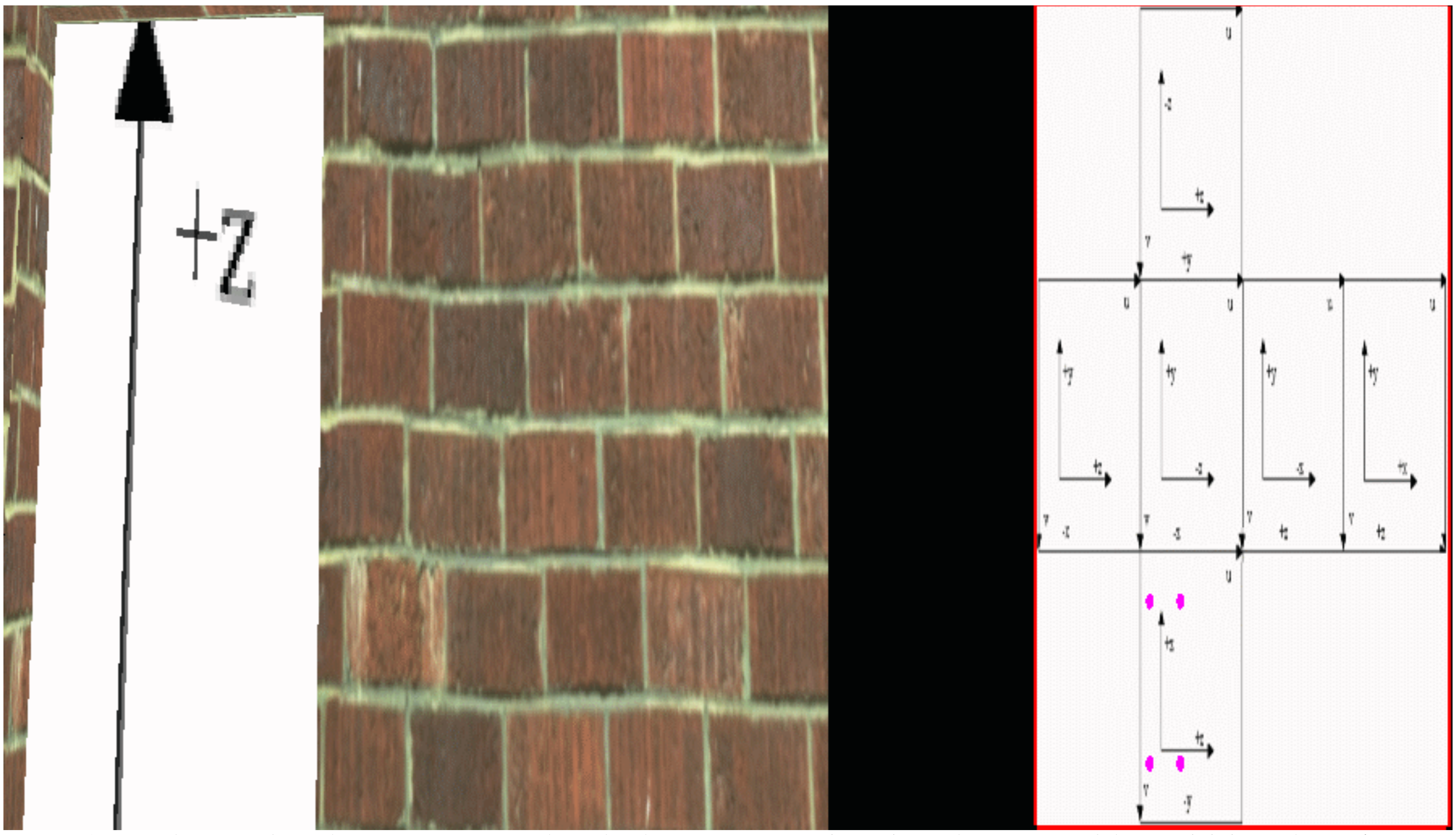


*Figure 43: Another view from the same position to show tha the map is view independent. The contents of the quad did not change because the viewer's orientation towards the quad changed.*

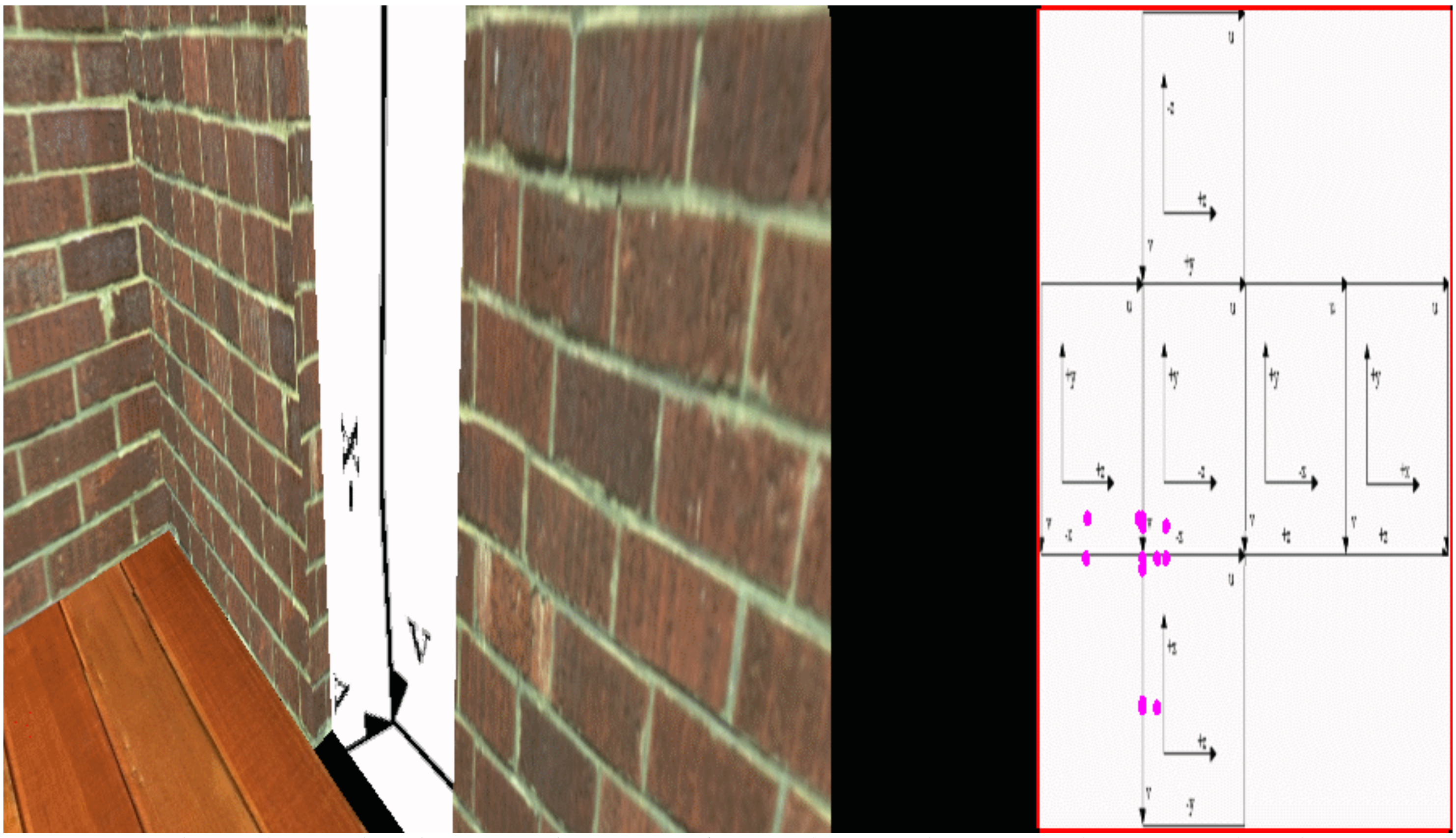


*Figure 44: Looking at the map, the indexing around a corner can be seen. The seemingly excessive indexing points (the pink points) are part of the cube map code in Alan Watt's book. The quad being environment mapped is projected onto the three faces resulting in the extra points. A benefit of the indexing method is that a face has an enclosing quad of texel values.*

The cube map simulator can be found here. It is about 1.8 megabytes in size. This executable only runs on cube map enabled hardware such as the nVidia GeForce chip. A snippet of code showing how the view independence was achieved can be found here. The executable has the limit that the reflected vectors must be on adjacent faces. If the reflected vectors are not on adjacent faces, the image on the right of the window will not display.

## Conclusion

Cube mapping is an environment mapping technique with low distortion and simple indexing equations. However, the hardware requirements and texel storage requirements are higher than sphere mapping.

## Dual Paraboloid Environment Mapping

Dual paraboloid environment mapping was invented in 1998 by Wolfgang Heidrich and Hans-Peter Seidel. It uses two texture maps to represent the entire environment. As with all the mappings presented so far, there is some distortion when representing the environment as two dimensional textures. The distortion involved in dual paraboloid mapping is on the order of cube mapping. A typical dual paraboloid environment map can be seen in Figure 45.

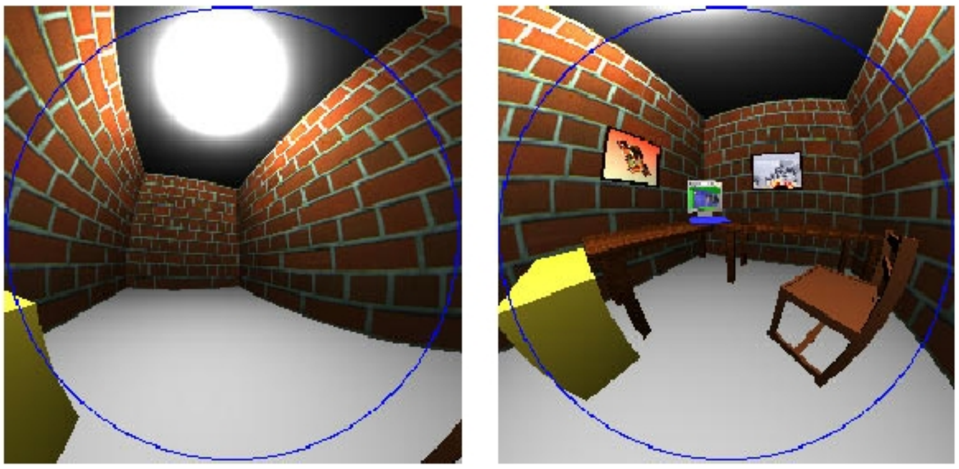


Figure 45: A dual paraboloid map. This map is from the 1998 Eurographics paper *View-Independent Environment Maps* by Heidrich and Seidel.

At first this map might be confused with two sphere map images but this is not the case. The underlying geometry of the mapping is based on a parabola which has the desireable property of containing a single point of focus for incoming rays. This property is the basis of telescopes. For dual paraboloid mapping, instead of focusing light at a particular point, the mapping tries to capture the reflected rays originating at a common origin. Therefore, even in the method used to capture the rays (such as a mirrored ellipsoid or parabolic lense system) has a radius, the image on the parabola is one of rays from a common point of origin. The map is generated at the origin of the object to be environment mapped. In this configuration, the image of the environment is still considered to be orthographic (the incoming rays are parallel when generating the environment map). Figure 46 may help make this idea more concrete.

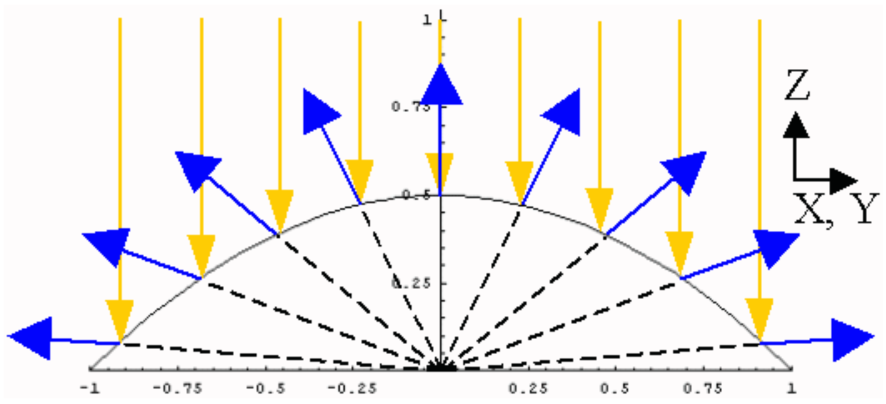


*Figure 46: Even though the parabola has a non-zero radius, the reflected rays provide information from a common origin.*

The formula for the paraboloid assumed in dual paraboloid mapping is

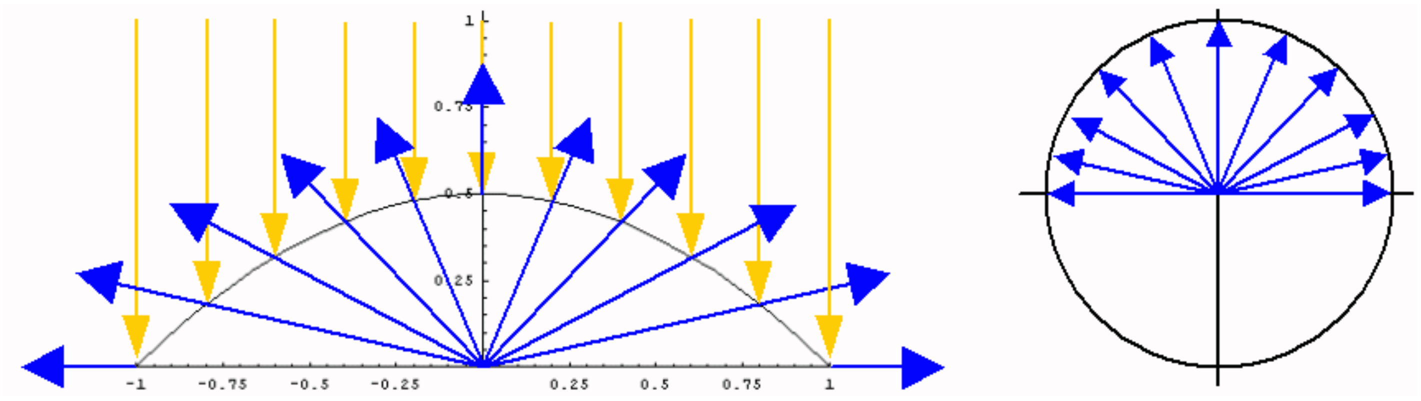
$$f(x,y) = \frac{1}{2} - \frac{1}{2}\left(x^2 + y^2\right), \quad x^2 + y^2 \le 1$$

## Properties of Dual Paraboloid Maps

Since there are two maps, the front and back of the environment receive the same number of pixels so the environment is divided 50/50. Since the reflected rays range from 0° to 90° on each map, the crossover point between front and back maps is precisely on their boundaries. This can cause some filtering difficulties so the maps are usually computed beyond the normal indexing boundary (the blue circles in Figure 45).

Since the paraboloid maps cover an area of a circle in the texture, the texture utilization similar to a sphere map. So 1*2- 2*(p*(1/2)$^2$) = 2 - p/2 = 42.9204%/2 = 21.7% of the pixels are not used in the proper form of indexing. These pixels may still contribute to filtering though.

Performing an analysis similar to the ones for sphere mapping and cube mapping provide some insight into the distortion involved in the map. Figure 47 indicate the set up for the analysis.



*Fire 47: The angle subtended by 11 incoming rays on a paraboloid map. The angle per pixel is greater towards the middle of the map.*

For a 512x512 paraboliod map, the analysis is very similar to the previous mappings and is simpler because the reflected vectors can be calculated direction from the position on the map. For step 0 to 1/256, the angle is 0.447621 whereas the final step from 255/256 to 256/256 makes an angle of 0.224249. The ratio from worst to best is roughly 1.996. Even when taking the diagonal across pixels, the mapping ratio is 1.994.

Heidrich and Seidel perform their own analysis based on differential areas and come up with a ratio of 4. A ratio of ~2 in each dimension agrees with their analysis

In general, the error for a single half of a paraboloid map is about twice that of a cube map based on 512x512 mappings. This is not necessarily a fair comparison though since the cube map has more pixels to work with than a paraboloid map. Figure 48 shows the graph of angle subtended per pixel in one dimension along 256 pixels from the center of a paraboloid map.

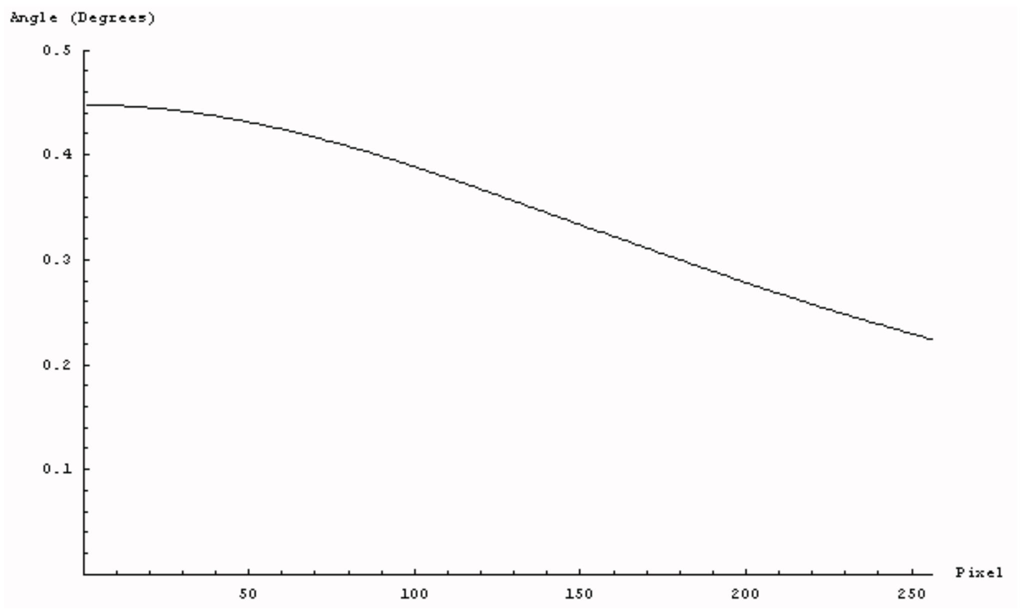


*Figure 48: A graph of angle subtended per pixel versus pixel position on the paraboloid map.*

The paraboloid mapping has very good mapping characteristics considering that it is using <1/3 the number of pixels that cube mapping requires.

## **Indexing Method (the mathematical way)**

The indexing method of dual paraboloid mapping is complicated so efforts will be made to explain each step in detail. This section closely follows Section 8.2 of Heidrich's thesis.

The map for dual paraboloid mapping is made from the point of view of the object being environment mapped and considers the object as the origin. The paraboloid map assumes that the map is created from a view looking down the positive z axis. The 'front' map is on the positive z axis while the 'back' map is facing negative z. Specifically, the vector is from the object to the eye and is specified as $\mathbf{d_o} = (0,0,1)^T$. This is usually the object space of the object. It is assumed that the transformation from the object space to the eye space is rigid and does not changed the length of vectors computed in object space.

When the viewer views the object, a vector is generated for the point being environment mapped from the surface to the eye. This vector is known as $\mathbf{v_e}$. This vector is in eye space (eye is considered the origin). Next the normal at that point is retrieved or calculated or interpolated. The normal is also in eye space and is referred to as $\mathbf{n_e}=(n_{e,x}, n_{e,y}, n_{e,z})^T$. A matrix M, called the *model/view matrix*, is assumed to exist. The model/view matrix transforms a vector from object to eye space. The eye space normal can be computed in this way from M and the object space normal. In this case, $\mathbf{n_e} = M^{-T}\,\mathbf{n_o}$ for an object space normal.

For a view vector and a normal, both in eye space, the reflected vector can be computed in eye space by calculating $\mathbf{r_e} = 2(\mathbf{n_e}.\mathbf{v_e})\mathbf{n_e} - \mathbf{v_e}$, the standard mirror reflection formula. This eye space reflected vector is then converted to object space by computing $\mathbf{r_o} = M^{-1}\mathbf{r_e}$. It is important that this object space reflected vector be of *unit length before matrices are applied*. The reflected vector is eye space has either a negative or positive z component. If the z component is 0 then either map can be referenced. If the z component is positive, the front face is indexed. If the z comonent of the object space reflected vector is negative then the back face is indexed.

The indexing method now switches to computing the normal that produced that reflected vector given the direction that the map was created in , namely $(0,0,1)^T$. Since there is a relatively simple equation for calculating the normal of a parabolic map, it is easier to use that to look up values in the texture than the reflected vector. So the problem now becomes in determining which normal on the parabola can create the object space reflected vector, $\mathbf{r_o}$, given a particular viewing direction $\mathbf{d_o}$. In particular, the equation $\mathbf{r_o} = 2(\mathbf{n}.\mathbf{d_o})\mathbf{n} - \mathbf{d_o}$ is trying to be solved for $\mathbf{n}$. Using the formula for the paraboloid originally given at the beginning of this section, the sought after normal is computed with the following formula:

$$\vec{n} = \frac{1}{\sqrt{x^2 + y^2 + 1}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{z} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

The x and y values can easily be related to texture coordinates now. Using this normal in the formula $\mathbf{d_o} + \mathbf{r_o} = 2(\mathbf{n}.\mathbf{d_o})\mathbf{n}$ creates another useful relation, namely

$$\vec{d}_o + \vec{r}_o = 2(\vec{n} \bullet \vec{v})\vec{n} = \begin{pmatrix} kx \\ ky \\ k \end{pmatrix}.$$

This is also a useful formula since it indicated that $\mathbf{d_o} + \mathbf{r_o}$ can also be used to calculate texture coordinates directly.

**Indexing Method (matrix way)**

The description given previously of the indexing method can be boiled down to a series of matrix manipulations. This is favorable to the paraboloid mapping because it means that the indexing method is amenable to graphics hardware. The generation of the (x,y) values for the parabola corresponds to the following matrix equation:

$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} = P \cdot S \cdot (M_l)^{-1} \cdot \begin{bmatrix} r_{e,x} \\ r_{e,y} \\ r_{e,z} \\ 1 \end{bmatrix}$$

This equation is performing all the steps described previously but using matrices to carry out the operations.

The matrix $M_l$ is the inverse of the affine portion of the model/view matrix and converts the reflection vector from eye space to object space. The eye space reflected vector should be normalized beforehand.

S performs the addition of the object space reflected vector from the viewing vector $\mathbf{d_o}$. S is defined as:

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

This definition of S is different than in the original paper describing paraboloid mapping. This is due to the fact that the reflection vector equation was the reverse of the equation presented here (and in Heidrich's thesis).

So far the equation looks like:

$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} = P \cdot \begin{bmatrix} kx \\ ky \\ k \\ 1 \end{bmatrix}$$

The only operation left is to divide the coordinates by k which is performed by the matrix P. P is defined as:

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

To map to (x,y), the k portion must be divided out. Now the coordinates map uniques onto the parabola which ranges from -1 to 1. In order to generate appropriate texture coordinates, a final matrix multiplication must be performed. Resulting in the final step:

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

Equations for a back facing reflected vector are very similar. In hardware, multiple texturing units are used simultaneously to apply both mappings onto a polygon. The alpha (or transparency) value of a map is used to decide between which mapping to show.

Although the mapping may seem completely linear, it is important to recall that the initial calculation of the reflected vector in eye space is calculated in software since it cannot be represented as a simple matrix multiply. Sphere mapping also performs the reflected vector calculation in software. To help this indexing method, graphics companies such as nVidia have created OpenGL extensions such as `GL_REFLECTION_MAP_EXT` and `GL_NORMAL_MAP_EXT`.

Now that the generation of texture coordinates for this map has been discussed, the assumptions, problems, and benefits of the map will be covered.

**Assumptions in Dual Paraboloid Mapping (in addition to the assumptions mentioned in sphere mapping)**

● The map is created from a viewing direction of (0,0,1). (or (0,0,-1) if some signs are reversed in the eqations)

**Problems with Dual Paraboloid Mapping**

● Requires multiple passes unless there is multitexturing hardware.

● Since the map is nonlinear, there some filtering irregularities towards the edge. A pixel subtends more of the environment in one direction than another so linear filtering is anisotropic.

● Edge pixels between maps need to perform special blending in order to reduce artefacts from using two maps.

● Almost 25% the pixels are not used.

**Benefits of Dual Paraboloid Mapping**

● Can be directed generated through special camera system making real world data obtainable with little processing or initial distortion.

● Nearly completely linear indexing scheme. This allows significant speed up with hardware support.

● Low distortion mapping.

● View independent. The mapping was designed at the outset to provide a view independent method of environment mapping.

● Good compromise between the hardware demands of cube mapping and the distortion of sphere mapping.

**Dual Paraboloid Mapping Simulator**
There were some difficulties in creating the dual paraboloid map simulator so it is not provided. The software indexing moves correctly and given a sense of the indexing performed in a dual paraboloid map, but the OpenGL implementation of the map did not move properly due to some error in state. A surprising aspect of implementing the indexing method is that one map needs to be upside down for proper indexing to occur across map boundaries.

The usual implementation in hardware requires an alpha value to be associated with each map. The maps have an alpha of 0 (solid) inside the blue circle and an alpha of 0 outside. With dual paraboloid maps, if one map is facing the viewer, the indexing of the other map falls outside the circle. In this way, the alpha value can be used to choose between the two maps automatically as each texture is applied to the object.

Figure 49 an idea of the indexing methods involved in dual paraboloid mapping.

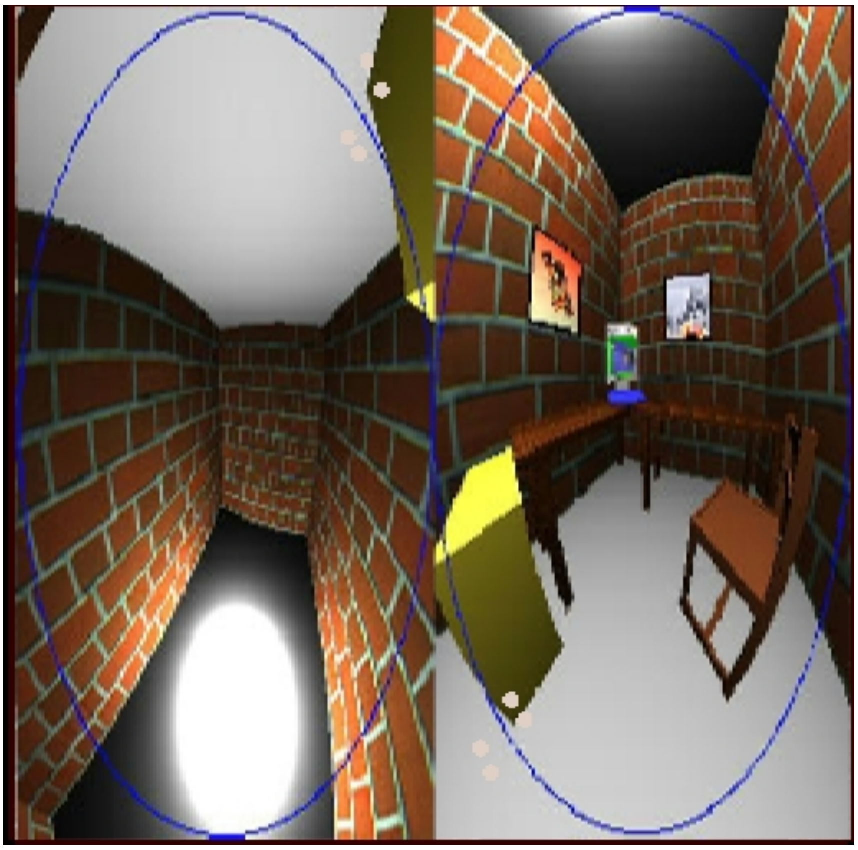


*Figure 49: Indexing across texture boundaries seems to wrap around from bottom to top and vice versa.*

## Conclusion

Dual paraboloid mapping is an good compromise between full blown cubic mapping and distortion prone sphere mapping.