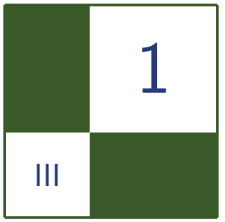

Quadtree Displacement Mapping with Height Blending

Michal Drobot

1.1 Overview

This article presents an overview and comparison of current surface rendering techniques, and introduces a novel approach outperforming existing solutions in terms of performance, memory usage, and multilayer blending. Algorithms and ideas researched during *Two Worlds 2* development are shared, and the article proposes strategies for tackling problems of realistic terrain, surface and decal visualization considering limited memory, and computational power on current-generation consoles. Moreover, problems of functionality, performance, and aesthetics are discussed, providing guidelines for choosing the proper technique, content creation, and authoring pipeline.

We focus on various view and light-dependant visual clues important for correct surface rendering such as displacement mapping, self-shadowing with approximate penumbra shadows, ambient occlusion, and surface correct texture blending, while allowing real-time surface changes. Moreover, all presented techniques are valid for high quality real-time rendering on current generation hardware as well as consoles (as Xbox 360 was the main target platform during research).

First, existing parallax mapping techniques are compared and contrasted with real-life demands and possibilities. Then we present a state-of-the-art algorithm yielding higher accuracy with very good performance, scaling well with large height fields. It makes use of empty space skipping techniques and utilizes texture MIP levels for height quadtree storage, which can be prepared at render time. Second, a soft shadows computation method is proposed, which takes advantage of the quadtree. We expand upon this to calculate an ambient-occlusion term. Next, we introduce an LOD technique which allows higher performance and

Figure 1.1. Normal mapped environment.

Figure 1.2. Fully featured surface rendering using the methods proposed in this article.

quality for minification. Then we focus on surface blending methods, proposing a new method that exhibits better resemblance to real life and allows aggressive optimization during blended height-field displacement mapping. The proposed methods—depending on combinations and implementation—guarantee fast, scalable, and accurate displacement mapping of blended surfaces, including visually pleasing ambient occlusion and soft penumbra soft shadowing (compare Figures 1.1 and 1.2). Specific attention will be given to the various implementations and the proper choice of rendering method and asset authoring pipeline.

1.2 Introduction

During the last change of console generation we have seen a dramatic improvement in graphics rendering quality. With modern GPUs pushing millions of triangles per second, we are looking for more fidelity in areas that are still being impractical for performance reasons. One of those is surface rendering, which is one of the most fundamental building blocks of believable virtual world.

Each surface at its geometric level has a number of complex properties such as volume, depth, and various frequency details that together model further visual clues like depth parallax, self-shadowing, self-occlusion, and light reactivity. The topic of light interactions depending on surface microstructure is well researched and so many widely used solutions are provided (such as Cook-Torrance's lighting model and its optimizations). However, correct geometry rendering is still problematic. The brute force approach of rendering every geometry detail as a triangle mesh is still impractical because it would have to consist of millions of vertices, thus requiring too much memory and computations. Moreover, surface blending such as widely seen on terrain (i.e., sand mixing with rocks) only complicate the situation in terms of blend quality and additional performance impact. Last but not least, we would like to manipulate surface geometric properties at render time (i.e., dynamic water erosion simulation, craters forming after meteor strike).

To sum up, we would like our surface rendering method to support:

- accurate depth at all angles (depth parallax effect);
- self-shadowing;
- ambient occlusion;
- fast blending;
- dynamic geometric properties;
- current-generation hardware (taking console performance into account);
- minimal memory footprint compared to common normal mapping.

Common normal mapping techniques (those which create the illusion of detailed surface by performing light computation on precalculated normal data set) fail to meet our demands, as they do not model visual geometry clues. However, we still find it useful in light interaction calculations, thus complementing more sophisticated rendering solutions.

The only rendering method class that is able to suit all our needs are height-field-based ray-tracing algorithms. The idea behind those algorithms is to walk along a ray that entered the surface volume, finding the correct intersection of the ray with the surface. They operate on grayscale images representing height values of surfaces, thus exchanging vertex for pixel transformations, which suits our hardware better in terms of performance and memory usage. Moreover, they mix well with existing normal mapping and are performing better as GPUs become more general processing units. However, none of them are aimed at high performance surface blending or ambient occlusion calculation.

During our research we were seeking for the best possible rendering method meeting our demands, being robust, functional and fast as we were aiming for Xbox360-class hardware. As our scenario involved fully-featured rendering of outdoor terrain with many objects and indoor cave systems, we were forced to take special care for an automatic LOD system. Several methods were compared and evaluated, finally ending with the introduction of a new solution that proved to be suiting all our needs. We describe our research and the motivation behind it, going in detail with each building block of the quadtree Displacement Mapping with Height Blending technique.

1.3 Overview of Ray-Tracing Algorithms

Every height-field-based ray-tracing algorithm is working with additional displacement data, commonly encoded in height map format (grayscale image scaled to [0; 1] range). Calculations are done in tangent space to allow computations for arbitrary surfaces. Correct surface depth is calculated by finding the intersection between viewing ray and height field. That ensures correct parallax offset for further color and lighting calculations.

Figure 1.3 illustrates the depth parallax effect and presents the general intersection calculation.

General height-field ray-tracing algorithms can be summarized as follows:

1. Calculate tangent-space normalized view vector V per-vertex, and interpolate for pixel shader.

2. Ray cast the view ray to compute intersection with the height field, acquiring the texture coordinate offset required for arriving at the correct surface

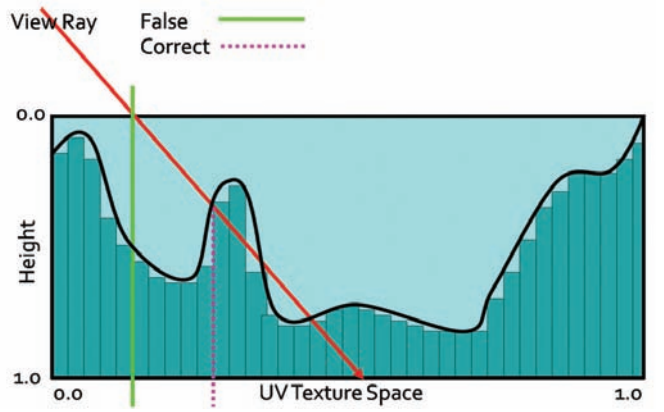

Figure 1.3. Height-field ray-trace scheme.

point. We start at texture input T1 coordinates, sampling along the surface's profile, finally computing new texture coordinates T2.

3. Compute the normal lighting equation using surface attributes sampled at the new texture coordinates T2.

The following algorithms implement various methods for intersection computation, varying in speed, accuracy and use of additional precomputed data.

1.3.1 Online Algorithms

Relief mapping. Relief mapping [Policarpo 2005] performs intersection calculation by linear search in two-dimensional height-field space followed by binary search.

We want to find the intersection point (p, r). We start by calculating point (u, v), which is the two-dimensional texture coordinate of the surface point where the viewing ray reaches a depth $= 1.0$. The point (u, v) is computed based on initial texture coordinates (x, y) on the transformed view direction with scaling factor applied. Then we search for (p, r) by sampling the height field between (x, y) and (u, v). We check for intersections by comparing ray depth with the stored depth at the current sampling point. When the latter is smaller, we have found the intersection and we can refine it using binary search. Figure 1.4 illustrates the process.

Binary search is taking advantage of texture filtering and operates in minimized space around the found intersection point. That ensures fast convergence and high accuracy. However, using that kind of search utilizes dependant reads on the GPU, thus vastly affecting performance. While a linear- and binary-search combo

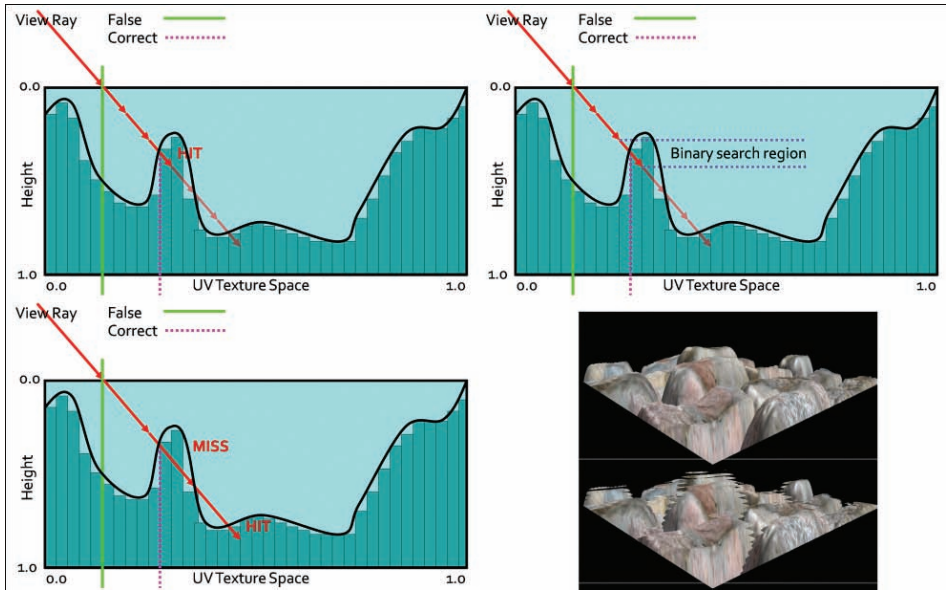

Figure 1.4. Relief Mapping. Top left: linear search. Top right: binary search around point found in linear search. Bottom left: possible miss of linear search. Bottom right: resulting aliasing artifacts as opposed to correct rendering.

is a known and proven solution, its linear part is prone to aliasing due to under sampling. During the search, when there are not enough search steps (step length is too big), we might miss important surface features as shown in Figure 1.4. Increasing search steps potentially minimizes the problem but severely affects performance, making this algorithm highly unreliable when sampling large height fields or surfaces exhibiting very large displacement scales. Nonetheless, it is still very effective in simple scenarios that do not require high sampling count, as the performance is saved on ALU instructions.

Parallax occlusion mapping. Several researchers tried to optimize this algorithm by omitting the expensive binary search. *Parallax occlusion mapping* [Tatarchuk 2006] relies on accurate high-precision intersection calculation (see Figure 1.5). A normal linear search is performed finding point (p, r) and last step point (k, l). Then the ray is tested against a line made of (p, r) and (k, l), effectively approximating the height profile as a piecewise linear curve. Moreover, solutions for additional LOD and soft shadows were proposed. POM, while being accurate enough and faster than relief mapping, is still prone to aliasing and so exhibits the same negative traits of linear search (Listing 1.1).

```
float Size = 1.0 / LinearSearchSteps;
float Depth = 1.0;
int StepIndex = 0;
float CurrD = 0.0;
float PrevD = 1.0;
float2 p1 = 0.0;
float2 p2 = 0.0;

while(StepIndex < LinearSearchSteps)
{
    Depth -= Size; //move the ray
    float4 TCoord = float2(p+(v*Depth)); // new sampling pos
    CurrD = tex2D(texSMP, TCoord).a; //new height
    if (CurrD > Depth) //check for intersection
    {
        p1 = float2(Depth, CurrD);
        p2 = float2(Depth + Size, PrevD); //store last step
        StepIndex = LinearSearchSteps; //break the loop
    }
    StepIndex++;
    PrevD = CurrD;
}

//Linear approximation using current and last step
//instead of binary search, opposed to relief mapping.
float d2 = p2.x - p2.y;
float d1 = p1.x - p1.y;

return (p1.x * d2 - p2.x * d1) / (d2 - d1);
```

Listing 1.1. POM code.

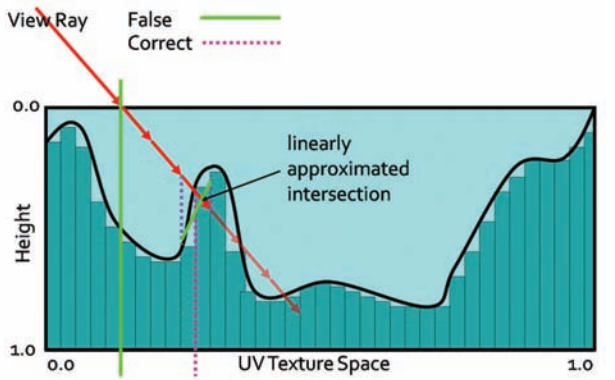

Figure 1.5. POM.

1.3.2 Algorithms Using Precomputed Data

In response to the arising problem of performance and accuracy, several solutions were proposed that make use of additional data to ensure skipping of empty space and prohibit missing surface features. However, additional memory footprint or preprocessing computation time limits their usefulness.

Per-pixel displacement with distance function. *Per-pixel displacement with distance function* [Donelly 2005] uses precalculated three-dimensional texture representation of the surface's profile. Each texel represents a sphere whose radius is equal to the nearest surface point. We are exchanging the well-known linear search for sphere tracing. With each sample taken we know how far we can march our ray without missing any possible intersection. Traversing that kind of structure allows skipping large space areas and ensures that we will not miss the intersection point. Moreover, the intersection search part is very efficient. However, memory requirements and precomputation time for this method make it impractical for real-time game environments. As stated in [Donelly 2005], even simple surfaces may require a three-dimensional texture size of $256 \times 256 \times 16$ with dimensions rising fast for more complex and accurate rendering. That increase in memory footprint is unacceptable for the limited memory of current consoles and PC hardware, not to mention the prohibitive preprocessing time.

Cone step mapping (CSM). CSM [Dummer 2006] is based on a similar idea. It uses a cone map that associates a circular cone with each texel of the height-field texture. The cone angle is calculated so that the cone is the largest one not intersecting the height field (see Figure 1.6). This information allows us

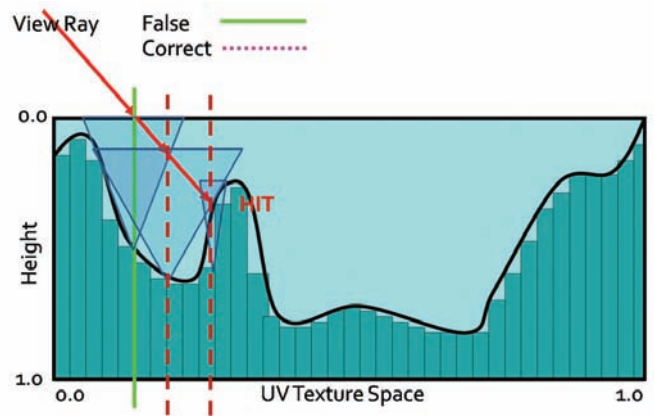

Figure 1.6. CSM. Ray traversal by cone radius distance.

to calculate a safe distance during sampling, as in per-pixel displacement with distance function. Consequently, the ray may skip empty spaces and never miss the correct intersection. Due to its conservative nature, the algorithm may require too many steps to actually converge. For performance reasons, it is required to set a maximum number of steps, which often results in stopping the ray trace too early and returning incorrect texture coordinates for further rendering.

Cone step mapping performance varies widely depending on the spatial coherency of the height field. Generally, it outperforms linear search algorithms while guaranteeing less noticeable errors. Its memory footprint is quite bearable as it requires only one additional 8-bit texture for cone maps. However, its preprocessing time makes it impossible to alter the height field at render time, as this would require recompilation of the cone map with every change. The precomputation algorithm is of complexity $O(n^2)$, where n denotes number of height-field texels, making it impractical on current GPUs. Moreover, properties of the cone map prohibit correct and easy surface blending.

Relaxed cone step mapping (RCSM). RCSM [Policarpo 2007] takes CSM one step further, making it less conservative. The idea is to use larger cones that intersect the height field only once. The search is performed the same way as in CSM. When the intersection is found, the correct point is searched, using binary search in space restricted by the last cone radius, therefore converging very quickly. The combination leads to more efficient space leaping, while remaining accurate, due to final refinement. Furthermore, an LOD scheme is proposed which, while it lacks accuracy, provides performance gains. In practice, RCSM is currently the fastest ray-tracing algorithm available, making it very useful in scenarios where neither long preprocessing times, disability of efficient blending, and dynamic height-field alteration are irrelevant.

1.4 Quadtree Displacement Mapping

We introduce a GPU-optimized version of the classic [Cohen and Shaked 1993] hierarchical ray-tracing algorithm for terrain rendering on CPU, using height-field pyramid, with bounding information stored in mipmap chain. It was presented on recent hardware by [OH 2006], yielding good accuracy and performance, but at the same time was less adequate for game scenario use. We describe our implementation, optimized for current GPUs, with an automatic LOD solution and accurate filtering. Moreover, we expand it for optimized surface blending, soft shadowing and ambient occlusion calculation.

QDM uses the mipmap structure for resembling a dense quadtree, storing maximum heights above the base plane of the height field (it is worth noting that our implementation is actually using depth maps as 0 value representing

maximum displacement, as we are storing depth measured under the reference plane. In consequence, maximum heights are minimum depths, stored in our data structure). We traverse it to skip empty space and not to miss any detail. During traversal we are moving the ray from cell boundary to cell boundary, until level 0 is reached—hence valid intersection region. While moving through the hierarchy, we compute the proper hierarchy level change. Finally, we use refinement search in the region of intersection to find the accurate solution when needed.

Gf 8800	256^2	512^2	1024^2	2048^2
Quad tree	0.15ms	0.25ms	1.15ms	2.09ms
CSM	$<$ 2min	$<$ 14min	$<$ 8h	/

Table 1.1. Data preprocessing time.

1.4.1 Quadtree Construction

The quadtree is represented by a hierarchical collection of images in a mipmap. The construction is simple, as it requires generating mipmaps with the min operator instead of average as during normal mipmapping. As a result, MIP level 0 (2^n) represents the original height field with the following levels 1 (2^{n-1}), 2 (2^{n-2}), ... containing the minimum value of the four nearest texels from levels above. The entire process can be run on the GPU. Due to hardware optimization, quadtree construction is very fast. The timings in Table 1.1 were obtained on a PC equipped with Intel Core 2 Duo 2.4 GHz and GeForce 8800. For comparison, timings for RCSM are given. The quadtree was computed on the GPU, while the cone map was on the CPU due to algorithm requirements.

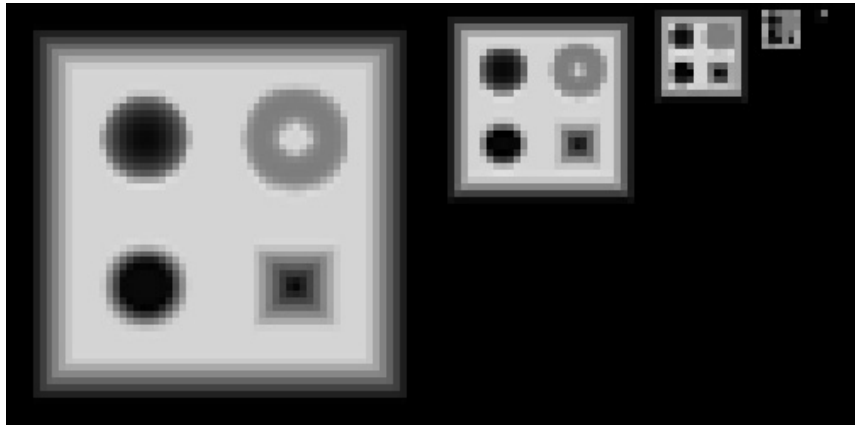

Figure 1.7. Generated QDM on mipmaps.

As we can see, quadtree computation time is negligible, even for on-the-fly generation, whereas cone maps could even be problematic for off-line rendering during texture authoring (see Figure 1.7).

1.4.2 Quadtree Traversal

The general steps of intersection search are shown by the pseudocode in Listing 1.2. We start the computation at the highest mipmap level. The stopping condition of the main loop is to reach the lowest hierarchy level, which effectively means finding the intersection region where the linear approximation can be performed. At each step, we determine if we can move the ray further or if there is a need for further refinement. We algebraically perform the intersection test between the ray and the cell bounding planes and the minimum depth plane.

In case the ray does not intersect the minimum plane, then the current cell is blocking our trace. We have to refine the search by descending in the hierarchy by one level. In the other case, we have to find the first intersection of the ray with the minimum plane or the cell boundary. When the right intersection is computed, we move the ray to the newly acquired point. In case we have to cross the cell boundary, then we choose the next cell via the nearest-neighbor method, thus minimizing error. At this point, we perform hierarchy level update for optimization (see the optimization section).

Figure 1.8 presents step-by-step ray traversal in QDM: Step (a) shows a ray coming from the initial geometry plane and stopping at the maximum level or minimum plane. Steps (b) and (c) illustrate further refinement while the search descends to lower hierarchy levels. Step (d) presents where the ray must cross the cell in order to progress. While the minimum plane of the current cell is not blocking the way, we have to move the ray to the nearest cell boundary. Steps (e) and (f) show further ray traversal while refining the search while (g) presents the main loop's stopping condition, as the ray has reached level 0. Therefore, we can proceed to linear interpolation between the nearest texels in Step (h).

```
While (hierarchy_level > 0)
    depth=get_maximum_depth(position, hierarchy level)
    If(ray_depth < depth)
        move_ray_to_cell_boundry_or_minimum_depth_plane
    else
        descend_one_hierarchy_level
    end
    find_intersection_using_linear_interpolation
```

Listing 1.2. General QDM search steps.

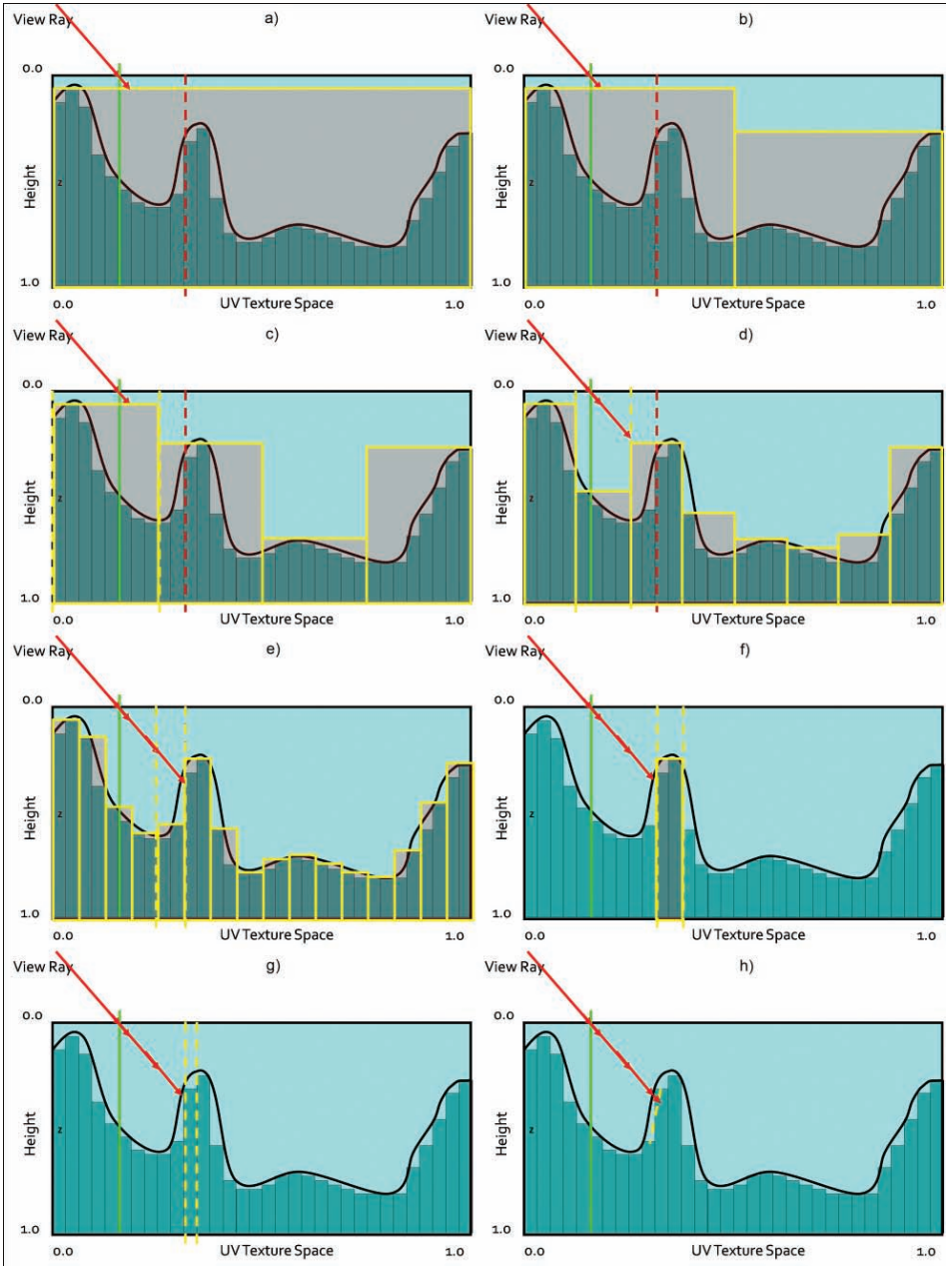

Figure 1.8. QDM traversal.

It is important to correctly calculate sampling positions since we are working with a discrete data structure. For correct results, we should use point-filtering on the GPU and integer math. However, if we cannot afford an additional sampler for the same texture using POINT and LINEAR, it is possible to use linear filtering with enough care taken for correct calculations. As SM 3.0 only emulates integer operations, we have to account for possible errors in calculations (using SM 4.0 is preferable, due to the presence of real integer math).

Listing 1.3 is heavily commented to explain the algorithm's steps in detail.

```
const int MaxLevel = MaxMipLvl;
const int NodeCount = pow(2.0, MaxLevel);
const float HalfTexel = 1.0 / NodeCount / 2.0;
float d;
float3 p2 = p;
int Level = MaxLevel;

//We calculate ray movement vector in inter-cell numbers.
int2 DirSign = sign(v.xy);

//Main loop
while (Level >= 0)
{
    //We get current cell minimum plane using tex2Dlod.
    d = tex2Dlod(HeightTexture, float4(p2.xy, 0.0 , Level)).w;

    //If we are not blocked by the cell we move the ray.
    if (d > p2.z)
    {
        //We calculate predictive new ray position.
        float3 tmpP2 = p + v * d;

        //We compute current and predictive position.
        //Calculations are performed in cell integer numbers.
        int NodeCount = pow(2, (MaxLevel - Level));
        int4 NodeID = int4((p2.xy , tmpP2.xy) * NodeCount);

        //We test if both positions are still in the same cell.
        //If not, we have to move the ray to nearest cell boundary.
        if (NodeID.x != NodeID.z || NodeID.y != NodeID.w)
        {
            //We compute the distance to current cell boundary.
            //We perform the calculations in continuous space.
            float2 a = (p2.xy - p.xy);
            float2 p3 = (NodeID.xy + DirSign) / NodeCount;
            float2 b = (p3.xy - p.xy);
```

```
            //We are choosing the nearest cell
            //by choosing smaller distance.
            float2 dNC = abs(p2.z * b / a);
            d = min(d, min(dNC.x, dNC.y));

            //During cell crossing we ascend in hierarchy.
            Level+=2;

            //Predictive refinement
            tmpP2 = p + v * d;
        }

        //Final ray movement
        p2 = tmpP2;
    }

    //Default descent in hierarchy
    //nullified by ascend in case of cell crossing
    Level--;
}
return p2;
```

Listing 1.3. QDM search steps.

1.4.3 Optimizations

Convergence speed-up. It is worth noting that during traversal, the ray can only descend in the hierarchy. Therefore, we are not taking full advantage of the quadtree. The worst-case scenario occurs when the ray descends to lower levels and passes by an obstacle really close, consequently degenerating further traversal to the linear search. To avoid that problem, we should optimally compute the correct level higher in the hierarchy during cell crossing. However, current hardware is not optimized for such dynamic flow. A simple one step up move in the hierarchy should be enough. For more complicated surfaces which require many iterations, we discovered that this optimization increases performance by 30% (tested on a case requiring >64 iterations).

Fixed search step count. While the algorithm tends to converge really quickly, it may considerably slow down at grazing angles on complex high-resolution height fields. Therefore, an upper bound on iterations speeds up rendering without very noticeable artifacts. The number of iterations should be exposed to technical artists to find optimal values.

Linear filtering step. Depending on surface magnification and the need for accurate results, final refinement may be used. One can utilize the well-know binary search which would converge quickly (five sampling points is enough for most purposes)

due to the tight search range. However, we propose linear interpolation for piecewise surface approximation, similar to the one proposed in POM. This approach proves to be accurate on par with binary search (considering limited search range), while being optimal for current hardware.

After finding the correct cell of intersection (at hierarchy level 0), we sample the height field in direction of the ray cast, one texel before and one texel after the cell. Then we find the intersection point between the ray and linearly approximated curve created by sampled points.

LOD scheme. Here we propose a mixed automatic level of detail scheme. First, we dynamically compute the number of iterations based on the simple observation that parallax is larger at grazing angles, thus requiring more samples, so we can express the correct value as a function of the angle between the view vector and the geometric normal. Notice that the minimum sampling number should not be less than the total number of hierarchy levels, otherwise the algorithm will not be able to finish traversal even without any cell crossing. Moreover, we observed that with diminishing pixel size on screen, parallax depth details become less visible. Thus, we can stop our quadtree traversal at a fixed hierarchy level without significant loss of detail. We determine the right level by computing the mipmap level per pixel scaled by an artist-directed factor. For correct blending between levels, we linearly interpolate between depth values from the nearest hierarchy level by the fractional part of the calculated mipmap level. After an artist-specified distance we blend parallax mapping to normal mapping.

This combined solution guarantees high accuracy via a dynamic sampling rate, and it guarantees high performance due to quadtree pruning, thus giving a stable solution overall. For performance and quality comparisons, see the results section.

Storage. For precision reasons, it is required that the stored quadtree is accurate. A problem arises when we want to store it with textures. Generally, textures compressed with DXT compression result in a significant speedup and memory footprint reduction. DXT is a lossy compression scheme; thus it is not recommended for accurate data storage (as opposed to, e.g., diffuse textures). However, we noticed that in the general case, storing the quadtree in the alpha channel of a DXT5-compressed texture results in minor artifacts (it highly depends on the surface's profile, so it must be done with care). Still, the preferable solution is to take the memory hit and store the additional data in 1-channel 8-bit lossless textures.

Comparison. Performance tests were conducted on a test machine equipped with Intel quad core 2.8Ghz CPU and GeForce 260 GTX in full HD, using three various levels from early beta of TW2. The results can be seen in Tables 1.2, 1.3, and 1.4.

Scenarios contained fully featured levels, using various height fields of resolution 512^2 to 1024^2. Each scene pushed around one million triangles, with the parallax displacement method of choice, covering the entire screen. Depth scale

Relief	CSM	QDM
$\sqrt{n}$	$\leq \sqrt{n}$	$\log n$

Table 1.2. Analytical performance.

Depth Scale	POM	QDM
1.0	5ms	5.7ms
1.5	6.65ms	6.7ms
5.0	18.9ms	9ms

Table 1.3. General scenario performance.

POM	QDM
73ms	14ms

Table 1.4. Extreme high detail performance.

dependence was measured, and iteration count was set for close quality match between methods. Table 1.4 shows the timing for ultra-high resolution and complexity case of the height field. Figures 1.9 and 1.10 show results for various implementations of ray- and height-field-intersection algorithms.

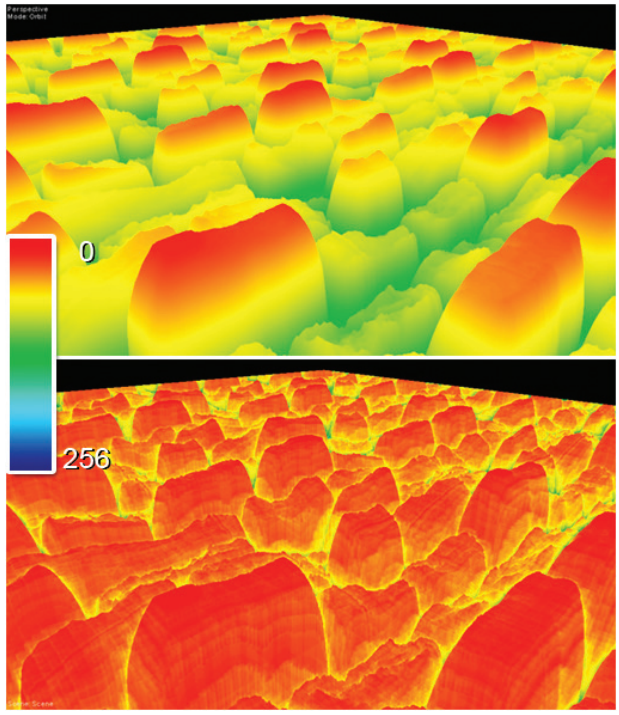

Figure 1.9. Convergence comparison.

Figure 1.10. Quality comparison. Left POM. Right QDM. Depth Scale: 1.0, 1.5, 5.0. From depth scale 1.5 and above artifacts are visible while using POM.

As we can see, QDM is converging in a pace similar to RCSM, but results in worse performance due to higher ALU cost and less efficient texture cache usage (as many dependant samples, from various MIP levels tend to cause cache misses). However, it is still comparably fast compared to linear search algorithms, outperforming them when the height-field's complexity, depth, or resolution increases. After further research we discovered that QDM is a faster solution onwards from 512×512 high-complexity textures or for any 1024×1024 and larger sizes. Moreover, an additional advantage is visible with depth scale increase. We do not take into consideration RCSM for resolutions higher than 1024×1024 as the preprocessing time becomes impractical.

1.5 Self-Shadowing

1.5.1 General Self-Shadowing

The general algorithm for self-shadowing [Policarpo 2005] involves ray tracing along the vector from L (light source) to P (calculated parallax offset position), then finding its intersection PL with the height field. Then we simply compare PL with P to determine whether the light is visible from P or not. If not, then that means we are in shadow. See Figure 1.11 for illustration.

This method generates hard shadows using any intersection search method, thus suffering from the same disadvantages as the chosen algorithm (i.e., aliasing with linear search). Moreover, the cost is the same as view and height-field

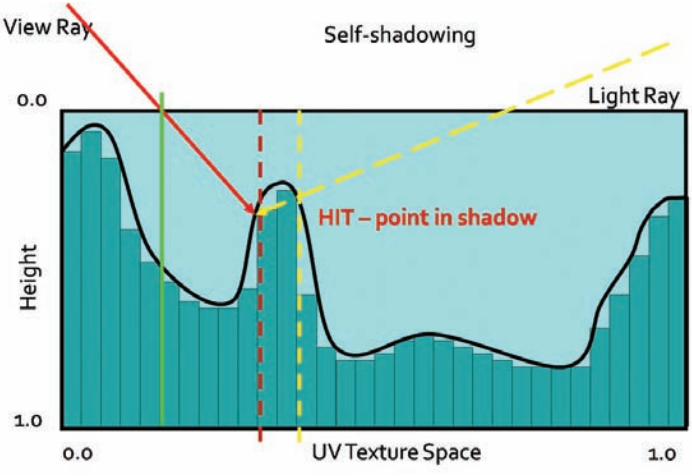

Figure 1.11. Shadow test.

intersection computations. When we are looking for soft shadows, we would have to perform several ray casts of the light vector, which is impractical for performance reasons.

A good approximation for soft shadow calculation was proposed in POM. We can cast a ray towards the light source from point P and perform horizon visibility queries of the height profile along the light direction. We sample the height profile at a fixed number of steps to determine the occlusion coefficient by subtracting sampled depth from original point P depth. This allows us to determine the penumbra coefficient by calculating blocker-to-receiver ratio (the closer the blocker is to the surface, the smaller the resulting penumbra). We scale each sample's contribution by the distance from P and use the maximum value, then we use the visibility coefficient in the lighting equation for smooth shadow generation. The algorithm makes use of linear sampling and produces well-behaving soft shadows. However, the alias-free shadow range is limited by the fixed sampling rate, so real shadows cannot be generated without searching through the entire height field, which effectively degenerates the algorithm to a linear search.

1.5.2 Quadtree Soft Shadowing

Fast horizon approximation. We propose a new algorithm based on POM soft shadowing. This algorithm makes use of the quadtree used in QDM.

First we introduce the algorithm for fast horizon visibility approximation. We use a method similar to POM by performing horizon visibility queries along a

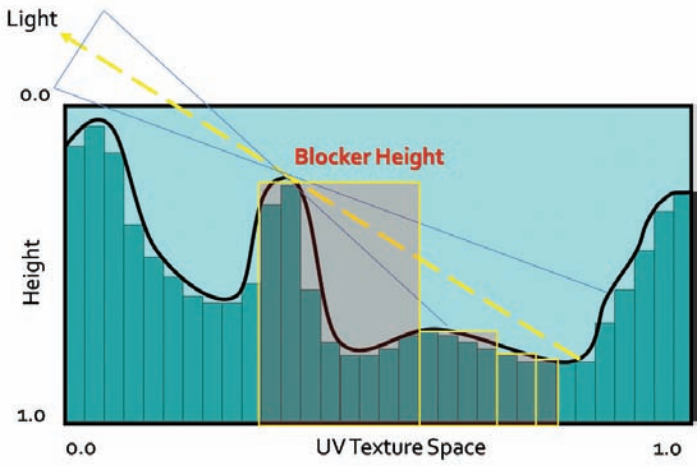

Figure 1.12. QDM horizon approximation.

given direction. The maximum height is taken from the calculated samples and is subtracted from the initial starting point P, thus giving the horizon visibility coefficient. See Figure 1.12 for illustration.

We use the observation that small scale details at distance have negligible impact on the result (especially during any lighting calculations). Thus we can approximate further lying profile features by using the maximum height data from higher levels of the quadtree. That way we can calculate the approximated horizon angle with a minimized number of queries. The full profile can be obtained in $\log n$ steps as opposed to n steps in POM, where n is the number of height-field texels along a given direction D. In all further solutions, we are using a slightly modified version of this algorithm, which is weighting each sample by a distance function. That makes it more suitable for penumbra light calculation as samples far away from P are less important.

QDM shadowing. For shadowing, we use a fast horizon visibility approximation using the parallax offset point P and the normalized light vector L. Accuracy and performance is fine-tuned by technical artists setting the plausible number of iterations ($\log n$ is the maximum number, where n is the height-field's largest dimension) and light vector scale coefficient as shown in Listing 1.4.

```
//Light direction
float2 lDir = (float2(l.x, -l.y)) * dScale;

//Initial displaced point
float h0 = tex2Dlod(heightTexture, float4(P,0,0)).w;
float h = h0;

//Horizon visibility samples
//w1..w5---distance weights
h = min(1,w1 * tex2Dlod(height, float4(P + 1.0 * lDir,0,3.66)).w);
h = min(h,w2 * tex2Dlod(height, float4(P + 0.8 * lDir,0,3.00)).w);
h = min(h,w3 * tex2Dlod(height, float4(P + 0.6 * lDir,0,2.33)).w);
h = min(h,w4 * tex2Dlod(height, float4(P + 0.4 * lDir,0,1.66)).w);
h = min(h,w5 * tex2Dlod(height, float4(P + 0.2 * lDir,0,1.00)).w);

//Visibility approximation
float shadow = 1.0 - saturate((h0 - h) * selfShadowStrength);

return shadow;
```

Listing 1.4. QDM soft shadows, fast, hard-coded solution.

Results. As we can see in Table 1.5, plausible soft shadows using the quadtree are significantly faster than traditional methods while still delivering similar quality

POM Shadows	QDM Shadows
1.6ms	0.5ms

Table 1.5. One light shadows calculation time for the test scene.

(see Figure 1.13). For further optimization we are calculating shadows only when $N \cdot L \geq 0$.

Figure 1.13. Soft shadows ON/OFF.

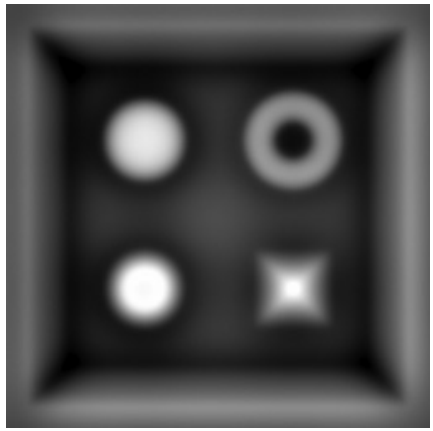

Figure 1.14. Generated high quality AO.

1.6 Ambient Occlusion

Ambient occlusion is computed by integrating the visibility function over the hemisphere H with respect to a projected solid angle:

$$A_0 = \frac{1}{\pi} \sum_H V_{p,\omega}(N \cdot \omega) d\omega,$$

where $V_{p,\omega}$ is the visibility function at p, such as $V_{p,\omega}$ is 0 when occluded in direction ω and 1 otherwise.

Ambient occlusion adds a great deal of lighting detail to rendered images (see Figure 1.14). It is especially useful for large-scale terrain scenarios, where objects can take the occlusion value from the terrain (i.e., darkening buildings lying in a valley).

1.6.1 QDM Ambient Occlusion

Dynamically calculating ambient occlusion for surfaces was thought to be impractical for performance reasons, as the visibility function and the integration were too slow to be useful. Now with fast horizon visibility approximation we can tackle that problem in a better way.

We approximate the true integral by integrating the visibility function in several fixed directions. We discovered that for performance reasons integration in four to eight directions lying in the same angle intervals yields acceptable results. Moreover, we can increase quality by jittering and/or rotating the directions by a random value for every pixel.

Accuracy	AO
4 directions	2.1ms
8 directions	6.3ms
4 jittered	2.6ms

Table 1.6. AO calculation time

1.6.2 Performance

As we can see from Table 1.6, the algorithm requires substantial processing power, being even less efficient with randomized directions (as it is hurting GPU parallelism, but it is still faster than integrating more directions). However, it is used only when the surface height field is changing. Moreover, it can be accumulated throughout several frames, amortizing the cost.

1.7 Surface Blending

1.7.1 Alpha Blending

Blending is commonly used for surface composites, such as terrain, where several varied textures have to mix together (e.g., rocky coast with sand).

Typically, surface blends are done using the alpha-blending algorithm given by the following equation:

$$\text{Blend} = \frac{(v_1, \ldots, v_n) \cdot (w_1, \ldots, w_n)}{(1, \ldots, 1) \cdot (w_1, \ldots, w_n)},$$

where $(w_1, \ldots, w_n)$ is the blend weight vector and $(v_1, \ldots, v_n)$ denotes the value vector.

Commonly, the blend vector for a given texel is supplied by vertex interpolation, stored at vertex attributes (thus being low frequency). During pixel shading, interpolated values are used to perform the blending.

1.7.2 Raytracing Blended Surfaces

Any height-field intersection algorithm can be used in such a scenario. We should compute the parallax offset for each surface and finally blend them together using the blend vector. However, it is worth noting that the computational cost for such blending would be n-times higher than one surface, where n is the total number of surfaces. Moreover, using vertex-interpolated values results in view-dependant surfaces floating near blend zones. Figure 1.15 illustrates the problem. However,

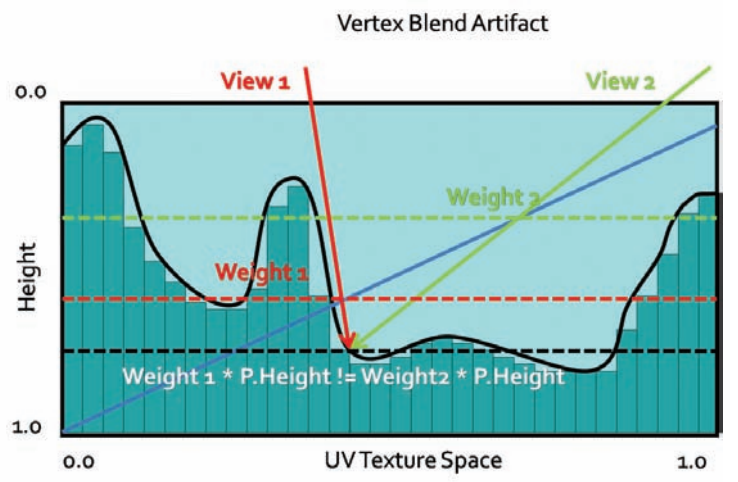

Figure 1.15. Depth floating artifact using vertex blend.

with enough care taken to minimize depth scale near blend zones, it should not be too distracting.

One possible solution is to use per-pixel blend weights that would modify the surface on-the-fly during intersection search. However, this would require sampling an additional weight texture with every iteration, thus doubling the sample count.

Let us consider four surface blends. Optimally, we can encode up to four blend weights in one RGBA 32-bit texture, so with each sample we get four weights. The blend texture can be fairly low-resolution, as generally it should resemble per-vertex blending (it can be even generated on the fly from vertex weights). Having a four-scalar blend vector, we can perform the intersection search on the dynamically modified height field simply by sampling all four height fields with each step and blending them by the blend vector. Moreover, we can compose all four height fields into one RGBA 32-bit texture, thus finally optimizing the blend intersection search.

The pseudocode in Listing 1.5 shows the modification for the intersection search method of choice.

```
d = tex2D(HeightTexture,p.xy).xyzw;
b = tex2D(BlendTexture,p.xy).xyzw;
d = dot(d,b);
```

Listing 1.5. Height profile blend code.

Modification requires only one additional sample and one dot product. However, we are sampling four channels twice instead of one channel (as in the single surface algorithm).

This solution is therefore very fast but lacks robustness, as it would require us to preprocess height-field composites, creating possibly dozens of new textures containing all possible height profiles composites. We can of course try sampling the data without composites, but that would result in additional sampling cost and cache misses (as four samplers would have to be used simultaneously, which would most probably result in a bandwidth bottleneck).

Another problem is that we cannot use this method for algorithms using precomputed distance data, as it would require us to recompute the distance fields (i.e., cone maps) for blend modified height fields, which effectively prohibits using advanced ray-casting algorithms.

1.7.3 Height Blending

To overcome the aforementioned problems, we introduce a new method for surface blending, which seems to fit the task more naturally, and it guarantees faster convergence.

First, let us consider typical alpha blending for surface mixing. In real life, surfaces do not blend. What we see is actually the highest material (the material on the top of the surface).

Therefore, we propose to use height information as an additional blend coefficient, thus adding more variety to blend regions and a more natural look as shown in Listing 1.6.

This method is not computationally expensive, and it can add much more detail as opposed to vertex-blended surfaces (as can be seen in Figure 1.16).

The most important feature is that we pick the highest surface, so during the intersection search phase, we need only to find the highest point.

Therefore, the new height field is produced by the new blend equation:

$$\text{Blend} = \max(h_1, \ldots, h_n).$$

Using this blend equation we are interested only in finding the intersection point with the highest surface profile modified by its blend weight. That effectively means taking a minimal number of steps, as we will stop the ray cast at the

Relief Mapping	POM	POM with HB
3ms	2.5ms	1.25ms

Table 1.7. Surface blend performance comparison

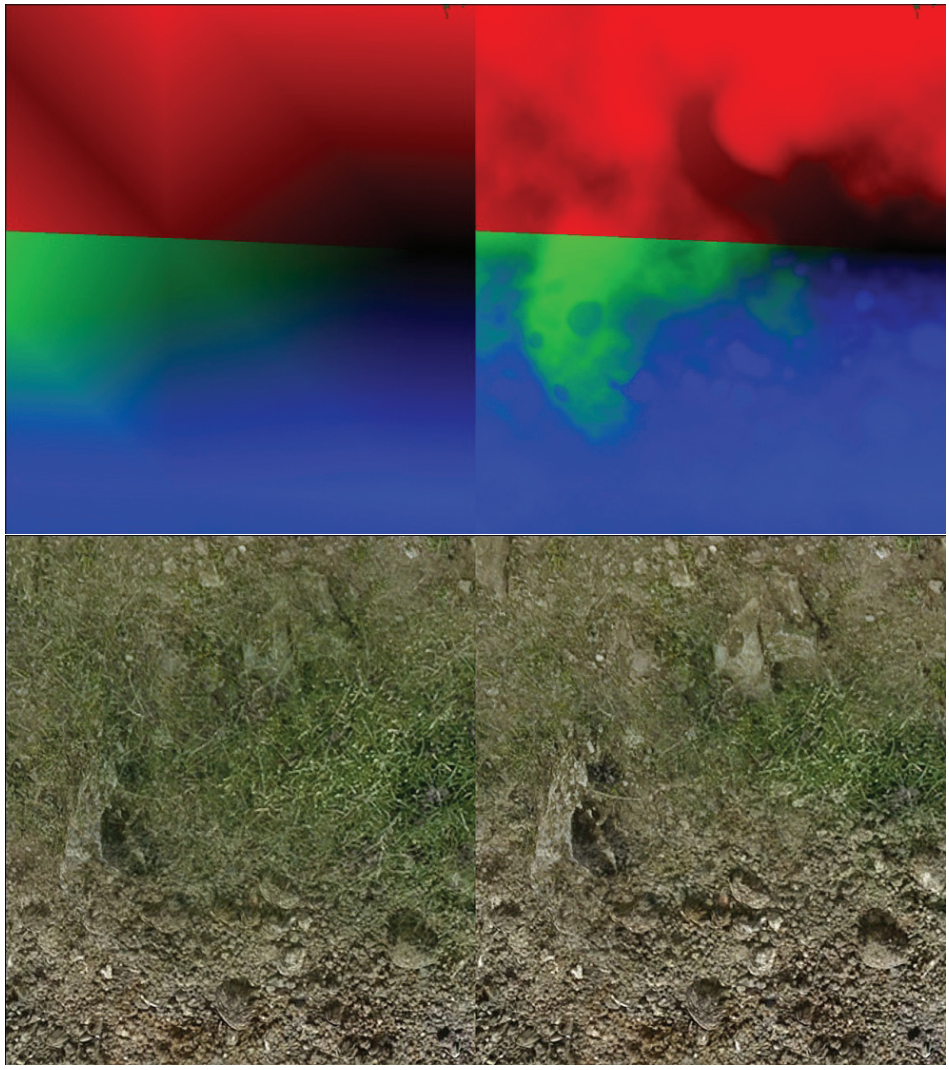

Figure 1.16. Vertex blend and height blend comparison.

```
float4 FinalH;
float4 f1, f2, f3, f4;

//Get surface sample.
f1 = tex2D(Tex0Sampler,TEXUV.xy).rgba;

//Get height weight.
FinalH.a = 1.0 - f1.a;
f2 = tex2D(Tex1Sampler,TEXUV.xy).rgba;
FinalH.b = 1.0 - f2.a;
f3 = tex2D(Tex2Sampler,TEXUV.xy).rgba;
FinalH.g = 1.0 - f3.a;
f4 = tex2D(Tex3Sampler,TEXUV.xy).rgba;
FinalH.r = 1.0 - f4.a;

//Modify height weights by blend weights.
//Per-vertex blend weights stored in IN.AlphaBlends
FinalH*= IN.AlphaBlends;

//Normalize.
float Blend = dot(FinalH, 1.0) + epsilon;
FinalH /= Blend;

//Get final blend.
FinalTex = FinalH.a * f1 + FinalH.b * f2 + FinalH.g * f3 +
           FinalH.r * f4;
```

Listing 1.6. Surface blend code.

first intersection with highest blend weight modified height profile, which—by definition—is the first one to be pierced by the ray.

With each intersection search step, we reconstruct the height-field profile using the new blend operator as shown in Listing 1.7.

As can be seen in Table 1.7, this method proved to minimize the convergence rate by 25% on average in our scenario without sacrificing visual quality (see Figure 1.17, and is more plausible for our new height blend solution. It can be used with blend textures or vertex blending, as well as every intersection search algorithm.

```
d = tex2D(HeightTexture,p.xy).xyzw;
b = tex2D(BlendTexture,p.xy).xyzw;
d *= b;
d = max(d.x,max(d.y,max(d.z,d.w)));}
```

Listing 1.7. Surface height blend code.

Figure 1.17. Surface blend quality comparison. Top: relief. Bottom: POM with height blending.

1.7.4 QDM with Height Blending

We still cannot use the height blend operator directly for algorithms based on precomputed data. However, QDM is based on depth data, so it is relatively easy

to obtain new correct data structure. Note that

$$\max(x_1, x_2, \ldots, x_n) \cdot \max(w_1, w_2, \ldots, w_n) \geq \max([(x_1, x_2, \ldots, x_n) \cdot (w_1, w_2, \ldots, w_n)]).$$

Thus multiplying one min/max quadtree by another gives us a conservative quadtree, and that is exactly what we need for correct surface blending. We can pack up to four blend quadtrees in one RGBA 32-bit texture with mipmaps containing blend vector quadtrees. Then in QDM, to reconstruct the blended surface quadtree, we simply sample and blend it at the correct position and level, and compute the dot product between it and the height-field vector sampled from the height-field composite.

The blend texture should map quadtree texels as close as possible. However, we discovered that while using hardware linear sampling and accepting small artifacts we can use sizes as small as 32^2 (while blending 1024^2 height fields) when the weight gradients are close to linear. Such blended quadtrees can be constructed on the fly in negligible time, allowing dynamic surface alterations.

Effectively, we can use QDM with all its benefits while blending surfaces for artifact-free rendering (see Figure 1.18). Convergence will be slower, due to the conservative quadtree, and more iterations may be needed depending on the height-field's complexity. In practice, the conservative approach needs <10% more iterations than what should be really used. This method proves to be the fastest method for dynamic accurate surface rendering of high complexity height fields.

In our implementation we decided to use vertex blending to avoid high texture cache misses. However, we were forced to accept small depth-floating artifacts.

As QDM is converging really fast in empty space regions, the algorithm can make the best use of faster convergence, due to height blending.

1.7.5 Self-Shadowing and Ambient Occlusion for Blended Surfaces

Self shadowing and ambient occlusion can be done while rendering blended surfaces. However, a naïve approach of calculating shadowing terms for each surface and blending the results is simply impractical for current generation hardware. We propose to use QDM and the height blend and perform computations for the highest modified height profile only. Proper height-field selection requires additional dynamic branching, further restricting GPU parallelism. Consequently, self shadowing and/or ambient occlusion are viable only for high-end hardware.

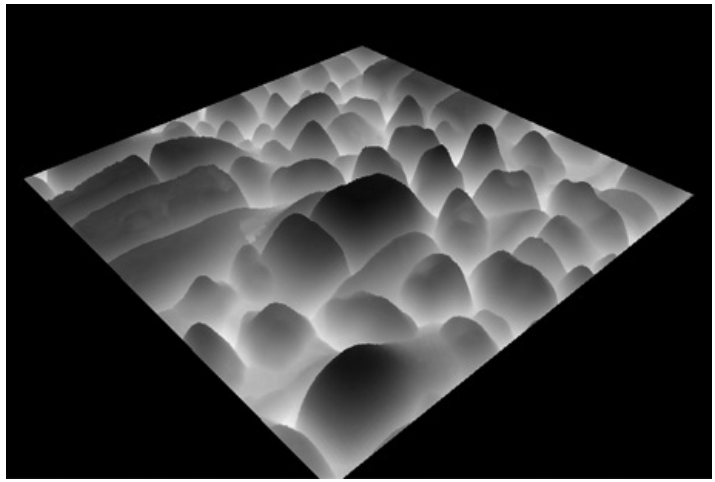

Figure 1.18. QDM height blend surface.

1.8 General Advice

In this article we proposed and discussed several battle-proven surface rendering methods, varying in ALU/Texture sampling performance, accuracy, and flexibility. Most solutions were introduced as general building blocks from which, depending on requirements, an optimal solution can be built.

1.8.1 Case Study

During *Two Worlds 2* production we decided to settle on several solutions used under specific circumstances. We present a case study of each method used:

General terrain blend. Our terrain exhibits small-scale height features such as cracks, small rocks, etc. The maximum number of blending surfaces was capped at four to allow texture packing. We are using linear search with linear piecewise approximation, automatic LOD, and height blend optimization. Blending is done on a per-vertex basis. Depending on texture configuration, parallax can be switched off for each surface individually. The specular term and normal vectors are generated on-the-fly due to the Xbox360's memory restrictions.

Special terrain features. Several extreme detail terrain paths were excluded as additional decal planes. We are rendering them at ultra-high quality (high resolution, high accuracy) and alpha-blending them with existing terrain. Decal planes may present roads, paths, muddy ponds, and craters, etc. For rendering, we are using

QDM with automatic LOD and soft shadows. Where needed, QDM per-pixel height blending is used. Blend-based decals are for PC only.

General object surface. For general surface rendering, we are using linear search with linear piecewise approximation and automatic LOD. Soft shadows are used at the artist's preference. Surfaces with extreme complexity, depth scale, or resolutions over 1024^2 are checked, and using QDM is optimal. The same method is used on Xbox360 and PC.

1.8.2 Surface Rendering Pipeline

During asset finalization, technical artists optimized height-field-based textures, checking whether high resolution or additional details (such as soft shadows) are really needed. It is worth noting that low frequency textures tend to converge faster during the intersection search, so blurring height fields when possible is better for performance and accuracy reasons when using linear search-based methods.

One important feature of our surface rendering pipeline is the preference for generation of additional surface properties on-the-fly, as it allows us to save memory and performance on texture-fetch-hungry shaders.

Texture-heavy locations (such as cities) are using mostly two 24-bit RGB compressed textures per object. The specular term is generated from the diffuse color and is modified by an artist on a per-material-specified function such as inversion, power, or scale. The generated coefficient generally exhibits high quality.

Data generation is taken to the extreme during terrain rendering as individual terrain texture is using only 32-bit RGBA DXT5 textures, from which per-pixel normal vectors, specular roughness, and intensities (as the default lighting model is a simplified Cook-Torrance BRDF) are generated.

1.9 Conclusion

We have discussed and presented various surface rendering techniques with several novel improvements for industry proven approaches. Combinations of parallax mapping, soft shadowing, ambient occlusion, and surface blending methods were proposed to be in specific scenarios aiming for maximum quality/performance/memory usage ratio. Furthermore, a novel solution—Quadtree Displacement Mapping with Height Blending—was presented. Our approach proves to be significantly faster for ultra-high quality surfaces that use complex, high resolution height fields. Moreover, we proposed solutions for efficient surface blending, soft shadowing, ambient occlusion, and automatic LOD schemes using the introduced quadtree structures. In practice, our techniques tend to produce higher quality results with less iterations and texture samples. This is an advantage, as we are

trading iteration ALU cost for texture fetches, making it more useful for GPU generations to come, as computation performance scales faster than bandwidth.

Surface rendering techniques research allowed us to make vast graphic improvements in our next-gen engine, thus increasing quality and performance. We hope to see the techniques described herein being used in more upcoming titles.

Bibliography

[Cohen and Shaked 1993] D. Cohen and A. Shaked. "Photo-Realistic Imaging of Digital Terrains." *Computer Graphics Forum* 12:3 (1993), 363–373.

[Donelly 2005] W. Donelly. "Per-Pixel Displacement Mapping with Distance Functions." In *GPU Gems 2*, edited by Matt Pharr, pp. 123–36. Reading, MA: Addison-Wesley, 2005.

[Dummer 2006] J. Dummer. "Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm." 2006. Available at http://www.lonesock.net/files/ConeStepMapping.pdf.

[OH 2006] K. Oh, H. Ki, and C. H. Lee. "Pyramidal Displacement Mapping: a GPU based Artifacts-Free Ray Tracing through Image Pyramid." *In VRST '06: Proceedings of the ACM symposium on Virtual Reality Software and Technology* (2006), 75–82.

[Policarpo 2005] F. Policarpo, M. M. Oliveira, and J. L. D. Comba. "Real Time Relief Mapping on Arbitrary Polygonal Surfaces." *In Proceedings of I3D'05* (2005), 155–162.

[Policarpo 2007] F. Policarpo and M. M. Oliveira. "Relaxed Cone Step Mapping for Relief Mapping." In *GPU Gems 3*, edited by Hubert Nguyen, pp. 409–428. Reading, MA: Addison-Wesley Professional, 2007.

[Tatarchuk 2006] N. Tatarchuk. "Practical Parallax Occlusion Mapping with Approximate Soft Shadow." In *ShaderX5.* Brookline, MA: Charles River Media, 2006.