Title: Derivative-Based Relief Mapping.

This project presents a new, and more versatile, method for performing Relief Mapping (also known as Parallax Occlusion Mapping), utilizing rates of change in texture coordinates across a polygon surface to calculate the texture sampling offsets used in the ray-tracing portion of the Relief Mapping algorithm. This new technique relies solely upon the texture-coordinates, vertex-coordinates, and a depth-map texture, allowing Relief Mapping to be utilized on complex meshes without the need for additional per-polygon information such as texture-aligned tangents or normal maps.
Derivative-Based Relief Mapping

by
Michael Tichenor

A PROJECT REPORT
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of
Master of Science

Presented October 29, 2012
Commencement June 2013
ACKNOWLEDGEMENTS

There are a great many people that I would like to thank, for their support, knowledge, and assistance in the completion of both this project and my education at Oregon State University.

First, I would like to thank my family for their love, support, and guidance, especially my mother Janice, my father Allen, my sister Jennifer, and my aunt Caryl.

Second, I would like to thank the faculty and staff at OSU who have contributed to my education, especially Professors Mike Bailey, Ron Metoyer, Alex Groce, Carlos Jensen, Timothy Budd, and Eugene Zhang.

Third, I would like to thank all of the students and alumni of OSU who have helped me or worked with me over the past decade, especially Islam Almusaly, Patrick Neill, Benjamin Jansen, Sanchit Karve, Faezeh Bahmani, Sanchit Karve, Alvaro Cervantes, and the many members of the OSU Graphics Group.

Last, I would like to thank all of my closest friends for their support and camaraderie, especially Brian Ewell, Allan Tokuda, Gregory Courtox, Kate McLean, Chris Gasser, and Andrew Leach.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>Rasterization and Ray-traced imaging</td>
<td>3</td>
</tr>
<tr>
<td>A Review of Relief Mapping</td>
<td>6</td>
</tr>
<tr>
<td>Illustration 1:</td>
<td>6</td>
</tr>
<tr>
<td>Illustration 2:</td>
<td>7</td>
</tr>
<tr>
<td>Texture coordinate offset calculations</td>
<td>9</td>
</tr>
<tr>
<td>Illustration 3:</td>
<td>9</td>
</tr>
<tr>
<td>Illustration 4:</td>
<td>10</td>
</tr>
<tr>
<td>Per-Fragment Derivatives</td>
<td>12</td>
</tr>
<tr>
<td>Illustration 5:</td>
<td>12</td>
</tr>
<tr>
<td>Height Map Sampling Techniques and Results</td>
<td>14</td>
</tr>
<tr>
<td>Illustration 6:</td>
<td>14</td>
</tr>
<tr>
<td>Illustration 7:</td>
<td>15</td>
</tr>
<tr>
<td>Illustration 8:</td>
<td>16</td>
</tr>
<tr>
<td>Illustration 9:</td>
<td>17</td>
</tr>
<tr>
<td>Table 1:</td>
<td>18</td>
</tr>
<tr>
<td>Dynamic Surface Normal Calculations</td>
<td>20</td>
</tr>
<tr>
<td>Illustration 10:</td>
<td>20</td>
</tr>
<tr>
<td>Illustration 11:</td>
<td>22</td>
</tr>
<tr>
<td>Occlusion Detection For Self-Shadowing</td>
<td>23</td>
</tr>
<tr>
<td>Illustration 12:</td>
<td>23</td>
</tr>
<tr>
<td>Illustration 13:</td>
<td>24</td>
</tr>
<tr>
<td>Illustration 14:</td>
<td>25</td>
</tr>
<tr>
<td>Illustration 15:</td>
<td>26</td>
</tr>
<tr>
<td>Results and Conclusions</td>
<td>27</td>
</tr>
<tr>
<td>Illustration 16:</td>
<td>28</td>
</tr>
<tr>
<td>References</td>
<td>29</td>
</tr>
<tr>
<td>Appendix</td>
<td>31</td>
</tr>
</tbody>
</table>
Derivative-Based Relief Mapping
by Michael Tichenor
INTRODUCTION

Attempts to apply Relief Mapping to complex geometric models in existing applications have been hindered by the need to compute the 2D texture-aligned tangent and bitangent vectors that form the mapping of 3D Cartesian coordinate space to 2D texture coordinate space. This is partially because many of the existing geometric models do not have precomputed per-vertex tangents and bitangents. Many 3D modeling software packages and file formats lack the support for creating and storing texture-oriented tangents and bitangents as well.

The technique discussed in this paper circumvents the requirement for per-vertex 2D texture tangents and bitangents by utilizing per-fragment derivatives to construct a dynamic 3D-to-2D texture coordinate mapping, thus allowing Relief Mapping to be utilized on nearly all existing geometric models that have per-vertex texture coordinates and per-vertex surface normals. This can even be done without per-vertex surface normals, although it results in a flat-shading effect.

Understanding the Derivative-Based Relief-Mapping technique requires some understanding of Relief Mapping. The Relief Mapping technique is a hybrid combination of the two primary means of creating computer generated images: Rasterization and Ray-Tracing. Understanding how the Relief Mapping algorithm works requires some basic knowledge of rasterization and ray-tracing. Thus, this paper is organized as follows:

1. A brief overview of rasterization and ray-tracing.
3. Explanation of the derivative-based relief mapping algorithm.
4. A comparison of some of the relief mapping sampling methods.
5. A discussion of advanced techniques such as self-shadowing.
6. My conclusions and future work.
RASTERIZATION AND RAY-TRADED IMAGING

Images generated by computers currently fall into two categories: Ray-Tracing and Rasterization. Ray-Tracing is done by tracing the path of photons through a simulated scene consisting of various surfaces (simulated via planes, polygons, quadrics, bézier curves, non-uniform rational B-spline surfaces, radiosity photon-mapping, etc.). While ray-tracing can produce much higher quality images (some are indistinguishable from real-life photos), ray-traced images take much more time to generate. Because of the computational complexity, ray-tracing is not typically used for real-time image rendering of complex scenes. However, as demonstrated by NVIDIA, real-time ray-tracing in video games and simulations is not terribly far off [GTC2012].

Rasterization is done by rendering the scene as simple geometry (polygons, lines, or points), transforming the vertex positions of the geometry to map the geometry to a flat surface, and then progressively coloring each point on that surface. The coloring of the surface is accomplished by performing lighting calculations and applying textures or materials to the surface. Attempts to increase image quality usually revolve around polygon density, polygon texturing, or sophisticated lighting calculations.

The increase in polygon density is referred to as tessellation. More modern graphics processing units (GPUs) are capable of performing tessellation without a drastic loss of frame rates, but tessellation's acceptance has been gradual in the video game industry due to the need to support older graphics cards.

In the mid-1990s, constructing a real-time scene was a balancing act of maximizing scene detail while minimizing polygon counts. This was in part due to early graphics accelerators, such as the very popular ATI Rage 128 and 3DFX Voodoo graphics cards having a linear polygon processing pipeline. Rendering a small number of polygons with a texture to change the color was faster than rendering a large number of polygons that
were independantly colored via a material. This also simplified the matter of illuminating the surfaces, as the light intensities could be calculated at each vertex and interpolated across the polygon.

Real-time lighting computation, such as Phong [PHONG1975], Lambert, and Blinn [BLINN1977] lighting equations, rely upon surface normals to determine shading. As mentioned above, increasing image quality could be done through increasing polygon density or by texturing the polygons. Because of the speed at which texture lookups were performed in these early graphics cards, compared to the speed of the polygon transformation pipeline, it was a logical next step to store surface normal information in a texture in order to simulate a more complex surface. Thus was born the technique called Normal Mapping [COHEN1998].

Normal Mapping is a relatively straightforward technique where the surface normal vector offsets, relative to the actual surface normal, are stored in the red, green, and blue channels of a texture. The texture is loaded just as a color texture would be, but is used to calculate surface illumination at each point of the surface rather than calculating it at the vertices and interpolating it across the surface. Normal Mapping hardware support spread rapidly between 1999 and 2002, to the point where nearly all graphics cards supported it.

This trend of utilizing textures to store geometric information continued, with other alternative techniques being developed. One of those techniques was called Bump Mapping [BECKER1994]. Instead of storing surface normals, Bump Mapping attempted to simulate the illusion of geometric complexity by storing height offsets in a single texture color channel. Bump Mapping changes the normal direction based upon the change in depth across the surface.

The two most recently developed advancements in texture-based real-time
rendering techniques are called Parallax Mapping [KANEKO2001] and Relief Mapping [OLIVIERA2000]. These methods use the height map and/or normal map information along with the viewing direction in order to change both the surface normal and the texture lookup location. In essence, they create the illusion of depth on a flat surface.

The Parallax Mapping equation works by distorting the texture coordinate interpolation across a surface, increasing or decreasing the texture density based upon the difference between the surface-to-camera vector and the surface normal vector, as calculated via a bump map. Because the texture coordinates are merely compressed or expanded across a surface, this doesn't work at very steep viewing angles. Relief Mapping, on the other hand, does.
A REVIEW OF RELIEF MAPPING

Relief Mapping, also known as Parallax Occlusion Mapping (POM), treats the texture distortion as a ray-tracing problem. This algorithm works by effectively tracing a ray from the camera position through the surface of the polygon, checking the height map texture values that correspond to the surface at intervals along the ray. If the ray would "intersect" the surface if it existed at that height, rather than the real surface location, the corresponding texture coordinate would be used for color lookups. This can also be used to dynamically generate surface normals for illumination based upon the height map texels adjacent to the height map intersection coordinate.

This ray-trace starts at the original location where the ray from the camera intersects the surface position. From that surface position, a new ray is generated that is progressively increased in length until it reaches the maximum allowed parallax depth. The distance of the ray with respect to the surface is calculated, as is the depth of the ray below the surface. Illustration 1 demonstrates this process.

Illustration 1:
A depiction of a parallax ray originating from the camera position going through a polygon surface.
The next step is to calculate what the texture coordinates would be at each point on the polygon surface directly above the ray. This requires that some form of mapping be constructed to translate 3D cartesian coordinates into 2D texture coordinates, the primary problem this paper addresses. The original Relief Mapping algorithm solved this problem through precomputed 3D surface tangents and bitangents that were oriented along the axis of the texture. Together, they could be used to calculate a texture coordinate at any point on the surface. This paper details a new, and more versatile, method of calculating the surface texture coordinates via surface fragment derivatives (See Texture Coordinate Offset Calculations, below).

Once the texture coordinate has been determined at each point on the surface corresponding to the ray, the depth map at that point is sampled and the result is compared with the depth of the ray with respect to the surface. If the depth of the ray is greater than the value of the depth map at that texture coordinate, the ray has intersected the 'surface' that is generated by the depth map. This is shown in Illustration 2.

Illustration 2:
Locating the intersection of the parallax ray and the height map.

It is worth noting at this point that the subject of finding the height map intersection
point is a topic of great interest, and a large number of different algorithms have been developed to minimize the number of height map samples. Methods range from simple linear and binary sampling to sphere-based [DONELLY2005] and cone-based [POLICARPO2008] maximal sample offset distance approximations (See Height Map Sampling Techniques and Results, below).

Once the texture coordinate at the intersection point has been determined, the last step is to use that texture coordinate for color map lookups and surface normal calculations using one of the traditional lighting and illumination models, such as Phong or Blinn. While the original method utilizes normal map lookups, it is possible to use surface fragment derivatives to dynamically generate surface normals as well (See Dynamic Surface Normal Calculations, below).

In short, the Relief Mapping algorithm uses per-pixel ray-traced texture coordinate calculations to give the illusion of depth. While this is a substantially more computationally expensive method of simulating parallax, it is far more accurate than the Parallax Mapping technique developed by Kaneko et al.
TEXTURE COORDINATE OFFSET CALCULATIONS

As mentioned above, one of the more difficult parts of the Relief Mapping algorithm is calculating the texture offset with respect to the surface coordinates. The reason for this is because it is necessary to know how the texture coordinate changes with respect to the surface position, establishing a mapping from 3D vertex coordinate space to 2D texture coordinate space.

The equations published initially by Fabio Policarpo et al. [POLICARPO2005] rely upon having a tangent and a bitangent that are both properly oriented with respect to the texture coordinates, and scaled down to the size of one texel in the direction they face. Alternatively, one could say that they orient the texture with respect to the tangent and bitangent. This is shown in Illustration 3.

Illustration 3:
Example of a polygon surface tangent and bitangent that have been aligned with respect to a texture.

Since the tangent and bitangent have already been scaled down to the distance of a single texel (as shown in illustration 4), the dot product of the tangent and parallax ray
yields the pixel offset in the vertical axis of the texture with respect to the parallax ray. The dot product of the bitangent and the parallax ray yields the offset in the horizontal axis of the texture with respect to the parallax ray.

![Illustration 4: An example of a precomputed tangent and bitangent scaled to the size of a single pixel in the texture being mapped onto the polygon.](image)

This approach doesn't work nearly as well for complex meshes when the surface tangents for each polygon are not oriented with respect to the texture coordinates, or where the tangents do not exist at all. This, in turn, is why Relief Mapping is used mostly on perfectly square quadrilaterals: most existing complex meshes do not have tangents and bitangents, much less tangents and bitangents that are properly scaled and oriented with respect to the texture for correct relief mapping texture offsets.

This 3D-to-2D mapping technique is actually the only one that I was able to find in the literature, despite extensive searches. However, while trying to come up with alternative mapping techniques, it became apparent to me that the technique described above is just a means of determining the rate of change in the texture with respect to the surface coordinates; i.e., the first derivative of the texture coordinates with respect to
vertex coordinates.

The problem with this alternative is that the vertex position and texture coordinates are already interpolated during the rasterization process before the lighting calculations take place at each fragment. In order to do this, some means must be found to pass both the vertex and texture coordinate information for each vertex of the polygon that is being rendered to the shader program. The difficulty of accomplishing this task led me to search for another alternative. I found that alternative in the rather novel use of the mip-map level-of-detail calculation functions.
PER-FRAGMENT DERIVATIVES

In the OpenGL Shading Language (GLSL), there are two functions used to determine the rate of change of a quantity across a surface fragment: dFdx() and dFdy(). These two functions calculate the derivative of a variable with respect to the change in x and y screen coordinates. They are intended for calculating the change in a texture, so that a graphics programmer can determine the correct mipmapping level to use during the texture mapping process. However, they can also be used to find the change in other variables, such as the surface position. The concept is demonstrated in Illustration 5.

Illustration 5:
A visualization of a polygon rendered to a framebuffer, showing how the texture is oriented with respect to the polygon, and the change in in the framebuffer values across a pixel (dFdx and dFdy).

The derivative of the polygon surface position in a given direction can also be thought of as a very small vector that follows the surface. This is, in effect, a tangent vector. Alternatively, taking the derivative in another direction would yield a bitangent vector.
As mentioned above, these functions are traditionally used for determining the change in textures. Because the changes in surface coordinates and the change in the texture coordinates are both given with respect to the length of the fragment, we can simply divide the derivative of the texture coordinate by the derivative of the surface coordinate. The fragment lengths cancel out, and we are left with the rate of change in the texture coordinates with respect to the surface coordinates.

\[
dFdx( ST ) / dFdx( P ) = dSTdx / dPdx
\]

It's a straightforward matter to determine the length of the surface ray vector with respect to the derivative-based surface tangent. Simply take the dot product of the ray and the tangent and multiply the result by the change in texture coordinates over the change in the tangent. Repeat the process for the bitangent. This establishes our mapping for 3D to 2D coordinates.

Once the mapping between the tangent, bitangent, and surface coordinates have been constructed, the rest of the Relief Mapping algorithm is then performed, sampling the texture coordinates at various points along the height map that correspond to the ray in order to determine the intersection location.
HEIGHT MAP SAMPLING TECHNIQUES AND RESULTS

A large number of different algorithms have been developed for the sampling component of the Relief Mapping technique. I'll discuss a few of them here, including the results of my evaluations of the various methods.

Illustration 6:
A linear parallax ray sampling technique. The linear sampling intersection point is slightly off from the correct intersection point.

The simplest method is linear sampling, where the texture map is sampled at uniform intervals along the ray (shown in illustration 6). If the sampled location is below the height map offset, that texture coordinate is the one used in the color-map when rendering the polygon color. While this has the benefit of being a very easy implementation, it causes the texture to look like it has been layered or "sliced", as shown in Illustration 7. This effect is a type of aliasing artifact, caused by the fact that the ray is sampled at linear increments, causing the texture coordinates that are chosen to “overshoot” the real texture coordinate that a more correct height map sampling method would give.
The accepted method for the search (which doesn't leave the image looking like it's been sliced) is to perform a linear search until an intersection point has been found, then go back to the previous sample point and perform a binary search along that interval segment until a more precise intersection location has been found. While this results in a correct result, it requires approximately 10% more samples.

Because of the number of samples that must be performed to achieve completely correct results, a minimum of one sample per texel followed by a binary search, the number of samples is $n + \log n$, or $O(n)$. This can certainly be improved, but not easily. Two more advanced methods include sphere and cone-based maximal sample offset approximations.

The sphere-based maximal sample offset distance approximation method detailed
in [DONELLY2005] uses the distance between the ray and the height map at each sample location to determine the distance at which it performs the next sample evaluation, based upon the principle that surfaces are generally uniform and do not involve large changes in the height across small segments of the height map. This is demonstrated in Illustration 8.

Illustration 8:
A visualization of how the sphere-based maximal sample-distance approximation algorithm works.

While the theory behind this is logical, and it reduces the search to $O(\log n)$, in practice it misses small segments with large height discrepancies very easily, as demonstrated in Illustration 9.
The cone-based maximal sample offset distance approximation method [POLICARPO2008] works on a similar principle, but with a cone shape instead of a sphere. It has a far greater chance of finding small segments with large offsets, but uses many more samples. It can be optimized to guarantee rendering accuracy via presampling the images to determine the maximum possible height offset between texels. While the theoretical speed of the cone-sampling method is \( O(\log n) \). In practice when the cone diameters are small enough to ensure complete accuracy, as every pixel must be sampled, the number of samples approaches \( O(n \log n) \).

I found during my own experimentation with various height map sampling methods that the only method that provides completely correct results regardless of height map complexity is to perform a linear per-pixel search over the uninterpolated height map, followed by a binary search over the interpolated values of the last two texels where the intersection is found in the linear search. This is, of course \( O(n) \).

One particularly interesting result that I found during my experimentation was that viewing angles had a significant effect on the number of frames per second. While I
would expect this if the number of samples was based on the number of pixels across the ray (as the ray is effectively longer at steeper angles), I did not expect it when I was using a set number of samples.

I suspect that this is because of the way that the texture lookup units in the GPU handle caching of the texture. It would make sense that, as the ray has to traverse larger quantities of the texture, the texture lookup units develop more cache misses. Here's an approximate list of FPS rates I was getting while experimenting with different sampling algorithms:

<table>
<thead>
<tr>
<th>Sampling Algorithm</th>
<th>Linear Samples</th>
<th>Binary Samples</th>
<th>Viewing Angle</th>
<th>Frames/Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>256</td>
<td></td>
<td>0°</td>
<td>~6666</td>
</tr>
<tr>
<td>Linear</td>
<td>1024</td>
<td></td>
<td>0°</td>
<td>~2000</td>
</tr>
<tr>
<td>Linear</td>
<td>256</td>
<td></td>
<td>45°</td>
<td>~2500</td>
</tr>
<tr>
<td>Linear</td>
<td>1024</td>
<td></td>
<td>45°</td>
<td>~1500</td>
</tr>
<tr>
<td>Linear + Binary</td>
<td>256</td>
<td>7</td>
<td>0°</td>
<td>~6666</td>
</tr>
<tr>
<td>Linear + Binary</td>
<td>1024</td>
<td>7</td>
<td>0°</td>
<td>~2000</td>
</tr>
<tr>
<td>Linear + Binary</td>
<td>256</td>
<td>7</td>
<td>45°</td>
<td>~2500</td>
</tr>
<tr>
<td>Linear + Binary</td>
<td>1024</td>
<td>7</td>
<td>45°</td>
<td>~1500</td>
</tr>
<tr>
<td>Sphere-based</td>
<td>N/A</td>
<td>N/A</td>
<td>0°</td>
<td>~20000</td>
</tr>
<tr>
<td>Sphere-based</td>
<td>N/A</td>
<td>N/A</td>
<td>45°</td>
<td>~10000</td>
</tr>
<tr>
<td>Cone-based</td>
<td>N/A</td>
<td>N/A</td>
<td>0°</td>
<td>~1500</td>
</tr>
<tr>
<td>Cone-based</td>
<td>N/A</td>
<td>N/A</td>
<td>45°</td>
<td>~1000</td>
</tr>
</tbody>
</table>

Table 1:
A listing comparing the approximate number of frames per second rendered while viewing a single quadrilateral occupying an entire 1024x1024 rendering buffer using various Relief Mapping sampling algorithms.
Testing was done on an Intel i5 3.3 Ghz CPU with a Radeon HD 6950 GPU, using the glman application created by Prof. Mike Bailey of Oregon State University [GLMAN]. Frames per second are approximate, as they are based on the time between frame buffer updates.
DYNAMIC SURFACE NORMAL CALCULATIONS

If the surface normal has not already been given, it's possible to take the cross product of the derivative-based tangent and bitangent vectors to get a surface normal. The only problem with this surface normal is that it is not interpolated across the surface, and consequently using it for lighting would result in a flat shading effect. Since most 3D mesh data already has surface normal information, this is of limited usefulness.

Something that is of far greater use is the ability to generate surface normals based on the height map information. This can be done by sampling the texels in the height map that are adjacent to the final texture coordinate. Using the reverse of the 3D to 2D mapping process, it is possible to go from 2D texture coordinate space for the adjacent texels to 3D vertex space, and from there generate a “relief mapped tangent” and bitangent that would correspond to the height map. Taking the cross product of this new tangent and bitangent yields a locally-accurate surface normal that can be used for illumination calculations. This “relief mapped normal” is shown in the illustration below. The tangent is shown, but the bitangent is not (as it would point out of the paper).

Illustration 10:
A diagram showing how the “relief mapped normal” is generated from points on the height map to generate a “relief mapped tangent” and “relief mapped bitangent” (not shown).
However, because this relief-mapped normal is only locally accurate, flat-shading artifacts are easily visible. There are two ways to eliminate the flat-shading effect that results from using this locally-accurate surface normal. The first method is to tessellate the surface (increasing the vertex count) and to displace newly-generated vertices that are inside the polygon in the direction of the surface normal. There are many different tessellation algorithms that accomplish this, but they are outside the scope of this document. The benefit of this method is that it is not necessary to rely upon precomputed surface normals. The cost of this method is that flat-shading artifacts are still visible, but the facets of the surface are much smaller, and it can require a huge degree of tessellation to eliminate them, dependent upon the area that the polygon occupies in screen space.

The second method, which is far easier, is to rely upon precomputed surface normals. While this increases the reliance of the algorithm on precomputed values, which is one of the primary items that this algorithm was designed to eliminate, this is not such a large setback as it might seem. Precomputed surface normals already exist in the majority of existing 3D models, and almost all major 3D modeling software packages (Autodesk 3ds Max, Autodesk Maya, Blender, LightWave 3D, Poser, Rhinoceros 3D, Sculptris, Sketchup, Solidworks, ZBrush) have the ability to generate surface normals and most file formats support them.

During the rasterization process, the precomputed surface normals are interpolated across the surface, and passed into the shader program. In GLSL, this is stored in the compatibility mode variable gl_Normal, which can be accessed in the vertex shader program and then passed into the fragment shader program. We simply take the difference between the derivative-based surface normal and the interpolated surface normal (shown in the following illustration), and add that difference to the relief-mapped normal. This process yields a surface normal that is smooth shaded across the surface of the polygon and is accurate according to the height map offsets at each position.
Illustration 11:

This shows the displacement between normals which is added to the “relief mapped normal” to create a correct surface normal.
OCCLUSION DETECTION FOR SELF-SHADOWING

The ray-tracing technique that is used in the relief-mapping algorithm can also be utilized in a similar manner to apply shadows to a relief-mapped surface. The concept is simple: trace a ray from the height map intersection point to a light source. If the ray intersects the height map during it's path to the light source, that portion of the surface is in shadow, and the lighting from that light source can be discarded. Otherwise, we can apply the desired lighting and illumination model (Blinn-Phong, Cook-Torrance, Lambert, etc.). This is repeated for each light source in the scene.

Illustration 12:
A diagram showing the basic concept of tracing a ray from the parallax intersection point to a light source. The ray to the camera (not shown) extends further to the left. The parallax intersection point shown is occluded (in shadow) because part of the relief map interrupts the ray to the light source.

The actual setup for performing self-shadowing is substantially more lightweight than the setup for relief mapping, since most of it can be re-used. The first thing we need is a unit vector ray that goes from the final surface position (calculated in the relief mapping section) to the light. Once that's done, the light ray is divided by the dot product
of itself and the actual surface normal in order to scale it to cover the entire depth of the relief map over the given ray direction. The following illustration shows what this looks like.

Illustration 13:
Diagram showing the length of the unit vector from the parallax intersection to the light source, and the same vector scaled with respect to the dot product of itself and the surface normal unit vector.

Once again, we'll need to construct a mapping from this ray to the texture. However, because we already have the derivatives, it's a similar formula as that used in the relief mapping algorithm:

\[
dSTdT \cdot (\text{LightRay} \cdot T) + dSTdB \cdot (\text{LightRay} \cdot B)
\]

Where T is the tangent vector, B is the bitangent vector, and ST are the texture coordinates. After that, we use whatever algorithm desired to traverse the ray, sampling texture coordinates at each point. If the ray ever intersects the height map, then the surface location is in shadow. The following figures demonstrate relief-mapped surfaces with and without self-shadowing.
Illustration 14:

A screen capture of a surface rendered with Derivative-Based Relief Mapping without Self-Shadowing. The light source is located up and to the right of the camera.
Illustration 15:
A screen capture of a surface rendered with Derivative-Based Relief Mapping with Self-Shadowing. The light source is located up and to the right of the camera.
RESULTS AND CONCLUSIONS

Derivative-based texture-offset calculations work nearly as well as the tangent space transformations that are used in existing relief mapping techniques. Images have been included in Appendix 1 which show a GLSL implementation of derivative-based relief mapping applied to a quadrilateral and the Utah Teapot. The GLSL code used in these examples is listed in Appendix 2.

There are a few limitations to derivative-based relief mapping. The first limitation is that the tangent and bitangent generated by the derivative functions are not completely accurate near the edges of polygons or where the angle between the camera and the surface normal approaches 90°. This results in artifacts near the edges of geometric models. One possible solution to this limitation is to reduce the maximum parallax depth, based upon the angle between the surface normal and the camera.

The second limitation is that because the tangent and bitangent are based on the change in surface position, they are not interpolated across the surface of each polygon and will be consistently the same. This will result in the texture tearing at the edge of each polygon, where the texture offsets at the end of one polygon do not match the texture offsets at the beginning of the next. While this is not noticeable when the angle between the polygons (the crease angle) is less than 10°, it is very noticeable when the crease angle is greater than 20°, as shown in the following illustration. When the shader program was applied to a icosahedron, one observer commented that it was akin to looking at an object embedded in a faceted crystal. One possible solution to this limitation is the use of tessellation shaders to add additional geometry, such as adding a vertex in the center of the polygon that is offset in the direction of the surface normal, based on surface curvature.
Illustration 16:
A depiction of texture tearing that occurs at with a high degree of parallax offset and steep angles between polygons, and the artifacts with angles approaching 90 degrees relative to the camera.
REFERENCES


APPENDIX

Parallax.glib (Glman scene specification source code):

```plaintext
Background 0 0 0
Perspective 70
LookAt 0 0 3 0 0 0 0 1 0
CubeMap 7 nvposx.bmp nvnegx.bmp \nvposy.bmp nvnegy.bmp \nvposz.bmp nvnegz.bmp
Texture2D 8 colormap.bmp
Texture2D 9 glowmap.bmp
Texture2D 10 glossmap.bmp
Texture2D 11 normalmap.bmp
Vertex Parallax.vert
Fragment Parallax.frag
Program Parallax
    Environment_Map 7 \Color_Map 8 \Emissive_Map 9 \Gloss_Map 10 \Normal_Map 11 \MaxParallaxOffset <0. .10 1.>
QuadXY 1. 1. 1 1
# Sphere 1. 32 32
# Torus 1.0 1.0 32 32
# Teapot
```

Parallax.vert (GLSL vertex shader application source code):

```plaintext
#version 330 compatibility
uniform vec4 Light0_Position = vec4( 10., 10., 10., 1. );
out vec3 EC_SurfacePosition;
out vec3 EC_CameraPosition;
out vec3 EC_SurfaceNormal;
out vec3 EC_LightPosition;
out vec2 vST;
void main( )
{
    EC_SurfacePosition = vec3( gl_ModelViewMatrix * gl_Vertex );
    EC_CameraPosition = vec3( 0., 0., 0. );
    EC_SurfaceNormal = vec3( gl_NormalMatrix * gl_Normal );
    EC_LightPosition = vec3( Light0_Position );
    // Define the texture coordinates.
    vST = ( gl_TextureMatrix[0] * gl_MultiTexCoord0 ).st;
    // Mandatory gl_Position calculation.
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
Parallax.frag (GLSL fragment shader application source code):

```glsl
#version 330 compatibility

// This controls the parallax depth. Right now the maximum depth offset is 1 OpenGL coordinate unit.
uniform float MaxParallaxOffset = 0.05;

// Set up a simple default point light.
uniform vec4 Light0_Ambient = vec4( 0.1, 0.1, 0.1, 1. );
uniform vec4 Light0_Diffuse = vec4( 1., 1., 1., 1. );
uniform float Light0_Intensity = 1.1;
uniform float Light0_Constant_Attenuation = 1.1;
uniform float Light0_Linear_Attenuation = 0.1;
uniform float Light0_Quadratic_Attenuation = 0.1;

// Set up a simple default surface material.
uniform vec4 Material_Ambient = vec4( 0.1, 0.1, 0.1, 1. );
uniform vec4 Material_Diffuse = vec4( 0.5, 0.5, 0.5, 1. );
uniform vec4 Material_Specular = vec4( 0.3, 0.3, 0.3, 1. );
uniform vec4 Material_Emission = vec4( 0., 0., 0., 1. );
uniform float Material_Shininess = 50.;

// Get texture IDs for various texture samplers.
uniform sampler2D Color_Map;
uniform sampler2D Emissive_Map;
uniform samplerCube Environment_Map;
uniform sampler2D Gloss_Map;
uniform sampler2D Normal_Map;

// Get interpolated surface information from vertex shader.
// NOTE: EC = Camera Coordinates (relative to camera).
in vec3 EC_SurfacePosition;
in vec3 EC_CameraPosition;
in vec3 EC_SurfaceNormal;
in vec3 EC_LightPosition;
in vec2 vST;

void main( )
{
   // IMPORTANT NOTE: The GLSL vector swizzle mask is .rgba, so we use .r for channel A8, .gba for channels R8 G8 B8. Just a heads up.

   float ParallaxScalar = MaxParallaxOffset;
   vec3 P = EC_SurfacePosition;
   vec3 E = normalize( EC_CameraPosition - EC_SurfacePosition );
   vec3 N = normalize( EC_SurfaceNormal );
   vec2 UV = vST;

   // Generate unit vectors using derivatives.
   vec3 T_Flat_Shading = normalize( dFdx( P ) );
   vec3 T = T_Flat_Shading;
   vec3 B_Flat_Shading = normalize( dFdy( P ) );
}
```

vec3 B = B_Flat_Shading;
vec3 N_Flat_Shading = normalize( cross( T, B ) );
if( dot( N_Flat_Shading, EC_SurfaceNormal ) < 0. ) N_Flat_Shading *= -1.;

// Calculate Sub-Surface Ray for relief map.
vec3 SSR = (-E ) / dot( -N, -E );
// Calculate Sub-Surface Ray depth and corresponding surface ray.
vec3 SR = SSR + ( N * dot( -N, SSR ) );

// Determine UV offsets for the unit vectors.
vec2 dUVdT = dFdx( UV ) / length( dFdx( P ) );
vec2 dUVdB = dFdy( UV ) / length( dFdy( P ) );

// Calculate the change in UV over SR
vec2 dUVdSR = ( dUVdT * dot( SR, T ) ) + ( dUVdB * dot( SR, B ) );

// Set up Parallax Sampling Loop
float Sample = 0.;
float Max_Samples = 128.;
float Max_Offset = 1.;
float Min_Offset = 0.;
float Curr_Offset = 0.;
float OffsetPerSample = Min_Offset + ( Max_Offset / Max_Samples );

// Perform Linear Parallax Ray height comparison sampling.
for( Sample = 0.; Sample <= Max_Samples; Sample += 1. )
{
    float SSR_H = Curr_Offset;
    float Tex_H = ( 1. - texture( Normal_Map, UV +
        ( dUVdSR * Curr_Offset ) ).r ) * ParallaxScalar;
    if( SSR_H > Tex_H ) break;
    Curr_Offset += OffsetPerSample;
}

// Perform Binary Parallax Ray height comparison sampling.
float Offset = Curr_Offset;

// Update UV texture coordinates with final offset.
vec3 SP = P + ( SR * Offset );
P = P + ( SSR * Offset );
UV = UV + ( dUVdSR * Curr_Offset );

// Recalculate the normal based surface differences at new position.
vec2 UV1 = UV + ( dUVdT * OffsetPerSample );
vec2 UV2 = UV - ( dUVdT * OffsetPerSample );
vec2 UV3 = UV + ( dUVdB * OffsetPerSample );
vec2 UV4 = UV - ( dUVdB * OffsetPerSample );

float H1 = 1. - texture( Normal_Map, UV1 ).r;
float H2 = 1. - texture( Normal_Map, UV2 ).r;
float H3 = 1. - texture( Normal_Map, UV3 ).r;
float H4 = 1. - texture( Normal_Map, UV4 ).r;

vec3 P1 = SP + ( T * OffsetPerSample ) + ( ParallaxScalar * H1 * -N );
vec3 P2 = SP - ( T * OffsetPerSample ) + ( ParallaxScalar * H2 * -N );
vec3 P3 = SP + ( B * OffsetPerSample ) + ( ParallaxScalar * H3 * -N );
vec3 P4 = SP - ( B * OffsetPerSample ) + ( ParallaxScalar * H4 * -N );

T = normalize( P1 - P2 );
B = normalize( P3 - P4 );
N = normalize( cross( T, B ) );
vec3 Normal_Difference = normalize( EC_SurfaceNormal ) -
    normalize( N_Flat_Shading );
N += Normal_Difference;
if( dot( normalize( N ), normalize( EC_SurfaceNormal ) ) < 0. ) N *= -1.;

// End of Parallax Rendering
// 2. Color Initialization //
////////////////////////////////////////////////////////////////////////////////
// Set up the color variables we need to fill out.
// NOTE: Object opacity will be replaced with refracted color.
vec3 BLACK = vec3( 0.,0.,0. );
vec3 Ambient_Color = BLACK;
vec3 Diffuse_Color = BLACK;
vec3 Specular_Color = BLACK;
vec3 Emissive_Color = BLACK;
vec3 Phong_Color = BLACK;
vec3 Refraction_Color = BLACK;
vec3 Reflection_Color = BLACK;
vec3 Final_Color = BLACK;

// Read in various values that will be used in lighting calculations.
float Shininess = 255. * texture( Gloss_Map, UV ).r; // [0,1] -> [0, 255]
float Eta = 2. * texture( Gloss_Map, UV ).g; // [0,1] -> [0, 2]
float Mirror = texture( Gloss_Map, UV ).b; // [0,1] -> [0,1]
vec3 Reflect = normalize( reflect( -E, N ) );
vec3 Refract = normalize( refract( -E, N, Eta ) );

////////////////////////////////////////////////////////////////////////////////
// 3. Per Light Calculations //
////////////////////////////////////////////////////////////////////////////////
vec3 L = normalize( EC_LightPosition - P );
Ambient_Color = Light0_Ambient.rgb * texture( Color_Map, UV ).gba;
float Light_Intensity = 1.;
vec2 LightRay = ( L ) / dot( L, N_Flat_Shading );
vec2 LightTexCoord = UV;
if( dot( L, N_Flat_Shading ) <= 0.01 ) Light_Intensity = 0.;
else
{
    float Light_H = Curr_Offset;
    for( float LightSample = Sample; LightSample >= 0.; LightSample -= 1. )
    {
        Light_H -= OffsetPerSample;
        LightTexCoord += LightTexOffsetPerSample;
        float Tex_H = ( 1. - texture( Normal_Map, LightTexCoord ).r );
        if( Light_H > Tex_H )
        {
            Light_Intensity = 0.;
            break;
        }
    }
}

float Diffuse_Intensity = dot( N, L ) * Light_Intensity;
Diffuse_Color = Diffuse_Intensity * texture( Color_Map, UV ).gba;
float Specular_Intensity = pow( max( dot( reflect( -L, N ), E ) ), 0. ), Shininess
    * Light_Intensity;
Specular_Color = Light0_Specular.rgb * Material_Specular.rgb * Specular_Intensity;
Emissive_Color = texture( Emissive_Map, UV ).rgb;
Phong_Color = Emissive_Color + Ambient_Color + Diffuse_Color + Specular_Color;

////////////////////////////////////////////////////////////////////////////////
// 4. Reflection and Refraction Calculations //
Reflection_Color = texture( Environment_Map, Reflect ).rgb;
Refraction_Color = texture( Environment_Map, Refract ).rgb;
Final_Color = mix( Refraction_Color, Phong_Color, texture( Color_Map, UV ).r );

Final_Color = mix( Final_Color, Reflection_Color, Mirror );

// NOTE: Interesting visual effects can be achieved, such as using code below:
// float Surface_Depth = dot( SP - P, EC_SurfaceNormal ) / MaxParallaxOffset;
// Final_Color.rgb = vec3( Surface_Depth, 0., 0. );

// Set the final fragment color.
gl_FragColor.rgb = Final_Color.rgb;
gl_FragColor.a = 1.;
}