Contents

1. Introduction

The goal of this thesis is creating a method to support displacement mapping for ray tracing for procedurally generated textures. The textures are generated using an input texture and tiling and blending areas of it creates an infinitely large texture that can be evaluated on demand at a specified coordinate. As displacement mapping for ray tracing is usually done by pre-tessellation the surface, this would prevent the procedural texture from being updated and changed at runtime. By adjusting a different method that performs displacement mapping without pre-tessellation by creating a bounding volume hierarchy inside the intersection shader on-the- fly, it is possible to render and change these procedural textures at semi-interactive rates. Alongside this thesis, an application is developed that implements this new approach using the Vulkan ray tracing pipeline to create a semi-interactive demo.

2. Fundamentals

2.1 Ray Tracing

As the method presented in this paper is based on ray tracing a short overview over ray tracing is given here. In the real world, cameras produce an image by registering the amount of light that hits the sensor at discrete places, the pixels. This process can be simplified to a light source emitting photons, which fly in certain directions and whenever they hit an object they bounce off and change directions accordingly. If one of these photons, after many bounces, happens to hit a point on the sensor of the camera, then light is registered there. The more photons hit one spot, the brighter the image in that place gets. Ray tracing reverses this process and instead of emitting photons off a light source, tracing their path through the scene and hoping that they hit the sensor at some point, we trace the inverse way. We can think of it as shooting particles from the camera, tracing where they go and if they hit a surface, we can trace another ray from the intersection point to our light source to find out, how much light gets bounced off of it into our camera. Instead of shooting particles, we shoot rays and trace their way through the scene, hence the name. Apart from hitting objects, the direction the photon is going will not change, at least in most cases this change is neglectable, so substituting them with a ray is a sufficient approximation to produce good looking images. One of the challenges of ray tracing is finding out if and where the ray intersects the geometry so its path can be adjusted. Computing these intersections can be really costly in terms of performance and many optimizations are needed to speed it up.

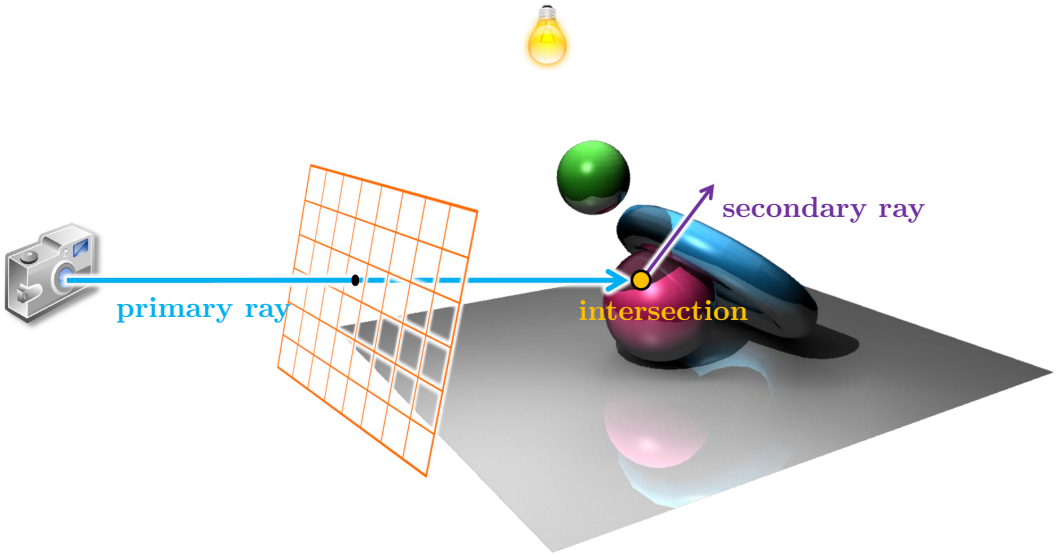


Figure 2.1: A ray is shot from the camera, hits an object and bounces off of it. Usually the rays shot from the camera are called *primary rays* and the ones shot from an *intersection* point are called *secondary rays*. Figure adapted from [Dac18].

2.2 Acceleration Structure

One of the methods for increasing the speed for intersection with the scene is to build a hierarchical structure for the scene. To further increase the speed, intersection tests are only performed against the real geometry at the leafs of the hierarchical structure and every other intersection test is performed against axis aligned bounding boxes (AABB). Intersection against an AABB is fast for mathematical reasons. In Figure 2.2 such a bounding volume hierarchy (BVH) can be seen with AABBs as nodes while the leafs contain the real geometry. As all the children of one node are fully bounded by its AABB, if the intersection test against this AABB fails, it is clear that every intersection with any children of it will also fail. This way large chunks of the scene don't have to be intersected against and thus increase performance.

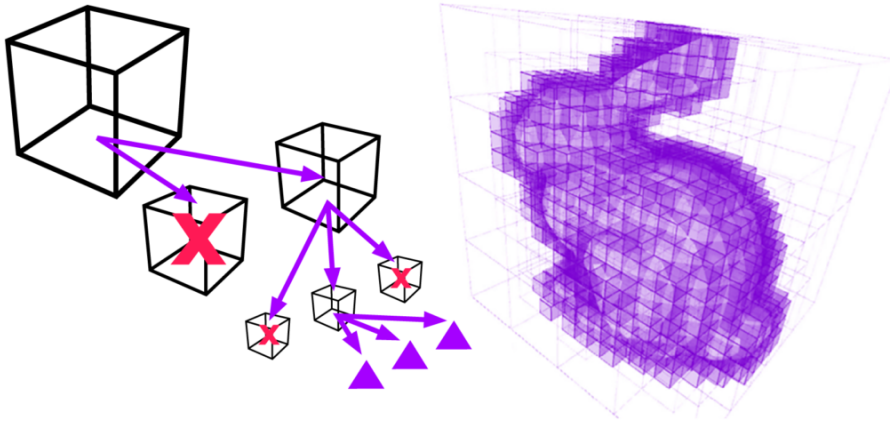


Figure 2.2: Traversal of the BVH. On the **left** the scene graph is shown. Red crossings mean that the intersection against a node failed, so no intersections with any children of this node have to be evaluated. On the **right** the positive intersection tests are highlighted in pink. Figure from [Ima].

Vulkan Ray Tracing. The acceleration structure used by the Vulkan API for ray tracing exposes two different levels to the programmer, the top-level acceleration structure (TLAS) and the bottom-level acceleration structure (BLAS). Both the TLAS and the BLAS are in itself hierarchical structures and they serve as interaction points for the programmer. The BLAS usually contains individual models while the TLAS is built from multiple BLASes and corresponds to an entire scene [LGBA].

2.3 Displacement Mapping

Displacement mapping is the progress of changing the height of a base mesh based on the values stored in a texture. This way, a high level of detail can be created without needing to model it by hand. As they are scalar values, height values, ranging from 0 to 1, are stored in only one color channel in the displacement texture. To mathematically describe a displaced surface $\mathbf{S}(u, v)$ that is displaced from specified uv-coordinates, that are used to fetch the texture, one can use the following equation. $\mathbf{P}(u, v)$ is the position on the base mesh, $\hat{\mathbf{N}}(u, v)$ the normalized interpolated normal and $h(u, v)$ is the displacement map sampled at coordinates (u, v).

$$\mathbf{S}(u, v) = \mathbf{P}(u, v) + h(u, v)\hat{\mathbf{N}}(u, v) \tag{2.1}$$

To achieve this deformation of the base mesh, tessellation can be used in combination with displacing the vertices. For rasterization, the tessellation shader in the shader pipeline can be used to create the additional geometry while in ray tracing mostly pre-tessellation of the surface is performed.

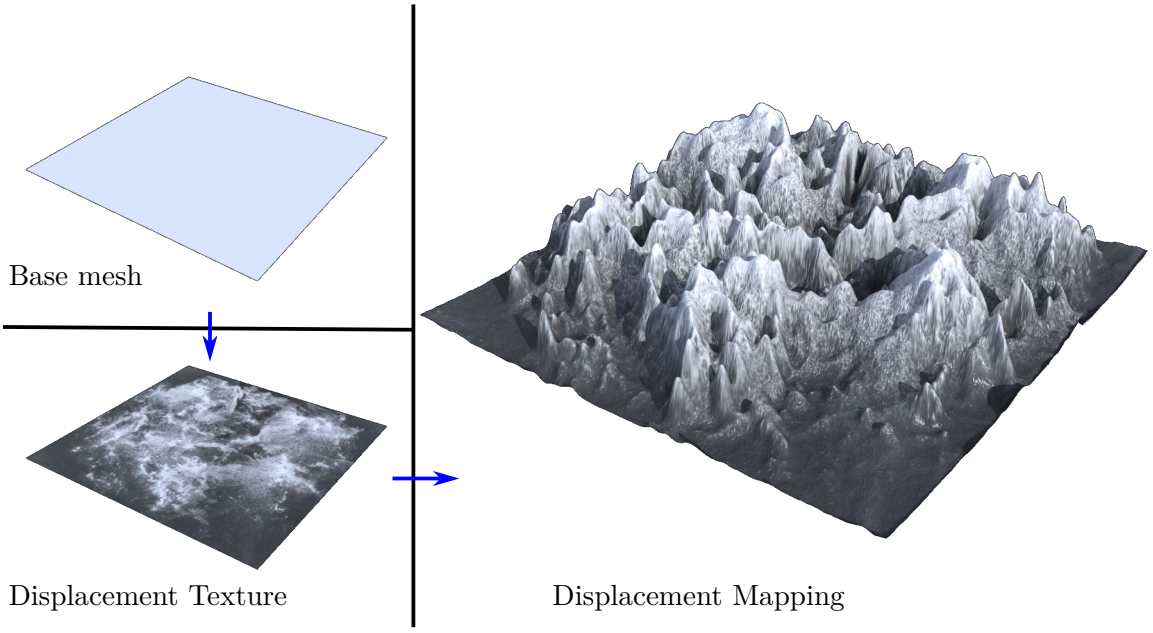


Figure 2.3: A base mesh is transformed by displacement mapping. The height values are visualized ranging from 0 (*black*) to 1 (*white*). Figure adapted from [L. 08].

2.4 Affine Arithmetics

Affine Arithmetics are used in Section 4.2 and Chapter 5 to perform computations on intervals. Because they are first order approximations, they have a higher accuracy then interval arithmetics [SdF03]. A quantity x is therefore represented by its affine form $\left[x\right]$:

$$\left[x\right] = x_c + x_1\epsilon_1 + x_2\epsilon_2 + ... + x_n\epsilon_n \tag{2.2}$$

The coefficients x_i are floating point numbers, the ϵ_i are symbolic real variables inside the interval $[-1, +1]$ and x_c is the center point. We can transform the affine form $\left[x\right]$ to interval form, which consists of a minimum $\underline{x}$ and a maximum $\overline{x}$, with the following equation:

$$\underline{x} = x_c - \sum_{i=1}^{n} |x_i| \tag{2.3}$$

$$\overline{x} = x_c + \sum_{i=1}^{n} |x_i| \tag{2.4}$$

Throughout this thesis we will only keep track of 3 coefficients, x_u and x_v for UV-coordinates and x_K for any other approximations that are not based on the UV-coordinates. For readability purposes the affine form $\left[x\right]$ will be written as $\left[x_c, x_u, x_v, x_K\right]$. As Affine Arithmetics are used as a tool, only the equations used in this thesis are described here. Addition between two affine forms [TBS+21]:

$$\left[x\right] + \left[y\right] = \left[x_c + y_c, x_u + y_u, x_v + y_v, x_K + y_K\right] \tag{2.5}$$

Dot product between two affine forms [TBS+21]:

$$\left[x\right] \cdot \left[y\right] = \begin{bmatrix} x_c \cdot y_c \\ x_u \cdot y_c + x_c \cdot y_u \\ x_v \cdot y_c + x_c \cdot y_v \\ |x_K \cdot y_c| + |x_c \cdot y_K| + (|x_u| + |x_v| + x_K) \cdot (|y_u| + |y_v| + y_K) \end{bmatrix} \tag{2.6}$$

The equation for matrix-vector product can be derived from the dot product (Equation 2.6) and addition (Equation 2.5). As the square root of an affine form is only needed in combination with computing the interval bounds, the square root can be performed on the interval bounds itself (the interval bounds are computed using Equation 2.3 and Equation 2.4):

$$\sqrt{\left[x\right]} \in \left[\sqrt{x_c - |x_u| - |x_v| - |x_K|}, \sqrt{x_c + |x_u| + |x_v| + |x_K|}\right] \tag{2.7}$$

3. Related Work

3.1 Parallax Mapping

First introduced by Kaneko et al. [KTI+01], parallax mapping is a method to displace surfaces. It creates a displaced surface by using per-pixel image distortion once a pixel is rasterized (or an intersection is found when ray tracing). As it is a texture based approach, any procedural texture should be able to be used for parallax mapping as well. Although, as the name suggests, it correctly creates a parallax effect, there are some artifacts visible when the camera is moving. Also it only supports inward displacement.

3.2 Sphere Tracing

Donnelly [Don05] proposes a method to create displacement through sphere tracing a distance field that is computed from a displacement texture. Similar to parallax mapping the displacement is inward and starts when the surface is first rasterized (or in case of ray tracing an intersection is found). It produces fast and accurate results, however, it can only be applied to flat surfaces as it produces distortion when used on curved surfaces [Don05].

3.3 Procedural Displacement Shaders

Heidrich and Seidel [HS+98] introduce a method that displaces a surface based on a mathematical function provided inside the shader. They create bounding boxes on-the-fly in a bounding box hierarchy to perform efficient intersections against them, thus narrowing down the intersection point. This works only for mathematically described displacement and not for displacement textures. In its overall approach it works quite similar to the method outlined in Section 4.2.

4. Underlying Work

4.1 Procedural Textures by Tiling and Blending

Creating details on large surfaces via texturing is a common technique. When the same texture is used over and over again though, sometimes the same regions of this texture become visible and the observer can recognize the base texture through these patterns. Eric Heitz and Fabrice Neyret [HN18] propose a method that takes an example texture as input and generates an infinitely large procedural texture from it on-the-fly. Their approach works best on textures that contain rough patterns and the main property of this method is that the histogram of the input texture is preserved in the procedural one. As the generation is only done on demand, no precomputation of this procedural texture has to be done.

4.1.1 Overview

This method partitions the UV-space, which is mapped onto a triangle for texture access, into a lattice of equilateral triangles. Inside each of these lattice triangles, barycentric coordinates are used as weights for blending between 3 different regions fetched from the input texture with the variance-preserving blending operator which is later described in detail. As the blending operator is only histogram-preserving for Gaussian distributed values, the histogram of the input texture has to be transformed to a Gaussian distributed histogram in advance and later, after the blending, the inverse transformation is applied to get the histogram back to its original distribution.

4.1.2 Histogram Transformations

As noted before, the input texture has to be transformed so that its histogram is Gaussian distributed. According to Eric Heitz and Fabrice Neyret [HN19] an expectation of $\mu = \frac{1}{2}$ and a standard deviation of $\sigma = \frac{1}{6}$ are a good choice for the parameters. To transform one distribution into another, the cumulative distribution function (**CDF**) and its inverse ($\mathbf{CDF}^{-1}$) can be used according to Bergen and Heeger [HB95]. First, the $\mathbf{CDF}_I$ of our input texture is applied to transform our distribution into uniformly distributed values. Then, the inverse Gaussian $\mathbf{CDF}_G^{-1}$ is used to transform this uniform distribution into a Gaussian one. Burley [Bur19] uses this property to perform the histogram transformation per channel using the following Gaussian $\mathbf{CDF}_G$ and inverse Gaussian $\mathbf{CDF}_G^{-1}$ of the

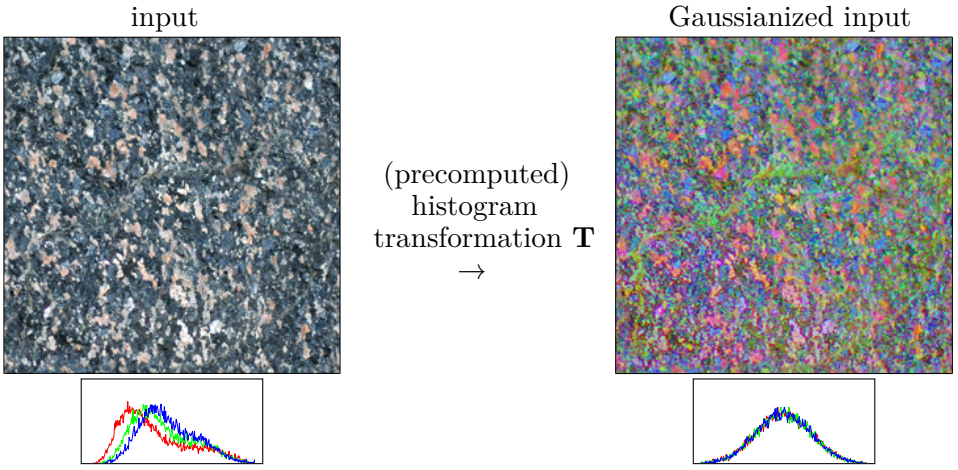


Figure 4.1: Transforming the histogram per channel to get a Gaussian distributed one. Figure adapted from Fig.8 in [HN18].

infinite Gaussian distribution.

$$\mathbf{CDF}_G(x,\sigma) = \frac{1}{2}\left(1 + erf\left(\frac{2x-1}{2\sqrt{2\sigma^2}}\right)\right) \tag{4.1}$$

$$\mathbf{CDF}_G^{-1}(x,\sigma) = \frac{1}{2} + \sqrt{2\sigma^2}\cdot erf^{-1}(2x-1) \tag{4.2}$$

The final transformation $\mathbf{T}$ and its inverse $\mathbf{T}^{-1}$ are therefore retrieved as follows.

$$\mathbf{T}(x,\sigma) = \mathbf{CDF}_G^{-1}(\mathbf{CDF}_I(x),\sigma) \tag{4.3}$$

$$\mathbf{T}^{-1}(x,\sigma) = \mathbf{CDF}_I^{-1}(\mathbf{CDF}_G(x,\sigma)) \tag{4.4}$$

The transformation $\mathbf{T}$ is used once per pixel on the input texture to produce a Gaussianized texture which has a Gaussian distributed histogram. The inverse transformation $\mathbf{T}^{-1}$ will be used as the final step after the blending to restore the original histogram. $\mathbf{T}^{-1}$ is be stored inside a 1D look-up table that can be pre-computed.

Further Optimizations. Burley [Bur19] discusses various methods to avoid clipping artifacts in his paper which would exceed the scope of this thesis. However, many of these could also potentially be applied to the method introduced in Chapter 5.

4.1.3 Tiling

Before the actual blending can be done, those values have to be selected whith are being blended in the first place. For this, an equilateral triangle lattice and weighting is by done like for Simplex Noise [Per02]. First we apply an $offset$ and $scale$ to the UV-coordinates of the base triangle we want to render. The $offset$ is a 2-dimensional vector whereas the $scale$ is of scalar value.

$$uv_{new} = offset + (uv - 0.5)\cdot scale \tag{4.5}$$

Lattice Space. As we use the triangle lattice to determine our values, we then have to transform the coordinates uv_{new} into this lattice space. For this purpose, we use the following transformation matrices which guarantee us our desired properties. $\mathcal{T}_l^{uv}$ transforms lattice coordinates to UV-space and its inverse $\mathcal{T}_{uv}^{l}$ transforms UV-coordinates to lattice-space [Bit16].

$$\mathcal{T}_l^{uv} = \begin{pmatrix} triangleSize\cdot cos\left(\frac{\pi}{3}\right) & triangleSize\cdot sin\left(\frac{\pi}{3}\right) \\ triangleSize & 0 \end{pmatrix} \tag{4.6}$$

$$\mathcal{T}_{uv}^{l} = (\mathcal{T}_l^{uv})^{-1} \tag{4.7}$$

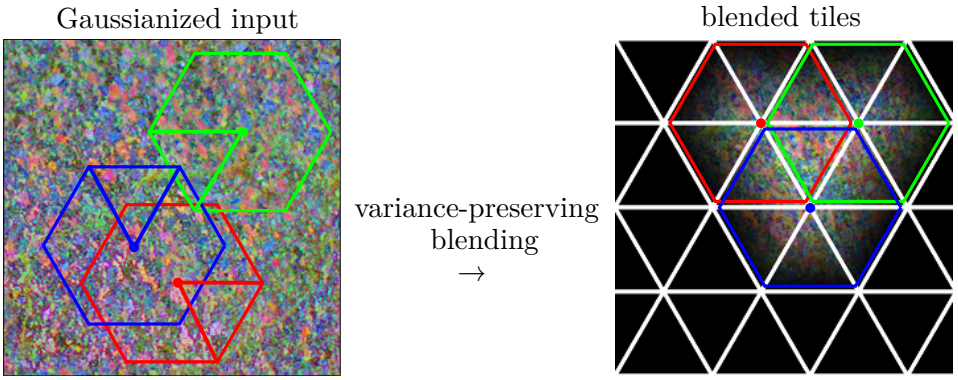


Figure 4.2: On the **left** there is the UV-space of the Gaussianized input texture. The center points of the hexagons on the **left** are hashes of the lattice-coordinates on the **right**. Blending is performed for a point inside the lattice triangle on the **right** by fetching 3 values, one for each hexagon, based on the position of the point relative to the center of the hexagon. The areas, from which values are fetched, from within the hexagons are the highlighted triangles on the **left**. The weights used for blending are the barycentric coordinates of the point. Figure adapted from Fig.8 in [HN18].

The *triangleSize* determines how long an edge of an equilateral triangle is in UV-space and a value of around $\frac{1}{4}$ works pretty well in general [HN19].

Texture Fetching. After transforming uv_{new} with $\mathcal{T}_{uv}^{l}$ into lattice-space, the lattice triangle is selected which surrounds these coordinates. As this is done in lattice-space, these lattice triangles are no longer equilateral, but isosceles triangles with a right angle and the length of the isosceles edges is one. Each of the vertices of this lattice triangle is associated with a random point in texture space through hashing. This way each of the vertices can contribute to 6 different lattice triangles and this guarantees that there are no breaks in the final texture. The hashed point can be thought of as the center point of a hexagon in UV-space from which the values are fetched according the offset of uv_{new} from that center point. Through accessing the texture this way, the content of each lattice triangle is a result of blending 3 triangles from UV-space together.

4.1.4 Variance-Preserving Blending

Weights for Blending. For the blending operator, 3 weights w_1, w_3 and w_3 are needed. These determine the influence of the values that are fetched for each lattice triangle vertex. The closer our sample point to a vertex, the higher is its influence on the blended result. Therefore the use of barycentric coordinates comes in handy, as they are one at their respective vertex and fade to zero the farther away they are from it.

Blending. The actual blending is done using a variance-preserving blending operator introduced by [HN18] which is exactly histogram-preserving if the values are Gaussian distributed [HN18]. That is why the histogram transformation in Section 4.1.1 was necessary. The blended result G is calculated with that operator:

$$G = \frac{w_1 v_1 + w_2 v_2 + w_3 v_3 - \frac{1}{2}}{\sqrt{(w_1)^2 + (w_2)^2 + (w_3)^2}} + \frac{1}{2} \tag{4.8}$$

The $\frac{1}{2}$ is the expectation of our Gaussian distribution. As noted before, our weights w_i are the barycentric coordinates of our sample point in relation to the surrounding lattice triangle, whereas the v_i are the values that got extracted from the 3 different hexagons

from the texture.

Inverse Transformation. To get the final result, G has to be transformed back via the inverse histogram transformation $\mathbf{T}^{-1}$ to restore the original histogram.

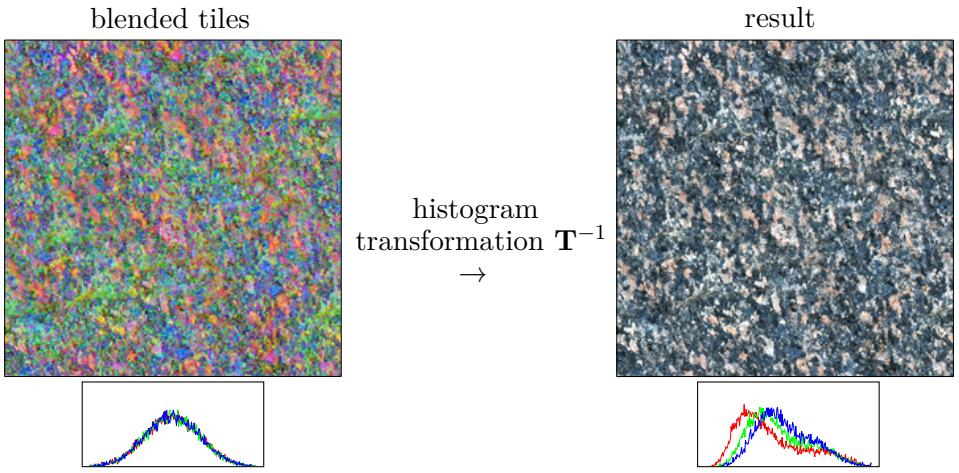


Figure 4.3: Transforming the histogram from the Gaussian distribution back to the original histogram. Figure adapted from Fig.8 in [HN18].

4.2 Tessellation-Free Displacement Mapping

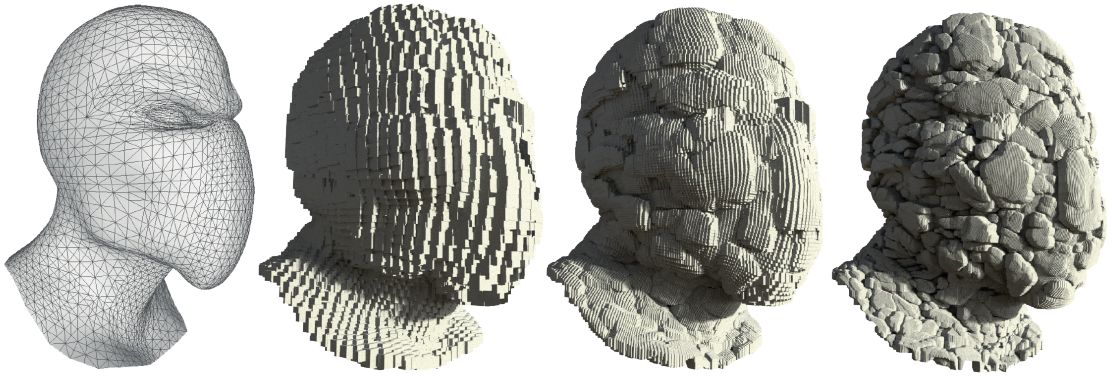


Figure 4.4: Base geometry on the **left**, different levels of detail of Tessellation-Free Displacement Mapping to the **right**. Figure from Fig. 6 in [TBS+21].

Although there are many ways to approach displacement mapping for rasterization, the way to go for ray tracing and offline rendering is pre-tessellating the displaced surface to the desired level of detail and applying the displacement on the vertices in advance. This comes at a high memory cost as millions of triangles have to be created in order to achieve this.

4.2.1 Overview

Tessellation-Free Displacement Mapping [TBS+21] is a new method to apply texture based displacement to a surface when ray tracing, without pre-tessellation of the base mesh. The major selling point of this approach is its reduced memory usage on the GPU which comes at the cost of performance. Thonat et al. use a custom acceleration structure on top

of the classic BVH to iteratively perform intersection tests against the displaced surface. They call it a Displacement-BVH, or short D-BVH, which is a texture based approach that computes bounding boxes on-the-fly inside an intersection shader.

4.2.2 Minmax-Mipmap

The main structure that is being worked on throughout the method is a structure similar to that of a mipmap. However, instead of reducing 4 texels of a level to their linear interpolation, their minimum and maximum is calculated and stored inside a texture. This serves as storage for information that can later be used to create a hierarchical intersection test that propagates from the highest mip level back to the original texture while being able to discard large parts of it along the way.

4.2.3 D-BVH Traversal

The traversal of the D-BVH tree inside the intersection shader will be described first and the details on how the bounding boxes are calculated per texel will be outlined afterwards.

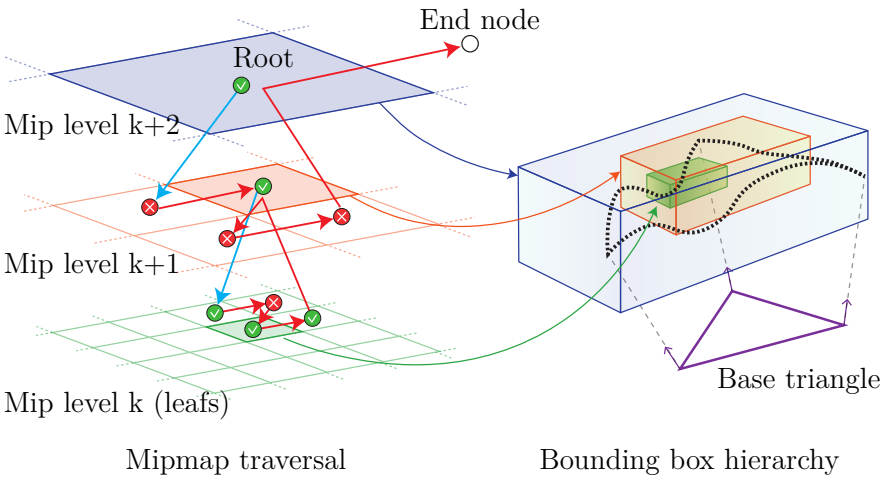


Figure 4.5: On the **left** the minmax mipmap is traversed where blue arrows are *down* operations and red arrows are *next* operations. Each *down* operation reduces the mip level while every 4th *next* operation on the same mip level calls the *up* operation which increases the Mip level. Traversal works similar as depth first search in a tree structure. On the **right** bounding boxes are calculated on-the-fly as the minmax mipmap is traversed for intersection tests. A green check-mark indicates a successful intersection with the corresponding bounding box while the red crossings are failed intersections. Figure by [TBS+21].

Root selection. Inside the intersection shader, the D-BVH is traversed by first selecting a root texel together with its LoD from which to start the traversal progress. The most important property of the root is that it completely encompasses the triangle in UV-space so that no texel outside the root is mapped onto the triangle. A naive approach for the root selection would be to just take the highest mip level, which is a one-by-one texture, and start the traversal process there. While this does not yield optimal performance, it is generally a correct root for an arbitrary triangle.

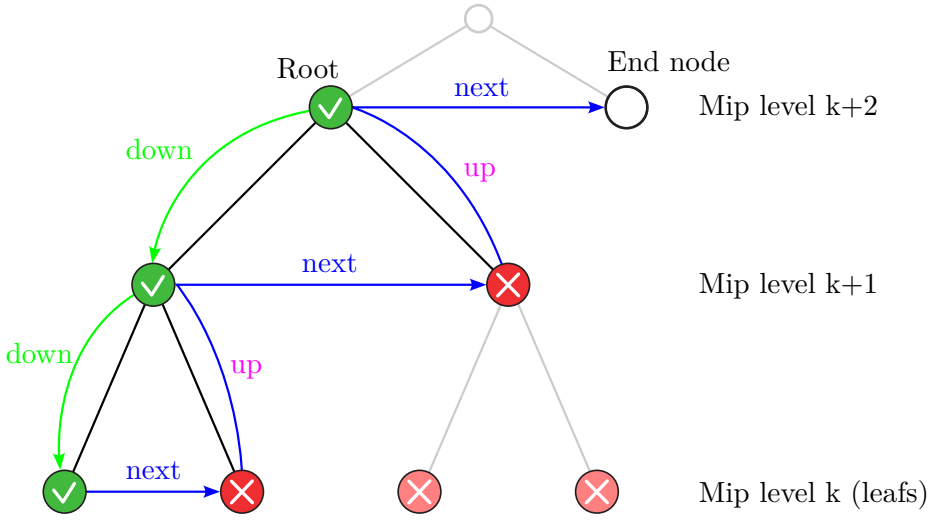


Figure 4.6: The D-BVH traversal works similar as this depth-first search (DFS) of a binary tree. *Next*, *up* and *down* operations work the same in DFS as in D-BVH traversal. Due to the the tree structure some leafs can be discarded without intersection (see the 2 bottom right leafs).

Algorithm 1 down function used in traversal, adapted from Algorithm 1 in [TBS+21].

```
function DOWN(Texel texel):
    texel.LoD--
    texel.ij *= 2
```

Algorithm 2 up function used in traversal, adapted from Algorithm 1 in [TBS+21].

```
function UP(Texel texel):
    ++texel.LoD
    texel.ij /= 2
```

Algorithm 3 next function used in traversal, adapted from Algorithm 1 in [TBS+21].

```
function NEXT(Texel texel):
    while true do
        switch 2(texel.i%2) + texel.j%2
            case 1 do
                --texel.j
                ++texel.i
                return
            case 3 do
                up(texel)
                continue
            else
                ++texel.j
                return
```

Algorithm 4 Pseudocode for the traversal of the D-BVH. The *down*, *up* and *next* define the traversal order. The *box* function computes the AABB for a texel (see Section 4.2.4), *local_intersection* intersects the generated surface (see Section 4.2.5). The *root* function selects a root for traversal (see Section 4.2.3) and the *miss* function is based on AABB intersection. Adapted from Algorithm 1 in [TBS+21].

```
function TRAVERSAL(ray,triangle,targetLoD):
    intersection = emtpy
    texel, endtexel = root(triangle)
    next(endtexel)
    while texel ≠ endtexel do
        if outside(texel,triangle) or miss(ray,box(texel)) then
            next(texel)
        else if texel.LoD ≤ targetLoD then
            if hit = local_intersection(texel,ray) then
                update intersection from hit
            next(texel)
        else
            down(texel)
    return intersection
```

Traversal. If the intersection test with the root texel is successful, we move down a mip level. As one texel of the higher mip level encompasses 4 texels of the lower mip level, intersection tests against all 4 texels have to be made. This works similar as a depth first search with a binary tree like in Figure 4.6 but instead of 2 options per node we have 4 and like this we are able to discard large areas of the texture easily. The order in which the 4 texels are traversed can be fixed although according to Thonat et al. a spatial sorting yields slightly faster results for coherent rays [TBS+21]. When intersecting with the desired LoD the overall hit result is updated if this new hit is closer to the ray origin than the existing intersection point. After the whole tree got traversed, while constantly skipping nodes that are not intersected, the closest hit point is returned.

4.2.4 Bounding Box Calculation

The intersection test with a texel is done by calculating bounding boxes on-the-fly. To ensure that the calculated AABB is as precise as possible, it is first created in UV space over the triangle and is later transformed to world space. The goal of this process is to create a bounding box that contains the minimum and maximum of the displaced surface, similar to a BLAS construction.

Transformations. To transform between UV-coordinates and barycentric coordinates, we use $\mathcal{T}_{uv}^{b}$. This way we can calculate the position $\mathbf{P}$ and interpolated normal $\mathbf{N}$ of our triangle at a certain UV-coordinate through Equation 4.10 and Equation 4.11.

$$\mathcal{T}_{uv}^{b} = \begin{pmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ 1 & 1 & 1 \end{pmatrix}^{-1} \tag{4.9}$$

$$\mathbf{P}(u,v) = \begin{pmatrix} | & | & | \\ p_1 & p_2 & p_3 \\ | & | & | \end{pmatrix} \mathcal{T}_{uv}^{b} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{4.10}$$

$$\mathbf{N}(u,v) = \begin{pmatrix} | & | & | \\ n_1 & n_2 & n_3 \\ | & | & | \end{pmatrix} \mathcal{T}_{uv}^{b} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{4.11}$$

AABB in UV space. A two-dimensional AABB can be thought of as two separate 1-dimensional intervals, one for each axis. Each interval consists of a median point and a deviation from it. Affine Arithmetics (see Section 2.4)is an approach to work with such intervals while reducing any error accumulation that would occur when performing calculations on this interval. The extent of the interval in UV space is simply half the size of the texel (note that the extent goes in two directions: positive and negative) which can easily be calculated from the size of the original texture and the used LoD. The median point is calculated similarly:

$$medianU = 2^k \cdot \frac{i+0.5}{W}, \quad extentU = \frac{2^k}{2W} \tag{4.12}$$

$$medianV = 2^k \cdot \frac{j+0.5}{H}, \quad extentV = \frac{2^k}{2H} \tag{4.13}$$

$$\tag{4.14}$$

$$\left[\begin{pmatrix} u \\ v \\ 1 \end{pmatrix}\right] = \left[\begin{pmatrix} medianU, & extentU, & 0, & 0 \\ medianV, & 0, & extentV, & 0 \\ 1, & 0, & 0, & 0 \end{pmatrix}\right] \tag{4.15}$$

AABB in world space. The 2D bounding box is then transformed into a range of positions $\left[\mathbf{P}(u,v)\right]$ with Equation 4.10 and also into a range of normal directions $\left[\hat{\mathbf{N}}(u,v)\right]$ with Equation 4.11, all in world space. Matrix-vector multiplication of Affine Arithmetics needs to be used here. As the AABB should encapsulate the displaced surface, the height of the displacement is still needed to satisfy Equation 2.1. The range for the height can easily be fetched from the min-max-mipmap, as shown in Equation 4.16.

$$\left[h(u,v)\right] = \frac{1}{2}\left[\overline{M}_{i,j}^{k} + \underline{M}_{i,j}^{k}, 0, 0, \overline{M}_{i,j}^{k} - \underline{M}_{i,j}^{k}\right] \tag{4.16}$$

$$\left[\mathbf{S}(u,v)\right] = \left[\mathbf{P}(u,v)\right] + \left[h(u,v)\right]\left[\hat{\mathbf{N}}(u,v)\right] \tag{4.17}$$

After evaluating the displacement equation with Affine Arithmetics, a now world space AABB can be retrieved by taking the minimum and the maximum of the range and adding it to the median point. Note that $|\cdot|$ is the per channel absolute value, our $\mathbf{S}(u,v)$ is a 3-dimensional vector after all.

$$\mathbf{S}(u,v) \in \left[\mathbf{S}_c - |\mathbf{S}_u| - |\mathbf{S}_v| - |\mathbf{S}_K|, \mathbf{S}_c + |\mathbf{S}_u| + |\mathbf{S}_v| + |\mathbf{S}_K|\right] \tag{4.18}$$

All that is left to do is to intersect this world space AABB with the ray and we get an intersection result.

4.2.5 Local Intersection Tests

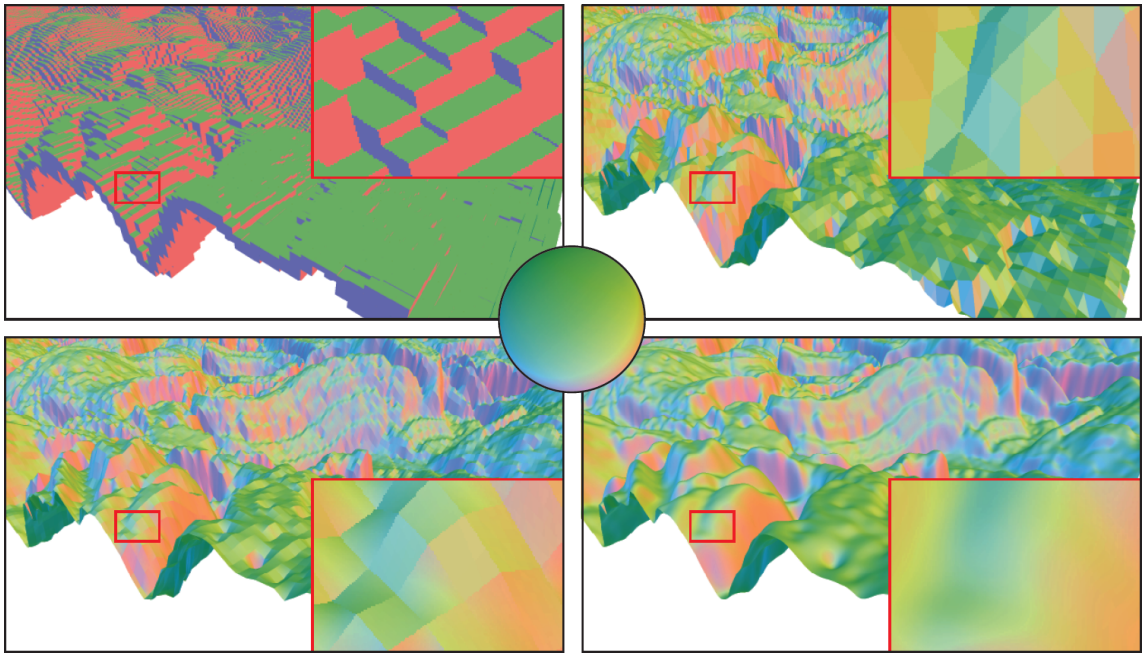


Figure 4.7: Different local intersection tests and their geometric normals (which are visualized here). Leaf boxes on the **top left**, local triangulation with 2 triangles per texel on the **top right**, bilinear height sampling on the **bottom left** and B-spline height sampling on the **bottom right**. Upper half hemisphere of directions is shown in the center. Figure and description from Fig. 7 in [TBS+21].

Thonat et al. suggest performing local intersection tests when the leaf boxes are reached [TBS+21]. This way, the geometric normal that results from these further intersections helps when shading the displaced surface. The results of different local intersection tests can be seen in Figure 4.7.

Bounding Box. The simplest method for a local intersection test is just reusing the result from the leaf box intersection. This produces the fastest results of all the local intersection methods, but the extracted normals are only axis aligned vectors. Also on closer inspection the individual boxes can be seen.

Local Triangulation. Two triangles are created on-the-fly and intersected with a ray. The result someone represents that of pre-tessellation. Thonat et al. note however that the surface can have small discontinuities at the base primitive borders [TBS+21].

Bilinear and B-spline Height Sampling. These more performance costly intersection tests are done against an on-the-fly created parametric surface. It produces the smoothest results, but as with the triangulation, discontinuities can occur along the base primitive borders. For mathematical details, see Appendix B in [TBS+21].

4.2.6 Triangle BLAS

To invoke the intersection shader in a ray tracing pipeline, we need to calculate a bounding box for our triangle to be used as a BLAS in the acceleration structure (see Section 2.2). It is important that this bounding box completely encompasses the displaced surface, therefore conservative bounds for the height are needed for the AABB calculation. After a

range for the height is chosen, the bounding box calculation can be done via Affine Arithmetics like in Section 4.2.4. A naive approach would be to create a 2D AABB around the triangle in UV-space and take the global minimum and maximum of the displacement map as input for the height and compute the world space 3D AABB using Affine Arithmetics like before. The resulting bounding box is not the tightest fit, a better method for the calculation is mentioned in the next section.

4.2.7 Optimization

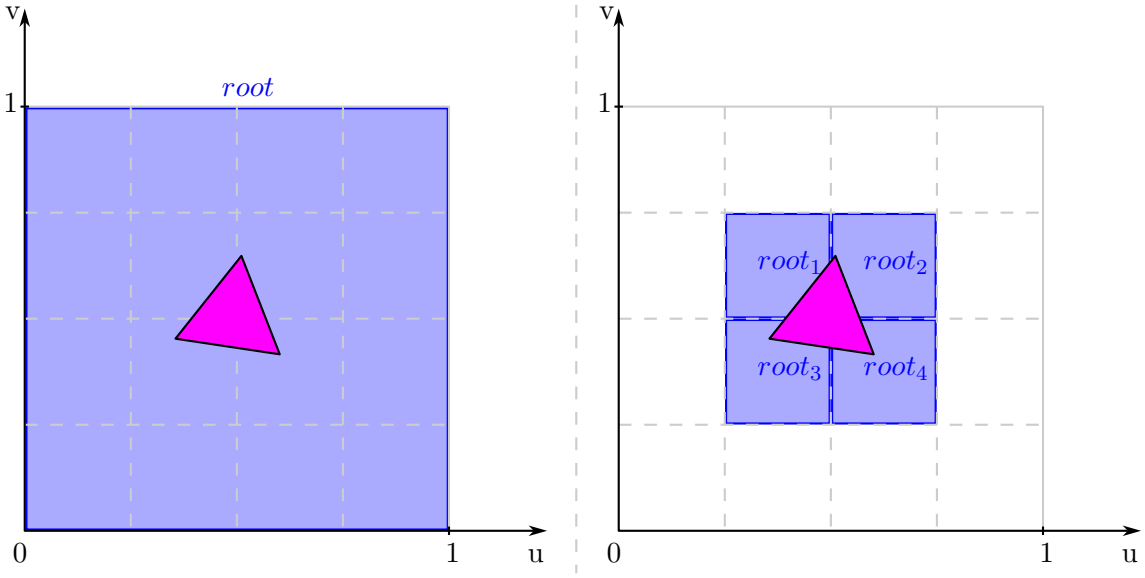


Figure 4.8: On the **left** side only one root is selected and because the triangle contains the middle point of the texture, this root has to be the highest mip level (1x1 texture). On the **right** side root selection is done by selecting up to 4 neighbouring texels which enables discarding large portions of the texture before traversal. The root texels are highlighted in blue, the base triangle in pink is the same for both **left** and **right**.

Root Selection. As mentioned before, taking the one-by-one mip level as root doesn't always yield fast results. Especially for triangles that are small in UV-space, large portions of the texture can be discarded when choosing a better root. A slightly better approach is, instead of choosing the highest mip level, to find the texel of the lowest mip level which fully encompasses the triangle in UV-space. While being a better root for most triangles, there are some edge cases where the calculated root is still not optimal. This applies when the triangle in UV space overlaps two large texels with the worst case being that it contains the middle of the texture. As suggested by Thonat et al. [TBS+21] choosing up to 4 neighbouring roots solves this problem (see Figure 4.8). The traversal will be done for each of those roots and like before the closest hit of them will be returned as the intersection result.

Early Texel Discard. For texels that are completely outside of the triangle in UV space, no bounding box has to be created nor intersected. Thonat et al. [TBS+21] perform a triangle-square collision in the 2-dimensional UV-space to determine if an AABB creation is necessary. The collision algorithm that is used is a modification of the GJK algorithm [GJK88] which uses the Minkowski difference to determine whether the square, which in our case is a texel, is outside the triangle or not. For implementation details, see Algorithm 4 in Appendix in [TBS+21].

BLAS Construction. Another performance optimization can be done when calculating the initial BLAS for the displaced triangle which invokes the intersection shader when the ray tracing pipeline finds an intersection with it. Better bounds for the height can be found by performing the root selection that was highlighted in this section to find up to 4 roots and to then take the minimum and maximum of those roots. According to Thonat et al. [TBS+21] further optimization can be done by considering the triangle a union of 3 parallelograms and performing the AABB creation for each of those with Affine Arithmetics and merging the boxes later.

5. Procedural Displacement by Tiling and Blending

The main focus of this thesis is to use a texture that is procedurally generated from the method introduced in Section 4.1 as a displacement texture that is being applied to the surface in a similar manner as Tessellation-Free Displacement Mapping from Section 4.2. As this procedural texture only exists implicitly, some modifications have to be made to find the correct intersection point with the ray. This method will be evaluated per displaced triangle. This world space triangle, which is the base geometry, will be called base triangle throughout this chapter.

5.1 Overview

The task at hand is to find an intersection point with the displaced surface given a ray. To iteratively narrow down this intersection point, the lattice-space, in which the implicit texture exists in, can be traversed. This is done in two steps: First traverse the lattice-space (Section 5.3) until a one-by-one region is found and then perform the triforce traversal (Section 5.4). The one-by-one region can be split up into two lattice triangles and for each of those the triforce traversal is performed. Using the lattice triangles for the final traversal is important because the value of the blended texture depends on the vertices of these lattice triangles as mentioned in Section 4.1.1. Therefore the only goal of the first traversal, the lattice traversal, is to get to two lattice triangles in order to perform the second traversal, the triforce traversal, on them to find the final intersection result. Axis aligned bounding boxes are created on-the-fly inside each of those traversals to perform intersection tests against. The triforce traversal will also make use of the minmax mipmap that is being created from the input texture.

5.2 Pre-Calculations

5.2.1 Gaussinization

As described in Section 4.1.1 a histogram transformation has to be applied to our input texture in advance. The process to do so is exactly the same, by applying the $\mathbf{T}$ transformation to every pixel. The inverse transformation $\mathbf{T^{-1}}$ will also be used throughout this chapter whenever a blended value has to be evaluated.

5.2.2 Minmax Mipmap

After the histogram transformation, the minmax mipmap is calculated on this gaussinized texture. The procedure remains the same as in Section 4.2.2. It will be used throughout the trifroce traversal in Section 5.4 for height value approximation.

5.3 Lattice Traversal

The lattice traversal works similar to the traversal used in Section 4.2, only with small modifications. Because we work in lattice space, we first have to calculate the lattice coordinates of each vertex of our base triangle to get its projection in the lattice space. This is normal coordinate transformation, the same one that is used in Section 4.1.1, which adds an optional offset and a scale to the coordinates. After that a root has to be found which fully encompasses our base triangle in lattice space. As we are working with squares as nodes here, we will also subdivide one node further into 4 nodes, each with only have the size of their parent. This process is repeated until a one by one square in lattice space is reached.

5.3.1 Root Selection

BLAS Calculation. Our first intersection with the base triangle is done by adding a BLAS to our acceleration structure, its AABB being calculated the naive way from Section 4.2.4. As the variance-preserving blending operator is not a convex operation, newly calculated bounds have to be used as minimum and maximum height. When the weights are unknown, the following formula gives us the global minimum $\underline{M}_g$ and the global maximum $\overline{M}_g$ for the blending operator.

$$\underline{M}_g = min(\underline{M}^0_{i,j}, \frac{\underline{M}^0_{i,j} - 0.5}{\frac{\sqrt{3}}{3}} + 0.5) \tag{5.1}$$

$$\overline{M}_g = max(\overline{M}^0_{i,j}, \frac{\overline{M}^0_{i,j} - 0.5}{\frac{\sqrt{3}}{3}} + 0.5) \tag{5.2}$$

$\underline{M}^0_{i,j}$ and $\overline{M}^0_{i,j}$ are the global minimum and maximum of our gaussianized input texture. Before using $\underline{M}_g$ and $\overline{M}_g$ as our bounds for the height, we have to de-gaussianize these values by applying the inverse transformation $\mathbf{T^{-1}}$. As the histogram transformation is a combination of monotonically increasing operations, it itself is monotonically increasing. Therefore the bounds remain true even after applying $\mathbf{T^{-1}}$.

Root Selection. The two important properties of the root are full encapsulation of the base triangle in lattice space and a size of the power of two. This way, subdivisions until reaching the desired one-by-one square can be done by dividing the size of the square in half with each iteration, leaving us 4 more children nodes for each parent node. As we are not bound to any underlying mipmap structure here, our bottom left point of our root will be the minimum on the lattice x-axis and the lattice y-axis floored to the next integer value. All that is left to do is to calculate a size of our root square so the base triangle in lattice space fits into it and its value is a power of two. It is easy to calculate the maximum size of the base triangle and finding the next highest power of two value by iteratively multiplying the size by two while starting at one.

5.3.2 Coordinate Transformation

As mentioned before, the lattice traversal is done similarly as the traversal in Section 4.2 with one difference being the coordinate transformation and the height calculation. As our

traversal does not take place in UV-space, we can't use Equation 4.9 as our transformation matrix for calculating the bounding box, but instead we need the following matrix which transforms lattice coordinates to barycentric coordinates of our base triangle. The l_i are the lattice coordinates at the vertex p_i and $l_i.x$ being its coordinate along the x-axis and $l_i.y$ its coordinate along the y-axis.

$$\mathcal{T}_l^b = \begin{pmatrix} l_1.x & l_2.x & l_3.x \\ l_1.y & l_2.y & l_3.y \\ 1 & 1 & 1 \end{pmatrix}^{-1} \tag{5.3}$$

$$\tag{5.4}$$

We can now calculate our $\mathbf{P}(u, v)$ and $\mathbf{N}(u, v)$ like before in Equation 4.10 and Equation 4.11 but instead of using $\mathcal{T}_{uv}^b$ we use our newly calculated matrix $\mathcal{T}_l^b$. This way we can transform between our lattice coordinates and the position and normal of our base triangle.

5.3.3 Height Calculation

We know that the height is certainly bound by the global blended minimum $\underline{M}_g$ (Equation 5.1) and global maximum $\overline{M}_g$ (Equation 5.2) that our procedural texture can be. The blending operator is variance-preserving, so this is actually not a bad way to calculate the bounds. Possibilities of further optimization will be discussed in Chapter 7. Again, $\underline{M}_g$ and $\overline{M}_g$ have to be de-gaussianized through $\mathbf{T}^{-1}$. The AABB can now be created like in Section 4.2.4 using Affine Arithmetics and our adjusted height bounds and coordinate transform.

5.4 Triforce Traversal

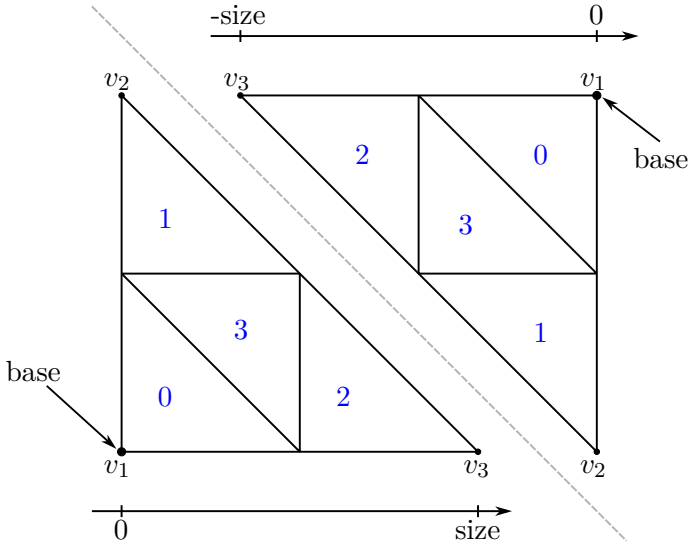


Figure 5.1: Traversal order, subdivision scheme and terminology is shown for the subdivision lattice triangle. These are the two configurations that a subdivision triangle can be in, with the base at the bottom left or at the top right. For the triangles with the base at the top right, the size is a negative value. Traversal order of the lattice triangles is highlighted in blue. It represents the *next* function from Algorithm 6.

After finding an intersection with the AABB created from a one-by-one square of lattice space, the triforce traversal is performed on the two lattice triangles. As the underlying geometry is this time not a square but a triangle, the traversal differs quite a bit from Tessellation-Free Displacement Mapping (see Algorithm 5 and Algorithm 6) while the on-the-fly calculation of the bounding boxes remains similar. As before, the traversal happens in lattice space where our lattice triangles are isosceles right triangles.

Algorithm 5 The underlying data structure used for subdivision triangles is the **STriangle**. The *base* and *size* can be used to get the extent of the triangle, *counter* is used for traversal order. *down* and *up* functions are used like in the traversal in Algorithm 4.

```
struct STriangle {
    ivec2 base;
    int size;
    int counter;
};
function DOWN(STriangle sTri):
    sTri.size /= 2
    sTri.counter = 0
function UP(STriangle sTri):
    sTri.base += sTri.size
    sTri.range *= −2
    ▷ temp only for counter recalculation
    ivec2 temp = sTri.base%sTri.size ∗ 2
    temp = ivec2(temp / sTri.size) %2
    sTri.counter = 2 ∗ temp.x + temp.y
```

Algorithm 6 The *next* function is used like in the traversal in Algorithm 4.

```
function NEXT(LTriangle sTri):
    while true do
        switch 2(texel.i%2) + texel.j%2
            case 1 do
                sTri.base += (sTri.size, −sTri.size)
                ++sTri.counter
                return
            case 2 do
                sTri.base.y += sTri.size
                sTri.range = −sTri.range
                ++sTri.counter
                return
            case 3 do
                up(sTri)
                continue
            else
                sTri.base.y += sTri.size
                ++sTri.counter
                return
```

5.4.1 Subdivision

Each isosceles right triangle can be subdivided into 4 similar triangles (see Figure 5.1)on each of which this subdivision can then be performed again. Any of these isosceles right

triangles, that originated through subdivision will be called subdivision triangles. For convenience, the lattice triangle which is the root for the triforce traversal, can be also viewed as a subdivision triangle. As the bounds of the lattice triangle are still inside a one-by-one square in lattice space, we can subtract the bottom left corner from it and thus our further calculations are easier because our coordinate origin is the bottom left corner of the one-by-one square. We also want to subdivide this one-by-one space further and to avoid floating point operations, we can just specify a certain power of two range that determines how often we perform the subdivision at max. This also defines the quality of our displaced surface. The higher the range, the more subdivisions can be done and the results of the method will be more precise.

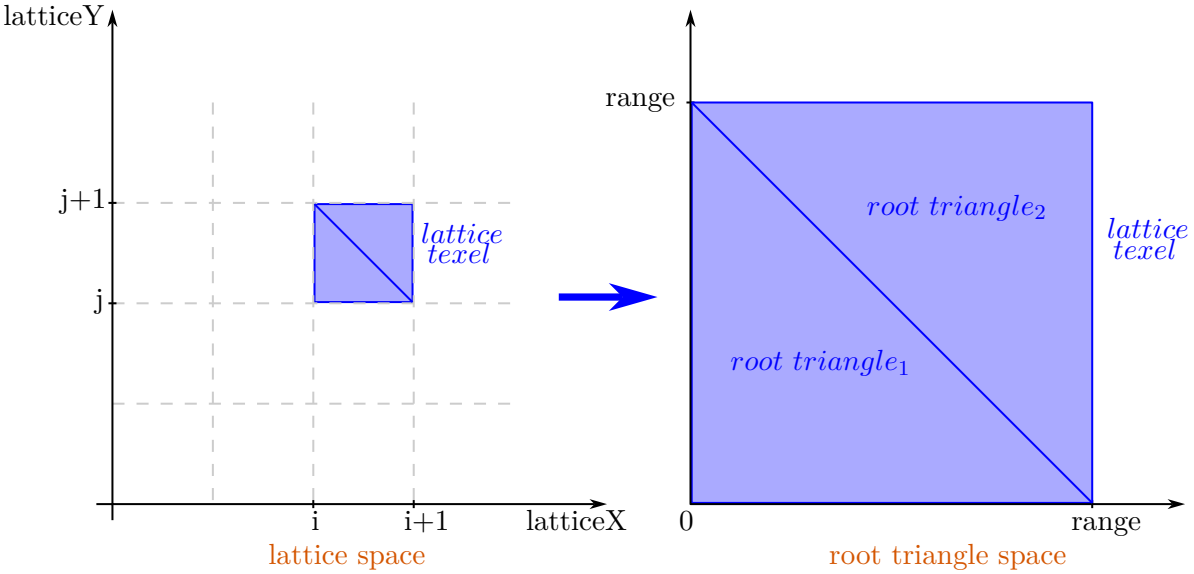


Figure 5.2: One leaf of the lattice traversal (**left**) gets converted into two root lattice triangles (**right**), on each of which the triforce traversal is performed.

Data Structure. To access our subdivision triangles, we can store the position of our vertex where the angle is 90 degrees (also called base vertex) alongside with the current size of the triangle (see textbfSTriangle in Algorithm 5). The size does not only determine the length of the legs of the subdivision triangle but also the direction into which the next vertices are. Because we have some subdivision triangles that have their base at the bottom left and some that have their base at the top right, we can perform all our calculations the same on both of them with the negative size trick. The vertices v_i in lattice space of this subdivision triangle can be calculated as follows:

$$v_1 = base \tag{5.5}$$
$$v_2 = base + (0, size) \tag{5.6}$$
$$v_3 = base + (size, 0) \tag{5.7}$$

Each subdivision triangle is also associated with an integer counter that ranges from 0 to 3. This slightly reduces the computations when calling the *next* operation and also shows the traversal order.

Traversal Order. The overall structure of the traversal remains similar as in Section 4.2 although the *next*, *up* and *down* operations have to be altered to fit with our subdivision triangle data structure (see Algorithm 5 and Algorithm 6).

5.4.2 Bounds for Height

First of all, we will calculate the values v_1, v_2 and v_3 that are to be blended together.

Triangles in UV-space. Our first step is to associate each of the vertices of our root triangle with a hexagon from the gaussianized texture as we did in Section 4.1. For each of those hexagons we then determine the positions of all 3 vertices of our subdivision triangle in UV-space the same way as we did for the position of our sample point in Section 4.1. This leaves us with 3 triangles in UV-space, one for each hexagon.

Height Bounds per Triangle. The following is then done to each subdivision triangle to determine bounds for the minimum and maximum value of our gaussianized texture. We can reuse the algorithm from the root selection in Section 4.2.7. This way we only need to evaluate up to 4 texels of the minmax mipmap to get cheap bounds for our height. Of course we still need to compute the minimum and the maximum of those texels so we are left with 2 values. $\underline{h}_i$ is the minimum and $\overline{h}_i$ the maximum of our height value h_i.

5.4.3 Blending

As the result of this traversal should ultimately converge towards a procedural texture created by Tiling and Blending, the blending process has to be done inside this triforce traversal. The blending will be done like in Section 4.1 using 3 values and weighting them with their barycentric coordinates inside the root lattice triangle. However in our case we do not evaluate our procedural texture at just one point in lattice space, but instead we evaluate the subdivision triangles that are being traversed inside the triforce traversal. Because of this we are not left with a single set of barycentric coordinates for our blending process but instead have to deal with a range of them. To get conservative bounds for blending, the bounding is done separately for the numerator and denominator of the blending operator.

Numerator Bounds. Fortunately, due to the construction of our subdivision algorithm, the edges of our subdivision triangles are all parallel to the edges of the root lattice triangle. We can apply the theorem of intersecting lines here and find out that for each edge, one barycentric coordinate relative to the root lattice triangle is constant. For edges that are on top of an edge of the root lattice triangle, the barycentric coordinate relative to the opposite point is 0. With this information we can deduce that the barycentric coordinates at the vertices of our subdivision triangle provide us exactly with the weights we need for bounding the numerator of our blending operator (Equation 4.8). So for bounding the numerator, it has to be evaluated 3 times, once per vertex of the subdivision triangle. h_i is the height value at vertex v_i whereas the $b_i(v_i)$ are the 3 barycentric coordinates b_1, b_2 and b_3 at the vertex v_i.

$$numer_1 = b_1(v_1) \cdot h_1 + b_2(v_1) \cdot h_2 + b_3(v_1) \cdot h_3 - \frac{1}{2} \tag{5.8}$$

$$numer_2 = b_1(v_2) \cdot h_1 + b_2(v_2) \cdot h_2 + b_3(v_2) \cdot h_3 - \frac{1}{2} \tag{5.9}$$

$$numer_3 = b_1(v_3) \cdot h_1 + b_2(v_3) \cdot h_2 + b_3(v_3) \cdot h_3 - \frac{1}{2} \tag{5.10}$$

The minimum of the numerator $\underline{numer}$ is calculated by taking the minimum of the $numer_i$ and its maximum $\overline{numer}$ is calculated by taking the maximum of $numer_i$.

Denominator Bounds. The evaluation at the 3 vertices does not suffice to create conservative bounds for the denominator. To determine the bounds of it, Affine Arithmetics

can used. Because $b_3 = 1 - b_1 - b_2$, we only need a range for b_1 and b_2:

$$\sqrt{(b_1)^2 + (b_2)^2 + (b_3)^2} = \sqrt{2((b_1)^2 - b_1 + (b_2)^2 - b_2 + b_1 b_2) + 1} \tag{5.11}$$

As noted before, the one barycentric coordinate along the edges of our subdivision triangle remains constant. This also means that for two vertices of the subdivision triangle, one barycentric coordinate is constant. We can thus calculate our range of b_1 and b_2 by taking the minimum and maximum of the barycentric coordinates at the 3 vertices of the subdivision triangle, which we already needed for the bounds of the numerator. This gives us $\underline{b}_1$ as minimum of b_1 and $\overline{b}_1$ as the maximum of b_1. $\underline{b}_2$ and $\overline{b}_2$ are calculated the same way.

$$\left[b_1\right] = \frac{1}{2}\left[\left(\overline{b}_1 + \underline{b}_1\right), \left(\overline{b}_1 - \underline{b}_1\right), 0, 0\right] \tag{5.12}$$

$$\left[b_2\right] = \frac{1}{2}\left[\left(\overline{b}_2 + \underline{b}_2\right), 0, \left(\overline{b}_2 - \underline{b}_2\right), 0\right] \tag{5.13}$$

With the Affine Arithmetics we can now evaluate the denominator $\left[denom\right]$.

$$\sqrt{\left[denom\right]} = \sqrt{2\left(\left[b_1\right] \cdot \left[b_1\right] - \left[b_1\right] + \left[b_2\right] \cdot \left[b_2\right] - \left[b_2\right] + \left[b_1\right] \cdot \left[b_2\right]\right) + 1} \tag{5.14}$$

As the square root of an interval is just the square root of its bounds, we need to calculate this to get the bounds for the denominator. As before, $\underline{denom}$ is the minimum and $\overline{denom}$ the maximum value.

$$\underline{denom} = \sqrt{denom_c - |denom_u| - |denom_v| - |denom_K|} \tag{5.15}$$

$$\overline{denom} = \sqrt{denom_c + |denom_u| + |denom_v| + |denom_K|} \tag{5.16}$$

The barycentrics range $\left[b_1\right]$ and $\left[b_2\right]$ is forming a square rather than a triangle. This way, for subdivision triangles that share an edge with the hypotenuse of the root lattice triangle this range exceeds the bounds of the root lattice triangle. To compensate unwanted values beyond those possible, we have to clamp this $\underline{denom}$ and $\overline{denom}$ to their global minimum ($= \frac{\sqrt{3}}{3}$) and global maximum ($= 1$).

Blending Bounds. To get the final bounds for the blending operator within our subdivision triangle, the numerator bounds and the denominator bounds need to be combined. As we have a minimum and maximum for each the numerator and denominator, we get 4 possible combinations of them. Evaluating all of these is necessary as the numerator can have positive or negative values and the denominator is ≤ 1 but $\geq \frac{\sqrt{3}}{3} \approx 0.577$ which means it can scale those values up.

$$blend_1 = \underline{numer} \,/\, \underline{denom} + \frac{1}{2} \tag{5.17}$$

$$blend_2 = \underline{numer} \,/\, \overline{denom} + \frac{1}{2} \tag{5.18}$$

$$blend_3 = \overline{numer} \,/\, \underline{denom} + \frac{1}{2} \tag{5.19}$$

$$blend_4 = \underline{numer} \,/\, \overline{denom} + \frac{1}{2} \tag{5.20}$$

The minimum $\underline{blend}$ can be evaluated by taking the minimum of $blend_i$ and the maximum $\overline{blend}$ by taking the maximum of $blend_i$. The only thing left to do for the bounds calculation is to de-gaussianize these values by applying the inverse transformation $\mathbf{T}^{-1}$

(Equation 4.4) to it. As $\mathbf{T}^{-1}$ is a monotonous increasing function, the application does not change the minimum or maximum.

Fast Barycentrics Calculation. It is easy to calculate our barycentric coordinates relative to the root lattice triangle because of its properties. Two of its edges are aligned with the axes and $v_1 = (0, 0)$. As our root lattice triangle can have two orientations we have to distinct between those. We first look at the root lattice triangles with a positive size. We know that $v_2 = (size, 0)$ is the only vertex containing a y-coordinate other than 0 and $v3 = (0, size)$ is the only vertex containing a x-coordinate other than 0. This enables us to easily calculate the influence of v_2 and v_3 by taking the according coordinate of our point p, that we want to know our barycentric coordinates of, and normalizing it by dividing through the size of our root lattice triangle.

$$b_2 = \frac{p.y}{root.size} \tag{5.21}$$

$$b_3 = \frac{p.x}{root.size} \tag{5.22}$$

$$b_1 = 1 - b_1 - b_2 \tag{5.23}$$

For root lattice triangles with a negative size, we can first translate the root lattice triangle such that $v_1' = (0, 0)$ by subtracting $v_1 = (size, size)$. We then have similar properties with v_2 being the only point containing a y-coordinate other than 0 and the same for v_3 and the x-coordinate. The barycentric coordinates calculation remains the same but instead of p we are calculating them using p':

$$p' = p - v_1 \tag{5.24}$$

Note that in the case where we use p' our size is always negative.

5.4.4 Bounding Box Calculation

To test the ray against intersection, a bounding box has to be created from our subdivision triangle together with the bounds for the height. This is done very similar as in Section 4.2.4 with only a few modifications.

AABB in lattice space. Because our whole traversal takes place in lattice space, we also calculate a 2D bounding box in this space and then transform it into world space using Affine Arithmetics (see Section 2.4). For this we need our median and extent in lattice space and we can calculate them using our subdivision triangle data structure. The 2D vector $squareMin$ is the bottom left corner of the one by one square that called our triforce traversal, the $range$ is the value we chose on how to subdivide our one-by-one square which determined the level of detail. The $median$ is also a 2D vector containing the coordinates of the median point of our bounding box. As we are working with isosceles triangles, the $extent$ is only a one dimensional number that will be used for direction in x-axis and y-axis.

$$median = squareMin + base + \frac{1}{2 \cdot range}(size, size) \tag{5.25}$$

$$extent = \frac{1}{2 \cdot range} \tag{5.26}$$

Next the Affine Arithmetics for this are stored inside a vector before transforming it from lattice space into world space like in Equation 4.15.

$$\left[\begin{pmatrix} l_x \\ l_y \\ 1 \end{pmatrix}\right] = \left[\begin{pmatrix} median.x, & extent, & 0, & 0 \\ median.y, & 0, & extent, & 0 \\ 1, & 0, & 0, & 0 \end{pmatrix}\right] \tag{5.27}$$

Then we transform this 2D lattice space bounding box into a range of positions $\left[\mathbf{P}(l_x, l_y)\right]$ and a range of normals $\left[\hat{\mathbf{N}}(l_x, l_y)\right]$, but instead of using the $\mathcal{T}^b_{uv}$ matrix from Equation 4.9, we use our lattice to barycentrics matrix $\mathcal{T}^b_l$ from Equation 5.3.

$$\left[\mathbf{P}(l_x, l_y)\right] = \begin{pmatrix} | & | & | \\ p_1 & p_2 & p_3 \\ | & | & | \end{pmatrix} \mathcal{T}^b_l \left[\begin{pmatrix} l_x \\ l_y \\ 1 \end{pmatrix} \right] \tag{5.28}$$

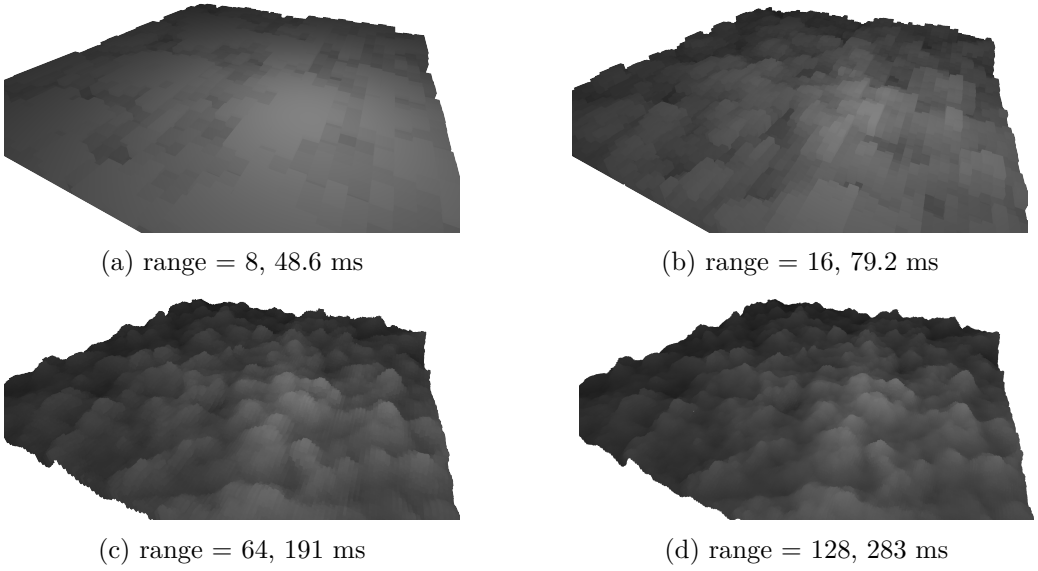
$$\left[\hat{\mathbf{N}}(l_x, l_y)\right] = \begin{pmatrix} | & | & | \\ p_1 & p_2 & p_3 \\ | & | & | \end{pmatrix} \mathcal{T}^b_l \left[\begin{pmatrix} l_x \\ l_y \\ 1 \end{pmatrix} \right] \tag{5.29}$$

For the final AABB calculations, we use the blended and de-gaussianized bounds of the height h_{min}, h_{max}, which we calculated in Section 5.4.2. Just like in Equation 4.17 we calculate our displaced surface and from there an AABB.

$$\left[\mathbf{S}(l_x, l_y)\right] = \left[\mathbf{P}(l_x, l_y)\right] + \left[(h_{min}, h_{max})\right] \left[\hat{\mathbf{N}}(l_x, l_y)\right] \tag{5.30}$$

$$\mathbf{S}(l_x, l_y) \in \left[\mathbf{S}_c - |\mathbf{S}_u| - |\mathbf{S}_v| - |\mathbf{S}_K|, \mathbf{S}_c + |\mathbf{S}_u| + |\mathbf{S}_v| + |\mathbf{S}_K|\right] \tag{5.31}$$

6. Results



(a) range = 8, 48.6 ms

(b) range = 16, 79.2 ms

(c) range = 64, 191 ms

(d) range = 128, 283 ms

Figure 6.1: Procedural displacement at varying LoDs with frametimes in milliseconds (ms).

In this chapter the results of the procedural displacement are outlined. All images are rendered with the Vulkan Ray Tracing Pipeline (the program is based on NVIDIAs Vulkan Ray Tracing Tutorial [LGBA] which uses nvpro core [NVI]) at 2560 x 1440 resolution on a GTX 1080 (images may be cropped to their content in this chapter). The scene contains one omnidirectional light, the color is representing the displacement amount, ranging from black (no displacement) to white (maximum displacement). Frametime are averages over the last 120 times and as the implementation is not fully optimized, frametimes may improve with future implementations.

6.1 Performance

	Faces	Frametimes for range:					
		8	16	64	128	256	512
plane	2	48.6 ms	79.2 ms	191 ms	283 ms	425 ms	656 ms
sphere	960	100 ms	193 ms	503 ms	748 ms	1058 ms	1533 ms

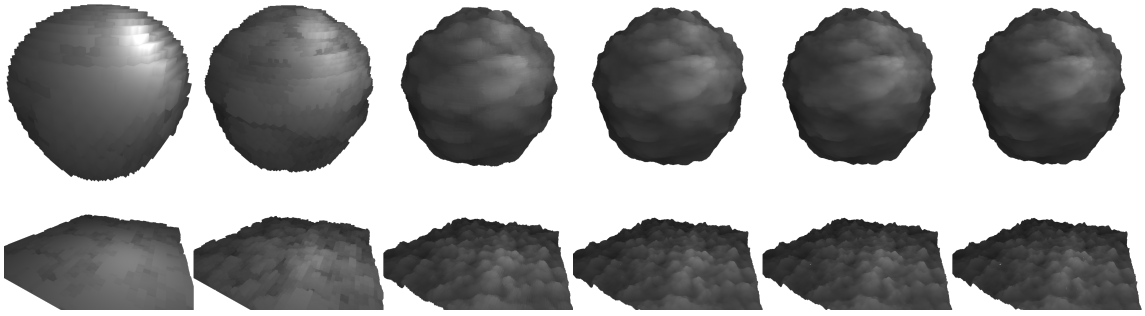
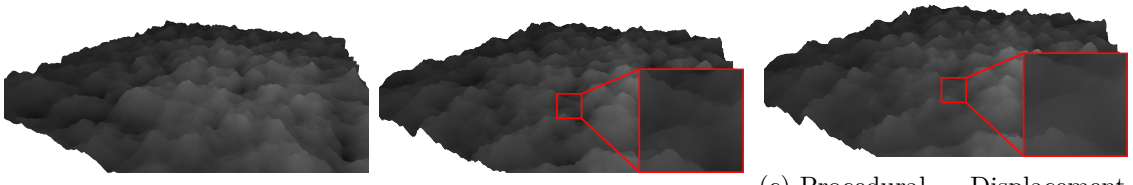


Figure 6.2: From left to right ranges 8, 16, 64, 128, 256 and 512

Table 6.1: Although the sphere has a much higher face count, its overall frametime is not proportionally slower than that of the plane. The UV-range covered by one of the triangles of the sphere is much smaller, so the traversal per triangle is quicker due to root selection and early discard.

The frametimes for simple scenes are mostly well below interactive rates. At a decent quality, these frametimes can be 20 times slower than performing a fixed step size raymarching on the surface at roughly the same quality (Figure 6.3). The frametime is however somewhat proportional to the level of detail, regulated via the range parameter, so lower LoDs are still a lot faster to render. As shown in Table 6.1 the frametime per triangle is lower when a small UV-range is mapped onto it due to root selection and early discard.

6.2 Quality



(a) TFDM, 31.5 ms (b) Raymarching, 29.3 ms (c) Procedural Displacement, 695 ms

Figure 6.3: On the **left** there is an image produced by Tessellation-Free Displacement Mapping with no procedural texture. In the *middle* there is raymarching with a fixed step size and a maximum of 300 steps per ray, performed on the procedural texture. On the **right** the new method is shown with a range of 256, also with the procedural texture. The underlying plane consists of 2 triangles with UV-coordinates ranging from (0,0) to (1,1).

The underlying triangle structure is not visible and for higher ranges even without local intersection test, the resulting surface looks good. Lower values for the range parameter can be used to create smaller LoDs on-the-fly which can also be used to prevent aliasing artifacts. In Figure 6.2 the increase in quality can be seen as the range parameter increases. For the test scenes, a range of 128 is usually enough to create a nice surface.

7. Discussion

As this thesis is a proof of concept, there are many options left to improve this method. In this chapter, the benefits, problems and possible solutions of this approach are discussed.

7.1 Memory Usage

This approach uses only a small portion of memory like Tessellation-Free Displacement Mapping. Using this method in real-time application however this not practical due to the frametimes which are too long compared to the desired 60 frames per second (=16.66 ms) that are common for smooth real-time graphics. It could still be used for offline rendering to reduce the already high memory usage by avoiding pre-tessellation of displaced surfaces.

7.2 Level of Detail

LoDs are provided out of the box by using different values for the range parameter. However, this way the lowest LoD will still perform a full lattice traversal and depending on how much UV-space is mapped onto the base triangle, this can still take a significant amount of frametime.

7.3 Performance Optimization

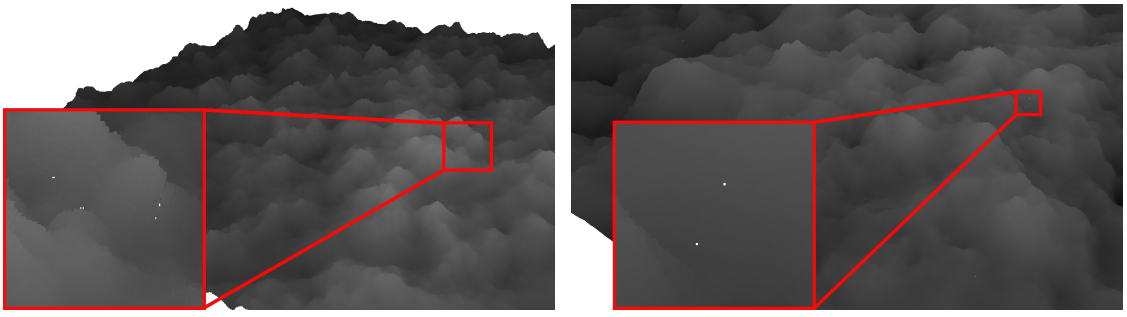
As performance remains the major problem with this approach, a few optimizations would be necessary for its use in real-time. At the moment, for the lattice traversal, the global minimum and global maximum is used as a height input. With various pre-computations for a certain area inside the lattice-space tighter bounds could be found at the cost of memory and flexibility. One would have to know which part of the lattice space are being mapped onto the triangles which would reduce or prevent changing the offset of the texture at runtime. It would still be possible to select the desired offset without these pre-calculations and only set it to a fixed value for maximum performance at runtime.

7.4 Local Intersection Tests

At the moment all intersections are performed against AABBs. When looking close enough or when the range value is small the bounding boxes can still be visible. As done in Section 4.2.5, local intersection tests performed on the leafs of the triforce traversal should improve quality and would make calculation of surface normals for the displaced surface possible. Because the triforce traversal uses triangles as traversal nodes and Tessellation-Free Displacement Mapping uses squares, the local intersection tests have to be adjusted to fit this approach.

7.5 Possible Errors

For higher values of the range parameter, small holes appear in the surface as seen in Figure 7.1. These holes also appear in my implementation of Tessellation-Free Displacement Mapping, so it could just be an inaccuracy produced by my implementation. More research on these holes would be needed to find out if they are created by a flaw in the bounding box calculation or if they only appear because of my implementation.



(a) Procedural displacement (range = 256) (b) Tessellation-Free Displacement Mapping

Figure 7.1: Small holes appear at higher values for the range parameter. As the same thing happens to Tessellation-Free Displacement Mapping at higher qualities, this could be an implementation error.

8. Conclusion

Performing intersection tests with a surface that is displaced by a procedural texture works without pre-tessellation with this approach. It has a low memory usage and provides solid results. Through traversing the lattice-space, in which the implicit procedural texture exists, the intersection of a ray with the displaced surface is achieved by introducing a 2-level-traversal. The intersection point is narrowed down through these traversals while discarding large possible areas because of the inherent hierarchical structure. As there are only a few techniques to achieve texture based procedural displacement for ray tracing (like pre-tessellation or raymarching) this method introduces a new approach to the toolbox. Like every approach, it has benefits and disadvantages, and can be weighted against the other methods based on the application. Although interactive rates are not yet achievable with this approach, it could make a good addition for offline rendering techniques and could help artists to cover larger reasons with a procedural displacement that follows an example texture. With the semi-interactive framerates and the help of built-in LoDs, editing in real-time is still possible. Overall it can serve as an alternative to pre-tessellation for displacement textures that are specifically created by the method outlined in Section 4.1.

Bibliography

[Bit16] B. Bitterli, "Histogram Tiling," 2016, Accessed: 13.07.2022. [Online]. Available: https://benedikt-bitterli.me/histogram-tiling/

[Bur19] B. Burley, "On Histogram-Preserving Blending for Randomized Texture Tiling," *Journal of Computer Graphics Techniques (JCGT)*, vol. 8, no. 4, pp. 31–53, November 2019, Accessed: 19.04.2022. [Online]. Available: http://jcgt.org/published/0008/04/02/

[Dac18] C. Dachsbacher, "Computergrafik, Vorlesung im Wintersemester 2018/19, Kapitel 2: Raytracing," p. 12, 2018, Accessed: 17.02.2019.

[Don05] W. Donnelly, "Per-pixel displacement mapping with distance functions," *GPU gems*, vol. 2, no. 22, p. 3, 2005, Accessed: 17.09.2022. [Online]. Available: https://dimap.ufrn.br/~motta/dim102/Projetos/perPixelDisplacementMapping.pdf

[GJK88] E. Gilbert, D. Johnson, and S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *IEEE Journal on Robotics and Automation*, vol. 4, no. 2, pp. 193–203, 1988, Accessed: 18.09.2022. [Online]. Available: https://ieeexplore.ieee.org/document/2083

[HB95] D. J. Heeger and J. R. Bergen, "Pyramid-Based Texture Analysis/Synthesis," in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 229–238, Accessed: 18.09.2022. [Online]. Available: https://doi.org/10.1145/218380.218446

[HN18] E. Heitz and F. Neyret, "High-Performance By-Example Noise Using a Histogram-Preserving Blending Operator," *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 1, no. 2, aug 2018, Accessed: 05.01.2022. [Online]. Available: https://doi.org/10.1145/3233304

[HN19] ——, "Procedural Stochastic Textures by Tiling and Blending," in *GPU Zen 2: Advanced Rendering Techniques*, ser. GPU Zen Series, W. Engel, Ed. Independently Published, 2019, ch. IV.2, pp. 177–200, Accessed: 05.01.2022. [Online]. Available: https://eheitzresearch.wordpress.com/738-2/

[HS+98] W. Heidrich, H. Seidel *et al.*, "Ray-tracing procedural displacement shaders," *Language*, vol. 20, no. 10, p. 24, 1998, Accessed: 17.09.2022. [Online]. Available: citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.9760&rep=rep1&type=pdf

[Ima] Imagination Technology, "PowerVR-Photon," Accessed: 18.09.2022. [Online]. Available: https://www.imaginationtech.com/products/gpu/graphics-architecture/powervr-photon/

[KTI+01] 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, Accessed: 18.09.2022. [Online]. Available: https://www.gamedevs.org/uploads/detailed-shape-representation-with-parallax-mapping.pdf

[L. 08] L. K., Nicola, "Displacement Mapping.jpg," 2008, Accessed: 18.09.2022. [Online]. Available: https://commons.wikimedia.org/wiki/File:Displacement_Mapping.jpg

[LGBA] M.-K. Lefrancois, P. Gautron, N. Bickford, and D. Akeley, "NVIDIA Vulkan Ray Tracing Tutorial," Accessed: 18.09.2022. [Online]. Available: https://github.com/nvpro-samples/vk_raytracing_tutorial_KHR

[NVI] NVIDIA, "nvpro core," Accessed: 18.09.2022. [Online]. Available: https://github.com/nvpro-samples/nvpro_core

[Per02] K. Perlin, "Improving Noise," in *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 681–682, Accessed: 04.05.2022. [Online]. Available: https://doi.org/10.1145/566570.566636

[SdF03] J. Stolfi and L. de Figueiredo, "An Introduction to Affine Arithmetic," *Trends in Computational and Applied Mathematics*, vol. 4, no. 3, pp. 297–312, 2003, Accessed: 13.08.2022. [Online]. Available: https://tema.sbmac.org.br/tema/article/view/352

[TBS+21] T. Thonat, F. Beaune, X. Sun, N. Carr, and T. Boubekeur, "Tessellation-Free Displacement Mapping for Ray Tracing," vol. 40, no. 6, dec 2021, Accessed: 05.01.2022. [Online]. Available: https://doi.org/10.1145/3478513.3480535