Multi-scale extreme simplification and rendering of point sets

Reniers, D.

Award date:
2004

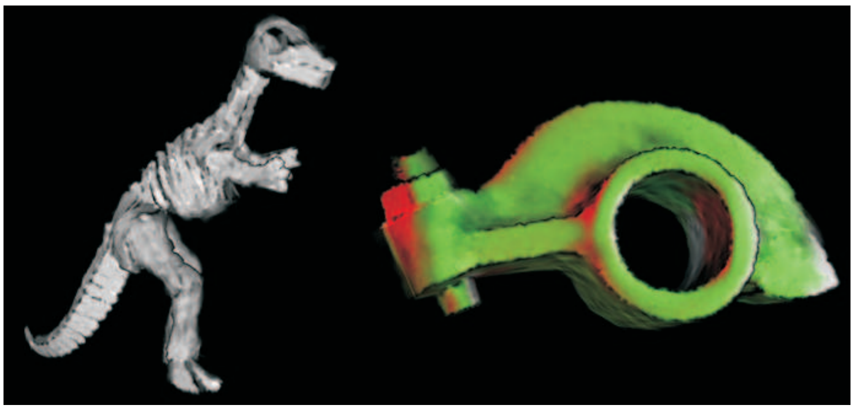
TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Multi-Scale Extreme Simplification and Rendering of Point Sets

by
D. Reniers



Supervisor: dr. ir. A. C. Telea

Eindhoven, November 2004

Contents

Preface

This master's thesis introduces a novel approach for simplification and rendering of point-set surfaces. The approach is especially suitable for extreme simplification, that is, when the model is rendered using only a few primitives (in the order of 100). The model's surface is first decomposed into low-curvature regions at a chosen scale, by using an algebraic multigrid algorithm (AMG). These regions are then efficiently rendered using textured polygons. An important feature of AMG is that the decomposition is "fuzzy": a point may belong to multiple regions at once. This degree of membership can be captured in the transparency map that is applied to the polygon. Rendering these probabilities yields a smoother image, at little extra expense. A non-trivial rendering algorithm is presented to render the polygons using blending. We demonstrate our technique on various models.

The graduation project for my studies in Computer Science at the Eindhoven University of Technology was performed from December 2003 to November 2004. I would like to thank Alexandru Telea for supervising my project. His unremitting enthusiasm and creativity were inspiring. Thanks to the proofreaders of this thesis for their comments and time. Most importantly, I want to thank my parents, my sister, and my girlfriend for their love and support.

Chapter 1

Introduction

1.1 Point-based rendering

Point-based rendering deals with rendering three-dimensional surfaces represented as point sets. A point set is a sampling of a continuous surface, be it real or virtual. Each point sample in a point set describes the original surface in a small vicinity around the point. In most cases, a point is comprised of a position, radius and normal. Depending on the representation, it can also be assigned color, shape, area, and curvature. Although the idea of using points for rendering surfaces dates back to 1985 [21], it is only recently that it received much attention. There are several reasons for this.

Recent 3D scanning technology delivers large point sets (up to 100 million points). Converting point sets to traditional triangle meshes is cumbersome and unnecessary when point sets can be rendered directly. Point sets are naturally rendered using point-based rendering, i.e. rendering each point sample by a point primitive. The idea is that the point primitive efficiently and effectively describes the appearance of the original surface around the point sample. Rendering the point set can be considered as reconstructing the original surface. Point-set representations are more compact than traditional triangle meshes, because they contain no connectivity information.

Complex 3D models are often created using higher order modelling primitives, such as splines and implicit functions. However, these cannot be rendered directly. They are broken down into triangles somewhere in the graphics pipeline instead. Point-based rendering is not limited to visualizing point sets; a triangle mesh may be point-sampled to create a point-set representation. Recent graphical hardware allows models to become so complex, that triangles occupy about one pixel when projected onto the screen. At that point, the efficiency of scanline rendering is lost, and it becomes more efficient to render even simpler primitives than triangles, namely point primitives.

Point-based modeling, related to point-based rendering, allows easier editing of the model than a polygonal representation, because there is no connectivity information that needs to be maintained. In this way, points can be easily inserted and removed from the model.

1.2 Problem statement

Our objective is to increase the performance of rendering point sets, while keeping the image visually convincing. The advantage of a more efficient rendering method for point sets is clear. The hardware requirements become less demanding, so that point rendering becomes more widely available, or more complex models can be rendered with the same hardware.

In many real-time applications, such as games, one wants to render a large number of models. The screen size of these models can, depending on the viewing parameters, often become quite small (e.g. less than 100 by 100 pixels). Rendering high-resolution models is in this case wasteful of resources. By reducing the complexity of the models we can achieve more efficient rendering. Moreover, the small screen size of the models gives us more leeway to compute the simplified models while keeping them visually convincing.

1.3 State of the art

Point-rendering algorithms try to reduce the amount of primitives they render to improve efficiency. They differ in the way they perform this reduction. For example, some algorithms use visibility culling while others exploit local surface characteristics. We will now discuss some relevant point-based rendering algorithms and hybrid rendering systems. We will also discuss a mesh-based approach called Billboard Clouds that has some similarities with our approach.

QSplat [1] is a tool for efficient rendering of very large point sets, using simple oriented disc primitives. It builds a bounding-sphere hierarchy of point samples. This enables QSplat to perform visibility culling, thereby reducing the amount of primitives rendered. The hierarchy is a *multi-resolution* organization, which allows level-of-detail control. With level-of-detail control, the tradeoff between image-quality and framerate can be dynamically adjusted during rendering.

The bounding spheres of the leaf nodes are made large enough to close all holes in the surface of the model. To render the model, the tree hierarchy is traversed from root to leaves. Traversal stops when either a leaf node is reached, the bounding sphere of the current node is not visible, or its projected screen space has become smaller than a certain threshold. A bounding sphere is rendered as a splat.

A *splat* is a representation of one or more sample points and can be rendered with a range of primitives. The simplest is the non-antialiased point. Better is an opaque disc, either rendered as a triangle fan or as a texture mapped square. The texture can have alpha values that fall off radially according to a Gaussian function. This is called a fuzzy spot and is the best and most expensive primitive. For each primitive, the orientation towards the viewer can be taken into account.

QSplat's point primitives are very simple, so that each primitive can be rendered fast in itself. However, because the number of parameters for these primitives is small, a primitive can only accurately describe a small part of the surface. A detailed representation of a model requires a lot of simple primitives, thereby diminishing the advantage of their simplicity. When the number of primitives is small, the resulting image looks bad.

Differential points [6] are rendering primitives that exploit local surface characteristics, in order to speed up rendering. A differential point (DP) is constructed from a sample point and has the position, principal curvatures and principal directions as parameters. The size of a DP is a function of its curvature. Areas of low curvature can be represented by larger DPs than high-curvature areas. After all DPs have been constructed, a simplification process is performed that prunes redundant DPs. A DP is redundant when its geometric information is already covered by other DPs. This simplification step (which can be seen as a resampling of the model) reduces the number of primitives to be rendered and thus speeds up rendering. For rendering, the DPs are quantized into 256 different types and each point is approximated by the closest quantized DP. A normal map of the normal distribution is applied to the rectangle, to mimic the normal variation around the sample point.

A differential point has more parameters than a QSplat point primitive so that it can capture the curvature around its sample point. Therefore, a differential point is able to approximate a larger area of the surface than a simple point primitive. The area is still limited however, and cannot grow arbitrarily. Consequently, the differential point approach does not yield a dramatic reduction in primitive count.

Surface splatting [8] focusses on high quality texture filtering. Surfels are point samples that can store pre-filtered textures. Surface splatting speeds up rendering by moving rasterization and texturing from the rendering pipeline to a preprocessing step. It is positioned between geometry rendering and image-based rendering. It uses a novel technique called visibility splatting. Forward warping is used to project surfels onto the screen, and the remaining holes are detected and filled using Gaussian filtering. Surfels work well for high-detailed surfaces, but not for flat surfaces.

Surfels have the same area limitation as differential points. Furthermore, the image-reconstruction technique for filling the holes is costly because it is not fully hardware-accelerated.

POP [10] is a multi-resolution, hybrid, point-and-polygon rendering system that is built on top of the QSplat system. It can be seen as a level-of-detail algorithm. Points are used to speed up the rendering of distant objects, while triangles are used to ensure the quality of close objects. POP uses the polygons of a triangle mesh as the leaves in the bounding-sphere hierarchy. Switching between points and triangles is determined on-the-fly based on their screen-projection size. QSplat has the problem that when models are viewed up close, the image becomes blocky because the samples are more than one screen pixel apart. Hybrid systems do not have this problem.

POP is similar to QSplat. It uses the same simple point primitives and hierarchical structure extended with triangle primitives. The triangles it uses come from the original mesh and are not generated by POP itself. They are only used for improved image quality when viewing up close. No effort is done to exploit local surface characteristics to speed up rendering.

Hybrid Simplification (HS) [11] is another multi-resolution hybrid system, which uses polygons at all levels in the hierarchy, while POP only uses them at the lowest level. The algorithm automatically determines which portions of the model are rendered with points and which with polygons, while considering the performance characteristics of both on the particular architecture. HS applies an edge-collapsing algorithm on the polygons to reduce the number of polygons, and performs point-replacement operations to reduce the number of point primitives. These simplifications are lead by a screen-space metric. The primitive reduction could be better if intrinsic features of the model were used [12]. Another limitation is that the screen-space metric does not account for color error, and is hence unsuitable for colored models.

Point to Mesh Rendering (PMR) [12] is a multi-resolution hybrid system which is similar to POP and Hybrid Simplification in that it also uses points and triangles adaptively. However, it is fundamentally different in generating the hierarchy. This has several advantages. PMR builds a multi-resolution hierarchy of points as well as triangles. This is unlike POP, in which only points are used at multiple resolutions. Secondly, the hierarchy is dependent on intrinsic, scale-independent features of the model. This is unlike the scale-dependent screen-space error-metric of Hybrid Simplification. The feature detection is based on Voronoi diagrams. The height of the Voronoi cell for a point measures the local feature size at that point and the ratio $\frac{radius}{height}$ gives an estimate of the sample density with respect to the local feature size. Thirdly, PMR does not require a polygonal mesh as input, which POP and Hybrid Simplification do require. Like HS, PMR is not prepared for colored models, because the triangles have a single color.

Billboard Clouds [15] is an approach that combines the strengths of mesh decimation and billboards. A billboard is a polygon with a texture applied that captures the 2D image of the model from one point of view. The algorithm takes a triangle soup as input, determines a set of viewpoints that is representative for the geometry, and projects the model's polygons onto the viewplanes associated with these viewpoints. It is especially suitable for extreme simplification, that is, when the model is simplified to only a few primitives (in the order of 100). This approach is similar to our approach, in that it renders complex models with only a few large and flexible primitives. This is in contrast to the other approaches mentioned above; their result is visually bad when few primitives are used.

A more extensive survey on point-based rendering systems is presented in [13].

1.4 Contribution and conceptual overview

As stated, we are interested in increasing the performance of point-based rendering. This can be achieved by either rendering simpler primitives or rendering less primitives. Point primitives are already very simple, so the second option seems the most rewarding.

The number of primitives can be reduced by resampling the point set with less samples. Low-curvature areas of the surface contain less information than high-curvature areas, and can thus be described by less point samples. When each sample is rendered by one primitive, the number of primitives is reduced. This approach is taken by DP. The total surface area of a point is bounded by the surface characteristics at that point. A drawback of this method is that it only takes local (fine scale) features of the surface into account. Global (coarse scale) features such as ridges and creases remain unexploited. Our approach should take global features into account, in order to minimize the number of primitives.

Surface splatting also reduces the number of primitives. It does not prevent holes in the rendering, but instead fills these holes with an image-reconstruction technique. Unfortunately, this technique is not hardware accelerated. We want to develop a fully hardware accelerated approach, in order to gain a maximum speedup.

Hybrid systems add polygons to the rendering. They come to the observation that points and polygons both have their strengths and weaknesses, and models can best be rendered using both. POP and HS use polygons to ensure that the rendering does not become blocky when viewed from up close. PMR also tries to exploit surface characteristics, something we also plan to do. Like PMR, we do not want to require a mesh as input. While one can always compute a mesh from the point sets, this is undesirable for large point sets. We may also want to develop a hybrid approach.

To satisfy the requirements above, we create a new primitive that exploits surface curvature characteristics in a global way. The primitive replaces a set of point samples that form a compact, low-curvature (or *quasi-flat*) area of the surface, called a *domain*. We will call our new primitive a *domain primitive*. Rendering a domain primitive should be hardware accelerated. Since our domain primitive is quasi-flat, we can exploit this to efficiently render domain primitives using polygons. To minimize the number of primitives, no limits should be imposed on the size of the primitive.

All algorithms discussed in the previous section provide an image-quality and framerate tradeoff; the framerate can be increased (by reducing the number of primitives) at the cost of a decrease in image quality or vice versa. In multi-resolution algorithms this can be adjusted dynamically during rendering. Our approach will offer a tradeoff that performs better when relatively few primitives are used, because a domain primitive can grow without losing much in accuracy. However, the relative complexity of domain primitives makes them relatively less efficient for replacing small parts of the surface, which is the case when a better image quality is required. The tradeoff difference between typical point-based approaches such as QSplat and our approach is depicted in figure 1.1. In this figure, the part of the axis between p_{low} and p_{high} indicates the number of primitives that is ideal for our approach. Left of p_{low}, the image quality is relatively bad compared to the point-based approaches. Right of p_{high}, the image quality increases slower and is worse than with point-based approaches.

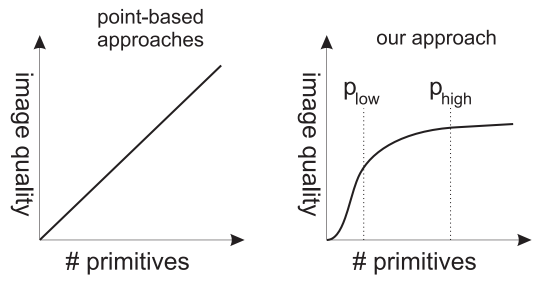


Figure 1.1: The difference between the image-quality and framerate tradeoff between typical point-based approaches and our approach. Note that the framerate is inversely proportional to the number of primitives

Before the domain primitives can be created, the surface must be decomposed into a number of domains, i.e. low-curvature areas. Therefore, the surface decomposition algorithm should take global surface features into account. Developing such a surface decomposition algorithm is a difficult task. We propose for this task the application of Algebraic Multigrid (AMG). AMG has already been used successfully on point sets in the past [16]. It will be discussed in detail in the next chapter.

A characteristic of the AMG decomposition is that its domains can be "fuzzy". This means that a point on the surface may (and usually does) end up in two or more domains. In detail, while two areas of the surface that are separated by a strong ridge end up in two disjunct domains, areas that cannot be clearly distinguished as being separated by strong curvature discontinuities may overlap. We can use this in our rendering by blending overlapping domain primitives with each other. Blending is a hardware-accelerated operation. We hope that blending enables us to improve rendering quality with little extra cost.

Figure 1.2 shows the general idea of our approach.

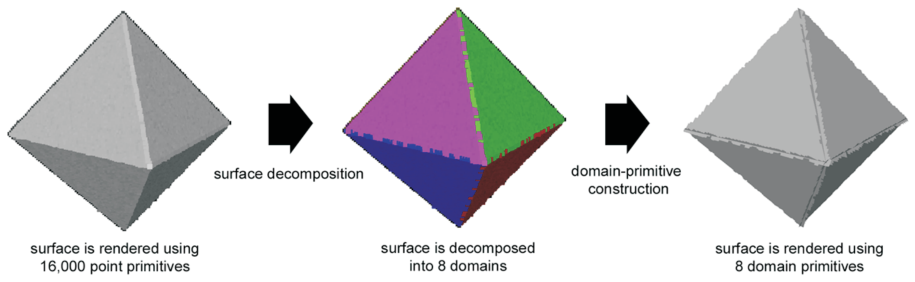

surface decomposition
domain-primitive construction
surface is rendered using 16,000 point primitives
surface is decomposed into 8 domains
surface is rendered using 8 domain primitives

Figure 1.2: A global overview of our approach. On the left, a model of an octahedron containing 16,000 point samples is rendered using point primitives. In the middle is the decomposition of the surface into 8 domains: the faces of the octahedron. The different domains are indicated by different colors. On the right, the surface is rendered using 8 domain primitives: one primitive for each domain. The advantage of our approach is clear; we render 8 hardware-accelerated domain primitives (i.e. 8 textured polygons) instead of 16,000 point primitives. Although a domain primitive is more complex than a point primitive and is a factor slower to render, this factor does not come close to 2,000.

1.5 Description of the remaining chapters

In this chapter, we have introduced the field of point-based rendering, discussed the state of the art of this field, and have given a brief overview of our approach. In the next chapter, we describe the surface decomposition algorithm and its prerequisites. After the decomposition of the model into domains, we discuss our new domain primitive in chapter 3. Chapter 4 deals with reconstruction of the surface by rendering the domain primitives. We present screenshots and statistics of our implementation in chapter 5. We give our conclusions in chapter 6. The software structure, build process and user manual are given in the appendices.

Chapter 2

Surface decomposition

2.1 Introduction

The input of any point-based algorithm is an unstructured set of points: $P = \{p_i\}$, let N be the cardinality of P. Each point sample is a tuple of coordinates, radius, normal and possibly color. The point set is a sampling of the original continuous surface S, which is a two-manifold. The output of our method is a visually plausible reconstruction of S by using P.

Many applications of point rendering need some local estimate of a surface's curvature. In particular, we need this information to be able to decompose the surface into quasi-flat areas. To calculate the surface curvature, we use a statistical method called PCA which we will use for other purposes as well. This method is explained in the next section. In section 2.3 we explain how the surface variation is calculated. In section 2.4 we discuss AMG, our tool for decomposing the surface into quasi-flat areas. We explain how the domains can be generated from these areas in section 2.5. At the end of this chapter, we identify the remaining degrees of freedom for our approach and make some design decisions.

2.2 Principal Component Analysis

Principal component analysis (PCA) is a statistical method used to find correlation among variables. In geometry, PCA on a set of points can be used to find an orthonormal basis that denotes the principal directions of variation.

Let Q be a set of points in $\mathbb{R}^3$, k the cardinality of Q, and $\bar{q}$ the centroid of Q. The 3×3 covariance matrix C captures the correlation between each two variates.

$$C = \frac{1}{k} \begin{bmatrix} q_1 - \bar{q} \\ \cdots \\ q_k - \bar{q} \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} q_1 - \bar{q} \\ \cdots \\ q_k - \bar{q} \end{bmatrix}, q_i \in Q \qquad (2.1)$$

Because this is a symmetric matrix, we can determine the eigenvectors, which are the principal directions of variation. Let $\vec{v}_0$, $\vec{v}_1$ and $\vec{v}_2$ be the eigenvectors of C. These can for instance be calculated by using Jacobi iteration over C [17]. The respective eigenvalues are called λ_0, λ_1 and λ_2, $\lambda_0 \leq \lambda_1 \leq \lambda_2$. The two vectors with the largest associated eigenvalues, $\vec{v}_1$ and $\vec{v}_2$, form a basis for the oriented tangent plane through Q, $\vec{v}_0$ is the normal of that plane. This is depicted in figure 2.1. If the three eigenvalues are equally large, this indicates that the tangent plane is not clearly defined and the points in Q form a sphere. When only one eigenvalue is large and the two other are small, this means that there is only one principal axis, thus the points in Q form a line.

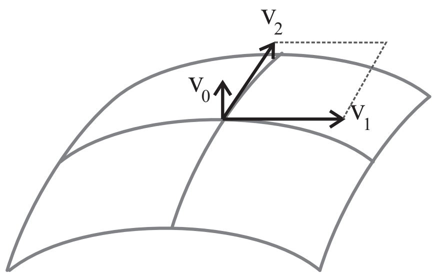


Figure 2.1: The two eigenvectors $\vec{v}_1$ and $\vec{v}_2$ with the largest associated eigenvalues form a basis for the tangent plane, $\vec{v}_0$ is the normal for that plane.

2.3 Surface variation

The curvature of the surface at a point p can be estimated by using principal component analysis. Let Q be the local neighborhood of a point p by taking the set of k-nearest neighbors and perform PCA on Q. Surface variation at a point p can be expressed by using the variation (λ_0) in the direction perpendicular to the tangent plane ($\vec{v}_0$), relatively to the variation of the points on the tangent plane spanned by ($\vec{v}_1$,$\vec{v}_2$). This can be expressed in the following formula:

$$\sigma_k(p) = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2} \tag{2.2}$$

When λ_0 is 0, this means that there is no variation of the points in Q along the normal $\vec{v}_0$. The points all lay on the tangent plane and the surface variation is said to be 0. The higher λ_0, the larger the variation along $\vec{v}_0$ and the higher the surface variation.

The size k of the neighborhood can be used as a discrete scale parameter. When k increases, more points in the neighborhood Q contribute to the surface variation estimate. Note that k cannot be increased arbitrarily. Eventually, points will be selected that do not belong to the same connected part of the underlying surface, which is a prerequisite. The size at which this happens is called the critical neighborhood size. It can be detected by looking for jumps in σ_k [2].

In figure 2.2, the surface curvature for a model is shown with four different neighborhood sizes. The color red indicates a low curvature (low σ_k), while a blue color indicates a high curvature (high σ_k). A small neighborhood size implies that the surface curvature is calculated very locally and makes it hard to see the surface's feature lines. A larger neighborhood size makes the surface variation calculation more stable. The feature lines are clearly visible as thick blue lines. Overall, this technique lets us robustly determine which areas of

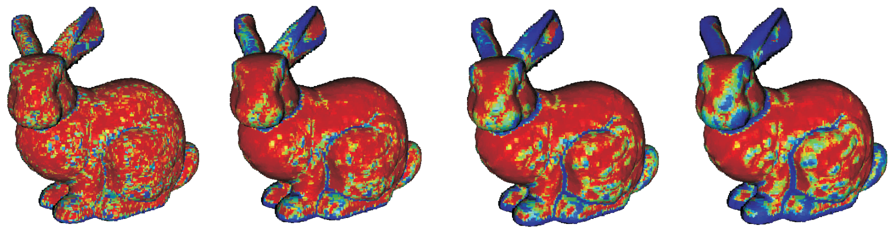


Figure 2.2: The surface variation is calculated four times, with the neighborhood sizes 10, 30, 50 and 80 from left to right. A lookup table is applied to visualize the values. Red values represent the lowest curvature, while blue values represent the highest.

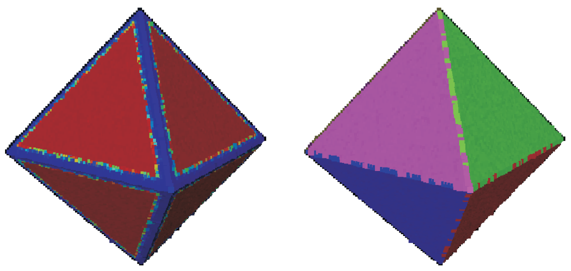


Figure 2.3: Left, the local surface variation classifier is shown for each point of an octahedron model. The classifiers serve as input for the surface decomposition algorithm that delivers the decomposition shown on the right.

the surface are flat and which are curved. This technique is similar to other techniques used in dealing with point sets [2, 19].

2.4 Algebraic Multigrid

In the previous section, we showed how we can locally classify a surface as flat or curved, based on the local surface variation. However, our goal is to decompose the surface globally into a number of quasi-flat domains, using the local surface classifier (see figure 2.3). We are going to use the Algebraic Multigrid technique [16] for this.

Algebraic Multigrid (AMG) can be used to decompose the model into quasi-flat regions that are separated by high-curvature ridges, at multiple scales. The term “quasi-flat” is used, because the regions are of low curvature in comparison to the ridges. AMG was originally used for accelerating the solving of large sparse linear systems of equations. It is a hierarchical and purely matrix-based approach. This makes it robust and applicable to many different problems, including geometrical ones. For our purposes, we can treat AMG as a black box; we are not concerned with its precise internal functioning. When the local surface variation classifier is used as its input, AMG can be used to decompose the surface at various scales, with the constraint that the regions are separated by high-curvature feature lines, and that those feature lines have been captured by the classifier.

The input of AMG is a (sparse) matrix M^0 of $N \times N$ elements: one element M_{jk} for each point pair (j, k). Each element M_{jk} describes how strongly points j and k are coupled, i.e. how similar they are. The elements for neighboring points are set to nonzero. More precisely, it is set to a function of the two surface variation detectors $\sigma(j)$ and $\sigma(k)$. This function is not further explained here, but one could take the average of the two detectors for example. See [16] for further details. Note that determining the neighbors of a point in a point-set is not trivial, as there is no connectivity information present. A sophisticated method for determining the neighbors of a point is presented in [7]. Non-neighboring points are considered not coupled and their respective matrix elements are set to zero. An example is given in figure 2.4.

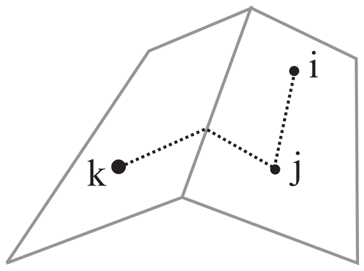


Figure 2.4: Here, M_{ij} is high because i and j are strongly coupled, i.e. are in the same quasi-flat region. M_{jk} is low because j and k are in two different regions separated by a ridge.

Application of AMG results in a set of matrices M^i. Each matrix M^i corresponds to a so-called *basis* B^i on the surface. A basis B^i consists of a set of basis functions $\{\varphi_j^i\}$ defined on the surface, i.e. $\varphi_j^i : S \rightarrow \mathbb{R}^+$. The basis functions in a basis sum to 1 at each point of the surface S. This is called the *partition of unity*. The initial coupling matrix M^0 we construct actually corresponds to the basis B^0, and thus a basis function exists for each point in P. For two strongly coupled points j and k, the basis functions for j and k will have overlapping bounded supports, where the *bounded support* of a basis function is defined as the points where the basis function is nonzero. In other words, the basis function φ_j^0 is 1 at point j and decreases with the distance until it is 0 at k. For two weakly coupled points, the bounded supports will not overlap. The basis functions in B^0 are linear affine functions. From our rendering perspective, each basis function approximates the surface in the vicinity around its associated point.

Now, for each scale i, AMG clusters the basis functions from basis B^i and forms the basis B^{i+1}. This clustering is based on the coupling between basis functions: only the strongest coupled basis functions are clustered. The initial basis B^0 we constructed has the smallest (in terms of support area) basis functions: the bounded support of a basis function does never reach further than the neighbors of its associated point. Essentially, at the first level, the matrix M^0 is a finite-element discretization of the classical diffusion stiffness matrix, weighted by the surface classifier σ (see [16] for more details). Each basis B^{i+1} that AMG produces has less functions than the previous basis B^i, because the basis functions from B^i are combined into B^{i+1}. This means that there are less basis functions, but with larger bounded supports. The AMG implementation that we use tries to reduce the number of basis functions between level i and $i+1$ by a factor of roughly 2. Because the initial couplings in M^0 created by us are based on the local surface variation classifier and because AMG clusters similar bases first, the subsequent couplings produced by AMG say something about the surface variation at larger scales. A basis B^{i+1} that is of a larger scale than B^i, is said to be *coarser* than B^i. Basis B^0 is the *finest* basis. The coarsest basis B^L will only contain a few basis functions, but with large bounded supports. For most models, L is set between 10 and 15.

Given the AMG properties stated above, it follows that a basis function φ_j^i defines a quasi-flat region j in scale i. For points where φ_j^i is close to 1, there is little overlap[1] with other basis functions, given the partition-of-unity property. This means that these points can clearly be categorized in one region and are separated from other regions by a strong feature line. Points for which φ_j^i is significantly smaller than 1 cannot be categorized clearly. Consider the decomposition of the octahedron in figure 2.5. In **a**, the bounded supports of the individual basis functions are indicated by different colors. The colors for overlapping basis functions are blended. Indeed, the yellow points do not blend with the purple points because they are separated by a strong feature line: a ridge of the octahedron. The purple and green points however do blend. These points cannot be categorized into either the green or purple basis function, because they are not separated by a feature line: they lie on the same plane.

One might wonder why the green and purple basis functions were not combined by AMG into one basis function that covers the entire side of the octahedron, like the yellow basis function. This is a “problem” of AMG. Although AMG guarantees that a region never crosses a ridge, it does not guarantee that several regions with a smaller area are combined to form one region with a larger area. This also depends on the scale; the two basis functions are probably combined at a coarser scale. With the final purpose of the basis functions in mind, namely rendering each region by one domain primitive, this is a disadvantage. Basis functions with a smaller support imply smaller regions, which in turns implies that more domain primitives are constructed and rendered than necessary. In section 3.4 we shall develop an algorithm to merge regions when possible.

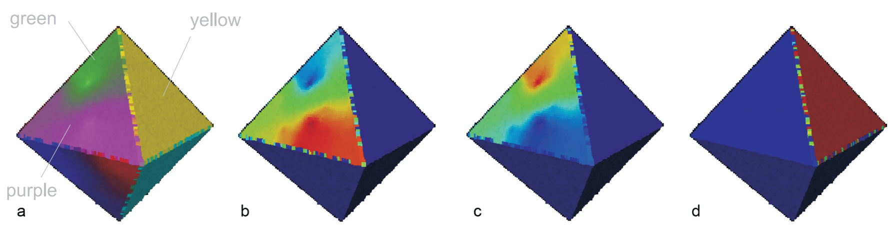


Figure 2.5: **a** is an AMG decomposition of the octahedron. Each color represents a different basis function. For domains that overlap, their colors are blended. The basis functions of the purple, green and yellow domains are depicted separately in **b**, **c** and **d** respectively.

In figure 2.6, the decomposition of the bunny model at the four coarsest scales is shown. Again, the different basis functions are denoted by different

[1]We say that basis functions overlap when their bounded supports overlap, i.e. when the supports have at least one point in common.

colors and colors are blended for supports that overlap. As can be seen, the basis functions blend with each other at parts where they aren't separated by a feature line that has been detected by the surface variation classifier. For example, the green and yellow domains in figure 2.6.**a** blend at the back of the bunny, because there is no feature line on the back. However, the same two domains are strictly separated at the bunny's legs, because a strong feature line is present.

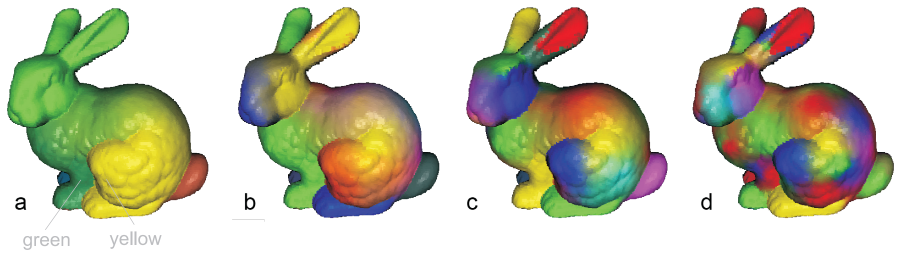


Figure 2.6: The four coarsest bases of the bunny, **a** being the coarsest.

2.5 Creating domains from regions

The idea is to create a domain primitive for each basis function. Each basis function defines a quasi-flat region and is thus suited for approximation by a domain primitive. Each basis function is, strictly speaking, defined on the whole surface S. However, especially on the finer decomposition scales, the basis function will be 0 or almost 0 for a large part of the surface. For the construction of a domain primitive, we want to limit the number of points we use from P. Otherwise, the bounding rectangle will become larger than necessary, wasting resources accordingly. For each basis function, we associate a *domain* D which is a subset of P. The obvious way to define D is by setting a point-select threshold K:

$$D_j^i = \{p \mid \varphi_j^i(p) \geq K\} \tag{2.3}$$

However, when K is higher than 0 and we reconstruct the signal by using the domains instead of the basis function, the partition of unity is violated. The sum of the basis functions at each point p will be less than 1, resulting in a loss of *energy*. Ultimately, this will make the final rendering darker than it should be. K can be chosen such that the energy-loss is not or barely noticeable. An empirically-found typical value of K is 0.01. Domains generated by using a threshold K are called *thresholded domains*. For the sake of discussion, we assume that partition of unity still holds for the rest of this document, although it is slightly violated by the threshold K. We define $\bar{\varphi}_j^i$ to be the *bounded* basis function:

$$\bar{\varphi}_j^i(p) = \begin{cases} \varphi_j^i(p) & \text{if } p \in D_j^i \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

There are other ways to define the domains. *Strict domains* are defined by maximizing the basis functions:

$$D_j^i = \{p \mid \forall_k \varphi_j^i(p) \geq \varphi_k^i(p)\} \tag{2.5}$$

The four coarsest strict domains of the bunny are shown in figure 2.7.

Of course, there is a lot of energy loss with strict domains. Reconstructing the surface using strict domains is therefore more complex (and thus less elegant) than when using thresholded domains. We will therefore use thresholded domains in the rest of this document, unless otherwise noted. Also, when we write φ we shall mean the bounded basis function $\bar{\varphi}$.

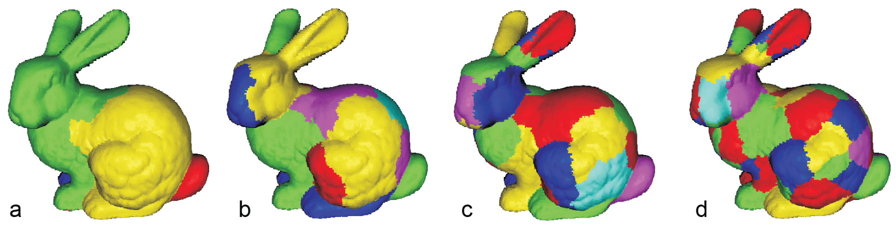


Figure 2.7: The four coarsest strict domains of the bunny.

2.6 Degrees of freedom

So far, we have presented our idea of decomposing the surface into quasi-flat domains. Recall that our goal is to render a point set faster than normal point-based rendering. We want to achieve this by rendering less primitives. We therefore decompose the surface into quasi-flat domains, that are suitable for approximation by our new domain primitive. However, there are still some choices, or degrees of freedom, that we have not examined yet. In this section, we identify these degrees of freedom to show the flexibility of our solution and we outline the tradeoffs. We make design decisions for every degree to narrow down our solution.

2.6.1 Scale selection

AMG gives a complete decomposition of the surface at a number of scales. The question arises for which scale or scales we construct domain primitives. Using domains from more than one scale might be advantageous. For example, AMG does not always deliver the largest possible domains, as stated in section 2.4. We could use one domain at a coarser scale to substitute several domains at a finer scale. Despite this advantage, we use only one scale, because otherwise we have problems to maintain partition of unity. When we would mix domains from different scales, we have to make sure that the partition of unity is not violated, because it is only guaranteed for the basis functions within one scale.

Now remains the question which one scale we use. A finer scale means more basis functions and thus more domain primitives. The approximation of the surface is generally more precise with more primitives, but slower to render. This is exactly the image-quality and framerate tradeoff that we spoke of in section 1.4. We provide the scale choice as a parameter for the user to enable image-quality and framerate tradeoff control. Of course, this means that the tradeoff is limited by the possibilities of AMG. For example, the tradeoff-control granularity is determined by the number of scales AMG can produce.

2.6.2 The domain primitive

We have stated some requirements of the domain primitive in section 1.4. It should be fast to render, arbitrarily sizeable, and it should adequately describe flat and low-curvature areas of the surface. The planar polygon is a representation that fits these requirements perfectly. It can describe large flat domains with only a few vertices and it is supported by graphics hardware. Textures can be applied to the domain primitive to capture colors, transparency and normals. Although the normal probably does not vary much over a domain, it is a good idea to capture the normal variation, because the human eye is very sensitive to shading discontinuities. Polygons can take any shape, but complex shapes require more vertices. Shape can also be modelled by using transparency values 0 and 1. Parts of the polygon that have a transparency of 0 are transparent, while the other parts are opaque. This requires less vertices at the cost of a texture in which the transparency is encoded. Since we are going to use a texture anyway to capture colors and possibly normal information, adding an extra channel for transparency (called an alpha channel) introduces little extra cost and is easier to implement than modelling shape geometrically.

To better capture the surface curvature, we could use multiple polygons to represent a domain. When using multiple polygons, we need an algorithm to subdivide the domain and represent each part by a polygon. However, the AMG algorithm should take care of this when the user selects a finer scale. We therefore use only one polygon per domain.

2.6.3 Approximation error

We can define an approximation error for domain primitives. By the approximation error, we understand the difference between the representation of the domain using the domain primitive and the original surface sampled by points in that domain. The error can be defined either in screen (2D) space or in object (3D) space. We will now discuss three error metrics that are often used in computer graphics.

The most relevant metric, from a rendering perspective, is the *visual difference.* The visual difference is a screen space error-metric that somehow compares the image of the original domain rendered by point samples and the image of the domain primitive. It measures shape, color and shading errors. This measure is not feasible in real-time. Furthermore, it is difficult to implement as it is very hard to quantify the error. For example, the error is not simply proportional to the amount of different pixels between the two images.

The *geometrical distance* is an object-space metric and only measures shape errors, by calculating the geometrical distance between the original and approximated model. This metric is feasible in real-time. The *screen space distance* is similar to the geometrical distance, but also takes the viewer into account by measuring the projected geometrical distance. This makes it a screen space metric. The screen space distance is also feasible in real-time.

For our purposes, we only measure the geometrical distance. This error metric can be used by the program to determine which domains cannot be accurately approximated by one domain primitive. The program can then choose to treat these domains differently, for example by subdividing them into multiple domains (see section 3.5). How the geometrical distance is calculated is explained in section 3.2. The screen-space distance is not very useful to us, because we have chosen to use only one AMG scale. When more AMG scales would be available, the domain primitives from different scales could dynamically be mixed during rendering. This would create an interesting level-of-detail algorithm. However, a solution must be found to maintain the partition of unity when different scales at the same time are used. Because our time is limited, we refrain from going in this direction. For the same reason, we do not use the difficult to evaluate visual-difference metric.

Chapter 3

Domain Primitive

In the previous chapter, we have described how we decompose the surface S into domains. In this chapter the construction of the domain primitives is described. For each domain D_j^i, we can independently construct a domain primitive, comprising a support polygon and a texture. In the remainder of this chapter, we fix the scale i and the basis function index j and use the notation D instead of D_j^i.

3.1 Creating the support polygon

In all discussion so far, we have stated that the domains D_j are quasi-flat. For rendering purposes however, we must replace them by purely flat polygons, because there are no curved primitives that are hardware accelerated. This section discusses how we construct a polygon for a quasi-flat domain, by means of projection. The support polygon will serve as a support for the texture, that will capture the color and basis function.

First we have to choose a plane T on which the points in D can be projected best. Information is lost when projecting points. To minimize this loss, we project the points onto a plane that is perpendicular to the axis that has the least information (the axis defined by eigenvector $\vec{v}_0$). This makes the plane locally tangent to the original surface S. PCA can be used to find this axis. The position of the plane is given by the centroid already calculated by PCA (see section 2.2). However, the points in D are not equally important to the domain primitive. Indeed, the value of the associated basis function at a point p can be interpreted as the importance of point p for domain j. The coordinates are weighted before PCA is performed to take this into account. See figure 3.1 for the difference between weighted and non-weighted PCA. The points in D are then projected onto the tangent plane T. The projected set of points is called D'.

The normal of the plane could be either $\vec{v}_0$ or $-\vec{v}_0$, PCA does not decide this. To resolve this, the weighted average $\vec{w}$ of the normals of the points in D is calculated. When $\vec{w}.\vec{v}_0 > 0$, $\vec{v}_0$ is taken as the normal and $-\vec{v}_0$ otherwise.

When projecting the points, the radii can also be projected. Let r_p be the radius of point p and $\vec{n}_p$ the normal. The radius r_p then becomes $(\vec{n}_p \cdot \vec{v}_2) r_p$. The impact of projecting the radii is minimal, as most point samples are tangent to

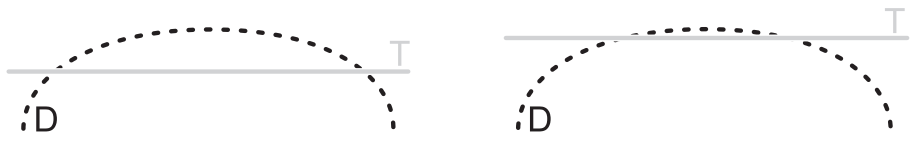


Figure 3.1: The plane T resulting from non-weighted PCA on the points in domain D is shown on the left. The result of weighted PCA is shown on the right.

the plane. When point samples are projected, their density may change. This may have an impact on texture construction (see section 3.3.2). Projecting radii can be useful to prevent the increase in density. However, it may also introduce holes because the radii are decreased. Therefore, we leave projecting the radii as an option for the user.

Next, the support polygon can be constructed. The polygon must lie on T and enclose the points in D'. Although we could calculate an n-sided bounding polygon that exactly encloses the points, we have chosen for the much simpler calculation of a bounding rectangle. Finding the optimal bounding polygon requires calculation of the convex hull, which is difficult. Calculation of the bounding rectangle only requires PCA, which we have already implemented. Because the bounding rectangle has a larger area than the bounding polygon, a larger texture must be applied, hence wasting more resources[1]. An advantage of the bounding rectangle is that it requires less vertices than the bounding polygon.

We use PCA to determine a two-dimensional orthonormal basis that has the centroid of D' as its origin. We project the points D' onto the two axes and determine the maximum and minimum components on both axes. The two sides of the bounding rectangle are aligned to the axes and sized so that it encloses the maximum and minimum components. Note that the construction of the optimal bounding rectangle actually requires calculation of the convex hull. In practice, using our PCA method yields a solution that is only slightly sub-optimal. Because PCA condenses the information, the largest side of the bounding rectangle is made as large as possible while the smallest side is made as small as possible. This yields a bounding rectangle with the smallest area.

Figure 3.2 shows the steps involved in the construction of the support polygon.

3.2 Calculating the approximation error

The domain primitive is an approximation for the original surface at the domain D. We introduce an error by approximating the domain D by the projected domain D'. In other words, we introduce an error by approximating a low-curvature part D of the surface by a planar polygon. In section 2.6.3 we decided to use the geometrical-distance metric. The error can be determined by calculating the geometric distance between D and D'. That is, we compute

[1]This disadvantage may be diminished by the fact that recent hardware can adequately compress "empty" parts of textures. However, note that this requires an adequate implementation and is not performed automatically.

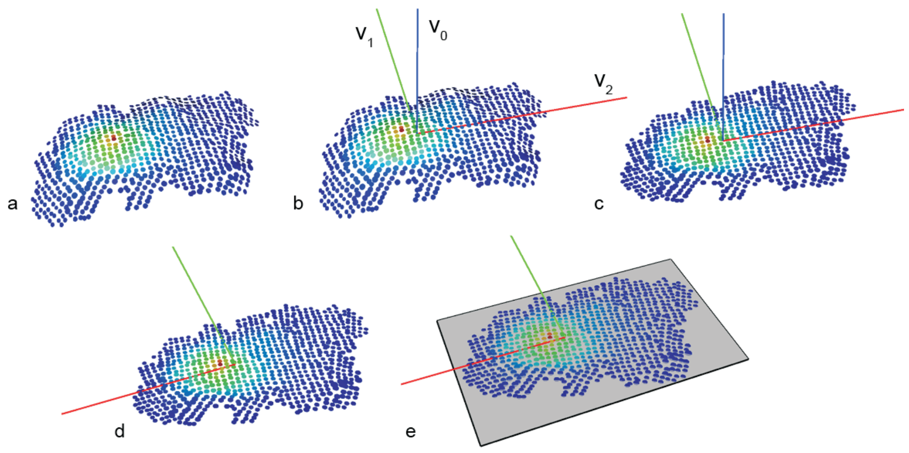


Figure 3.2: **a** shows the points of a domain. PCA is done to find the principal axes. These axes are shown in **b**. In **c**, the point samples are projected onto the plane that is defined by the axes v_1 and v_2. Then, PCA is performed again but now on the projected points. The result is shown in **d**. All the point samples are projected onto the two axes to find the minimum and maximum components on both axes. These components determine the extent of the bounding rectangle. The bounding rectangle is shown in **e**.

the average distance between each point $p \in D$ and its projected counterpart $p' \in D'$.

A similar metric that is simpler to calculate is the curvature of the domain D. The curvature can be measured in the same way as the local surface variation was calculated in section 2.3: $\frac{\lambda_0}{\lambda_0+\lambda_1+\lambda_2}$. The eigenvalues can be taken from the PCA calculation that has been performed to find the local tangent plane in the previous section. We use the first metric, as it is more accurate.

3.3 Texture construction

3.3.1 Preliminaries

We have so far shown how to construct a flat, rectangular polygon that approximates our quasi-flat domain for rendering purposes. Now, we must somehow transfer the remaining information from the domain to the polygon. This information comprises point colors, normals and basis function values. To efficiently encode this information in our domain primitive, we use a (single) 2D texture on the domain primitive's polygon.

A *texture* is a regular grid of texels, where *texel* is a square-sized texture element. A texture may consist of multiple *channels*. Each channel assigns a value (e.g. a scalar or vector) to each texel. We propose using an alpha channel to store the discretization of the associated basis function and a color channel to capture the colors of the point samples. The use of the alpha channel is justified by our proposed method of blending domain primitives (see section 1.4). If desired, more channels can be used to store other information, such as normal information. However, note that the availability of such extensions is subject to the graphics hardware support. Because our hardware did not support normal maps, we did not implement them.

In essence, a texture is a regular sampling of one or more 2D functions. The midpoints of the texels are the sample points. Hence, the texture's resolution determines the sampling rate. So far, the texture's sampling is not correlated with the projected point-samples. For constructing our domain primitive's texture we need to do a resampling of the discrete, non-uniformly sampled surface defined by the projected points D' (see figure 3.3). The alpha channel will capture the basis function φ that is associated with D'. The color channel will capture the colors of the point samples in D'.

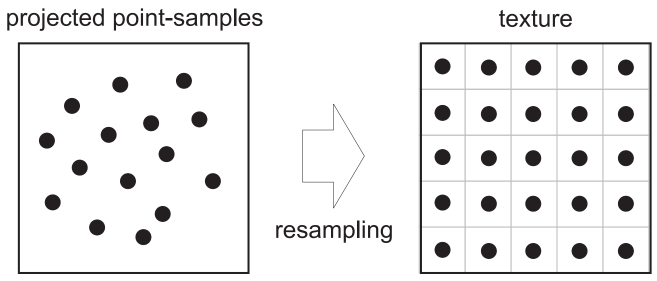


Figure 3.3: Resampling the projected point-samples to a regular sampling.

Essentially, the goal of the resampling process during texture construction is to produce a texture which, when rendered, conveys the same impression as the original point-samples. In other words, the signal we reconstruct by rendering the 2D texture must closely match the original signal we would reconstruct by rendering the 3D point samples. It follows that an important choice in our resampling strategy is the basis function set we use in this reconstruction.

During rendering, OpenGL performs a viewpoint-dependent resampling of texels to pixels. The basis functions used during this reconstruction are fixed by graphics hardware and can either be linear or constant. We discuss this in more detail in section 3.3.4. However, we *can* choose the basis functions used during texture construction. We propose two kinds of basis function: radial and linear affine. They are discussed in the following two sections. Since we compute the 2D texture out of the *projected* 3D points, we will actually consider the basis functions reconstructing those projected points. For the sake of conciseness, we use the word "texel" when we in fact mean the sampling midpoint that represents the texel.

3.3.2 Radial basis functions

The first resampling method we propose here uses radial planar basis functions around the 2D projected points in D'. We center a basis function $\phi_p : \mathbb{R}^2 \rightarrow [0..1]$ (do not confuse these basis functions with those from AMG) at each point $p \in D'$. We consider three basis function shapes. We say that d is the distance between texel t and the projected point sample p, r_p is the radius of p, and w is the so-called "width factor", that can be chosen by the user. The product wr_p is called the basis function width or support size. The three different basis function shapes are:

- constant: $\phi_p(t) = \begin{cases} 1 & d \leq wr_p \\ 0 & d > wr_p \end{cases}$

- linear: $\phi_p(t) = \begin{cases} \frac{wr_p - d}{wr_p} & d \leq wr_p \\ 0 & d > wr_p \end{cases}$

- Gaussian: $\phi_p(t) = e^{-d^2/2\sigma^2}$. As a rule of thumb, the Gaussian can be considered 0 when $d = 3\sigma$, for the value is only 0.01 then. Therefore, we choose $\sigma = \frac{1}{3}wr_p$.

Using a higher degree basis functions results in a smoother reconstructed signal and thus in a smoother image. However, texture construction using Gaussian basis functions takes considerably longer than when using one of the other two, due to the costlier computation of the Gaussian. In section 5.2.4 we evaluate this image-quality and speed tradeoff in texture construction.

We construct the signal f by summing all basis functions: $f(t) = \sum_p \phi_p(t)$. The function f is supposed to be 1 at points that lay within the domain and be 0 for points that lay outside the domain. At the border of the domain, f decreases slowly or suddenly, depending on the chosen basis function. Figure 3.4.**a** shows a simplified 2D view of a domain containing three point samples p_1, p_2 and p_3 with linear basis functions assigned. In figure 3.4.**b** the signal f is shown.

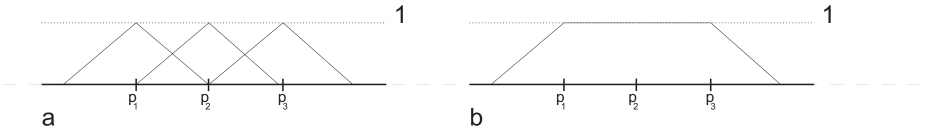


Figure 3.4: Linear basis functions are placed on three point samples in **a**. The function f is the sum of these basis functions and is shown in **b**.

We can easily construct a color for a texel t by scaling the colors of point samples by their basis function value $\phi_p(t)$. In this way, for each t we effectively interpolate the colors of point samples in the vicinity of t. The vicinity size is determined by the support size of the basis functions. The basis function value $\varphi(t)$ that is needed for the alpha channel can be constructed likewise. To construct the color channel $c(t)$, the equation becomes:

$$c(t) = \sum_p (\phi_p(t) \cdot g(p)) \tag{3.1}$$

where $g(p)$ is the color of point sample p. To construct the alpha value $a(t)$ for each texel t of the alpha channel we calculate:

$$a(t) = \sum_p (\phi_p(t) \cdot \varphi_j(p)) \tag{3.2}$$

It is important that f is exactly 1 within the domain (i.e. the partition of unity must be obeyed), so that the color of a texel is an interpolation between the colors of surrounding point-samples and the alpha value is an interpolation of their basis function values. What can also be seen in equation 3.1 is that the width of the basis function determines the amount of blending between colors of neighboring point-samples. When we calculate the color for a texel t that is at the same position as projected point-sample p, we expect the color to be $g(p)$. Substituting p for t in equation 3.1, we get $c(p) = \sum_p (\phi_p(p) \cdot g(p))$. When we choose ϕ_p so that it is 1 at its associated projected point-sample p and 0 otherwise, we have indeed that $c(p) = g(p)$. However, when we choose a higher width-factor in order to increase the basis function width, the color $g(p)$ might well be interpolated with colors of other point samples. A problem is that we cannot exactly choose ϕ_p so that it is 1 at p and 0 at its neighbors, because we use radial supports. We can guarantee this when the support of a basis function explicitly makes use of the neighboring points, something we will do in section 3.3.3 by the use of linear affine basis functions.

We have seen that it is important that f is exactly 1 within the domain. Unfortunately, this cannot be guaranteed. When for example a constant function is chosen as the shape of the basis function in figure 3.4, the signal would not be 1. The same holds when the width factor of the linear basis function would be different. Another reason is that point samples do not need to be regularly spread across the surface. Although the point density is the same in general, it can differ locally. The QSplat framework that we use (see also section 5.1) guarantees that the surface of the model is completely covered by point samples and that no holes appear when rendering them using point primitives. It does not guarantee that the coverage is efficient; there may be points at the surface that are covered by more point samples than necessary. Furthermore, the projection from D to D' causes the density to change for parts of the domain that are not parallel to the tangent plane. Figure 3.5.**a** shows a domain containing four points p_1 to p_4. Because point samples p_3 and p_4 happen to be close together, the resulting signal f contains a bump.

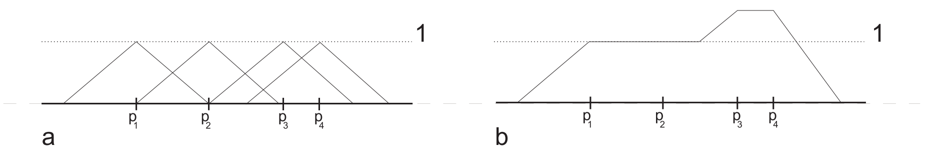


Figure 3.5: The irregular density around p_3 and p_4 causes a bump in f (**b**).

We will ensure that f is 1 by normalization. Normalization per domain means that the whole function is scaled down or up. Because the extremes of f are preserved by this scaling, this does not make $f(t) = 1$ for every t. Therefore, we normalize per texel. We can use two different criteria for normalization, both of which have their own advantages and limitations:

- Normalize when $f(t) \geq 1$. Equation 3.1 then becomes:

$$c(t) = \begin{cases} \sum_p (\phi_p(t) \cdot g(p) \cdot \frac{1}{f(t)}) & \text{if } f(t) \geq 1 \\ \sum_p (\phi_p(t) \cdot g(p)) & \text{otherwise} \end{cases} \tag{3.3}$$

  For the alpha channel equation, replace $g(p)$ by $\varphi_j(p)$.

- Normalize when $f(t) > 0$. Equation 3.1 then becomes:

$$c(t) = \begin{cases} \sum_p (\phi_p(t) \cdot g(p) \cdot \frac{1}{f(t)}) & \text{if } f(t) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

For the alpha channel equation, replace $g(p)$ by $\varphi_j(p)$.

For constant basis functions, these two options make no difference: $f(t) > 0 \Rightarrow f(t) \geq 1$, because $\phi_p(t)$ is either 0 or 1 when ϕ is a constant basis function. For linear basis functions, the first option yields a function f that decreases at the border, while the second makes f fall down immediately from 1 to 0 at the border of the domain. Furthermore, for the first option the basis function must be chosen wide enough so that $f(t) \geq 1$ for points that fall within the domain. This depends on the minimum overlap between point samples, which in turn depends on the model. The wider we choose ϕ, the larger the energy of f. This causes the partition of unity to be violated in the final rendering. Also, the blending between original point-sample colors increases. The second option normalizes even the smallest value of f to 1, so when the basis function is as wide as the point sample we do not have this problem. See figure 3.6 for an example.

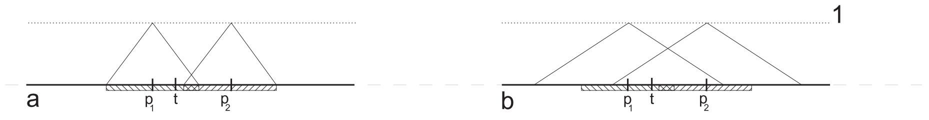


Figure 3.6: Two point samples p_1 and p_2 are shown. The two radii are indicated with a different hatching. An arbitrary texel t that falls within the domain is also shown. In **a**, the basis function width is chosen as large as the point's radius. In this case this means that $f(t) < 1$. In **b** it is chosen larger, so that $f(t) \geq 1$. However, the basis functions now extend over the domain's border. This causes the domain primitive to represent the domain as if it were larger than it in fact is.

For Gaussian basis functions, criterion 3.3 makes f decreasing at the border of the domain, as is the case with the linear basis functions. However, when normalization is done for $f(t) > 0$ the function f becomes 1 for every texel t, because $\phi_p(t) > 0$ for every t when ϕ is a Gaussian. This may be useful to cover every texel by at least 1 basis function, so that each texel can be assigned a color (see section 3.3.4).

In this section, we have discussed texture construction using radial basis functions. In the next section, we will discuss texture construction using linear affine basis functions. In figure 3.7, the visual difference between radial basis functions in **a** and linear affine basis functions in **b** can be seen. Most noticeable is the reduction of over-bright areas in **b** when compared to **a**. We will elaborate on these differences in section 5.2.4.

3.3.3 Linear affine basis functions

In the previous section, the normalization problems come from the fact that we cannot choose ϕ_p so that it is 1 at p and falls down to 0 at its surrounding point samples. This requires a differently shaped basis function for each point sample p. In this section we propose to construct a linear affine basis on the 2D plane containing the point projections in D'. This section presents a resampling method that creates these basis functions.

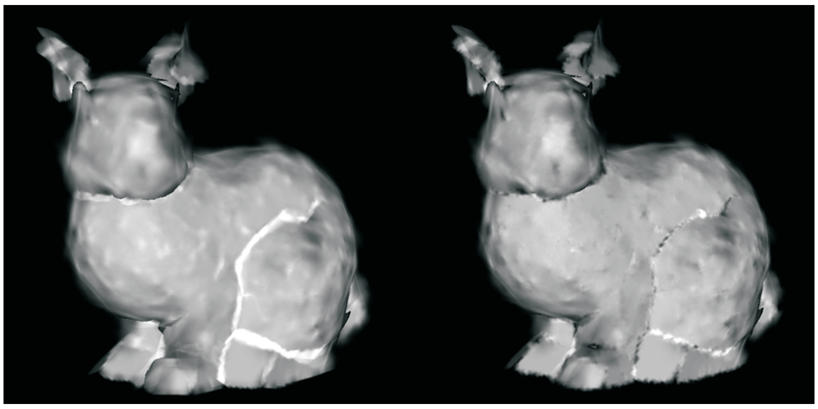


Figure 3.7: In **a**, textures are constructed using radial basis functions, with a width factor of 2. In **b**, the textures are constructed using linear affine basis functions.

We create a Delaunay triangulation using for the projected points in D'. For this we use the Delaunay triangulator [20]. Then, for each texel t we determine in which triangle it falls and linearly interpolate the colors of the point samples at the corners of the triangle to come to a color for t. This is similar to the Gouraud shading technique often used in computer graphics, or the linear affine basis functions used in finite element applications. The alpha value can be found likewise by interpolating the basis function values φ_j at the corners.

As is, this method has two problems. The first is that the triangulation algorithm also fills concavities in the domain by triangles, although points within these triangles should not be considered within the domain (see figure 3.8). We can solve this problem in several ways. The first method is by requiring that the texel is within the radius of at least one of the point samples at the corners of the triangle the texel falls in. Another way to solve this is by removing triangles that are too big, that is, when one of their edges is larger than the sum of the two incident point's radii. We implemented the latter method, because it yields less triangles. This saves texel-in-triangle tests and thus increases performance. Also, it makes hardware acceleration more suitable (see section 3.6).

The other problem is that texels that fall just outside the triangulation do not lie within a triangle, but may lie within the radius of a point sample. The texture may thus represent a domain that is smaller than the original domain. We solve this by using radial linear basis functions for those texels that fall outside the triangulation but within at least one point's radius.

To summarize, texels that fall within the triangulation are assigned a color and alpha value that is a linear interpolation between three point samples using three linear affine basis functions. For texels that fall outside the triangulation, but within at least one point sample's radius, radial basis functions are used because we do not have three point samples to interpolate between.

An advantage of this method as compared to the radial basis functions is that it has no parameters to be set by the user, whereas for the radial basis functions method the width factor must be chosen.

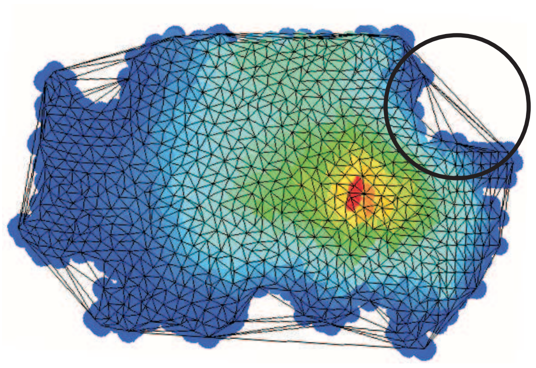


Figure 3.8: A triangulation for a domain is shown. The point samples are displayed with proportionally correct radii. A concavity of the domain is encircled for which triangles were constructed by the triangulation algorithm.

3.3.4 OpenGL texture filtering

While rendering textures, we can tell OpenGL how to reconstruct signals from the texture and alpha channels, as outlined at the end of section 3.3.1. This is the final resampling pass: from texture space (texels) to screen space (pixels). This resampling is called *texture filtering*. OpenGL can perform linear or nearest-neighbor texture-filtering. Linear texture-filtering uses linear basis functions, while nearest-neighbor filtering uses constant basis functions. Where linear filtering effectively maps several texels to one pixel, nearest-neighbor filtering maps only one texel to a pixel, and is therefore faster. Linear filtering is preferred because the slight performance loss does not weigh up against the increased image quality.

However, linear filtering may pose a problem in our approach. The texels of the color channel are assigned no color (or black) at points that are sufficiently far from points in D', because they are not influenced by any point sample. As a result, OpenGL will linearly interpolate colored texels with black texels at the border of D', resulting in black borders. However, because in practice the transparency often increases near the borders of D' (because the basis function value always decreases near the border of the domain), this problem may not be noticeable in the final image. The problem is more noticeable for strict domains, because they fall off more sudden at their borders. Recall however that we do not use strict domains in our final reconstruction. Both methods can be adapted so that they assign a color to each texel.

For the radial basis functions, we can solve this problem in the following way. We choose basis functions that do not fall off to 0, but extend over the whole polygon so that every texel is influenced by a basis function (in fact, influenced by every basis function). We normalize when $f(t) > 0$ so that each texel is an interpolation between point samples.

For the linear affine basis functions, we assign to a texel the color of the closest point sample if the texel does not fall within the radius of a point sample.

3.4 Domain merging

Recall from section 2.4 that AMG does not necessarily deliver the largest (in the sense of support area) basis functions possible. AMG may be reluctant to cluster small basis function into larger ones, to meet the requirement that the number of basis functions for a basis is reduced by a factor two for the next coarser basis. This means that for each basis there may be basis functions that can be merged without increasing the approximation error, although this is more likely to happen at a finer basis. In this section, we develop an algorithm for merging two domains, or equivalently, two basis functions. For the sake of simplicity, we only merge two domains at a time in each iteration of our algorithm. By doing multiple iterations, we may merge more than two domains to form one large domain.

Three criteria for merging two domains are used. First of all, the two domains must overlap ($D_j \cap D_k \neq \emptyset$). This ensures that the merged domain is compact, like the domains that AMG delivers. The two other criteria are that the domains must be flat and that they face in the same direction. Both criteria ensure that the resulting domain has a minimal approximation error. To measure the flatness of a domain, the simple curvature metric from section 3.2 is used. Thresholds for these last two criteria are empirically found and can be adjusted by the user when desired.

Merging two domains is done by summing their basis functions. Let φ_l be the merge of φ_j and φ_k. For φ_l holds that $\varphi_l(p) = \varphi_j(p) + \varphi_k(p)$. Note that no renormalization of the sum is needed, because the basis functions delivered by AMG obey the partition of unity. Each iteration, our greedy algorithm merges the two domains for which the profit is most, namely for which the overlap is the most. When the absolute overlap (the number of shared point samples) is large, the merging is more worthwhile than when the absolute overlap is small. Indeed, the points that are represented in both domains beforehand, are only represented in one domain afterwards. Therefore, the area of the merged domain primitive is smaller than the combined areas of the two domain primitives. This reduction improves rendering performance. Iteration stops when one of the criteria drops below its threshold.

In figure 3.9 an example is shown for the rockerarm model.

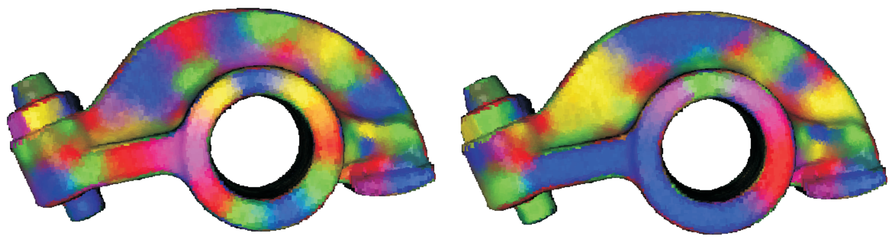


Figure 3.9: An example of domain merging. On the left, the basis functions before merging are indicated by different colors. The basis functions on the right are the result after merging. As can be seen, the number of basis functions is reduced at flat parts of the model.

3.5 Domain subdivision

Subdividing a domain into multiple domains can be used to decrease the approximation error. The criterion for subdividing a domain is thus that the approximation error exceeds a certain threshold. However, subdividing is more difficult than merging domains. In how many parts should the domain be subdivided, along which lines, and how should the basis function be divided among those parts?

Running AMG at a finer scale for the whole surface is of course possible, but this decreases the approximation error in general and not just for the domain under consideration, because a whole new decomposition is made by AMG. Furthermore, this possibility is already used as the image quality and framerate control for the user (see section 2.6.1). We want a more refined way to subdivide domains. One idea would be to run AMG for the target domain but on a finer scale. Unfortunately, this is not possible because (our implementation of) AMG only works on closed surfaces. We could run AMG at a finer scale for the whole surface, and only use those (smaller) domains that overlap with the target domain. However, it is in general not possible to replace a coarse domain by a number of domains at a finer scale, because it is not guaranteed that a domain lays completely within another domain at a coarser scale. Since we cannot use AMG for adaptive subdivision, we decided to drop the idea completely. The problem of how the basis function should be divided among the new domains seems to be too big.

3.6 Acceleration

In this section we outline some opportunities for accelerating texture construction by making use of the graphics hardware. We have not yet implemented these possibilities, because we did not have enough time to work them out and they require more advanced graphics hardware than was available to us anyway.

For the radial basis functions method, we can render the basis functions ϕ directly to a temporary texture with the same resolution as the domain primitive's texture and containing floats. Each texel in this texture holds the value of f for that texel. Of course, the basis functions need to be quantized into a texture before they are rendered. The width of a basis function can be controlled by scaling the texture. Instead of overwriting the contents of the temporary texture, we choose to sum the values by using additive blending. After we have rendered the basis function for each point sample, we have effectively summed the basis function values for each texel. Normalization is a more difficult problem and requires more recent graphics hardware, but is certainly possible. After normalization, we can read the values of f for each texel from the temporary texture and use these to construct the color and alpha channel of the domain primitive's texture. By using this acceleration technique, the radial basis functions method can be made a lot faster. Indeed, where without acceleration all basis functions have to be summed for each texel, with acceleration each basis function is only rendered once to the texture. The basis functions are summed for each texel by the graphics hardware.

For the linear affine basis functions method, we can accelerate the interpolation by rendering the constructed triangle mesh using Gouraud shading directly to the color channel of the domain primitive's texture. Because alpha values can also be interpolated using Gouraud shading, this also works for the alpha channel. Note that for this method, it is necessary to remove triangles that fill up concavities before rendering them, as explained in section 3.3.3. Correct handling of the texels outside the triangulation is not that easy and requires further thought. Although the triangulation algorithm is still needed when using acceleration, the interpolation of colors is done completely by the hardware, which is optimized for this task.

Chapter 4

Rendering

4.1 Introduction

In the previous chapter, we have described how we construct a domain primitive for each domain. Each domain primitive consists of a support polygon with a texture applied that captures the color and basis function information. In this chapter, we explain how we combine these domain primitives in order to reconstruct the original surface. We do this by rendering the domain primitives, for which we use OpenGL. As we will see, the nature of our domain primitives makes the reconstruction a difficult task.

In the following section, we explain some OpenGL terminology that is necessary for a good understanding of the graphical operations involved. In section 4.3 we describe the fundamental element of reconstruction: blending. Thereafter, we will describe the reconstruction algorithms that we have developed. In the rest of this chapter we consider the AMG scale i to be fixed.

4.2 OpenGL fundamentals

OpenGL is a 3D graphics library, concerned with rendering 3D primitives to the 2D screen framebuffer, using graphics hardware when available. *OpenGL primitives* are the point, line segment and polygon. Each primitive is defined by one or more vertices. The *framebuffer* is a two-dimensional grid of pixels. The framebuffer consists of a combination of *logical buffers*: color, alpha, stencil and depth buffers. In the color buffer, we distinguish the front buffer and back buffer. Typically, the front buffer is displayed on the monitor, while the back buffer is invisible and is used for off-screen rendering.

Primitives are offered one by one to OpenGL by the program. Each primitive goes through a pipeline before it is displayed on the monitor. First, the vertices of the primitive are transformed and lit and the primitive is clipped to the viewing frustum. Next, the *rasterization* phase projects the primitive onto the view plane and a *fragment* is produced for each pixel that the primitive occupies when projected onto the screen. A fragment is a tuple of a framebuffer address and a value for each logical buffer. After rasterization, per-fragment operations are performed on the fragments and the fragments are placed in the framebuffer. Recent hardware supports user-defined fragment programs to replace or complement the fixed-function fragment operations. Unfortunately, this hardware was not available to us, so we had to use the fixed function pipeline only.

The *depth buffer* is a logical buffer that stores one depth value per pixel. It is necessary for visible-surface determination [18]. Visible-surface algorithms solve the visibility problem: the question which surfaces of objects are visible and which are occluded. The depth buffer does this on a per-pixel basis. The z-buffer and w-buffer are particular implementations of the depth buffer. The *depth test* is a per-fragment operation that places the incoming fragment into the framebuffer if the incoming fragment is closer to the viewpoint than the current fragment at that pixel in the framebuffer. Otherwise, the fragment is discarded. The depth test and depth buffer together effectively depth-sort fragments, and enable OpenGL primitives to be drawn in an arbitrary order while maintaining the correct fragment order.

The *alpha test* is a per-fragment operation that accepts or discards incoming fragments based on their alpha value. Fragments whose alpha values are accepted are allowed to continue through the pipeline. Discarded values are thrown away and will not contribute to the final image.

Blending is a per-fragment operation that combines the color and alpha of the incoming fragment from the current primitive with the color and alpha of the fragment already in the framebuffer at the position of the incoming fragment. The programmer can specify how these fragments are combined, although the possibilities are limited. This also depends on the particular hardware used. A particular form of blending, called *additive blending*, means that the color and alpha values of the incoming fragment are combined by adding them to the values of the fragment that is already in the framebuffer.

Figure 4.1 depicts the stages in the OpenGL pipeline mentioned above.

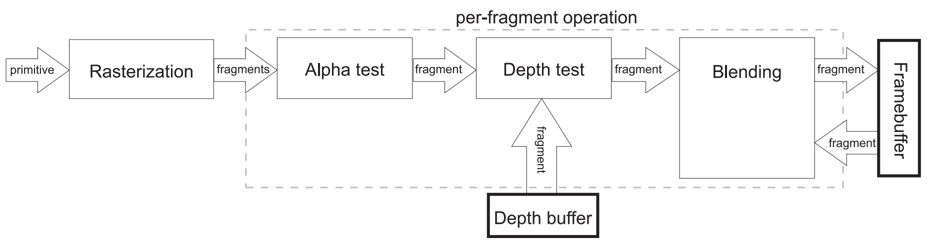


Figure 4.1: A graphical representation of the OpenGL pipeline.

4.3 Reconstruction using blending

Rendering the domain primitives can be considered as a reconstruction of the original surface S. The reconstruction is view dependent and has to be repeated for every viewpoint. With thresholded domains, a point sample is often represented in several domain primitives because thresholded domains can be overlapping. With strict domains however, every point sample is represented in exactly one domain primitive. Therefore, simply rendering all thresholded domain primitives yields a reconstruction in which point samples are over-represented, because each point sample is used more than once in the reconstruction. The rendering of strict domain primitives does not have this problem (see figure 4.2).

Note that the rendering of the strict domains has the problem of holes in the surface. This is because the strict domain primitives are created independently and no connectivity information is available. Thresholded domains often have points in common with other domains, and as a result their support polygons will often overlap.

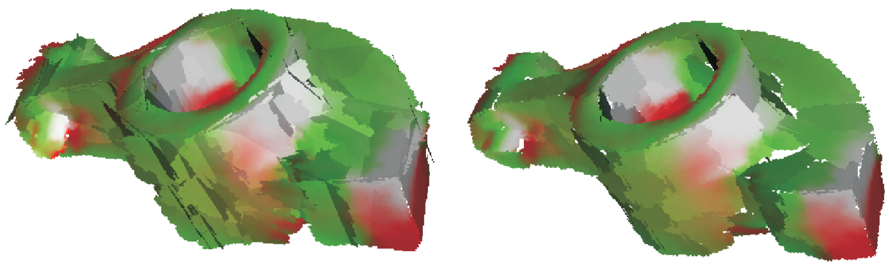


Figure 4.2: Left, a rendering of the thresholded domain-primitives is shown. On the right is a rendering of the strict domain-primitives.

We have the following idea. For each domain primitive j, we use the associated basis function $\varphi_j \in B$ to determine the contribution of each texel t on the polygon to the final image. This is in fact implemented by scaling the color at t by the value of the basis function at t, which is stored in the alpha channel. Although each point p on the surface S may be represented in multiple domain primitives, the color rendered for p is actually an interpolation of the colors in the different domain primitives, because of the partition of unity of the basis B. Therefore, the over-representation mentioned above is solved.

Let's look at it from a different perspective. One can categorize each point p on the surface S in one or more quasi-flat regions. Some points can clearly be categorized in one region, i.e. they have a high probability to belong to that region. Other points may be categorized in several regions. We use AMG to automatically classify these regions and create domains from them. The value of the basis function φ_j at point p indicates the probability that point p belongs to the domain associated with that basis function. Our idea is in fact to render these probabilities. In contrast to categorizing each point in one domain, this will produce a smoother image as we will see in the next sections.

4.4 Visible-surface determination

In the previous section, we have globally indicated our reconstruction technique. We use the thresholded domains and for each domain primitive we use the alpha channel in which the associated basis function is stored to scale the colors. We use additive blending while rendering the domain primitive to sum these scaled colors. In this section we work out our technique further.

Suppose that we want to render two polygons using blending. One polygon is in front of the other. When the depth test is enabled and we render the front polygon first, the fragments of the back polygon will be discarded and will not blend with the fragments of the front polygon. The solution to this problem is to disable the depth test.

Now, imagine that we simply render all domain primitives with blending enabled and disable the depth test. The result can be seen in figure 4.3. Because we disabled the depth test, we can look through parts of the surface that should actually be opaque and occlude other parts of the surface. In typical OpenGL applications, the depth buffer algorithm takes care of visible-surface determination. Whereas this algorithm delivers one fragment per pixel, namely the front-most fragment, we need multiple fragments. We therefore have to develop our own visible-surface determination algorithm.

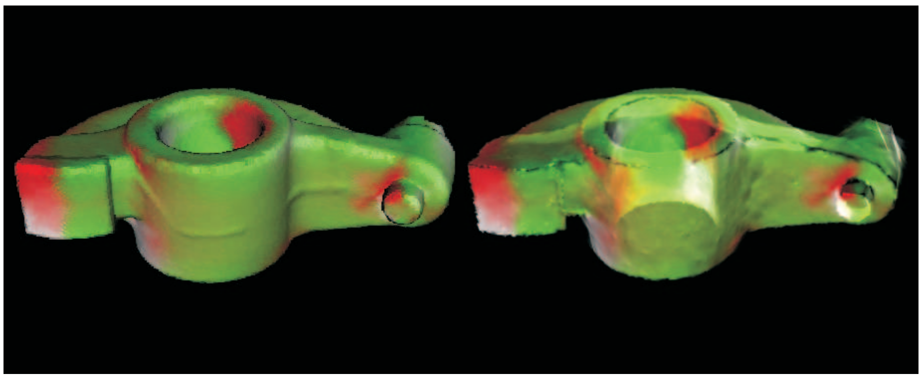


Figure 4.3: On the left, the model is rendered using point primitives. On the right, the model is rendered using domain primitives with additive blending enabled and the depth test disabled.

For solving the visibility problem, we come to the following observation. Two fragments for the same pixel from two different domains may only blend with each other when their respective basis functions overlap. Let us elaborate. The basis functions "live" on the original 3D surface. However, when we render the textures that capture the basis functions, the fragments are projected to 2D screen space. Depending on the viewpoint, fragments from arbitrary basis functions may be summed when they are coincidentally projected onto the same pixel. In order to solve this problem, we have to allow fragments from textures to blend *only* if the corresponding basis functions overlap on the original 3D surface. This ensures that two fragments are only blended when this is meaningful, namely when their associated basis functions overlap on the original surface and may thus be summed. Another important requirement for solving the visibility problem is that only fragments of the front-most domains must be visible on the screen.

Based on these requirements, we now present a conceptually correct visibility algorithm. For each pixel x in the framebuffer, we cast a ray into the scene from the viewpoint and determine the front-most intersection p of the ray with the original surface. If there is no intersection, the pixel will not be affected. If there is an intersection, we render the domain primitives that contain p but only allow pixel x to be affected by them. Because we determine the front-most intersection, we are sure that the pixel is a blending of domain primitives that approximate the front-most domain. See the example in figure 4.4. Unfortunately, we cannot implement this algorithm using current graphics hardware. It is an image-based approach, instead of the object-based approach that most graphics hardware and OpenGL employ. An object-based approach iterates over the primitives, whereas an image-based approach iterates over the pixels. Furthermore, we do not want to use the original surface in our rendering.

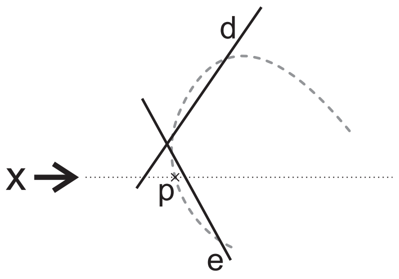


Figure 4.4: An image-based algorithm. A ray cast from pixel x intersects the surface at p. The domain primitives d and e both have p in their domain and are rendered for pixel x.

We present another method that does not use the original surface and is more of an object-based approach. We render all primitives and while doing this, a sorted *fragment-list* for each pixel is constructed in hardware. We then determine the front-most fragment of each list and determine which fragments are allowed to blend with it. This is called *the prefix*, because it is a prefix of the fragment list. Consider figure 4.5. In this example, fragment f_1 should be visible as it is the front-most fragment. It should be blended with f_2, because their basis functions overlap (this is not explicitly shown in the picture). The prefix in this example thus exists of fragments f_1 and f_2. The basis functions associated with fragments f_3 and f_4 do not overlap with the basis function of the front-most fragment f_1. The algorithm performs erroneous when the front-most fragment for a list should in fact not be visible. This could happen when domain primitives of domains in the back intersect through domain primitives of domains in the front, like in figure 4.6. The problem is inevitable as long as we do not want to use information about the original surface. Fortunately, such a configuration of domain primitives is rare.

State-of-the-art graphics hardware might be able to sort incoming fragments and to determine the prefix explicitly by using fragment programs. Unfortunately, our hardware does not support fragment programs. Instead, we have come up with three methods for determining the prefix implicitly. Although we have stated that we should use the information with regard to which basis functions overlap, we have also implemented some simpler heuristics for creating the prefix. The three *rendering methods* are:

Two-pass Determines the prefix by using a depth heuristic. All fragments within a certain depth of the front-most fragment are considered to be in the prefix.

Dark back-face Uses back-facing polygons to create groups of fragments that are allowed to blend. A sort of painter's algorithm [18] is used to sort the groups and to select the front-most group as the prefix.

Overlapping domains Determines basis function overlap on the original surface to create the prefix.

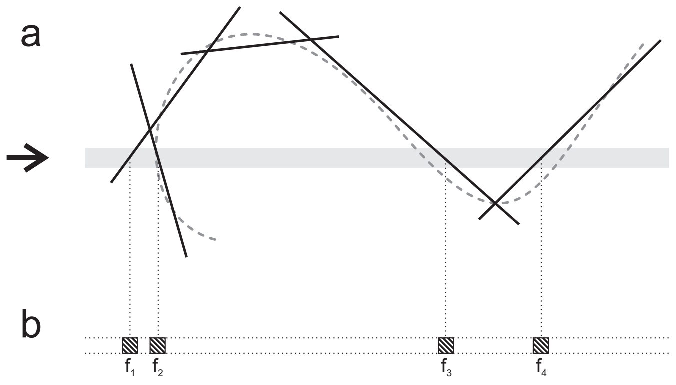


Figure 4.5: In **a**, a part of some surface is indicated by a dotted line, the domain primitives by black lines. The arrow indicates the viewing direction. The pixel for which the fragment list in **b** is constructed is indicated by a thick gray horizontal line in **a**.

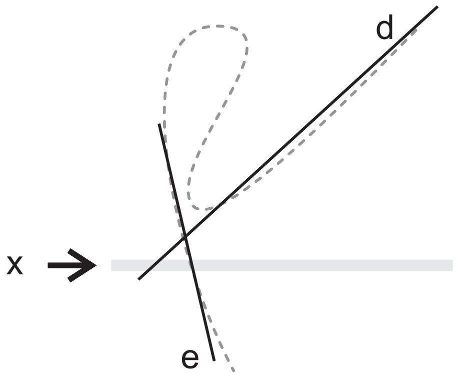


Figure 4.6: A problem of constructing the prefix is that the front-most fragment should not always be visible. In this example, the fragment for pixel x of domain primitive d lays in the front of the fragment of e, while in fact the domain of d lays behind the domain of e.

Both the dark back-face and overlapping domains methods sort the domain primitives. In the next section we will explain why sorting is important. In the three sections thereafter, we discuss the three methods above in more detail.

4.4.1 Sorting

In the previous section we have stated that we want to depth sort the fragments per pixel. However, creating a sorted list of fragments per pixel is not possible on our graphics hardware. OpenGL does provide the depth buffer algorithm, but this algorithm only delivers the front-most fragment per pixel. We can however offer the domain primitives in a sorted order to OpenGL, so that most of the incoming fragments enter OpenGL's pipeline in the same sorted order on a per pixel basis. This is similar to the well-known painter's algorithm [18]. In some applications the painter's algorithm is used to solve the visibility problem, for example when a depth buffer is not available. The painter's algorithm sorts primitives back-to-front based on either the minimum z-coordinate or the z-coordinate of the centroid. It then renders them back-to-front so that primitives in the front enter the pipeline later than those in the back and consequently overwrite those in the back.

One problem with sorting primitives is that the fragments are only processed in the correct order if the sorting is unambiguous, that is, when the polygons do not overlap with their z-extents. When primitives do overlap with their z-extents, which might occur in our case, the sorting cannot be made unambiguously and not all fragments are processed in the correct order (see figure 4.7.**c**). Luckily, we have some leeway as the ordering does not need to be exact in our case. The ordering of the fragments within the prefix is not important. If we look for example at figure 4.5, we see that the exact ordering of the fragments f_1 and f_2 within the prefix does not matter.

To totally resolve the ambiguity issue, a complicated algorithm (called the depth-sort algorithm [18]) would be required that splits primitives so that they can be sorted unambiguously. We choose not to perform such an algorithm, as it is tedious to implement and is computationally expensive. Moreover, we observed that in practice the ambiguities do not occur often because fragments do not have to be sorted within the prefix and that the ambiguities that do occur will not lead to disturbing artifacts.

We further observe that for a correct sorting of domain primitives, it is desired that the domain primitives are about equally sized. Otherwise, they cannot be ordered by simply taking their centroid's distance to the viewer as sorting criterion. We see this in figure 4.7.**b**. A similar problem occurs when we take the distance to the front-most vertex. Luckily, AMG does not deliver domains that are extremely out of proportions. We considered this as a problem of AMG in section 2.4, namely that basis functions do not grow as much as possible, but it turns out to be actually beneficial for rendering. As a result, merging domains as proposed in section 3.4 may lead to more severe artifacts.

4.4.2 Two-pass method

The two-pass idea is taken from the QSplat paper [1], in which it is used to render the alpha-blended fuzzy-spot primitive. The method limits the amount of fragments that blend together by a certain depth offset. Only fragments that

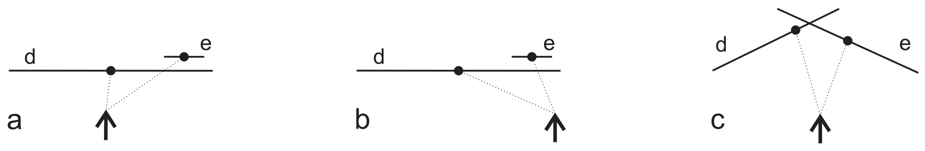


Figure 4.7: In **a** the distance from the viewpoint to the centroid of domain primitive d is smaller than to the centroid of e and is therefore correctly sorted. In **b** the configuration of the domain primitives is the same, but the viewpoint is such that e comes first in the ordering, because the distance is smallest. In **c** the sorting is ambiguous. This does not matter however when d and e are from overlapping domains and should blend.

fall within a chosen distance to the front-most fragment are considered to be in the prefix and are displayed on the screen. The heuristic we use here is that two fragments should only blend if the depth distance between them is below a chosen threshold that is uniform for the model. All fragments that fall within the range are considered to be in the prefix and will blend.

First, the domain primitives are rendered, but only to the depth buffer. The domain primitives need not be offered to OpenGL in a sorted manner. Next, a positive offset is added to the depth values in the depth buffer. This effectively translates the depth-buffer image further along the viewing direction. Then the domain primitives are rendered again in no particular order, with the depth test and blending enabled, but without writing into the depth buffer. Incoming fragments that fall in the range are blended, those that are behind it are discarded.

```
for each domain primitive $d$ do
    Render $d$ to the depth buffer, but not to the color buffer. A positive offset
    $o$ is added to each value written to the depth buffer.
end for
for each domain primitive $d$ do
    Render $d$ with blending enabled, depth-buffer reads enabled and depth-
    buffer writes disabled.
end for
```

A disadvantage of this algorithm is that the offset o is uniform across the entire image and must be set to the maximum depth-distance between any two fragments that should blend. We cannot in general choose the distance so that the correct prefix for all pixels is achieved. As a result, fragments that should not blend do blend when they fall within the offset o. This shortcoming is expected. The distance between two fragments only indicates the chance that they are allowed to blend. The smaller the distance between two fragments along the viewing axis, the bigger the chance that they are allowed to blend. See figure 4.8 for a graphical explanation of the two-pass algorithm and the associated problem.

4.4.3 Dark back-face method

The main idea of this method is that back-facing polygons separate the fragment list into groups. The fragments in such a group should blend with each other,

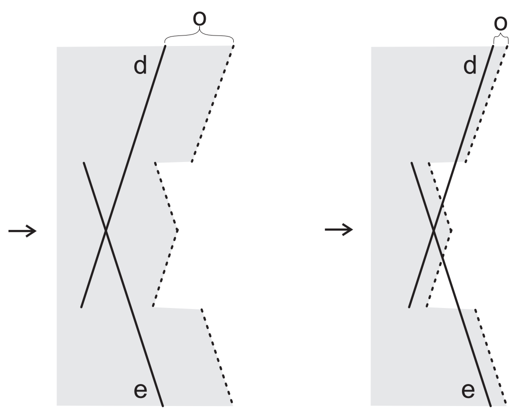


Figure 4.8: The two-pass algorithm. The dotted lines denote the depth values in the depth buffer. The area in which all fragments will be blended is denoted by solid gray. In the scene on the left, the offset o is chosen big enough so that domain primitives d and e completely blend (i.e. all of their fragments). In the scene on the right, the offset o is chosen too small. The result is that the domains do not blend at the places where the domains do not fall within the gray area.

while fragments from different groups should not blend. This heuristic requires that the surface is closed (i.e. has no holes in it). This requirement is met since it is also a requirement for AMG. To sort these groups so that only the front-most group is visible, i.e. the prefix, we sort the domain primitives.

The implementation is as follows. The domain primitives are sorted back-to-front. We render each front-facing domain-primitive with additive blending enabled. For each back-facing domain-primitive, we reset the colors in the framebuffer to 0 for alpha values that are greater than 0 (using the alpha test).

Consider diagram 4.5. Initially, the color for the pixel under consideration is black. The dark back-face algorithm will first render the hindmost fragment f_4, which is blended with the black fragment already in the framebuffer. Fragment f_3 comes from a back-facing polygon, so the color in the framebuffer is reset to black, thereby removing the influence of f_4. Next, fragment f_2 is blended with the black fragment. Finally, fragment f_1 blends with fragment f_2, which is the final result on the screen.

An improvement can be made by subtracting the alpha value α of the incoming fragments as a luminance value from the framebuffer pixel (and clamping the value to the lower bound 0), effectively darkening the pixel by the amount α. When α is 1, the pixel's color will become totally black. When α is 0, the color will remain the same. Anything in between makes the color proportionally darker. The advantage of these "soft" back-faces above the "hard" back-faces, is that the contours of the model become softer, hence prettier. See figure 4.9.

One problem of the dark back-face algorithm is that it assumes the model is concave. Although the original surface may be concave, the approximation by the domain primitives is not. Domain primitives do not form a closed surface, which is especially noticeable when the surface is viewed perpendicular to the surface's normal.

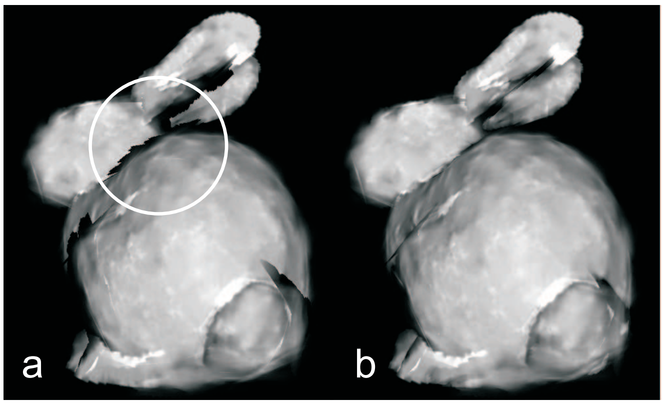


Figure 4.9: The difference between the hard back-faces in **a** and soft back-faces in **b** can clearly be seen at the back of the bunny, indicated by the circle.

4.4.4 Overlapping domains method

Both the two-pass and dark back-face method do not use information about overlapping domains, but instead use two simple heuristics. The overlapping domains method does take domain overlap into account.

The algorithm works as follows. In each iteration we render the front-most domain primitive d that we have not rendered yet, and we render all the domain primitives whose domains overlap with d (called *neighbors*). Now we know that we have created a prefix for each pixel of d, because we render the domain primitives from the front to the back and because we have blended all domain primitives with d that are allowed to. We *lock* the pixels of d so that these prefixes can not be overwritten later by other domain primitives. For the pixels of the neighbors that are not in d, the prefixes aren't complete yet and will not be locked. These prefixes will be completed in later iterations.

To lock the pixels we use the depth buffer. We are sure that by rendering a domain primitive d to the depth buffer we will effectively lock the pixels, because domain primitives of later iterations lay behind d, assuming that the domain primitives could be sorted unambiguously (section 4.4.1).

We present the pseudocode for a better understanding of the algorithm.

```
$L \leftarrow$ list of domain primitives sorted front-to-back
$R \leftarrow$ list of booleans initially false, list has same size of $L$
**for** $i$ from 1 to $length(L)$ **do**
  **if** $L_i$ is facing the camera **then**
    **if** not $R_i$ **then**
      render $L_i$ to color buffer with blending enabled
      $R_i \leftarrow$ true
    **end if**

    **for** each neighbor $L_j$ of $L_i$ **do**
      **if** not $R_j$ **then**
        render $L_j$ to color buffer with blending enabled
        $R_j \leftarrow$ true
```

```
            end if
        end for
        render $L_i$ to depth buffer
    end if
end for
```

An example is shown in diagram 4.10. In the first iteration, domain primitive d is rendered because it is closest to the viewer (in this diagram we assume an orthogonal projection). All rendering is done with blending enabled. We render the only neighbor e of d and then lock the pixels of d by writing them to the depth buffer. The next iteration we consider e. Because e has already been rendered, we do not render it again. Indeed, the pixels of e would be influenced twice by the basis function of e, because blending is enabled. We do however render the neighbor f of e, after which we lock the pixels of e. In the next iteration we see that f has already been rendered, so we do not render it again but only lock its pixels. In the final iteration, we render g. The incoming fragments of g will be occluded by the pixels of d and e that are already in the depth buffer. Their z-components are further away from the viewer than the fragments from d and e, and will be discarded by the depth test. Consequently, only the hatched part of g will be seen in the final image.

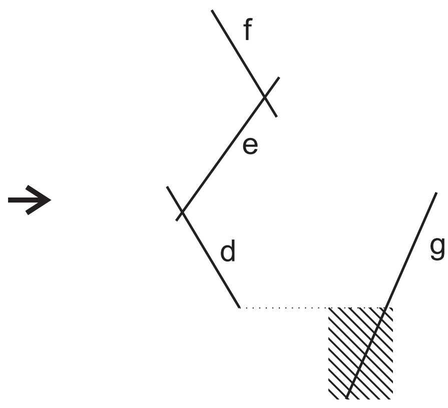


Figure 4.10: An example of the overlapping domains method.

We have stated that overlapping domains are considered neighbors. This means that two domains are considered neighbors even if they only share one point. This can also happen when the domains lay perpendicular, and then it is often not desired that the two domains are considered neighbors and are blended. We get better results if we raise the number of common points by which two domains are considered neighbors. In figure 4.11.**a** the teeth model is rendered using points, in **b** and **c** using domain primitives. In **b** two artifacts are encircled that result from incorrect classification of neighbors. The bright parts are caused by the incorrect blending of a molar domain primitive and a palate domain primitive. When the required number of common points is raised, the artifacts disappear as can be seen in figure 4.11.**c**. Two domains are considered neighbors if the fraction $\frac{\#commonpoints}{\#points}$ for at least one of the two domains exceeds a certain threshold, that is empirically found. In practice 0.05 performs well for most models. We have chosen a ratio instead of an absolute number of points, so that the threshold can be the same for most models and AMG scales.

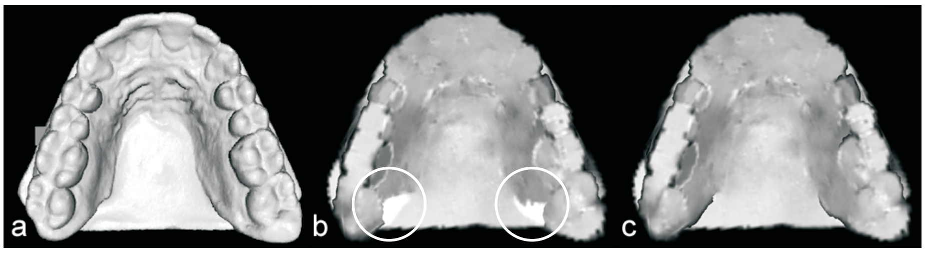


Figure 4.11: If two domains are considered neighbors even when they only share one point, artifacts may occur.

In figure 4.12 we visually compare the dark back-face method to the overlapping domains method. When using the dark back-face method we see areas that are too bright, because there are too much fragments in the prefix. This problem does not occur when using the overlapping domains method. Instead we see some dark silhouettes, for example around the tail, that do not occur in the dark back-face method.

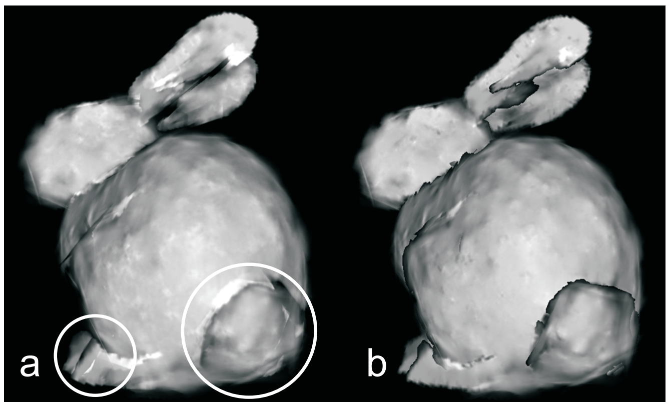


Figure 4.12: The bunny model is rendered using the dark back-face method in **a** and the overlapping domains method in **b**. We see areas that are too bright encircled in **a**, caused by an incorrect prefix (defined on page 37).

This problem of the overlapping domains method is caused by the fact that the depth buffer is used to protect pixels from being overwritten. All pixels that a domain primitive occupies are locked, also the pixels for which the fragment has a low alpha value, meaning that the influence of the domain primitive is small. This may result in a black silhouette around domain primitives. This effect can be diminished by increasing the minimum alpha value for which pixels are locked, by using an alpha test. In figure 4.13.**a** the balljoint model is rendered by using point primitives. In figure 4.13.**b** the domain primitives are rendered using the overlapping domains method. A dark silhouette around the ball is clearly visible. The alpha test reference value is increased to 0.1 in figure 4.13.**c**, and the black silhouette is clearly thinner. Of course, the reference value may not be too high or otherwise holes may appear. The alpha test is supported by graphics hardware and can therefore be dynamically adjusted by the user during rendering to his or her liking, although the default setting of 0.1 often suffices.

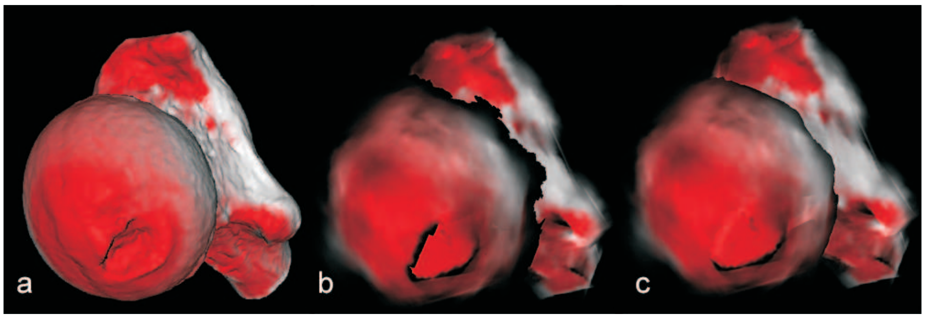


Figure 4.13: A problem of the overlapping domains method is that a dark silhouette may occur, because pixels are locked (defined on page 42) even for small alpha values. In **a** the balljoint model is rendered using points, in **b** and **c** using domain primitives. In **b** the dark silhouette can clearly be seen. It is diminished by increasing the alpha reference value to 0.1 in **c**.

Chapter 5

Applications

5.1 Implementation

In the previous chapters, we have explained how we can decompose a point-set model into domains, construct a domain primitive for each domain and render those domain primitives using OpenGL. In this chapter we will first present the results of our approach by means of screenshots and statistics. Thereafter, we outline the problems that still remain to be treated.

We have integrated our pipeline into the QSplat software [1], because the QSplat system is relatively simple to extend and the source code is publicly available. The operating system used is Linux. We did not have to implement the surface classification and AMG algorithm, as it was already implemented before the start of this graduation project. For all screenshots and timings we used a point-select threshold K (see section 2.5) of 0.01, the overlapping domains method is used for rendering and OpenGL linear texture filtering is used. Our test machine had an Intel Pentium IV 2.4 GHz with 256 MB memory and a GeForce4 MX440.

All timings that are marked by an asterisk * are influenced by disk swapping. This has been detected by looking at the hard disk activity. Our program often used more memory than was physically available on our machine, which causes swapping. Swapping negatively affects timing.

5.2 Results

5.2.1 Screenshot compilation

Figures 5.1, 5.2 and 5.3 contain screenshots of 8 different models. Figure 5.1 contains the (Stanford) bunny, balljoint and santa models, figure 5.2 the teeth and female models, and figure 5.3 the rockerarm, lion and dinosaur models. Each model is rendered once using point primitives and once using domain primitives for comparison. The parameters for the domain primitive construction, namely AMG scale and texel size, are chosen so that the reconstructed surface looks visually convincing. The texel size determines the size of a texel in world coordinates, by multiplying it by the average point-sample radius. The smaller the texel size, the higher the texture's resolution becomes and the more detail that is captured. The most apparent artifacts visible are the dark spots, for example on the bunny. These are caused by the projection error, which is explained in section 5.3.1.

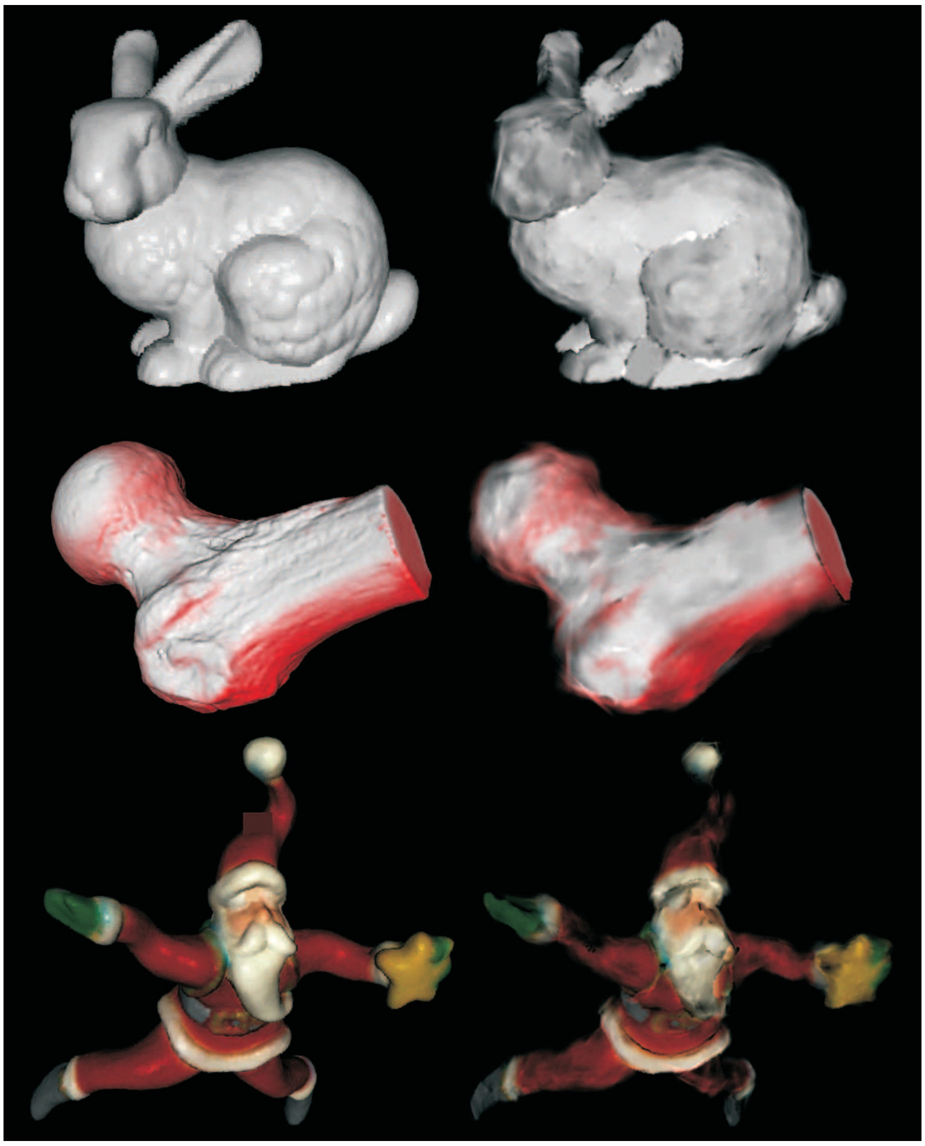


Figure 5.1: The bunny, balljoint and santa models.

Table 5.1 lists the particular parameters that we have chosen for each screenshot. Columns 1 and 2 show the model's name and number of point samples it consists of. Column 3 indicates which AMG level, or scale, was used to create the screenshot, level 0 being the coarsest scale. The amount of polygons, or domain primitives, that result from this level is shown in column 4. The fifth column shows the total number of texels that was used for rendering. This directly depends on the chosen texel size. The column labelled "tc" shows which

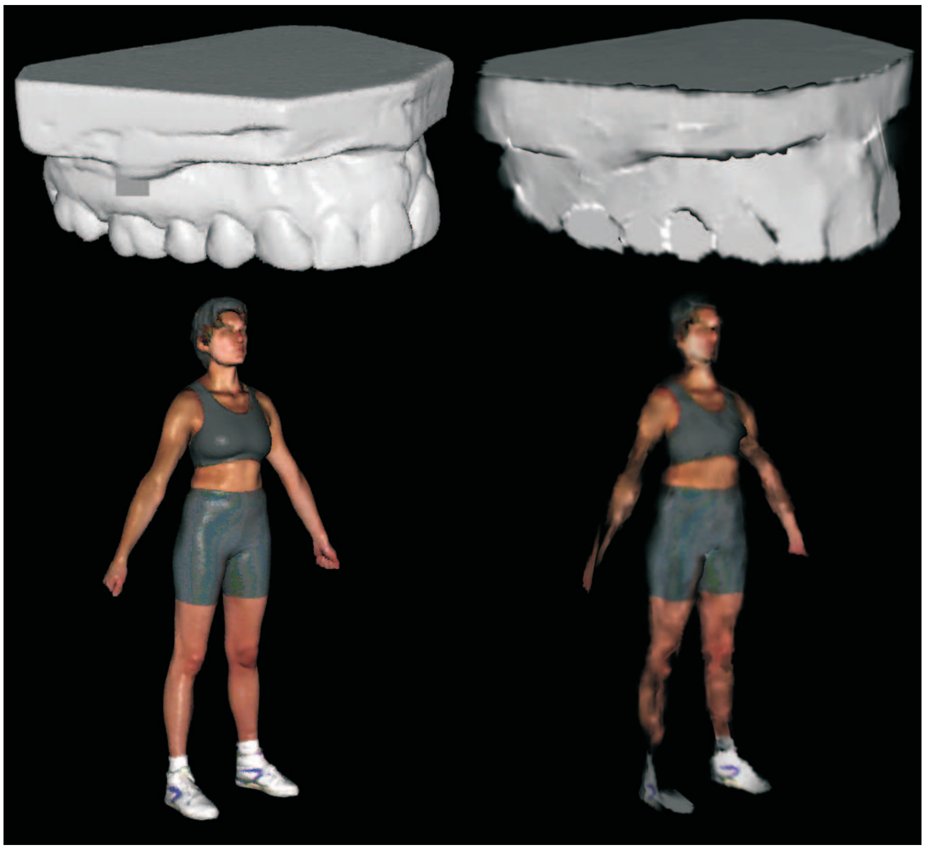


Figure 5.2: The teeth and female models.

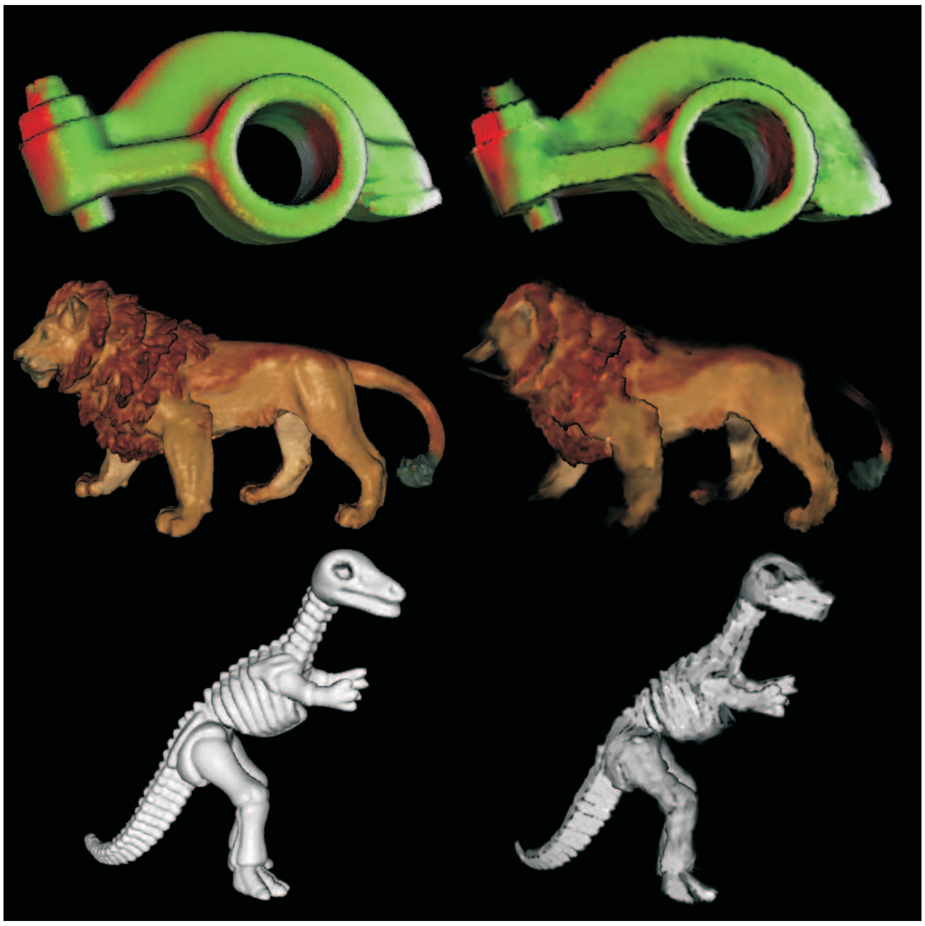


Figure 5.3: The rockerarm, lion and dinosaur models.

basis functions where used in texture construction, where "r" stands for radial and "la" stands for linear affine. For radial basis functions, the linear shape is used. Columns 7,8 and 9 contain timing information. The column "AMG" shows the time needed to construct the initial matrix M^0 and to run the AMG algorithm. This step has to be executed only once for each model, as the resulting matrices can be stored and reused when another level is chosen. The column "cb" shows the time needed to construct the selected basis from the AMG output matrices. The column "dpc" shows the time needed to construct the domain primitives from the basis functions, from finding tangent planes to texture construction.

We see that a lot of timings are influenced by disk swapping, especially for models with a large number of points. Of course, these models require more memory and often do not fit completely in memory. Moreover, our program is not very cache efficient: the program iterates frequently over a large number of points, for example when creating the basis. For this reason, we could not try AMG level 6 for the female model, as the duration became unworkable. This is unfortunate, since this model would benefit greatly from more domain primitives. The dark spots on the legs in figure 5.2 are caused by the fact that too few domain primitives are used to approximate the legs.

						time (s)		
Model	#points	level	#poly	#tex	tc	AMG	cb	dpc
bunny	35k	5	383	66k	la	10	2	8
balljoint	137k	4	129	110k	r	45	5	250*
santa	75k	5	364	60k	la	18	36*	216*
teeth	116k	5	225	99k	la	35	20*	190*
female	120k	5	255	102k	r	48	20*	138*
rockerarm	40k	5	445	73k	la	12	4	20
lion	180k	4	160	63k	r	87	100*	265*
dinosaur	56k	6	563	74k	la	15	47*	170*

Table 5.1: Statistics for the screenshots in figure 5.1, 5.2 and 5.3.

5.2.2 Statistics per AMG level

We pick the rockerarm model for more detailed statistics about the differences between AMG levels. Figure 5.4 shows screenshots of the AMG levels 0 to 5 from left to right, top to bottom. We see that from level 3 and above, the image looks reasonable. These models use more than the minimum number of primitives p_{low} mentioned in section 1.4.

Table 5.2 shows statistics for the different AMG levels of the rockerarm model, using texel size 1 and radial basis functions for texture construction. We see that the polygon count of an AMG level i is half the size of the level $i + 1$, which is in accordance with the statement in section 2.4 that AMG tries to reduce the number of basis function by roughly a factor 2.

Column "error" shows the approximation error calculated by averaging the distances between each point and its projected counterpart (see section 3.2). As expected, the average error decreases when the AMG level is increased. In the column "normalized", the error is normalized by the average point sample radius, which is 0.000417 in the case of the rockerarm model. We also see that the number of texels increase with higher AMG levels. This is because there are more domain primitives at higher levels and although these domain primitives are smaller, the increased total overlap causes the number of texels to increase. We further see that domain primitive construction ("dpc") takes less time for higher AMG levels (except for level 6 for which timing is off by swapping). This can be explained by the fact that texture construction using radial basis functions is in fact an $\mathcal{O}(n^2)$ algorithm: for each texel we have to sum the basis functions of all points in the domain and the number of texels (indirectly) depends on the number of points. Indeed, multiple smaller domains will then be processed faster than a few large domains. For the linear affine basis functions the same trend occurs, as the time complexity of the triangulation algorithm is $\mathcal{O}(n \log n)$, for n projected points.

We further note that the granularity of the scales is somewhat coarse. The jump from 445 polygons for level 5 to 1012 polygon in level 6 is quite large. Unfortunately, we cannot resolve this because the factor 2 is inherent to the AMG algorithm.

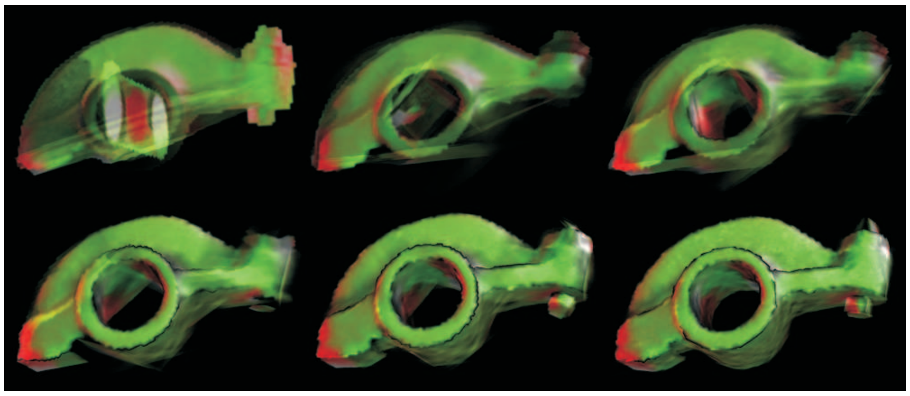


Figure 5.4: Screenshots of the different AMG levels of the rockerarm model.

level	#poly	#tex	error	normalized	cb	dpc	fps
0	5	20841	0.016802	40.3	< 1	103	805
1	17	43645	0.016296	39.1	< 1	89	690
2	37	56349	0.011802	28.3	< 1	53	635
3	84	60056	0.005707	13.7	1	24	600
4	193	68354	0.002904	6.96	2	15	510
5	445	78245	0.001455	3.49	4	18	412
6	1012	85972	0.000765	1.83	31	137*	285

Table 5.2: Statistics for the different AMG levels of the rockerarm model.

5.2.3 Frames per second

In this section we discuss the speed of our renderings, when using the overlapping domains method. For measuring the framerate, we zoom out so that the screen size of the model is about 100 by 100 pixels. This is a typical screen size we have in mind for our approach, as stated in the problem statement.

The framerate for each AMG level of the rockerarm model is listed in the column "fps" of table 5.2. We see a framerate decrease with higher AMG levels. This is expected, since the number of polygons increase with each AMG level. For all AMG levels, the framerate is higher than when rendering the model using point primitives, which is 140, what is what we aimed for. For more complex models than the rockerarm, the speed gain is even greater. Note that rendering time measurements are always hard to interpret, because there are a lot of factors involved.

In table 5.3 we compare the framerate using point primitives to the framerate using domain primitives for all remaining models. For measuring the frames per second when using point primitives, we used the original QSplat software. Although our program is able to render the model with point primitives like the original QSplat does, the rendering code has been modified by us for debugging purposes, which would negatively affect the framerate. The speedup for these models range from 16 times as fast for the complex lion model to 3 times as fast for the simpler bunny model, in favor of the domain primitives.

Model	fps points	fps dp	speedup
lion	36	590	16
teeth	38	550	15
balljoint	71	580	8.2
dinosaur	77	600	7.8
female	83	580	7.0
santa	83	510	6.1
bunny	140	410	2.9

Table 5.3: A comparison of the framerate using point primitives and the framerate using domain primitives for several models.

5.2.4 Texture construction methods

Figure 5.5 depicts the differences between texture construction methods for the bunny model. In **a**, **b** and **c**, radial linear basis functions are used with basis function widths of respectively 0.5, 1, and 2. Recall the discussion about basis function width in section 3.3.2. The width factor must be chosen such that $f(t)$ sums to at least 1 for texels than fall within the domain. For **a**, the basis function width is clearly chosen too small. Texels for which $f(t) < 1$ appear as dark dots. In **b** and **c**, the width is chosen large enough. However, domains are represented larger than they are because basis functions are wider than the point samples. This can be seen at the borders of the domains as over-bright areas. The same problem occurs with the radial constant basis functions in **d** (width factor 1), and the radial Gaussian basis functions in **e**. For the constant functions we observe some high-frequency noise. Indeed, these are the lowest order functions and thus result in the least smooth image. The difference between the Gaussian shape and linear shape is minimal. The image reconstruction using linear affine basis functions is shown in **f**. For these functions, we do not have to choose the basis function width. They are constructed by making explicit use of neighboring point samples instead. Therefore, there are less over-bright areas.

In table 5.4, timings for the different types of texture construction are listed, for three different texel sizes. For the radial basis functions, we measured the constant, linear, and Gaussian shapes. We observe that the radial Gaussian basis functions take the longest, because the Gaussian function is computationally expensive. Second comes the linear affine basis functions, due to the expensive Delaunay triangulation.

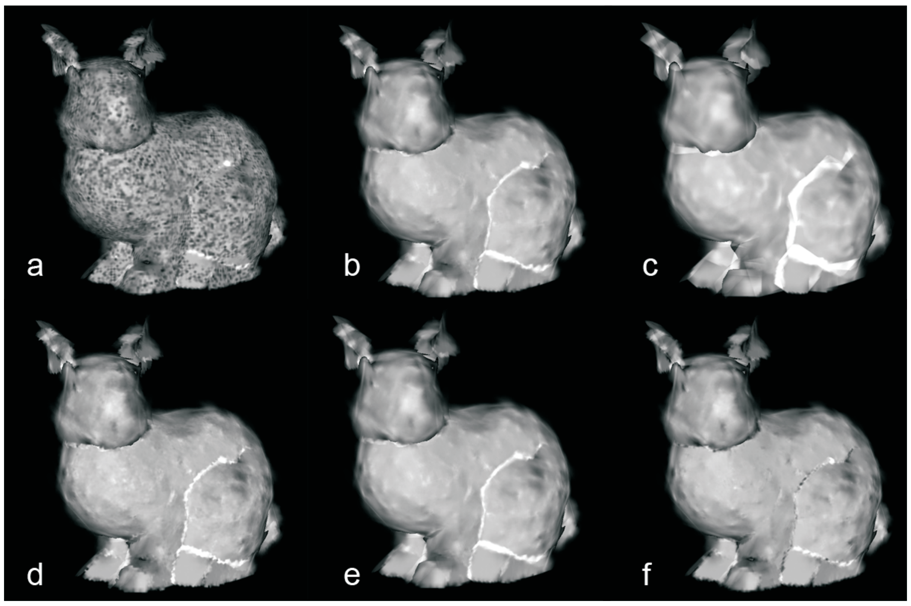


Figure 5.5: Screenshots of the different texture construction methods for the bunny model.

texel size	#texels	constant	linear	Gaussian	linear affine
2	66 k	7	7	37	8
1	255 k	16	18	135	30
0.5	1083 k	68	68	542	102

Table 5.4: Domain primitive construction times for the different texture construction methods for the bunny model, in seconds.

5.3 Problems

This section discusses the artifacts that are sometimes visible in our renderings. We will discuss the projection error, cracks, and sorting artifacts respectively.

5.3.1 Projection error

The projection error is caused by the fact that we approximate curved domains by flat domain primitives. Consider diagram 5.6. Let p be the intersection for pixel x with the surface represented by the dotted curve. The color and basis function values of point p are captured in the color and alpha channels at the perpendicular projection of p on the domain primitive, namely at p'. However, when the domain primitive is rendered, x will be filled with the fragment at q, not p'. The error increases when the angle between the viewing direction (indicated by the arrow) and the domain primitive's normal n increases. There is no error when the viewing direction is perpendicular to the domain primitive. As a result of the projection error, we might sum two fragments that do not represent the same part of the surface, although they do both belong to the same basis function. This may be visible as a spot on the surface that is too dark, too bright, or has the wrong color.

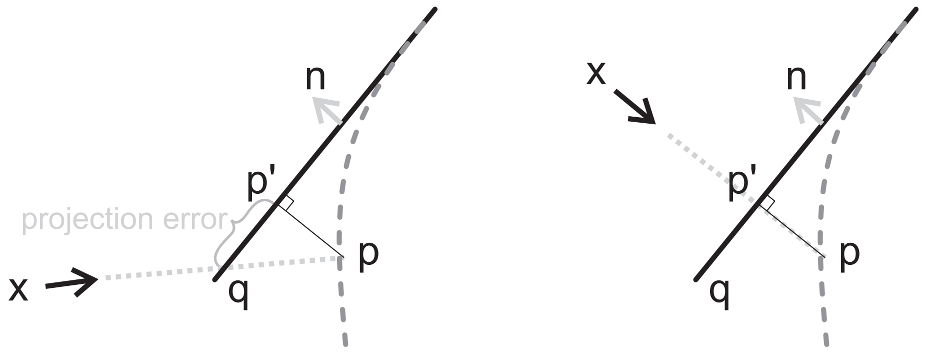


Figure 5.6: The projection error increases when the angle between the viewing direction and the planes normal n increases. There is no error in the scene on the right.

An extreme result of the projection error is shown in figure 5.7. Domain primitive d that represents p will not be rendered because it is back-facing with respect to the viewing direction. The result is that the pixel x will be darker than it should be. This effect is greatest when the viewing direction is perpendicular to the surface normal, so it is most noticeable at the contours of a model. In figure 5.8.**a**, the dark spots at the back of the bunny are clearly visible. Indeed, when we move the viewpoint in figure 5.8.**b** so that we look top-down at the back, the dark spots are no longer visible as the projection error diminishes.

The projection error is less severe when the AMG level is higher, and the number of polygons is thus larger. The normal of a domain primitive that has intersection point p within its domain will be a better approximation for the normal of the original surface at p then when the number of polygons is smaller.

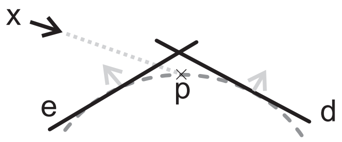


Figure 5.7: The projection error may cause dark spots. Only primitive e is rendered for pixel x, because d is back-facing.

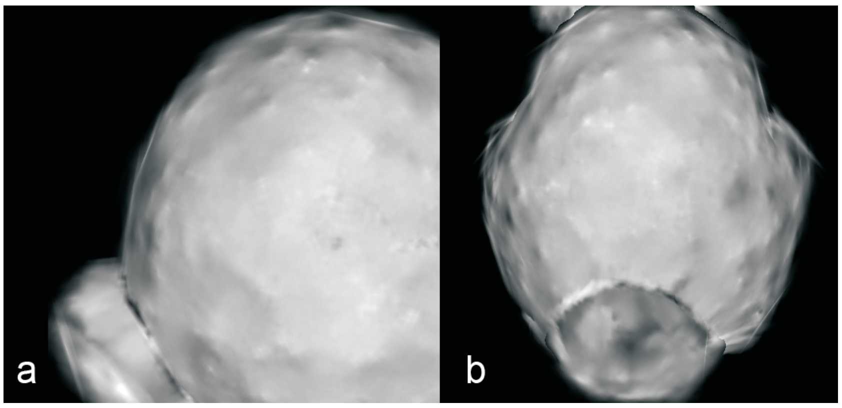


Figure 5.8: Screenshot **a** shows dark spots at the contours of the bunny. When looking directly at the back in **b** those spots disappear.

5.3.2 Cracks

Another problem occurs at strong discontinuities in the surface. Surface decomposition will (correctly) create two disjunct domains for both sides of the discontinuity. The domain primitives will not intersect and depending on the viewpoint, a “crack” occurs in the rendering. This is depicted in figure 5.9 for the rockerarm model.

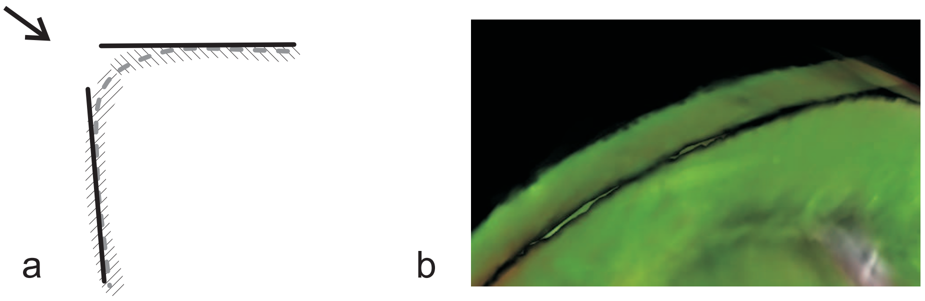


Figure 5.9: Diagram **a** shows the cause of cracks that are visible in screenshot **b**.

5.3.3 Sorting artifacts

Sorting primitives does sometimes introduce artifacts. The ordering of the domain primitives depends on the viewpoint and as a result may suddenly change when the viewpoint changes. These changes in ordering may be noticeable depending on the algorithm it is used in. Indeed, both the dark back-face and overlapping domains method suffer from these artifacts, but only seldomly. Unfortunately, these artifacts cannot be captured by still screenshots. They have to be seen in motion by running our software.

Chapter 6

Conclusions

6.1 Achievements

We have presented a novel approach for extreme simplification of point-set models that is based on intrinsic features of the surface at multiple scales. Our approach enables us to render complex point-set models more efficiently. It is best suited for detailed models that will be rendered to only a small screen size. This way, small details can be discarded by the simplification without losing visible accuracy.

Our most important idea is to use AMG for decomposing the surface into domains defined by basis functions and to use this "soft" decomposition of AMG in our rendering. By using the basis functions in our rendering as a transparency map for our domain primitives, we essentially do a signal reconstruction. We exploit the blending functionality of the graphics pipeline to render more information with little extra cost, which we have not yet seen with any other method (see also section 1.3). We developed a new primitive, the domain primitive, that is able to capture the basis function values and colors for a domain and is fast to render. Also, we presented and implemented a straightforward pipeline that creates a domain primitive for each basis function. The fact that each domain primitive is created independently adds to the elegance of our approach.

Reconstruction of the surface by rendering the domain primitives is less straightforward and was one of the harder problems to solve. It involves calculation of the overlap of domains on the original 3D surface for the best results. This seems inevitable as the domain primitives are constructed independently from each other. Other than their position, they possess no information about the relation to other domain primitives. We first outlined an algorithm that would correctly reconstruct the surface, but concluded that this was an image-based algorithm and thus not suitable for implementation on current graphics hardware. We then presented a more or less similar object-based approach, but concluded that it can only be implemented on very recent graphics hardware, which was not at our disposal. Instead, we developed three alternative algorithms that can be implemented on simple (i.e. non-programmable) graphics hardware, of which we implemented two. The best method we developed performs reasonably. Nevertheless, the difficulty of reconstructing the surface diminishes the elegance and efficiency of our approach somewhat.

Despite our efforts, some artifacts are still visible in the renderings. We identified various causes. Especially noticeable are the dark spots that are caused by the fact that we approximate the original surface by flat polygons that do not form a connected mesh. Unfortunately, this is fundamental to our approach and no easy solution seems at hand. The cracks are indirectly caused by the AMG algorithm. The sorting artifacts are caused by our rendering algorithm and the sorting that is required for it.

6.2 Comparison to billboard clouds

The Billboard Clouds approach (BC) [15] is similar to ours. Both approaches perform extreme model simplification (i.e. the simplification of static 3D models to only a few primitives) intended for distant geometry. In this section, we discuss the similarities and differences of both approaches.

First of all, our approach requires that the point set defines a smooth manifold, whereas BC works on a larger class of models, including triangle-soup models. Although our current implementation requires a point set as input, it could be adjusted to allow triangle meshes. Furthermore, the billboard clouds approach relies on a direct analysis of the model, in which planes that are nearly tangent to the model are favored to ensure a good surface approximation. Similarly, our approach tries to find tangent planes by the use of the AMG algorithm. The AMG algorithm is robust, and has a solid background [16]. BC simplifies the 3D model onto a set of planes with independent size, orientation and texture resolution, that capture the geometry of the model well. The tangent planes that we construct from domains have the same features. Both methods apply color textures, transparency maps, and possibly normal maps to the planes. The transparency map allows for visual complexity, while there is no need for a complex boundary description. Our approach does not only encode geometry in its transparency map, but also captures the basis functions that are delivered by AMG in it. This extra information is used during rendering too deliver a smoother image. In contrast, BC's transparency map only holds digital values, and is in fact a transparency mask. This makes the result look less smooth, but eliminates the blending problems we must solve for our method. In both methods, no connectivity information is needed for the tangent planes, which allows more aggressive simplification than typical mesh decimation algorithms that preserve connectivity.

Our approach works in a multi-scale, bottom-up fashion. Basis functions are constructed from the finest scale first, whereafter increasingly coarser scales are repeatedly constructed in a bottom-up fashion. In contrast, BC employs a single-scale, greedy, and top-down approach. BC tries to optimize the compactness of textures, to prevent billboards from having mostly empty space. With our approach this is guaranteed automatically, because the basis functions have compact supports. The only parameters for the BC approach are an error bound and the texel size. Our approach only requires the AMG level and the texel size as parameters. The AMG level indirectly controls the average approximation error.

In table 6.1 we have made a comparison of computation times between our approach on the left and the billboard clouds approach on the right. Because the BC software was not available to us, we took statistics for the madcat model from their paper. We used another model with similar complexity, since the madcat model was not available to us. For a good comparison, we generated the same amount of texels. We chose the AMG level for which the number of domain primitives was closest to the number of BC planes. As can be seen, our computation time for the dinosaur model took only half the time that BC needed for the madcat model. Of course, this is just an indication and a more thorough comparison is desired.

model	dinosaur	model	madcat
#points	56,194	#triangles	59,855
AMG level	5	error bound	6%
#domain primitives	241	#planes	171
#texels	1,617 k	#texels	1,638 k
time (s)	237	time (s)	529

Table 6.1: A comparison of computation times between our approach on the left and the billboard clouds approach on the right.

6.3 Directions of further research

A lot of opportunities for further research can come from the use of more powerful graphics hardware. Fragment programs may enable us to implement a rendering method that has less artifacts. The sorting of fragments may be implemented in another way than by sorting the domain primitives, thereby eliminating those artifacts that stem from sorting domain primitives. Furthermore, recent hardware allows the use of normal maps: a texture channel that captures normals, which can be used during rendering to convey a more detailed impression of the original domain, at little extra expense. Finally, powerful hardware enables the hardware acceleration techniques discussed in section 3.6. However, these ideas need to be worked out further before they can be implemented.

Another possible direction for research is to combine points with domain primitives in the rendering and come to a hybrid approach. They may aid in solving the problems around strong discontinuities, such as the problem of cracks and dark spots. Indeed, points are more suitable for high-curvature parts of the surface than domain primitives. However, rendering points and domain primitives together requires a more sophisticated rendering algorithm. Point primitives may certainly not be sorted as we do with domain primitives, for this would slow things down considerably.

We have not yet optimized our implementation for resource usage. With some more work, primitive construction and rendering can definitely be made faster. Memory usage of our pipeline can also be decreased, which is important for machines with little memory to prevent disk swapping. Texture compression can be implemented to decrease graphics memory usage.

Appendix A

Software structure

We have integrated our code with QSplat's code. You can find the most important files and classes below.

Geometry.h Contains generic geometry classes that we need in our pipeline, such as for representing points, vectors and polygons.

DomainFactory.cpp The singleton class `CDomainFactory` is the main class of our system, containing the model's point samples and the basis functions from the chosen AMG scale. It manages a set thresholded domains and optionally, a set of strict domains. Although we do not use the strict domains in our pipeline, they can be constructed for experimental purposes. The class also contains the rendering algorithms for surface reconstruction.

DomainSet.cpp The `CDomainSet` class contains a set of domain primitives. It is able to create an ordered list of domain primitives and the neighbor information for the rendering algorithms.

Domain.cpp The `CDomain` class holds the information for one domain primitive. It contains the unprojected and projected points, the bounding polygon and the texture. It contains methods necessary for constructing the domain primitive from a given basis function, and to render the domain primitive.

DomainOptions.cpp The singleton class `CDomainOptions` contains parameters that control both our pipeline and rendering algorithms, such as the selected AMG scale and point select threshold.

DomainTool.cpp The class `CDomainTool` contains the user-interface elements that allows the user to select the parameters in `CDomainOptions`.

qsplat_traverse_v11.h This QSplat file contains the traversal code for rendering the points. We modified it to enable us to render the points from selected domains, for debugging purposes.

xforms_gui.cpp We added some of our user-interface controls to this QSplat file.

Appendix B

Build process

Our program has been built and tested using Redhat Linux 8.0, kernel version 2.4.20 and gcc version 3.2 20020903. Dependencies are ANN 0.2 [14], Triangle 1.3 [20], glibc 2.1, OpenGL dynamic libraries (OpenGL 1.2 or higher) and the XForms dynamic library 0.88. A makefile is included to build the software. To build the software, execute `make` in the main directory.

Appendix C

Software user manual

In this section we describe the user-interface elements of our program. References to figure C.1 are denoted in parenthesis.

1. Start the program by executing `./qsplat "Models/bunny.qs"`.
2. Click with the right mouse button on the domain primitive construction tool (1).
3. A menu (2) appears in which the parameters of the domain primitive construction phase can be configured. The defaults settings can also be used.

   AMG level The AMG level for which domain primitives should be constructed.

   Point Select Threshold The minimum basis function value for which a point is considered to be within that basis function. See section 2.5.

   Texel Size The size of a texel in world coordinates as a percentage of the model's average point sample radius.

   Enlarge Bounding Rectangle The percentage by which the bounding rectangle sides will be enlarged, to ensure that the point samples fall completely within the bounding rectangle. This setting usually does not need to be altered.

   Use Model Colors When unchecked, the default QSplat gray color is used for all point samples instead of the model's colors.

   Adjust Brightness The value component (from the HSV color space) is increased by this amount. This setting can be used to brighten the point samples and domain primitives.

   Project Radii When checked, the radii of point samples are also projected. See section 3.1.

   Weighted Tangent Plane When checked, the tangent plane is calculated by weighting the points by their basis function value. See section 3.1.

   Merge Basis Functions When checked, basis functions are merged during primitive construction. See section 3.4.

Texture Construction The texture construction method that will be used. Either radial basis functions or linear affine basis functions can be chosen.

Basis Function Shape The basis function shape can be chosen from "constant", "linear" and "Gaussian". Only applicable to the radial basis functions method.

Basis Function Width The width factor w as described in section 3.3.2. Only applicable to the radial basis functions method.

Basis Function Normalize Criterion Either $f(t) \geq 1$ or $f(t) > 0$. Only applicable to the radial basis functions method.

Extend Color Channel When checked, every texel of the color channel is assigned a color. See section 3.3.4.

Set Decomposition Colors When checked, the domain primitives are given an alternative coloring so they can be easily distinguished. For debugging purposes. Can only be applied when the strict domains are calculated.

Calculate Strict Domains When checked, the strict domains and domain primitives are constructed. Strict domain primitives are not used by our approach. See section 2.5.

Overlapping Domains Ratio Mentioned in section 4.4.4.

4. After the parameters are configured, click "Ok".

5. Choose "DP construction" (3). The program tries to find the AMG matrix associated with the model. If it cannot find it, the user is asked for the location.

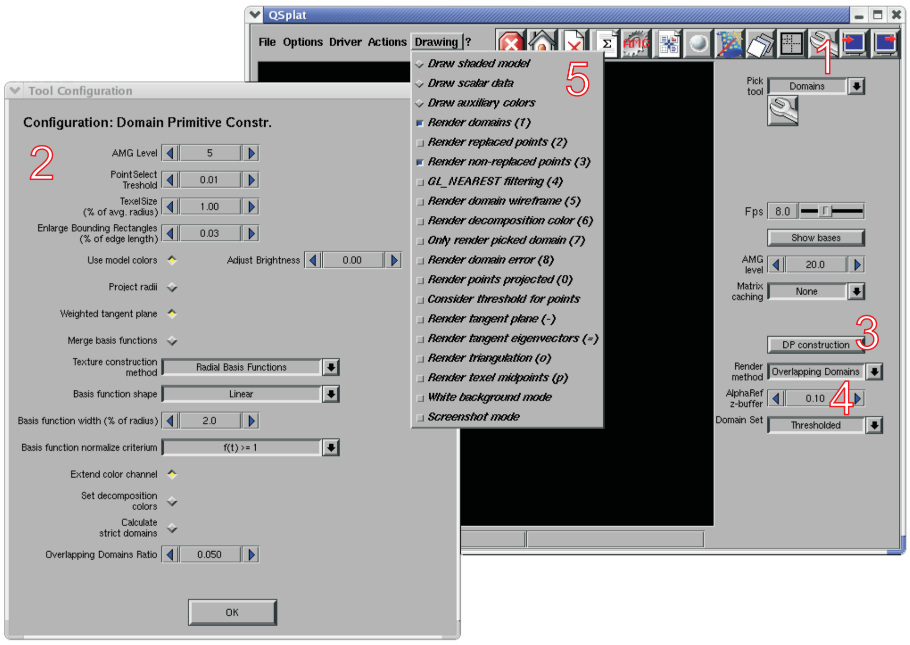
6. Next, the basis functions are constructed from the AMG matrix and then the domain primitives, which can take quite some time depending on the chosen parameters. After the construction has finished, the model will be rendered using domain primitives.

7. In the menu denoted by (4), rendering settings can be dynamically adjusted by the user. The "Render Method" pull-down menu allows the user to choose between different rendering methods. Among the overlapping domains and dark back-face methods described in sections 4.4.3 and 4.4.4, are the "Blend All" and "No Blend" methods. The "Blend All" method blends all domain primitives without solving the visibility problem. "No Blend" does not blend at all. This is useful for rendering the strict domains. They do not need to blend, for they are not overlapping.

   With the "AlphaRef" control, the user can control the alpha test reference value that is used for the overlapping domains method in section 4.4.4. The "Domain Set" pull down menu controls which domain set is rendered. Either thresholded or strict domains can be chosen.

8. In the "Drawing" menu (5), various rendering settings for debugging and informative purposes are available. Bounding rectangles, eigenvectors, texel midpoints and triangulation can be visualized using these settings. Also, domain primitives can be rendered separately by selecting the option

"Only Render Picked Domain". A domain primitive can be picked by clicking with the left mouse button on the domain primitive while holding the control button. To deselect a domain primitive, click the right mouse button while holding the control button.


QSplat
File Options Driver Actions Drawing ?
Draw shaded model
Draw scalar data
Draw auxiliary colors
Render domains (1)
Render replaced points (2)
Render non-replaced points (3)
GL_NEAREST filtering (4)
Render domain wireframe (5)
Render decomposition color (6)
Only render picked domain (7)
Render domain error (8)
Render points projected (0)
Consider threshold for points
Render tangent plane (-)
Render tangent eigenvectors (=)
Render triangulation (o)
Render texel midpoints (p)
White background mode
Screenshot mode
Tool Configuration
Show bases
DP construction
OK
1
2
3
4
5

Configuration: Domain Primitive Constr.

AMG Level 5
PointSelect Treshold 0.01
TexelSize (% of avg. radius) 1.00
Enlarge Bounding Rectangles (% of edge length) 0.03
Use model colors
Adjust Brightness 0.00
Project radii
Weighted tangent plane
Merge basis functions
Texture construction method Radial Basis Functions
Basis function shape Linear
Basis function width (% of radius) 2.0
Basis function normalize criterium f(t) >= 1
Extend color channel
Set decomposition colors
Calculate strict domains
Overlapping Domains Ratio 0.050

Pick tool Domains
Fps 8.0
AMG level 20.0
Matrix caching None
Render method Overlapping Domains
AlphaRef z-buffer 0.10
Domain Set Thresholded

Figure C.1: The user interface of our program. Visible are the main panel and the domain primitive construction panel. The "Drawing" menu is expanded.

Bibliography

[1] S. Rusinkiewicz, M. Levoy; "QSplat: A Multiresolution Point Rendering System for Large Meshes"; Proceedings of SIGGRAPH 2000, July 2000

[2] M. Pauly, R. Keiser, M. Gross; "Multi-scale Feature Extraction on Point-sampled Surfaces"; Eurographics 2003, Volume 22, Number 3

[3] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, C. Silva; "Computing and Rendering Point Set Surfaces"; IEEE Visualization, Vol. 9 No. 1, January-March 2003

[4] L. Linsen, H. Prautzsch; "Local Versus Global Triangulations"; Eurographics 2001, Volume 20, Number 3

[5] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, Werner Stuetzle; "Surface Reconstruction from Unorganized Points"; Computer Graphics (SIGGRAPH '92 Proceedings), Vol. 26, No. 2, July 1992, pages 71-78

[6] A. Kalaiah, A. Varshney; "Modeling and Rendering Points with Local Geometry"; IEEE Transactions on Visualization and Computer Graphics Vol. 9, No. 1, January 2003, pages 30-42

[7] U. Clarenz, M. Rumpf, A. Telea; "Finite Elements on Point Based Surfaces"; Proc. EG Symposium of Point Based Graphics (SPBG) 2004, The Eurographics Association

[8] H. Pfister, M. Zwicker, J. van Baar, M. Gross; "Surfels: Surface Elements as Rendering Primitives"; Proceedings of SIGGRAPH 2000, July 2000

[9] P. Cignoni, C. Montani, R. Scopigno; "A Comparison of Mesh Simplification Algorithms"; In Computers & Graphics, Volume 22, 1997, Pergamon Press.

[10] B. Chen; M. X. Nguyen; "POP: A Hybrid Point and Polygon Rendering System for Large Data"; In Proceedings IEEE Visualization 2001, pages 45-52

[11] J. Cohen, D. Aliaga, W. Zhang; "Hybrid simplification: Combining multi-resolution polygon and point rendering"; In Proceedings IEEE Visualization 2001, pages 37-44

[12] T. Dey, J. Hudson; "PMR: Point to mesh rendering, a feature-based approach."; In Proceedings IEEE Visualization 2002, pages 155-162

[13] J. Krivanek; "Representing and Rendering Surfaces with Points"; Postgraduate Study Report, Dept. of Computer Science and Engineering, CTU Prague, 2003; Available from http://www.cgg.cvut.cz/~xkrivanj/projects/points/project.htm

[14] D. Mount, S. Arya; "ANN: Library for Approximate Nearest Neighbor Searching"; 2003; Available from http://www.cs.umd.edu/~mount/ANN/

[15] X. Décoret, F. Durand, F. Sillion, J. Dorsey; "Billboard Clouds for Extreme Model Simplification"; Proceedings of the ACM Siggraph 2003; Available from http://www-imagis.imag.fr/Publications/2003/DDSD03

[16] U. Clarenz, M. Griebel, M. Rumpf, M. A. Schweitzer, A. Telea; "Feature Sensitive Multiscale Editing on Surfaces"; The Visual Computer, Volume 20, Number 5, July 2004, pages 329-343, Springer

[17] W. Press, W. Vetterling, S. Teukolsky, B. Flannery; "Numerical Recipes in C++"; Second Edition, Cambridge University Press, 1992, pages 468-474

[18] J. Foley, A. van Dam, S. Feiner, J. Hughes; "Computer Graphics: Principles and Practice"; Second Edition, Addison-Wesley, 1993, pages 649-720

[19] U. Clarenz, M. Rumpf, A. Telea; "Fairing of Point Based Surfaces"; Proceedings Computer Graphics International (CGI) 2004, IEEE CS Press

[20] J. Shewchuk; "Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator"; Available from http://www-2.cs.cmu.edu/~quake/triangle.html

[21] M. Levoy and T. Whitted; "The Use of Points as Display Primitives"; Technical Report TR 85-022, Univ. of North Carolina at Chapel Hill, 1985