Fast Per-Pixel Displacement Mapping with Screen-Space Depth Interval Grids

John Judnich and Nam Ling

Abstract—In this paper, we present a novel space skipping algorithm for per-pixel displacement map rendering on the GPU that significantly accelerates ray convergence. Without any preprocessing, this technique achieves raw displacement map rendering performance two to four times as fast as other techniques without loss of visual quality. Displacement mapping algorithms accurately render complex shapes projected within flat geometry by casting rays from the viewer into a tangent-space height-field; this enables otherwise unachievable resolutions of 3-D detail in games and simulations. Our algorithm accelerates ray convergence by first computing ray boundary intervals within screen-space uniform pixel blocks. For 4x4 pixel blocks, this computation is one sixteenth the complexity of ordinary ray casting. The final pass then uses the dynamically computed depth intervals to converge on a precise intersection point very rapidly. As a result, our algorithm enables rendering of raw displacement map data significantly faster than existing algorithms. Moreover, this technique can provide performance improvement to other existing per-pixel displacement mapping algorithms as well, with or without preprocessing.

Index Terms—Computer graphics, Computer science, Convergence, Information geometry, Geometry, Layout, Image quality, Image reconstruction, Pipelines, Ray tracing, Reconstruction algorithms, Rendering (computer graphics), Surface texture, Surface treatment, Virtual reality

1 Introduction

Displacement mapping adds high frequency geometric detail (*mesostructure*) to otherwise smooth geometry (*macrostructure*) by applying a tangent-space heightfield (usually referred to as a *displacement map*). Displacement mapping deforms an object's geometric shape by offsetting points in a direction normal to the surface at varying magnitudes specified by the displacement map [1] [2]. This way, complex high-resolution optionally self-shadowing [3] [4] [5] geometry can be added to formerly smooth surfaces, achieving photorealistic [6] rendering of rough or "bumpy" objects like small gravel, textured wood, and brick walls, for example [7].

Vertex displacement mapping is the most straightforward method to apply displacement maps, where the macrostructure geometry is subdivided (producing many more triangles), vertices displaced appropriately, then rasterized [8]. This works well for very large displacements that need to interact with the z-buffer (e.g. terrain geometry), but tends to produce prohibitively many triangles (millions to billions) when applying extremely fine displacement maps to surfaces.

Adding very fine high resolution mesostructure is often more efficiently achieved with *per-pixel displacement mapping*, where the displacement is applied in reverse by the fragment shader after rasterization [9] [10]. This entails computing ray-heightfield first intersections for each pixel rasterized from the macrostructure, in order to determine where the view ray would have struck had the geometry actually been displaced [11].

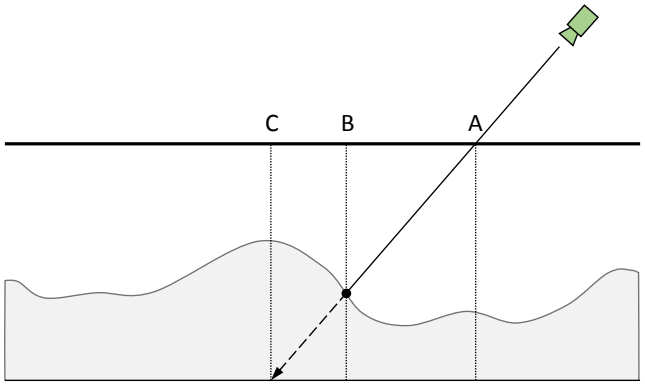

Fig. 1. Ray-heightfield intersection including entry point (A), intersection point (B), and exit point (C).

This process is illustrated in Figure 1. The top bold line represents the macrostructure geometry's surface. The bottom surface represents the "bottom" of the heightfield (height values of zero). Key points include the entry point A (where the ray meets the macrostructure surface), the exit point C (where the ray leaves the heightfield pseudo-geometry), and the first intersection point B. Note that for efficiency and simplicity on the GPU, ray intersection algorithms usually operate in *tangent space* (surface coordinates referencing pixels in a texture

• *J. Judnich and N. Ling are with the Department of Computer Engineering, Santa Clara University, Santa Clara, CA, 95050. E-mail: jjudnich@scu.edu, nling@scu.edu*

map) rather than world or object space [12] [13]. A more detailed theoretical background can be found in [14].

Per-pixel displacement mapping is particularly desirable because the computational complexity is easily made proportional to the number of pixels on-screen, rather than the resolution of the displacement maps applied. This enables the appearance of extremely high resolution geometry exceeding one triangle per pixel. Accomplishing the same results with actual triangular geometry performs very poorly in comparison, and tends to suffer from unpleasant aliasing artifacts (since most rendering pipelines are not optimized to render triangles taking less than one pixel on-screen) [15]. As a result, games and simulations often use per-pixel displacement mapping to enable otherwise unachievably high resolutions of 3-D detail.

In this paper, we present a novel approach to per-pixel displacement mapping on the GPU, where we take advantage of ray penetration depth coherence in screen-space to more efficiently converge on the first ray-heightfield intersection. Section 2 provides background by describing existing algorithms, Sections 3 to 7 present our algorithm, and Section 8 analyzes and compares performance under various conditions.

2 Related Work

The subsections below are divided between GPU algorithms that need only the raw heightfield data, and GPU algorithms that require preprocessed datasets. If one can afford the time and memory overhead, preprocessing generally enables better performance and visual quality. (Our algorithm can benefit both categories.)

2.1 Preprocessing-Free Methods

When only the raw heightfield data is available to the fragment shader, most displacement mapping algorithms are fundamentally based on brute-force *linear search*, if not by direct volume rendering [16].

Linear search starts sampling height values at the ray entry point and proceeds in small steps towards the exit until the ray penetrates below the heightfield [17] [18]. Without sufficiently small steps however, linear search is prone to aliasing artifacts where subsequent iterations will overshoot small or sharp geometric features [19].

Faster converging approximations exist, but on their own are useless at grazing view angles. While they excel at quickly finding an intersection point, the likelihood of it being the *first* intersection rapidly diminishes as the depth complexity (or amount of self-occlusion) increases. This leads to unacceptable warping artifacts, tearing, and holes that occur for viewing directions not nearly perpendicular to the surface. As a result, most of such techniques are used as a second stage refinement after a linear search.

Among these fast-converging approximations are *Parallax Mapping* [20] [21], *Iterative Parallax Mapping* [22], *Binary Search* [23], *False Position Method* and *Secant Method* [24], etc. All use some kind of binary interval reduction or approximate line intersection to rapidly converge on an intersection assuming at most one intersection point exists in the interval (i.e. no self-occlusion).

Relief Mapping [25] [23] performs a coarse linear search, followed by a binary search. The linear search locates the interval in which the first intersection should occur, and the binary search converges quickly to the exact intersection point. *Parallax Occlusion Mapping* (POM) [26] [27] [28] extends this by dynamically adapting the linear search step size depending on each ray's angle relative to the macrostructure surface. POM improves performance as surfaces become more perpendicular to the view vector, but degenerates to Relief Mapping at grazing angles.

2.2 Preprocessing-Based Methods

Among the best preprocessing-based algorithms are *Pyramidal Displacement Mapping*, *Sphere Tracing*, *Cone Step Mapping*, and *View-Dependent Displacement Mapping*.

Generating a Pyramidal Displacement Map [29] simply involves computing a mipmap pyramid (a feature supported by GPUs where textures are stored at multiple resolutions), where pixels are combined with a maximum operator rather than by averaging. The lower mip levels allow the ray to advance very quickly, before converging more precisely using the next higher mip level (space skipping similar to [30]). This repeats until it reaches the highest resolution data. However, rays will often enter a higher mip level falsely, only to determine no intersections occur within. Handling this consistently can become complex and slow. *Quadtree Displacement Mapping* (QDM) is a variant of this algorithm which handles such cases by degenerating into a linear search. QDM trades guaranteed computational complexity for much better average-case performance on the GPU.

Sphere Tracing involves considerable memory overhead and preprocessing time, as it expands the height map into a 3-D volume texture [31]. Each voxel specifies a one byte distance value representing the maximum radius of a sphere one could center at the voxel's location without intersecting any heightfield geometry. The algorithm proceeds like an adaptive linear search, where a guaranteed safe step size is provided by the voxel's radius value [32] [33]. Unfortunately, the memory overhead for Sphere Tracing (commonly 32 to 64 times the memory of regular displacement maps) is usually prohibitive for most practical applications.

Cone Step Mapping (CSM) computes a supplementary "cone map" consisting of one byte per pixel which stores the maximum angle of an upright inverted cone (vertex down) placed on the heightfield geometry without intersecting [34]. (Some variants of CSM adapt the cone in slightly different ways [35] [36], but the general principle remains the same.) Like Sphere Tracing, this creates bounding volumes which can be used to adaptively advance the ray test at guaranteed safe step sizes until

the first intersection is found. Unfortunately, computing accurate CSM data for moderately large heightfield textures can take many hours on a quad core CPU. Moreover, a well-tuned Parallax Occlusion Mapping implementation tends to perform nearly as well as CSM with similar visual quality [37]. As a result, CSM is not often used in practice.

View-Dependent Displacement Mapping (VDDM) takes a full approach to precomputation by defining a displacement map as a 5-dimensional function containing precomputed ray intersection results for any possible input ray [38]. This look-up-table approach to displacement mapping enables a constant-time solution for each pixel's ray. Moreover, a compression algorithm is provided to help mitigate memory consumption. Unfortunately, even when employing extensive compression [39] [40], using VDDM even for medium resolution displacement maps requires far too much video memory for most practical applications.

3 Algorithm Overview

Our algorithm exploits the screen-space coherence of ray penetration depth to drastically narrow the first intersection candidate interval. This is accomplished with a two-pass render process on the GPU.

In the first pass, we consider the screen divided into a grid of blocks of $n \times m$ pixels. For each block, we want to compute and store in a temporary buffer the minimum and maximum ray depths that may be encountered. We call this buffer the *Depth Interval Grid* (DIG).

The second pass reads from the Depth Interval Grid, using the min and max ray depths within each pixel block to confine the search interval. For most pixels, this interval will be very small, in which case a simple binary search will suffice in finding the exact intersection. For larger intervals, an adaptive confined version of any regular per-pixel displacement mapping algorithm (such as relief mapping) may be used to accurately find the intersection within the provided interval. An example of bounded ray intervals is shown in Figure 2.

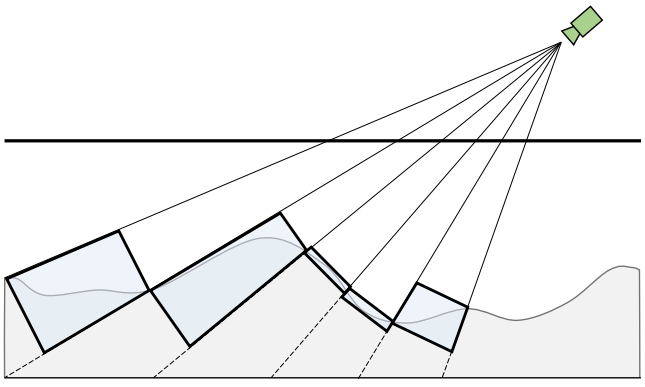

Fig. 2. A 2-D cross-section (side view) illustrating an extremely low resolution Depth Interval Grid projected into a scene.

Without preprocessing, it is not possible to determine minimum and maximum ray depths with guaranteed accuracy any faster than a regular displacement mapping algorithm. However, it is possible compute a very good approximation which runs in $r/(n \times m)$ time, where r is the time to render the displacement map ordinarily, and $n \times m$ is the number of pixels in a grid block.

Moreover, we efficiently store the depth interval grid such that it consumes only $(S_x \times S_y)/(n \times m)$ bytes of memory (where S represents screen resolution) for arbitrarily complex scenes. Even with a 2560×1600 screen resolution and 4×4 blocks, this amounts to only *256 kilobytes* of memory overhead.

At a high level overview, our algorithm performs the following steps each frame:

1) First Pass: Generate Depth Interval Grid
   a) For each pixel block, compute estimated ray depth interval (min and max ray depths).
2) Final Pass: Constrained Ray Refinement
   a) Retrieve the depth interval corresponding to a given pixel.
   b) Within such a depth interval, perform a linear search at fixed step sizes.
   c) Within new interval returned by linear search, perform a binary search.
   d) Render the pixel using the identified intersection point.

Intuitively, the process can be thought of as a low-resolution render pass followed by a graceful upscaling pass informed by actual heightfield data rather than simple "dumb" interpolation. In theory, if the DIG approximation is sufficiently accurate then the resulting image will be pixel-perfect.

4 Depth Interval Grid

To compute depth intervals with guaranteed accuracy, there is no substitute for casting rays from each pixel into the heightfield while computing a min and max reduction. We use a simple approximation instead, where for a given block of pixels the min and max values are estimated by sampling a small subset of rays within the block.

4.1 Sampling Approach

Sampling at the four corners of each block works very well in practice. This is illustrated in Figure 3; notice how the minimum and maximum depths among the corner ray intersections (represented by black dots) generate the correct depth interval.

Although further increasing the number of samples per block improves a depth interval's accuracy, it unnecessarily discards information. Increasing the number of DIG blocks instead achieves the same accuracy improvement (same increase of samples overall) while simultaneously leading to smaller and more efficient depth invervals. Note: If aliasing artifacts resulting from

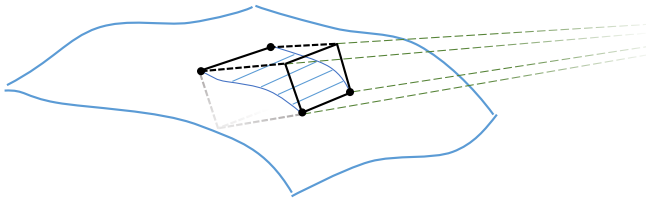

Fig. 3. Illustration of a depth interval sampled from screen-space block corner rays.

uniform grid sampling is a concern, it is trivial to randomize the sample locations used for DIG pixel blocks.

Since each block shares corners with others, computing the min and max can be deferred until after the corner ray depths have been computed. Then, retrieving the depth interval for a given block is as simple as computing the maximum and minimum ray depths found in the surrounding four corner points. We therefore compute and store the ray depth only for the top left corner of every block in the DIG, and reconstruct the interval with Equation 1.

$$\begin{aligned} D_{adj} &= \{d_{x,y}, d_{x+1,y}, d_{x,y+1}, d_{x+1,y+1}\} \\ interval(b_{x,y}) &= (max(D_{adj}) + \epsilon_1, min(D_{adj}) - \epsilon_2) \end{aligned} \quad (1)$$

Where $d_{i,j}$ represents the ray depth for the top-left pixel of block (i, j), and min and max are functions which compute a set's minimum/maximum values.

This can be done very efficiently during the final pass where each pixel within the block is refined, since sampling adjacent pixels from texture memory is extremely fast on modern GPUs (due to modern memory architecture and two dimensional cache locality).

Notice that in Equation 1 the actual interval length is biased by an added factor of $\epsilon_1 + \epsilon_2$. These epsilon values are added to compensate for approximation error. If all corners strike low frequency dominant surfaces, almost no epsilon is required. However, depending on the magnitude of high frequency components in the heightfield, higher epsilon values may be necessary for correct results. In practice, relatively small epsilon values suffice for all but extremely chaotic displacement maps.

4.2 Generation and Storage

The Depth Interval Grid is allocated in memory as an 8 bit per pixel 2-D texture map, where each pixel represents a depth value at the top-left corner of the corresponding DIG block. Since we store one depth value per block, the DIG texture size should be $(S_x/b_x) \times (S_y/b_y)$, where S represents screen resolution and b represents the size of a single pixel block. Only one such DIG buffer is needed for scenes of arbitrary complexity.

Rather than storing world-space ray depths, it is easier and more efficient to store values representing ray depths normalized to $[0, 1]$ where 0 represents rays reaching the bottom of the heightfield and 1 represents rays only just entering the heightfield. This value is equivalent to the tangent-space height of the ray point, as shown in Figure 4.

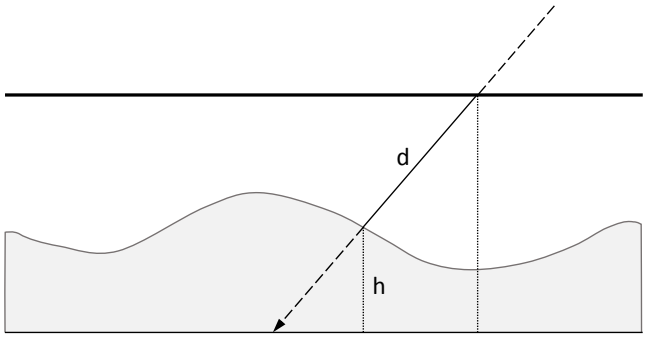

Fig. 4. Diagram of the relationship between ray depth (d) and tangent-space height (h) values.

Populating a DIG is performed using the GPU's *render to texture* capability. This first pass renders the macrostructure geometry to the DIG with a ray casting algorithm (applied via the fragment shader) outputting normalized ray depths. Ray depths can be computed using any *accurate* ray casting algorithm (though it need not be very *precise* since the final pass will refine the ray anyway). We use a simple linear search for this stage with a fixed step size, storing the last depth before penetration. Therefore in this case, from Equation 1 we must choose $\epsilon_1 \geq s_z$ where s_z is the step size of a single linear search iteration.

See Figure 5 for an example image depicting DIG depth intervals from a complex displacement map.

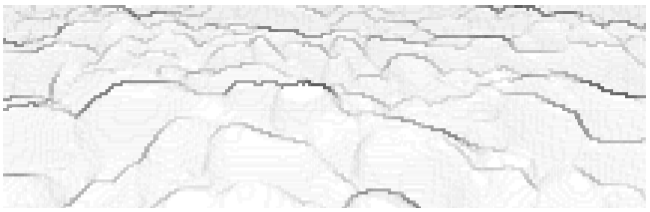

Fig. 5. Depiction of interval lengths extracted from a DIG. Darker tile colors represent longer intervals.

Note that the model, view, and projection matrices used for this stage should be identical to the ones used in the final pass.

5 Constrained Ray Refinement

Once the DIG is computed, the final pass refines ray intervals to locate exact intersection points, then applies a lighting model to generate the pixel's final color.

Similar to Parallax Occlusion Mapping [27], our final pass performs an adaptive linear search followed by a binary search within the remaining interval.

The number of iterations performed by the linear search depends on the length of the depth interval. The number of iterations is computed in such a way that the

horizontal step size between samples is always the same for every ray (see Equation 2). Therefore, configuring the linear search component of the refinement pass consists of setting a sampling step size rather than an iteration count.

$$I_L = \lfloor |R_{xy}|/s \rfloor \quad (2)$$

Where R_{xy} is a 2-D vector representing the horizontal components of the tangent-space ray interval, and s is the search step size in normalized UV magnitude units.

The following sub-sections detail the operation of linear search and binary search adapted to a pre-confined search interval; readers already familiar with this concept may skip Sections 5.1 and 5.2.

5.1 Linear Search Phase

The linear search algorithm [17] [18] proceeds as follows:

1) Calculate texture (UV) coordinates for ray entry and exit points.
2) Calculate begin and end points from the corresponding DIG block's ray interval.
3) Perform I_L iterations, sampling heightfield values ranging from the begin to end points:
   a) Calculate tangent-space ray height.
   b) If ray height is less than sampled height, exit.
4) Store the last two ray height values.

5.2 Binay Search Phase

When a linear search sample dips below the heightfield, the precise intersection point must exist between the just-collided sample and the previous sample. We then converge to the exact intersection by binary search [23].

The number of iterations used for binary search (I_B) is chosen independently of those for the linear search phase. Since binary search converges to the exact intersection at an exponential rate, very few iterations (around 5) are typically needed.

Binary search proceeds as follows:

1) Set candidate interval min and max to the last two ray height values from binary search.
2) For I_B iterations:
   a) Set current ray to a position in the middle of the candidate interval.
   b) Sample the height at the current position.
   c) If the ray height is less than the sampled height, set the lower bound of the candidate interval to the ray height, else set the upper bound of the candidate interval to the ray height.
3) Return the texture coordinates corresponding to the middle of the candidate interval.

The resulting computed texture coordinates should correspond precisely to the first intersection point of the ray.

6 Curved Surfaces

When applying displacement maps to smoothly curved mesostructure surfaces, most displacement algorithms do not attempt computing the intersection of straight rays against curved heightfield geometry. Instead, they often solve this situation by treating rays as curved paths in tangent space, which are tested against regular flat-base heightfield geometry. As a result however, some algorithms require special consideration to accommodate curved rays.

Fortunately, DIG displacement map rendering requires no such modification (beyond the obvious change from linear stepping to a curved search path). Because the depth interval grid simply represents ray depth ranges, the algorithm works well as long as the final pass is able to resume within the designated depth interval. Ray depths normalized to $[0, 1]$ in the DIG work equally well for curved rays; rather than storing tangent-space ray height (as shown in Figure 4), we normalize the ray depth such that 0 represents rays reaching the exit point, and 1 represents rays just entering the heightfield.

7 Limitations

Since our algorithm consists of a two pass process, the GPU receives two draw calls for every object that uses DIG-based displacement mapping. While the overall computational complexity of ray casting (via fragment program) is considerably reduced, the amount of vertex program work is doubled. Therefore, if the majority of a displacement mapped object's computation time is already spent in vertex processing, applying this algorithm may not improve performance. In practice however, such cases are extremely rare. In most modern video games for example, pixel operations tend to constitute the dominant portion of the GPU's workload.

Nonetheless, this trade-off is important to consider when applying this algorithm to a visualization application. In general, DIG displacement mapping considerably improves performance when applied to objects where a great deal of detail is added through displacement maps (for example: terrain, close-up objects, animated oceans, walls, ceilings, etc.)

One additional potential limitation stems from the approximate accuracy of the DIG. The depth interval grid's interval approximation cannot guarantee correct output images in all cases. However, the same is true for other preprocessing-free methods which do not traverse each and every height map texel along the ray's path; for example, practical implementations of relief mapping use linear search with a number of iterations that may skip over dozens of pixels at a time. Where POM, Relief Mapping, and others require increased iterations to accomodate high frequency geometry, DIG may require greater epsilon values.

8 RESULTS

In the following tests, we compare the performance of three relevant algorithms using no precomputation:

1) Depth Interval Grid Disp. Mapping (DIG-DM)
2) Parallax Occlusion Mapping (POM) [28]
3) Relief Mapping (RM) [23]

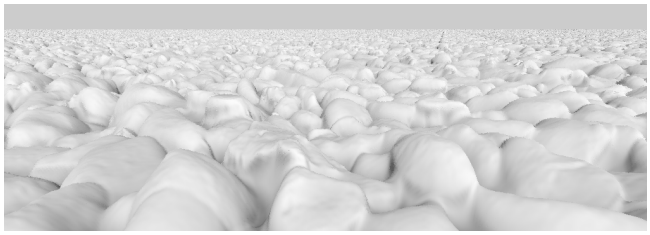

Fig. 6. Example screenshot of a repeated "Rocks" displacement map applied to a large plane with DIG-DM.

An example of an image rendered by DIG Displacement Mapping is shown in Figure 6. Additional coloring is intentionally omitted from this image, leaving only the effects of the displacement map itself visible. After displacement mapping is performed, it is trivial to apply custom coloring and lighting effects as desired.

8.1 Test Methodology

Grazing view angles are the most difficult case to render, due to the very high depth complexity incurred (mesostructure self-occlusion). Therefore, our tests focus on measuring performance and visual quality primarily at grazing angles. We apply displacement maps to an effectively infinite plane viewed at angles ranging from 40 degrees (at the bottom of the screen) to less than 1 degree (near the horizon) – see Figure 6.

Measurements, recorded in frames per second (FPS), represent the average performance over one thousand rendered images during which the viewer continuously rotates around the vertical axis. Moreover, we ensure that images resulting from different algorithms are indistinguishable in visual quality from one another before making any performance comparisons.

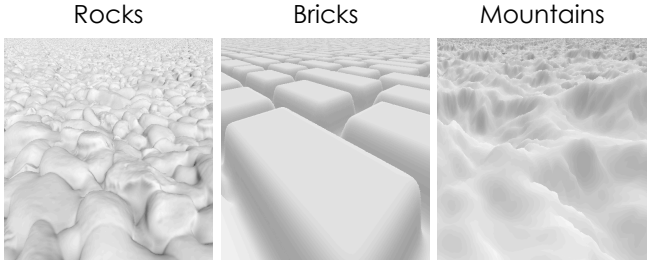

Fig. 7. Three different displacement maps used in testing.

To measure performance consistency, we perform each test on a variety of displacement maps with varying characteristics. These displacement maps are shown and labeled for future reference in Figure 7. These cover three important heightfield types: roughly uniform height distribution (Rocks), top heavy (Bricks), and bottom heavy (Mountains). Section 8.3 describes the significance of height distribution.

8.2 Configuration and Visual Quality

To ensure indistinguishable visual quality between all algorithms, we provide the same linear iteration step size to each.

RM and POM are given the same maximum iteration count $iter$. Similarly, 4×4 DIG blocks are generated with $iter$ linear search iterations, and $\epsilon_1 = \epsilon_2 = (1/iter)$. This epsilon corresponds to the tangent-space step size of a single iteration. Similarly, DIG-DM's refinement stage uses a step size of $(1/iter)$. All algorithms use 5 binary search iterations.

Fig. 8. Images comparing the visual quality of RM vs. DIG-DM using the "Rocks" test displacement map.

Figure 8 shows RM and DIG-DM both running at a maximum of 100 iterations. POM (not shown here) produces visual quality no better than RM, since POM performs no more than 100 iterations.

While some minor pixel differences exist between the two, there appears to be no difference in visual quality. Note that the image resolution shown is intentionally low, since microscopic differences become even less apparent at higher resolutions.

8.3 Iterations to Intersection

The efficiency benefit of DIG-DM is readily apparent when counting the number of search iterations required to find an intersection. In Figure 9, we plot the total iteration count for each pixel's ray. As shown, DIG-DM dramatically reduces the iterations required for the majority of rays.

Notice how this graph suggests that RM/POM convergence efficiency is heavily dependent on how much empty space appears in the displacement map. On the other hand, DIG-DM's sampling performance is much more resilient, offering relatively consistent convergence rates across a wide variety of displacement maps.

8.4 Performance Analysis

We measure performance (in frames per second, or FPS) on two machines:

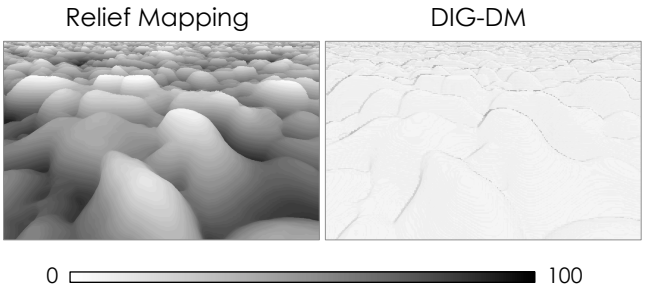

Fig. 9. Graphical representation of ray iterations. Each pixel color represents the number of linear search steps performed before ray intersection.

1) NVIDIA GeForce GTX 580, Intel Core 2 Duo 3 GHz
2) NVIDIA GeForce 320M, Intel Core 2 Duo 1.85 GHz

Results from the first (a relatively modern GPU architecture) are shown in Table 1, and the second (an older mobile GPU configuration) in Table 2. In each, we measure all permutations of algorithms, data, and quality settings (iterations ranging between 75 and 150).

From these tables we make several observations. Firstly, DIG-DM benefits performance most dramatically at higher levels of visual quality. With more iterations, there is more opportunity for DIG space skipping to save time. At 75 iterations on a recent GPU, DIG-DM is **1.64** to **2.90** times as fast as POM, the fastest alternate. At 150 iterations, DIG-DM is **2.18** to **4.04** times as fast.

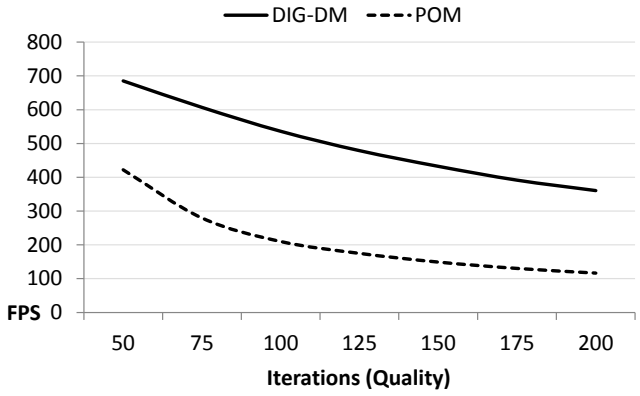

Fig. 10. Performance (frames per second) plotted versus visual quality (iterations), measured from the GeForce GTX 580 rendering the "Rocks" test displacement map.

To better illustrate the variation of performance improvement versus iteration count, refer to Figure 10. Here we plot tests of the Rocks dataset ranging from 50 iterations (unrealistically low quality) to 200 iterations (artifact-free at 2560x1440 screen resolution).

Note that with under 100 iterations, linear search artifacts already begin to appear (affecting all algorithms equally) at these extremely difficult grazing angles unless the depth of the mesostructure geometry is scaled back considerably. Therefore an alternate way of viewing these results are that DIG-DM enables complex displacement mapped surfaces 2x - 4x as deep with no loss of performance or visual quality.

TABLE 1
Performance (frames per second) measurements from an NVIDIA GeForce GTX 580 at 2560x1440 resolution.

Algorithm	Data	$i = 75$	100	125	150
DIG-DM	Rocks	607.0	536.2	478.9	432.5
POM	Rocks	279.4	209.8	174.6	148.8
RM	Rocks	263.1	209.7	168.0	139.8
DIG Speedup		**2.17**	**2.56**	**2.74**	**2.91**
DIG-DM	Mountains	544.1	477.6	425.8	383.2
POM	Mountains	187.6	137.9	111.5	94.8
RM	Mountains	173.7	135.8	109.4	90.3
DIG Speedup		**2.90**	**3.46**	**3.81**	**4.04**
DIG-DM	Bricks	723.8	648.1	588.3	539.2
POM	Bricks	440.6	339.8	284.5	247.2
RM	Bricks	415.1	335.2	276.7	234.0
DIG Speedup		**1.64**	**1.91**	**2.07**	**2.18**

TABLE 2
Performance (frames per second) measurements from an NVIDIA GeForce 320M at 1440x900 resolution.

Algorithm	Data	$i = 75$	100	125	150
DIG-DM	Rocks	72.7	63.9	59.7	55.0
POM	Rocks	44.2	35.6	29.8	25.7
RM	Rocks	39.3	32.5	27.4	24.1
DIG Speedup		**1.65**	**1.80**	**2.00**	**2.14**
DIG-DM	Mountains	55.2	49.0	45.0	40.4
POM	Mountains	26.5	20.5	16.6	14.3
RM	Mountains	23.3	18.3	15.5	13.4
DIG Speedup		**2.08**	**2.39**	**2.71**	**2.83**
DIG-DM	Bricks	91.9	85.4	82.6	78.9
POM	Bricks	66.4	57.0	49.1	42.7
RM	Bricks	60.6	52.4	45.8	40.8
DIG Speedup		**1.38**	**1.50**	**1.68**	**1.85**

Second, we observe performance variation across displacement map data for each algorithm. As noted in Section 8.3, DIG-DM not only converges much faster than POM and RM, but much more consistently. Due to the sensitivity of POM and RM's convergence rate to ray depth, they perform much more favorably for displacement maps with very "top heavy" height value distribution where early ray termination benefits most.

In contrast, while DIG-DM is not impervious to this effect, it is greatly reduced due to the DIG.

As a result, we see (with a modern GPU) that DIG-DM sometimes performs **1.64** times faster for data POM/RM handles unusually well (shallow displacement maps). Conversely, for data POM/RM do not handle well (deep, high-resolution, complex displacement maps), we find DIG-DM performs up to **4.04** times faster (see Table 1).

9 Conclusion

In this paper we presented a new space skipping algorithm, enabling extremely efficient raw displacement map rendering. Without any preprocessing, our algorithm rapidly accelerates convergence of ray-heightfield intersection through exploitation of screen-space ray penetration depth coherence.

As a result, our *Depth Interval Grid Displacement Mapping* (DIG-DM) algorithm renders high quality output images significantly faster than other algorithms. As we have demonstrated, DIG-DM reaches frame rates over twice as fast on average than Parallax Occlusion Mapping [27] and Relief Mapping [23] on modern GPUs. Our algorithm achieves even better performance (up to four times faster) when rendering challenging displacement maps at grazing viewing angles.

DIG-DM performs very consistently across a wide variety of data, while other algorithms suffer severe performance fluctuations. Moreover, since the accelerated ray convergence enabled by the depth interval grids is easily adapted to almost any existing displacement mapping implementation, our algorithm offers potential performance improvements even to existing systems.

Acknowledgments

The work of the first author is supported in part by the Santa Clara University School of Engineering.

References

[1] R. Cook, L. Carpenter, and E. Catmull, "The reyes image rendering architecture," in *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4. ACM, 1987, pp. 95–102.

[2] R. Cook, "Shade trees," in *ACM Siggraph Computer Graphics*, vol. 18, no. 3. ACM, 1984, pp. 223–231.

[3] J. Snydre and D. Nowrouzezahrai, "Fast soft self-shadowing on dynamic height fields," in *Computer Graphics Forum*, vol. 27, no. 4. Wiley Online Library, 2008, pp. 1275–1283.

[4] C. Chang, B. Lin, Y. Chen, and Y. Chiu, "Real-time soft shadow for displacement mapped surfaces," in *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*. IEEE, 2009, pp. 1254–1257.

[5] P.-P. J. Sloan and M. F. Cohen, "Interactive horizon mapping," in *Eurographics Workshop on Rendering Techniques*. Springer-Verlag, 2000, pp. 281–286.

[6] H. Lensch, J. Kautz, M. Goesele, W. Heidrich, and H. Seidel, "Image-based reconstruction of spatial appearance and geometric detail," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 2, pp. 234–257, 2003.

[7] M. Tarini, P. Cignoni, C. Rocchini, and R. Scopigno, "Real time, accurate, multi-featured rendering of bump mapped surfaces," in *Computer Graphics Forum*, vol. 19, no. 3. Wiley Online Library, 2000, pp. 119–130.

[8] T. Boubekeur and C. Schlick, "Generic mesh refinement on gpu," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 2005, pp. 99–104.

[9] J. Patterson, S. Hoggar, and J. Logie, "Inverse displacement mapping," in *Computer Graphics Forum*, vol. 10, no. 2. Wiley Online Library, 1991, pp. 129–139.

[10] J. Logie and J. Patterson, "Inverse displacement mapping in the general case," in *Computer Graphics Forum*, vol. 14, no. 5. Wiley Online Library, 1995, pp. 261–273.

[11] G. Schaufler and M. Priglinger, "Efficient displacement mapping by image warping," in *Eurographics Workshop on Rendering*, 1999, pp. 175–186.

[12] K. Yerex and M. Jagersand, "Displacement mapping with ray-casting in hardware," in *ACM SIGGRAPH 2004 Sketches*. ACM, 2004, p. 149.

[13] J. Hirche, A. Ehlert, S. Guthe, and M. Doggett, "Hardware accelerated per-pixel displacement mapping," in *Proceedings of Graphics Interface 2004*. Canadian Human-Computer Communications Society, 2004, pp. 153–158.

[14] L. Szirmay-Kalos and T. Umenhoffer, "Displacement mapping on the gpustate of the art," in *Computer Graphics Forum*, vol. 27, no. 6. Wiley Online Library, 2008, pp. 1567–1592.

[15] H. Qu, F. Qiu, N. Zhang, A. Kaufman, and M. Wan, "Ray tracing height fields," in *Computer Graphics International, 2003. Proceedings*. IEEE, 2003, pp. 202–207.

[16] J. Dufort, L. Leblanc, and P. Poulin, "Interactive rendering of meso-structure surface details using semi-transparent 3d textures," in *Proc. Vision, Modeling, and Visualization*, 2005, pp. 399–406.

[17] M. McGuire and M. McGuire, "Steep parallax mapping," *I3D 2005 Poster*, pp. 23–24, 2005.

[18] M. Levoy, "Efficient ray tracing of volume data," *ACM Transactions on Graphics (TOG)*, vol. 9, no. 3, pp. 245–261, 1990.

[19] M. McGuire and K. Whitson, "Indirection mapping for quasi-conformal relief texturing," in *Proceedings of the 2008 symposium on Interactive 3D graphics and games*. ACM, 2008, pp. 191–198.

[20] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi, "Detailed shape representation with parallax mapping," in *Proceedings of ICAT*, vol. 2001, 2001, pp. 205–208.

[21] T. Welsh, "Parallax mapping with offset limiting: A perpixel approximation of uneven surfaces," Infiscape Corporation, 2004.

[22] M. Premecz, "Iterative parallax mapping with slope information," in *Central European Seminar on Computer Graphics*. Citeseer, 2006.

[23] F. Policarpo, M. Oliveira, and J. Comba, "Real-time relief mapping on arbitrary polygonal surfaces," in *Proceedings of the 2005 symposium on Interactive 3D graphics and games*. ACM, 2005, pp. 155–162.

[24] E. Risser, M. Shah, and S. Pattanaik, "Faster relief mapping using the secant method," *Journal of Graphics, GPU, and Game Tools*, vol. 12, no. 3, pp. 17–24, 2007.

[25] M. Oliveira, G. Bishop, and D. McAllister, "Relief texture mapping," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 2000, pp. 359–368.

[26] Z. Brawley and N. Tatarchuk, "Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing," *ShaderX3: Advanced Rendering with DirectX and OpenGL*, pp. 135–154, 2004.

[27] N. Tatarchuk, "Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering," in *ACM SIGGRAPH 2006 Courses*. ACM, 2006, pp. 81–112.

[28] ——, "Dynamic parallax occlusion mapping with approximate soft shadows," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 63–69.

[29] K. Oh, H. Ki, and C. Lee, "Pyramidal displacement mapping: a gpu based artifacts-free ray tracing through an image pyramid," in *Proceedings of the ACM symposium on Virtual reality software and technology*. ACM, 2006, pp. 75–82.

[30] A. Kolb and C. Rezk-Salama, "Efficient empty space skipping for per-pixel displacement mapping," in *Proc. Vision, Modeling and Visualization*, 2005, pp. 407–414.

[31] W. Donnelly, "Per-pixel displacement mapping with distance functions," *GPU gems*, vol. 2, no. 22, p. 3, 2005.

[32] J. Hart *et al.*, "Sphere tracing: Simple robust antialiased rendering of distance-based implicit surfaces," *Modeling, Visualizing and Animating Implicit Surfaces. SIGGRAPH*, vol. 93, 1993.

[33] J. Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12, no. 10, pp. 527–545, 1996.
[34] J. Dummer, "Cone step mapping: An iterative ray-heightfield intersection algorithm," Tech. Rep., 2006. [Online]. Available: http://www.lonesock.net/files/ConeStepMapping.pdf
[35] F. Policarpo and M. Oliveira, "Relaxed cone stepping for relief mapping," *GPU gems*, vol. 3, pp. 409–428, 2007.
[36] Y. Chen and Y. Chuang, "Anisotropic cone mapping," in *Proceedings: APSIPA ASC 2009: Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference*. Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference, International Organizing Committee, 2009, pp. 660–663.
[37] L. Lee, S. Tseng, and W. Tai, "Improved relief texture mapping using minmax texture," in *Image and Graphics, 2009. ICIG'09. Fifth International Conference on*. IEEE, 2009, pp. 547–552.
[38] L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H. Shum, "View-dependent displacement mapping," in *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3. ACM, 2003, pp. 334–339.
[39] J. Wang and K. Dana, "Compression of view dependent displacement maps," in *Texture 2005: Proceedings of the 4th International Workshop on Texture Analysis and Synthesis*. Citeseer, 2005, pp. 143–148.
[40] G. Peyré and S. Mallat, "Surface compression with geometric bandelets," in *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3. ACM, 2005, pp. 601–608.