Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces

Terry Welsh
terry@infiscape.com
Infiscape Corporation
January 18, 2004
Revision 0.3

Abstract
This paper describes parallax mapping and an enhancement that makes it more practical for general use. Parallax mapping is a method for approximating the correct appearance of uneven surfaces by modifying the texture coordinate for each pixel. Parallax is exhibited when areas of a surface appear to move relative to one another as the view position changes. This can be observed on any surface that is not flat. Parallax can be simulated well by constructing a complete geometric model of a surface, but it can be computationally expensive to draw all the necessary polygons. The method presented here requires no extra polygons and approximates surface parallax using surface height data.

1 Introduction
The reader is assumed to have an understanding of computer graphics concepts and experience using a graphics language such as OpenGL.

Modern graphics hardware allows us to replace traditional vertex transformations and lighting equations with custom programs, known as “vertex programs” and “fragment programs.” A vertex program acts upon vertex data, transforming vertices from Cartesian coordinates to screen coordinates. Not only can vertex programs perform these necessary transformations, they can also modify and output vertex positions, colors, normals, texture coordinates, and other attributes. These outputs are interpolated across the surface of a triangle defined by three such vertices to provide input for fragment programs. A fragment program is executed once for every pixel that is drawn, receiving inputs in the form of interpolated vertex data, texture data, and possibly other values. All this information is used to compute the final color of each pixel.

Using vertex and fragment programs, it is possible to replace the traditional vertex projection and lighting equation used by real-time computer graphics hardware. The method presented in this paper applies these capabilities to achieve an approximation of the parallax that can be observed on pitted or uneven surfaces.

The following text presents pieces of vertex and fragment programs that use the syntax of OpenGL's ARB_vertex_program version 1.0 and ARB_fragment_program version 1.0.

2 Related Work

2.1 Parallax Mapping

This technique approximates the parallax that can be observed on uneven surfaces. Parallax mapping [3] alone will only approximate parallax and will not simulate occlusions, silhouettes, or self shadowing. It is a 2D process, so no extra polygons are required. It will be explained in detail in Section 4.1.

2.2 Displacement Mapping

The most straightforward way to achieve parallax effects when simulating a complex complex surface is to build a detailed model of the surface. Displacement mapping [2] is a technique that subdivides a surface into a large number of small triangles. The vertices of these triangles are then displaced perpendicularly to the surface normal according to some height function. The vertices can also be assigned new normals to accurately simulate the surface lighting.

2.3 Relief Texture Mapping

Relief texture mapping (RTM) [4] is very similar to parallax mapping in that it attempts to simulate parallax with an image warping technique. Relief textures are pre-warped through a software process before being sent to the graphics hardware. Using many relief textures in a single application is likely to prevent the application from achieving real-time frame rates. However, RTM gives high quality results. It correctly handles self-occlusions and produces texel-accurate silhouettes.

2.4 View-Dependent Displacement Mapping

View-dependent displacement mapping (VDM) [5] is a per-pixel rendering technique capable of self-shadowing, occlusions, and silhouettes. It stores a many-dimensional surface description in several texture maps, which are pre-computed from a height map. Although it requires a large amount of memory for textures, VDM produces convincing images at high frame rates. A good parallax effect is achieved because VDM textures store the relation ship between texels along many viewing directions.

3 Background

The concept of tangent space is probably the most important piece of background information required to understand parallax mapping. Tangent space is a coordinate system that is oriented relative to a given surface. It is defined by three axes, the tangent, binormal, and normal. The tangent and binormal lie in the plane of the surface, while the normal is perpendicular to the plane of the surface. If a surface is curved, then tangent space rotates as you move along that surface.

To apply bump mapping [1] with a normal map, it is necessary to compute the dot product of a vector pointing toward the light with a normal for each pixel. Because the normal map is mapped to the surface, it already lies flat in tangent space. To compute the necessary dot products, the light vectors must lie in the same coordinate system, so they are transformed into tangent space as well. Parallax mapping uses the concept of tangent space in the same manner.

4 Parallax Mapping

4.1 The basics

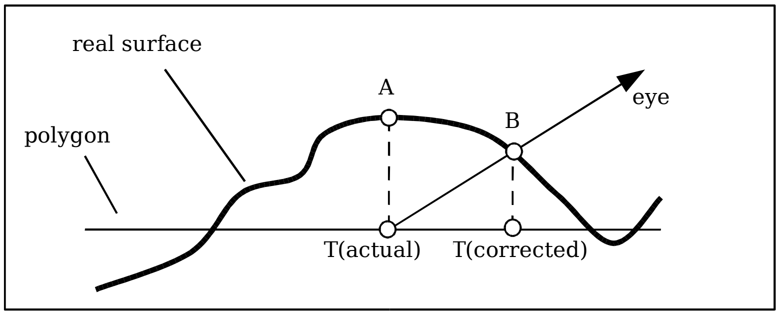

Figure 1. The observed texel is incorrect because the texture map has been flattened onto the polygon.

When a texture map representing an uneven surface is applied to a flat polygon, the surface appears to get flattened. In Figure 1 you can see that when viewing the polygon along the depicted eye vector, you will see point A of the surface. However, if you were viewing the actual surface instead of a texture mapped polygon, you would see point B. If the texture coordinate corresponding to point A could be corrected, then you could view point B instead. By offsetting all the texture coordinates individually, high areas of the surface would shift away from the eye and low areas of the surface would shift toward the eye. Thus parallax would be achieved for the simulated surface. The process of parallax mapping requires that, for each pixel drawn, a texture coordinate, used to index one or more texture maps, be corrected by some displacement. This technique approximates the parallax seen when moving the eye relative to an uneven surface.

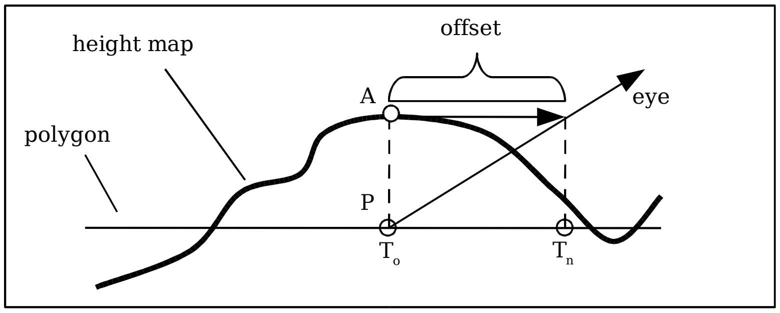

Figure 2. Calculating the offset.

Three components are required to compute an offset texture coordinate for a pixel: a starting texture coordinate, a value of the surface height, and a tangent space vector pointing from the pixel to the eye point. In Figure 2 you can see how it is possible to modulate the eye vector by the surface height to obtain a texture coordinate offset.

A standard height map is used to represent the varying height of the surface. The height map correlates to the surface's regular texture map and stores one height value per texel. These values are in the range {0.0, 1.0}.

Since any parallax effect depends on point of view, a vector from a point on the surface to the eye is required. The original texture coordinate and height values are already represented in tangent space, so the eye vector must be as well. The eye vector can be created in global coordinates by subtracting a surface position from the eye position and transforming the resulting vector into tangent space. An application programmer must supply a tangent, binormal, and normal at each vertex. These can be used to create a rotation matrix at every point on the surface which will transform vectors from the global coordinate system to tangent space.

To compute a texture coordinate offset at a point P, the eye vector must first be normalized to produce a normalized eye vector V. The height h at the original texture coordinate T_o is read from the height map. The height is scaled by a scale factor s and biased by a bias b in order to map the height from the range {0.0, 1.0} to a range that better represents the physical properties of the surface being simulated. For example, a brick wall texture might cover a 2x2 meter area. Also imagine the surface of the bricks and the recessed grout give the surface a thickness of 0.02m. The correct scale factor for this material would be 0.02 / 2 = 0.01. Coupling this with a bias of 0.0 would then remap the height field to a range of {0.0, 0.01}. Using a bias of -0.01 would remap the range to {-0.99, 0.0}. The average of these two extremes is probably best for most textures.

$$b = s \cdot -0.5$$

A bias that excludes 0.0 from the range would be impractical since that assumes that the surface being simulated does not intersect the polygon that represents it. The new scaled and biased height h_{sb} is given by

$$h_{sb} = h \cdot s + b$$

An offset is then computed by tracing a vector parallel to the polygon from the point on the surface directly above P to the eye vector. This new vector is the offset and can be added to T_o to produce the new texture coordinate T_n.

$$T_n = T_o + (\, h_{sb} \cdot V_{\{x,y\}} \,/\, V_{\{z\}} \,)$$

The new texture coordinate is used to index a regular texture map and any other maps applied to the surface.

4.2 Implementation

Now to implement this on some fast graphics hardware. Since vector data can be interpolated between connected vertices by graphics hardware, the simplest solution is to calculate a vector from each vertex to the eye point within a vertex program. In the language of ARB_vertex_program, these vectors can be calculated with the following code.

```
# input data
PARAM mvit[4] = {state.matrix.modelview.invtrans};
ATTRIB tangent = vertex.texcoord[1];
ATTRIB binormal = vertex.texcoord[2];
ATTRIB normal = vertex.normal;

# vector pointing to eye
SUB eyevec, mvit[3], vertex.position;

# transform eye vector into tangent space (DO NOT NORMALIZE)
DP3 result.texcoord[3].x, eyevec, tangent;
DP3 result.texcoord[3].y, eyevec, binormal;
DP3 result.texcoord[3].z, eyevec, normal;
MOV result.texcoord[3].w, 1.0;
```

The tangent, binormal, and normal attributes represent the coordinate axes of the tangent space surrounding each vertex. The eye vector is stored in texcoord[3], which is a parameter that will be interpolated across the surface so that each pixel drawn will receive a unique eye vector.

The remainder of the computation is performed on a per-pixel basis in a fragment program.

```
TEMP height, eyevects, newtexcoord, temp;

# normalize tangent space eye vector
DP3 temp, fragment.texcoord[3], fragment.texcoord[3];
RSQ temp, temp.x;
```

```
MUL eyevects, fragment.texcoord[3], temp;

# calculate offset and new texture coordinate
TEX height, fragment.texcoord[0], texture[2], 2D;
MAD height, height, 0.04, -0.02;  # scale and bias
RCP temp, eyevects.z;
MUL height, height, temp;
MAD newtexcoord, height, eyevects, fragment.texcoord[0];
```

The variable newtexcoord stores the shifted texture coordinate, which is used to index other texture maps.

4.3 Offset Limiting

So far this paper has described a 2D approximation with a significant flaw. The computation presented in Section 4.1 assumes that the point at T_n has the same height as the point at T_o, which is rarely true. Any surface that exhibits parallax will have varying heights.

At steep viewing angles, texture coordinate offsets tend to be small. A small offset means that the height at T_n is likely to be very close to the height at T_o. Consequently, the offset will be nearly correct. As the viewing angle becomes more shallow, offset values approach infinity. When offset values become significantly large, the odds of T_n indexing a similar height to that of T_o fade away to random chance. This problem can reduce surfaces with complex height patterns to a shimmering mess of pixels that don't look anything like the original texture map.

A simple solution to this problem is to limit the offsets so that they never grow longer than h_{sb}. Since parallax mapping is an approximation, any limiting value could be chosen, but this one works well enough and it reduces the code in the fragment program by two instructions. The new equation is

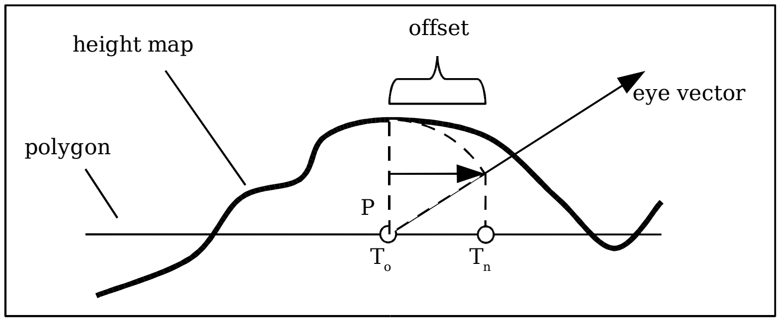

Figure 3. Calculating the offset with limiting.

$$T_n = T_o + (h_{sb} \cdot V_{\{x, y\}})$$

Figure 3 shows that the texture coordinate offset will be no longer that the height at T_o. And the simplified fragment code is

```
TEMP height, eyevects, newtexcoord;

# normalize tangent space eye vector
DP3 temp, fragment.texcoord[3], fragment.texcoord[3];
RSQ temp, temp.x;
MUL eyevects, fragment.texcoord[3], temp;

# calculate offset and new texture coordinate
TEX height, fragment.texcoord[0], texture[2], 2D;
MAD height, height, 0.04, -0.02;  # scale and bias
MAD newtexcoord, height, eyevects, fragment.texcoord[0];
```

This simplification keeps textures looking good at shallow angles, but it also reduces the parallax effect, which isn’t as bad as it sounds. Parallax effects are best viewed close up, but surfaces viewed at shallow angles tend to be relatively far away from the eye point. As surfaces recede into the distance they become smaller, losing detail to minification and texture filtering.

5 Applying Parallax Mapping

Parallax mapping is a computationally inexpensive, approximate 2D effect that. All that it does is modify texture coordinates in order to warp a texture mapped image. Because of its ease of implementation, it can be used in conjunction with other types of mapping. The new texture coordinates can be used to index regular texture maps for coloring a surface, normal maps for bump mapping and computing reflection vectors, shininess maps for computing reflectivity, and probably any other standard type of texture data that your application may use.

The modified texture coordinates are not perfect, though, so care must be taken in producing height maps and specifying scale and bias values. Because of the faulty assumption that height values are the same from one texture coordinate to the next, height maps with wildly varying heights tend to cause problems. Parallax mapping is incapable of detecting areas of a surface that occlude other areas, which is common on a surface with steep angles. This technique will produce the most attractive results when using height maps with gradual gradations and shallow angles. The opposite extreme of height map can result in distracting artifacts when a surface is viewed at shallow angles.

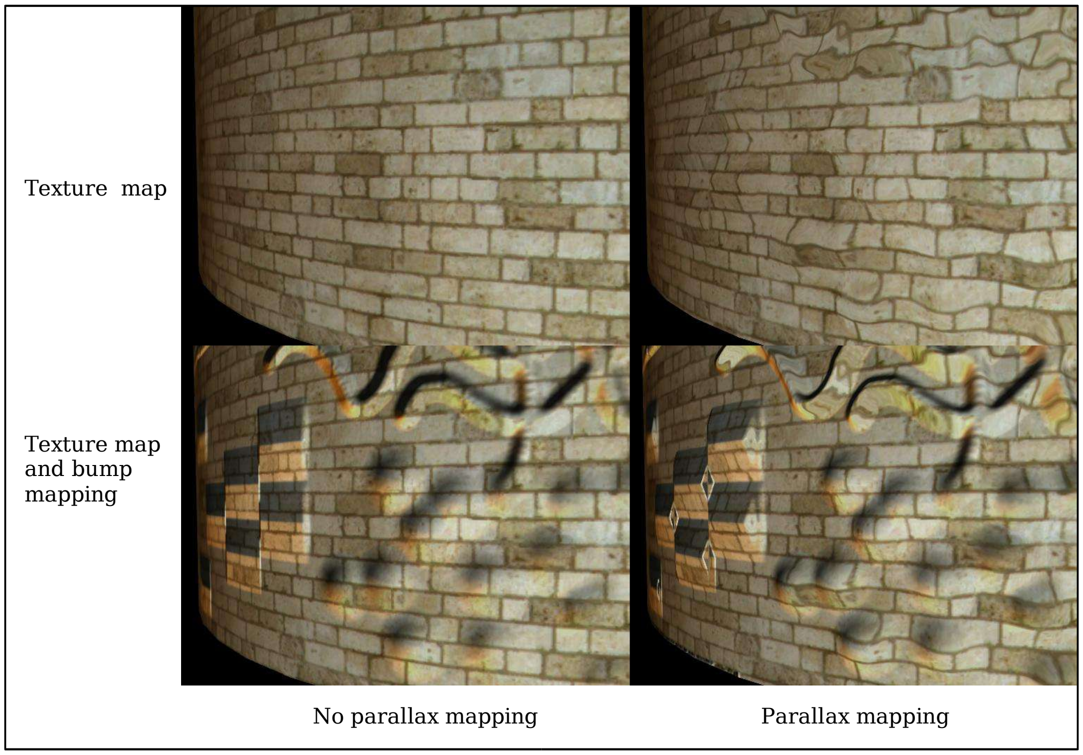

Figure 4. Comparison of various rendering techniques. Notice the artifacts caused by steep angles in the height map. Subtle height changes like those in the lower-right yield the best results.

6 References

1. James F. Blinn. “Simulation of Wrinkled Surfaces,” Computer Graphics (SIGGRAPH ' 78 Proceedings), volume 12, pages 286-292, August 1978.

2. Robert Cook. "Shade trees", Computer Graphics (SIGGRAPH '84 Proceedings) 13(3), 223-231, 1984.

3. Tomomichi Kaneko, et al. "Detailed Shape Representation with Parallax Mapping," ICAT, 2001.

4. Oliveira M., Bishop G., and McAllister D. "Relief Texture Mapping," Computer Graphics (SIGGRAPH '00 Proceedings), 359-368, 2000.

5. Lifeng Wang, et al. "View-Dependent Displacement Mapping," ACM Transactions on Graphics, Volume 22, Issue 3, July 2003.

7 Appendix

A vertex program for computing bump mapping and parallax mapping:

```
!!ARBvp1.0

PARAM mvi[4] = {state.matrix.modelview.inverse};
PARAM mvit[4] = {state.matrix.modelview.invtrans};
PARAM mvp[4] = {state.matrix.mvp};

ATTRIB tangent = vertex.texcoord[1];
ATTRIB binormal = vertex.texcoord[2];
ATTRIB normal = vertex.normal;

TEMP light0pos, light0vec;
TEMP light1pos, light1vec;
TEMP eyevec, eyevects;
TEMP temp;

# vector pointing to light0
DP4 light0pos.x, mvi[0], state.light[0].position;
DP4 light0pos.y, mvi[1], state.light[0].position;
DP4 light0pos.z, mvi[2], state.light[0].position;
DP4 light0pos.w, mvi[3], state.light[0].position;
SUB light0vec, light0pos, vertex.position;

# transform light0 vector into tangent space (DO NOT NORMALIZE)
DP3 result.texcoord[1].x, light0vec, tangent;
DP3 result.texcoord[1].y, light0vec, binormal;
DP3 result.texcoord[1].z, light0vec, normal;
MOV result.texcoord[1].w, 1.0;

# vector pointing to light1
DP4 light1pos.x, mvi[0], state.light[1].position;
DP4 light1pos.y, mvi[1], state.light[1].position;
DP4 light1pos.z, mvi[2], state.light[1].position;
DP4 light1pos.w, mvi[3], state.light[1].position;
SUB light1vec, light1pos, vertex.position;

# transform light1 vector into tangent space (DO NOT NORMALIZE)
DP3 result.texcoord[2].x, light1vec, tangent;
DP3 result.texcoord[2].y, light1vec, binormal;
DP3 result.texcoord[2].z, light1vec, normal;
MOV result.texcoord[2].w, 1.0;

# vector pointing to eye
SUB eyevec, mvit[3], vertex.position;

# transform eye vector into tangent space (DO NOT NORMALIZE)
DP3 result.texcoord[3].x, eyevec, tangent;
DP3 result.texcoord[3].y, eyevec, binormal;
```

```
DP3 result.texcoord[3].z, eyevec, normal;
MOV result.texcoord[3].w, 1.0;

# regular output
DP4 result.position.x, mvp[0], vertex.position;
DP4 result.position.y, mvp[1], vertex.position;
DP4 result.position.z, mvp[2], vertex.position;
DP4 result.position.w, mvp[3], vertex.position;
MOV result.color, vertex.color;
MOV result.texcoord[0], vertex.texcoord[0];

END
```

A fragment program for computing parallax mapping and bump mapping:

```
!!ARBfp1.0

PARAM light0color = state.light[0].diffuse;
PARAM light1color = state.light[1].diffuse;
PARAM ambient = state.lightmodel.ambient;

TEMP eyevects;
TEMP rgb, normal, height, temp, bump, total;
TEMP light0tsvec, light1tsvec;
TEMP newtexcoord;

# normalize tangent space eye vector
DP3 temp, fragment.texcoord[3], fragment.texcoord[3];
RSQ temp, temp.x;
MUL eyevects, fragment.texcoord[3], temp;

# calculate offset and new texture coordinate
TEX height, fragment.texcoord[0], texture[2], 2D;
MAD height, height, 0.04, -0.02;  # scale and bias
MAD newtexcoord, height, eyevects, fragment.texcoord[0];

# get texture data
TEX rgb, newtexcoord, texture[0], 2D;
TEX normal, newtexcoord, texture[1], 2D;

# remove scale and bias from the normal map
MAD normal, normal, 2.0, -1.0;

# normalize the normal map
DP3 temp, normal, normal;
RSQ temp, temp.r;
MUL normal, normal, temp;

# normalize the light0 vector
DP3 temp, fragment.texcoord[1], fragment.texcoord[1];
RSQ temp, temp.x;
MUL light0tsvec, fragment.texcoord[1], temp;

# normal dot lightdir
DP3 bump, normal, light0tsvec;

# add light0 color
MUL_SAT total, bump, light0color;

# normalize the light1 vector
DP3 temp, fragment.texcoord[2], fragment.texcoord[2];
RSQ temp, temp.x;
MUL light1tsvec, fragment.texcoord[2], temp;

# normal dot lightdir
DP3 bump, normal, light1tsvec;
```

```
# add light1 color
MUL_SAT bump, bump, light1color;
ADD_SAT total, total, bump;

# add ambient lighting
ADD_SAT total, total, ambient;

# multiply by regular texture map color
MUL_SAT result.color, rgb, total;

END
```