Adaptive Sampling and Tessellation for Displacement Mapping Hardware

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Johannes Hirche
aus Tübingen

Tübingen
2003

Tag der mündlichen Qualifikation:	17.12.2003
Dekan:	Prof. Dr. Ulrich Güntzer
1. Berichterstatter:	Prof. Dr.-Ing. Dr.-Ing. E.h. Wolfgang Straßer
2. Berichterstatter:	Prof. Dr. Wolfgang Rosenstiel

Abstract

In this thesis, new algorithms and architectures are presented for the acquisition and rendering of displacement maps. Displacement mapping is a popular rendering technique commonly found in commercial rendering packages, it modifies the surface of an otherwise flat triangle by displacing its points according to a height field, giving the impression of a structured surface. Although widely used in software renderers for many years, hardware accelerated rendering was long missing, because of the difficulties involved. The main focus of this work is on adaptively rendering techniques, especially adaptively tessellating triangular meshes to improve rendering performance and reduce bandwidth requirements.

Following a general definition of displacement maps, possible sources of displacement maps are presented and examples are given. To allow adaptive tessellation, sophisticated sampling techniques are required, and multiple sampling strategies are described and compared with respect to quality and ease of implementation. The presented strategies range from locally defined geometry tests to image-based methods like edge-detection.

Possible modifications to available graphics hardware pipelines are discussed to enable support for adaptive tessellation with as little modifications as possible.

With the latest generation of commodity graphics chips, it has become possible to approximate displacement mapping by a variation of ray-casting. Sample implementations are given and compared to other approaches, implemented on the same hardware platform.

Zusammenfassung

In dieser Arbeit werden neue Verfahren und Architekturen zur Erzeugung und Darstellung von *Displacement-Maps* vorgestellt. Das *Displacement-Map* Verfahren ist eine populäre Technik zur Darstellung von Oberflächenstrukturen die in kommerziellen Programmpaketen zur Bilderzeugung Verwendung findet. Bei diesem Verfahren wird die Erscheinung eines sonst flachen Dreiecks durch Verschieben von Punkten der Oberfläche wie in einem Höhenfeld vorgegeben modifizert, um den Eindruck einer fein strukturierten Oberfläche zu erwecken. Obwohl es in Programmpaketen zur Bilderzeugung bereits breite Verwendung gefunden hat, gab es lange Zeit keinerlei Beschleunigung durch dedizierte Hardwareentwicklungen, hauptsächlich bedingt durch die Komplexität der notwendigen Berechnungen. Das Hauptaugenmerk dieser Arbeit ruht auf adaptiven Verfahren, besonders der adaptiven Triangulierung von Dreiecksnetzen, um die Darstellungsgeschwindigkeit zu erhöhen und die notwendige Bandbreite zu reduzieren.

Nach einer genaueren Definition von *Displacement-Maps* werden Verfahren zu ihrer Generierung aus unterschiedlichen Ausgangsdaten vorgestellt. Für die adaptiven Verfahren sind ausgeklügelte Abtastverfahren nötig. Verschiedene Verfahren werden vorgestellt und in Hinsicht auf Qualität und Komplexität einer Implementierung verglichen. Die vorgestellten Verfahren reichen von lokalen Geometrietests, hin zu bildbasierten Verfahren, wie beispielsweise Kantendetektion.

Erweiterungen für aktuell erhältliche Graphikprozessoren auf Basis von NVidia NV3X oder ATI Radeon R3X0 zur adaptiven Triangulierung von Dreiecksnetzen werden vorgestellt, die möglichst geringfügige Änderungen am bestehenden Aufbau benötigen.

Diese neueste Generation von kommerziell erhältlichen Graphikprozessoren erlaubt es, eine Approximation von *Displacement-Mapping* mittels Strahlverfolgungsverfahren durchzuführen. Es werden Beispielimplementierungen vorgestellt und mit anderen Ansätzen, die auf der gleichen Zielarchitektur arbeiten, verglichen.

Contents

List of Figures

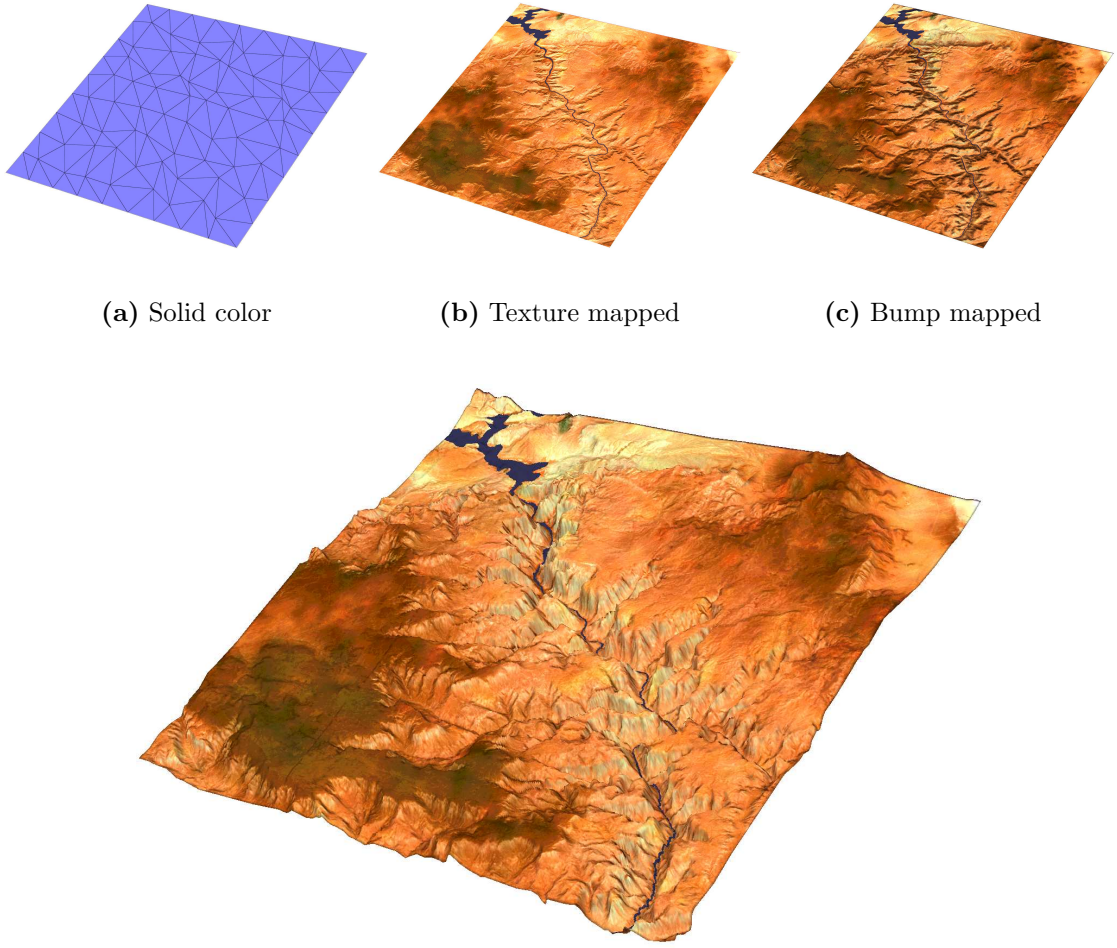
Chapter 1

Introduction

1.1 Introduction to Displacement Maps

Ever since the early 1970s, when raster graphics was introduced with the Alto system of Xerox PARC, where its rise began, it has not stopped to evolve and dominates our technical life, starting with digital wrist watches ranging up to high end visualization systems. With the ever increasing computational power as stated in Moore's law, raster graphics was soon used to display 3D graphics, first off-line, later in real-time. 3D graphics has come a long way from crude line drawings to highly detailed and realistically colored renderings. The driving force has always been – and still is – the quest for more visual realism. The prominent drawing primitive in todays 3D graphics architectures is still a triangle, as effective and fast rasterization routines exist. Rasterization is the process of converting an analytical geometrical object like a triangle, square or even a line to a representation that can be shown on a raster graphics display. For raster graphics the data element is a *pixel*, an abbreviation for picture element, that is stored in dedicated memory, the framebuffer. Thus, rasterization is the process of mapping a geometric primitive to the discrete 2D grid of the screen.

Although triangles may be very easy to rasterize and manage, they are hardly an adequate primitive for modeling our natural, highly irregular environment. In order to make objects defined with triangles look more realistic a great number of techniques has been developed, enhancing the rasterization process to add more surface detail and make the result look more realistic. Already in 1976, Blinn and Newell [18] introduced a technique for adding color to every rasterized pixel of a surface according to a 2D bitmap image, called *texture map*. Additionally a technique for simulating reflective surfaces was described. Later in 1978, Blinn added another technique called bump mapping [17], where the surface of a triangle gets a structured appearance using lighting effects. All these tech-



(a) Solid color **(b)** Texture mapped **(c)** Bump mapped

(d) Displacement mapped

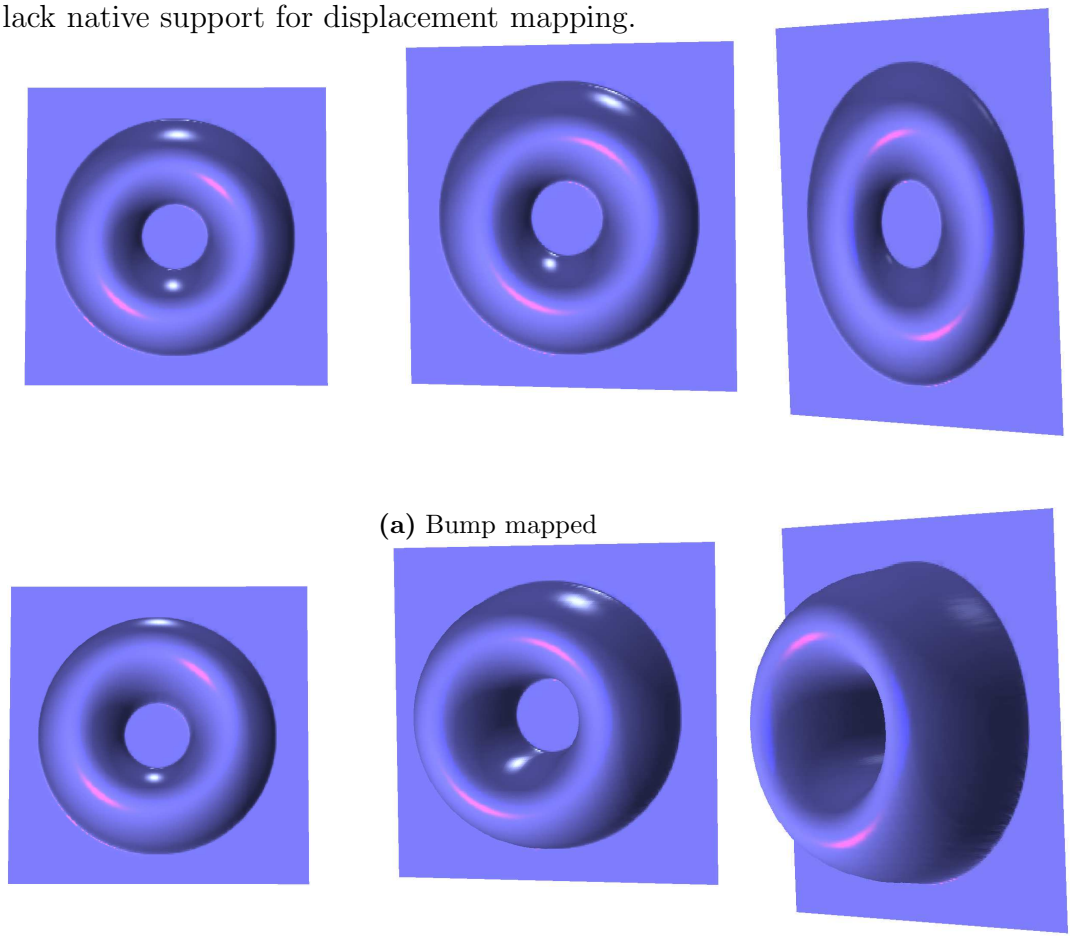
Figure 1.1: The evolution of rendering algorithms for surface rendering used on a terrain model of the Grand Canyon[1].

niques have been widely adopted and are implemented in almost any graphics hardware available today. Because of its widespread use, texture mapping has been analyzed thoroughly [28] and many improvements have been made in the area of filtering the textures [47, 41].

A common feature of all these algorithms is that the given geometry is not changed but only the visible appearance is modified, giving the illusion of a structured surface. This causes problems in cases where the desired visible effects are not only local to a pixel but also have a global influence, as for example the silhouette and self-occlusion. Displace-

[1]Data is obtained from The United States Geological Survey (USGS), with processing by Chad McCabe of the Microsoft Geography Product Unit.

ment mapping as introduced by Cook [20] in 1984 on the other hand actually modifies the geometry thus avoiding all the problems with bump mapping. Displacement mapping was adopted by commercial rendering software packages like Maya [44] or Softimage [30], and is widely used in that area. Hardware support for accelerated rendering of displacement maps was not available until recently the Matrox Parhelia [38] was announced, although other rendering techniques as texture mapping or bump mapping are available even in commodity graphics hardware for quite some time now [32]. Most hardware vendors however lack native support for displacement mapping.



(a) Bump mapped

(b) Displacement mapped

Figure 1.2: Rendering of a half donut shaped height field, in (a) bump mapped and in (b) displacement mapped.

Figure 1.2 demonstrates the difference between bump and displacement mapping with a half-donut shaped height field applied to a quadliteral. In 1.2(a) the polygon is bump mapped and in 1.2(b) the same polygon is rendered with the height field is applied as a displacement map instead. From a viewpoint perpendicular to the polygon surface,

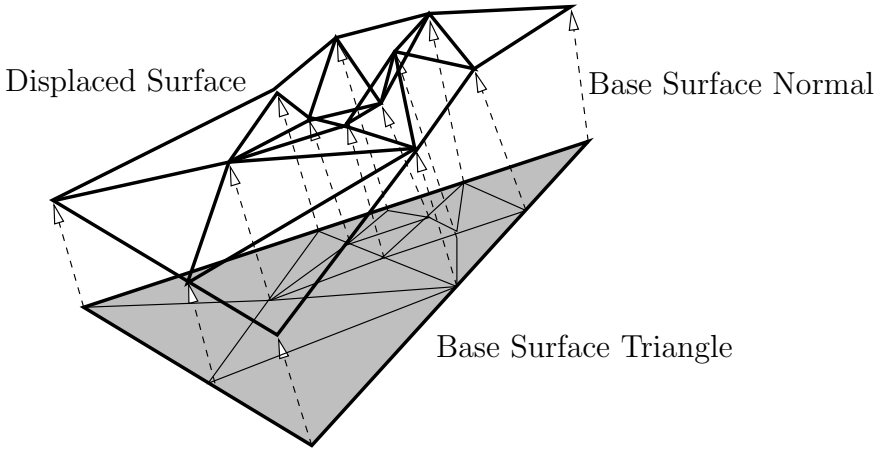


Figure 1.3: Tessellated triangle with displacement performed along the surface normals.

it both algorithms produce similar results and the bump mapped polygon gives a good visual impression of an elevated surface. If the polygon is tilted to the side however, the bump mapped version quickly looses its realistic appearance, and if tilted more, the missing silhouette becomes more obvious.

Displacement maps allow for simple adding of more detail to any surface that can be parameterized. Surfaces are usually defined as triangle meshes with texture coordinates defined at the triangle vertices. With these texture coordinates, it is possible to address a displacement map to modify the polygon surface. If the surface is given in another representation it is generally tessellated with triangles before rendering. A displacement map can be defined as a height field that is applied by elevating a point on the surface by the height read from the height field at the point's texture coordinates. In Figure 1.3 an example of a tessellated triangle that is being displaced by its surface normals resulting in a structured surface. This thesis' focus is on efficient rendering of such displacement maps, mainly targeted at graphics hardware architectures.

Chapter 2

Displacement Map Theory

2.1 Types of Displacement Maps

In general, a displacement map can be any function f that for a set $M \subset \mathbb{R}^n$ with a parameterization $p : U \subset \mathbb{R}^m \mapsto M$ defined as

$$f : U \times M \mapsto \mathbb{R}^n \tag{2.1}$$

or

$$f(x, p(x)) \mapsto \mathbb{R}^n, x \in U. \tag{2.2}$$

The function describes a displacement of all points of M. For a given point x of the set M, the corresponding displaced point x' can be expressed as

$$x'_x := f(p^{-1}(x), x). \tag{2.3}$$

For practical uses, only displacement maps defined on a 2D parameterization of a 2-manifold are considered and are used throughout the following chapters. The manifold that is to be displaced is called the *base domain surface* or just *base surface*. In case of a 2D parameterization $p(u, v)$ the surface normal $\vec{n}_x$ for a point $x = p(u, v)$ on the surface M is defined by:

$$\begin{aligned} \vec{n}_x &:= \frac{\vec{s}_x}{|\vec{s}_x|} \\ \text{with} & \\ \vec{s}_x &:= \frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v} \end{aligned} \tag{2.4}$$

where $\frac{\partial p}{\partial u}$ and $\frac{\partial p}{\partial v}$ are the partial derivatives of the surface p along the surface directions u and v.

For the displaced surface, the new surface normal $\vec{n}_{x'}$ is defined as

$$\begin{aligned} \vec{n}_{x'} &:= \frac{\vec{s}_{x'}}{|\vec{s}_{x'}|}. \\ \text{with} & \\ \vec{s}_{x'} &:= \frac{\partial f(u,v,p(u,v))}{\partial u} \times \frac{\partial f(u,v,p(u,v))}{\partial v} \end{aligned} \tag{2.5}$$

The new surface normal can be costly to calculate, in particular when the employed map is discrete. As a result of this the surface normal is often precomputed from the height field.

2.1.1 Vector Displacement Maps

This is a generalized type of a displacement map where the displacement is given by a vector $\vec{d}_x$ along which the surface points x are moved.

$$\forall x \in M : x' := x + \vec{d}_x. \tag{2.6}$$

The displacement vector can either be defined as a replacement of the surface normal, or relative to it, for example as a bump map.

Vector displacement maps are particularly difficult to handle and render as little or no assumptions can be made about the resulting geometry. As a result they are used very rarely. If no special care is taken about the base domain that is being displaced with a specific vector displacement map the topology of the resulting surface can easily change substantially from the original base domain surface. Useful combinations can be obtained when performing mesh compression by generating a low resolution base domain mesh and a corresponding displacement map containing the detail as the resulting surface is already known, and well behaved.

2.1.2 Scalar Fields

The simplest type of a displacement map is a scalar field or height field. In this case only a scalar displacement value is used and the displacement is performed as an elevation of the surface with the relative height given by the scalar value.

The points x on the base domain surface M are displaced along their respective surface normals $\vec{n}_x$ with the scalar factor $d_x \in D$, with D the applied displacement map.

$$\forall x \in M : x' := x + d_x \cdot \vec{n}_x. \tag{2.7}$$

In Figure 2.1 a very simple example with a one-dimensional height field is shown.

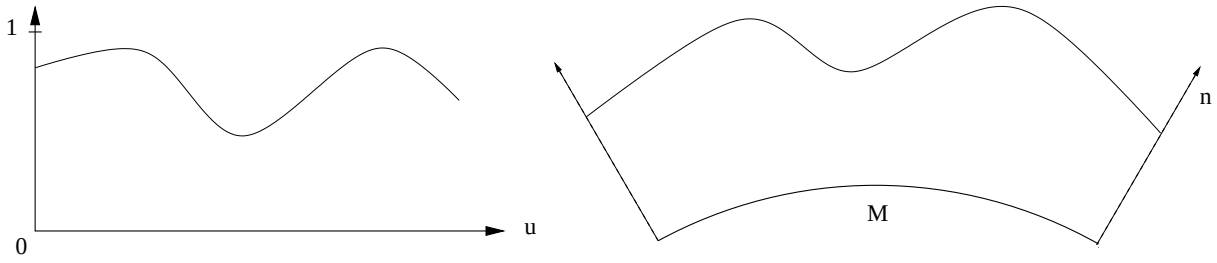


Figure 2.1: One-dimensional height field applied to a curved line. The one-dimensional displacement function on the left is applied to the base domain M in direction of the gradient n.

The scalar field can be thought of as a special case of a vector displacement map with only the height stored in the map and the direction of displacement defined by the surface normal of the base domain surface. As the base domain surface's shape is likely to change considerably when it is displaced, the surface normals have to be recalculated. The new surface normal at a specific position cannot be directly determined by the displacement value at its position, the neighborhood has to be taken into account. Analytical calculation of the new surface normal as described in Equation 2.5 is only suitable for precomputation which is not always desirable or possible. If precomputation is used, the surface normals are generally not stored absolutely, but rather relative to a surface normal. Storing the normal relative to a normal vector as a perturbation allows to apply the precomputated normal to any base domain surface geometry. This technique is commonly called also bump mapping [33]. If the shape of the base domain surface contains non-convex regions,

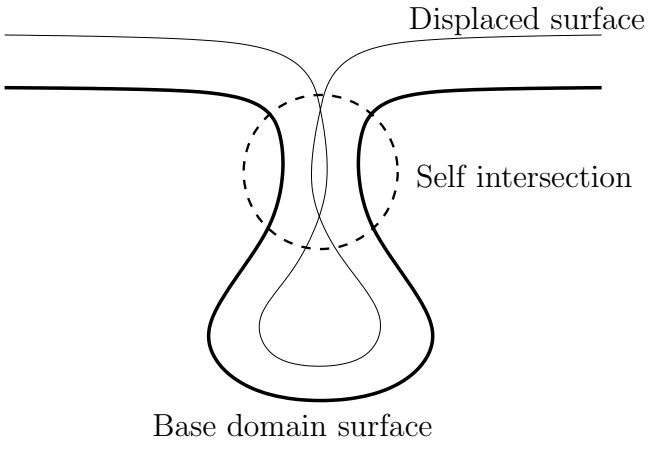


Figure 2.2: Non-convex base domain surface resulting in self-intersections when displaced.

problems may occur as the displaced surface may contain unexpected changes in topology like self-intersections as it is shown in Figure 2.2, where a height field with constant height

was used.

2.2 Data Sources for Displacement Maps

In general any kind of data that represents or approximates a surface can be used for constructing a displacement map. The most difficult task is to find a suitable base domain surface for applying the displacement map to, especially when the desired type of displacement map is a height field.

Once a suitable base domain surface is known the displacement map can be obtained by casting rays along the surface normal from the base domain surface and determining the intersection with the source surface. If the source surface is not continuous, like in a point cloud, a local surface approximation has to be calculated first, using moving least squares or other methods [16]. In order to avoid sampling artifacts the number of rays cast has to be sufficiently high, especially when the sampling distance for the new displacement map is constant, which is most likely the case. The quality of the resulting displacement map is strongly dependent on the base domain surface used as can be seen in Figure 2.3. In this case the base domain surface does not adapt good enough to the shape of the source surface, resulting in undersampling in the specified area.

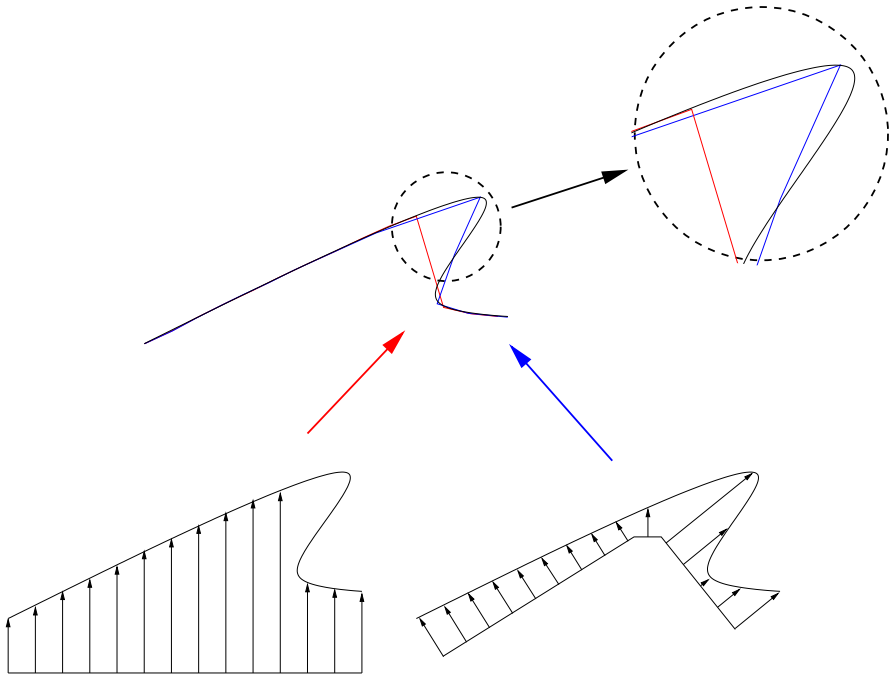


Figure 2.3: Badly adapted base domain surface (left) and improved base domain surface (right).

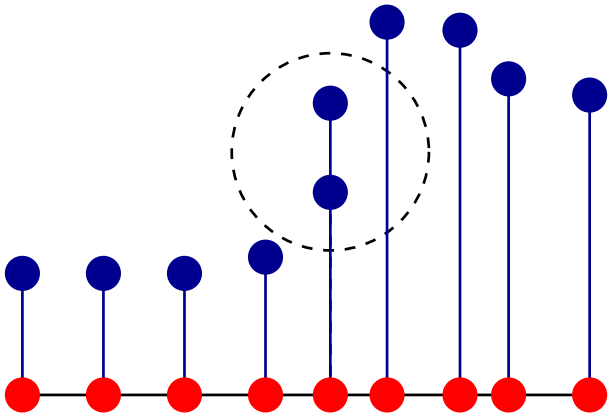


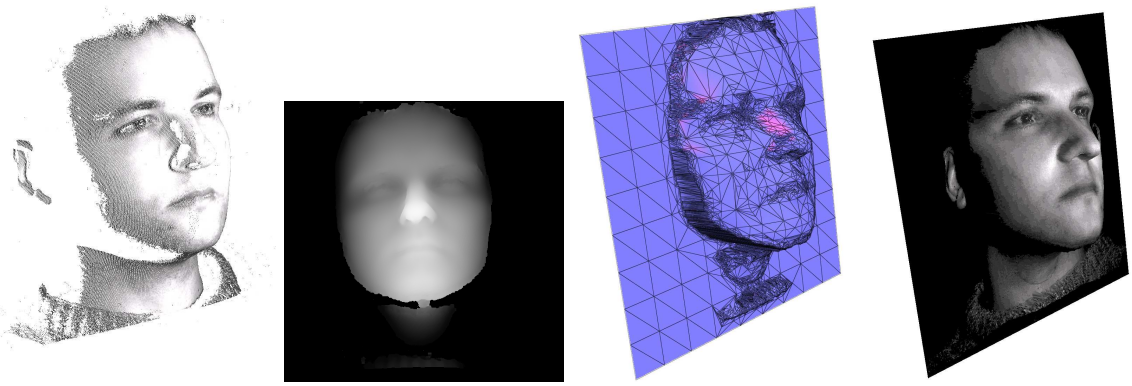
Figure 2.4: Problem with badly adapted base domain surface and the resulting projection of two source points to the same location.

In some cases, especially when using point clouds as source data, it may be easier to work in the reverse direction and to project the source points onto the prospective base domain surface. The problem here is finding the corresponding base domain surface point with the fitting normal, as the displacement can only be performed along this direction. As an approximation the point on the surface with the minimum distance to the source point may be used, if the used base domain surface is not well adapted to the source data problems may arise with multiple source points being projected to the same base domain surface position as shown in Figure 2.4.

2.2.1 Point Clouds and Range Scanner Data

As described before the problem with scattered data is the lack of a surface for the intersection calculation. A number of techniques for approximating a surface from point clouds exist, given that the density of points is sufficient for a robust surface approximation [29]. If at all possible it should be avoided to triangulate the point cloud before sampling. It is favorable to directly project the points to a suitable base domain surface.

When using range scanner data the geometric constellation of the machinery used to obtain the data is known and can be used to simplify the construction process. Commonly used scanners as the Cyberware CyberScanner [22] often have a cylindrical geometry. The distance is measured using a laser scanner rotating around the object to be sampled. In this case the resulting base domain surface is a cylinder and the distance can be directly assigned to a displacement map as the laser scanner is performing the sampling in the same way as described before for the point clouds. In Figure 2.5 the process of generating a height field from a point cloud is shown. The points were projected onto a plane passing



(a) Point cloud **(b)** Resulting filtered height field **(c)** Wireframe rendering **(d)** Additional texture applied

Figure 2.5: Generation of a height field from a point cloud.

through the center of the point cloud and the distance to the plane was stored in 2D gray scale image. The raw height field was enhanced using standard image processing algorithms to smooth out noise and fill holes.

2.2.2 Mesh Data

To obtain a displacement map for a highly refined mesh, for example a triangle mesh, a standard mesh reduction algorithm is used, and after the reduced triangle mesh is sufficiently small, rays are cast from this surface to the source mesh. If the targeted rendering hardware supports displacement mapping, this provides a simple form of geometry compression. Lee [35] used a subdivision surface generated by simplifying the source data as an intermediate base domain surface. The simplified subdivision surface is first subdivided two to three times using Loop's [36] subdivision scheme and afterwards rays are cast from the now smooth subdivided surface to the source surface, capturing the fine detail. This can also be used as a simple form of mesh compression, as a small base domain surface mesh and a corresponding displacement map, which can be very easily compressed using well-investigated and highly effective image compression algorithms, typically consume far less memory than the fully tessellated mesh.

As an example the mesh of a molded plate was converted to a displacement map. The mesh in Figure 2.6 was sampled with 1024×1024 rays from a flat base domain surface. The obtained height field in Figure 2.7 was then applied to a base domain surface with the same dimensions as the source mesh projected to a plane, but with only two triangles. The

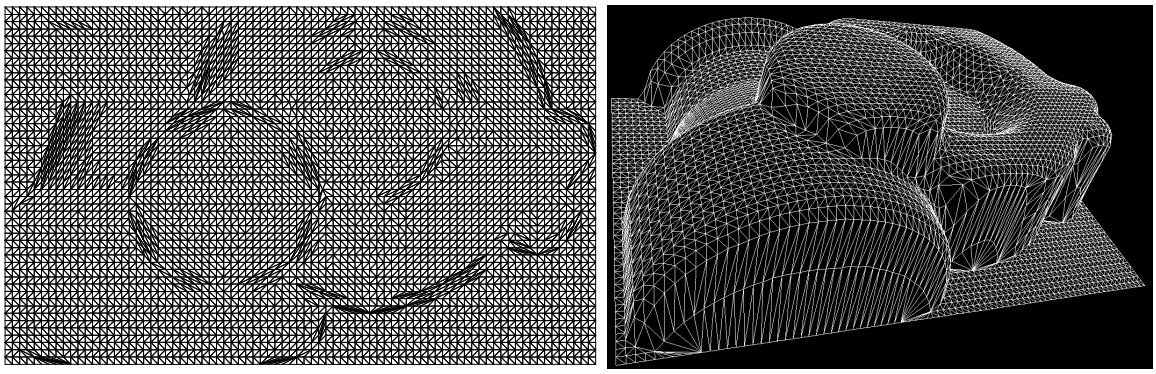


Figure 2.6: Source mesh for acquiring a displacement map. The input mesh has about 8000 triangles.

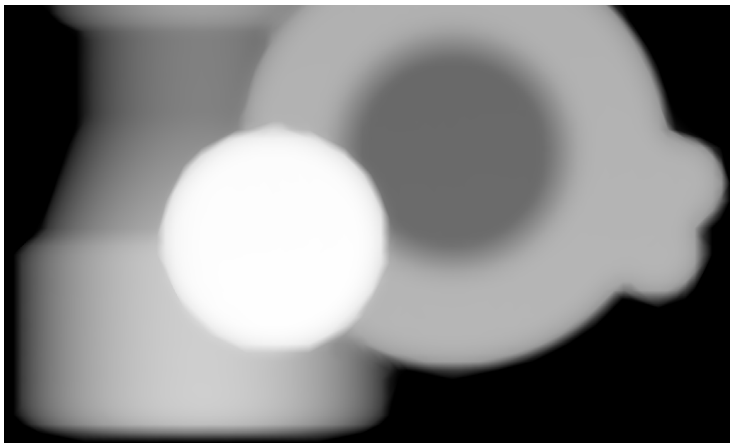


Figure 2.7: Height field acquired from the input mesh in Figure 2.6. The mesh was sampled by casting 1024×1024 rays from a flat base surface.

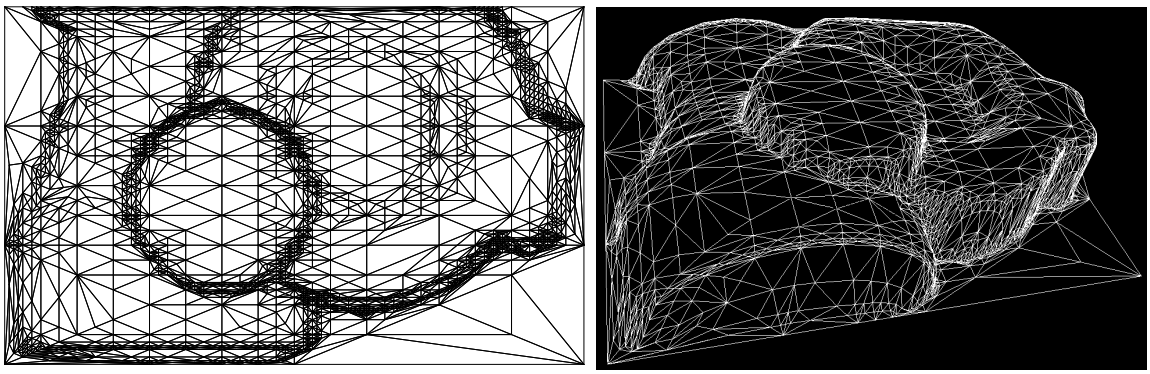


Figure 2.8: Result of applying the height field obtained in Figure 2.7. For generating the mesh adaptive tessellation was used, described in chapter 4. The mesh shown has only 4500 triangles.

result of the displacement process can be seen in Figure 2.8. Using adaptive tessellation the triangle count could be reduced by almost 50% compared to the input mesh.

2.2.3 Continuous Forms of Data

Similarly as the point clouds and triangle meshes, any data source that forms some sort of a surface can be used as input. In particular mathematical objects like free form surfaces as non-uniform rational B-splines (NURBS), which are difficult to render otherwise, can be rendered through the use of a displacement map. Direct hardware accelerated rendering of NURBS surfaces is a difficult task and a couple of hardware architectures have been proposed, but no implementations have emerged. Especially if the NURBS surface is not triangulated [34], and not rendered using a conventional triangle rasterizer, but a special purpose scan converter is used, a lot of difficulties arise, complicating a possible implementation. In [1] a graphics hardware architecture for adaptively tessellating a triangle mesh with a user definable rule set is presented, maintaining the connectivity information when adding triangles. This not only allows for adaptively rendering displacement maps – with the displacement map sampling tests presented later on in section 3 for controlling the adaptive tessellation – but also for the rendering of subdivision surfaces or other curved surfaces. If hardware accelerated support for displacement maps would be available, they could be used as a cheap replacement for an otherwise costly scan converter.

In essence, the same rules apply for generating a base domain surface/displacement map pair. The crucial part is again finding an appropriate base domain surface. After a base domain surface is found, rays can be cast and the NURBS surface is sampled.

2.3 Displacement Map Filtering

As a displacement map can be thought of as a texture map, the same sampling and filtering problems might arise. As it turns out, displacement maps are even more subject to sampling artifacts and the effect of undersampling can cause more severe errors, even at close distances. A standard approach for texture filtering is mipmapping as presented by Williams [47]. This provides an effective means of retrieving levels of detail in color textures that match the screen size of an object. But for displacement mapping, the averaging effect of mipmapping will smooth over areas of high detail in the displacement map, creating popping artifacts when the levels of detail are switched and distinct, sharp peaks are being smoothed out. In [27], it is proposed to use mipmapping, but with a maximum filter to overcome this.

2.4 Conclusions

Displacement maps are commonly used in the form of simple height fields or vector displacements, whereas the vast majority are simple height fields. Apart from synthetically created displacement maps, they can be obtained by sampling other surfaces, either smooth continuous NURBS surfaces or other piece-wise linear surfaces like meshes. By using an appropriate projection, point clouds are likewise usable as a data source for a displacement map.

Chapter 3

Sampling

Correct sampling of a displacement map plays a vital role for the quality of almost any rendering algorithm. It is important to detect where to sample and especially how accurately a region needs to be sampled. Depending on the type of algorithm and the target architecture – software or hardware – used for rendering, the amount of information available about the map can be very different. As the rendering algorithms usually use texture coordinates and texture mapping hardware is found in commodity graphics cards, the possibilities for filtering are important. It has to be analyzed whether the filtering methods used in texture mapping algorithms can be transferred to displacement mapping algorithms.

3.1 Sampling Tests

In this Section different test schemes for sampling are discussed and evaluated according to their possible applications and limitations.

3.1.1 Surface Normal Variance Test

This new test [2] operates solely on the surface normal of the displaced surface. The surface normals of two or more sample points are compared for example by calculating the absolute distance of the normal vector components. If the result is above a given threshold, more sampling is needed between the points, or if one of the sample points was a candidate for insertion, the point needs to be added.

As an example consider the situation as shown in Figure 3.1. Here a triangle mesh is used as a rendering primitive and detail is added by adaptively inserting triangles. Between the two endpoints V_1 and V_2 of an edge a candidate for insertion $V_{1,2}$ was

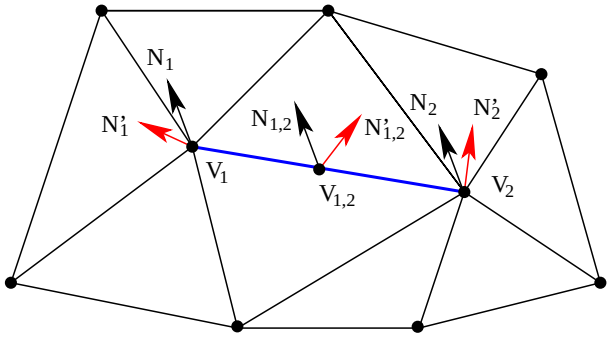


Figure 3.1: Example candidate for insertion in a base domain surface triangle mesh.

placed in the middle of the edge. To perform the actual test, the resulting displaced surface normals N'$_1$,N'$_2$ and N'$_{1,2}$ of the corresponding surface normals N$_1$,N$_2$ and N$_{1,2}$ have to be compared.

The actual comparison of the normals can be done with any kind of norm that is suited to detect change in direction of the normal vector. In this case it is sufficient to calculate the difference between the vector components and to compare all of them to a threshold. If the difference between any of the components is greater than the given threshold *nthr*, the vertex is added at $V_{1,2}$.

A boolean value for the Normal Test, *nt* can be defined as

$$\begin{aligned} nt \quad = \quad & (\mathrm{N'}_{1,2|x} - \mathrm{N'}_{1|x} < nthr) \text{ OR } (\mathrm{N'}_{1,2|y} - \mathrm{N'}_{1|y} < nthr) \text{ OR} \\ & (\mathrm{N'}_{1,2|z} - \mathrm{N'}_{1|z} < nthr) \text{ OR } (\mathrm{N'}_{1,2|x} - \mathrm{N'}_{2|x} < nthr) \text{ OR} \\ & (\mathrm{N'}_{1,2|y} - \mathrm{N'}_{2|y} < nthr) \text{ OR } (\mathrm{N'}_{1,2|z} - \mathrm{N'}_{2|z} < nthr) \end{aligned}$$

This test is well suited to detect change in normal direction which often occurs in regions where the displacement map contains higher frequencies. It can also be seen as a high pass filter for the displacement map. As such it will miss low frequency changes in the map like larger changes in height. Additionally as the test uses point sampling it is subject to aliasing. The threshold *nthr* has to be specified by the user and is strongly dependent on the structure of the displacement map. If the test is combined with a second test, which will detect low frequency changes in the map, better results can be achieved. The test is relatively cheap to implement, because only the new surface normals have to be calculated – which has to be done anyway if the sample point is added – and no additional preprocessing is necessary.

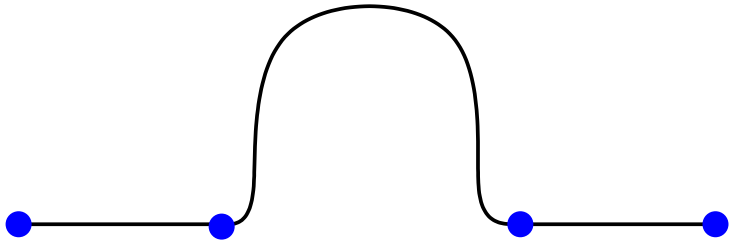


Figure 3.2: Simple point sampling missing a strong change in height although being very close to it.

3.1.2 Local Area Average Height Test

In contrast to the normal test in Section 3.1.1 this new test [2] works well for detecting low frequency changes in the displacement map or average changes in height. The basic idea is to compare the height or displacement of two or more sample points. The simplest way to do this is to compare the height at the desired sample points and to check if the difference is above a given threshold. As this is a form of point sampling it is subject to aliasing and would not yield satisfying results. As shown in Figure 3.2 it will miss even strong changes in height in close proximity to the sample points, which is undesirable.

To avoid problems with point sampling filtering techniques may be used. Good results can been achieved when using a Summed-Area Table, introduced by Crow [21]. A Summed-Area Table is a two dimensional array containing at each cell the sum of all values that fall inside the rectangle formed by that cell and one corner of the array. To calculate the sum of all values within a rectangular area in the table only the four values at the corners of the area are needed. The Summed-Area Table can be represented as a bivariate function $SAT(x, y)$ that returns the sum of all heights within the region $(0 \rightarrow x, 0 \rightarrow y)$, where the origin is in the bottom left of the table. The sum of the values within a rectangular area is calculated using the function $S(\mathbf{Z})$ defined as

$$\begin{aligned} S(\mathbf{Z}) &= SAT(x_{tr}, y_{tr}) - SAT(x_{tl}, y_{tl}) - \\ &\quad SAT(x_{br}, y_{br}) + SAT(x_{bl}, y_{bl}) \end{aligned} \tag{3.1}$$

where $\mathbf{Z}$ is $(x_{tr}, y_{tr}, x_{tl}, y_{tl}, x_{br}, y_{br}, x_{bl}, y_{bl})$, the corners of the rectangular area in the Summed-Area Table, using the subscripts tr top right, tl top left, br bottom right, bl bottom left for the four corners of the rectangle .

A Summed-Area Table can be precalculated for a displacement map the same way as for a normal texture map. As an example for using the test consider again the example in Figure 3.1. The vertex $V_{1,2}$ is to be checked against V_1 and V_2. A rectangle in the Summed-Area Table corresponds to the sum of all heights of the same rectangle in the

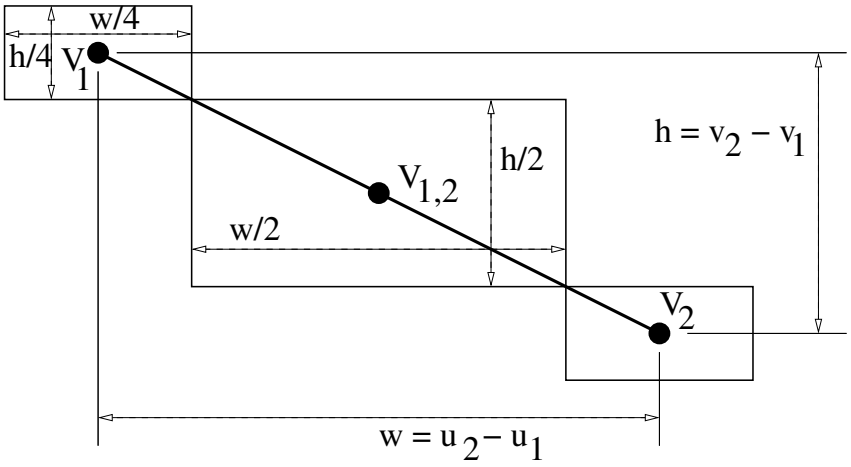


Figure 3.3: Using the texture coordinates of the vertices to calculate the summed height.

displacement map, thus we can compare the average height around the three sample points if we normalize the values read from the Summed-Area Table with the rectangle size. The size of the rectangle has to be chosen carefully as a too large area will smooth out large differences between the sample points, due to the averaging nature of the Summed-Area Table.

As an example in Figure 3.3 the area surrounding the midpoint $V_{1,2}$ was enlarged to detect changes in a larger area between the vertices. To calculate the four corner points of the rectangle around the vertices of the edge the texture coordinates at the vertices U_1, U_2 and $U_{1,2}$ are used as shown in Figure 3.3. Using the texture coordinates the differences of the areas can be calculated and compared with a threshold.

A boolean value for the height test, *sht* can be defined as

$$sht = \left(\frac{S(V_{1,2})}{2} - (S(V_1) + S(V_2)) \right) < shthr$$

where *shthr* is the summed height threshold. Note the normalization of the values caused by the difference in size of the compared areas.

The Summed Height Test combines very well with the Normal Test of Section 3.1.1 as it is able to detect the low frequency change in a displacement map. In Figure 3.4 two examples for a possible cross Section of a displacement map are shown where one of the tests fails and the other succeeds.

3.1.3 View Dependency Test

The new view dependency test [2] differs from the proceeding ones because it has a negative response, it is used to avoid resampling where no more information can be added

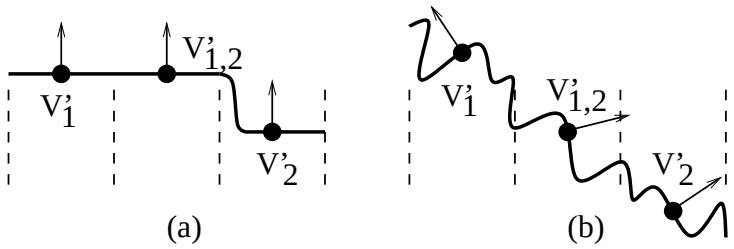


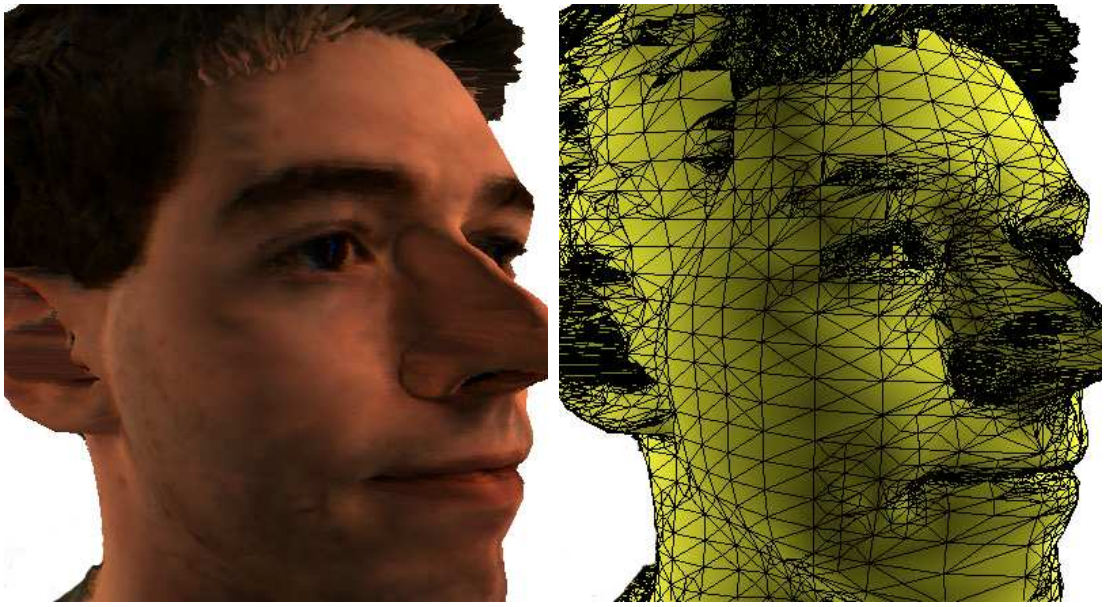
Figure 3.4: The solid line represents a contour line across the displacement map between the texture coordinates of the vertices of one edge of a triangle with the newly inserted point in the middle. The dashed lines indicate the area over which the height is averaged to calculate the Summed Height value. In (a) the Normal Test fails, but the Summed Height Test succeeds. In (b) the Normal Test succeeds but the Summed Height Test fails.

to the outcome. his specific test can be used for controlling re-triangulation by limiting the insertion of new vertices. The assumption is that the region between two points differing less than one pixel in position when transformed to screen space doesn't need to be resampled any more. Thus when a transformation unit is available, two points around the region of interest – in case of a triangle edge, the starting point V_1 and end point V_2 of the connecting edge – are transformed to screen space using the viewing transformation. If the Manhattan distance between the two sampling points is below one, no more sampling is necessary because adding any more sampling points between the two points would only add a sub-pixel sized triangle. In Figure 3.5 the displacement map of a human head was applied to a cylinder at different viewing distances. When rendered at a larger distance, the detail added is limited by the minimum triangle size. As this test works in screen space it is view dependent and provides view dependency for any algorithm that uses re-triangulation or adaptive sampling.

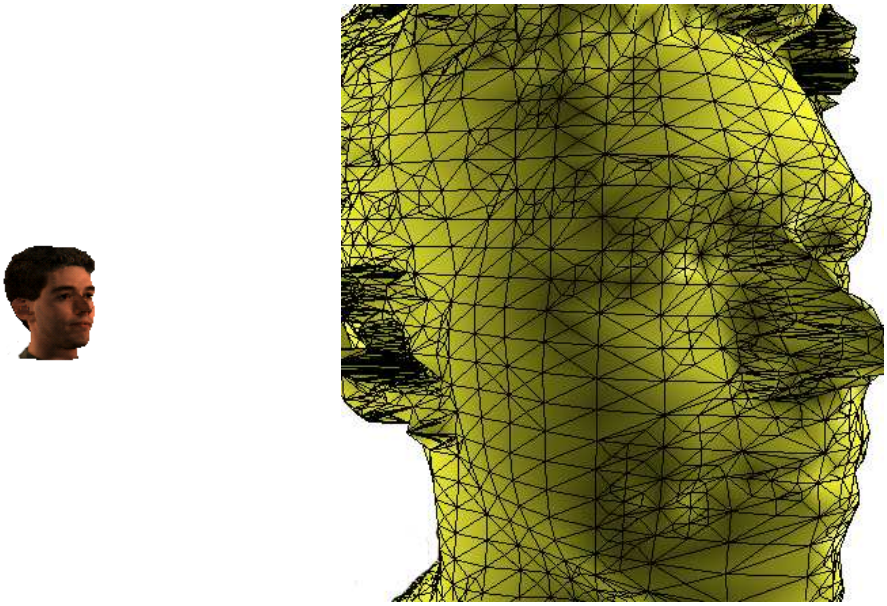
3.1.4 Refinement Limit Test

Similarly like the view dependency test in Section 3.1.3 the new refinement limit test [2] has a negative response for stopping resampling. Its use is limited to discrete types of displacement maps as it solves a problem only occurring with discrete maps. Consider the example in Figure 3.6, where the sampling distance is already below the distance between two entries in the displacement map.

Due to always occurring rounding errors when performing the interpolation of the new sample point positions and the approximation errors, several unnecessary sample points might be inserted close to the original displacement map value or between the two sample



(a) Rendering of a human head at close distance. The result is fully tessellated with fine detail in the region of the eyes and mouth.



(b) The same base domain and displacement map as in (a) but at a bigger distance. On the right side the tessellation result is enlarged, showing the reduced detail.

Figure 3.5: Result of applying the view dependency test from Section 3.1.3 to the displacement mapped rendering of a a human head.

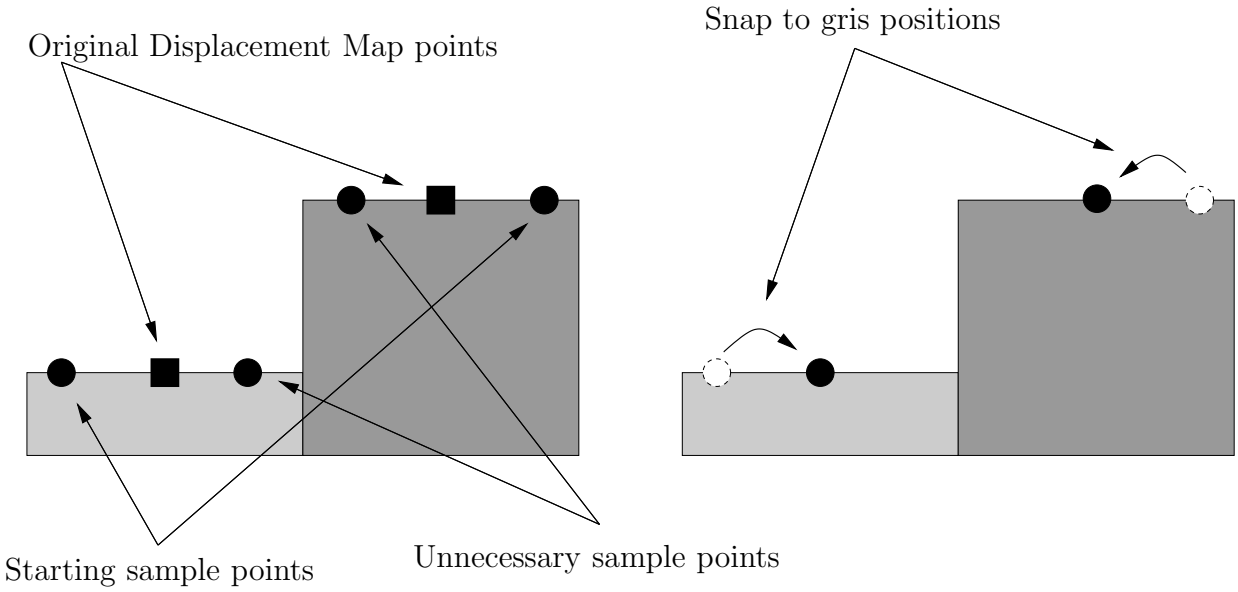


Figure 3.6: Oversampling caused by rounding and approximation errors. On the right side the starting sample points are snapped to the original displacement map positions.

points in Figure 3.6.

To avoid this unnecessary sampling a comparison of the sample positions can be used. The idea is to stop sampling when the resolution of the displacement map has been reached and to move the sample positions to the next displacement map grid position.

Given two sample positions p_1, p_2 with the respective texture coordinates t_1, t_2, with $t_i = (u_i, v_i)$, and a displacement map with the dimensions $d_w \times d_h$. The texture coordinates have to be limited to the interval $[0, 1]$ which can be easily achieved by removing the integer part of texture coordinates. The texture coordinates are then scaled with the displacement map size:

$$t'_1 = (d_w u_1, d_h v_1) = (d_w, d_h) \cdot t_1$$
$$t'_2 = (d_w u_2, d_h v_2) = (d_w, d_h) \cdot t_2.$$

By comparing the integer parts of t'_1 and t'_2 the oversampling occurring in Figure 3.6 can be easily detected. If the integer parts are equal or differ by less than one when using a Manhattan distance calculation no more sampling is necessary as the resolution of the displacement map has been reached.

3.2 Curvature Based Tests

3.2.1 Curvature Calculation

As noted before, increased sampling should occur in regions of the displacement map with strong changes in height or high curvature.

To calculate the curvature of the displacement map a number of methods can be used. Mathematically the curvature can be approximated with derivatives of the original map. Given a 2-dimensional displacement map

$$f : \mathbb{R}^2 \mapsto \mathbb{R}, \tag{3.2}$$

the gradient ∇f of the displacement function f can be expressed with partial derivatives [23]:

$$\nabla f(x_0, y_0) = \left(\frac{\partial f(x_0, y_0)}{\partial x}, \frac{\partial f(x_0, y_0)}{\partial y} \right). \tag{3.3}$$

The gradient $\nabla f(x_0, y_0)$ has a number of useful properties:

- $\nabla f(x_0, y_0)$ points in the direction of the largest slope at position (x_0, y_0).
- The gradient is orthogonal to the surface defined by the height field
- The norm of the gradient $\nabla f(x_0, y_0)$:

$$|\nabla f(x_0, y_0)| = \sqrt{\left(\frac{\partial f(x_0, y_0)}{\partial x} \right)^2 + \left(\frac{\partial f(x_0, y_0)}{\partial y} \right)^2} \tag{3.4}$$

  corresponds to the slope of the displacement function f.

The gradient function is not directly usable to detect change in the displacement function. To obtain the actual change in the map, the second order derivative defined as

$$\nabla^2 f(x, y) = \frac{\partial^2 f(x, y)}{\partial x^2} + \frac{\partial^2 f(x, y)}{\partial y^2} \tag{3.5}$$

$$\Delta f = \nabla^2 f. \tag{3.6}$$

has to be used. Δf is called the Laplace Operator and describes change in the gradient which directly corresponds to curvature in the original map.

In case the displacement map is given in a discrete form, the Laplacian Operator needs to be discretized before use. To compute the discretized form of the Laplacian Operator

the components of Equation 3.5 can be calculated independently:

$$\begin{aligned}\frac{\partial^2 f(x,y)}{\partial x^2} &= \frac{\partial}{\partial x}\left(\frac{\partial f(x,y)}{\partial x}\right) \\ &\approx \frac{\partial}{\partial x}\left(\frac{f(x+\Delta x,y)-f(x,y)}{\Delta x}\right) \\ &\approx \frac{f(x+\Delta x,y)-f(x,y)-(f(x,y)-f(x-\Delta x,y))}{\Delta x^2} \\ &= \frac{f(x+\Delta x,y)-2f(x,y)+f(x-\Delta x,y)}{\Delta x^2}.\end{aligned}$$

If we set $\Delta x = 1$, which is the minimum because of the pixel grid, we obtain:

$$\frac{\partial^2 f(x,y)}{\partial x^2} \approx f(x+1,y) - 2f(x,y) + f(x-1,y). \tag{3.7}$$

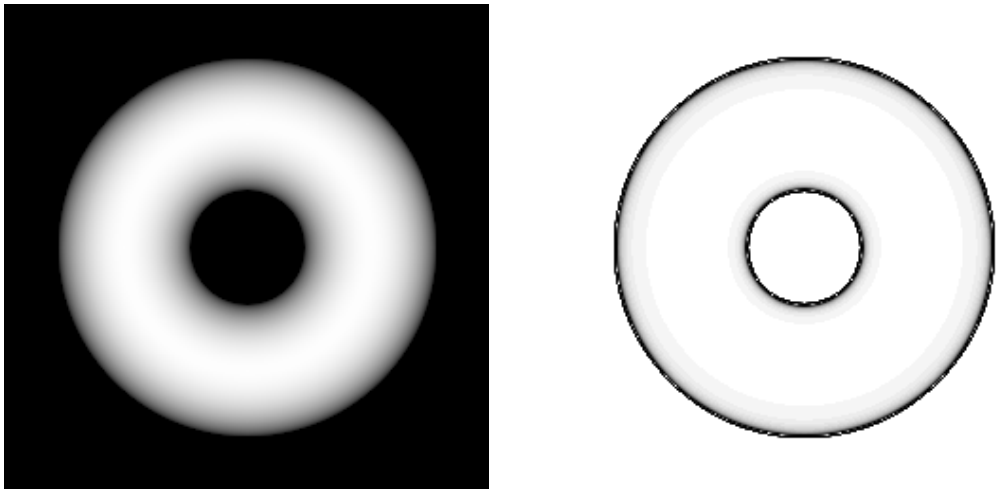
Symmetrically the second partial derivative along y is given as:

$$\frac{\partial^2 f(x,y)}{\partial y^2} \approx f(x,y+1) - 2f(x,y) + f(x,y-1). \tag{3.8}$$

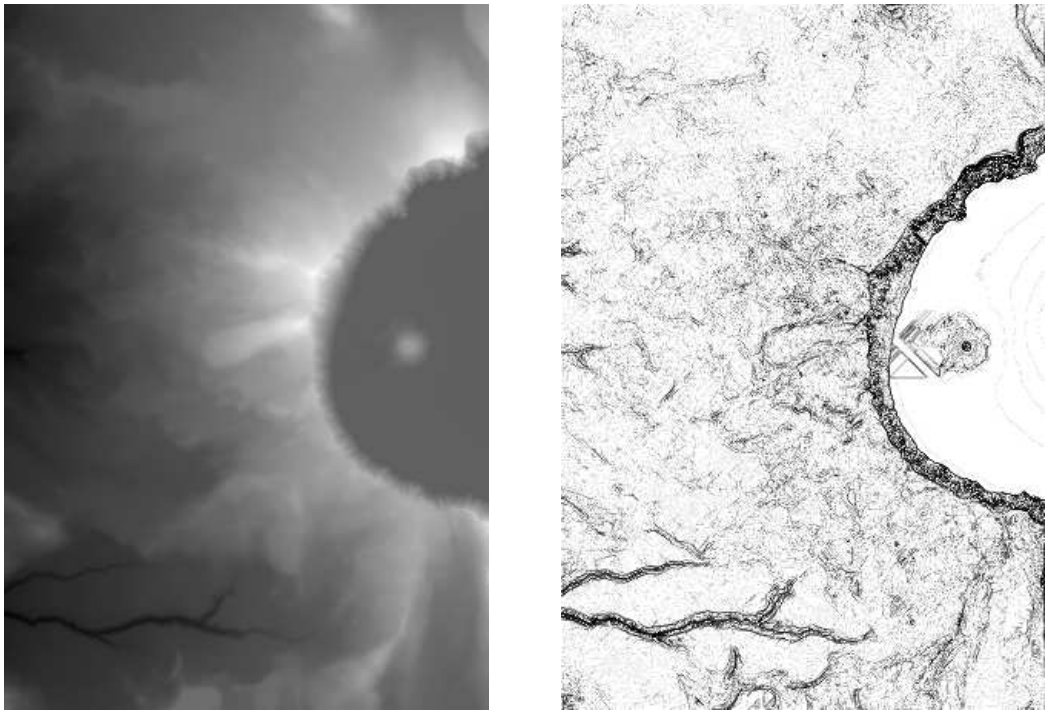
Expressed as a convolution filter the discretized Laplacian Operator L is defined as:

$$\begin{aligned}\Delta f &= \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} \\ &\approx f(x+1,y) - 2f(x,y) + f(x-1,y) + \\ &\quad + f(x,y+1) - 2f(x,y) + f(x,y-1) \\ &= \begin{pmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} = L\end{aligned} \tag{3.9}$$

In Figure 3.7 the Laplacian filter kernel was applied to two example height fields, a synthetic half donut shaped height field and a range scan of Crater Lake. While the filter works as expected on the synthetic donut shaped height field, the result of the Crater Lake height field is very noisy and the relevant geometric features are not well detected. Since the footprint of the employed Laplacian is rather small it is very sensitive to small local changes and high frequency noise often present in data obtained from laser range scanners and thus is almost unusable without any low-pass filtering applied beforehand.



(a) Half donut shaped height field, brighter colors corresponding to higher elevation. On the right the filter kernel from Equation 3.9 was applied



(b) Height field of Crater Lake with the same discrete Laplacian filter kernel applied on the right.

Figure 3.7: Discretized Laplacian filter kernel defined in Equation 3.9 applied to example height fields. In the filtered images (right side), darker colors correspond to higher curvature values.

3.2.2 Laplacian-of-Gaussian

A commonly used filter function for removing high frequency noise is the Gaussian function:

$$G_\sigma(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}}\left(e^{\frac{-(x^2+y^2)}{2\sigma^2}}\right). \tag{3.10}$$

For the actual application of the Gaussian function from Equation 3.10 it has to be convolved with the input signal, in this case with the displacement map. The convolution operator $*$ is defined as:

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(y)g(x-y)dy. \tag{3.11}$$

Marr and Hildreth [37] combined the Laplacian Operator with the Gaussian function to the so called *Laplacian-of-Gaussian* (*LoG*) Operator:

$$\begin{aligned} LoG &= \Delta G_\sigma(x,y) && (3.12)\\ &= \nabla^2(G_\sigma(x,y)\\ &= -\frac{1}{\sqrt{2\pi\sigma^4}}\left(2-\frac{(x^2+y^2)}{\sigma^2}\right)e^{-\frac{(x^2+y^2)}{2\sigma^2}} && (3.13)\end{aligned}$$

To apply the *LoG* Operator to a displacement function f it has to be convoluted with it:

$$LoG(x,y) * f(x,y) = \Delta G_\sigma(x,y) * f(x,y). \tag{3.14}$$

The Gaussian function acts as a low-pass filter and the Laplacian Operator acts as a high-pass filter effectively forming a bandpass filter.

A discretized version of the *LoG* Operator is shown in Equation 3.15. Here, a size of 9×9 sample points was used as footprint for the *LoG* Operator:

$$LoG = \begin{pmatrix} 0 & 1 & 1 & 2 & 2 & 2 & 1 & 1 & 0 \\ 1 & 2 & 4 & 5 & 5 & 5 & 4 & 2 & 1 \\ 1 & 4 & 5 & 3 & 0 & 3 & 5 & 4 & 1 \\ 2 & 5 & 3 & -12 & -24 & -12 & 3 & 5 & 2 \\ 2 & 5 & 0 & -24 & -40 & -24 & 0 & 5 & 2 \\ 2 & 5 & 3 & -12 & -24 & -12 & 3 & 5 & 2 \\ 1 & 4 & 5 & 3 & 0 & 3 & 5 & 4 & 1 \\ 1 & 2 & 4 & 5 & 5 & 5 & 4 & 2 & 1 \\ 0 & 1 & 1 & 2 & 2 & 2 & 1 & 1 & 0 \end{pmatrix}. \tag{3.15}$$

In Figure 3.8, the discretized *Laplacian-of-Gaussian* filter kernel was applied to the two example height fields already used in the previous secion, a range scan of Crater Lake

and a synthetic half donut shaped height field. As an effect of the smoothing operation the result is less noisy and distinct features as the river, visible on lower left corner of the Crater Lake image, are enhanced.

3.2.3 Curvature Based Testing Schemes

Direct evaluation of the curvature of a displacement map at a specific position is quite expensive due to the amount of memory accesses necessary and thus it is impracticable for hardware implementation or even real time applications. On the other hand it is not necessary to store the absolute curvature values of the displacement map in a second map and evaluating it in a test function. Instead the information about where to sample is stored in another map. Two example schemes are presented here, a very simple and space efficient scheme and a more sophisticated algorithm.

Decision Maps

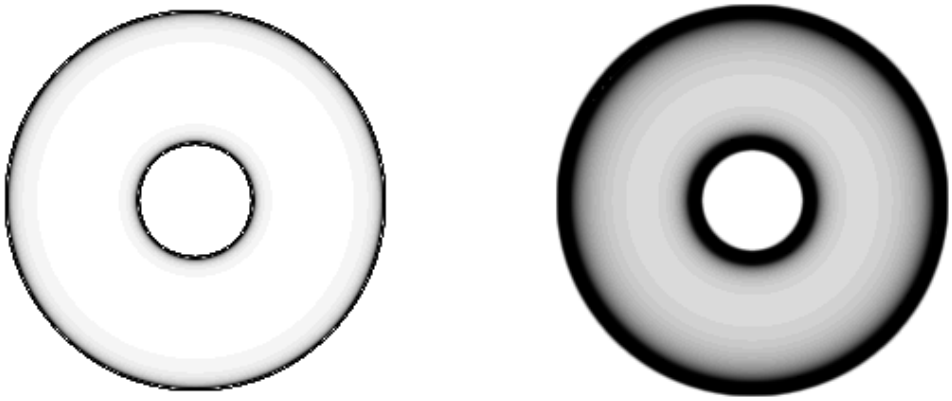
This new scheme is fairly easy to implement and requires almost no preprocessing other than calculating the curvature. It is very light on resources while performing the actual sampling [3].

After the curvature of the displacement map has been calculated all curvature values are compared to a user defined threshold and all values above the threshold are marked. The resulting one bit deep image has non-zero values at positions where the displacement map has high curvature values, thus needs to be sampled more accurately. This one bit image is further-on called *Decision Map* as it controls the sampling process.

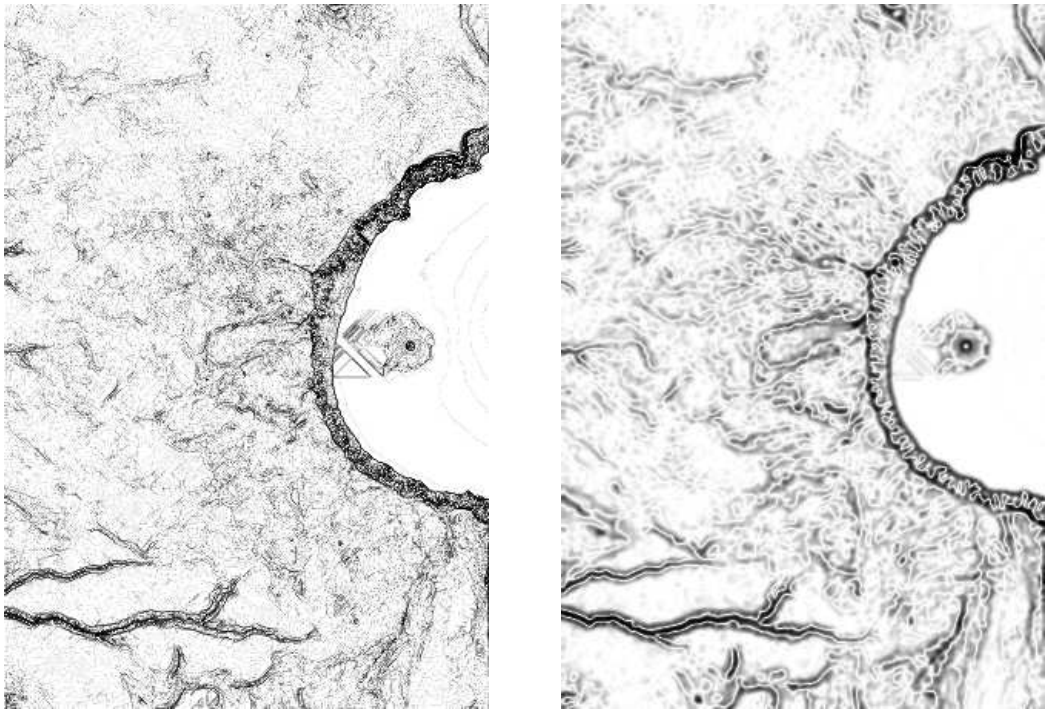
Although this information alone is already sufficient to control the sampling process the generated decision bit is again subject to point sampling artifacts when structures as in Figure 3.2 are present. To avoid this, all non-zero values in the *Decision Map* are spread out in a circular manner by a user defined radius as shown in Figure 3.9, effectively enlarging the area of influence of values with higher curvature than the threshold.

In Figure 3.10 the *Decision Map* creation process for the Crater Lake height field is shown. Later in the rendering process it is only necessary to lookup the value in the *Decision Map* to decide whether to sample more accurately at a specific position. The additional storage necessary for the *Decision Map* is very moderate and the additional lookup can be avoided when the map is stored with the displacement map by enlarging the displacement map entries with the one extra bit.

The specification of the threshold and the enlargement size of a sample above the threshold is not straightforward and depends strongly on the displacement map and the



(a) Discretized Laplacian (left) and the Laplacian of Gaussian filter kernel (right) applied to the half donut shaped height field.



(b) The same filter kernels as in (a) applied to the Crater Lake height field.

Figure 3.8: Discretized *Laplacian-of-Gaussian* defined in Equation 3.15 applied to the same height fields as in Figure 3.7 with the result from Figure 3.7 shown on the left for comparison.

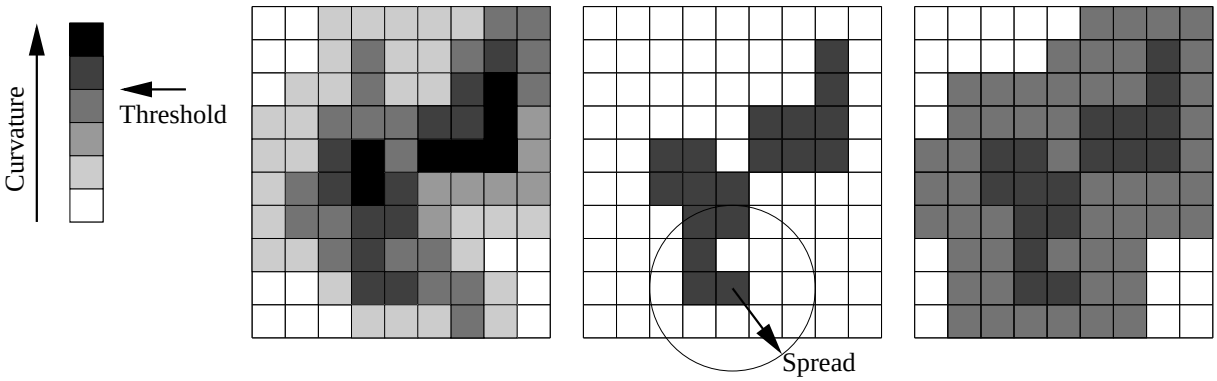


Figure 3.9: A small section of the curvature of a displacement map with the resulting *Decision Map*. On the left the curvature of the map is shown with darker values corresponding to higher curvature. In the middle the result of the thresholding and on the right the spreading is applied, too.

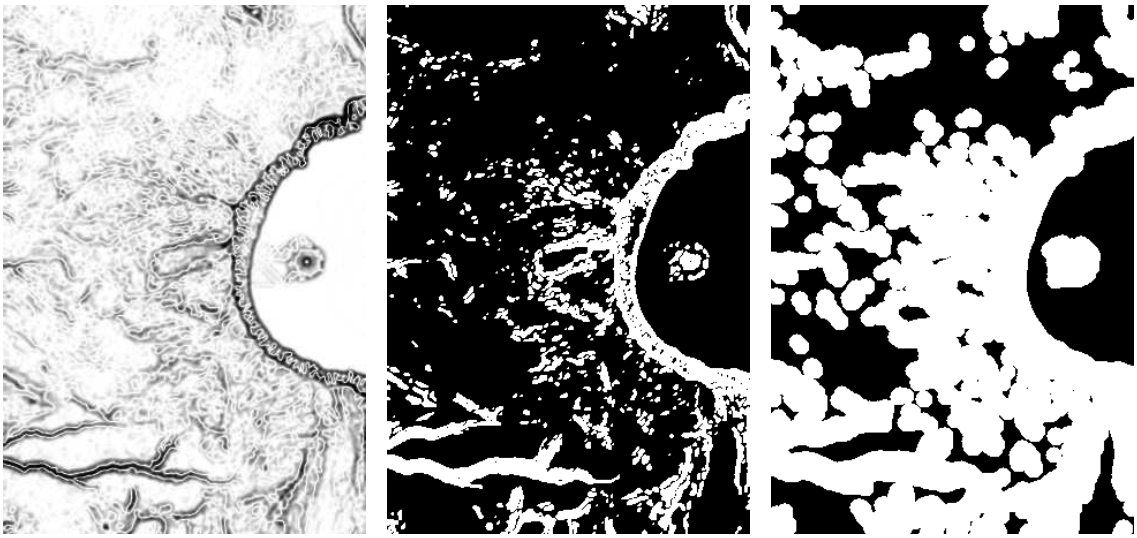
geometry it is applied to, which makes this scheme somewhat difficult to use. The sampling information is due to the only one bit deep *Decision Map* of course very limited and every region that has a non-zero *Decision Map* value will be sampled until the maximum sampling accuracy of the rendering algorithm is reached.

Curvature Maps

The Decision Map introduced in Section 3.2.3 can produce satisfying quality if some care is taken about the parameters and the base domain surface sampling density. To make the sampling process more adaptive to the displacement map's shape, more information than just one bit has to be stored. In the new *Curvature Map* [4] not only the information where to sample is stored, but also under which circumstances, more specifically when to stop.

Typically adaptive sampling is a tail-recursive process. Starting from a coarse level of resolution a refining process is performed, that continues until a chosen error metric drops below a given threshold or a maximum recursion level is reached. In order to get more control over the adaptive sampling process the error threshold should be dependent on the present recursion level. When the recursion level increases, the threshold should increase as well because the covered area of the sample point is much smaller and thus should only be resampled when large changes in curvature values are present.

In Figure 3.11 an example of a recursive refinement scheme is shown. The algorithm begins with the outer two black sample points and evaluates the middle point of the edge, h. The curvature at point h is, although smaller than at point e, large enough to trigger



(a) Laplacian of Gaussian **(b)** Thresholded curvature **(c)** Final *Decision Map*

Figure 3.10: *Decision Map* creation for the Crater Lake height field. After thresholding the *LoG* filtered image all samples are spread out to avoid sampling errors. White areas in the thresholded image correspond to curvature values above the threshold and white entries in the *Decision Map* mark areas where more sampling is necessary.

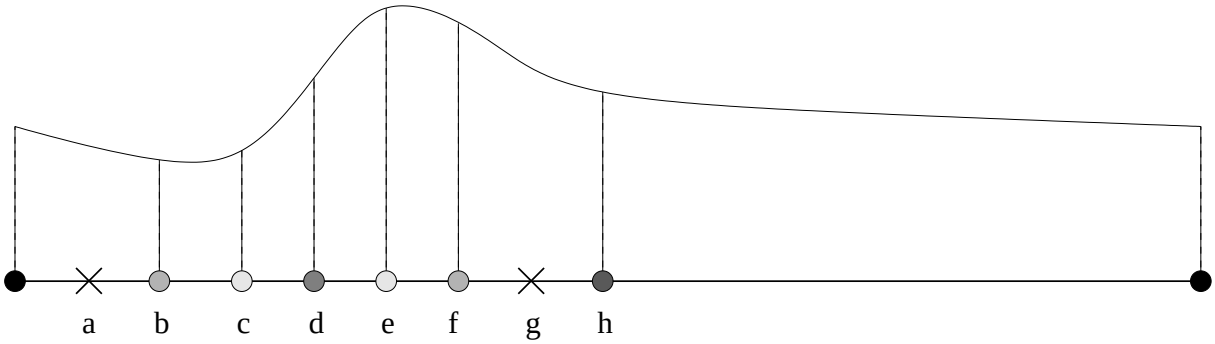


Figure 3.11: Example of an adaptive sampling scheme. Lighter sample colors indicate a higher recursion level.

the insertion of the point. The algorithm then continues in a similar manner to insert points b, d and f. After point f has been inserted, e and g are tested. At point e the curvature is high enough to trigger the insertion of the point even though the recursion level is higher but at point g the curvature is, although slightly higher than at point h, not high enough. This corresponds nicely to the shape of the curve as no more detail would be added by adding point g and unnecessary oversampling would occur at that position. Similarly for point a, where the additional point wouldn't add any additional accuracy to the sampling result.

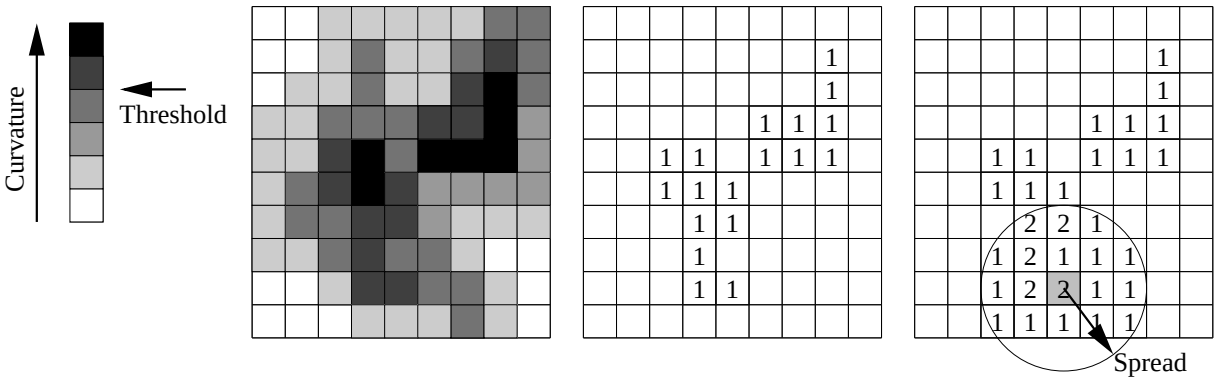


Figure 3.12: A small section of the curvature of a displacement map and an example step when creating the corresponding *Curvature Map*. On the left the curvature of the map is shown with darker values corresponding to higher curvature. In the middle the result of the thresholding and on the right side the spreading for the darker colored sample has been done. To complete the *Curvature Map* the spreading has to be performed for all positions in the middle image.

To calculate the *Curvature Map* a similar approach as for the Decision Map is used. The displacement map is processed using the *LoG* filter and the resulting curvature values are compared against a user defined threshold and all values above the threshold are marked with one and stored in a map. To avoid missing small features the non-zero values are spread out by drawing a circle with a radius depending on the curvature value, higher curvature corresponding to a larger circle, around the entry in the map. The circle is drawn by increasing the map entries by one as shown in Figure 3.12.

The spreading of the thresholded values results in higher values in the *Curvature Map* for samples in proximity of values of high curvature. To ease the evaluation of the *Curvature Map* during rendering, only the maximum tessellation recursion level for that further sampling should occur is stored in the *Curvature Map*, so that by a simple comparison a decision can be made while rendering.

As the range of possible recursion levels is usually far smaller than the range of possible entries in the *Curvature Map*, it has to be quantized to allow for a one-to-one mapping from recursion level to *Curvature Map* entry. The quantization can be performed either in a linear fashion with equidistant ranges per recursion level or an exponential mapping with increasing ranges for higher recursion levels, as a preprocessing step.

To map an entry c from the unquantized *Curvature Map* we have to find the interval f_d, where d corresponds to the corresponding recursion level, so that:

$$d \in [0 : r_{max}] : c \in f_d = \left[e^{dk}, e^{(d+1)k}\right[\tag{3.16}$$

where r_{max} is the maximum recursion level, and k is derived from the maximum value f_{max} in the unquantized *Curvature Map*:

$$k = \frac{1}{r_{max}} \log (f_{max}), \tag{3.17}$$

In Figure 3.13 unquantized *Curvature Maps* and final *Curvature Maps* for the example displacement maps from Figure 3.7 are shown.

3.3 Quality and Error Control

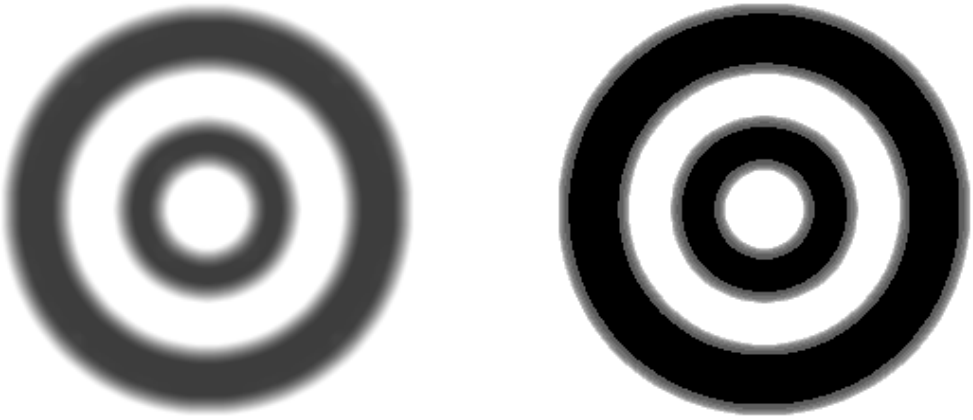
As there exists an unlimited number of possible displacement maps and base domain surfaces it is important to measure the performance of the different sampling tests and combinations of them. When the sampling tests are used for tessellation as described in Section 4.3 the quality of the tessellation result can be measured by calculating the difference between the resulting mesh and an optimally tessellated mesh. An optimal mesh with respect to accuracy for a discrete displacement map can be easily created by using one sample point per position in the original displacement map and adding a vertice at that position. Any additional vertex will not add any more information to the mesh as it is only a linear combination of the already existing points.

The distance between two triangle meshes can be measured using the algorithms devised by Cignoni et al. [19]. The distance is calculated by sampling the surfaces with different algorithms like *Montecarlo* sampling, *Subdivision* sampling and *Similar Triangles* sampling. This error measure is basically the integral of the simple vertical error across the triangle mesh surface.

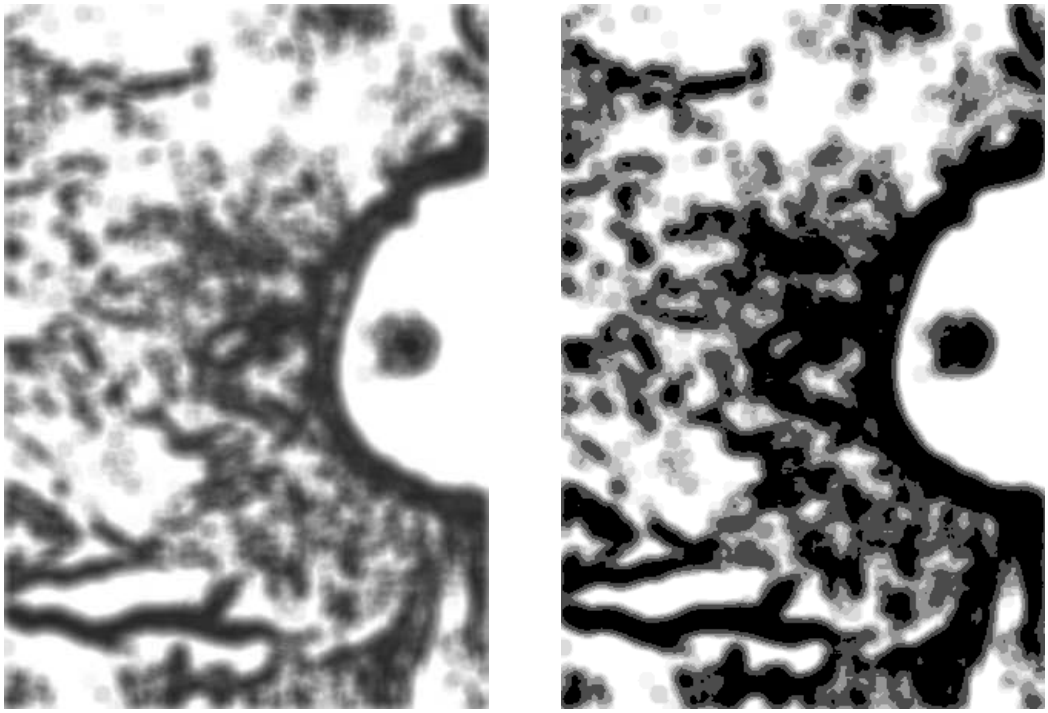
3.4 Conclusions

Different schemes for sampling displacement maps are feasible with distinct advantages and weaknesses. In order to obtain satisfactory results a combination of the schemes is often necessary. The most difficult problem with sampling displacement maps is the locality of changes, it is almost impossible to predict the shape of a displacement map at a specific position, even when the surroundings are fully known. To avoid sampling artefacts by missing distinct geometric changes in the map it is necessary to presample the map to some extend and identify important areas.

Efficient ways to detect these areas are to measure the curvature of the map and to store it in a second map, a *Curvature map*. Altough the *Curvature map* contains only



(a) *Curvature Maps* for the half donut shaped height field before (left) and after(right) quantization



(b) Unquantized and quantized *Curvature Maps* for the Crater Lake height field

Figure 3.13: Curvature Maps for the donut shaped displacement map and the Crater Lake height field from Figure 3.7.

the areas of change in the displacement map, so that it is very easy to decide whether a specific position needs to be sampled more accurately, the problem of finding the position in the first place persists. By spreading out the information about the curvature it can be at least detected whether a point in the map is in close proximity to a change in height. If a suitable spread radius is chosen sampling errors caused by missing features can be avoided. The process of finding an appropriate radius is somewhat cumbersome as it depends not only on the input height field, but also on the used base domain surface.

Alternatively the average height of an area can be precomputed and stored in a summed-area table. This allows to determine the average height of any rectangular area in the map, and especially to compare the average height of regions in the map. Hardware support for fast reading from summed-area tables is not available nor expected anytime soon so it has to be emulated using normal texture memory, requiring four read accesses.

A convenient way to detect changes in a map is to simply compare the surface normals of the resulting displaced vertices. The distinct advantage is that no extra memory or lookup tables are necessary to perform the test. Although this is of course again subject to aliasing, being a form of point sampling, it works very well when the input data has only small features like ripples or small bumps and combines very well with the height average test.

Chapter 4

Algorithms for Hardware Rendering

Rendering displacement mapped surfaces is a process that involves a significant number of geometric and arithmetic operations. When applied to a triangle mesh, it involves prior retessellation of the base domain surface and transformation of the vertices and normals. Even on fast CPUs, it is a time consuming operation, wasting bandwidth and processing power. As for all expensive rendering algorithms, the ultimate goal is to reimplement them using dedicated hardware or reusing present (graphics-) hardware to get interactive rendering performance and reduce the strain on the CPU. Currently only one commercially available graphics card [38] is capable of directly rendering displacement maps. Adding displacement map rendering to currently available hardware architectures presents several problems. For analyzing the capabilities of present hardware architectures, the OpenGL rendering pipeline can be used as a starting point.

4.1 The OpenGL Rendering Pipeline

A defacto standard for 3D graphics rendering is the OpenGL graphics API introduced by SGI in 1993 [42]. It provided an open standard for writing 3D graphical programs and a standardized way to access the available 3D graphics hardware. In Figure 4.1 a schematic diagram of OpenGL is shown. The vertex and pixel data flow as shown in the Figure is still mostly valid for todays graphics hardware, only the computing power of some units has seen a massive increase in the last few years, in particular the *Vertex* and *Fragment Processor*. Apart from the vastly improved computing power the flexibility and programmability of these processors has increased. Some important units of the pipeline are described in more detail in the next sections.

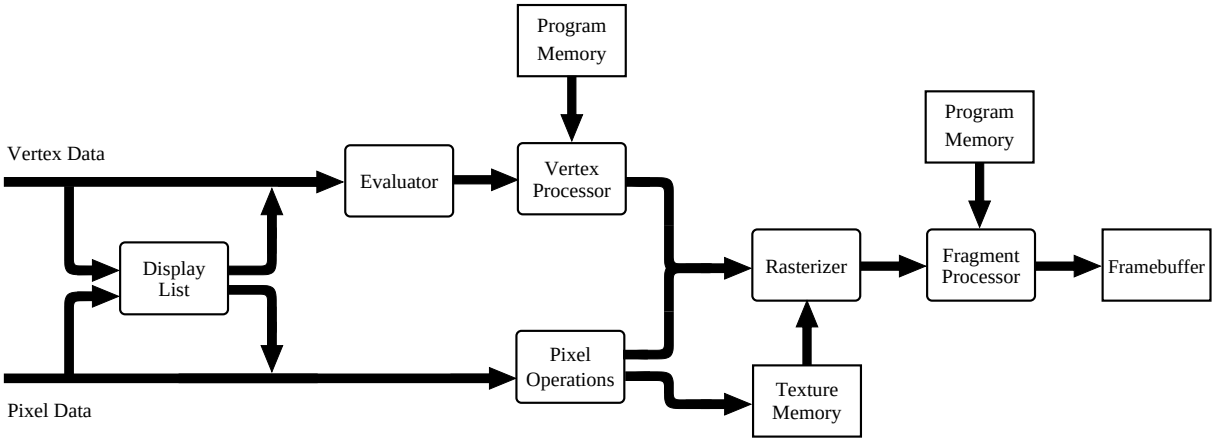


Figure 4.1: Block diagram of the OpenGL pipeline.

4.1.1 The Vertex Processor

The *Vertex Processor* is responsible for executing all operations that have to be done per vertex like the transformation from object space to view space, and the lighting calculation for Phong shading [40]. The first generations of OpenGL graphics architectures featured only hardwired transformation and lighting operations and were actually calculated by the CPU since the computing power in the graphics hardware pipeline was not sufficient. This has undergone a great change, the fixed operation unit was replaced with an programmable vector processor.

The *Vertex Processor* processes one vertex at a time, completely isolated from the surrounding geometry and has no concept of higher order geometric shapes or even simple objects like triangles. All the data associated with the vertex being processed is transferred to the vertex processor, the geometry data – vertex position and surface normal if applicable in model space – color information and texture coordinates. The input data is then transformed either using a fixed function transform with a user supplied transformation matrix, or in case of a programmable processor a corresponding vertex processor program. The transformed data is thereafter fed into the rasterizer, where all coordinates are linearly interpolated across the primitive the vertex belongs to.

4.1.2 The Fragment Processor

The *Fragment Processor* operates on the interpolated fragment or pixel values generated by the rasterizer. In contrast to the *Vertex Processor* it has access to the graphic hardwares texture memory. The amount of texture memory and the number of accesses to the memory that can be made for every pixel is implementation specific, as well as the

number of operations that can be performed for every pixel. The performance and programmability of this unit has increased in an equal manner as the *Vertex Processors*. In current implementations, the unit is provided with interpolated position, surface normal, color and multiple texture coordinates.

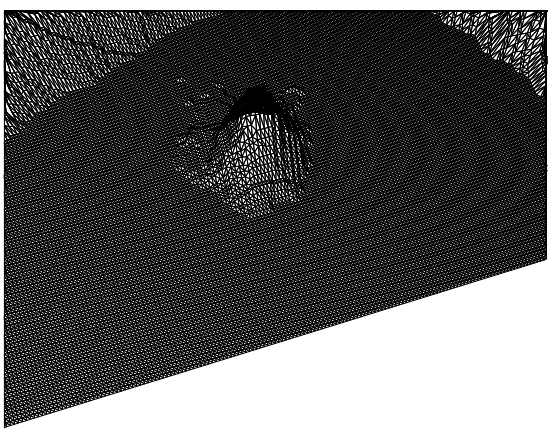
The command set is usually smaller and less powerful than in the vertex stage, which is due to the fact that the data throughput is considerably higher in the fragment processing stage, since the operations are applied to every pixel and not only to every vertex.

4.2 Rendering using Tessellation

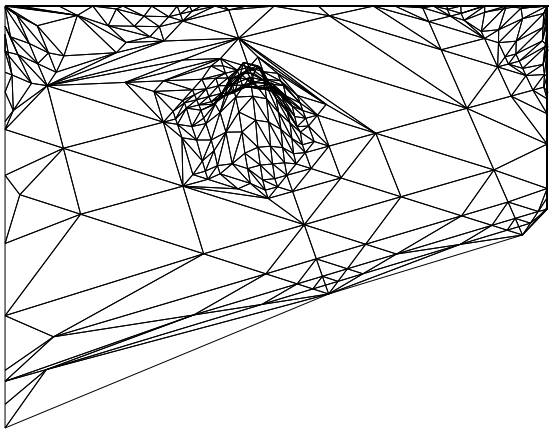
A straightforward way to enable present hardware architectures is to retriangulate the original mesh according to the displacement map and displace the vertices before the actual rasterization is performed. In general, the resolution of the base domain surface is not sufficient to capture the detail contained in the displacement map, as it would result in one vertex per sample point of the displacement map. Although this obviously leads to a very high accuracy, the amount of triangles that needs to be rendered is also very high. For a $n \times m$ sized displacement map, $2 \times n \times m$ triangles. There is no straightforward way to limit the tessellation of the base domain surface, if the displacement map is not discrete. The amount of triangles created is also completely independent of the structure of the applied displacement map, and many unnecessary triangles are possibly rendered.

4.3 Adaptive Tessellation

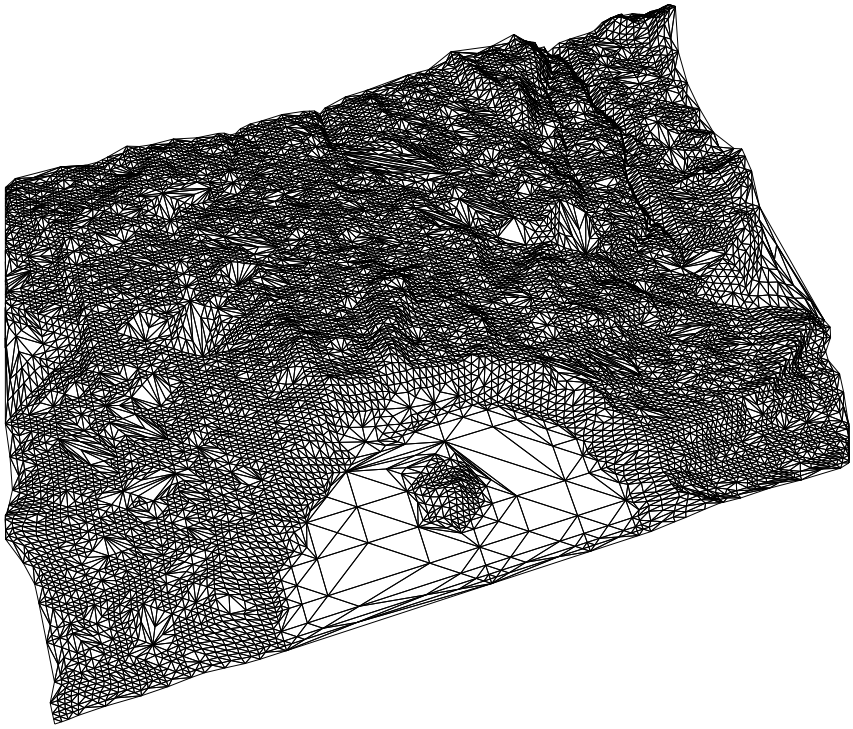
A possibility to overcome these problems is to tessellate the individual triangles sequentially and to adaptively add triangles where necessary, until a desired level of accuracy is reached. When performing adaptive tessellation, it is important to split adjacent triangles consistently to avoid t-vertices, as shown in Figure 4.3. This simple example contains two adjacent triangles, where only one was scheduled to be split. As a result on the common edge of the two triangles a vertex is added only on the triangle to be split. When the displacement is applied to the triangle vertices afterwards, the edge on the unsplit triangle will be linearly interpolated, while for the split edge the new vertex is displaced according to the corresponding displacement map entry. If the displacement map does not happen to change linearly between the start and end point of the adjacent edge a gap will open between the vertex and the unsplit edge resulting in a hole in the rendering of the triangle mesh.



(a) Fully tessellated



(b) Adaptive tessellation



(c) Zoomed out view of an adaptively tessellated triangle mesh with the Crater Lake height field applied

Figure 4.2: Unnecessary triangles in an even region of the source displacement map. Adaptive tessellation, as used on the right side, can reduce the triangle count massively. The complete mesh is shown below.

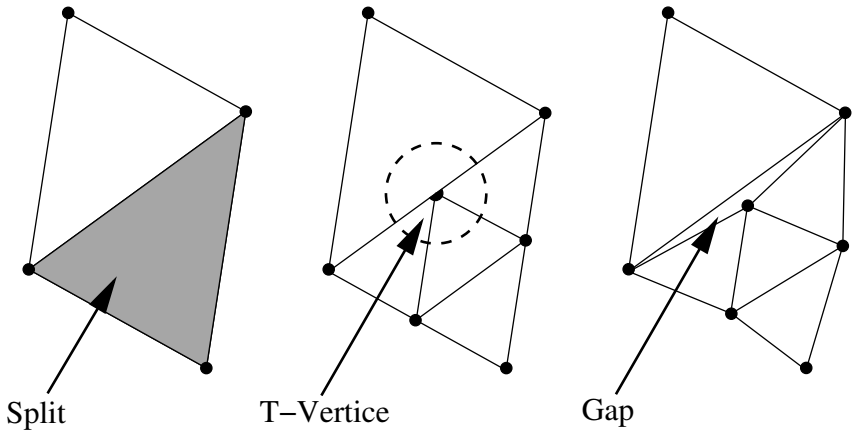


Figure 4.3: T-Vertice caused by an asymmetric split of adjacent triangles. Displacing the vertices afterwards may result in a gap as shown on the right.

While many remeshing algorithms for terrain or height field approximation using triangular meshes exist, few are suitable for hardware implementation because of memory constraints. A common strategy for finding an optimal solution is the introduction of an error metric for a triangulation and iteratively minimizing the global error. Measuring the global error of a triangle mesh involves storing the whole triangulation result – $O(n)$ space and time is necessary – which may be easy though expensive in CPU oriented algorithms, but in an algorithm targeted at existing graphics hardware architectures the amount of random access memory is usually very limited and not optimized for this kind of operation.

Garland et al. [25] analyze variations of the greedy insertion algorithm. The greedy insertion algorithm tessellates by successively inserting vertices at the position with the largest error and continues until either a given number of vertices is present or the maximum error is below a given threshold. A naive implementation of this will consume $O(n)$ time per iteration, and for an iteration limit of m resulting in m inserted vertices $O(nm)$ time. With the use of a heap, and updating the error only where necessary, Garland reduces the time cost to $O((m+n)\log m$, but the memory consumption is increased to $O(m+n)$. The heap enables the algorithm to easily pick the point with the largest error and also to compare it to a some threshold. The algorithm starts with a quadrilateral and inserts points, selecting the position with the largest error. The resulting mesh is a Delaunay triangulation. As discussed in Section 4.1, graphics pipelines are usually stream oriented with respect to the geometry, and the random access memory buffers that might exist for storing such data are strictly limited in size, and only portions of the mesh can be stored at a time. The necessary presegmentation of the input triangle mesh compli-

cates the retessellation process in particular at the segment borders. A simpler scheme for remeshing, that may not produce the optimal result but rather uses conservative assumptions about quality and still be adequate for a possible hardware implementation is needed. Thus, only local error criterias can be used, specifically criterias local to at most one triangle.

4.3.1 Local Edge only Tessellation

It is already a difficult task to maintain connectivity inside a mesh, while retessellating when using a CPU based algorithm. When moving to a custom hardware algorithm, it becomes even more challenging, mainly due to memory access restrictions. While storing and using the connectivity of a triangle may be feasible, accessing the actual data of an adjacent triangle would require random read and write accesses to the memory where the triangle list of the mesh is stored. This would hurt pipeline performance because of memory latency and because of the necessary write access, in this case to store the information about the updated connectivity, would make parallel execution very complicated if not impossible.

To avoid these problems the tessellation is performed on a per-triangle-basis where information local to that triangle is used.

This effectively limits the possibilities of the vertex insertion decision to criterias local to a triangle. Furthermore, if t-vertices are to be avoided, the available information is limited to the common edge, as using information about the third, opposing triangle vertex or the area within the triangle will not be accessible by the adjacent triangle. Thus, the decision is made solely based on information contained in the edge that is to be split. The tradeoff of this scheme is that every test performed for the decision is performed twice, for both triangles adjacent to an edge, although this could be partly avoided through the use of an edge buffer in the testing stage.

The decision whether and where to insert vertices can be made upon the sampling tests introduced in Chapter 3.

4.3.2 Triangle Tessellation Strategies

A great number of schemes for tessellating triangles exist. Because of the limitations imposed on algorithms targeted at a hardware implementation, only few are actually considered in this thesis. For retessellating a triangle new vertices have to be inserted, possibly on an edge or inside the triangle. It is necessary to allow inserting vertices

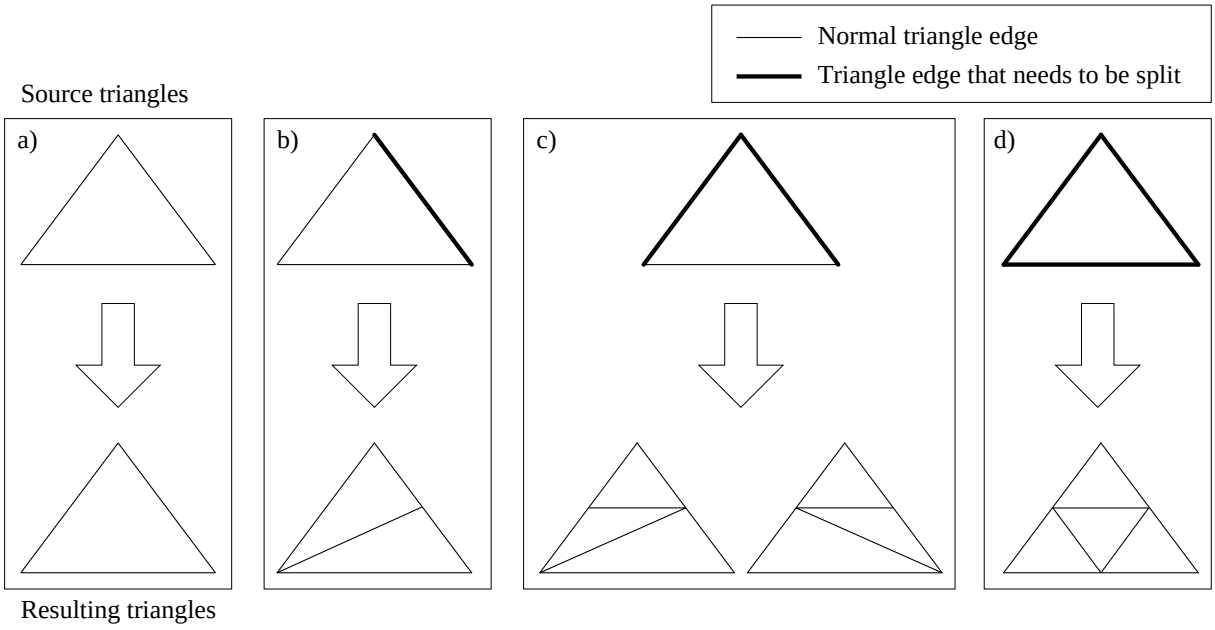


Figure 4.4: Possibilities for splitting triangles.

along the edges since otherwise the base triangle edges remain unchanged throughout the tessellation process. If a vertice is added on a triangle edge, it is important to ensure that the neighboring triangle is split equally along the shared edge as t-vertices could be created and cracks may appear in the surface, as shown in Figure 4.3. If we limit the insertion of vertices to one per triangle edge, the number of possible constellations for the resulting new triangles is limited to four different cases. In Figure 4.4, the four cases with the resulting splits are shown. For the cases *b*) and *c*) the symmetries for the input triangles are not considered. In case *c*) the resulting triangulation is ambiguous as two symmetrical cases exist. As a solution to this and to avoid long and narrow triangles, the shape of the source triangle can be used. As it can be seen in Figure 4.5, the length of the bisecting line of the opposite corner to the split edge can be used as criteria. If the shorter bisecting line is inserted as in *a*) the resulting triangles are more regular in shape and the long and narrow triangles as in *b*) are avoided.

4.3.3 Vertex Insertion and Position Modification

Often the quality of a mesh with displaced vertices can be improved without adding new vertices, just by moving vertices to a position that adapts better to the displacement maps shape. In Figure 4.6, the inner two vertices of the edge were moved to the closest points on the edge with a maximum in curvature. The resulting linear approximation of the curve

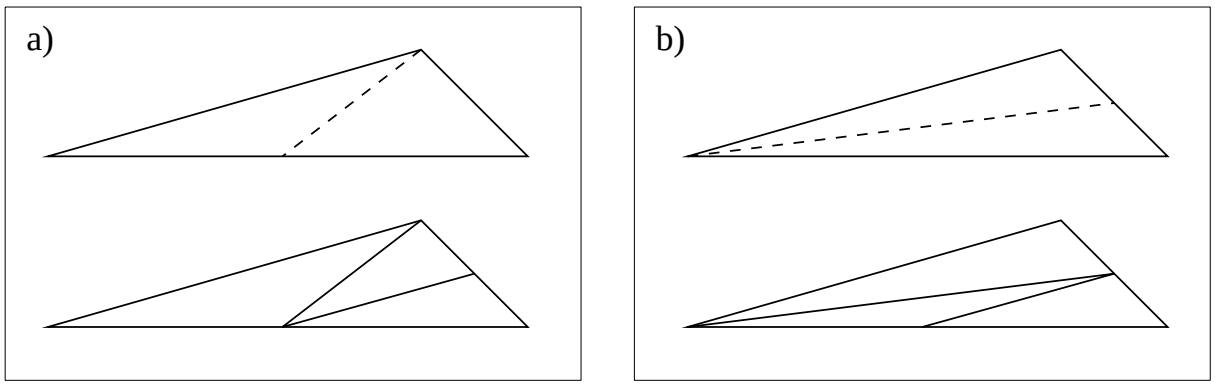


Figure 4.5: The two symmetrical tessellation options for a triangle with two split edges. In case *a*) the bisector used for splitting the triangle is shorter than in case *b*) resulting in smaller and more regular triangles.

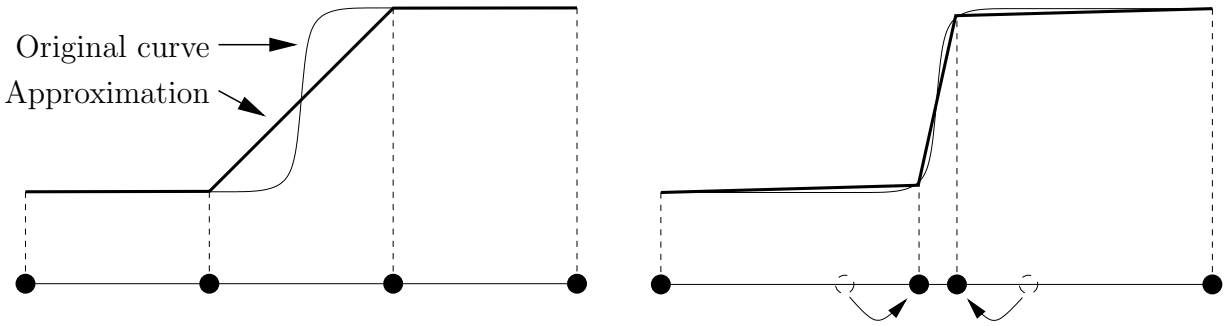


Figure 4.6: Movement of vertices on an edge to improve the sampling and the resulting linear approximation of the original shape. The thin line represents the original displacement map shape and the thicker lines the resulting approximation created by the respective sampling. On the left side the original sampling is shown and on the right side the sample points were moved to points with higher curvature thus improving the sampling result.

adapts far better to the sharp edge in the middle of the curve. If the original vertices cannot be modified due to the nature of the remeshing scheme, the positioning of new vertices also has great impact on the outcome. In Figure 4.7 an edge with a sharp peak on the right end is sampled. On the left side new vertices are only added in the middle of the edge, and on the right side the new vertice is added at the point of maximum curvature along this edge. To achieve a similar sampling quality with the midpoint insertion, a lot more sample points are necessary as if the sampling position is variable. The position adaption to the curvature features is difficult to implement though, as it is non-trivial to find these feature points without sampling the whole edge.

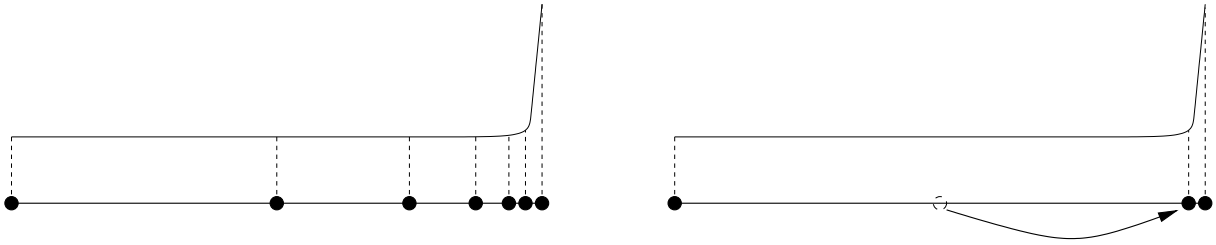


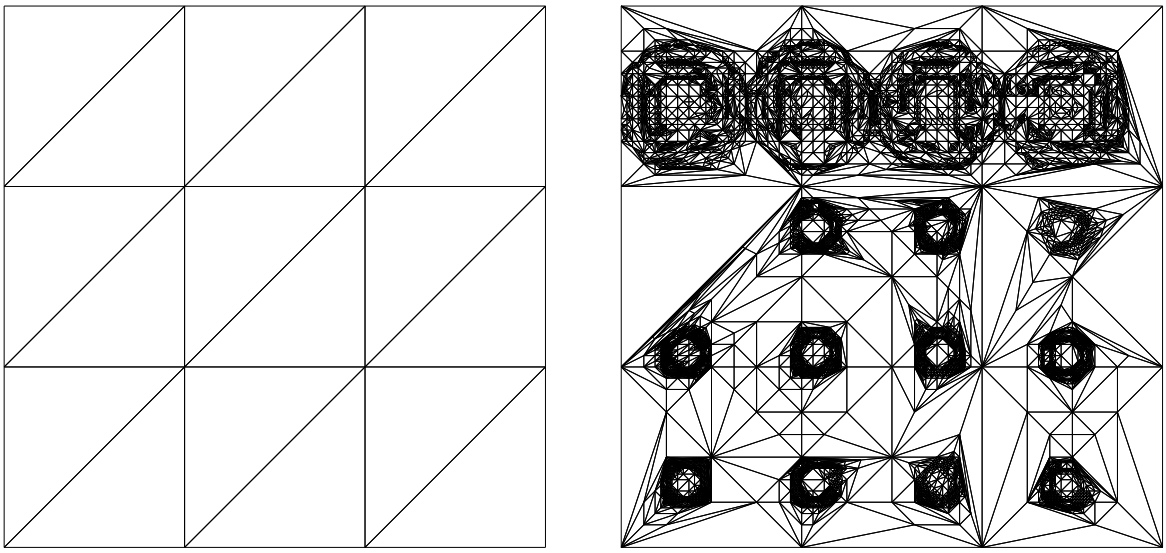
Figure 4.7: Midpoint insertion causing unnecessary oversampling. On the left side the new vertices are only inserted in the middle of the edge while on the right side the new vertices are moved to the point of highest curvature on the edge.

4.3.4 Base Domain Surface Tessellation

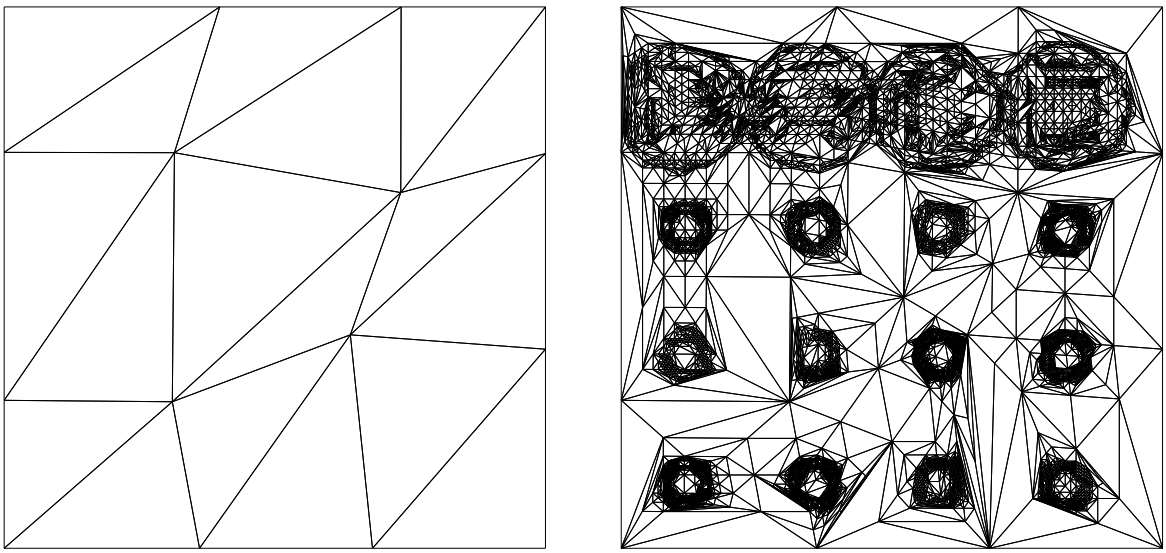
Not only the amount of vertices in the base domain surface has a strong influence on the quality of the outcome, the arrangement of the vertices inside the base domain surface may have an effect. Highly regular base domain surface triangulations, as they are usually used when the base domain is synthetic, a flat square for example, can cause visible artefacts, because the base triangulation tends to remain somewhat visible even in the final tessellation result. Additionally, if the applied displacement map is also very regular with a frequency close to the base tessellation vertex distances, it can lead to severe sampling errors as shown in Figure 4.8. Because of this it is favorable not to use too regular tessellations or add a little jitter to the inner vertices.

4.4 Basic Tessellation Pipeline

To combine the functionality of the tests with the triangle splitting a pipeline has to be designed. In Figure 4.9 a top-level view of a basic tessellation pipeline is shown. It works by recursively subdividing triangles with the use of a triangle stack. The base domain surface triangles are fed to the pipeline and buffered in the *Triangle FIFO*. In case that the *Triangle Stack* is empty the first triangle in the FIFO is selected and forwarded to the *Triangle Test Unit*, where some of the tests described in Section 3 are performed on the triangle. If the *Triangle Test Unit* detected that some edges need to be split the triangle is moved to the *Triangle Split Unit*, otherwise the triangle is forwarded to the rasterization pipeline. When a triangle needs to be split the *Triangle Split Unit* tessellates it according to a user definable rule, for example as described in Section 4.3.2, and feeds the new triangles to the *Triangle Stack*. To avoid an overflow in the *Triangle Stack*, it has a higher priority than the input *Triangle FIFO*. The *Triangle FIFO's* status is supervised by the operating systems driver, and no overflow should occur. All the test for the performance



(a) Regularly tessellated base mesh with a height field containing a sequence of sharp dents.



(b) Jittered base mesh tessellation avoiding the sampling error on (a) caused by the too regular base mesh tessellation

Figure 4.8: Applying jitter to the inner vertices of a very regular triangle mesh to avoid visible artefacts and sampling errors.

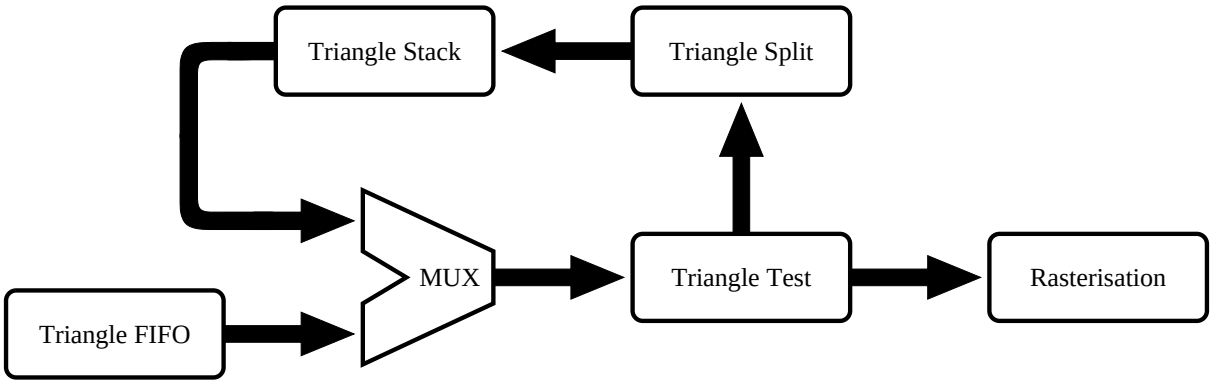


Figure 4.9: Basic tessellation pipeline for adaptive tessellation.

and quality of the tests introduced before in Section 3 and the triangle splitting and vertex insertion strategies were tested with a similar pipeline architecture. Some of the tests can reuse functional units present later in the rendering pipeline. For example the *Surface Normal Variance Test* presented in Section 3.1.1, needs to perform a bump operation on the surface normal of a vertex that can also be calculated in the vertex shader stage or even a dedicated bump mapping unit. If possible, the redundant implementation should be avoided, for example through the use of a feedback loop from the rendering pipeline to the testing unit.

4.5 Tessellating with the OpenGL Pipeline

A standard OpenGL pipeline as described in Section 4.1 is capable of rendering the previously tessellated and displaced triangle meshes. However, in order to reduce the computational load on the CPU, the tessellation should be done on the graphics hardware. In the following Section the lacking functionality is identified and the necessary additions are presented. As a base for the analysis currently available high-end commodity graphics hardware like a *nVidia NV3X* or *ATI R3XX* was used.

The computational power in the *Vertex Processor* – called *VP* further on – stage is sufficient to perform the necessary operations, but some essential functionality is missing. For displacing the vertices, a crucial feature is access to some sort of texture memory to read the displacement map data. Without any texture storage, only procedural displacement maps or very small maps that fit into the constants register storage usually present in *VP*s, can be applied. Once the displacement amount is obtained the actual displacement of the vertice is a fairly trivial operation, and a previously tessellated mesh can be displacement mapped. For performing the tessellation in place more connectivity

information is needed though, as the *VP* only operates on single isolated vertices, with no information about neighboring vertices or connectivity, which is important for most sampling tests, as they rely on information about an edge. Additionally it is not possible to insert new vertices into the rendering pipeline as it is required for creating new triangles. Considering the highly optimized pipeline in the *VP*, it makes presumably more sense to insert a tessellation unit before the pipeline and only calculate the displacement and normal perturbation of the vertices in the *VP*. A possible simple tessellation unit was demonstrated in Section 4.4.

4.6 Vertex Processor Feedback Loop

The *VP* in almost any currently available card has great computational power and flexibility, as already described before in Section 4.1.1, and could be of great use for the tessellation process. To allow the tessellation units to exploit the computational power in the *VP* some changes have to be made to the pipeline. If a vertex buffer is added after the *VP*, creating a feedback loop back to the testing units, the *VP* could be used as a sort of co-processor in the tessellation process. As a major advantage of this new approach, the change to the existing pipeline is fairly minimal and unintrusive. New and possibly better testing schemes can be added, as the flexibility and arithmetic performance of new *VP*s increases. The resulting pipeline is shown in Figure 4.10.

The arithmetically complicated test unit required in the basic pipeline in Figure 4.9 can be replaced with a mere control unit. Whenever a triangle that needs to be tessellated – for displacement mapping for example – a *VP* program for performing the tests is applied to the triangle vertices. For accessing the displacement map data, the control unit needs access to some sort of dedicated graphics memory. If the *VP* has texture memory access of its own, it can of course be omitted in the control unit. The calculation result is then fed back through a vertex buffer to the tessellation control unit, where further tessellation or the final rendering is initiated. To avoid pipeline stalls, the triangles are fed back into the vertex stream through buffers, after the necessary splits have been calculated. The changes to the rendering pipeline are completely transparent to the *VP* as it makes no difference if the vertices are transformed and rendered, or some other testing operations are performed.

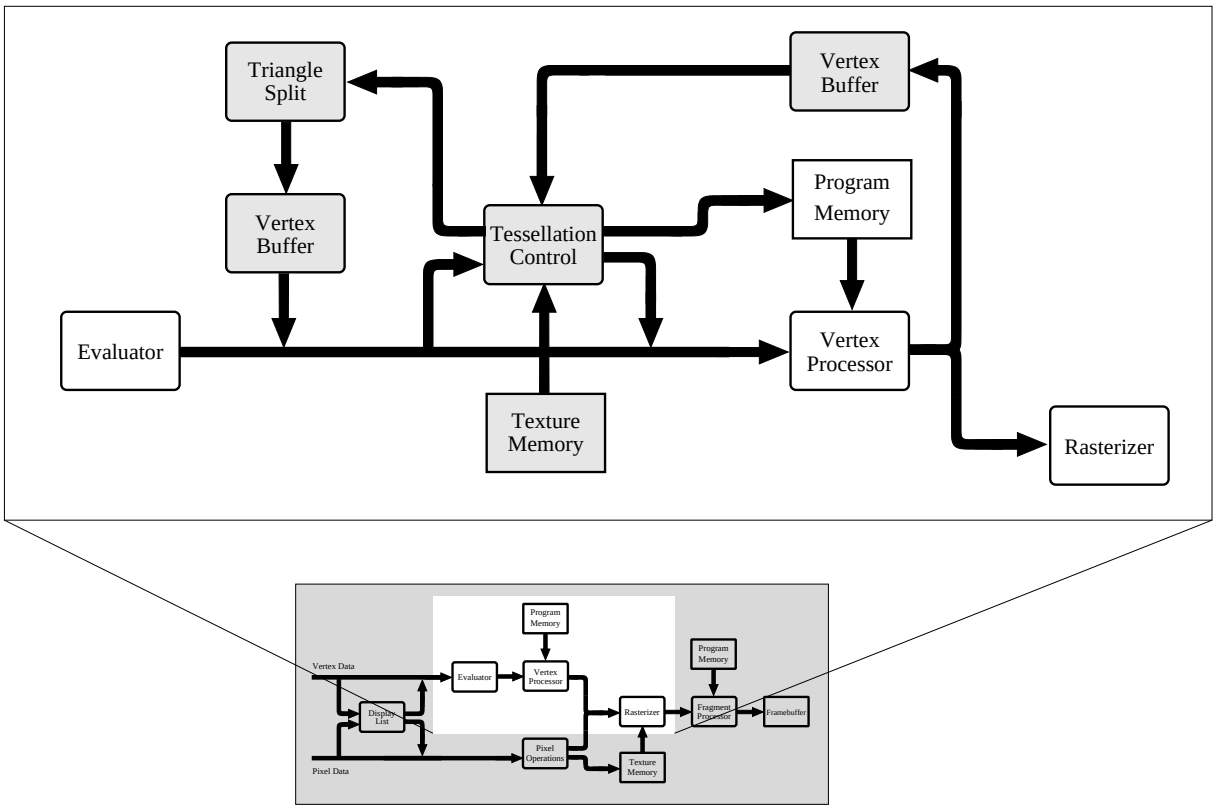


Figure 4.10: The basic OpenGL pipeline from Figure 4.1 modified for using a vertex processor as an arithmetic unit for the tessellation pipeline. Additional units are grayed out in the zoomed view.

4.7 Retessellation Results

In this section, the different sampling strategies presented in Section 3 are applied to combinations of displacement maps and base domain surfaces, and the results are compared using the measurement algorithms described in Section 3.3. The tessellation results were compared to an exact tessellation, as described in Section 3, with one vertex per displacement map point.

4.7.1 Used Sampling Tests

The base domain mesh used as a starting point for the tessellation process has great influence on the quality of the outcome. In particular the resolution must be taken into account when choosing the thresholds for the sampling algorithms. The algorithms were tested on meshes with 4, 25 and 100 vertices with different parameters to demonstrate the influence of the thresholds on the different algorithms. All thresholds are defined in the range from 0 to 1. The following tests were used:

Displacement Map Name	Size	Description
Ashby	346 × 452	Ashby Gap, Virginia, USA
Crater Lake	336 × 459	West half Crater Lake, Oregon, USA
Ozark	369 × 462	Ozark, Missouri, USA
Volker	512 × 456	Cylindrical scan of the head of Volker Blanz
Donut	512 × 512	Synthetic half-donut shaped map
Peaks	512 × 512	Synthetic flat map with narrow peaks

Table 4.1: Table of Displacement Maps used for tessellation tests.

- Surface Normal Variance Test(SNV), Section 3.1.1
- Local Area Average Height Test(LAAH), Section 3.1.2
- View Dependency Test(VD), Section 3.1.3
- Refinement Limit Test(RL), Section 3.1.4
- Decision Maps(DM), Section 3.2.3
- Curvature Maps(CM), Section 3.2.3.

The following combinations were used to perform the actual tests:

- Local Area Average Height Test, Refinement Limit Test and View Dependency
- Surface Normal Variance Test, Refinement Limit Test and View Dependency
- Local Area Average Height, Surface Normal Variance and Refinement Limit Test
- Decision Maps, Refinement Limit Test and View Dependency
- Curvature Maps, Refinement Limit Test and View Dependency.

The two negative response tests – the *View Dependency Test* and the *Refinement Limit Test* – make little sense to be used individually, since they only remove vertices.

4.7.2 Tessellation Results for the Ozark Displacement Map

The Ozark displacement map is a terrain height field with plenty of high frequency in the geometry and fine detail. An example rendering of a tessellated and displaced rectangular mesh is shown in Figure 4.11. On the right side the mesh is drawn with only the triangle

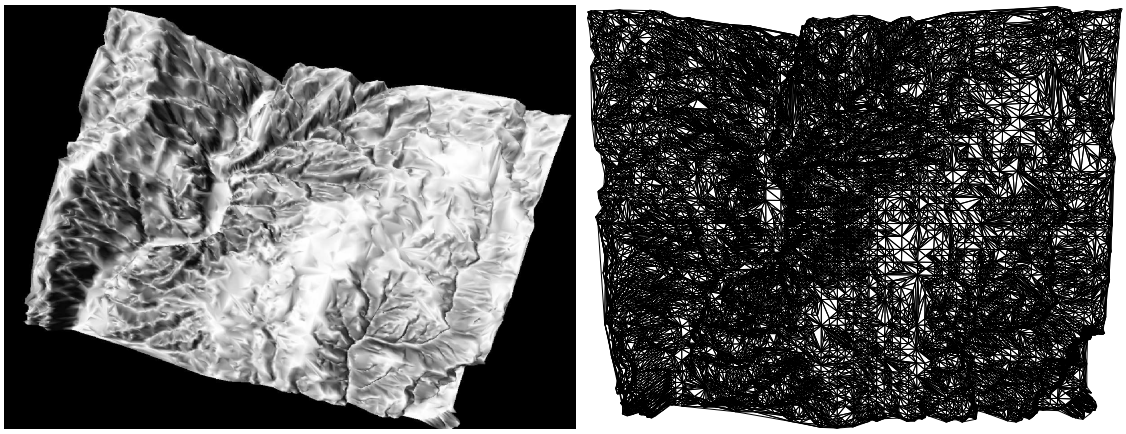


Figure 4.11: Rendering of the Ozark displacement map with the corresponding triangle mesh shown.

edges shown. Table 4.3 shows the tessellation results for the Ozark displacement map for different combinations of testing algorithms and base mesh triangulations. In this case the *Local Area Average Height Test* (LAAH) and the *Surface Normal Variance Test* (SNV) are listed. The *Refinement Limit Test* was always applied.

Already for a base mesh with as little as four vertices or two triangles very satisfying results can be obtained. Increasing the amount of triangles in the base mesh reduces the error further and more importantly reduces the number of iterations necessary. The tests were used alone and combined, to show that the combination will yield better results than expected in Chapter 3. As the displacement map contains many of small changes the *Surface Normal Variance Test* performs already very well, and the *Local Area Average Height Test* misses a lot of the fine detail due to its averaging nature. If the starting tessellation is very coarse as in the 2×2 case the combined result is tessellated far more and also better than with the individual ones, although the individual thresholds are equal. This is due to the fact that the SNV test is likely to miss changes on a too large scale. If the thresholds for the individual tests are lowered, the SNV test will produce substantially more triangles than the combination of the tests and still yield a higher error. The LAAH test produces satisfying results with a lower threshold but the recursion level has to be increased and the number of triangles created is much higher, too. Equal results can be seen with the 5×5 and 20×20 base meshes. For the 5×5 case the combined tessellation tests even result in a lower error although the triangle count is reduced. The combined tests obviously adapt better to the source geometry than the individual tests.

Thresholds		Vertices	Error in %			Volume Error
SNV	LAAH		Maximum	Mean	Mean Square	
Starting tessellation 2x2 vertices						
disabled	0.01	3626	8.4	0.35	0.9	2998
0.3	disabled	4625	7.95	0.22	0.55	1995
0.3	0.01	10837	1.3	0.07	0.1	1221
0.1	disabled	13578	3.4	0.08	0.11	1543
disabled	0.001	13124	1.45	0.07	0.1	1436
Starting tessellation 5x5 vertices						
disabled	0.01	12167	1.8	0.08	0.12	1758
0.3	disabled	23996	3.5	0.07	0.11	3467
0.3	0.01	23108	1.36	0.06	0.06	2324
Starting tessellation 20x20 vertices						
disabled	0.1	17303	1.33	0.07	0.1	2273
0.3	disabled	20031	1.46	0.07	0.1	2493
0.3	0.1	27015	0.9	0.06	0.08	2914

Table 4.3: Tessellation results for the Ozark displacement map applied to rectangular base meshes with 2×2, 5×5 and 20×20 vertices.

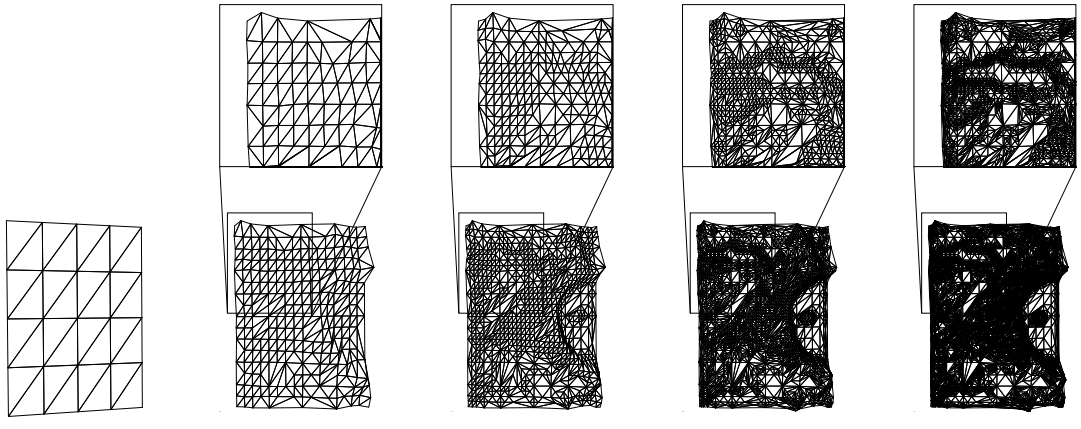


Figure 4.12: Different recursion levels generated while tessellating a rectangular triangle mesh for the Crater Lake displacement map. The upper left corner is zoomed in to show the adaptive tessellation surrounding a river valley.

Recursion Process

To illustrate the process of retessellating the base mesh, the triangulations at different iteration levels were calculated and are shown in Figure 4.12. The upper left region of the result is zoomed in, showing the course of a river with increasing level of detail. The river area demonstrates the advantages of the adaptive tessellation process, as the very fine triangulation is only generated at the river valley and not in the surrounding smoother areas.

Curvature Map Results

When precomputing the curvature maps the user is presented with not only one threshold, but with three input parameters that have a strong influence on the tessellation result. Apart from the curvature threshold controlling the initial entry in the curvature map, also the radius of the spreading circle and the maximum tessellation recursion level are necessary. In table 4.5 tessellation results for the Ozark displacement map are shown when using *Curvature Maps* for controlling the tessellation process. Additionally to the errors the percentage of tessellated triangles is given. The percentage depends on the maximum recursion level allowed and is a good indication for the effectiveness of a tessellation. The tests with a very low curvature threshold show a very high percentage of tessellation, very close to the theoretical maximum. While this results in very good quality, there is little or no adaptiveness of the tessellation and little gain from using an adaptive algorithm at all. Thresholds around 0.5 give very good results and show even better results than the

more complicated testing algorithms tested before in table 4.3. Increasing the threshold further, slowly decreases the triangle count and increases the error which is expectable due to the reduced triangle count.

4.8 Conclusions

A base domain surface with a displacement map applied to it can be rendered by tessellating the base surface triangles first, and displacing the new vertices. The new vertices are necessary, because the base domain surface resolution is usually far too low to allow a satisfying adaption to the shape contained in the map. Ideally, the new vertices should be added only where necessary to reduce the triangle count when rendering afterwards. The sampling tests presented in the previous Chapter can be used for controlling the adaptive tessellation process and lead to very satisfying results. A typical implementation of the OpenGL pipeline, as it is found in most graphics hardware architectures available today, lacks important features in the geometry processing stage to enable it to perform an adaptive tessellation of the base domain surface. With some simple additional units the graphics pipelines can be changed to very flexible tessellation units that can do more than just a simple adaptive tessellations for displacement mapped surfaces.

Threshold	Spread	Recursion	Vertices	Error in %		Triangles in %
				Maximum	Mean	
0.1	5	7	16460	0.56	0.066	99.41
0.2	5	7	15692	0.56	0.067	94.96
0.3	5	7	14336	0.69	0.069	86.77
0.4	3	7	12149	1.35	0.074	99.75
0.4	5	7	12636	0.91	0.073	76.46
0.4	7	7	13236	0.91	0.071	80.05
0.4	9	7	13672	0.69	0.071	82.64
0.5	3	7	10470	2.57	0.078	63.40
0.5	5	7	10907	0.91	0.076	65.95
0.5	7	7	11564	0.91	0.075	69.92
0.5	9	7	12235	0.91	0.073	73.93
0.6	3	7	8652	2.57	0.084	52.37
0.6	5	7	9081	1.93	0.83	54.92
0.6	7	7	9872	1.94	0.811	59.70
0.6	9	7	10567	1.23	0.076	63.84
0.4	3	8	43220	0.92	0.052	65.68
0.4	5	8	44965	0.88	0.052	68.32
0.4	7	8	47013	0.69	0.051	71.42
0.4	9	8	49637	0.64	0.051	75.42
0.5	3	8	36076	1.07	0.054	54.83
0.5	5	8	37889	0.9	0.054	57.56
0.5	7	8	40725	0.9	0.053	61.86
0.5	9	8	43272	0.69	0.052	65.73
0.6	3	8	29303	2.5	0.058	44.54
0.6	5	8	30943	2.1	0.058	47.02
0.6	7	8	34078	2.1	0.056	51.77
0.6	9	8	37016	0.9	0.054	56.22

Table 4.5: Tessellation results for the Ozark displacement map applied to a rectangular base mesh with 2×2 using *Curvature Maps*.

Chapter 5

Direct Rendering Algorithms

Although currently available graphics hardware mostly lacks support for inserting new triangles or vertices and sampling a texture map in the geometry processing stage rendering of a displacement mapped surface is still possible. In contrast to the rendering schemes from Chapter 4, the base domain surface is not modified according to the displacement map, but rather simple ray tracing or image based approaches are used. This became possible with the great increase in flexibility seen in the last few generations of commodity graphics hardware.

5.1 Prism Renderer

Most approaches to displacement mapping require that the geometry of a given base mesh can be modified, especially in the sense of adding more detail in the form of retessellated triangles. Currently available hardware, at which this algorithm is targeted, does not allow vertices to be added, once the geometry has been transferred to the graphics card. To work with this restriction this new algorithm [5] does not generate geometry on the card, but instead creates triangles that cover the area on the screen that could be affected by the displaced base mesh triangle. When the covering triangles are rasterized a per pixel calculation is performed to detect an intersection with the displaced surface. The number of covered triangles should be kept to a minimum to reduce the geometry transfer overhead. The bounding volume of the surface with a displacement map applied to it is given by a prism obtained by displacing the base triangle along the vertex normals to the maximum displacement height.

The area needed to cover the prism is given by its projection on the viewing plane. If no additional geometry is to be generated the base triangle can be projected parallely to the viewplane and expanded so that it contains the projected prism. Since the projection

can result in a polygon with more than six vertices as shown in Fig. 5.1, where the projection is a non convex polygon with seven vertices, the cover triangle is costly to compute when an optimal solution is desired with only few unnecessary pixels rendered. A far simpler and more robust approach is to use a quad instead as a cover for the prism. The cover triangles can be chosen arbitrarily as long as they cover the projected prism of

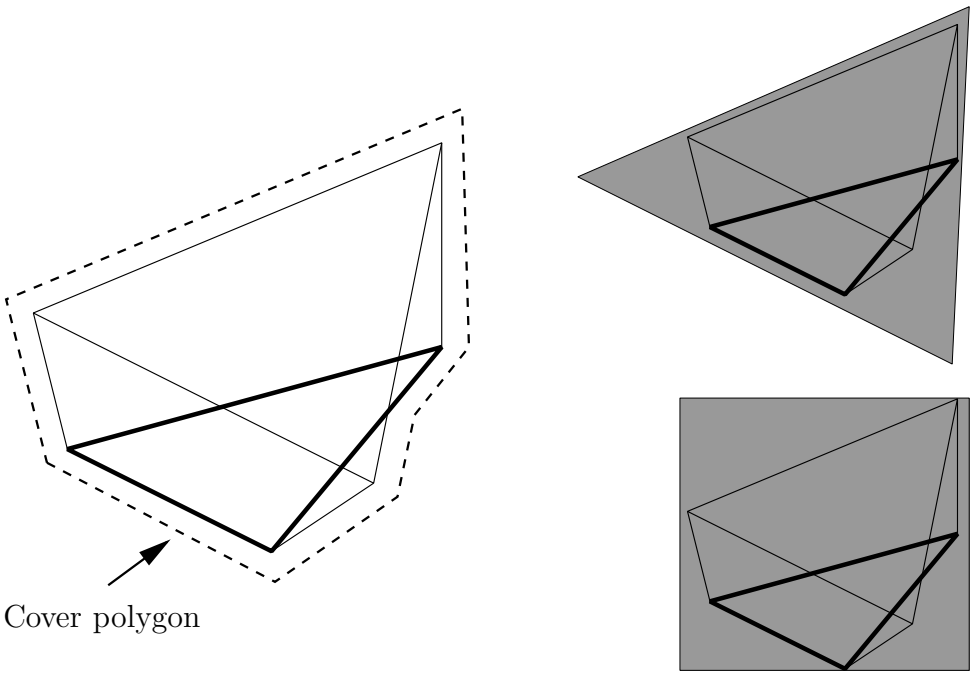


Figure 5.1: A possible projection of a prism on the viewing plane resulting in a polygon with seven vertices. Two possible covers calculated with a single triangle and a quad are also shown.

the displaced surface.

Since the cover triangles have to cover the extruded prism from all viewing positions, they either have to be recalculated whenever the viewpoint changes or have to be chosen in a manner such that they always cover the prism. The most straightforward way to do this is to render the prism completely by triangulating it. At each pixel of the prism's triangles a non-trivial intersection of the viewing ray with the prism has to be performed, placing a very high burden on the pixel shader pipeline. Since backfacing triangles can be culled, the amount of used pixels to be drawn is relatively limited. The resulting triangles are shown in Figure 5.2. The sides of the prism are quads and have to be split into two triangles, the bottom and top of the prism remain unchanged resulting in eight triangles to be rendered per base triangle. The displacement prism is displayed by rendering the triangles and casting rays into the prism from every interpolated pixel. The rays are cast in the viewing direction from each pixel position. To find out whether the ray intersects

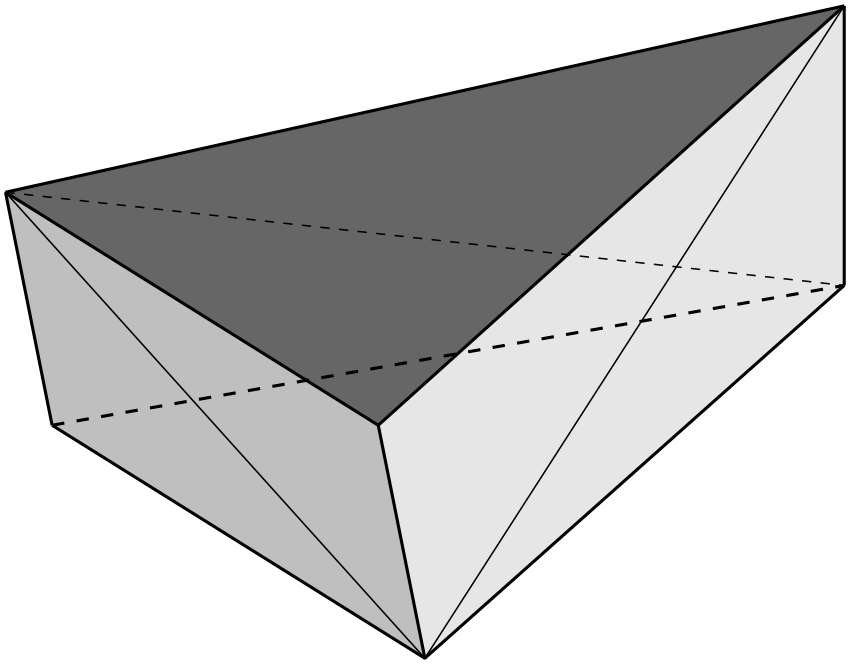


Figure 5.2: The prism with its resulting triangles used for rendering.

the displacement map, the height of the sampling position is compared to the height of the displacement map at the interpolated texture coordinate of the sampling position. The height ranges from zero at the base mesh level to one on the top of the prism. The 3D texture coordinates need to be interpolated inside the prism, along the viewing direction. The texture coordinates are local to the prism and a base transform has to be made at all vertex positions to obtain the viewing direction in local texture space. Given a triangle with vertices $V_i = (x_i, y_i, z_i)$ with normals N_i and texture coordinates $U_i = (u_i, v_i)$ for $i = 1, 2, 3$, the first step is to add a third coordinate defined by the height of the vertex in the prism:
$U_i' := (u_i, v_i, 0)$ for vertices of the base triangle and $U_i' := (u_i, v_i, 1)$ for vertices of the displaced triangle. To calculate the transformation for vertex V_1 for example, on the base triangle, a local base $B_{Texture}$ is defined with the texture directions e_1, e_2 along the triangle edges:

$$e_1 := U_2' - U_1'$$
$$e_2 := U_3' - U_1'$$
$$B_{Texture} := (e_1, e_2, 1)$$

In the same manner a local base B_{World} is defined with the world coordinates of the

vertices:

$$f_1 := V_2 - V_1$$
$$f_2 := V_3 - V_1$$
$$B_{World} := (f_1, f_2, N_1)$$

The basis transformation from B_{World} to $B_{Texture}$ can be used to transform the viewing direction at the vertex position V_1 to local texture space.

To avoid sampling outside of the prism, the exit point of the viewing ray has to be determined. In texture space the edges of the prism are not straightforward to detect and a 2D intersection calculation has to be performed. This can be overcome by defining a second local coordinate system which has its axes aligned with the prism edges. For this 3D coordinates are assigned to the vertices as shown in Figure 5.3. The respective name for the new coordinate for a vertex V_i is O_i. Then the viewing direction can be transformed in exactly the same manner to the local coordinate system defined by the edges between the O_i vectors:

$$g_1 := O_{(i+1)modulo3} - O_i$$
$$g_2 := O_{(i+2)modulo3} - O_i$$
$$B_{Local} := (g_1, g_2, 1).$$

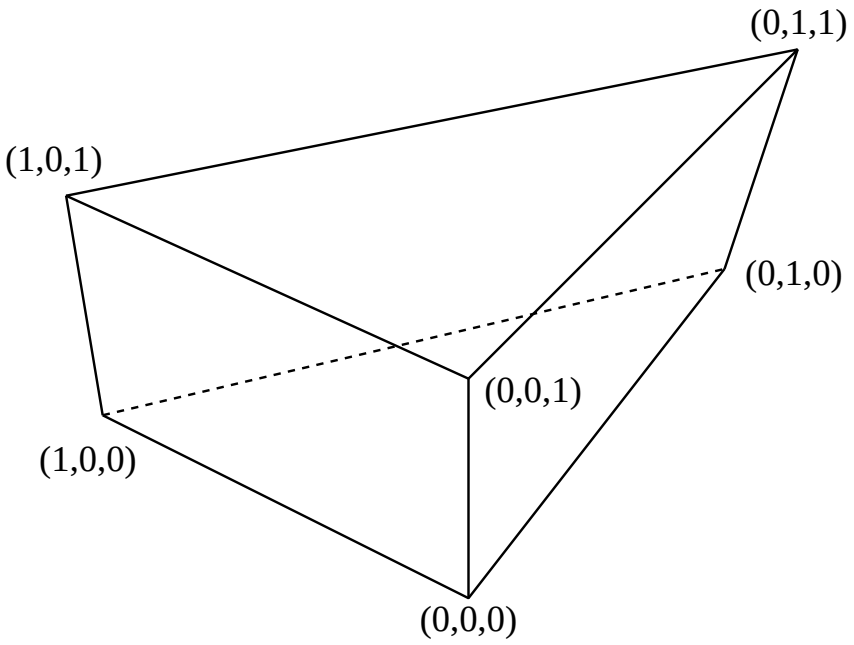


Figure 5.3: The vectors used to define the second local coordinate system for simpler calculation of the ray exit point.

In the following the local viewing direction in texture space is called $View_T$, and in the B_{Local} base representation $View_L$. It is assumed that the viewing direction changes linearly over the face of a prism triangle. The local viewing direction in both coordinate systems are assigned to 3D texture coordinates and used as inputs to the fragment shader pipeline in order to get linearly interpolated local viewing directions. The interpolated $View_L$ allows us to very easily calculate the distance to the backside of the prism from the given pixel position as it is either the difference of the vector coordinates to 0 or 1 depending on which side of the prism being rendered. With this Euclidean distance the sampling distance can be defined in a sensible way which is important as the number of samples that can be read in one pass is limited, and samples should be evenly distributed over the distance. An example of this algorithm is shown in Figure 5.5. In this case four samples are taken inside the prism. The height of the displacement map is also drawn for the vertical slice hit by the viewing ray. The height of the third sample which is equal to the third coordinate of its texture coordinate as explained earlier, is less than the displacement map value and thus a hit with the displaced surface is detected. To improve the accuracy of the intersection calculation, the sampled heights of the two consecutive points with the intersection inbetween them, are substracted from the interpolated heights of the viewing ray, as shown in Figure 5.4. Because of the intersection the sign of the two differences must

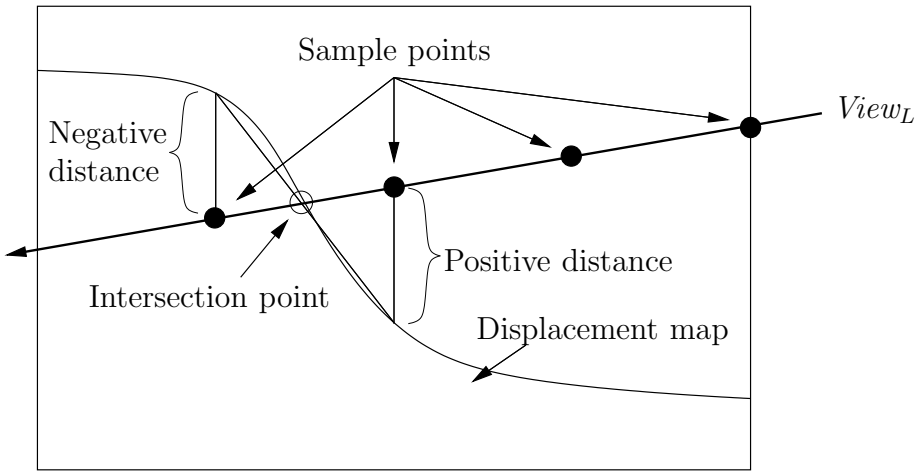


Figure 5.4: Improving the intersection point by linearly interpolating between the two consecutive sample points containing the zero crossing inbetween, shown on a slice of the displacement map.

differ and the zero-crossing of the linear connection can be calculated. If the displacement map is roughly linear between the two sample points, the new intersection at the zero-crossing is closer to the real intersection of the viewing ray and the displaced surface than the two sampled positions. Although the pixel position on the displaced surface is

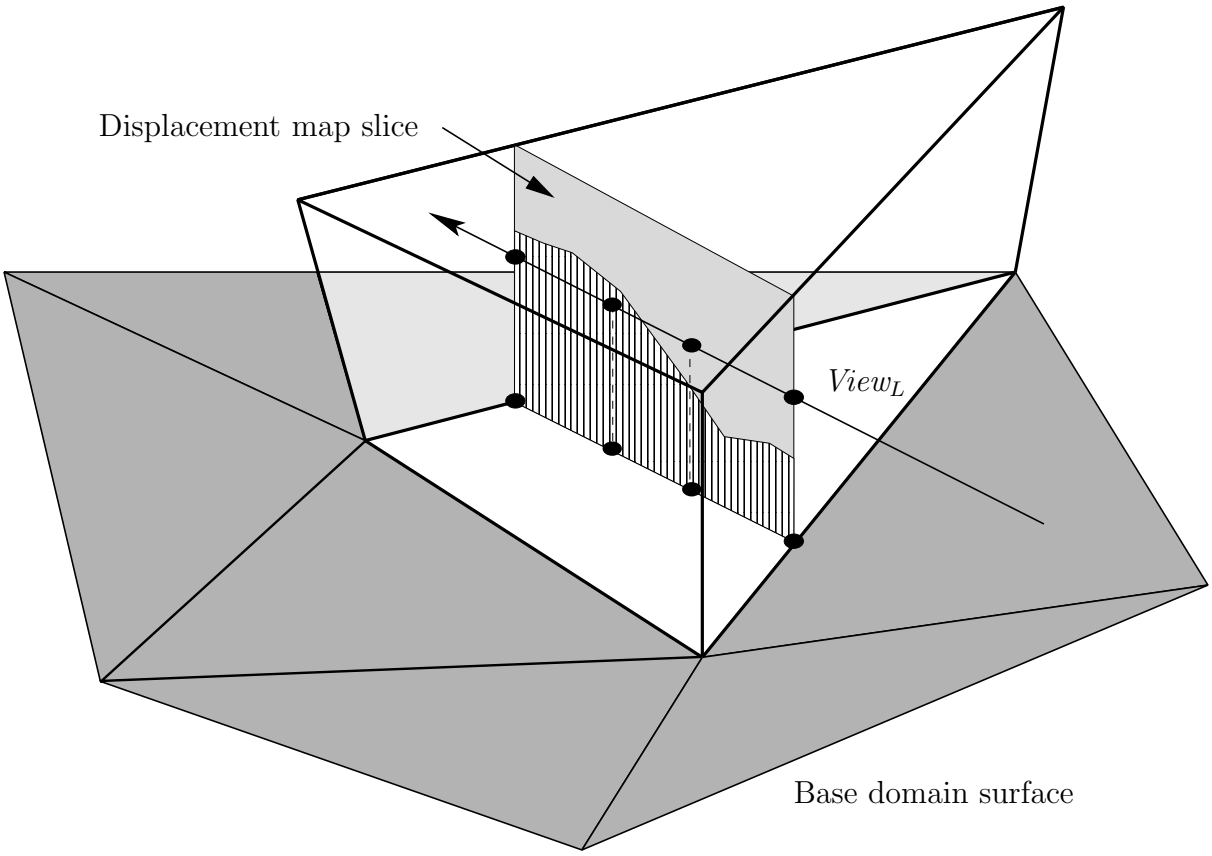


Figure 5.5: Sampling within the extruded prism with a slice of the displacement map shown.

now calculated, the normal at this position is still the interpolated normal of the base mesh triangle. It has to be perturbed for correct shading, in this case standard bump mapping using a precalculated bump map derived from the used displacement map is used. The bump map is obtained by convoluting the displacement map with a Sobel Operator [37] in both horizontal and vertical direction and storing the result together with the displacement map in one texture, with the displacement stored in the alpha channel.

The algorithm was implemented using OpenGL vertex and fragment programs and run on ATI r3xx and nVidia nv3x class cards. The performance was similar on both cards. The implementation also showed the limitations of the fragment shader accuracy. The edges of the prism are obviously very sensitive to rounding errors in the shading pipeline. On the prism edges the length of the sampling ray into the prism should be equal to zero, which was arithmetically not always the case, leading to a false miss.

5.2 Tetrahedral Renderer

Numerical problems can be reduced by simplifying the shape used that the rays are cast through. The prism is geometrically complex for performing intersection calculations. Obviously the prism can be split into three tetrahedrons as shown in Figure 5.6. The main difference in using tetrahedrons instead of the prism is that the texture space coordinates of the entry and exit point can be interpolated at the same time by the rasterization units. The sampling points between the entry and exit point can then be obtained by just linearly interpolating in between them. The tetrahedrons can be rendered using an adaption of the Projected Tetrahedra (PT) Algorithm by Shirley and Tuchman[43].

5.2.1 Mesh Construction

Using tetrahedrons requires the construction of a tetrahedral mesh from the base domain surface. It has to be ensured that neighboring tetrahedral edges are aligned in a consistent way to avoid aliasing effects between adjacent triangles. This can be achieved without knowledge of the connectivity in the tetrahedral mesh, by just setting up an enumeration of the vertices in the mesh that allows for an index comparison. The enumeration can be obtained from the vertex indices as they are usually given in an array. The algorithm iterates over all faces in the triangle mesh folding up a prism by displacing every vertex of the base triangle along the vertex normal direction. To globally adjust the amount of displacement, the normal can be multiplied with a user defined scalar. Every prism is then split into three tetrahedrons following the ordering scheme as schematically shown in Figure 5.6. The indices v_0, v_1, v_2 are assigned to the lower vertices and v_3, v_4, v_5 to the upper base vertices. Now every prism is tiled into the three tetrahedrons $T(v_0, v_1, v_2, v_5), T(v_0, v_1, v_4, v_5)$ and $T(v_0, v_3, v_4, v_5)$. An additional requirement is that $v_0 < v_1 < v_2$ with respect to the consistent numbering scheme of the mesh as noted before. Hence the algorithm simply works this way:

```
FOR_EVERY_TRIANGLE_FACE(f)

  IF(v0 > v1)
    SWAP(v0, v1)
    SWAP(v3, v4)

  IF(v0 > v2)
    SWAP(v0, v2)
    SWAP(v3, v5)
```

```
IF(v1 > v2)
  SWAP(v1, v2)
  SWAP(v4, v5)

CREATE_TETRA(v0, v1, v2, v5)
CREATE_TETRA(v0, v1, v4, v5)
CREATE_TETRA(v0, v3, v4, v5)
```

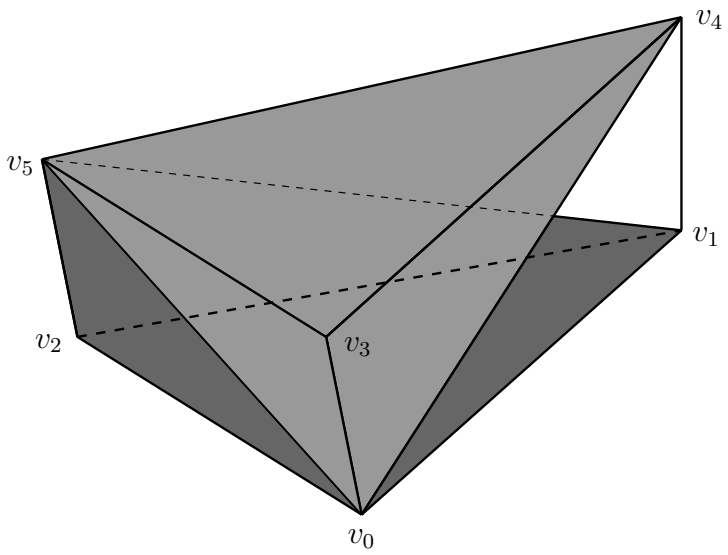


Figure 5.6: Subdivision of a prism into three tetrahedrons: $T(v_0, v_1, v_2, v_5), T(v_0, v_1, v_4, v_5), T(v_0, v_3, v_4, v_5)$

5.2.2 Rendering

To adapt the PT-algorithm to displacement mapping only a few modifications have to be applied. In contrast to the standard algorithm where each vertex needs color and opacity, each vertex is attributed with its respective tangent space consisting of normal, tangent and bi-normal, each a 3d-vector. The tangent and bi-normal are necessary for performing the bump map operation while shading the surface. Additionally two texture coordinates, one for the bump and displacement map, the other for a freely usable texture, are assigned to each vertex. Before the geometry is sent to the rendering pipeline a view-dependent preprocessing step has to be performed, where the tetrahedrons are decomposed into triangles according to the PT-algorithm. In Figure 5.7 the possible projections of tetrahedrons and the respective decompositions are shown. At point S in the

diagram the connecting edge between frontside and backside of the decomposed triangles is calculated. The backside vertex is also called the secondary vertex of all the triangles.

So far all the processing has to be done on the driver side by the host computer's CPU.

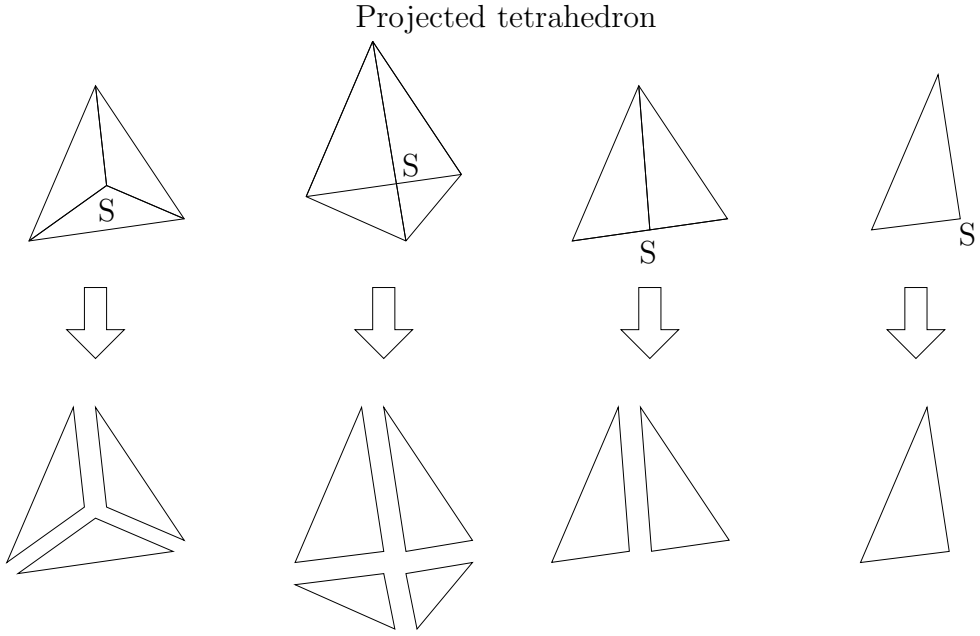


Figure 5.7: Possible decompositions of projected tetrahedrons into triangles.

Every triangle vertex (primary vertex) sent into the first stage is attributed with texture coordinates and tangent space vectors. Likewise the vertex on the backside (secondary vertex) of the decomposed tetrahedron is transfered as attribute including its texture coordinates and tangent space vectors. With these parameters the vertex shader computes homogeneous texture coordinates for the primary and secondary vertex. It also computes the modelview projection transformation of the vertices and finally transforms per vertex viewing and light direction into tangent space. In the second stage of this pipeline the pixel shader performs the intersection calculation between eye vector and the displacement map. To achieve this the pixel shader performs four lookups in the displacement map given by the interpolated texture coordinates of the primary and secondary vertex and two interpolated positions in between. The intersection between eyevector and displaced surface is then calculated by substraction of the sampled displacement value from the interpolated texture coordinates. A sign change indicates the interval where the eyevector hits the displaced surface. In case no surface was hit the pixel is removed. Otherwise the pixel will undergo a final shading step. Here, bump mapping was used to perturb the interpolated normal and Phong shading using the fragment shader stage. An overview of

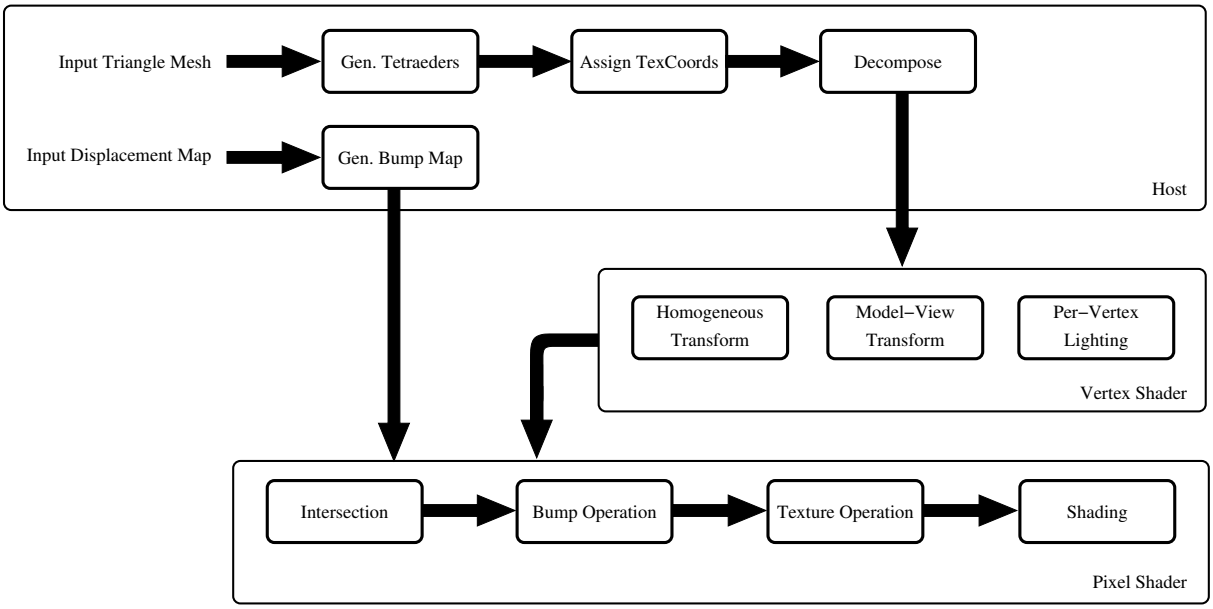


Figure 5.8: Overview of the pipeline for displacement map rendering using projected tetrahedrons.

the rendering process is shown in Figure 5.8.

5.2.3 Accuracy and Performance

The presented approach is very flexible since no assumptions are made about the shape of the displacement map or the base domain surface and apart from tetraeder generation no preprocessing is necessary. The only thing that has to be ensured is a reasonable base domain surface resolution to avoid sampling errors. Currently, the very limited amount of sample points presents a serious problem. In the example shown in Figure 5.9, only four samples inside a tetrahedron are taken. If the tetrahedrons are too large, features are easily missed and the error is untolerable. On the other hand the amount of triangles should be kept as low as possible, as the cost for every additional triangle is very high. The price of the flexibility is that the amount of additional geometry is considerable. For each base domain triangle a prism consisting of three tetrahedrons is created, with four triangle faces each, resulting in 12 triangles all together. For each of these 12 triangles' pixels a costly fragment shader program needs to be executed.

5.2.4 Adaptive Tetrahedrons

In order to improve accuracy without creating more triangles the shape of the prisms can be adapted to the rendered displacement map, for the cost of loosing some flexibility.

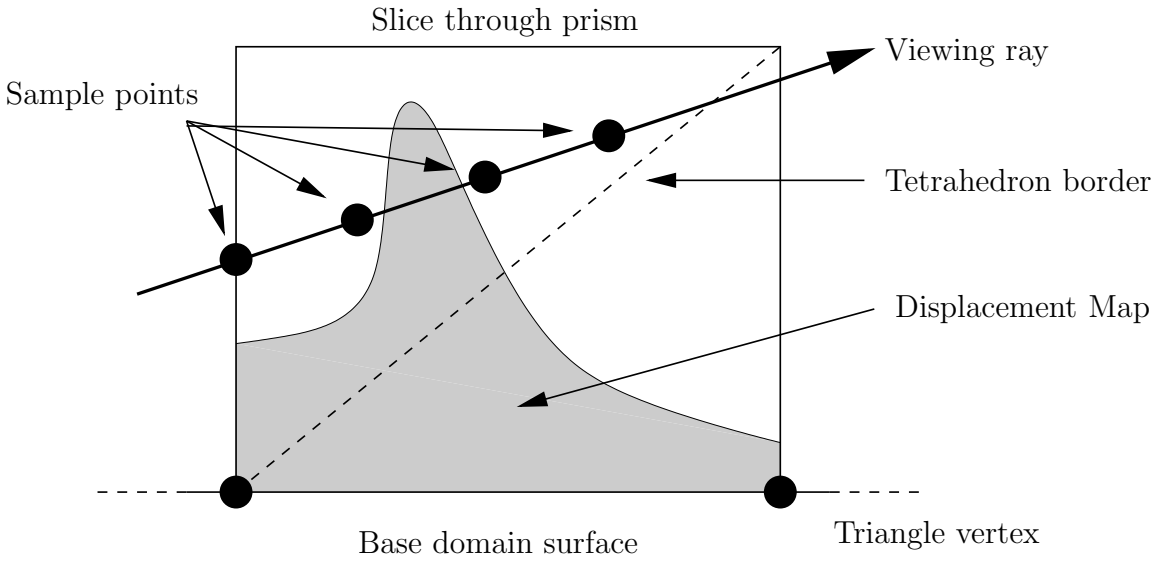


Figure 5.9: Sampling error caused by a too coarse base domain triangulation and the limited number of sample points.

Apart from pathological cases the difference in height in a local area of a displacement map is limited. Since the extruded prism always covers the full range of values that the displacement map may contain, mostly empty space is sampled, thus reducing effective accuracy. As a possibility to avoid sampling empty space we adapt the height of the extruded prism and the resulting tetrahedrons to the displacement map's shape. For a given base domain triangle the minimum and maximum height values of the covered displacement map need to be calculated. The exact values are difficult to obtain, since that involves rasterizing the texture coordinates of the triangle. A rough but already sufficient approximation can be provided by a mipmap which is commonly used for texture map filtering as described in section 2.3. In contrast to the common mipmapping algorithm where the different mipmap levels are calculated by averaging the samples of the lower levels, the minimum and maximum values are stored.

When generating the extruded prisms for the tetrahedrons the lower and upper bound are read from the previously created mipmap and the bottom and the top of the prism are adapted as shown in Figure 5.10. The effective distance that needs to be sampled is reduced to a fraction of the original prism. As a result, the base domain triangulation resolution can be reduced.

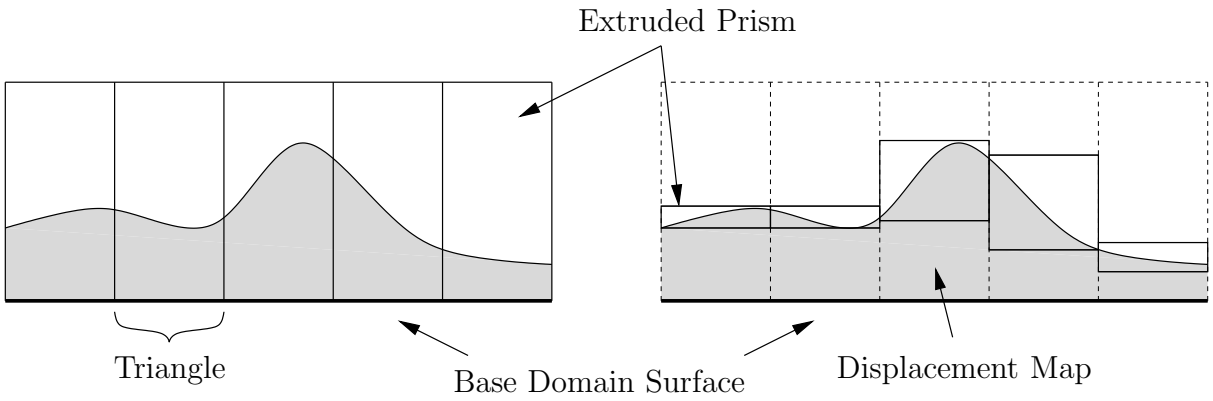


Figure 5.10: Improving the sampling by adapting the size of the extruded prisms. On the left the standard approach with the full sized prisms is shown and on the right the adapted prisms. As a result the distance that needs to be sampled is reduced and accuracy is improved.

5.3 Vertex Streams

In [45] Vlietinck demonstrates hardware accelerated displacement mapping through the use of vertex streams. Vertex streams were introduced by Microsoft with the *DirectX* API in version 8 [39] and allow the use of multiple vertex input streams in the *VP*. The algorithm uses a modified mipmapping scheme to store multiple levels of detail of the displacement map and feeds two consecutive levels of detail into the *VP* using the vertex stream API. The *VP* interpolates trilinearly between the two detail levels and displaces the vertices. As the *VP* cannot insert new vertices to create triangles, the triangles that may have to be inserted are transmitted as degenerate triangles and are discarded when no triangulation is necessary.

Although a lot of computation can be performed on the *VP* the algorithm still consumes CPU time and in particular bandwidth as the two levels of detail that are needed have

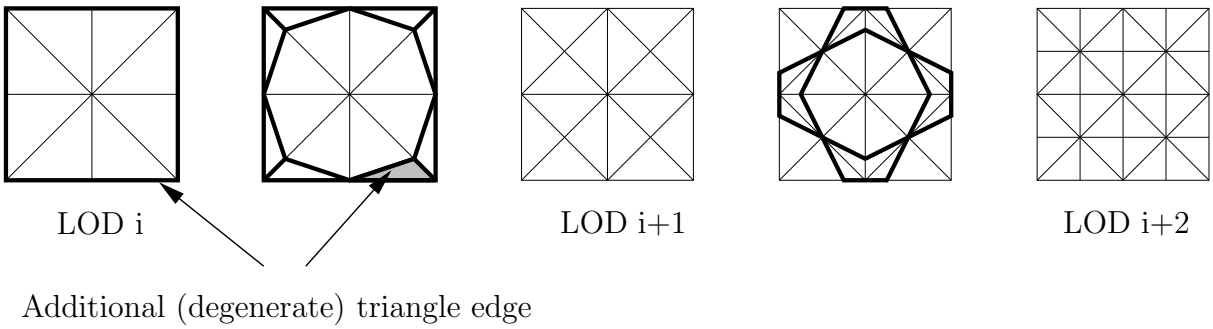


Figure 5.11: Three levels of detail of a tessellation and the trilinearly interpolated vertices between them. For levels i and $i+1$ the degenerate triangles are drawn with thicker outlines.

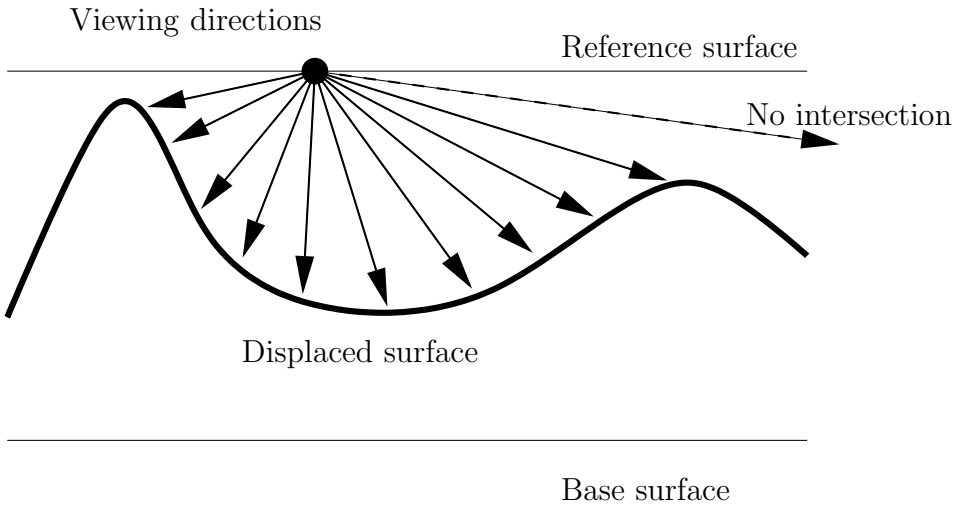


Figure 5.12: Precomputing viewing directions for the View-Dependent Displacement Maps. The ray on the right which doesn't intersect the surface is marked specially.

to be first identified and tessellated on the CPU and then sent to the graphics pipeline.

5.4 View-Dependent Displacement Maps

In [46] Wang et al. presented an alternative approach to directly rendering displaced surfaces using precomputation of possible viewing directions. Similarly as Kautz in [31] proceeded for rendering BRDFs using pixel shading pipelines, Wang precomputes the distance to the displaced surface for every possible viewing direction and texture coordinate on the surface, shown in figure 5.12. With a simple lookup on the generated precomputed so called *View-Dependent Displacement Maps* (VDM) the distance to the underlying surface can be obtained and also whether or not there is an intersection with the surface at all, which is necessary for correct rendering of the surface's silhouette. As the base domain surface's shape also influences the distance, the map has to be precomputed for different values of local curvature of the base surface. To reduce the high memory requirements a singular value decomposition is carried out, but still the size of possible source displacement maps is limited, as well as the amount of precomputed viewing directions. In the presented implementation it is limited to $32\times$ 16 viewing rays. Because the distance to the displaced surface is known for every position in the map, it can be not only calculated for the viewing direction, but also for the direction of a light source, enabling simple detection of self shadowing of the surface. In Figure 5.13, an example of a shadowed point is shown. In this case the distance of point P to the surface along the light direction (*Light-intersection distance*), is larger than the distance of the intersection

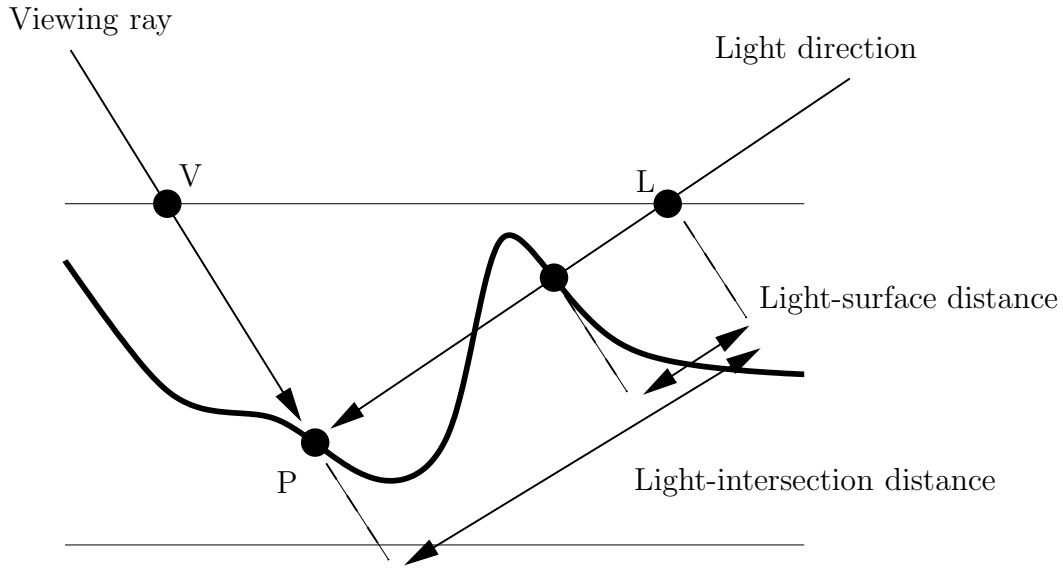


Figure 5.13: Determining whether point P is shadowed, by comparing the light-intersection distance from point P to L to the surface distance stored in the *VDM* at point L along the light direction.

point L of the light direction with the surface (*Light-surface distance*), which is stored in the *VDM*. The smaller *Light-surface distance* must be caused by an obstacle in direction of the light source, and point P is shadowed.

5.5 Conclusions

Efficient utilization of the available resources in available graphics hardware allows for approximations of displacement mapping by either a form of ray casting, precomputation of possible distances or linear interpolation between two levels of detail and the use of vertex streams. When used with an appropriate base domain triangulation the extruded prism perform very well without the need for any kind of precomputation, thus allowing for real time animations through the use of multiple textures or even streaming video data. Due to the limited memory bandwidth available, the sampling is subject to aliasing, but with increasing memory performance of upcoming graphics architectures this is likely to change. What cannot be neglected though is the great amount of additional geometry, that needs to be transformed and rasterized, although if more samples can be taken per ray in the tetrahedrons, the base triangles can be enlarged und thus the base domain surface triangle count can be reduced.

The precomputed distance maps are less subject to these sampling artefacts, and have distinct features like self-shadowing, which is not possible with any other algorithm in

a direct fashion so far. The fair amount of preprocessing and limited size for the input displacement map, as well as the low resolution of viewing directions are disadvantages.

Chapter 6

Applications and Examples

Apart from the obvious application of adding more surface detail to a otherwise flat or bump-mapped surface, displacement maps can be used for solving other rendering tasks. As described in Section 2.2, almost anything that approximates a surface and can be sampled, is usable as a source for a displacement map. Otherwise hard to render surfaces like NURBS or procedural height fields, can be easily rendered when converted into a displacement map. As a more tangible example, displacement maps as a form of geometry compression are presented.

6.1 Geometry Compression

As the displacement map is usually only a gray-scale image it can be processed as such, in particular the highly sophisticated image compression algorithms can be applied to it when storing the map. Geometry compression algorithms for triangular meshes are equally sophisticated and often very complicated and require long computation times, when compressing. If some predicates are met, displacement maps can be used as a form of geometry compression. The shape of the source surface should in some way resemble a height field, at least partially. Shapes like a head with bent hair on it are unlikely to produce good compression results, because the resulting base domain surface will be almost as complicated as the source surface was. As an example the laser range scan of a human head[1] was chosen. The range scan contains 512×456 sample points. When the mesh is fully tessellated it contains 233 thousand vertices. Typical compression rates for triangular meshes range from 15 to 20 bits per vertex, resulting in roughly 400 to 600 kilobytes memory for storing the mesh [26]. Using image compression like PNG [24]

[1]The model was obtained by using a CyberWare [22] Scanner and was kindly contributed by Volker Blanz.

the height field information can be stored within 40 kilobytes. The base domain surface mesh does not need to be explicitly stored, only the radius and height of the cylinder is necessary in this case. Even though the potential compression rates are very competitive, displacement maps cannot be used as a general geometry compressing facility, since the achievable rate strongly depends on the topology of the input geometry.

Chapter 7

Conclusions

This thesis discussed various aspects of rendering displacement mapped surfaces, especially by rendering using adaptive tessellation. The main focus was to explore new techniques suitable for hardware implementation in order to reduce the bandwidth strain on the system bus by moving the tessellation process onto the graphics subsystem.

A number of sampling techniques that can be implememented in custom hardware in a straightforward way have been developed and analyzed using a sample implementation of an adaptive tessellation pipeline. With only minor user interaction or conservatively predefined input parameters the sampling schemes produce adaptive tessellations with very low error measures.

The sampling techniques range from pre-processed curvature measurements to the comparison of perturbed surface normals and summed-area tables. All implementations result in roughly similar tessellation quality with varying complexity of implementation, though. Depending on the available hardware resources of the targeted hardware platform an appropriate sampling scheme may be selected.

Adaptively tessellating a triangle mesh – especially in a hardware context – requires special treatment of neighbouring triangles to avoid rendering artefacts caused by T-vertices or badly shaped triangles. These problems were adressed by performing adaptive tessellation based on local information of the edges of the triangles.

To enable currently available commodity graphics hardware to render displacement mapped surfaces, a rendering algorithm is presented. This novel algorithm does not depend on adaptive tessellation or special features in the rendering pipeline, and employs a modified form of raycasting to sample the height field along a viewing ray. Sampling is very sparse due to limited memory bandwith, but this will be accommodated as soon as forthcoming graphics cards with more powerful graphics pipelines emerge.

Bibliography

Author's list of publications

[1] M. Boo, M. Amor, M. Doggett, J. Hirche, and W. Straßer. Hardware support for adaptive subdivision surface rendering. In *Proc. of Eurographics/SIGGRAPH workshop on graphics Hardware 2001*, 2001.

[2] M. Doggett and J. Hirche. Adaptive view dependent tessellation of displacement maps. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 59–66, August 2000.

[3] J. Hirche and A. Ehlert. Curvature-driven sampling of displacement maps. Presented at ACM SIGGRAPH 2002 as a Technical Sketch, July 2002.

[4] J. Hirche and A. Ehlert. Curvature maps. Technical Report WSI-2003-11, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, September 2002.

[5] J. Hirche, A. Ehlert, S. Guthe, and M. Doggett. Hardware accelerated per-pixel displacement mapping. *submittet to Computer Graphics International 2004*, 2003.

[6] M. Amor, M. Boo, M. Doggett, J. Hirche, and W. Straßer. A meshing scheme for memory efficient adaptive rendering of subdivision surfaces. Technical Report WSI-2000-21, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, 2000.

[7] M. Doggett and J. Hirche. Adaptive view dependent tessellation of displacement maps. In *Proc. of Eurographics/SIGGRAPH workshop on graphics hardware 2000*, pages 59–66, 2000.

[8] M. Doggett and J. Hirche. Displacement mapping rendering hardware using adaptive tessellation. Presented at ACM SIGGRAPH 2000 as a Technical Sketch, July 2000.

[9] U. Kanus, G. Wetekam, J. Hirche, and M. Meißner. VIZARDII: An FPGA-based Interactive Volume Rendering System. In *Field-Programmable Logic and Applications*, Proc. of the 12th International Conference on Field-Programmable Logic, pages 1114–1117, September 2002.

[10] A. Kugler and J. Hirche. Texturing ASIC specification. Technical Report WSI-98-16, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, 1998.

[11] A. Kugler and J. Hirche. Trirast: A rendering processor architecture for advanced texture shading. Technical Report WSI-98-17, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, 1998.

[12] M. Meißner, M. Doggett, U. Kanus, and J. Hirche. Accelerating volume rendering using an on-chip sram occupancy map. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 2001.

[13] M. Meißner, M. Doggett, U. Kanus, and J. Hirche. Efficient space leaping for ray casting architectures. In *Proceedings of the second Workshop on Volume Graphics*, 2001.

[14] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. VIZARDII: A reconfigurable interactive volume rendering system. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 137–146, September 2002.

[15] M. Meißner, G. Wetekam, J. Hirche, and U. Kanus. Accomodating pipeline latency for high clock frequency image order volume rendering architectures. Technical Report WSI-2001-13, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, 2001.

Literature

[16] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. *IEEE Visualization 2001*, pages 21–28, October 2001. ISBN 0-7803-7200-x.

[17] J. F. Blinn. Simulation of wrinkled surfaces. *Computer Graphics*, 12(3):286–292, 1978.

[18] J. F. Blinn and M. E. Newell. Texture and Reflection in Computer Generated Images. *Computer Graphics*, 10(3), 1976.

[19] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, June 1998.

[20] R.L. Cook. Shade Trees. *Computer Graphics, Proc. of SIGGRAPH 84)*, 18(3):223–231, July 1984. Held in Minneapolis, Minnesota.

[21] F.C. Crow. Summed-Area Tables for Texture Mapping. *Computer Graphics, Proc. of SIGGRAPH 84)*, 18(3):207–212, July 1984. Held in Minneapolis, Minnesota.

[22] Cyberware. `http://www.cyberware.com`.

[23] J. Encarnaçao, W. Straßer, and R. Klein. *Graphische Datenverarbeitung 2*. Oldenburg Verlag, 1997.

[24] International Organization for Standardization. ISO/IEC 15948: Portable network graphics (PNG): Functional specification, 2003.

[25] M. Garland and P.S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Sept. 1995.

[26] S. Gumhold and R. Amjoun. Higher order prediction for geometry compression. pages 59–66, 2003.

[27] S. Gumhold and T. Hüttner. Multiresolution rendering with displacement mapping. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 55–66, August 1999.

[28] P.S. Heckbert. Survey of texture mapping. In *Proceedings of Graphics Interface 1986*, pages 207–212, 1986.

[29] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. *Computer Graphics*, 26(2):71–78, 1992.

[30] Avid Technology Inc. *Softimage*. `http://www.softimage.com`.

[31] J. Kautz. Hardware rendering with bidirectional reflectances. Technical Report TR-99-02, Dept. Comp. Sci., U. of Waterloo, 1999.

[32] M. Kilgard. A practical and robust bump-mapping technique for today's gpus. GDC 2000.

[33] A. Kugler. *Pixel Shading Pipelines and Display Hardware*. PhD thesis, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, 1999.

[34] S. Kumar and D. Manocha. Efficient rendering of trimmed NURBS surfaces. *Computer-aided Design*, 27(7):509–521, 1995.

[35] A. Lee, H. Moreton, and H. Hoppe. Displaced Subdivision Surfaces. In *Computer Graphics, Proc. of SIGGRAPH 2000*. ACM, 2000.

[36] Ch. Loop. Smooth subdivision surfaces based on triangles, 1987. Master's thesis, University of Utah, Institute of Mathematics.

[37] D. Marr and E. Hildreth. Theory of edge detection. In *Proceedings Royal Society London*, volume B 207, pages 128–217, 1980.

[38] Matrox. http://www.matrox.com/mga/products/parhelia512.

[39] Microsoft. *Direct X8 API*. `http://www.microsoft.com/directx`.

[40] B.-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[41] A.G. Schilling, G. Knittel, and W. Straßer. Texram: A smart memory for texturing. *IEEE Computer Graphics & Applications*, 16(3):32–41, May 1996.

[42] M. Segal and K. Akeley. The OpenGL graphics interface. Technical report, Silicon Graphics Computer Systems, Mountain View, CA,USA, 1993.

[43] P. Shirley and A. Tuchmann. A polygonal approximation to direct scalar volume rendering. In *ACM Computer Graphics*, volume 24, pages 63–70, 1990.

[44] Alias Systems. *Maya*. `http://www.alias.com`.

[45] J. Vlietinck. *Trilinear displacement mapping of a flat surface with a v1.1 vertex shader*, 2003. `http://users.belgacom.net/xvox`.

[46] L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum. View-dependent displacement mapping. In *Computer Graphics, Proc. of SIGGRAPH 2003*. ACM, 2003.

[47] L. Williams. Pyramidal Parametrics. In *Computer Graphics, Proc. of SIGGRAPH 1983*. ACM, 1983.