Graphics Hardware (2006)
M. Olano, P. Slusallek (Editors)

Quadtree Relief Mapping

M. F. A. Schroders[†1] and R. v. Gulik[‡1]

[1]Eximion, Eindhoven, the Netherlands

Abstract

Relief mapping is an image based technique for rendering surface details. It simulates depth on a polygonal model using a texture that encodes surface height. The presented method incorporates a quadtree structure to achieve a theoretically proven performance between $\Omega(\log(p))$ and $\mathcal{O}(\sqrt{p})$ for computing the first intersection of a ray with the encoded surface, where p is the number of pixels in the used texture. In practice, the performance was found to be close to $\log(p)$ in most cases. Due to the hierarchical nature of our technique, the algorithm scales better than previous comparable techniques and therefore better accommodates to future games and graphics hardware. As the experimental results show, quadtree relief mapping is more efficient than previous techniques when textures larger than 512×512 are used. The method correctly handles self-occlusions, shadows, and irregular surfaces.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Three-Dimensional Graphics and Realism]: Color, Shading, Shadowing, and Texture; Raytracing I.3.5 [Computational Geometry and Object Modeling]: Hierarchy and geometric transformations

Keywords: pixel based displacement mapping, relief mapping, quadtree, image based rendering, surface details

1. Introduction

Image based methods for rendering surface details are getting more and more important in real time rendering. Traditional techniques such as texture and bump or normal mapping have increased the amount of perceived detail in rendered scenes without requiring additional geometry, and are now commonly used in games [Cat74, Bli78, Coo84]. While normal mapping appropriately shades pixels according to the perceived detail geometry, it doesn't take geometric depth of the surface into account. In recent years, many techniques have been proposed that go a step further by providing motion parallax, where the surface appears to move correctly with respect to the viewer. Some of these techniques actually add detail geometry by recursive tesselation and displacement of polygons [DH00, LMH00], while others rely on image space to simulate depth in the resulting scenery (such as relief texture mapping [OBM00]). A problem with the first group of techniques, based on the displacement mapping work by Cook [Coo84], is that the rendering of a large number of small polygons is not appropriate for real time applications. The image space methods aim to render displacement maps without explicitly rendering these small polygons, while still being able to handle self-occlusions and shadows.

Parallax mapping is, to our knowledge, the first technique that allowed efficient parallax effects on simple geometry [KTI*01]. This method approximates the motion parallax effect by shifting the texture coordinates using the view vector and a height map value. When implemented on the GPU, parallax mapping is about as efficient as normal mapping, while it appears much more realistic - especially for smooth, low frequency height maps. Steep bumps, however, are rendered incorrectly and the technique suffers from swimming artifacts at grazing angles [Wel04].

Instead of just approximating motion parallax by shifting texture coordinates, more recent techniques try to deliver a result that more closely resembles actual displacement mapping. For every viewing ray, the first intersection with a height map or surface should be computed. Figure 1 illustrates the difference between normal mapping, standard parallax mapping and the result new techniques aim for.

View-dependent and generalized displacement mapping

† e-mail: m.schroders@eximion.com
‡ e-mail: r.vangulik@eximion.com

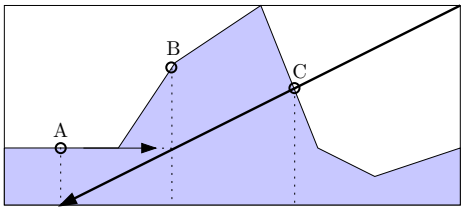

Figure 1: *When looking along the depicted ray, normal mapping gives the result at A, standard parallax mapping shows the pixel at B, and the actual intersection is at C.*

store the result of every possible ray intersection with a 3D volume in a 5D structure [WWT*03, WTL*04]. Even after compression, this requires quite a lot of storage. Also, the significant amount of preprocessing reduces the appeal of this method. For most game applications, 2D textures are preferable for various reasons.

Relief mapping [POC05, OP05] is a tangent space technique that tries to find the first intersection of a ray with a height field by walking along the ray in linear steps, until a position is found that lies beneath the surface. Then a binary search is used to more precisely locate the intersection point. Although this technique is very fast, and has been shown to support self-occlusions, shadows and even correct silhouettes, it has one large disadvantage; linear search requires a fixed step size. Therefore, to resolve small details, it is necessary to increase the number of steps. If the step size is too large, intersections with the surface can be missed as shown in Figure 2. This can result in gaps in the rendered geometry and aliasing artifacts if the frequency of the height field is too high for the sampling frequency along the viewing ray.

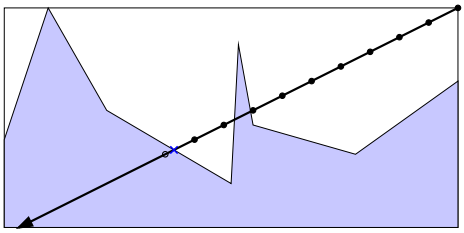

Figure 2: *Linear search can miss the first intersection with a surface if the step size is too small.*

Steep parallax mapping [MM05] tries to solve this problem by making the step size smaller than a pixel, but a more elegant solution would be to automatically adjust the step size as necessary. The distance mapping algorithm by Donnelly does exactly that by implementing sphere tracing [Don05]. His algorithm utilizes a distance map, in which for every point the shortest distance to the surface is stored. This distance is used to make a potentially large step along the ray, without overshooting the surface (see Figure 3). Unfortunately, the distance map is a 3D texture, with an average size of 512×512 pixels and a height of 16 to 32 layers.

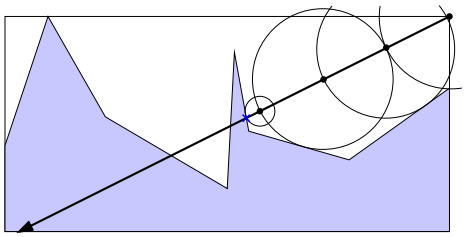

Figure 3: *With sphere tracing, the step size depends on the minimal distance from a point to the surface. This distance can be precomputed for any point in the enclosing volume. Sphere tracing is guaranteed to find the first intersection of a ray with a surface.*

In this paper we present a relief mapping variation that also takes large steps along the ray without overshooting the surface, but without requiring a 3D texture. This is achieved through the use of a quadtree on the height map and yields a method that has time complexity between $\Omega(\log(p))$ and $\mathcal{O}(\sqrt{p})$, where p is the number of pixels in the used texture. The storage requirements are only slightly higher than those of relief mapping (we need $1\frac{1}{3}$ times as much space to store all levels of the quadtree).

From a geometric point of view, the problem of finding the intersection of a ray with a surface boils down to ray shooting on axis-oriented polyhedra. Each point on the height map can be seen as a box which extends upward a certain height. For this problem, $\mathcal{O}(\log(p^2))$ query time can be achieved by using $\mathcal{O}(p^2)$ storage [Pel93]. Unfortunately, this method is useless for implementation on graphics hardware due to its storage requirements. In the same paper, Pellegrini gives a trade-off between storage space and query time. Using $\mathcal{O}(m)$ space, for $p^{1+\varepsilon} \leq m \leq p^{2+\varepsilon}$, he obtains a query time of $\mathcal{O}(\frac{p^{1+\varepsilon}}{\sqrt{m}})$ [Pel93]. For (almost) linear space, this would be a $\mathcal{O}(\sqrt{p^{1+\varepsilon}})$ query time. Although the method described in this paper does not significantly improve this result theoretically, our method is practically faster and easily implementable on graphics hardware using pixel shader version 3.0. Note that sphere tracing, steep parallax mapping, distance mapping and our method all have a worst-case query time of $\mathcal{O}(\sqrt{p})$, while Policarpo's relief mapping method has a constant query time because of the fixed number of steps. However, if we relate the step size of his method to the size of the texture used (to reduce aliasing), it also has a $\mathcal{O}(\sqrt{p})$ query time. As the results show, our relief mapping method approaches a running time of $\log(p)$ in most cases.

2. Quadtrees

Quadtree relief mapping is based on the observation illustrated in Figure 4. Assume we are given a subsegment s of a

viewing ray r, and a surface encoded in a height map. If we know that the maximum height of the part of a surface below s is lower than the minimal height of any point on s, then s lies completely above this surface. Thus, if we are searching the first intersection of r with the surface, we can jump over s knowing that this subsegment does not intersect the surface.

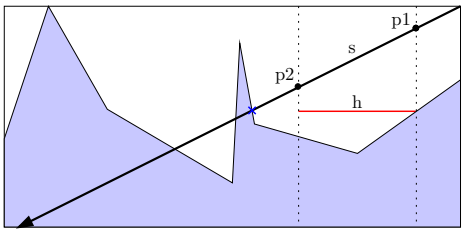

Figure 4: *Subsegment s of the viewing ray depicted above extends from p1 to p2. The maximum height h of the part of the surface that lies below s is indicated by a horizontal line segment. Since s lies entirely above this line, we can safely jump from p1 to p2 when searching for the first intersection.*

The remaining question is how to determine which subsegments of the viewing ray should be used. To approach logarithmic running time in any algorithm, an iteration should be able to potentially reject half of the remaining options. This leads to the use of a space subdivision scheme on the height map. We use a quadtree [Sam84] to handle ray height map intersections. Each node of the quadtree stores the maximum height value for the corresponding subtree (i.e. the maximum height of the corresponding part of the surface). This means that the lowest level of the quadtree stores the complete height map, and the root of the quadtree stores the maximum height over the entire surface.

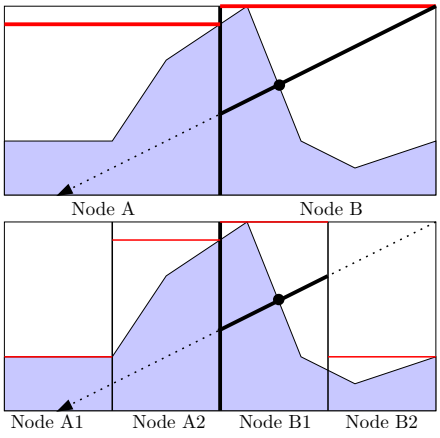

Figure 5: *The side view of 2 levels of a quadtree on a height map, together with the encoded surface. Maximum heights are indicated by red, horizontal line segments. The bold part of the viewing ray in the upper and lower picture is associated with respectively node B and node B1.*

Figure 5 shows a side view of the quadtree on maximum height values. Every node in the quadtree stores the maximum height of the corresponding part of the surface encoded in the height map. The four children of a node are each associated with one fourth of its associated surface. In the same way, a subsegment of the viewing ray can be associated with a node (as illustrated in Figure 5).

To find the first intersection of a ray with the surface, the quadtree is used as follows. When considering a node, we are actually asking whether the view ray can intersect the part of the surface that corresponds to this node. If the maximum height value stored in this node is lower than the lowest point on the associated subsegment of the viewing ray, this is clearly not possible. Thus, it is safe to jump over this node and the corresponding subsegment of the view ray.

However, if there *does* exist a point on the associated subsegment of the viewing ray that goes below the stored maximum, we need to investigate this node further. One or more children of this node have to be visited. Children are visited in front to back order to ensure that the first intersection with the surface is found. In Figure 5, children of node A as well as node B have to be visited, starting with node B. Node B2 can be safely skipped, so B1 is visited next. The algorithm stops when the first intersection is found.

If the viewing ray is (nearly) vertical, we only have to visit one node in every level of the quadtree. So under the best possible circumstances, the intersection can be found in logarithmic time in the number of pixels in the height map ($\mathcal{O}(\log(p))$). However, if the viewing ray is close and parallel to the surface, we may have to enter a significantly larger number of nodes. Consider a ray that lies diagonally over the surface. In the worst case such a ray needs to enter twice the square root of the number of nodes in every level of the quadtree. This implies that the total number of visited nodes is at most $\mathcal{O}(\sqrt{p})$.

3. Algorithm

This section explains the algorithm that calculates the intersection of the view ray and the surface encoded in the quadtree relief map. The algorithm is designed to take advantage of graphics hardware. It requires a start position and a direction (the view ray) and a preprocessed height map. The starting point is the point where the view ray intersects the actual geometry (i.e. the top plane of the box enclosing the surface), in tangent space. The required quadtree relief map is similar to a mip mapped version of the height map texture. But instead of taking the average over 2×2 pixels, the maximum is calculated. This way the value of a pixel equals the maximum over a certain area, which corresponds to the highest point in that area. Every pixel in the resulting texture represents a node in the quadtree. The highest level in the quadtree has only one pixel, which stores the maximum height of the surface.

The algorithm uses a cursor that points to the current position along the view ray. Initially this cursor is at the starting point (defined above). The cursor divides the view ray in two segments. The first segment, from the startpoint up to the cursor, does not intersect with the surface. Our goal is to advance the cursor as far as possible in each iteration, until the intersection with the surface is found.

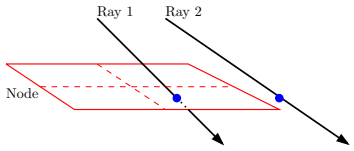

Figure 6: *Plane - Ray intersection. The height of the plane is the maximum height of the surface associated with the current node. Ray 1; continue in the lower right child. Ray 2; jump over this node and continue in a neighbouring node.*

We start at the root node and proceed as follows. At each node there are two options; either we go down the tree into one of the four children, or we jump to a neighbouring node. The decision depends on the maximum height associated with the current node. Figure 6 shows a plane drawn at the height stored in the node. This plane is intersected by the view ray. Ray 1 intersects the plane within the area of the node (in the lower right child), but ray 2 does not. Each observation has its own effect on the new position of the cursor.

In the first case, the cursor is moved to the intersection point of the view ray and the plane. Since the plane is at the maximum height of the area associated with the node, no intersection can occur above the plane. Therefore, moving the cursor down to this height is guaranteed to be safe. Now the algorithm has to continue in the correct child. This is easy, because the cursor encodes exactly where we are in any level of the quadtree. Regardless of the level, the cursor is always in the correct node. Each level of the quadtree corresponds to a texture, and for each texture the coordinates range from 0 to 1 even though the texture sizes are different. So going up and down in the quadtree boils down to selecting the correct texture. For memory reasons, the textures for all levels are still put together into one texture, which is twice as large as the original height map texture.

In the other case, the ray lies above the plane inside the boundary of the current node. The cursor is advanced to the boundary, since the intersection is not inside this node. Instead of moving up the tree to the common ancestor of the current node and its neighbour (which a raytracer would do), the algorithm continues directly in the adjacent node. Although both methods produce the exact same result, we have empirically found that the second approach is faster.

3.1. Boundaries

The boundary of the current node is needed to determine the new position of the cursor. It is used to determine whether a point lies inside the node, and if not, the cursor is moved to the boundary itself. To avoid calculating the intersections of the ray with all four boundaries of the node, the direction of the ray is taken into account. Figure 7 shows how this is done. Depending on the quadrant the direction of the ray points to, two boundaries are chosen to intersect with. In quadrant A this would be the right and upper boundaries (1 and 4). This results in two intersections (t_1 and t_2), of which the nearest is chosen as the new cursor position.

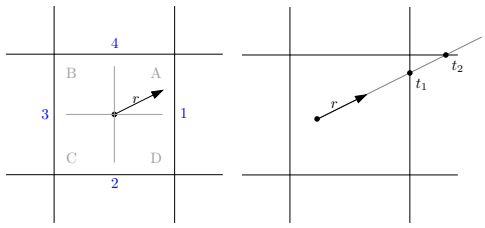

Figure 7: *The direction of the view ray determines with which boundaries we have to calculate an intersection (line 1 and 4 in this case). The intersection closest to the current position, t_1, is used as the new cursor position.*

When choosing the boundary as the new position of the cursor, we have to add a small offset. The boundary between two nodes does not belong to any node. By adding a small displacement, the cursor gets a correct new position.

4. Implementation

Pseudo-code for the algorithm is listed below.

Algorithm 1: ***findIntersect***$(ray, texCoord, reliefMap)$

1: $cursor \leftarrow texCoord$
2: $startPoint \leftarrow texCoord$
3: $quadrant \leftarrow (\text{sign}(ray) + 1) \text{ div } 2$
4: $delta \leftarrow \frac{1}{2 \cdot textureWidth}$
5: $level \leftarrow 0$
6: $t_{cursor} \leftarrow 0$
7: **while** $level \neq \log(textureWidth)$
8: $\quad height \leftarrow reliefMap[cursor, level]$
9: $\quad t \leftarrow \frac{height}{ray_z}$
10: $\quad bound \leftarrow \lfloor cursor \cdot 2^{level} + quadrant \rfloor$
11: $\quad t_{bound} \leftarrow \frac{\frac{bound}{2^{level}} - startPoint}{ray_{xy}}$
12: $\quad t_{min} \leftarrow \min(t_{bound_x}, t_{bound_y})$
13: $\quad t_{cursor} \leftarrow \max(t_{cursor}, \min(t, t_{min} + delta))$
14: $\quad cursor \leftarrow startPoint + ray_{xy} \cdot t_{cursor}$
15: $\quad$ **if** $t < t_{min} + delta$ **then**
16: $\quad\quad level \leftarrow level + 1$
17: $\quad$ **end if**
18: **end while**
19: **return** $cursor$

The first six lines are initialization, partly to speed up the algorithm. Lines 8 and 9 calculate the intersection with the plane set on the height found in the current node. The boundary is calculated in parametric representation in lines 10 - 12 (note that the division by ray_{xy} is component-wise). The observation of whether the cursor is in- or outside the current node is partly made in line 13, and partly in the if-statement. Regardless of whether the cursor is in- or outside the node, its new position is the furthest point along the ray that is known to be safe. This is either the closest boundary point, or the calculated intersection with the plane. Line 13 makes sure that when the cursor enters a neighbouring node that stores a higher height value, the cursor continues further down in the tree (instead of jumping back up). In line 14 the new position of the cursor is calculated. Next, an if-statement decides whether we need to go further down the tree. If not, we start in the neighbouring node in the next iteration.

Note that the dynamic while-loop is only available in graphics hardware that supports pixel shader model 3.0. However, we noticed that decent results are still obtained when the dynamic loop is replaced by a fixed loop with about $\log(p)$ iterations. For example, for a height map with 1024×1024 pixels, 20 iterations is enough.

5. Discussion

The quadtree relief mapping method does not function correctly when standard mip mapping with filtering is enabled on the height map texture. The problem is that the average operator does not propagate over the maximum operator. Therefore, a mip mapped version of the height map would no longer represent a valid quadtree relief map. Note that the levels of the quadtree relief map itself can be stored as different layers in a mipmap texture. We expect that using these mipmap layers, instead of bilinearly filtered mipmap layers, will give acceptable results.

A significant improvement both in visual result as in the number of iterations required would be to implement oriented quadtrees. Looking again at Figure 5 it is obvious that instead of the horizontal plane, an oriented plane that rests on top of the surface could also be used. For an oriented plane, there are four height values: one for each corner. Instead of using 1 color in the height map, all RGBA channels could be used to encode the four different heights. The benefits of needing fewer iterations to find the intersection are likely to outweigh the costs of extra calculations inside each iteration.

To add self shadowing to the algorithm, essentially the same ray-heightfield intersection method can be used by shooting a ray from the light source toward the computed intersection point. If this ray intersects the surface encoded in the height field, the point lies in shadow [POC05]. Further, it would be interesting to find out whether correct silhouettes can be obtained in a similar way as presented in [OP05].

6. Results

The algorithm used in quadtree relief mapping is guaranteed to find the first intersection of a viewing ray with a surface encoded in a height map. This significantly reduces artifacts on highly irregular surfaces (Figure 10). Still, the technique works with a 2D texture that can easily be generated from a standard height map. Because of the hierarchical approach, the algorithm scales well to large height maps. The measurements in Table 1 were made using a height map that encoded a highly irregular surface, as shown in Figures 9 and 10, and give an average framerate for a range of viewing angles. For less irregular surfaces our technique is even faster, since the number of steps performed depends on the frequency of the height map. The number of steps used in relief mapping is taken from Policarpo's paper for 256×256 textures. If the texture dimensions are twice as large, the number of linear steps is doubled and one additional binary step is performed.

	Parallax mapping	Relief mapping	Quadtree relief mapping
256×256	360	188	95
512×512	360	84	82
1024×1024	357	48	71
2048×2048	354	<9	63

Table 1: *Average fps for viewing angles ranging from top down to grazing, on two Geforce 6600GT cards in SLI mode.*

While the amount of fragments that need to be rendered determines how many times the pixel shader has to be executed, the amount of time spent for one fragment is higher when a larger height map is used. Developers are starting to use normal and texture maps of 2048x2048 resolution for large parts of their game content [Epi05]. Although textures of this size are mostly atlases, it makes sense for image based rendering techniques that aim to improve such games visually to support increasing texture sizes.

We showed that an intersection query of a ray with the height map takes between $\Omega(\log(p))$ and $\mathcal{O}(\sqrt{p})$ time, where p is the number of pixels in the height map. In practice, most intersection queries only need a logarithmic number of steps, as demonstrated in Figure 8. Figure 9 shows how quadtree relief mapping performs for various depths. Finally, Figure 11 compares our method with relief mapping with respect to the perceived height.

References

[Bli78] BLINN J. F.: Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (Aug. 1978), vol. 12 (3), pp. 286–292.

[Cat74] CATMULL E. E.: *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, Dec. 1974.

[Coo84] COOK R. L.: Shade trees. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (July 1984), Christiansen H., (Ed.), vol. 18, pp. 223–231.

[DH00] DOGGETT M., HIRCHE J.: Adaptive view dependent tessellation of displacement maps. *Eurographics workshop on graphics hardware* (2000), pp. 59–66.

[Don05] DONNELLY W.: *Per-Pixel Displacement Mapping with Distance Functions, in GPU Gems 2*. Addison Wesley, 2005.

[Epi05] EPICGAMES: Unreal 3 technology overview. `http://www.unrealtechnology.com/html/technology/ue30.shtml`, 2005.

[KTI*01] KANEKO T., TAKAHEI T., INAMI M., KAWAKAMI N., YANAGIDA Y., MAEDA T., TACHI S.: Detailed shape representation with parallax mapping. In *Proceedings of the ICAT* (2001), pp. pp. 205–208.

[LMH00] LEE A., MORETON H., HOPPE H.: Displaced subdivision surfaces. In *Proc. of the Computer Graphics Conference 2000* (jul 2000), ACMPress, pp. 85–94.

[MM05] MCGUIRE M., MCGUIRE M.: Steep parallax mapping. *I3D 2005 Poster* (2005). `http://www.cs.brown.edu/research/graphics/games/SteepParallax`.

[OBM00] OLIVEIRA M. M., BISHOP G., MCALLISTER D.: Relief texture mapping. In *Proc. of the Computer Graphics Conference 2000* (jul 2000), pp. 359–368.

[OP05] OLIVEIRA M. M., POLICARPO F.: *An efficient representation for surface details*. Tech. rep., 2005.

[Pel93] PELLEGRINI M.: Ray shooting on triangles in 3-space. *Algorithmica 9* (1993), pp. 471–494.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J. L. D.: Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D* (2005), pp. 155–162.

[Sam84] SAMET H.: The quadtree and related hierarchical data structures. *ACM Computing Surveys 16*, 2 (June 1984), pp. 187–260.

[Wel04] WELSH T.: Parallax mapping with offset limiting: a per pixel approximation of uneven surfaces. `http://pds1.egloos.com/pds/1/200603/10/62/parallax_mapping.pdf`, Jan 2004.

[WTL*04] WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: Generalized displacement maps. In *Proceedings of the 2004 Eurographics Symposium on Rendering* (june 2004), pp. 227–234.

[WWT*03] WANG L., WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: View-dependent displacement mapping. In *ACM TOG* (2003), vol. 22 (3), pp. 334–339.

Figure 8: *Quadtree relief mapping using a 512×512 height map. White means not found within 10, 15 and 25 iterations.*

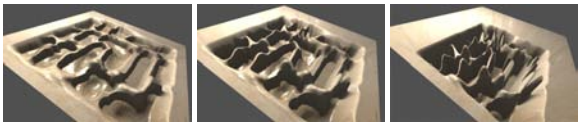

Figure 9: *Quadtree relief mapping using various heights.*

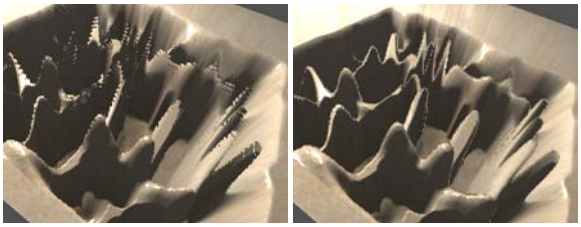

Figure 10: *Artifacts in relief mapping (left) and quadtree relief mapping (right) when rendering an irregular surface.*

Figure 11: *Relief mapping (top) and quadtree relief mapping (bottom). Note the differences in perceived height when looking at the screws from a grazing angle.*