Per-Pixel Displacement Mapping and Distance Maps

Daniel Book
Advanced Computer Graphics
Rensselaer Polytechnic Institute

Abstract

Based on a standard software ray-tracer with soft shadowing and glossy reflection capabilities, I present a per-pixel displacement mapping algorithm. This algorithm modifies the intersection routine already used for shadows and reflections and so they are preserved. The algorithm relies on a distance map, which along with a normal map for lighting information, can be generated for each point in the displacement map input. The distance map allows efficient calculation of dynamic sampling of the displacement map by an intersecting ray. As per-pixel displacement mapping can be implemented in a modern GPU pixel shader, this algorithm has many future possibilities.

1. Introduction

1.1 Motivation

Displacement mapping has long been a general method of mapping an object with a height field texture to render complex textures. This is an alternative to bump-mapping, which simply changes the normals on a surface to affect the lightning, giving the illusion of depth. Displacement mapping changes the actual depth of points on a surface. Usually, this is done by creating many small triangles (tessellating) to represent the displacement map on the surface. Per-pixel displacement mapping is often more computationally expensive than mesh tessellation, as well as more exact and easily refineable. Most importantly for future applications, it can be implemented in the pixel shader of modern GPUs, where the tessellation method cannot.

1.2 Related Work

The displacement mapping system presented in this paper is implemented in a ray-tracing system based on the ray tracing system presented by Turner Whitted [1] with distributed shadow and reflection samples as presented by Cook, Porter & Carpenter [2]. As only the intersection methods are modified, these algorithms maintain shadows and reflections seamlessly on displacement-mapped surfaces as well.

The displacement-mapping algorithm presented is based on both AMD's [3] and nVidia's [4] papers on per-pixel displacement mapping. The first presents extrusion to create bounding boxes for displacement map intersection. The second presents distance maps as a solution for undersampling. Both of these papers discuss implementation in hardware, but their methods are easily implemented in software. The resulting rendering is still relatively fast in software, as these algorithms are optimized for the restricted environment of the GPU pixel shader.

2. Data Structures

2.1 Height Map

The height map is essentially the representation of the actual displacement map. It is a texture of floating-point values ranging from 0 to 1, stored as a two-dimensional array. These values represent how far the face should be displaced at that point, and can be accessed directly, or interpolated for specific points on a face. In this implementation, the values are linearly interpolated for any given coordinates.

The height map can be created at any resolution of at least 2 in each dimension. The heights can be determined from a given texture file, or procedurally. I chose in my implementation to separate the dimensions of the face and the depth of displacement from the height map, as this allows a single displacement map to be applied to different faces using different depths. However, this requires each individual face to handle its own normal and distance map, as I describe below.

2.2 Normal Map

The normal map is very similar to the height map, and is created at the same resolution, but at each position stores a normalized vector indicating the normal of the displacement map at that position. Originally, I created and stored the normal map with the displacement map, but as differing depths and geometry of a face can change the normals, I found it necessary to handle normals at the individual faces so these dimensions can be accounted for.

Similarly to height map interpolation, an interpolated normal can be accessed at any coordinate within the face. Using the normal map alone, this algorithm could easily produce a bump-mapped surface; the surface would appear to be perturbed due to lighting, as determined by surface normals, but it still would be flat.

The normal map can be either read in from a file with the height map, generated procedurally alongside the height map, or generated using a brute-force algorithm based on the defined height map.

2.3 Distance Map

A common problem in per-pixel displacement mapping is sampling density. Coarse sampling can result in missing fine or sharp details, or missing the surface altogether. As a displacement map may contain sharp points, it is not simple to determine how finely the map must be sampled to adequately display the detail.

The distance map is presented by William Donnelly [4] as a solution to this problem. Alongside the height and normal maps, a 3-dimensional distance map is created. At each point in this map, the shortest distance to the height map is stored. When a ray intersects the displacement map, the point p where it intersects is queried on the distance map. This distance d is used to advance the sampling of the ray. The method works because the point $d' = p+d$ must either be the closest point on the height map to p, or has not yet intersected the height map.

I originally created the distance map in each displacement map as with the normal map. However, as different face geometry can change the relative distances between points on the displacement map, the distance map must also be created for each face, and is thus stored there. Also, I implemented distance map interpolation similarly to the height and normal maps, extended to 3 dimensions.

The distance map can be read from a file, procedurally generated, or generated using the height map. The last method (as I implemented it) results in some artifacts with certain maps, however. At sharp changes in the height field, a portion of the interpolated height map can pass closer to p than the points in the actual height map, which the brute-force generated distance map is determined from. This results in rays 'missing' the height map, so the face is not rendered at those pixels.

2.4 Convex Hull Bounding Box

To prevent the unnecessary displacement-map calculation for rays that completely miss, as well as to facilitate the distance map algorithm, a bounding box is created around each displacement map. This method is based loosely on the paper by Hirche, Ehlert, Guthe, & Doggett [3]. For each face with a displacement map, 5 faces are created. The first is simply a copy of the original face, displaced by the maximum depth of the displacement map. The vertices of this face and the original are then used to create

the four faces joining the two. These six faces are used for intersection. If a ray intersects any of them, the point at which it first intersects can then be used to determine actual intersection with the displacement map.

3. Algorithms

3.1 Linear Interpolation

All three of the data-maps used for this algorithm are accessed at arbitrary floating-point coordinates. As such, the discrete values are interpolated to allow infinite resolution. The height and normal maps are linearly interpolated in two dimensions, and the distance map in three.

This interpolation is done by first finding the four (or eight) bounding points of the desired location. If the point lies at or outside (due to floating-point rounding error) the data structure, the point is interpolated just between the edge points. The interpolation is done by multiplying each bounding point's value by a weight defined by its distance from the interpolation point.

3.2 Global/Map Translation

The three data-maps are stored as on a plane in x-y coordinate space, with z being the height value. A face, however, can be oriented in any plane and scaled in any manner. Thus, functions for translating points between global coordinates and data-map coordinates are necessary.

To translate a data-map point to a global point is simple. The proportion of the data-map point's x-y position to the resolution of the data-map is used to determine the point on the face that it represents. Then, the z-value (height on the height map of that x-y position) is multiplied by the face's normal, and the point on the face is displaced by this vector.

Translating a global point to a data-map point proved more difficult, and there is likely a clearer method than what I am using. First, the global point is projected on the face to produce a projected point. The difference between these points determines the z proportion. This projected point is then projected onto the left side of the face and its distance along the side is used as the y proportion. It is also projected onto the bottom side to determine the x proportion. These three proportional values are then used (and negated if the point is on the wrong side of the face) to determine a position in data-map space, which can then be queried for values.

3.3 Normal/Distance Map Generation

As displacement maps are not likely to be accompanied by normal and distance maps, and some procedural height maps are difficult to procedurally generate these maps for, it is important to be able to generate these using the height map. For this implementation, I used brute-force methods, which could probably be easily improved upon for both efficiency and accuracy.

To generate normal maps, I determine the global position p (using the height map) of each point in the map. Then, the global positions of two of its neighbors p_1 and p_2 are also determined. The cross-product of the vectors pp_1 and pp_2 is then used as the normal, after normalizing it and inverting it if it is opposite the direction of the face normal.

Generating distance maps is very computationally expensive in this brute-force manner. For each point p in the three-dimensional map, the distance (in world coordinates) between the height at that point and p is set as an initial distance. Then, all points within that distance on the height map are checked. If a point p' is closer to p than the current distance, the distance is updated.

This method of populating the distance map can be very slow depending on the displacement map complexity. Also, it has flaws as I detailed earlier. These are both possibilities for extension

to this technique. For accuracy, the algorithm should also check the closest points on the lines between each point on the height map, though this would make it far more expensive.

3.4 Displacement Map Intersection

As with a standard ray tracer, a ray is cast from each pixel, and each face in the scene is queried for intersection. Faces without displacement maps are forwarded to standard face-intersection routines. If a face does have a displacement map, then it and its 5 bounding boxes are tested for intersection using the same face-intersection routines. If none are intersected, the face is 'missed.'

If any face is intersected, this intersection ('hit') is passed to the displacement map intersection routine, a recursive function. The hit's position is translated to data-map space. For recursion purposes, if the hit is outside the range of the data-maps, or if too many iterations pass, the ray-cast is regarded as a 'miss.'

The hit's position is then queried against the height map. If they are (within epsilon of) equal, or the hit has just crossed the map, the displacement map intersection is terminated. The location is set as the ray's hit if it is closer to the eye point than the current hit recorded for that ray. The material properties of the face and the normal at that position in the normal map are also recorded. If it is not closer, then that hit is discarded and intersection with the displacement map terminates.

Finally, if the hit has not intersected the height map, the distance map is queried to determine how far to advance the hit along the ray. The method is then recursively called using this new sample point.

4. Results

Initially, I tested this algorithm using a simple procedural height map '*flatmap*' that simply generated the maximum height at each position (Figures 1 & 3). The normals (normal of the face) and distances (z position – maximum height) are also simply generated for this map. Using this assisted me in debugging the basic intersection routine, but several bugs remained hidden until I used more complicated maps.

The second map, '*gaussian,*' is a procedural Gaussian map (Figures 2 & 4). The heights are generated procedurally using the formula for a 2-D Gaussian curve with maximum height 1:

$$h(x,y) = \exp(-\pi*(x^2+y^2))$$

The normal and distance maps are then generated by brute-force. This map was critical for debugging the map generation functions as well as the intersection function. It did not display self-shadows, however.

For shadows, I created a third procedural map called '*studded*' (Figures 5-9). This map uses a modulus function to create alternating areas of maximum and minimum height. The normal and distance maps could be procedurally generated, but I did not implement that. This map makes painfully obvious the flaw in distance map generation as I described, but shows key functionality of the ray tracer.

As you can see in Figure 8, this clearly shows that shadowing and soft shadows still work correctly. I also used this map to show that reflections are present (Figure 7), though multiple reflections are difficult to see without a correct distance map to create a proper rendering of the edges.

See the README.txt for detailed information on the provided screenshots and rendering times.

5. Future Work

There are many possibilities for extension on this displacement map ray tracing algorithm; some have already been done.

- As in the ATI [3] and nVidia [4] papers, this could be implemented in a GPU, though the map-generation algorithms should stay CPU-based.

- The accuracy of the normal-map generation could be improved by creating surrounding faces and averaging the normals of those faces.

- The accuracy of the distance-map generation could be improved by checking the closest points on line segments (or faces) between points on the height map.

- The normal-map and distance-map generation algorithms could be optimized with other data structures.

- This could be extended relatively simply to allow displacement maps on spheres, and even on arbitrary objects, as Donnelly [4] describes.

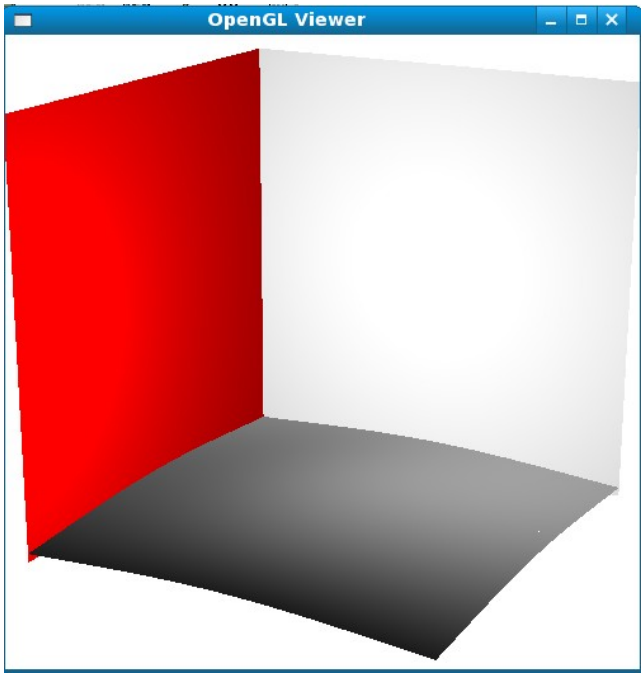

Figure 1: Gaussian with wrong normals

(Note lighting on red panel from light in lower left; compare with lighting on bottom face)

References

[1] Whitted, Turner, "An improved illumination model for shaded display," Communications of the ACM, v.23 n.6, p.343-349, June 1980.

[2] Cook, Robert L., Thomas Porter, & Loren Carpenter, "Distributed Ray Tracing," ACM SIGGRAPH Computer Graphics, v.18 n.3, p137-145, July 1984.

[3] Hirche, Johannes, Alexander Ehlert, Stefan Guthe, & Michael Doggett, "Hardware Accelerated Per-Pixel Displacement Mapping," Graphics Interface, 2004.

[4] Donnelly, William, "Per-Pixel Displacement Mapping with Distance Functions," GPU Gems 2, p123-136, 2005.

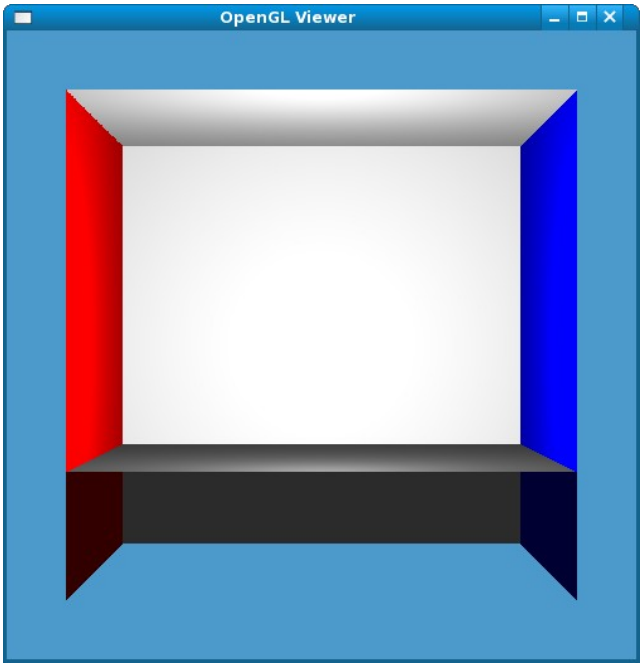

Figure 2: cornell_elevated.png

(With simple shadows, the center of the light is blocked for everything below the displaced face. Soft shadows mostly just create a lot of noise in this particular scene, due to the large light being intersected by the displaced faces.)

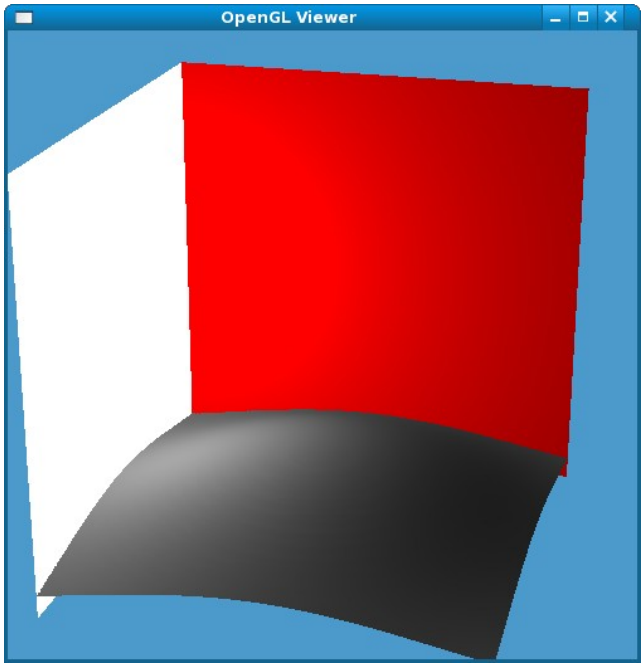

Figure 3: cornell_gaussian.png

(Although the above scene has only one instance of the displacement map and the scene below has three, the above scene took 6x as long to generate the normal/distance maps, due to the higher resolution of the map.)

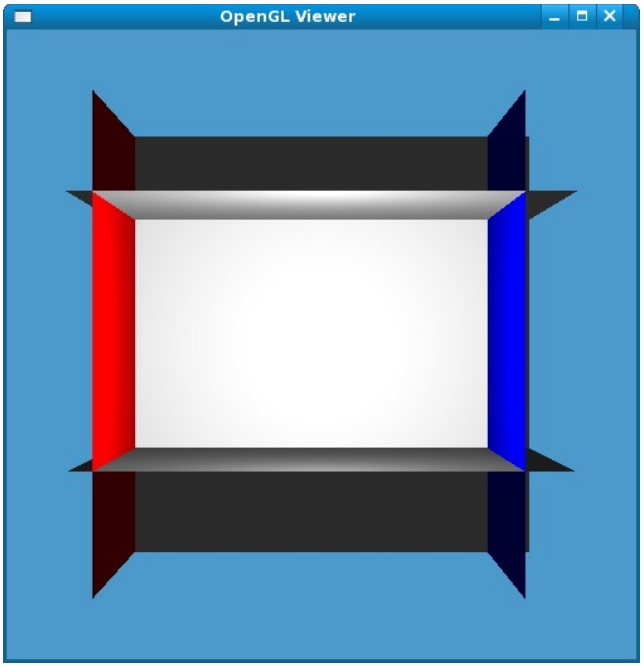

Figure 4: cornell_crossing.png

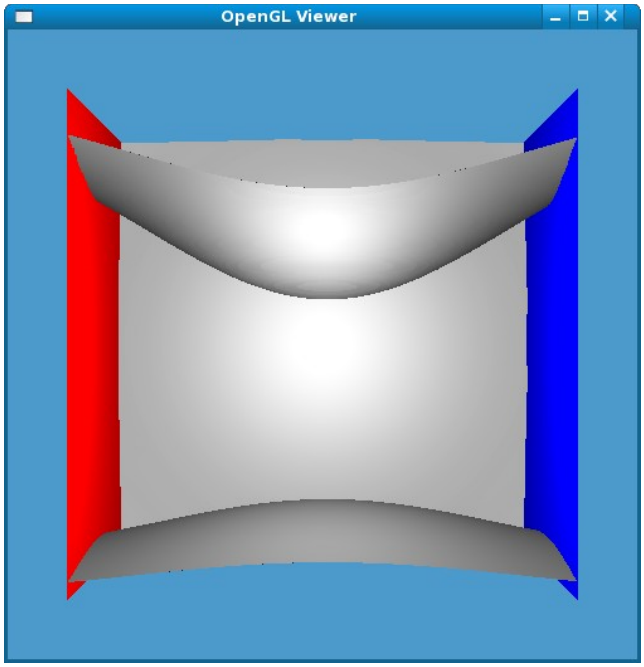

Figure 5: cornell_gausscross.png

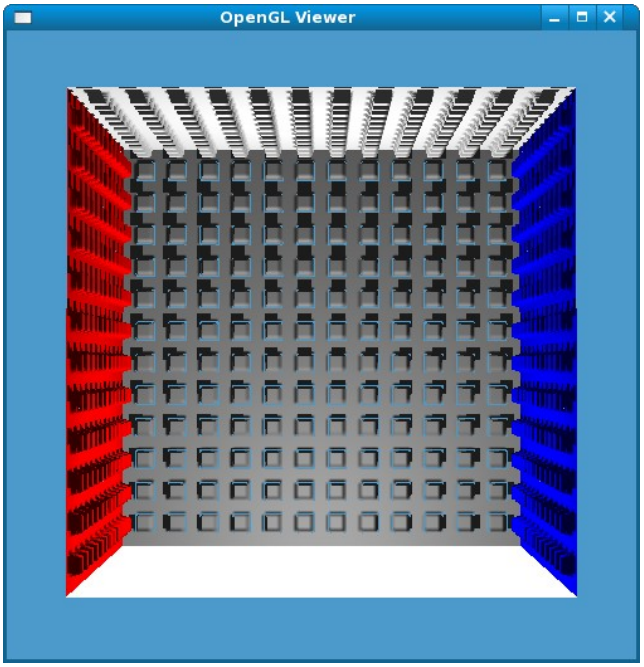

Figure 6: cornell_studded.png

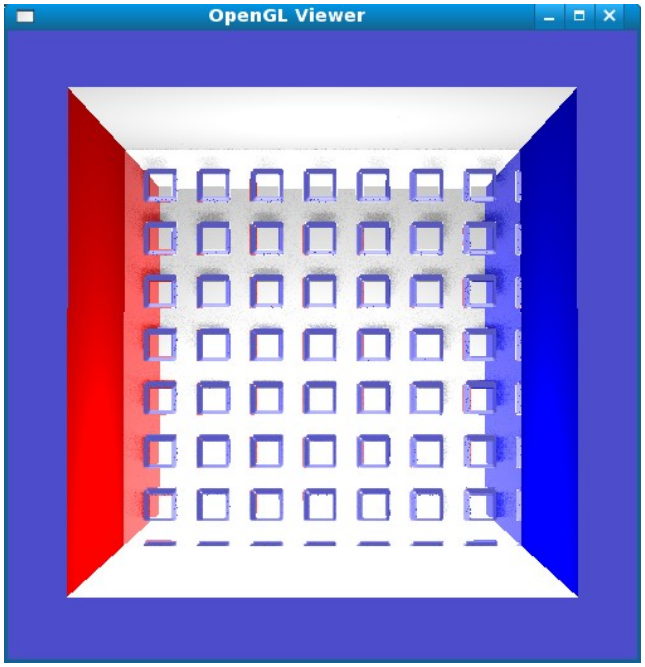

Figure 7: cornell_stud_reflection.png

(The issue with displacement map generation using only distances to discrete points is readily apparent with the studded map, as are shadows.)

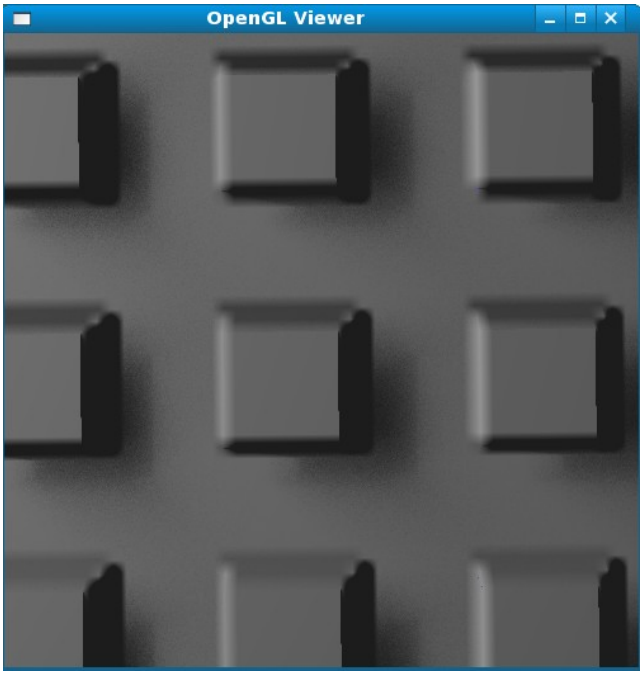

Figure 8: cornell_studded_shadows.png

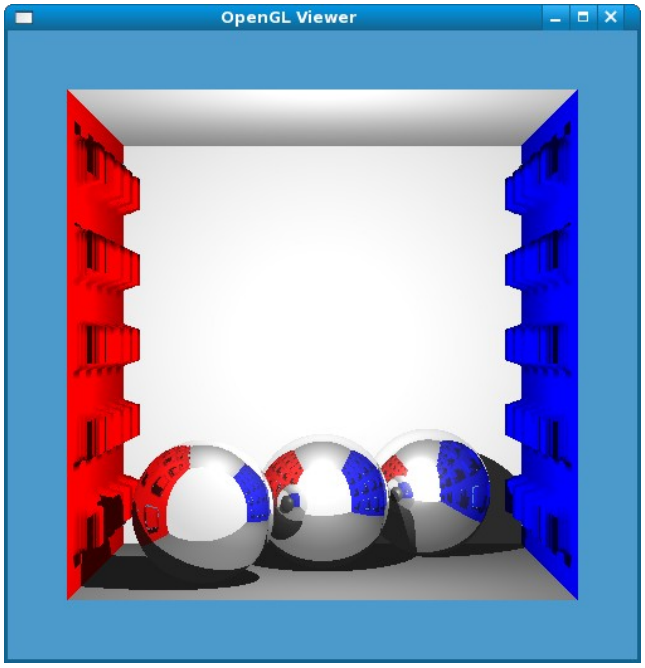

Figure 9: cornell_reflect.png