Rendering Geometry with Relief Textures

Lionel Baboud, Xavier Décoret

Rendering Geometry with Relief Textures

Lionel Baboud* Xavier Décoret†

ARTIS-GRAVIR/IMAG-INRIA‡

ABSTRACT

We propose to render geometry using an image based representation. Geometric information is encoded by a *texture with depth* and rendered by rasterizing the bounding box geometry. For each resulting fragment, a shader computes the intersection of the corresponding ray with the geometry using pre-computed information to accelerate the computation. Our method is almost always artifact free even when zoomed in or at grazing angles. We integrate our algorithm with reverse perspective projection to represent a larger class of shapes. The extra texture requirement is small and the rendering cost is output sensitive, so our representation can be used to model many parts of a 3D scene.

CR Categories: I.3.1 [Raytracing]; I.3.1 [Color, shading, shadowing, and texture];

Keywords: heightfield, ray-tracing, GPU

1 INTRODUCTION

A large part of the budget in a game is dedicated to hire artists that will model visually appealing environments. This includes creating textures and shaders (*appearance* modelling) and populating the world with enough details (*geometric* modelling). The texturing and shading capacities of modern graphics cards allow for very complex appearances; there is virtually no limit other than creativity. On the other hand, artists must refrain from creating too detailed environments for current graphics card. This results, even in the most recent games, in flat looking surfaces, polygonalized silhouettes and noticeably "empty" environments. Several techniques have been proposed to overcome this limitation. Bump mapping [2] simulates the interaction of light with uneven surfaces, providing a feeling of relief, but has obvious errors at grazing angles. It is consequently limited to small scale relief. Another solution is to concentrate the available geometric budget onto nearby parts and use coarser *level of details* (LOD) for distant parts or barely visible objects [14]. Once again, this yields artifacts at oblique angles. For example, if distant buildings are rendered by "flattening" all architectural details into a texture-mapped cube, the viewer will see embossed parts such as window panes. View-dependent level of details adress this problem but require data structures and algorithms that are too costly.

As an alternative to geometry, *Image Based Rendering* (IBR) uses reference "views" of parts of the model, either statically or dynamically generated, to synthetize the current views [10]. Most methods are intended for automatic replacement of distant parts and are not meant as a modelling tool. IBR can involve costly and non trivial pre-processing and, although very efficient results have been published [11], they these methods suffer from artifacts (disocclusion, flickering) or limitations (memory requirements, integration with standard geometry) that games cannot afford. Following that trend, recent works introduced the idea that geometric information can be stored not only with vertices and faces but also within textures either using transparency [5] or depth information [18, 19].

*Lionel.Baboud@inrialpes.fr

†Xavier.Decoret@inrialpes.fr

‡ARTIS is a team of the GRAVIR/IMAG laboratory, a joint effort of CNRS, INRIA, INPG and UJF.

2 PREVIOUS WORK

The observation that details are inefficiently represented by many small triangles has led to the use of *displacement maps*[4]. Points on a surface are moved along their normal by an amount specified by a texture that is mapped onto the surface. With the exception of terrain, displacement maps are typically used for adding small scale detail such as bumps.

To render displacement maps, the surface can be adaptively retesselated [16]. Another approach is to render a bounding volume and perform a ray intersection with the local height field over the base surface [9]. Wang et. al proposed precomputing of all such intersections by sampling the 5D space of rays [24]. The result is compressed and encoded in a way that allows real-time texturing of arbitrarily curved surfaces, with self-shadowing and complex lighting [25]. Even with compression, the method requires about 4 MB texture memory for a 128x128 pattern, so it is aimed at visualizing one object and cannot be applied to all surfaces of a large scene. Moreover, the method is intended for mesostructure rendering as the sampling process cannot capture large variations in the displacements. Kautz et al[13]. uses slices the heightfield and uses alpha test to render each slices. Policarpo et al.[19] perform approximate ray/height field intersection using a binary search. The result is theoretically incorrect despite an initial phase to alleviate some of the problematic cases. In practice, the method works very well with visually appealing results and real-time performances for small to medium scale relief. It can also be used to compute better reflection and refractions than with simple environment mapping[22].

Instead of finding what part of the displaced surface is seen at a given pixel location, Oliveira et al.[18] determine where the elements of the displacement map should project on screen. Based on depth, they pre-warp the texture for the current viewpoint and map the result to a bounding surface. This guarantees correct parallax effects, automatic hole filling with interpolation and proper filtering. The main drawback of pre-warping is that it is not output sensitive: whatever the screen space projection, the pre-warping traverses the whole texture.

Another alternative is parallax mapping[12], optionally with offset limiting[26], where texture coordinates are shifted based on the height to produce approximate parallax. The method is not exact but yields visually pleasing results for reliefs like bricks, rough surfaces, etc.

In several of these works, displacement maps are also extended to store more information than a single depth value. Policarpo et al. [19] uses dual-depth textures to to encode a front and a back surface displaced over a base patch. Oliveira et al. [18] use 6 relief texture on the face of a cube to render complex objects such as a statue. Parilov et al. [21] combines pre-warping with LDI so that they can represent objects with a higher depth complexity such as

trees.

3 Rendering algorithm

We start with a 3D model, place a bounding box around it and project the geometry on the bottom face to generate a *texture with depth*. Each texel stores a color and a normalized distance. We define normalized coordinates by mapping the bounding box to the unit cube of the projection.

Rendering is done by drawing the bounding box. For each fragment produced, we determine the ray from the eye to the corresponding point on the bounding box and we intersect it with the heightfield. For that we walk along the ray until it passes below the heightfield[1]. The problem of ray/heightfield intersection has been well studied for terrain rendering[2] and many solutions and optimization data structures have been proposed. The problem here is that the intersection must be computed in a fragment shader, which constrains our choice of data structures and algorithms. An increasing amount of work has also been dedicated to generic ray-tracing on GPU [20, 3, 7] but they do not take advantage of the particular case we are considering.

3.1 The heightfield representation

The heightfield is defined as the bi-linear interpolation of values sampled at the center of texels. Lines joining these centers are called *centerlines*, lines between texels are called *borderlines*. An important point is that the intersection of the heightfield with a plane perpendicular to the ground is guaranteed to be piecewise linear only if the plane aligns with a centerline. Otherwise, the intersection can indeed contain curved parts as shown on Fig. 1. This

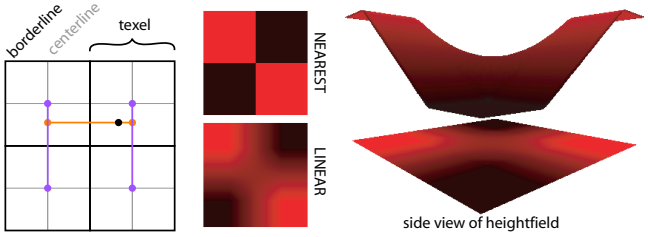

Figure 1: Bilinear interpolation yields curved heightfields

point will be important later.

3.2 Binary search for intersection

In [19], the proposed solution to quickly find an intersection is a binary search. In most cases, this method finds the correct intersection but there is no guarantee it will. In particular, the ray is sampled independently of its tilt which amounts to slicing the heightfield regularly along the vertical direction. To alleviate this, the ray is first walked by a few fixed-size steps. This does reduce the number of false intersections but some will always remain. This will particularly be the case if the heightfield contains large depth discontinuities or narrow peaks and when it is viewed at a grazing angle. Since [19] mostly demonstrates their techniques on small scale relief, this does not show up. But for more general heightfields, the artifacts can be very noticeable (Fig. 11). They can be reduced by increasing the number of steps of the preliminary walk, but this reduces the benefits of the binary search. Moreover, using fixed steps has a fundamental flaw: it both misses potential information and performs redundant tests (Fig. 2 (a)). To find the correct intersection, we must consider all texels covered by the projection of the ray.

[1]The heightfield can always be chosen to be 0 on its boundary so every ray entering the bounding box starts above the heightfield.

[2]`www.vterrain.org`

3.3 Rasterization-based intersection

The problem of identifying the intersection is actually that of rasterizing the projection of the ray in the texture. We found the simplest way to visit all texels is a 2D version of the algorithm proposed in [1]. The current position on the ray is parameterized by an abscissa t. Starting from the entry point $t = 0$, we find the abscissae $T.x$ and $T.y$ of the next intersection with a horizontal or a vertical centerline (Fig. 2 (b)). We move t to the minimum value and then update $T.x$ and $T.y$. This update simply requires adding to $T.x$ or $T.y$ the difference of abscissa $d.x$ or $d.y$ between two consecutive intersections with horizontal and vertical borders. These $d.x$,$d.y$ values are constant and are computed once before the loop.

This approach to walk the ray gives much better results (Fig. 11). Nevertheless, it is not exact. The problem is that we assume the heightfield varies linearly along the ray between two centerlines which is not the case as discussed in Section 3.1. Thus, we can miss intersections such as the one shown on Figure 3. The error

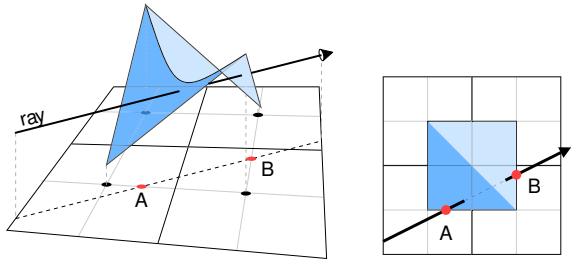

Figure 3: A and B are above the heightfield which does not vary linearly along $[AB]$. Thus the rasterization-based approach will miss the intersection with the saddle shape.

incurred, however, is usually very small except for pathological heightfields. Apart from this problem, the approach performs the optimal number of texture lookups for arbitrary heightfields. The shader is simple and fast, but is not as fast as a binary search. The latter requires roughly $\log n$ steps while the former takes n steps. Another problem is that we must traverse all texels along the ray. This can be expensive if the ray traverses a large empty region before intersecting the heightfield. Thus, views at grazing angles with large textures have reduced performance.

3.4 Pre-computed robust binary search

The conclusion of the two previous sections is that the binary search is fast and texture independent, but the line rasterization is more accurate if the only information available is the heightfield. Thus we propose to analyze the heightfield in a preprocess step and to store extra information in the depth texture. Similar ideas has been proposed by Donnelly[6], who uses distance functions to accelerate the rendering of displacement maps.

We define the *safety radius* as a function $r(x, y, \theta)$ which gives a lower bound on the distance to the second intersection, if any, for unblocked rays. In other word, unblocked rays can have at most one intersection with the heightfield in the neighborood defined by r (Fig. 4).

The safety radius is used to find intersections in the following manner. Assume the current position along the ray is (x_t, y_t, z_t) and is above the heightfield. Instead of moving a fixed amount dt, we advance an amount corresponding to $r(x_t, y_t, \theta)$. By property

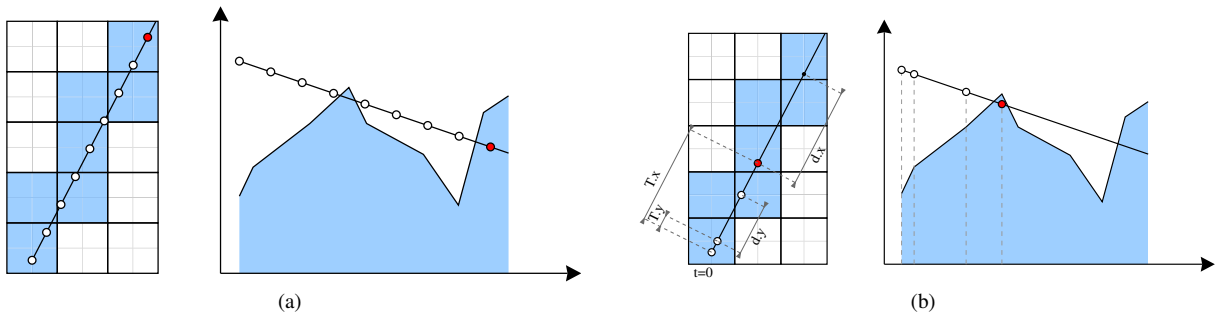

Figure 2: (a) Walking a ray by taking fixed steps yields redundant lookups and missed feature, no matter the sampling rate, as can be seen by considering a ray arbitrarily close to a texel's center (b) In Amanatides and Woo[1], the ray is walked from one centerline to the next; no texel is missed and the correct intersection is found.

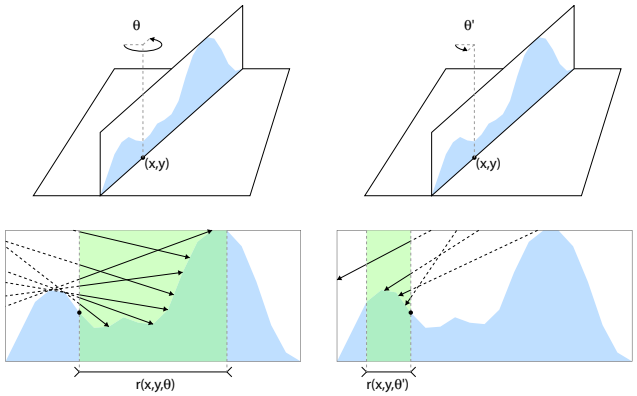

Figure 4: *The safety radius $r(x,y,\theta)$ indicates a region in which rays passing above pixel (x,y) with direction θ can have at most one intersection with the heightfield.*

of the safety radius, there can be at most one intersection between positions t and $t+dt$. If the new position is above the heightfield, there is no intersection and we keep advancing. If it is below, there is exactly one intersection between t and $t+dt$ and we run a binary search to find it. Figure 5 shows an example.

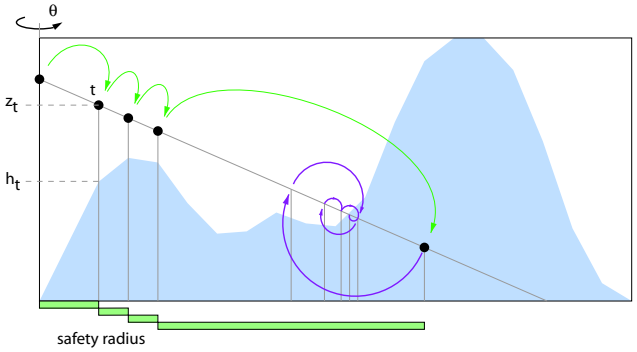

Figure 5: *Example of robust binary search: from left, we walk along the ray of amounts corresponding to safety radii (green steps) as soon as we pass below the heightfield, we start a binary search (purple steps).*

We encode a conservative discrete 2D version of the safety radius in a 2D texture. For a texel (i,j), the safety radius is now a number of pixels n such that any ray, whose projection crosses the centerlines within the texel, has at most one intersection with the heightfield within the $2n \times 2n$ square centered on (i,j). As seen in Figure 6, this square is supported by centerlines.

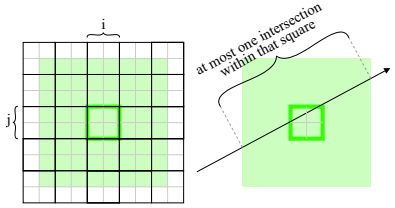

Figure 6: Conservative safety radius. For texel (i,j), the safety radius is 2 meaning any ray crossing the texel has at most one intersection with the heightfield in green square.

The intersection algorithm is as follows. We first determine if the ray is more horizontal than vertical by comparing $d.x$ and $d.y$ where d is the direction of the ray. Suppose the ray is horizontal $(d.x > d.y)$. We walk backwards on the ray to the first intersection with a vertical centerline. Then we fetch the safety radius r for the corresponding texel. If it is non zero, we advance by r vertical centerlines. We keep doing this until we pass below the heightfield at which point we run a binary search.

The case of zero radius is special. It occurs when the heightfield is locally non concave because we can always find a ray that has two intersections with the local peak (Fig. 7). The problem with

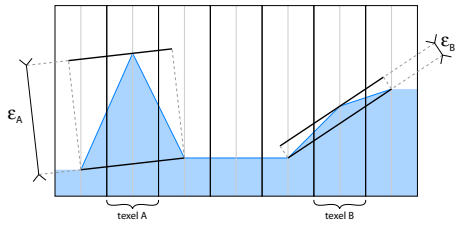

Figure 7: For texel A and B, the safety radius is 0 as we can clearly find rays intersecting the heightfield twice arbitrarily close to the texel's centers. Setting a non zero radius will create incorrect silhouettes of size ε.

zero radius is that we can no longer advance our position on the ray. There are two ways of dealing with this. The first one is to move "manually" to the next texel by performing up to two iterations of the exact algorithm. This solution yields exact computations but the code for the loop becomes more complex and there is a performance penalty. The other solution is to clamp the radius to 1. This

introduces an error along the silouhettes but we can bound the error. It is given by the local gradient of the heightfield as shown on Figure 7. We will give more details on this error in next section.

3.5 Computation of the safety radius

For a given texel, we need to consider all rays that pass above it. For each ray, we compute the distance to the second intersection with the heightfield and take the minimum of these distances. Note that we ignore rays that are blocked by the heightfield *before* they pass above the texel (Fig. 8). We heavily sample the directions of

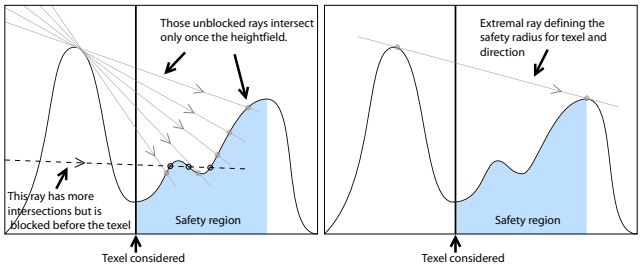

Figure 8: (left) blocked rays are ignored when computing the safety radius (right) the safety radius is defined by the extremal ray.

rays. For each direction, we compute the safety radius and keep the maximum over all directions. For a given direction, the problem is restricted to a 2D slice in which the safety radius is obtained by considering an *extremal ray*. It is the "most horizontal" ray that does not intersect the heightfield before the considered texel and which is tangent to the heightfield after the texel with the point of tangency at a minimal distance (Fig. 8). Clearly it is defined by the two points of tangency. We determine this ray by starting with the tangent at the considered texel and iteratively moving the two tangent points while maintaining the tangency. The algorithm is simple, fast and can be implemented on the GPU. Our implementation takes about 10s to compute the safety radii for a 128×128 heightfield.

Our approach is very close to that of [19]. We first describe the elements involved and analyse the problem of ray/heightfield intersection. We then describe our approach for fast accurate intersection.

4 Reverse perspective heightfields

In the previous section, we did not specify the projection used to flatten the geometry into a depth texture. The most natural projection is an orthogonal one, resulting in a heightfield that's easier to understand. However, the algorithm does not actually place any constraint on the kind of projection used, as long as a ray in world space is transformed into a line. This can be used to increase the expressiveness of relief mapping.

If we try to replace a building with an orthogonally projected relief texture, we will not get any information about the facades. This is a well known problem with image based rendering: some information is not captured. Several methods have been proposed to address this problem[15]. Reverse perspective consists in shooting an image with an inverted frustum so that shortfortening of objects works the other way. In cubist textures, [8] proposed to use it to get textures that capture details on the sides of buildings. We incorporated their approach in relief mapping.

Instead of placing a bounding box around the geometry, the user manually defines a reverse frustum around the object and this frustum is used for projection (Fig. 9). The rendering algorithm is barely changed: instead of rendering the bounding cube, we render the frustum, and we pass the corresponding perspective projection

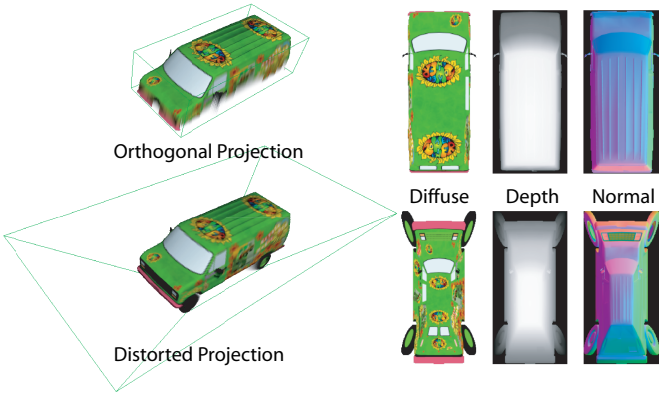

Figure 9: Reverse perspective heightfield (right) better captures the shape of the van than orthogonal one (left).

matrix to the frustum to the shaders. This matrix is used to compute the normalized coordinates of fragments on the frustum, and of the eye. The remainder of the algorithm being expressed in the unit cube does not change.

4.1 Clipped frustum

Reverse perspective heightfields can replace more complex geometry. However, using it naively incurs a performance penalty. Indeed, the frustum is typically much larger than the bounding box, so more fragments are rasterized. But for many of these fragments, the corresponding ray does not actually intersect the geometry. This is very simply addressed by cutting the bounding box out of the frustum and rendering it with the appropriate normalized coordinates. This is done as a preprocess, does not change the shader at all, and brings the performance rates back to those of the orthogonal projection.

4.2 Multiple heightfields

Most objects are not globally representable as heightfields, but very often they can be quite faithfully represented by the combination of several heightfields. Oliveira et al. uses 6 relief textures mapped on a bounding box to replace objects like statues[19]. We use a similar approach with our reverse perspective heightfield except that we split a bounding cube in 6 perspective frustums.

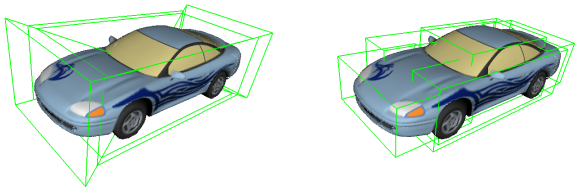

Figure 10: (left) Using 6 reverse persepective heightfields to represent an object (right) clipping the reverse persepective heightfields to reduce fillrate.

4.3 Independent resolutions

The intersection search depends only on heights. Once it is found, color is obtained with a color-texture lookup. Depth and color tex-

tures can therefore have different resolutions. In practice, we use a larger resolution for color for rich visual appearance and smaller resolution for depth so that rendering is fast.

4.4 Interaction with the Z-buffer

Since we render the bounding box, the z-buffer contains the depth of fragments "on the bounding box". Outputting the correct depth for the fragments is trivial as this depth is known during the ray walk. It just requires adding a line to the shader. In most cases, however, this is not necessary and even not desirable. Indeed, the problem *a priori* is that rendering the bounding box in the depth buffer would hide more things than what is actually hidden, because rays to a point on the bounding box are not necessarily blocked by the heightfield. Fortunately, modern GPUs allow us to discard fragments when no intersection is found. Thus our algorithm fills the z-buffer with an incorrect but conservative depth. If no object penetrates the heightfield's bounding box, visibility will be correctly resolved. The advantage about not altering the depth within the fragment shader is that this is detected by the driver, which optimizes the pipeline and avoids evaluation of the shader for eventually hidden fragment. Thus, when the bounding box is partially occluded, we save computations for the hidden fragments.

5 Results and discussion

We implemented the different algorithms discussed in this paper using a GeForce 6800GT. Using distorted heightfields and higher resolution color texture results in renderings of surprinsingly high fidelity (Fig. 11). Thus, as already foreseen in [18], relief textures can be used to replace complex geometry. For that statement to hold, we must evaluate the cost of that representation. Table 1 gives the rendering time for various models and texture resolution.

	mesh Hz	size. pixels	exact Hz	robust Hz	binary Hz
terrain	800	32^2	115	116	109
		64^2	87	105	109
		128^2	58	97	110
		256^2	36	96	108
grid	800	32^2	102	110	110
		64^2	72	93	109
		128^2	46	79	109
		256^2	27	73	108
car	450	32^2	110	113	109
		64^2	67	105	109
		128^2	44	98	110
		256^2	25	90	106
camera	345	32^2	70	84	101
		64^2	44	72	100
		128^2	25	67	101
		256^2	14	65	101
van	350	32^2	104	105	109
		64^2	77	102	110
		128^2	52	98	110
		256^2	30	96	107

Table 1: Framerates for the different approaches. The bounding box covers about 800×600 pixels and is viewed at 45°.

As expected, binary search is the fastest of the three methods, except for very low texture resolution (due to the fixed steps preceding the binary search but one would probably deactivate them for such resolutions). Moreover, its cost is directly proportional to the fill rate, but independent of the texture resolution. However, as shown on Figure 11, the result obtained is potentially incorrect. These artifacts are particularly noticeable when the model is moving.

Our proposed method alleviates this problem at the cost of slower rendering times. The exact rasterization is between 3 and 4 times slower for the viewing angle we chose. In our experiments, it is even slower for grazing angles, but faster for a front view. For rays almost perpendicular to the ground plane, the intersection with the heightfield is very close to the one with the plane, so it is found immediately. Thanks to shader's early exit, our approach is faster than the binary search, which always performs the same number of operations. The key observation is that it is highly dependent on both the view angle *and* the texture resolution. The first dependency is unavoidable by the nature of our approach. The dependency on texture resolution is adressed by the robust binary search. When the resolution is doubled, the safety radius is also doubled and thus the number of iterations to find the intersection for a given ray remains the same, depending only on the shape of the heightfield underneath the ray (apart from a fixed cost to initialize the algorithm, which explains the small variations observed in Table 1).

The robust binary search is almost as fast as the binary search, has much fewer artifacts (Fig. 11, but is still not exact. As we saw, when the safety radius is zero, we set it to 1 to guarantee that the shader terminates, which potentially produces errors. The good point is that these errors are localized at silhouettes which makes them less noticeable. Moreover, this error is controllable and the user can trade quality for speed; allowing larger errors yields larger values of the safety radius which decreases the average number of iterations to find ray/heightfield intersections (Fig. 12).

The first observation of this timing analysis is that binary search should be used for small scale bump mapping as it is the fastest and produces no noticeable artifacts in that case. However, when larger geometry is used, the robust approach has better performance and visual quality.

The second observation, however, is that rendering the mesh is always faster for the examples we tested. This is due to several facts. First, we chose quite a large screen size (800×600) for viewing the heightfields so that we have enough dynamics to compare framerates. In practice, our representation would rather be used for objects with a smaller screen size. In such cases, our examples are as fast as the mesh, and for facade models that we tested they even exhibit less aliasing artifacts (small triangles produce a lot of flickering that is naturally filtered by our image based rendering). One would argue that for distant models, simplified meshes could also be used, but the key point is that such meshes cannot capture parallax effects, especially at grazing angles.

The second reason is that the meshes we chose were not very detailed (about 8000 polygons). This is typically due to the fact that artists are trained to produce low polygon count models for real-time rendering. Our representation can actually capture details corresponding to the texture resolution. For a 256×256 texture, this would equal a 64K mesh which would render slower than our heightfield representation. Note that our rendering approach is not limited to replace meshes. It can be used directly as a modelling primitive, with the artist "painting" the height field in the spirit of [17]. The last reason is that current graphic cards are amazingly fast at processing vertices but still have limited performance for complex fragment shaders that employ branching and dependent texture lookups (which we use). However, manufacturers report that future improvements will be on the fragment processing speed rather than of vertex processing, increasing the competitiveness of heightfield rendering.

The cost of our representation is simply the texture storage. However, since meshes are typically textured too, we do not take into account the color texture. In heightfield representation, we

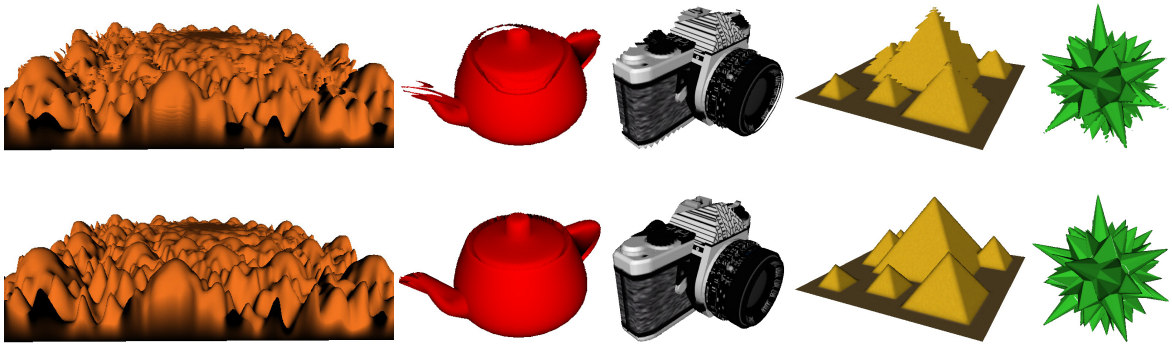

Figure 11: Artefacts with binary search (top) are avoided using our robust approach (bottom).

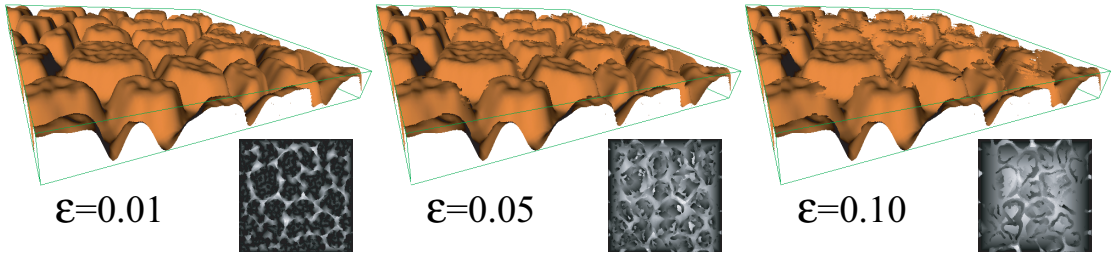

Figure 12: Varying the error allowed yields larger safety radii (grayscale images) meaning faster rendering but potentially more artifacts along silhouettes

need to store a depth, a normal and a safety radius per texel, that is 5 floats. In comparison, a mesh of the same resolution requires 6 floats per vertex, 3 for the position and 3 for the normal[3]. Thus the ratio is of 5 to 6 in our favor. The depth and normal can be packed in a single RGBA texture. Thus a 128×128 heightfield requires 320K if a 32 bits-per-channel float texture is used. In practice, one would typically use a 8 bits-per-channel and the cost would be 80K. This could be reduced further through quantization of the normals [23] and/or the depth.

6 Conclusion and Future Work

We presented an analysis of how to efficiently render relief textures without artifacts on a modern GPU. We focused on GPU friendly methods that are both efficient and compact. That is, they do not use overly costly pre-computed acceleration structure or that do not fit well on the hardware. From that analysis, we designed a quasi exact approach which is sufficiently fast and produces visually appealling renderings of complex models. We showed, in particular, how to incorporate distorted heightfields to capture more complex shapes. Thus, relief textures can be used not only to render uneven surfaces but also to replace complete objects, for example in the context of level of detail. An important limitation of our method is that we do not handle geometric aliasing. Using mipmapping for the color texture can be a solution to avoid visual artifacts. However, mipmapping the heightfield is not a solution (simply averaging heights does not make sense). Furthermore, finding the intersection of a heightfield with a cone, instead of a ray, is not trivial.

In the future, we would like to investigate automatic analysis of geometrically modelled scenes to determine when geometry should be replaced by heightfields. For this representation to be fully integrated with classical representations, we must also study how to integrate it with shading techniques. For the moment, we perform simple phong shading using normal maps. Integration with shadow rendering and in particular self-shadowing must be investigated. Finally, we want investigate deferred shading to avoid the computation of ray-intersections for pixels that will be later on covered by another object.

[3] Although x, y coordinates of the vertices are on a grid, they must be given if the mesh is to be stored on board, using a vertex buffer object for example.

Acknowledgments

We would like to thank Thierry Dezarmenien for the models, the reviewers for their fruitful comments, Karim Kochen for initial work and the Artis members for their support.

References

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10. Elsevier Science Publishers, 1987.

[2] James F. Blinn. Simulation of wrinkled surfaces. In *Proc. of SIGGRAPH'78*, pages 286–292. ACM Press, 1978.

[3] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[4] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press.

[5] Xavier Décoret, Frédo Durand, Franĉois X. Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. *ACM Trans. Graph.*, 22(3):689–696, 2003.

[6] William Donnelly. Per-pixel displacement mapping with distance functions. In Matt Pharr, editor, *GPU Gems 2*. Addison-Wesley, 2004.

[7] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[8] Andrew J. Hanson and Eric A. Wernert. Image-based rendering with occlusions via cubist images. In *Proc. of VIS '98*, pages 327–334. IEEE Computer Society Press, 1998.

[9] Johannes Hirche, Alexander Ehlert, Stefan Guthe, and Michael Doggett. Hardware accelerated per-pixel displacement mapping. In *Proc. of GI'04*, pages 153–158. Canadian Human-Computer Communications Society, 2004.

[10] Stefan Jeschke, Michael Wimmer, and Werner Purgathofer. Star: Image-based representations for accelerated rendering of complex scenes. In *Proc. of Eurographics*, 2005.

[11] Stefan Jeschke, Michael Wimmer, Heidrun Schumann, and Werner Purgathofer. Automatic impostor placement for guaranteed frame rates and low memory requirements. In *Proc. of I3D'05*. ACM Press, 2005.

[12] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. 3-dimensional shape representation with parallax mapping. *Human Interface Society*, 4(1):91–96, 2002.

[13] Jan Kautz and Hans-Peter Seidel. Hardware accelerated displacement mapping for image based rendering. In *GRIN'01: No description on Graphics interface 2001*, pages 61–70, Toronto, Ont., Canada, Canada, 2001. Canadian Information Processing Society.

[14] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2002.

[15] C. Mei, V. Popescu, and E. Sacks. The occlusion camera. *Computer Graphics Forum*, 24(3), 2005.

[16] Kevin Moule and Michael D. McCool. Efficient bounded adaptive tessellation of displacement maps. In *Proc. of GI'02*, pages 171–180, 2002.

[17] Byong Mok Oh, Max Chen, Julie Dorsey, and Frédo Durand. Image-based modeling and photo editing. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 433–442, New York, NY, USA, 2001. ACM Press.

[18] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *Proc. of SIGGRAPH'00*, pages 359–368. ACM Press, 2000.

[19] Fáabio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proc. of I3D'05*, pages 155–162. ACM Press, 2005.

[20] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA, 2002. ACM Press.

[21] W. Stuerzlinger S. Parilov. Layered relief textures. *Journal of WSCG*, 10(2):357–364, 2002.

[22] László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum (Proc. of EG'05)*, 24(3), 2005.

[23] M. Tarini, P. Cignoni, C. Rocchini, and R. Scopigno. Real time, accurate, multi-featured rendering of bump mapped surfaces. In M. Gross and F. R. A. Hopgood, editors, *Computer Graphics Forum (Eurographics 2000)*, volume 19(3), 2000.

[24] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Trans. Graph.*, 22(3):334–339, 2003.

[25] Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. Generalized displacement maps. In *Proc. of the Eurographics Symposium on Rendering*, 2004.

[26] Terry Welsh. Parallax mapping with offset limiting: A per-pixel appproximation of uneven surfaces. `www.infiscape.com/doc/parallax_mapping.pdf`, 2004.