Multiresolution Rendering With Displacement Mapping

Stefan Gumhold*, Tobias Hüttner†

WSI/GRIS University of Tübingen

Abstract

In this paper, we present for the first time an approach for hardware accelerated displacement mapping. The displaced surface is generated from a 2D displacement map by remeshing a coarse triangle mesh according to the screen projection of the surface The remeshing algorithm is implemented in hardware. Filtered access to the displacement map makes our approach competitive with available view dependent multiresolution techniques. The advantage of displacement mapping is the compact representation. A displacement mapped surface consumes together with all filter levels only a fraction of the storage space needed for a hardware compatible representation of an equivalent triangle mesh

A possible design of the displacement mapping rendering pipeline is proposed. Previously described hardware components are used as often as possible. Our approach can be smoothly integrated into all available graphics application programming interfaces. Most existing graphics applications can be extended to the new feature with marginal effort.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Raster display devices I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and framebuffer operations, Display algorithms, Viewing algorithms I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: hardware, multiresolution, displacement mapping

1 Introduction

Our approach for displacement mapping extends currently available 3D raster graphics hardware in order to allow for real time applications. The realism of rastered surfaces depends greatly on the detail of the surface and the type of illumination. Available graphics hardware is based on the rasterization of triangles.

As the number of rendered triangles is limited by several factors, additional detail can be added with the help of texture mapping. Textures are mapped onto triangles via a two dimensional parametrization with rectangular domain. The coordinate axes in the domain are commonly denoted with u and v. Each triangle vertex is supplied with texture coordinates. The texture coordinates inside the triangle are linearly interpolated in world coordinates As the triangle is rastered in screen coordinates the texture coordinates must be perspectively corrected [12]. To avoid aliasing the access to texture maps must be filtered. The commonly realized approach is MIPmap filtering [28], which is not optimal as a screen pixel can be mapped through the inverse perspective projection to long and thin footprints in the texture domain. Therefore, adaptive filtering techniques have been proposed which sample this footprint more precisely [5, 11, 27].

In available hardware accelerators illumination is mostly based on Gouraud shading. Here the illumination is evaluated at the triangle corners. The resulting color values are just interpolated over the triangle. Although several architectures for Phong shading have been proposed [25, 1, 6, 19, 3], only some specialized raster hardware support it, as the surface normal must be interpolated and renormalized for each processed pixel.

Although the initial idea dates far back on Blinn [2], hardware architectures for bump mapping have been proposed only more recently [8, 24, 18]. Bump mapping is a further possibility to add detail to a surface by varying not the diffuse color attribute as in case of texture mapping but the normal vector which is used for illumination. This allows to simulate more realistic illumination of a surface with small bumps, that vary the surface normal. To realize bump mapping the interpolated surface normal vector is changed by adding a deflection vector and probably by renormalizing the result. The surface normal is interpolated from the normals at the triangle corners as in the case of Phong shading. The deflection vector is defined by an access to the bump map which is parametrized in the same way as a texture map. Even the same coordinates might be used The difficulty of the bump mapping approach is that the deflection vector must be defined in a local coordinate system which needs to be computed during rasterization at least partially. Blinn [2] proposes to compute the deflection vector from a displacement map, which stores for each parameter position the displacement of the surface in direction of the interpolated normal. For this the derivatives of the displacement map in u and v direction must be approximated. Blinn simplifies the formula for the deflection vector further by assuming that displacements are small. The approach of Kugler [18] defines the deflection of the current normal by two rotation angles which are stored in the bump map. This approach does not use an approximation and is therefore also suitable for large displacements. One general drawback of bump mapping is that it only applies to small variations of the surface. The lack of the actual geometrical displacement of the surface produces two kinds of artifacts. Firstly, the silhouette looks still as in the undisplaced case and secondly the geometrical displacement causes occlusions and movement of surface points which can shift intersections of eye rays with the displaced surface significantly. Finally, we want to mention the difficulties in filtering normal fields as discussed by Fournier [10].

Most problems of bump mapping can be solved by actually performing the geometrical displacement of the base surface. Three approaches have been proposed for displacement mapping so far. Cook [4] describes a shader which can vary the pixel position on the screen. This feature can be used for displacement mapping, but no details are given. The problem of moving pixels around is that

*Email gumhold@uni-tuebingen de

†Email thuettne@gris uni-tuebingen de

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee
1999 Eurographics LosAngeles CA USA
Copyright ACM 1999 1-58113-170-4/99/08.. $5.00

the surface will get cracks if the displacement varies too much and that the displacement map cannot be sampled appropriately in view direction which is extremely important to represent the silhouette of the rendered surface. The second approach is to resample the displacement in volumetric textures [15, 9, 22]. The storage space for this approach is increased by one dimension, which restricts the applicability to either repetitive textures or small displacements. A second problem is that illumination can only be handled by either increasing the computational costs or the consumed storage space immensely. Finally, the approach of Patterson[23] is based on ray-casting and therefore not compatible with common graphics hardware.

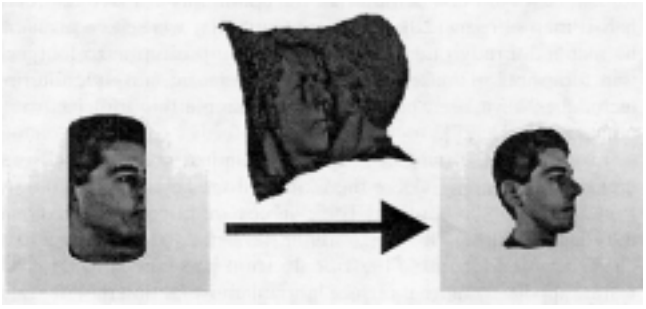

Figure 1: Principle of displacement mapping.

Up to now all approaches to displacement mapping assumed that the displacements are rather small and that the displacement map is basically repetitive. This often results in simple examples, which primarily look like a sphere with some bumps on it. In this paper we want to demonstrate the possibilities provided by displacement mapping. Figure 1 shows a displacement map obtained with a cyberware scanner. The displacement is mapped onto a cylinder consisting of thirty two triangles. The advantage of the displacement map representation is the small size of the data set and the regular arrangement of the data, what makes hardware access easy. The geometry of the example surface in figure 1 is represented in $512^2 \times 2$ bytes for the displacement map plus 34×12 bytes for the cylinder mesh which is specified in a triangle strip. Altogether this sums up to 512.2 KB which is significantly less than the 3 MB, which are consumed by an equivalent triangle mesh in triangle strip representation, when each coordinate is represented with only two bytes. We present an approach for displacement mapping which remeshes the coarse mesh view dependently with triangles of pixel size. Our remeshing algorithm might produce too many fine triangles as it does only depend on the screen resolution and not on the surface variation, but the triangles are produced directly on the graphics board and can therefore be processed much faster.

This example shows that our displacement mapping hardware can be used for view dependent multiresolution rendering. In this area two basic techniques have been available so far. Discrete multiresolution models as described in [13, 17, 21, 29] are most flexible and provide best error control. But even in a storage space efficient representation as described in [26] the connectivity and hierarchy information increases the size of the view dependent multiresolution model to several times the size of the original triangle mesh. The other multiresolution techniques are based on wavelet representations of the surface as described in [20]. Wavelet based surface representations are very compact. We do not know of a view dependent rendering algorithm, what is surely feasible. But it is not clear if such an algorithm is suitable for a hardware implementation.

In displacement mapping the multiresolution representation is stored in filter levels similar to MIPmaps. The screen space error produced with this approach depends on the quality of the filter levels and on the access strategy. We propose simple solutions to both problems in section 4. But we cannot control the screen space error completely. This is at the moment the only drawback we see compared to other multiresolution representations. We will work on this in future and believe that displacement is a beautiful compromise between control of the screen projected error on the one hand and compactness of the representation and simplicity of accessing the surface on the other hand.

In what follows we describe a possible design for a displacement mapping hardware. The design is meant as a proposal for a redesign of existing graphics accelerators.

2 Displacement Mapping

In this section we give an overview of the displacement mapping hardware and describe a complete rendering pipeline including displacement mapping. But first we define the notion of a displaced surface in a mathematical framework.

2.1 Principles of Displacement Mapping

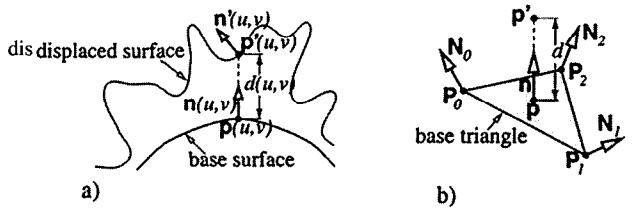

Figure 2: a) Base and displaced surface. b) Base triangle.

Figure 2 a) illustrates the basic components of a displaced surface: a base surface $\mathbf{p}(u,v)$ which is parametrized over a two dimensional domain, a normal field $\mathbf{n}(u,v)$ and a displacement scale field $d(u,v)$, both defined over the same domain as the base surface. Often the normal field is computed from the normalized cross product between the partial derivatives of $\mathbf{p}$ in u and v direction. But this is not mandatory for the definition of the displaced surface $\mathbf{p}'$

$$\mathbf{p}'(u,v) \stackrel{\text{def}}{=} \mathbf{p}(u,v) + d(u,v) \cdot \mathbf{n}(u,v). \qquad (1)$$

For application in a rasterization hardware the base surface is discretized and approximated by a triangulation. The coarse triangles which approximate the base surface are called *base triangles*. Figure 2 b) shows such a base triangle. Each base triangle vertex contains a position $\mathbf{P}_i$, a normal vector $\mathbf{N}_i$, a pair $\mathbf{T}_i \stackrel{\text{def}}{=} (u_i, v_i)$ of texture coordinates and a color vector $\mathbf{C}_i \stackrel{\text{def}}{=} (r_i, g_i, b_i, \alpha_i)$. All the vertex data denoted by $\mathbf{V}_i$ is linearly interpolated to points inside the triangle. The normals are additionally normalized after interpolation or could be interpolated via the angle as described in [19]. Interpolated vertices and vertex data is denoted with small letters in contrast to base triangle vertices which are denoted with capital letters. During the interpolation the vertex positions define the base surface, the vertex normals the normal field and the texture coordinates the inverse of a two dimensional parametrization. With the help of the parametrization the displacement scale field can be realized with a two dimensional array of scale values which are multiplied to the interpolated normal and in this way define the displacement. This array is called *displacement map*. The values of the displacement map are 16 bit integers. They are scaled to the range between the minimal and maximal displacement in world coordinates, which is both stored with the displacement map.

For a correct illumination of the displaced surface the knowledge of the surface normal $\mathbf{n}'(u,v)$ is very important. The normals are

computed with one of the available bump mapping approaches. We favorite the approach of Kugler [18] as it is not based on an approximation, which is especially important for us as we want to support large displacements.

We do not want to change the user graphics application programming interface (API). Only the functionality for specifying the displacement map is added. Everything else stays the same. Therefore, base triangles arrive at the displacement mapping unit in an arbitrary order and nothing about the adjacent base triangles is known. In order to avoid cracks at common edges of adjacent base triangles the displacement along the edges must be applied consistently. For this two conditions must be fulfilled. Firstly, the computation of the displaced surfaces along the triangle edges may only depend on the vertex data of the two incident vertices and not on the data of the third vertex. This condition may never be neglected. It imposes a severe constraint as it influences the design of the remeshing algorithm and the access to the filter levels. We call this constraint the *edge constraint*. The second condition is that the common vertices of adjacent triangles must contain the same vertex positions and vertex normals. The second condition must be fulfilled by the programmer and can easily be ensured by the use of vertex arrays.

2.2 Displacement Mapping Hardware

In this section we describe a complete rendering pipeline which is oriented at the commonly used *OpenGL* rendering pipeline and similar components are used. The new rendering pipeline is shown in figure 3. The view dependent remeshing and the displacement calculations take place in a special unit, the *Displacement Unit*. It is located between the *Primitive Assembly* stage and the *Perspective Transformation* stage of the standard *OpenGL* pipeline, see Figure 3. Parts of the *OpenGL* pipeline are reused, for example the illumination unit and the perspective transformation.

In order to integrated the displacement unit into the *OpenGL* pipeline, two interface units are inserted before and after the displacement unit. The *Controller Unit* directs the to be displaced base triangles into the displacement unit. The remaining triangles are directly transferred to the *FIFO/Triangle Buffer Unit*, which synchronizes triangles from the controller with triangles generated in the displacement unit. The triangle buffer is locked by either the displacement unit or the controller unit and unlocked after the triangles have been transmitted. The buffer also allows to cover latencies in the displacement pipeline.

The displacement unit is organized as a pipeline and consists of five stages. The first stage is the setup for the base triangles, where the vertex data is prepared for the remeshing algorithm. The input to the triangle setup consists of three vertices, each of them containing the vertex position $\mathbf{P}$, the vertex normal $\mathbf{N}$, the texture coordinates $\mathbf{T}$ and the color $\mathbf{C}$. For correct illumination computations the positions and normals must be given in the same coordinate system as the light sources and the view point. We call this the *world coordinate system* and choose it with the origin in the view point, the z direction in view direction and the x and y directions parallel to the corresponding viewport coordinates. In the setup stage the vertex positions are transformed to screen space and the base triangle clipping is performed. The vertex data is handed on to the next stage, the remeshing stage, which performs a sweep-line algorithm in order to produce quadrilaterals and triangles of pixel size. The remeshed vertices are computed in world coordinates and denoted with lower case letters in contrast to the upper case notation of the original vertices. These two units implement the remeshing algorithm, which is the core contribution of this paper and is comprehensively described in section 3.

All vertices which are newly produced by the remeshing stage are piped through the last three stages of the displacement unit. First the bump and the displacement mapping is performed in parallel. The bump mapping unit generates the surface normal $\mathbf{n}'$ of the displaced surface in world coordinates. Any additional data needed by a specific bump mapping technique is also interpolated during the remeshing and handed on to the bump mapping unit. The displacement mapping unit computes the displacement scale by a filtered displacement map lookup and generates the new surface point via formula 1. Next the illumination unit performs Phong shading. Finally, the vertex contains the illuminated color and the displaced position and is ready for stripping.

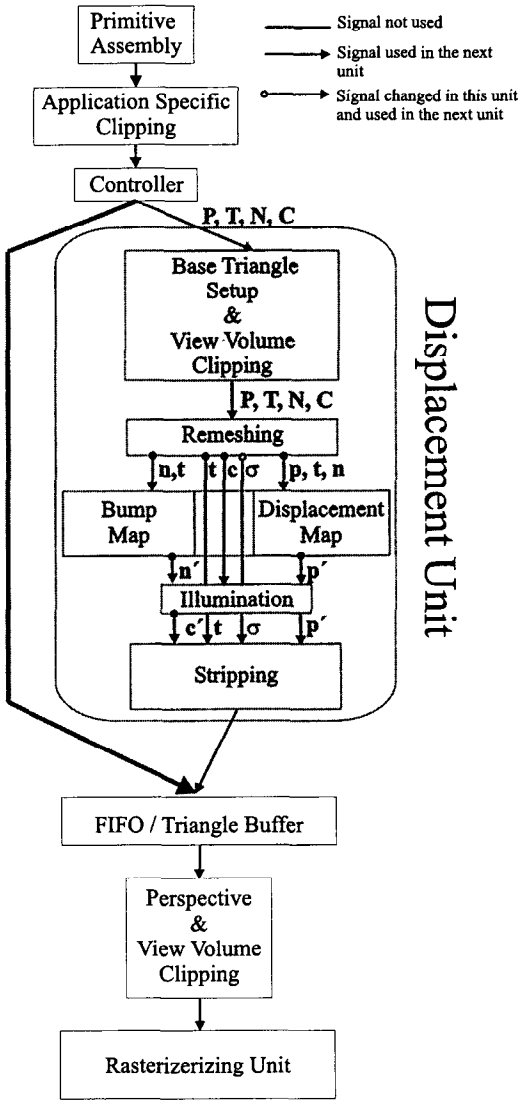

Figure 3: Displacement Mapping rendering pipeline.

The stripping unit is a mixture between generalized triangle strips as descibed in [14] and the generalized mesh as proposed by Deering [7]. It contains a buffer with four registers each of which can hold one vertex. The vertices are referenced by a two bit index. The stripping unit always produces sequential triangle strips, which can be built with stored vertices and new ones. Therefore, the stripping unit is controlled by the three commands *init*, *setref i* and *ref i*. The *init* command starts a new strip. The *setref i* command loads register i with a new vertex and adds the vertex to the current strip. Finally, the *ref i* command adds the buffered vertex in register i to the current strip. The commands are sent from the remeshing unit through the signal σ to the stripping unit. Each time

a triangle is produced in the current strip, the triangle vertices are placed into the triangle buffer. The stripping unit was introduced to allow the reuse of more displaced and illuminated vertices but could be skipped as well.

3 View Dependent Remeshing

In this section we describe the main contribution of our paper, a view dependent remeshing technique, which can be implemented in hardware. Base triangles are subdivided according to their extend in screen space. With the filtered access to the displacement maps, a view dependent multiresolution remeshing is achieved.

An appropriate remeshing is especially important for base triangles with normals orthogonal to the viewing direction, as the corresponding displacements contribute most to the silhouette of the displaced surface. Therefore, we extend the viewport in the viewing direction as described in section 3.1. In this way the remeshing can be done not only in x and y directions but also in x and z or in y and z directions.

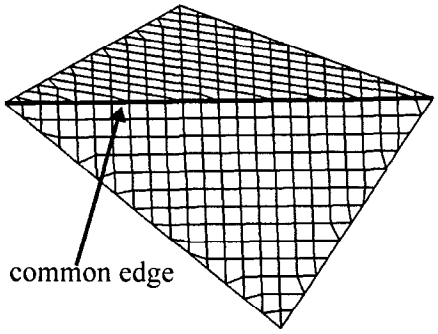

Figure 4: The proposed remeshing does not produce any t-vertices on a common edge of adjacent base triangles.

The most important constraint for the remeshing algorithm is the edge constraint (see section 2.1). If the vertices, which are newly generated by the remeshing, on an edge are not the same for both of its adjacent base triangles, t-vertices are generated. As the displacement map normally is not a linear map, cracks will appear at the t-vertices. They will be visible even if their size is smaller than a pixel, because a rounding operation in the rasterizer might produce different results or because the orientation of some triangles around the t-vertex can be flipped inconsistently such that the triangles are back face culled away by mistake. Our remeshing algorithm avoids the t-vertices by subdividing the edges only dependent on the two incident vertices. Figure 4 illustrates this feature of our algorithm. The two base triangles were remeshed with a resolution 45 times coarser than the screen resolution. The lower triangle is remeshed in x and y direction and the upper triangle in x and z. On their common edge no t-vertices arise.

The organization of the rest of this section is oriented at figure 5, which gives an overview of the remeshing unit containing the base triangle setup and the view dependent remeshing.

The setup unit is fed with three vertices given in world coordinates. Each vertex contains the position, the normal vector, the texture position and the color vector as noted in brackets on the top right side in figure 5. The first stage transforms the three vertices to screen space and to grid space. The latter is used for the remeshing and described in section 3.1. After the transformation the vertices are prepared for perspective correct interpolation. Therefore, section 3.2 describes the vector interpolation and the perspective correction approach. This also includes the last stage of the remeshing unit. The next stage in figure 5 clips the base triangles. For this the view volume is extended according to the smallest z position of the base triangle and the maximal displacement as described in section 3.3. If the base triangle is clipped away, execution terminates, what is illustrated in figure 5 by the bold arrow pointing out of the unit. The remaining stages of figure 5 constitute the core of the remeshing algorithm and are described in section 3.4. For the orientation of the reader, in section 3.4 the beginning of a new stage is marked with a new subsection heading.

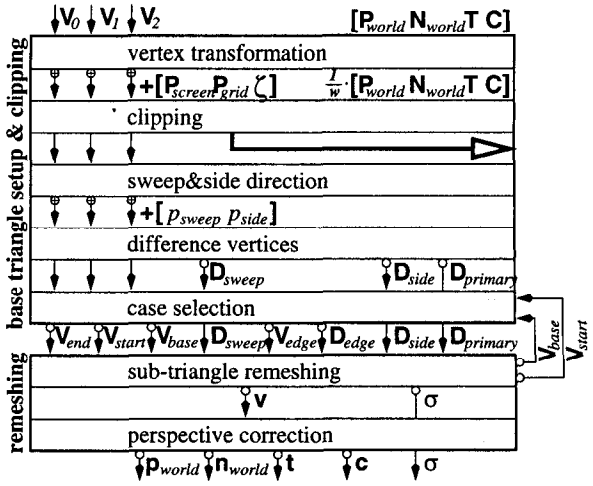

Figure 5: Overview diagram of remeshing unit.

3.1 Screen Space vs Grid Space

As mentioned in the introduction to this section, the remeshing might be done in z direction. It is very important that the remeshing resolution in z direction decreases according to the perspective projection with increasing distance from the view point. The first idea which comes to mind is to sample the z direction in screen space, which is the vector space resulting from the perspective projection including the division by the homogenous w coordinate. The problem is that for the z direction no appropriate scaling is known as in the case of the x and y coordinates, which is given by the viewport mapping. It is not possible to choose a constant scaling in z direction consistently over the complete z range. Our experiments showed for typical perspective projections that for any constant choice of the z scaling zones arise in the z direction where the pixel size will be stretched or compressed in z direction by a factor of ten and more. This is a serious problem if we imagine a remeshing in z direction of base triangles which also have some slope in x or y direction. Then one step in the z direction can produce jumps of up to ten pixels in the other direction, what conflicts with the idea of the view dependent remeshing.

Let us have a look at the problem in world coordinates. Figure 6 shows on the left the two dimensional viewport of width W and height H pixels in world coordinates taken at the near clipping plane. On the right the whole scenario is turned around the y axis by ninety degrees. The viewport is extended into the z direction in a way that the resulting voxels are all similar to cubes, i.e. all edges of one voxel have nearly the same length. A unit voxel grid in screen space is different from this intuitive voxel grid. Although the fraction of the voxel heights over the voxel widths is constant all over the z_{world} range, the fraction of the voxel heights or widths over the voxel depths varies significantly depending on the z loca-

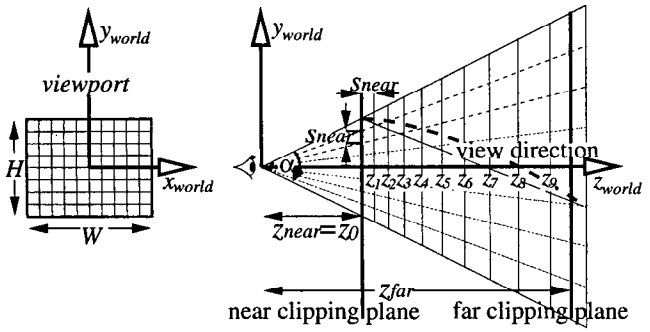

Figure 6: Extension of pixel grid in view direction.

tion. The z_{world} positions z_k (see figure 6) of the cube-like voxel grid can be calculated in a recursive manner from the pixel size s_{near} on the near clipping plane and the distance z_{near} of the near clipping plane from the view point. Knowing z_0 to be z_{near}, z_k is calculated for $k \geq 1$ via

$$z_k = z_{k-1} + \frac{z_{k-1}}{z_{near}} s_{near} \quad \Rightarrow \quad z_k = z_{near} \cdot \left(1 + \frac{s_{near}}{z_{near}}\right)^k. \tag{2}$$

With a view angle α in y direction and a viewport height of H pixels, s_{near} computes to

$$s_{near} = 2 \tan \frac{\alpha}{2} \frac{z_{near}}{H}. \tag{3}$$

Solving equation 2 for k yields a continuous mapping $z_{grid}(z_{world})$ from the z world coordinate to the z grid coordinate of the voxel grid. Together with equation 3 this yields

$$z_{grid} = \frac{\ln z_{world} - \ln z_{near}}{\ln\left(1 + \frac{2 \tan \frac{\alpha}{2}}{H}\right)}. \tag{4}$$

The x and y coordinates x_{grid} and y_{grid} of the voxel grid are the same as the viewport coordinates x_{screen} and y_{screen}, which are computed by applying the perspective projection followed by the viewport mapping. The vector space defined over the voxel grid coordinates x_{grid}, y_{grid} and z_{grid} is called the *grid space* in distinction to the *screen space* which is formed by x_{screen}, y_{screen} and z_{screen}, where z_{screen} is scaled to the range $[-1, 1]$.

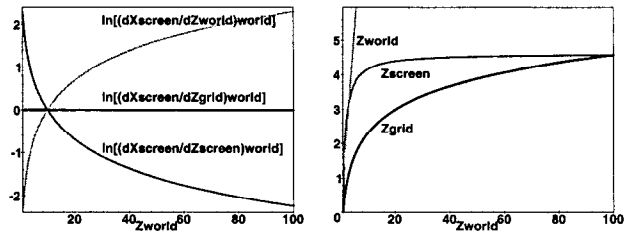

Figure 7: For $z_{near} = 1$, $z_{far} = 100$, $\alpha = 90°$ and $H = W = 300$: a) Fraction of voxel width over voxel depth computed for the screen space voxel grid in world coordinates as function of z_{world}; b) z_{grid} and z_{screen} as function of z_{world}.

Figure 7 illustrates once more the difference between grid space, screen space and world coordinates. Figure 7 a) plots the fraction of the voxel width over the voxel depth on a logarithmic scale as a function of the distance z_{world} from the view point. The voxel width was computed according to equation 3 as $2 \tan \frac{\alpha}{2} \frac{z_{world}}{H}$. Grid space is defined in a way that the voxel depth equals the voxel width not only in grid coordinates but also in world coordinates. Therefore, the corresponding fraction is constant equal to one and its logarithm zero. In screen space z_{screen} needs not be scaled to $[-1, 1]$ but can be scaled to any interval. In figure 7 a) we scaled it such that the fraction of the voxel width over the voxel depth computed in world coordinates equals one at $z_{world} = 10$. But the fraction is not at all equal to one anywhere else and therefore screen space is not suitable for a remeshing in the z direction. We also showed the fraction of width over depth for a voxel grid with $0.1 \cdot z_{world}$ as voxel depth.

Figure 7 b) compares the plots of $z_{grid}(z_{world})$ and $z_{screen}(z_{world})$. This time the z_{screen} coordinate was scaled such that both plots have the same values at $z_{world} = z_{near}$ and $z_{world} = z_{far}$.

3.1.1 Vertex Transformation

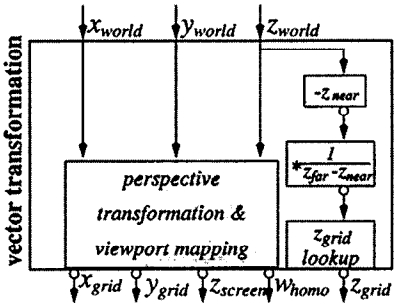

Figure 8: Hardware design for transformation from world positions to grid positions and screen positions.

Figure 8 shows a hardware design for the transformation from world positions to grid and screen positions. On the left of the diagram the world coordinates are perspectively transformed and mapped to viewport coordinates. The z coordinate is mapped to a device independent coordinate such that the range $[z_{near}, z_{far}]$ in world coordinates is mapped to $[-1, 1]$ in screen space. The perspective transformation outputs also the homogenous w coordinate w_{homo} of the vertex, which is needed for perspective correction.

On the right side of figure 8 the z world coordinate is scaled to the range $[0, 1]$ and then transformed to the grid coordinate with a linearly interpolating lookup into a one dimensional float valued array which realizes equation 4. This array and the parameters for the perspective transformation only need to be initialized once after each change of a camera parameter and or the viewport.

3.2 Vertex Interpolation and Perspective Correction

The remeshing in grid space bears a problem. As the transformation from z_{world} to z_{screen} is not linear nor perspective, a line and therefore also a base triangle in grid space will be bent in world coordinates. This is illustrated by the bold dashed line in figure 6. The line was drawn in the y/z plane of grid space with slope minus one half, i.e. it moves two grid voxels in z direction for each voxel in the negative y direction. In world coordinates and therefore also in screen coordinates the line is bent.

If the sweep-line algorithm was performed in grid space, the newly generated vertices would have to be projected back onto the base triangle to avoid curved base triangles, which is essential

for correct texture, bump and displacement mapping. We project the grid positions onto the triangle in screen space in a very simple way. Let $\mathbf{p}_{grid,1}, \mathbf{p}_{grid,2}$ and $\mathbf{p}_{grid,3}$ be the corner locations of a base triangle in grid space. Then any grid space location $\mathbf{p}_{grid}$ can be expressed with the help of its barycentric coordinates $\sigma_{grid,i}$ as $\mathbf{p}_{grid} = \sum_{i=1}^{3} \sigma_{grid,i}\mathbf{p}_{grid,i}$. If $\mathbf{p}_{screen,i}$ are the screen space corner locations of the base triangle, $\mathbf{p}_{grid}$ is projected onto the location $\mathbf{p}_{screen}$ on the base triangle in screen space by $\mathbf{p}_{screen} = \sum_{i=1}^{3} \sigma_{grid,i}\mathbf{p}_{screen,i}$. In this way we use z_{grid} to define a scaling for the z coordinate in screen space, which is adapted to the current triangle. Only for triangles which span the complete z range the grid space approach degenerates to the remeshing in screen space. In practice, no barycentric coordinates are needed. The vertex data is interpolated by applying the same vector operations which are performed in grid space in a one to one fashion to the vertex data in screen space.

As all the vertex data will be needed in world coordinates in successive units, the new vertices which are generated in screen space need to be perspectively corrected. The standard perspective correction method can be applied. We interpolate the vertex data in world coordinates divided by the homogenous w coordinate w_{homo} of the vertex, which is computed by the vertex transformation stage shown in figure 8. In addition to the vertex data we have to interpolate the homogenous w coordinate. From the relation between w_{homo} and the constant focal distance f_0, which can be found in standard literature,

$$w_{\text{homo}} = \frac{z_{world}}{z_{\text{screen}}} = \frac{f_0}{f_0 - z_{\text{screen}}} \tag{5}$$

we can deduce that the quantity

$$\zeta \stackrel{\text{def}}{=} \frac{1}{w_{\text{homo}}} - 1 = -\frac{z_{\text{screen}}}{f_0} \tag{6}$$

is proportional to z_{screen} and can therefore be interpolated in screen space. Thus, after the vertex transformation stage (see figure 5), the vertex data is multiplied by $\frac{1}{w}$ and ζ becomes part of the vertex data.

One could imagine also to interpolate the position and normals in screen coordinates in order to avoid the perspective transformation of the newly generated vertices before the rasterizing unit (see figure 3). This is actually possible but the increase of the hardware complexity for the interpolation and the correct computation of the displacement in screen coordinates is higher than the complexity for the perspective transformation unit. Simple pipelining of the units in figure 3 can avoid any computational overhead of the proposed approach.

The vertex data in world coordinates are obtained from the interpolated values by multiplying each component with $\frac{1}{\zeta+1} = w_{homo}$, which is done in the perspective correction stage in figure 5 at the end of the remeshing unit before a newly generated vertex is sent to the mapping, illumination and stripping units.

3.3 Base Triangle Clipping

After the transformation of the base triangle positions to grid and screen space, follows the clipping of the base triangles within screen space. As the displaced surface over the base triangle can be very complex and is not known before the remeshing stage, we propose a conservative clipping mechanism. As in each displacement map the minimum and the maximum displacement in world coordinates are stored, also the absolute value d_{max} of the maximum displacement is known in world coordinates. For each base triangle we extend the clipping volume as shown in figure 9 by the length d_{max}.

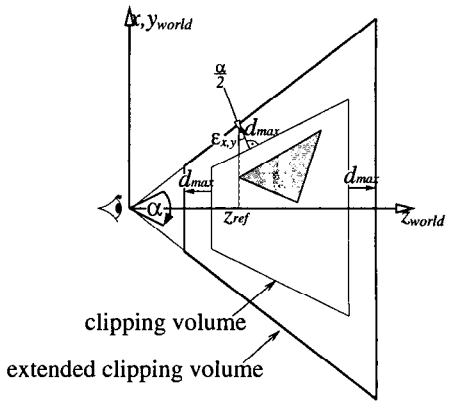

Figure 9: Extension of clipping volume for base triangle clipping.

The figure shows the situation in the more intuitive world coordinates. The influence of d_{max} onto the extension of the clipping volume in screen coordinates depends on the z coordinate. If the triangle is nearer to the view point the displacement can span more pixels as if the triangle is further apart. Therefore, we transform the length d_{max} at the minimal z coordinate z_{ref} (see figure 9) of the base triangle. In x and y direction we have to take care since the faces of the clipping volume do not lay in world coordinate planes. We have to substitute d_{max} with

$$\epsilon_{x,y} = \frac{d_{max}}{\cos\frac{\alpha}{2}}. \tag{7}$$

The perspective transformation and the viewport mapping yield the new clipping coordinates x'_{min}, x'_{max}, y'_{min}, y'_{max}, z'_{min} and z'_{max} from the original clipping coordinates x_{min}, x_{max}, y_{min}, y_{max}, -1 and 1:

$$\begin{aligned}
x'_{\text{min}} &= x_{\text{min}} - \frac{W\cdot\epsilon_{x,y}}{2\tan\frac{\alpha_x}{2}} & x'_{\text{max}} &= x_{\text{max}} + \frac{W\cdot\epsilon_{x,y}}{2\tan\frac{\alpha_x}{2}} \\
y'_{\text{min}} &= y_{\text{min}} - \frac{H\cdot\epsilon_{x,y}}{2\tan\frac{\alpha_y}{2}} & y'_{\text{max}} &= y_{\text{max}} + \frac{H\cdot\epsilon_{x,y}}{2\tan\frac{\alpha_y}{2}} \\
z'_{\text{min}} &= -1 - 2\frac{z_{far}d_{max}}{(z_{far}-z_{near})(z_{near}-d_{max})} \\
z'_{\text{max}} &= 1 + 2\frac{z_{near}d_{max}}{(z_{far}-z_{near})(z_{far}-d_{max})}
\end{aligned} \tag{8}$$

Better clipping results can be obtained by using for each base triangle an appropriate d_{max}. This can either be specified by the user or can be computed from an additional hierarchy over the displacement map, which stores the maximal displacement for each cell.

3.4 Sweep-Line Algorithm in Gris Space

As a first simplification of the remeshing algorithm we round the grid positions of the corner vertices to integer values, which are in pixel units. The screen positions are not rounded such that the base triangle will still be interpolated at the corner points. This is very important if the remeshing resolution is coarser than the screen resolution, since in this case the rounding of the screen coordinates would change the corner positions of the triangles by a visible amount. Our method has the side effect that the remeshing directions are slightly distorted from the screen space directions. This is the reason, why the lines in the lower triangle of figure 4 are not exactly parallel to the x and y directions.

The leading idea of the sweep-line algorithm is illustrated in figure 10. First the base triangle is projected into the best suited coordinate plane in grid space (figure 10 a)). The selected coordinate plane is called the *remeshing plane*. In the remeshing plane the voxel grid projects to a quadrilateral grid. The quadrilaterals have

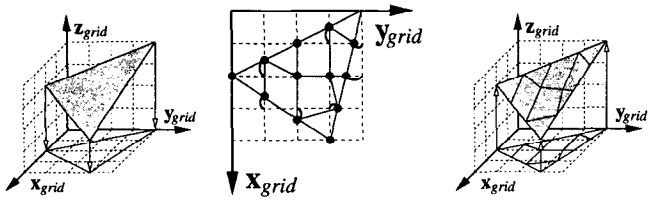

Figure 10: Leading idea of the remeshing: a) Base triangle is projected to the remeshing plane. b) Remeshing, vertices near edges are dragged onto edges. c) Partitioning is projected back onto the base triangle.

the size of a pixel on the screen and are ideally suited for remeshing the base triangle. Grid vertices near the edges of the triangle are dragged onto the edges as hinted in figure 10 b). Finally, this remeshing is used to partition the original base triangle into quadrilaterals and triangles as indicated in figure 10 c).

The edge constraint (see section 2.1) implies that the newly generated vertices on the edges may only depend upon the incident corner vertices of the base triangle, such that no t-vertices are produced. Therefore, we define for each edge the *main direction* as follows:

Definition 1 *The* main direction *of an edge from* $\mathbf{p}_1$ *to* $\mathbf{p}_2$ *is the coordinate direction* x_{grid}, y_{grid} *or* z_{grid}*, respectively, iff* $\Delta x \stackrel{def}{=} |\mathbf{p}_2.x_{grid} - \mathbf{p}_1.x_{grid}|$*,* Δy *or* Δz*, respectively, is the maximum among* $\Delta x, \Delta y$ *and* Δz*. If the maximum is not unique,* x_{grid} *has the highest priority and* z_{grid} *the lowest.*

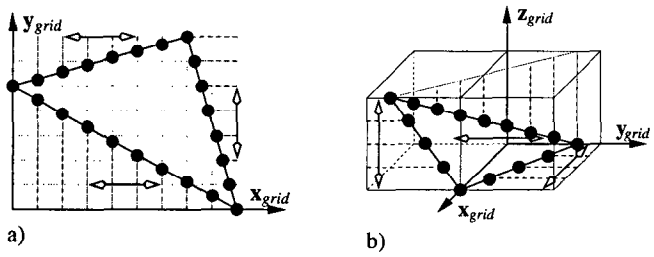

Figure 11: Arrows indicate main direction of edges. a) New vertices on the edges are generated along the main direction. b) Triangle with three different main directions.

Each edge is subdivided along the main direction in pixel units as shown in figure 11 a). For most base triangles the remeshing plane can be chosen in a way that all main directions of the edges lay within the remeshing plane. In this case each newly generated edge vertex lies at least on one grid line. But the three dimensional screen space bears triangles, where all three edges have different main directions as shown in figure 11 b). This yields some more special cases during remeshing because two different edge points might round to the same grid point when projected to the remeshing plane.

Back to simplifications of the remeshing algorithm. An edge is called a *primary edge* of the base triangle if the third triangle vertex lies inside the primary edge according to the main direction of the primary edge. More formal:

Definition 2 *An edge* $\mathbf{p}_1\mathbf{p}_2$ *with main direction* x_{grid} *is a* primary edge *of the triangle* $\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$*, iff* $\min(\mathbf{p}_1.x_{grid}, \mathbf{p}_2.x_{grid}) \leq \mathbf{p}_3.x_{grid} \leq \max(\mathbf{p}_1.x_{grid}, \mathbf{p}_2.x_{grid})$*. Similar definitions apply to main directions* y_{grid} *and* z_{grid}*.*

Theorem 1 *For each triangle exists at least one primary edge.*

Sketch of Proof: *If all edges of the triangle have the same main direction this is trivial. In case of two edges with the same main direction their common vertex must have a minimal or maximal main direction coordinate, otherwise all edges have the same main direction. The edge between the common vertex and the other vertex with extremal main direction coordinate is a primary edge. If all edges have different main directions, the proof leads all attempts to generate a counter example to a contradiction: The nine planes* $x = \mathbf{p}_1.x_{grid}/\mathbf{p}_2.x_{grid}/\mathbf{p}_3.x_{grid}, \ldots, z = \mathbf{p}_1.z_{grid}/\mathbf{p}_2.z_{grid}/\mathbf{p}_3.z_{grid}$ *form eight cubic cells with 27 vertices. The three triangle corners must be placed on the cell-vertices such that on each plane lies at least one corner. This implies that no two vertices lie on the same plane. There are only three different cases up to symmetry: 1) one corner-*[1]*, one face- and one edge-vertex; 2) two corner- and the interior vertex and 3) three edge-vertices. Next the main directions are assigned to the edges such that no edge spans the complete range of its main direction. There is one possibility for case 1), none for case 2) and two possibilities for 3). Finally, for each main direction two inequalities among the six distances of adjacent parallel planes are derived from the definition of main direction. All of the three possible assignments of main directions lead to contradicting inequalities.* □

3.4.1 Sweep & Side Direction

Theorem 1 justifies the following strategy for the sweep-line algorithm. First the longest primary edge is selected and the corresponding main direction will be used as the *sweep direction* for the sweep-line. Coordinates in the sweep direction are denoted with p_{sweep}. The other two edges will be referred to as *side edges*. Secondly, the remeshing plane is chosen from the two coordinate planes containing the sweep direction to maximize the projected area of the base triangle. The second coordinate direction in the remeshing plane is called *side direction* and coordinates in this direction are denoted with p_{side}.

3.4.2 Difference Vertices

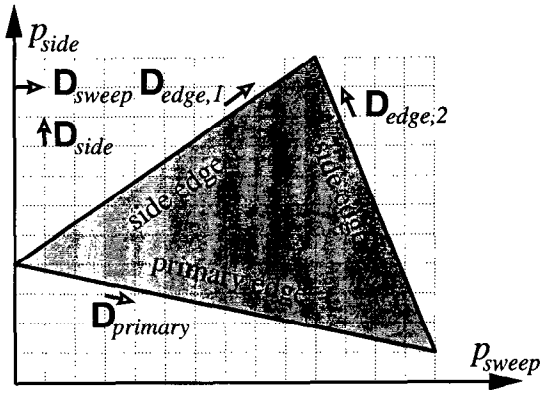

Figure 12: The vertex differences computed in the base triangle setup.

For the incremental update of the newly generated vertices one has to compute the difference vertices $\mathbf{D}_{primary}$, $\mathbf{D}_{edge,1}$, $\mathbf{D}_{edge,2}$, $\mathbf{D}_{sweep}$ and $\mathbf{D}_{side}$ (see figure 12). A difference vertex contains not only the difference in p_{sweep} and p_{side} but also the difference in all components of the vertex data. $\mathbf{D}_{primary}$ is the

[1]The notion of corner-, edge-, face- and interior vertex is applied to the convex hull of the eight cells.

difference vector in direction of the primary edge and can be computed by the vertex difference of the incident vertices divided by the difference in their p_{sweep} direction. $\mathbf{D}_{edge,1}$ and $\mathbf{D}_{edge,2}$ contain the difference vertices of the remaining edges and are calculated as $\mathbf{D}_{primary}$ except that they are divided by the difference in their main directions.

$\mathbf{D}_{sweep}$ and $\mathbf{D}_{side}$ have p_{sweep} and p_{side} difference coordinates of $[1,0]$ and $[0,1]$. Their vertex data need to be calculated from $\mathbf{D}_{primary}$ and $\mathbf{D}_{edge,1}$ or $\mathbf{D}_{edge,2}$. For numerical stability we choose among $\mathbf{D}_{edge,1}$ and $\mathbf{D}_{edge,2}$ the one with the larger slope in side direction, imagine this to be $\mathbf{D}_{edge,1}$. We calculate the matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ from

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{D}_{primary} \cdot p_{sweep} & \mathbf{D}_{edge,1} \cdot p_{sweep} \\ \mathbf{D}_{primary} \cdot p_{side} & \mathbf{D}_{edge,1} \cdot p_{side} \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \tag{9}$$

For the inversion of the 2x2-matrix, which is formed by the difference vertices $\mathbf{D}_{sweep}$ and $\mathbf{D}_{side}$, we only need two multiplications, one division, two negations and one subtraction. Finally, $\mathbf{D}_{sweep}$ and $\mathbf{D}_{side}$ can be computed

$$\mathbf{D}_{sweep} = a \cdot \mathbf{D}_{primary} + c \cdot \mathbf{D}_{edge,1} \tag{10}$$

$$\mathbf{D}_{side} = b \cdot \mathbf{D}_{primary} + d \cdot \mathbf{D}_{edge,1}. \tag{11}$$

3.4.3 Case Selection

In order to allow an incremental update of the vectors used by the sweep-line algorithm it is important that the vectors on the side edges have increasing coordinates in side direction. Therefore, we distinguish the three cases illustrated in figure 13. In the roof case

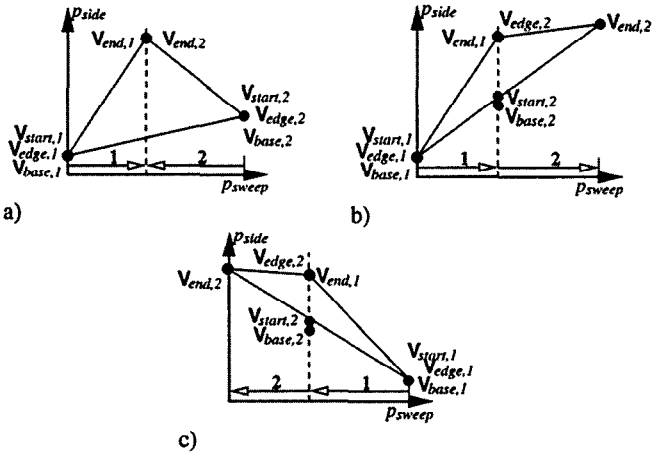

Figure 13: The three different cases of the orientation in side direction of the side edges: a) Roof. b) Increasing. c) Decreasing.

one side edge has an increasing slope in side direction and the other edge has a decreasing slope. In the other two cases both side edges are either increasing or decreasing. The base triangle is split in all three cases at the side vertex, which is common to both side edges, into two sub-triangles each consisting of one side edge, a part of the main edge and a new interior edge, which is parallel to the side direction. For each sub-triangle a separate sweep-line process is performed and the sweep direction is chosen such that the side coordinate on the side edge is increasing as illustrated in figure 13 by the arrows.

The base triangle setup unit (see figure 5) transmits for each sub-triangle four difference vertices and the four vertices $\mathbf{V}_{start}$, $\mathbf{V}_{end}$, $\mathbf{V}_{base}$ and $\mathbf{V}_{edge}$ (see figure 13). The meaning of these vertices is described in the next section. After the remeshing of the first sub-triangle new vertices $\mathbf{V}_{start}$ and $\mathbf{V}_{base}$ are transferred back to the case selection unit which are used in the Increasing and Decreasing case as input for the sub-triangle remeshing of the second sub-triangle.

3.4.4 Sub-Triangle Remeshing

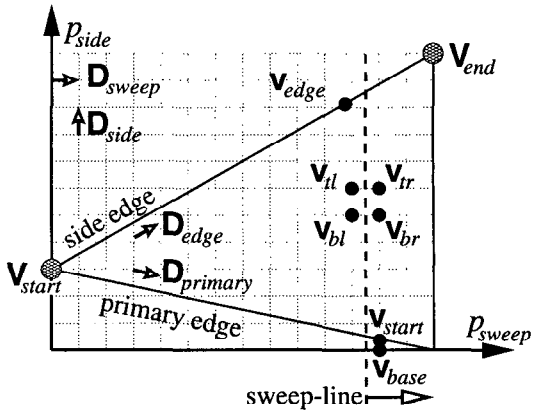

Figure 14: Vertices and vertex differences used during the sweep-line algorithm.

Figure 14 shows a sample sub-triangle together with the vertices and vertex differences used during remeshing. $\mathbf{V}_{start}$ is not stored and therefore twelve registers are needed on the sub-triangle remeshing unit for the storage of the different vertices and vertex differences. Each register each contains the remeshing coordinates p_{sweep} and p_{side} and the vertex data. The registers are indexed with a four bit index and the remeshing algorithm always works with these indices, such that assignment of vertices and comparison on identity can be simplified significantly. Vertices which are invalid are represented through an index larger than eleven.

The sweep-line moves from $\mathbf{V}_{start}$ to $\mathbf{V}_{end}$ always addressing two p_{sweep} positions such that quadrilaterals can be formed. In $\mathbf{v}_{start}$ the current vertex on the primary edge is stored, which is updated if the sweep-line moves on by adding $\mathbf{D}_{sweep}$. The newly generated vertices are denoted with lower case letters in contrast to the upper case notation of the vertex differences and these vertices which are computed only once per base triangle. At each sweep-line position the algorithm produces a quad-strip beginning at the current $\mathbf{v}_{start}$ and the next $\mathbf{v}_{start}$. The vertices $\mathbf{v}_{bl}, \mathbf{v}_{br}, \mathbf{v}_{tl}$ and $\mathbf{v}_{tr}$ are used to store the corner vertices of the current quadrilateral. The vertices $\mathbf{v}_{tl}$ and $\mathbf{v}_{tr}$ are computed from $\mathbf{v}_{bl}$ and $\mathbf{v}_{br}$ by adding $\mathbf{D}_{side}$. In the inner loop of the remeshing algorithm the so called *side walk* the quadrilateral is moved in side direction by setting $\mathbf{v}_{bl}$ to $\mathbf{v}_{tl}$, $\mathbf{v}_{br}$ to $\mathbf{v}_{tr}$ and by recomputing $\mathbf{v}_{tl}$ and $\mathbf{v}_{tr}$.

$\mathbf{v}_{bl}$ is initialized to $\mathbf{v}_{start}$ and $\mathbf{v}_{br}$ to $\mathbf{v}_{start} + \mathbf{D}_{primary}$. As the main direction of the primary edge is equivalent to the sweep direction and as the grid coordinates have been rounded to integer values, the p_{sweep} coordinates of both points are integer valued. On the other hand the p_{side} coordinates are normally not integer valued. For this the vertex $\mathbf{v}_{base}$ is placed on the nearest grid point to $\mathbf{v}_{start}$. If the sweep-line moves on, $\mathbf{v}_{base}$ is updated by adding $\mathbf{D}_{sweep}$ and probably $\mathbf{D}_{side}$ as needed. Thus the vertices $\mathbf{v}_{tl}$ and $\mathbf{v}_{tr}$ can also in the first step be computed by adding $\mathbf{D}_{side}$ but this time to $\mathbf{v}_{base}$ and the next $\mathbf{v}_{base}$. In this way a multiplication of $\mathbf{D}_{side}$ with the fractional part of $\mathbf{v}_{start} \cdot p_{side}$ can be avoided.

During the side walk the top vertices finally hit the second edge. This is detected by comparing the top vertices to $\mathbf{v}_{edge}$. As the case

selection prepared the sub-triangles in a way that the side coordinate of the upper edge is always increasing, $\mathbf{v}_{tl}$ must hit the upper edge first. This allows us to keep only one $\mathbf{v}_{edge}$ which is first used for $\mathbf{v}_{tl}$ and then moved on along the edge by adding $\mathbf{D}_{edge}$ to serve as edge detection for $\mathbf{v}_{tr}$. If a top vertex is on the grid point which is nearest to $\mathbf{v}_{edge}$, the top vertex is dragged to the edge vertex in order to avoid t-vertices. If the right top vertex $\mathbf{v}_{tr}$ hits the edge, the side walk terminates. This is detected in the quadrilateral/triangle generation unit which generates for each new quadrilateral or triangle the commands for the stripping unit. Afterwards the bottom vertices are set to the top vertices, such that the current quadrilateral is moved on in side direction.

The stripping unit is adapted to the sub-triangle remeshing algorithm and supports storage and reuse of the four vertices $\mathbf{v}_{start}$, $\mathbf{v}_{edge}$, $\mathbf{v}_{bl}$ and $\mathbf{v}_{br}$.

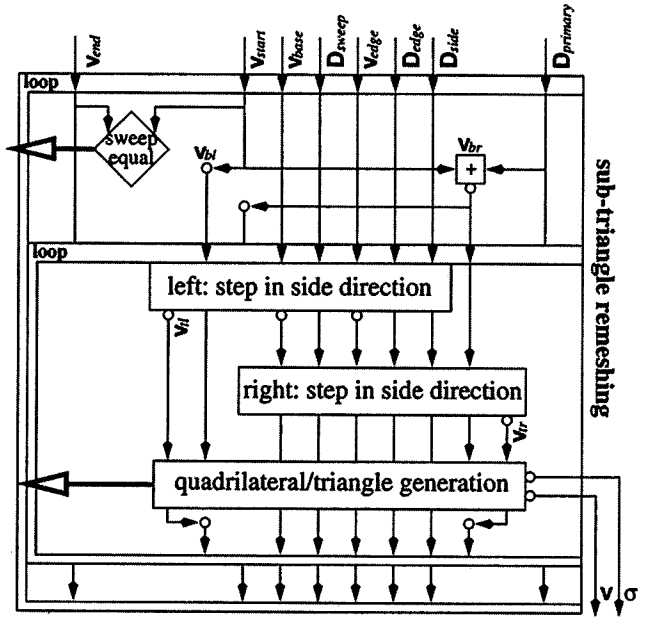

Figure 15: Sweep-Line.

Figure 15 shows the hardware layout of the sweep-line algorithm. Here the different vertex indices are shown in different columns. The two loops are shown as boxes. The outer loop is left when the current start vertex is on the same sweep position as the end vertex. The inner loop is left when the side walk has been terminated. All signals which enter a loop must also exit the loop. The signals $\mathbf{v}_{start}$ and $\mathbf{D}_{primary}$ are not used in the inner loop but therefore reappear at the end of the outer loop. The vertices $\mathbf{v}_{tl}$ and $\mathbf{v}_{tr}$ are only used in the inner loop and no arrows in the corresponding columns exit the inner loop.

The outer loop for the sweep line is very simple. $\mathbf{v}_{bl}$ and $\mathbf{v}_{br}$ are computed from $\mathbf{v}_{start}$ and $\mathbf{D}_{primary}$. After that $\mathbf{v}_{start}$ is set to $\mathbf{v}_{br}$. In the inner loop three major stages can be distinguished.

The first two stages compute the vertices $\mathbf{v}_{tl}$ and $\mathbf{v}_{tr}$. During the computation of $\mathbf{v}_{tl}$ the vertices $\mathbf{v}_{base}$ and $\mathbf{v}_{edge}$ are moved on in sweep direction if they have been used on the left side. The vertices $\mathbf{v}_{tl}$ and $\mathbf{v}_{tr}$ are dragged onto $\mathbf{v}_{edge}$ if needed. There are two cases when a top vertex must be set to invalid:

1. the corresponding bottom vertex is already on the upper edge
2. the bottom vertex on the other side has a smaller side coordinate as the bottom vertex on this side

The stage which computes $\mathbf{v}_{tl}$ must check during the update of $\mathbf{v}_{edge}$ if the next $\mathbf{v}_{edge}$ lies on the same grid point. In this case the triangle connecting $\mathbf{v}_{bl}, \mathbf{v}_{edge}, \mathbf{v}_{edge} + \mathbf{D}_{edge}$ is generated, $\mathbf{v}_{tl}$ is set to the next $\mathbf{v}_{edge}$ and $\mathbf{v}_{edge}$ is moved on once more in the edge direction.

The third stage generates a quadrilateral or a triangle depending on the validity of the four points $\mathbf{v}_{bl}, \mathbf{v}_{br}, \mathbf{v}_{tl}$ and $\mathbf{v}_{tr}$. It produces commands for the stripping unit. The new vertices $\mathbf{v}_{tl}$ and $\mathbf{v}_{tr}$ are placed into the register of the stripping unit with the same index. For this the interpolated vertex data is first perspectively corrected as described in section 3.2. After the perspective correction the resulting vertex position, the normal, the texture coordinates and the vertex color are all given in world coordinates projected back to the base triangle. Next the world coordinate data is sent to the mapping and the illumination units. Reused vertices are referred to at the stripping unit. After the triangle or quadrilateral has been generated either the side walk is terminated or the bottom vertices are set to the top vertices and the side walk is continued.

All stages of the sub-triangle remeshing unit need only the following four basic vector operations

1. vector addition
2. vector assignment
3. comparison on identity
4. comparison on same grid point.

The vector addition is the most expensive to build but can be parallelized optimally. It is used any time a vector difference is added to a vector. The vector assignment is done by assigning the indices which is optimally simple and fast. Also the comparison on identity is done through the indices and therefore very cheap. The comparison is needed to check whether the bottom vertices are equal to $\mathbf{v}_{start}$ or $\mathbf{v}_{edge}$. The comparison on the grid point equality is used to drag the top vertices onto the upper edge. For this a rounded comparison needs to be performed on the coordinates p_{sweep} and p_{side} only. The important observation is that the sub-triangle remeshing algorithm uses only additions and comparisons, no multiplications. In average only slightly more than two additions and some comparisons are needed per produced quadrilateral.

4 Multiresolution Filtering

The access to the displacement map will produce aliasing if the object moves far away from the observer, since then each quadrilateral produced by the remeshing unit spans a large area in the displacement map. To allow easy adaptation of current hardware we oriented our filtering approach at the MIPmap technique. Thus with each displacement map a filter level with half of the original extend and a level with a quarter of the original extend and so on are supplied by the user.

4.1 Filter Level Access

The access to the filter levels by the displacement mapping unit will be similar to texture mapping approaches except that the subsamples will not be combined with an averaging filter but by selecting the maximum. We recommend the maximum selection as it conserves mountains on the silhouette of the objects. It will flatten out valleys if they are of pixel width on the screen, but that is ok since on the silhouette mountains are much more important for a natural look of the object. If a displaced surface is viewed from the backside, not a maximum but a minimum filter is used.

In order to allow the displacement mapping unit to determine a footprint around each newly generated vertex, the remeshing algorithm supplies the displacement mapping unit with four additional

texture positions. These vertices are computed from the newly generated vertex by adding $\pm\frac{1}{2}\mathbf{D}_{sweep} \pm \frac{1}{2}\mathbf{D}_{side}$. For vertices on the edges only two additional texture positions can be computed by adding $\pm\frac{1}{2}\mathbf{D}_{edge}$ or $\pm\frac{1}{2}\mathbf{D}_{primary}$. The edge constraint does not allow to use $\mathbf{D}_{sweep}$ or $\mathbf{D}_{side}$ for computation of the footprint, because otherwise the displaced surface might get cracks. It is no serious problem that only two additional vertex positions can be generated on the base triangle edges since the projection into the sampling plane cannot stretch and compress a footprint by more than a factor[2] of about $\sqrt{2}$. Thus the footprint of the three texture positions along an edge can be stretched to a square, without introducing an uncontrolled error.

4.2 Filter Level Generation

The averaging filters which are used for MIPmap generation flatten out the displacement field. Applying an averaging filter to the head of Volker, which is shown in figures 1 and 18, shrinks the complete head and shortens the nose significantly. Neither works a sinc-filter which cuts away high frequencies of the displacement map.

The problem is similar to the mesh simplification problem. For our purposes it is most important to conserve the silhouette of the object. We believe that the best approach would be to minimize the two-sided Haussdorff distance as proposed in [16]. But so far we did not find an algorithm which is fast enough. Therefore, we minimize the volume between the original and the approximated displacement map, which is some kind of approximation to the two-sided Haussdorff distance.

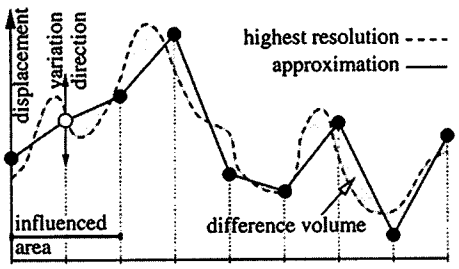

Figure 16: Variational approach for displacement map simplification which minimizes the volume between the original and the simplified surface.

Figure 16 shows the problem in a two dimensional projection. We used a variational approach to minimize the volume between the approximation in a certain filter level and the highest resolution of the displacement map. The grid positions of the vertices in the approximation are fixed, only their displacement can be varied. The difference volume is a function of the vertex displacements of the approximating grid. If the displacement of one vertex is varied, the approximation surface is changed only over a small influenced area, as illustrated in figure 16 for the second vertex. Thus we can compute the partial derivative $\partial_i V$ of the difference volume in direction of the displacement of the i^{th} vertex in the following way. First the volume over the influenced area V_i is computed, then the vertex is displaced by an ϵ and again the difference volume over the influenced area V_i' is computed. If A is the size of the influenced area, the partial derivative can be computed from

$$\partial_i V \stackrel{\text{def}}{=} \frac{V_i' - V_i}{\epsilon A}. \tag{12}$$

$\partial_i V$ is computed for all vertices of the approximation surface before the vertices are displaced at the same time by an amount of $-\lambda\partial_i V$. This process is iterated until the difference volume converges to a minimum, which might be only a local minimum.

All filter levels where adjusted with this algorithm by minimizing the difference volume between the computed level and the displacement map in the highest resolution. We used an ϵ of 10^{-4} and varied λ between 2^6 and 2^{-2} in order to control the convergence. We initialized the filter level at the beginning of the iteration to an averaged resampling of the original displacement map. The difference volume was in most cases reduced to about 70% of the initial value. Results can be found in the next section.

5 Results

To demonstrate the power of the proposed remeshing algorithm we implemented the graphics pipeline from figure 3 in software. We did not implement any of the bump mapping approaches, but calculated the vertex normals of the remeshed meshes from the vertex positions and the connectivity information.

We produced two examples of real world objects. The first example in figure 18 is the head of Volker. He was scanned with a cyberware scanner which produces a texture and a displacement map parametrized in cylindrical coordinates with 512^2 texels. The base mesh is a cylinder consisting of 32 triangles as shown in figure 1. The displacement field was corrected to compensate for the difference between the base triangle mesh and a perfect cylinder. Figure 18 shows in a), b) and c) remeshings at three different distances from the view point. In d), e) and f) the different remeshed meshes are illustrated in wireframe mode.

Figure 17 shows the same game for our second example, a model of the earth. The wireframe meshes are shown from above in order to visualize the adaptation of the remeshing to the voxel grid size. The remeshed meshes in to b) and c) are too fine for wireframe mode. Therefore the wireframe meshes in e) and f) were remeshes on a voxel grid five times larger than the pixel size. e) and f) illustrate the influence of the perspective on the remeshing. Areas near to the view point are remeshed finer than areas further apart.

The earth is rendered with standard texture mapping and a texture map with 1024^2 texels for the complete earth. But between the second and the eighteenth degree of longitude and between the forty forth and the fifty sixth degree of latitude we mapped a displacement field with 1916×1438 texels and a higher resolution texture map of the Alps with 1024^2 texels onto eight coarse triangles. The displacement map for the Alps was scaled by a factor of thirty in order to see the Alps on the huge globe. Again we corrected the displacement map by the difference between the triangles and the sphere.

The need of a sophisticated filter technique is illustrated in figure 19. The filtered and the unfiltered versions in b) and c) were remeshed on voxels with twenty times the pixel size as illustrated around figure 19 c). Figure 19 shows the Alps remeshed on voxels of pixel size. The red line in figure 19 b) and c) is the silhouette of the original alps. It can be seen that the unfiltered Alps significantly loose in height, which is avoided with the filtering proposed in section 4. Even some accummulations of mountains can be recognized in the filtered version.

[2] $\sqrt{2}$ is the tangens of the maximal angle between a base triangle and the sampling plane in screen space. Thus a slight correction of this factor can result from perspective correction according to the difference in w_{homo} which is small for pixel sized distances.

6 Discussion

With this paper we showed that hardware based displacement mapping is practical and that it allows for view dependent multiresolution rendering with a very compact surface representation. We start off a new area of research and want to discuss in this section some problems and directions of future work.

One problem is that the remeshing only depends on the view point and the base surface but not on the displacement. Therefore, details with width less than a pixel but height much larger than a pixel might vanish. This can only be avoid by adapting the remeshing to the slope of the displacement map. In areas of large slope a higher remeshing resolution is needed. This would produce a distribution of vertices which is not as regular as in the current approach and therefore the meshing will become more difficult.

A feature of our remeshing algorithm is that the remeshing resolution can be coarsened to a size larger than a pixel. This reduces the number of fine triangles on cost of the quality of the silhouette. An interesting avenue for future work would be to figure out how the remeshing resolution can be adjusted to the displacement map and the screen projection. Base triangles which are orthogonal to the view direction may be remeshed in a much coarser resolution since they do not contribute to the silhouette. Easier should be the adaptation of the sampling resolution to the slope of the displacement map as this relation is view independent. Areas of the displacement map with large slope must be remeshed in a higher resolution than areas with small slope.

A problem not discussed in this paper is back side culling of base triangles. The only solution seems to be a hierarchical map over the normal vectors which stores normal cones. A base triangle can be culled away if the normal cone applied to the base triangle vertex normals always points away completely from the view point. But there might be better solutions.

Since the remeshing is done in the fixed voxel grid defined as extension of the viewport and the objects move in world coordinates, the remeshing of the object changes with every movement. The change in remeshing is only of pixel size, but large slopes in the displacement map can produce wagging of steep edges on the silhouette. These wagging artifacts during object movement increase with decreasing sampling resolution.

For our examples we adjusted the displacement map to the base surface in order to compensate for the creases in the base mesh. This is not the way a user wants to specify a displacement map, especially not if the displacement map should move over the base surface or if the base mesh is changed. The only solution is to use a smooth base surface as for example bezier patches.

In future work better methods must be developed to generate the filter levels for the displacement map. A further interesting question is how to build nice displacement fields from an arbitrary surface. At the first glance the step from a typical mesh simplification algorithm to the generation of displacement maps seems small. But there are a lot of problems as for example how to parametrize an arbitrary surface, how to cut the parametrization into an atlas containing only rectangular maps and how to avoid cracks along map borders.

Acknowledgements

The authors like to thank Volker Blanz for providing us with his digitized head.

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the project D1 within the Sonderforschungsbereich 382.

References

[1] Gary Bishop and David M. Weimer. Fast Phong shading. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 103–106, August 1986.

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

[3] U. Claussen. Real time phong shading. In D. Grimsdale and A. Kaufman, editors, *Fifth Eurographics Workshop on Graphics Hardware*, 1989.

[4] R. L. Cook. Shade trees. *Computer Graphics*, 18(3):223–231, July 1984.

[5] Franklin C. Crow. Summed-area tables for texture mapping. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 207–212, July 1984.

[6] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):21–30, August 1988.

[7] Michael F. Deering. Geometry compression. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 13–20. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[8] I. Ernst, D. Jackel, H. Rüsseler, and O. Wittig. Hardware-supported bump mapping. *Computers and Graphics*, 20(4):515–521, July–August 1996.

[9] Kurt Fleischer, David Laidlaw, Bena Currin, and Alan Barr. Cellular texture generation. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 239–248. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[10] Alain Fournier. Filtering normal maps and creating multiple surfaces. Technical Report TR-92-41, Department of Computer Science, University of British Columbia, October 30 1992. Wed, 25 Nov 1998 20:32:49 GMT.

[11] Andrew Glassner. Adaptive precision in texture mapping. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 297–306, August 1986.

[12] Paul S. Heckbert. Texture mapping polygons in perspective. TM 13, NYIT Computer Graphics Lab, April 1983.

[13] Hugues Hoppe. View-dependent refinement of progressive meshes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[14] Silicon Graphics Inc. GL programming guide. 1991.

[15] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. *Computer Graphics (SIGGRAPH '89 Proceedings)*, 23(3):271–280, July 1989.

[16] R. Klein, G. Liebich, and W. Straßer. Mesh reduction with error control. In R. Yagel, editor, *Visualization 96*. ACM, November 1996.

[17] Reinhard Klein. Multiresolution representations for surfaces meshes. *Computer & Graphics*, 22(1), January 1998.

[18] Anders Kugler. Imem: An intelligent memory for bump- and reflection-shading. In *Proceedings of Eurographics/SIGGRAPH Hardware Workshop '98*, pages 113–122. ACM SIGGRAPH, August 1998.

[19] A. A. M. Kuijk and E. H. Blake. Faster Phong shading via angular interpolation. *Computer Graphics Forum*, 8(4):315–324, December 1989.

[20] Michael Lounsbery, Tony D. DeRose, and Joe Warren. Multiresolution analysis for surfaces of arbitrary topological type. *ACM Transactions on Graphics*, 16(1):34–73, January 1997. ISSN 0730-0301.

[21] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 199–208. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[22] Fabrice Neyret. Modeling, Animating, and Rendering Complex Scenes Using Volumetric Textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55–70, January 1998.

[23] J. W. Patterson, S. G. Hoggar, and J. R. Logie. Inverse displacement mapping. *Computer Graphics Forum*, 10(2):129–139, June 1991.

[24] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 303–306. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[25] Bui-Tuong Phong. Illumination for computer generated pictures. *CACM June 1975*, 18(6):311–317, 1975.

[26] S. Gumhold R. Klein. Data compression of multiresolution surfaces. In *Visualization in Scientific Computing '98*, pages 13–24. Springer, May 1998.

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

[28] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, pages 1–11, July 1983.

[29] Julie C. Xia, Jihad El-Sana, and Amitabh Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2), April–June 1997. ISSN 1077-2626.

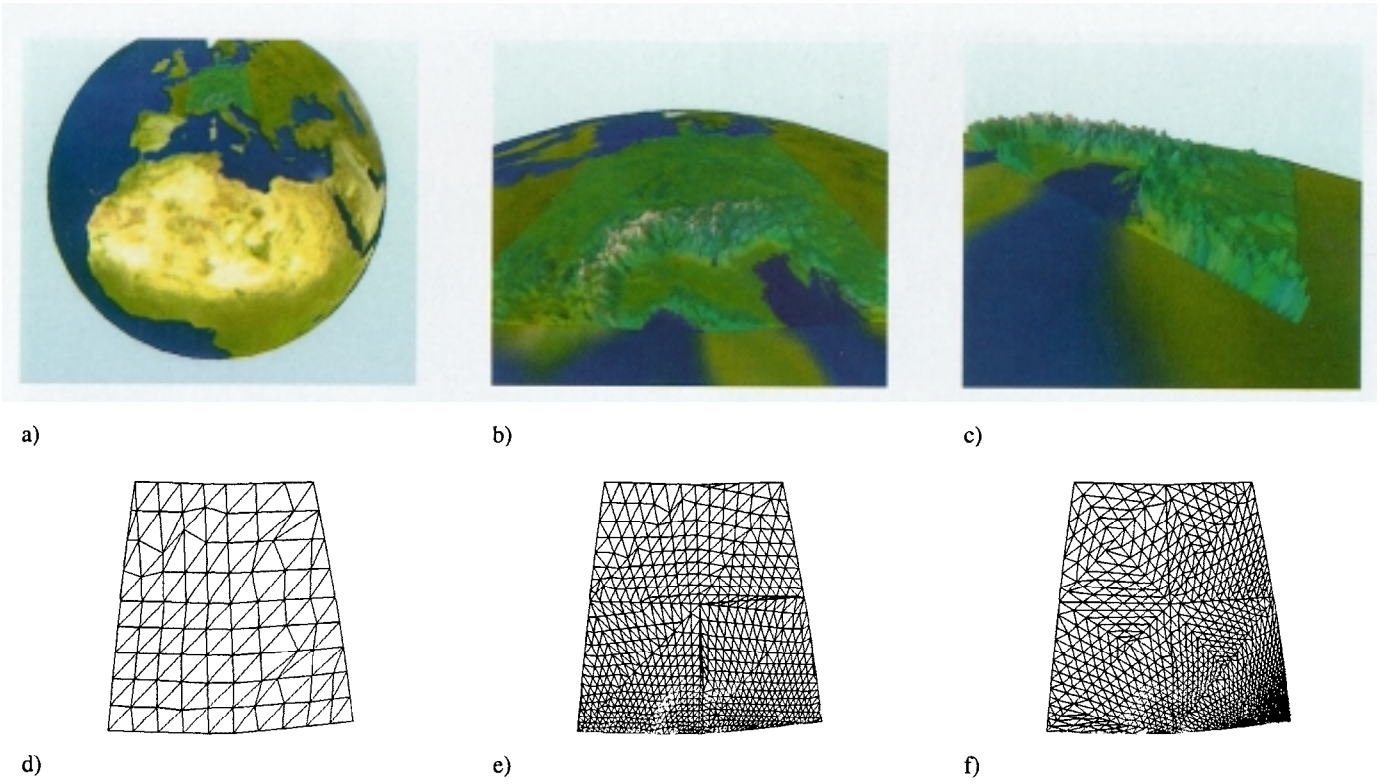

Figure 17: Displacement mapping for terrain modeling at the example of the Alps.

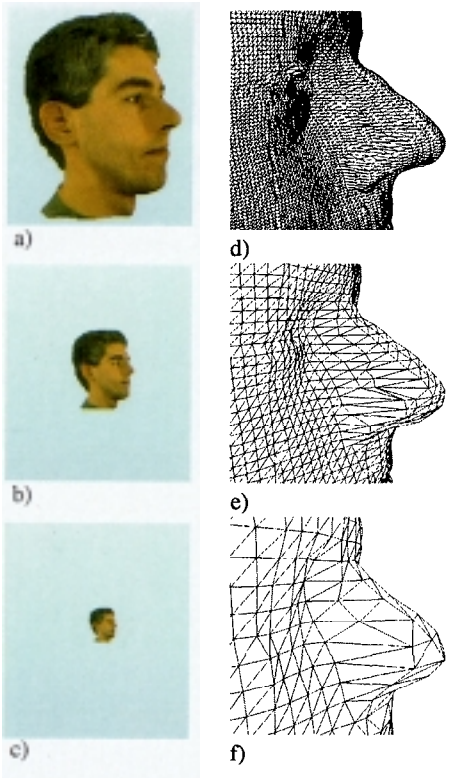

Figure 18: This is the head of Volker remeshed at three different distances from the view point.

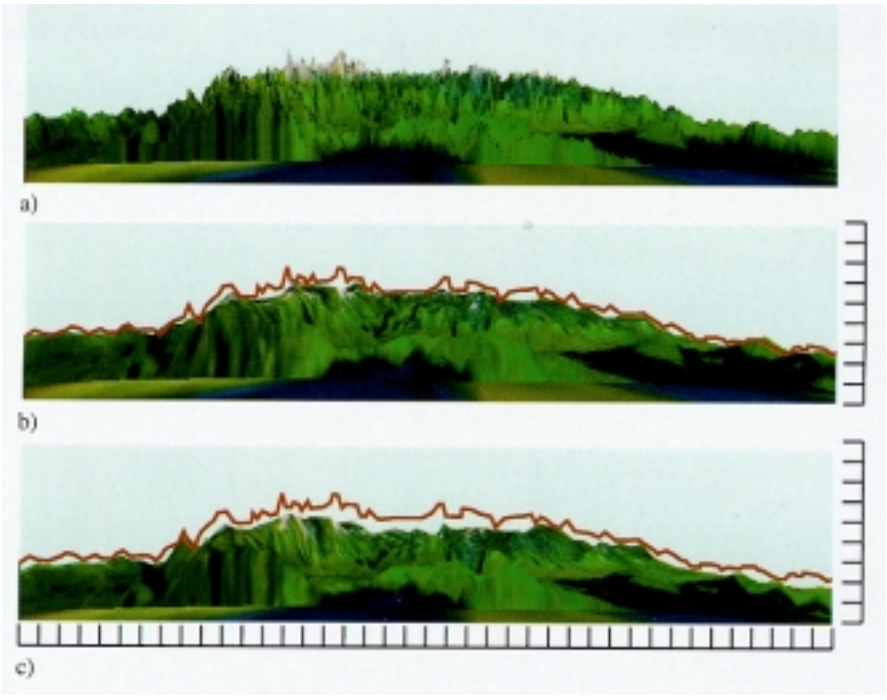

Figure 19: a) The original Alps. b) Alps remeshed with filter on twenty times coarser grid, which is illustrated below c) and on the right. c) Alps remeshed on the same coarse grid without filter.