Flexible Methods for Geometric Texturing

- From Terrain Visualization to Geometric Texture Mapping

Anders Torp Brodersen

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Flexible Methods for Geometric Texturing
- From Terrain Visualization to Geometric Texture Mapping

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Anders Torp Brodersen
March 22, 2007

Abstract

Texture mapping is a classical technique that is commonly used in computer graphics applications as a quick and intuitive way of adding details to a geometrical object. Several different techniques have been developed based on the initial texture mapping concept, ranging from the simple regular image texture mapping over bump mapping and parallax mapping to regular displacement mapping. Common to these methods is that they aim to add geometrical details to the object, either by adding visual cues (*e.g.* changing the base color and/or the shading) to simulate greater details, or by actually changing the overall shape of the surface. Either way, the goal is to add these effects without burdening the artist and/or the display system. In this thesis, we present a further development of the texture mapping paradigm, allowing the user to employ unconstrained geometry as a *geometric texture* and merge it onto the surface of another object while deforming it to fit the shape of the target surface.

The inspiration for this came from a somewhat unexpected source: my previous work on terrain rendering. Here, the focus was on creating a flexible and efficient terrain rendering algorithm with support for real-time editing of the terrain. During the work on creating terrain editing options, it became clear that there were many interesting challenges related to this. At the same time, it became apparent that with the rather restricted underlying terrain representation we would not be able to obtain the results we were aiming for. So, rather than restricting our further work to this limiting height map representation of the surface, we changed focus and based the remaining work on a significantly more flexible implicit representation of the surface, *e.g.* a *level set*. Thus, the contributions of this thesis spans two topics, real-time terrain visualization and geometric texture mapping, of which the geometrical texture mapping topic is by far the major one.

Flexible real time terrain visualization: we present a flexible framework for real-time rendering of large textured terrains. The strength of the presented algorithm lies in its flexibility. While it may not be as fast as the current state-of-the-art algorithms, it is significantly more flexible than those systems. In addition to the obvious capability of rendering terrains at interactive frame rates, our system handles arbitrarily large textures, on-the-fly altering of the terrain and even partial specification of the terrain with on-the-fly updating of the terrain if additional data should become available. The system is memory efficient without compromising the quality of the terrain: by employing a carefully designed data layout less than 2.5 bytes is stored in memory per vertex. Finally, the presented system is simple and easy to implement.

Geometrical texture mapping: we present robust and flexible techniques for warping and blending (or subtracting) geometric details, in the form of a geometric texture, onto level set surfaces. The presented methods are based on the use of implicit geometry. This makes it easy to merge the base and texture geometry into a single topologically connected object as well as smoothing the intersection between the base and texture geometry thereby guaranteeing a smooth surface with smooth normals. Furthermore, our mapping employs a flexible particle based parameterization, characterized by the concrete distribution of the particles. This allows us to change the parameterization by changing the way the particles are distributed. To demonstrate this flexibility, we present three different methods for distributing the particles. Finally, we demonstrate how the presented geometrical texture mapping techniques can be applied in an animation context as well.

Acknowledgments

First and foremost, I would like to thank my supervisor, Ken Museth, for his great guidance throughout this entire period. Without his help and support, this thesis would most likely not have been possible. At the same time, I would like to thank my in-house supervisor, Kaj Grønbæk, for believing in me and for guiding me through my studies. I would also like to thank the remainder of the Norrköping/Århus graphics group, in particular Michael Bang Nielsen, Ola Nilsson and Andreas Söderström. Michael has helped me in so many ways, in particular for commenting on this thesis, and for letting me use the software he has written, including his implementation of the DT-Grid data structure, his level set framework, and his scan conversion utility. As Michael, Ola also allowed me to use some of the software he has written, in particular his implementation of the fast sweeping reinitialization algorithm. In addition to this, I have had plenty of help and fruitful discussions with him regarding visualization of my results. The images in figure figure 8.13 are rendered by him. Finally, Andreas supplied the dataset used for the water animation in figure 10.1. In the last phase of my studies, I had the opportunity to work with Serban Porumbescu and Brian Budge. I would like to thank them both for all the hard work they put into the project and for letting me in on the secrets and limitations of their competing project, Shell Maps.

I would also like to thank Peter Ørbæk for his technical aid and discussions related to the terrain visualization algorithm, as well as the rest of the people at 43D, for their collaboration and support on this project. I would like to thank Ole Østerby for always being willing to answer my questions on numerical analysis and math in general. Niels Olof Bouvin for being the LaTeX guru ready to help whenever necessary. Louis Feng for generating the illustrations in figures 8.1, 8.2 and 8.3. Preben Mogensen and the rest of the Palcom group at the University of Aarhus, for giving me the opportunity to take some time off from the project so that I could finish my work on this thesis. And, the authors of the papers [40, 53, 56] for allowing me to reprint figures from their work.

Finally, I would like to thank my family for being very understanding and supportive even in the rough periods of my studies. In particular, I would like to thank my dear wife for being incredibly patient with me, for taking on an extra load at home to allow me to focus on my thesis, and for taking the time to read through my thesis and provide me with valuable comments.

Anders Brodersen,
Århus, March 22, 2007.

Contents

Part I

Overview

Chapter 1

Introduction

The trade-off between efficiency and flexibility is a classical problem in computer science. This is a well known schism between making a tool that does *one* thing to perfection or one that does several things well enough but typically with some trade-offs. A good example of this is the PC versus game consoles: The PC supports a wide application area, including gaming, whereas a game console is obviously tailored for gaming and can utilize all its resources to this one end which makes it superior to the PC for this one purpose. The same trade-off exists in the field of computer graphics, between developing highly efficient single-purpose algorithms and their flexible counterparts that address several complex problems. In this thesis, the focus is primarily on creating *flexible* algorithms, both with respect to terrain rendering and geometric texture mapping. In the following, I will motivate and discuss the trade-offs relating to both topics.

Terrain Rendering

Developing a terrain rendering engine requires one to make a number of design choices to either develop a highly optimized engine useful for a limited set of applications or a more flexible engine that works on a wider range of applications without necessarily excelling in any area. For instance, one has to choose between small or large terrains. Building a terrain engine meant for rendering of large or even massive terrains requires careful design of the data structures and algorithms. Furthermore, a level of detail technique has to be applied to avoid overloading the graphics hardware with too many triangles. This usually has a noticeable impact on the performance of the system. On the other hand, when designing an engine geared towards displaying smaller terrains, memory is less of an issue. This allows for a more hardware friendly memory layout resulting in a higher performance of the system. Furthermore, when focusing on maximizing the performance of a terrain engine, other aspects are often left behind. A good example of this is the geometry clipmap approach [49], which is currently one of the best terrain rendering algorithms in terms of performance and memory requirements. This system easily manages to represent and display a terrain spanning the entire USA at a 30m grid sampling. The limitations of this approach are, that in practice the textures applied to this terrain are either procedurally generated on the fly or limited to a 1 dimensional texture (*e.g.* a contour map). Furthermore, to facilitate the representation of the massive terrains, all terrain data are compressed in a pre-process (The USA dataset is compressed from 40GB to 355 MB, with an rms error of 1.8m). This, however, rules out performing any kind of run time alterations of the terrain. While systems such as the BDAM [14] exhibit similar performance as the geometry clipmap approach without sacrificing the ability to display high resolution textures, they still require the terrain data to go through a long pre-processing step. Thus, to the best of our knowledge, no system exists which is capable of displaying large scale terrains with high resolution textures and still allow the terrain to alter it's shape at runtime. While most terrain rendering applications have little use for the ability to edit the terrain at runtime, it is critical to a few important applications. Consider for example the work of a landscape architect [11]. When designing a park, it is for example desirable to change the flow of a small stream through the park, or to level out a part of the terrain. In this case, it is essential for the architect to have access to visualization tools that facilitates interactive editing of the terrain. This is the inspiration for one of the challenges this thesis is addressing, namely developing a system that is capable of displaying large terrains with high resolution textures at interactive frame rates - without sacrificing the ability to interact with and change the shape of the terrain.

Geometric Texture Mapping using Level Sets

During my initial work on terrain visualization, or more specifically editing of terrains, I quickly discovered a major limitations of the underlying terrain representation. Using a *height map* representation of the terrain severely limits the kind of terrain that can be represented, and consequently also the capability of a terrain editing tool. Using this representation, all editing operators are limited to applying a vertical offset to the affected parts of the terrain. Similarly, due to the limitations of the height map representation, caves, bridges etc. cannot be represented. This limitation inspired me to look for an alternative approach to editing or adding of details to terrains. After carefully analyzing the problem at hand, I realized, in conjunction with my supervisor, that we were in fact seeking the solution to a problem that had applications reaching far beyond just editing of terrains. In fact, we realized that we were looking for a more generic geometric manipulation tool that would allow one to easily add (or remove) any desired kind of details to any given base surface. In the search for a generic solution, texture mapping was a good source of inspiration; in its simplest form, texture mapping is conceptually very close to what we were looking for: a quick and intuitive way of adding details to a surface. In our context, the limitation of regular image texture mapping is that details are added to the surface simply by modifying the color of the surface rather than modifying the actual geometry. Amongst the more advanced texture mapping techniques, displacement mapping [75] is more in the line of what we were looking for. By encoding a height field into a texture, this method applies an offset to the vertices of a given base surface. Still, the deformation of the surface enabled by these techniques are limited to simple normal offsets of the surface, which is not much different from the type of deformations that could be performed on a terrain. Our idea was to develop a more general technique that would allow us to easily apply any kind of surface deformation. Or, to use a graphics terminology, to employ unconstrained geometry as a *geometric texture* and merge it onto the surface of another object while deforming it to fit the shape of the target surface. Rather than restricting this to the limited height map representation of the surface, we change focus and base the remaining work on a significantly more flexible implicit representation of the surface, *e.g.* a *level set*. Based on the level set representation of surfaces, we present a novel framework for surface editing using arbitrary geometrical objects as *textures* that can be either added onto or subtracted from the base surface. In both cases, the texture geometry is warped to follow the local shape of the base surface.

We are not the first to pursue this idea. Previous work in this area include the *Geometric Texture Synthesis by Example* approach by Bhat *et al.* [7], which, similar to our system, is based on an implicit representation of the geometry, and the more recent *Shell Map* method of Porumbescu *et al.* [67], which was developed in parallel with our work. The keywords that makes our system stand out are *flexibility*, *quality*, and *control*. In particular, the fact that the method of Bhat *et al.* is based on texture *synthesis* reduces the flexibility of the system. Basically, the user supplies 3 surfaces and the system attempts to *do to surface A, what was done to surface B in order to obtain surface C.* In contrast, using our method, the user has full control over where a geometric texture is applied, how many instances are applied as well as their size and orientation. These properties are shared by the Shell Map method, which due to the explicit representation of the base and texture geometry, is relatively fast. Our approach on the other hand can handle more complex topologies, produces connected surfaces, is significantly more flexible and generally leads to better results with less distortion.

To summarize, this thesis spans two topics, terrain visualization and geometric texture mapping. Although both topics are important to my PhD work, geometric texture mapping is by far the one that I have spent most of my time studying. Consequently, geometric texture mapping is to be considered the primary topic of this thesis, whereas the terrain visualization topic is should considered the secondary topic.

Contributions of the Thesis

The main contributions related to the terrain rendering are:

Optimized Geometrical MipMapping We present a number of optimizations to the geometrical mipmapping terrain renderer [18] that allows us to significantly reduce the amount of memory used to represent the terrain.

Support for high resolution textures through texture tiling Careful organization of the terrain data allows us to efficiently texture the terrain using very high resolution textures tiled into smaller sub-textures displayable by current graphics hardware.

Support for on-the-fly alteration of the terrain By taking advantage of our memory layout, we can allow the terrain to be modified at run time without imposing any significant impact on the overall performance of the system.

The main contributions related to geometric texturing are:

Flexible volumetric parameterization. We compute a low distortion volumetric parameterization with a minimum of user-interaction. Our parameterization does not depend on prior surface texture coordinates. Instead, it is based on a local parameterization generated on the fly, using a simple and easy to use point and click interface. Furthermore, our parameterization is characterized by the distribution of a set of particles, but is independent of how this distribution of the particles is obtained. This means that the particles can be distributed in a number of different ways, allowing for a vast number of unique mapping effects.

Implicit geometry mapping with smooth blending. We complement existing explicit geometry mapping techniques by using an implicit approach which smoothly blends mapped geometry to create closed surfaces suitable for rendering or for further manipulation or simulation. We do this using compact level set representations of the base and texture surfaces. Our mapping is based on a radial basis function interpolation of discretely sampled texture coordinates, giving us a smooth (differentiable) mapping. This is the first general texture space to shell space mapping technique utilizing implicits that we are aware of.

Fast semi-implicit geometry mapping. We also introduce a near real-time *semi-implicit* mapping approach that combines an implicit level set representation of the base surface with an explicit polygonal representation of the mapped textures. This technique is useful as a preview tool (prior to implicit mapping) and as a stand alone method for mapping explicit geometry.

Shell Map hybrid mapping. To overcome some of the performance issues with the implicit mapping, we further introduce a hybrid mapping combining our parameterization with the interpolation scheme used for the Shell Map approach [67]. The resulting mapping handles both explicit and implicit textures, and it is significantly faster than the radial basis function interpolating method. The price we pay for this is a mapping based on barycentric interpolation, and it is thus only C^0.

Texture based animations. We present different animation techniques based on our geometric texture mapping techniques. These techniques provides support for animating geometry textures, geometric texturing on animated base geometry, animations of the size and position of the mapped texture geometry on the base surface as well as a combination of these three techniques.

This thesis summarize my work carried out as a Ph.D. student at the University of Aarhus, Denmark from September 2003 to March 2007. So far, the work described in this thesis has resulted in two publications. The first one is a conference paper, on which I was the sole author, presented at the 3rd international conference on computer graphics and interactive techniques in Australia and

Southeast Asia (GRAPHITE 2005) entitled *Real-Time Visualization of Large Textured Terrains*. The second paper is a journal paper entitled *Geometric Texturing Using Level Sets*. This paper, which is conditionally accepted by IEEE *Transaction on Visualization and Computer Graphics*, is co-authored with my supervisor Ken Museth as well as Serban Porumbescu and Brian Budge, both from the University of California, Davis. These two papers are included for reference as appendices of this thesis. All the material described in those papers is however also described in this thesis.

Although the work presented in this thesis is indeed my own, several people have been to involved at different stages. Consequently I shall use "we" throughout the remainder of this thesis to acknowledge these collaborations.

The structure of the thesis is as follows. First, Part II presents our approach to visualization of large textured terrains. The main focus in this part of the thesis is the further development of the geometrical mipmap terrain rendering engine to facilitate visualization of large scale terrains without sacrificing the flexibility of the original algorithm.

Following this, Part III expands on the surface editing paradigm briefly introduced in Part II in the context of terrain rendering, and presents our resulting framework for a more generic geometric texture mapping technique. Following a brief introduction to the topic in Chapter 5, Chapter 6 introduces some of the required background theory on level sets and implicit surfaces, and Chapter 7 reviews some of the previous research related to our work. Chapter 8 presents the core of our proposed geometric texture mapping framework: This includes a flexible particle based local parameterization, two mapping methods used for mapping explicitly respectively implicitly defined geometric textures onto the base surface and finally, the tools needed to create a single topologically connected surface with a smooth intersection between the two (or more) previously separate surfaces. Chapter 9 presents a hybrid approach to solving the same problem, combining our particle based parameterization with the barycentric interpolation based mapping of Porumbescu *et al.* [67]. This method presents a significant speed up to the mapping of implicit geometric textures at the cost of a reduced visual quality. In Chapter 10, we demonstrate how the presented geometrical texture mapping methods can be applied to animation. We present methods for handling both animation of the base surface, key frame animations of the texture and a combination of both. We round off Part III with a brief evaluation of the proposed methods and a comparison to similar algorithms.

Finally, in Part IV, we sum up the contributions of this thesis and give a short presentation of some of the issues that we would like to address in the future.

Part II

Terrain Visualization

Chapter 2

Introduction

Real-time visualization of large terrains has been an active area of research for more than a decade. In the past few years, as a natural result of the constantly increasing capabilities of modern graphics hardware, the focus has shifted from CPU intensive algorithms, where level of detail is determined per triangle, to the more GPU intensive algorithms, where the level of detail is determined for a group of triangles rather than for each individual triangle. The result is a much simpler and faster level of detail calculation. The price we pay for this is that we need to render the entire group at the same level of detail, which has to be high enough that nothing is drawn at too low a level of detail. Thus, we are likely to render more triangles than actually required. This is however not a big problem, as current and future graphics hardware is capable of rendering the increased number of triangles with little or no extra cost.

There are several aspects involved in rendering large terrains, of which render speed and memory consumption obviously receive by far the most attention. Often, a terrain will be too large to be stored in main memory. It is therefore crucial to keep the overall memory requirements, measured in bytes per vertex, as low as possible, to avoid having to constantly swap data in and out of main memory. As for the rendering speed, often a terrain consists of far more triangles than the hardware can possibly display at interactive frame rates. This is typically handled using a combination of back face culling, frustum culling, level of detail and possibly also occlusion culling [52]. In most cases, however, the equally important issue of how to add a texture map to these large terrains has been ignored. The problem arises because modern graphics hardware is limited to displaying textures of sizes up to 2048×2048 or 4096×4096, which for most applications (particularly games) is more than sufficient, but for a terrain covering an area of 80km^2, such a texture would only provide one texel per 20 or 40 meters. Game programmers typically solve this problem by using several repeating textures, blended according to height, slope etc. to hide the repeating pattern. If on the other hand, the texture is a map or a (satellite) photo, that approach just is not good enough.

In the following, we demonstrate how a geometric mipmap based terrain engine [18] can be adapted to efficiently render large scale terrains and at the same time handle textures that are larger than the maximum texture size displayable by the graphics hardware. By combining carefully designed data structures with the use of vertex programs, the proposed system requires less than 2.5 bytes per vertex. The presented algorithm is simple, yet powerful and flexible enough that it remains possible to alter the terrain at runtime with no significant impact on the performance. Before we dwell further into the details of our algorithm, we will start by reviewing some of the previous work related to our work.

Chapter 3

Existing Approaches to Terrain Visualization

Most terrain rendering algorithms, including the one presented here, focus on rendering of height fields; that is, a two dimensional array of height samples. The algorithms for interactive rendering of height fields are typically divided into two categories: Algorithms that take advantage of the regular structure of the datasets to create an efficient hierarchical representation of the terrains; and algorithms based on a more general unconstrained triangulation of the terrain, triangulated irregular networks, or TINs.

The most popular algorithms in the first category are the quad-tree based approach of Lindstrom et al. [43], the triangle bin-tree based ROAM algorithm [23] or longest edge bisection as used by Lindstrom et al. [44, 45]. They all generate the mesh at run-time by progressive refinement or simplification. The refinement process starts with one or more isosceles right triangles, which are recursively refined by bisecting the longest edge, thereby creating two new isosceles right triangles, until a given mesh quality is achieved. Simplification is done by reversing those steps. When changing the viewpoint, the mesh is regenerated, either from scratch [43–45], or by refining and simplifying the existing mesh to fit the new viewpoint [23]. The first four levels of a triangle bin-tree is depicted in figure figure 3.1. Whether or not a given triangle is to be refined (or simplified) is determined by comparing a user defined threshold against an error associated with each triangle. This error is typically some variant of a *screen space error*, meaning that it is a measure for the visual error introduced by *not* refining this triangle (or in the case of simplification, the visual error introduced by merging this triangle and a triangle sharing an edge with it into one triangle), although more simple error metrics such as distance to the viewpoint are also used.

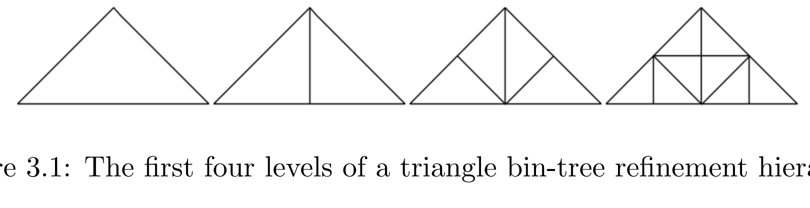


Figure 3.1: The first four levels of a triangle bin-tree refinement hierarchy.

In contrast, algorithms based on triangulated irregular networks pre-calculate a hierarchy of optimized meshes at different resolutions, which at runtime are combined to give the desired mesh quality from a given view point. The runtime performance of TIN based terrain algorithms are usually higher than that achieved by the runtime refinement algorithms, and they have been shown [25] to generate meshes of higher quality (smaller error) than other algorithms using the same number of triangles. However, the TIN based algorithms are usually much harder to implement, and the pre computation-time needed is considerable. One of the most prominent example of algorithms from this category is the view dependent progressive meshes of Hoppe [35].

Inspired by the massive evolution in consumer graphics hardware, a new group of algorithms have started to appear. Taking advantage of the increased triangle throughput of modern graphics hardware, existing algorithms have been modified [14, 42] and new algorithms have been invented [18, 49]. What these new algorithms have in common is that they utilize far simpler algorithms which, at the cost of rendering more triangles than required to achieve the desired mesh quality, has a significantly lower CPU overhead. In other words, most of the work is shifted from the CPU to the GPU. An interesting example of this is the BDAM algorithm [14], which is a combination of the ROAM algorithm [23] and a TIN based approach. The BDAM algorithm uses the same triangle bin-tree layout as used by the ROAM algorithm, but rather than having each node in the tree correspond to a single triangle, each node contains a small chunk of triangles forming a highly optimized TIN, typically containing between 200 and 8000 triangles. These TINs, which are constructed offline in a preprocessing step, are constructed such that they match up with the neighboring TINs of the same level or one level coarser sharing the nodes longest edge, and with the TINs of the same level or one level finer sharing one of the two of the nodes shortest edges. This constraint combined with the properties of the triangle bin-tree guarantees a fully connected triangulation of the terrain. The terrain is then drawn by traversing the triangle bin-tree. Once a branch of the tree is reached matching the desired quality, the entire chunk of triangles contained in that node is rendered, and traversal of that branch is terminated.

Although texture mapping is an important aspect of terrain rendering, most papers completely skip this issue [18, 43, 45], others leaves it for future work [23, 42] or only very briefly touches the subject [49]. The standard approach for the few systems that actually handle large textures, is to partition the texture into tiles, binding each tile to a certain part of the terrain [35]. In some cases the textures are arranged into a pyramidal structure to facilitate texture level of detail along with the geometric level of detail [14, 22]. Google Earth [31] appears[1] to be using a tiling scheme where the terrain is split up in small tiles each with its own texture. Textures and tiles are then streamed into memory as they are needed, starting with a low resolution model and texture and then refining the model as the required data gets loaded. This way, they can keep the entire world in a huge database while only keeping the currently used

[1]As Google Earth is a proprietary system, no details about the algorithms used are released to the public. Hence, the discussion of their methods is based solely on the system in use.

neighborhood in memory. In all cases, the texture handling is tightly coupled with the geometrical level of detail algorithm; the only truly general approach, in the sense that is completely independent from the geometrical level of detail algorithm used, is the clip-map [78], which unfortunately requires special hardware. The approach presented here is also based on texture tiling, but with texture level of detail limited to standard hardware controlled mipmapping. This system can easily be extended with texture management as in [22] and [14] with only minor changes to the rest of the system.

The system presented here is developed with flexibility in mind. Thus, while we cannot compete performance wise (rendering speed) with most of the high performance GPU optimized systems such as the geometry clipmap system [49] or the BDAM family [14, 15, 29], it does perform better than these systems in terms of flexibility. Most notably, our system allows for changing the terrain at runtime without incurring a noticeable performance penalty. In terms of rendering speed, our system still outperforms the more CPU intensive systems such as [23, 43–45].

Chapter 4

Optimized Geometric Mipmap Terrain Visualization

In the following chapter, we present our representation of large terrains as well as the algorithms we have developed to enable visualization of the terrain at interactive frame rates. The terrain engine presented here has been implemented as part of a commercial program; a program requiring more than just fast rendering of the terrain. We require the ability to render the terrain with a texture that is larger than the textures displayable by current hardware. Furthermore, we need to add support for altering the terrain on the fly; a requirement that effectively rules out any expensive pre-computations that would need to be repeated when the terrain is changed. Although our approach is based on an existing terrain rendering algorithm, we presents a number of novel ideas. Our main contributions are

1. We present a number of optimizations that allows us to significantly reduce the amount of memory used to represent the terrain.

2. Careful organization of the terrain allows us to efficiently texture the terrain using very high resolution textures.

3. By taking advantage of our memory layout, we can allow the terrain to be modified at run time without imposing any significant impact on the overall performance of the system.

4.1 Data Structures and Memory Layout

In designing a system that satisfies all of our requirements, we have turned to the GeoMipMap algorithm [18], where the terrain is divided into smaller patches, called GeoMipMaps, of size $(2^n+1)\times(2^n+1)$. The original GeoMipMap algorithm is clearly designed for smaller scale terrains used in games, as nothing is done to reduce the amount of memory used by the system[1], but we have extended it to be suitable also for rendering larger terrains.

Our system uses a three level data structure:

[1]More than 12 bytes are used per vertex.

```
class GeoMipMap
{
   //Current level of detail
   unsigned short LoD;
   //delta max for each detail level
   unsigned short *deltaMax;
   unsigned short *heights;
   VboElement *vboElm;
   //Bounding box
   unsigned short ymin, ymax;
};
```

Figure 4.1: Class Declaration of GeoMipMap. The VboElement entry is described in section 4.4.

- At the bottom level we have a 2D array of *GeoMipMap* structures, containing the heights of the vertices in that particular patch as well as some information needed for the level of detail algorithm. The class declaration is listed in figure 4.1.

- At the mid level, we have a 2D array of the *MapBlock* structure. This structure is used for controlling the texture mapping, ensuring that a given texture is applied to the correct GeoMipMaps. Each MapBlock holds a reference to $t_u \times t_v$ GeoMipMaps as well as one or more textures, which are used for decorating all of the referenced GeoMipMaps. Like [22], textures can be combined using any of the blending operators supported by OpenGL. For efficiency reasons, this structure also holds a (compressed) bounding box, and its indices into the 2D array of MapBlocks.

- Finally, at the top level we have a single *Terrain* structure, which forms the interface between the terrain engine and the rest of the system. The Terrain structure also holds all data that can possibly be reused between different MapBlocks or GeoMipMaps, such as the size of the GeoMipMaps and MapBlocks, the indices, render flags etc.

Storing the heights in the GeoMipMaps instead of having one large 2-dimensional array means that heights along the shared edge of two GeoMipMaps need to be stored in both GeoMipMaps. However, choosing this layout allows us to reuse the same set of indices for each GeoMipMap as well as a generic set of x and z coordinates. Thus, the net result is a reduction in memory usage rather than an increase.

4.2 Level of Detail

The level of detail algorithm used in our system is based on the geometrical mipmapping algorithm presented by de Boer [18]. We follow the convention from [18] where the y coordinate of our vertices represents the height of the terrain.

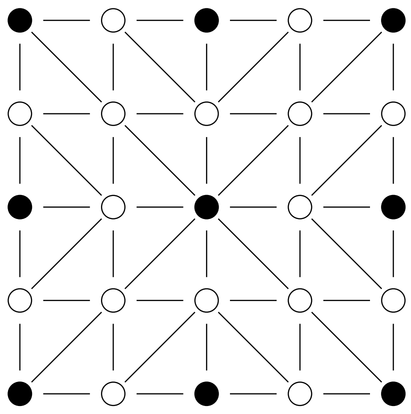


Figure 4.2: Mesh layout for a map size of 5×5. The black circles are the vertices used for lower detail level mesh (level 1). Both the white and black circles are used for the highest resolution mesh (level 0).

The terrain is subdivided into a number of smaller patches of containing $(2^n + 1) \times (2^n + 1)$ samples (typically either 17×17, 33×33 or 65×65). Each patch, also called a GeoMipMap, is then rendered at either full resolution, every second vertex only, every fourth vertex only etc. depending on the desired level of detail. In other words, the desired level of detail is determined for the entire GeoMipMap, making the refinement process both simple and fast. Changing from one GeoMipMap level to the next simply amounts to removing every second vertex in both directions, thus reducing the number of vertices from $(2^n + 1) \times (2^n + 1)$ to $(2^{n-1} + 1) \times (2^{n-1} + 1)$, see figure 4.2. At creation

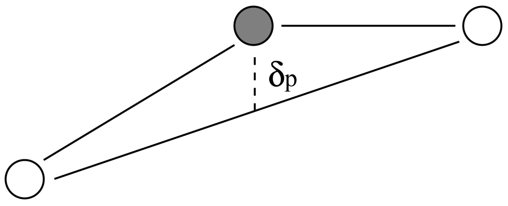


Figure 4.3: Geometrical error introduced by removing a single vertex as seen from the side. The length of the dotted line is the geometrical error introduced by removing the gray dot, and replacing the two lines from the white dots to the gray dot by a straight line between the two white dots. The maximum geometrical error is defined as $\max(\delta_{p_i})$, where δ_{p_i} is the error introduced by removing vertex $p_i \in P_l$, and P_l is the set of vertices removed when changing from level 0 to level l.

time we calculate, for each level of each GeoMipMap, the maximum geometrical error caused by changing from level 0 (highest resolution) to that level, see figure 4.3. Selection is then done at run-time, given the current view parameters and the world space bounding box of a GeoMipMap, by calculating the maximal geometrical error allowed inside that bounding box, with respect to a user supplied threshold. This value is compared to the error values for the GeoMipMap, and the lowest detail level with a maximum error below this value is chosen to be rendered.

Calculating the maximal geometrical error allowed is done using the *distant terrain* LoD scheme described in [24], but a more accurate scheme with no assumptions, or a simpler scheme, based for example only on distance from the view point, can easily be used instead.

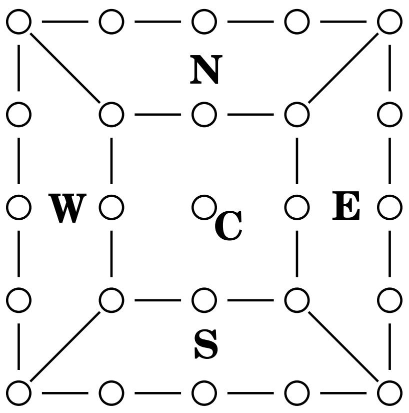


Figure 4.4: The 5 regions of a single GeoMipMap implicitly defined to simplify the task of avoiding cracks in the mesh. Note that when no more than 3x3 vertices remain, the center region is empty.

If two neighboring GeoMipMaps are rendered at different levels of detail, then the tessellation of their shared edge will be inconsistent, leading to holes, or cracks, in the mesh. To avoid these undesirable artifacts, we divide each GeoMipMap into 5 separate regions, see figure 4.4. While the tessellation of the center region is based entirely on the currently selected level of detail, the tessellation of the four border regions are based on the currently selected level of detail as well as the level of detail of the neighbor GeoMipMap sharing an edge with that region. If the neighbor is at a higher resolution, we add the missing vertices to the shared edge to ensure a consistent tessellation between GeoMipMaps, as demonstrated in figure 4.5. This is the opposite approach of [18] and [41], where vertices are removed rather than added. Removing vertices instead of adding them results in fewer triangles to render, but at the cost of removing triangles with a potential geometrical error larger than the calculated maximum. We believe that providing a tessellation of a (potentially) lower quality than implied by the user specified threshold is an ill design choice, and therefore prefer the slightly increased triangle count.

4.3 Reducing Memory Requirements

An important benefit from using a regular grid height map for terrain visualization is that the x and z coordinates can easily be calculated at runtime, and thus need not be stored. To avoid having to recalculate the x and z coordinate

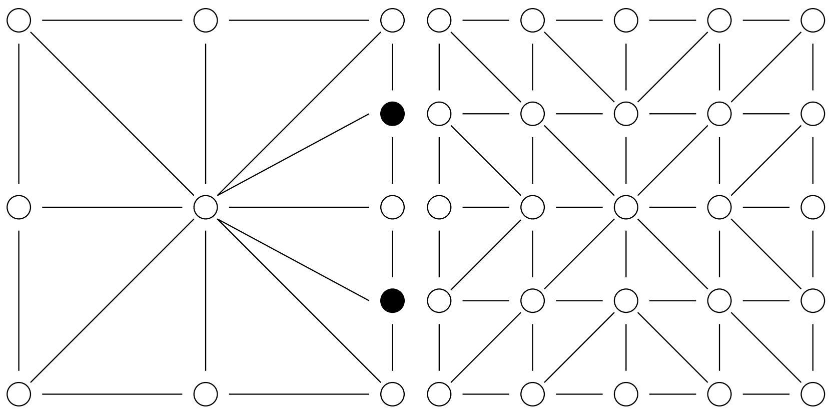


Figure 4.5: When two neighboring GeoMipMaps have different resolution, extra vertices (black dots) are added to the lower resolution GeoMipMaps representation of the shared edge, making the two edges identical.

whenever we need to draw a triangle, we take advantage of the fact that each map has its own copy of the y coordinates making up that map. By precalculating the x and z coordinates of *one* map, we can reuse these coordinates for all other maps, by simply translating the vertices to the position of the new map relative to the original map.

In practice, this means that we store the x and z coordinates of the lower left block in one array, and the y coordinates for each of the $n \times m$ maps in separate arrays (owned by the corresponding GeoMipMap object). When rendering, we then pass in the x and z coordinates using the 0th OpenGL vertex attribute array, and the y values are passed in through the 1st vertex attribute array. We then use a simple vertex program to assemble this into a full vertex.

Using the GeoMipMap structure listed in figure 4.1, the amount of memory needed to store a single GeoMipMap of size 17×17 is 604 bytes. As the MapBlock structure is a lightweight structure containing only a compressed bounding box (2 shorts), two indices, one or more texture ids, and a list of visible maps, it contributes little to the overall memory consumption. And as the top level Terrain structure is never instantiated more than once, even with all the indices stored (see section 4.4), the total memory consumption for even moderate size terrains[2] is less than 2.5 bytes per sample[3].

4.4 Rendering the Terrain

Vertex Buffer Object Management In order to achieve high frame rates, it is important that we have all the needed vertex data in graphics memory, however due to the large memory requirements of the textures combined with the additional use of graphics resources by the application itself (that is, textures and vertex data for other geometry displayed along with the terrain), it is equally important that we do not waste resources on parts of the terrain that

[2]The larger the terrain, the smaller is the influence of the single Terrain structure

[3]1.5 bytes if we stored heights as bytes instead of floats as *e.g.* in [43]

are not drawn.

For this reason we have implemented a manager for OpenGL vertex buffer objects. Each GeoMipMap stores a pointer to a VBOElement, which is a wrapper around an OpenGL vertex buffer object [59]. VBOElements are managed using a standard *least recently used* approach. Whenever we are about to render a GeoMipMap with no VBOElement assigned, we revoke the least recently used, and assign it to the GeoMipMap.

An important detail for this to work properly is that no VBO is allowed to be used more than once per frame. Should a VBO be used twice in the same frame, then the least recently used sharing approach will often cause most if not all VBOs to be updated, effectively killing performance. To avoid this, we keep track of how many VBOs are used each frame. If at any one time we have used all VBOs in our queue, and a GeoMipMap makes a request for a VBOElement, a new one is immediately created instead, growing the queue to fit the current requirements. If the number of VBOElements in use is less than the current size of the queue, we slowly shrink the queue[4] in order to reclaim graphics resources when possible.

Pre-calculated indices For efficiency, we pre-calculate the indices for all possible levels of detail configurations for a GeoMipMap and its 4 neighbors. For a GeoMipMap of size 17×17 this results in 354 different sets of indices, taking up a total of 88896 bytes of memory (58718 after being converted to tristrips), which are turned into triangle strips, and stored in a single VBO. A more memory efficient approach would be to separately render each of the sections of the five section GeoMipMap layout presented in section 4.1. This reduces the total memory requirement for the (stripified) indices to 4338 bytes (for a 17x17 GeoMipMap), but at the cost of having to call glDrawRangeElements five times per GeoMipMap rather than one. Using maps of size 17×17, we have seen a performance increase of up to 7% when using only one draw call and therefore recommend using that approach, however for map sizes larger than 17×17, the memory needed to store the indices may become a problem[5], in which case drawing the five regions separately may then be advisable.

Frustum Culling View frustum culling is performed using the optimized two point axis-aligned bounding box/plane intersection test with masking by Assarsson and Möller [6]. Rather than introducing a new hierarchy only to be used for culling, we have chosen to use the already existing 3 level layout presented in section 4.1. Because the bounding boxes of all three levels are aligned to the same coordinate system, the n-vertices and p-vertices used for culling, see [6], are the same for all structures, and therefore need only be found once per frame. As a result, frustum culling using the existing data structures is fast, even without introducing any additional data structures.

[4]By removing at most one element per frame

[5]Stripified indices for a 65×65 GeoMipMap requires 1238716 bytes.

Texture Handling As a consequence of splitting the terrain into smaller patches, handling of high resolution textures becomes relatively straight forward. Our approach is to cut the high resolution texture into smaller sub-textures, each sub-texture being small enough to be displayable by the graphics hardware. The high resolution texture is divided in a way that ensures that each sub-texture fits exactly $n \times m$ GeoMipMaps. Each sub-texture is then assigned to a MapBlock controlling exactly the $n \times m$ GeoMipMaps covered by that sub-texture. During the cull phase, each visible GeoMipMap is added to a visibility list of the corresponding MapBlock. The visible GeoMipMaps are then rendered, one MapBlock at a time, thus requiring the textures of that MapBlock to be bound only once per frame.

To avoid visible texture seams, the sub-textures are generated such that two adjacent sub-textures overlap by a one texel border. Splitting textures is done in an offline step for image textures that are independent of the actual terrain, while textures, such as light maps, that are tightly coupled to the geometry are generated and subdivided at runtime.

An outline of the rendering loop, including culling, level of detail calculations etc. is depicted in Algorithm 1.

```
Function Render-Terrain()
visibleMapBlocks.clear();
foreach MapBlock mb in mapBlocks do
    if mb is visible then
        visibleMapBlocks.add(mb);
        foreach GeoMipMap gmm in mb.geoMipMaps do
            if gmm is visible then
                mb.visibleGeoMipMaps.add(gmm);
                gmm.calculateDesiredLoD();
            end
        end
    end
end
foreach MapBlock mb in visibleMapBlocks do
    mb.setupTextures();
    mb.updateVertexProgramState();
    foreach GeoMipMap gmm in mb.visibleGeoMipMaps do
        gmm.getNeighbourLoDs(&n, &s, &e, &w);
        gmm.setupVertexArrays(n, s, e, w);
        gmm.updateVertexProgramState();
        renderPatch();
    end
end
```

Algorithm 1: Pseudo-code describing the rendering procedure.

4.5 Editing the Terrain

Most current terrain rendering systems assume the height-map to be static. With this assumption, it is acceptable to perform any number of expensive pre-calculations, as for example is the case in [49], where a 40GB terrain dataset is compressed in a five hour long preprocessing step. While this assumption is natural for a lot of applications, *e.g.* in a flight simulator or a racing game where one would expect the terrain to remain unchanged, there are some applications where this is not an acceptable assumption. One such example is an application targeted towards landscape architects. For example, when designing a new park it is often necessary to remove large parts of the terrain in order to level the area, or it could be required to change the path of a stream etc. In this case, it is necessary to be able to change the shape of the terrain at run time. As our system is in fact targeted towards these kind of applications, we intend to allow run-time alterations of the terrain. This can be done using an operator much like the polygon sculpt tool in Maya [4]. Such an operator would affect all vertices within a given distance from the point at which the operator is applied. Depending on the current configuration of the operator, this will either raise or lower the terrain within the region of influence of the operator, with the center being raised or lowered the most followed by a smooth falloff for vertices closer to the edge of the region of influence.

When the users alter the terrain, all the affected vertices must of course be updated. This triggers an update of the bounding boxes and geometrical errors thresholds for each GeoMipMap containing updated vertices. Fortunately, with the data layout described above, recalculating the bounding boxes and geometrical errors for the level of detail for a limited number of GeoMipMaps can easily be done without a noticeable impact on performance. Changing the height of some of the vertices also changes the shading of the terrain, which means that the light maps needs to be recalculated as well. This is a bit more costly than recalculating bounding boxes and geometrical error thresholds. Fortunately, the visual impact of the light maps are subtle, and we can therefore justify a short delay in the recalculation as long as the rest of the system remains unaffected. This means that we can dedicate a separate (low priority) thread to the task of recalculating the light maps. This way, they will be recalculated using only *spare* CPU cycles, and thus only incur a minor performance penalty. As the topology of the terrain is unchanged, indices etc. need not be recalculated.

4.6 Compressing Terrain Data

One of the most important aspects of a terrain rendering algorithm is keeping the memory usage at a minimum due to the large data set sizes. Section 4.3 showed how our system uses less than 2.5 bytes per height sample, however Losasso et al. [49] managed to get a memory requirement far below one byte per height sample by compressing the heights.

Unlike Losasso et al., we are not able to perform an expensive offline compression of the heights, as this would have to be repeated whenever the user alters the terrain. We can, however, implement a small scale compression of our heights by compressing the heights in each GeoMipMap separately. When we then need the uncompressed heights for a given GeoMipMap, they are uncompressed into a different array and kept as long as the GeoMipMap is in use. When no longer needed, the heights, if changed, are compressed, and the old compressed heights are overwritten.

Clearly, this will not have the same impact as compressing the entire terrain at once, as we will have to restrict ourselves to a compression scheme optimized for speed rather than compression ratio. It is however something that can be done at run time. Primarily because only a few GeoMipMaps need to have their heights compressed at a time, and as with updating the light maps, this can be done using spare CPU cycles *in the background*.

Due to time constraints, we have unfortunately not yet had the opportunity to implement and test the impact of compressing the height data.

4.7 Results and Evaluation

Our primary tests of the proposed system were performed on a laptop computer powered by an Intel Pentium M 1.5GHz processor with one gigabyte of memory and an nVidia GeForceFx 5650 Go chip with 128MB dedicated graphics memory.

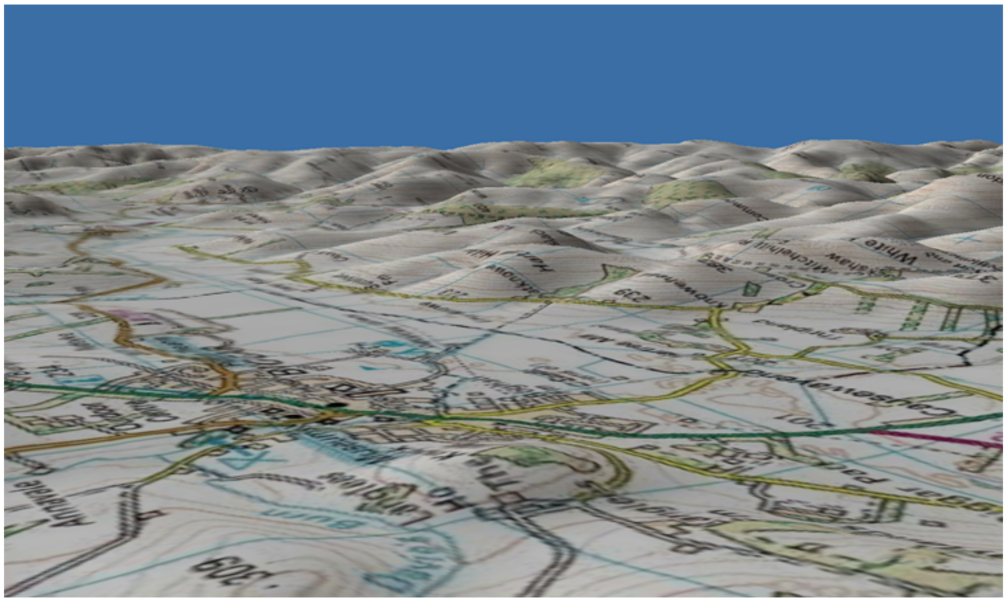
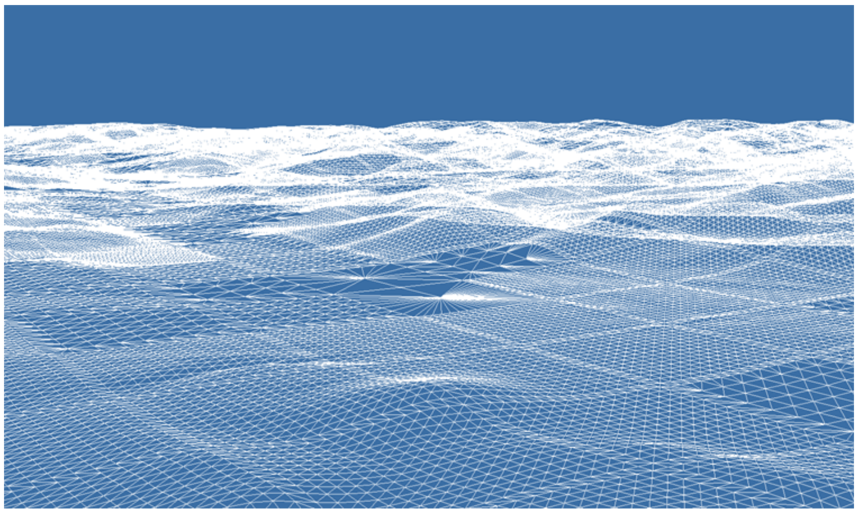


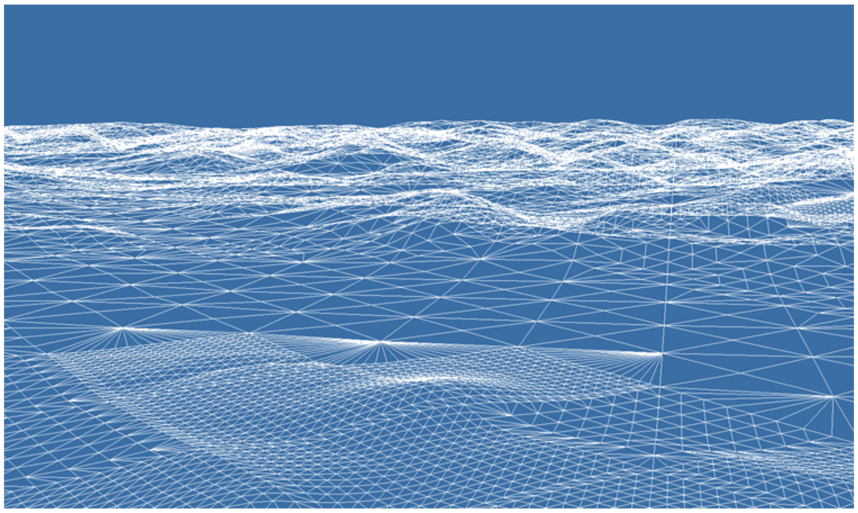
Figure 4.6: View of the Broad-Law terrain

Our test terrain is an area in Scotland known as Broad Law. The terrain is made up of 8193×8193 height samples, and rendered with a 8192×8192 RGB image tiled into 8×8 textures of size 1024×1024 and a 1024×1024 light map also tiled into 8×8 sub textures, see figure 4.6.

With a screen space error, τ, of one and a GeoMipMap size of 17×17,



(a) $\tau = 1$ pixel



(b) $\tau = 3$ pixel

Figure 4.7: Wire frame rendering of terrain with different error thresholds.

the terrain is rendered at on average 70 frames per second at a rate of up to 35MΔ/sec.

Choosing the optimal size for the GeoMipMaps is a matter of balancing the number of OpenGL function calls versus the number of unnecessary triangles drawn. When we decrease the size of the GeoMipMaps, more draw calls are required to render the terrain, but having too many draw calls can severely hurt performance. Similarly, by increasing the size of the GeoMipMaps, more redundant triangles are likely to be drawn. According to our initial tests on the above mentioned system, the best performance is obtained when using GeoMipMaps of size 17$\times$17, at which point the balance between the number of draw calls and number of unnecessary triangles drawn seems to be optimal. This correlates with the results obtained by Larsen and Christensen [41]. Further tests performed on a significantly faster system, powered by a nVidia GeForce 7800

GT card confirmed our suspicions that the optimal GeoMipMap size depends heavily on the underlying hardware. On the faster system, using GeoMipMaps of size 17×17, the limiting factor appeared to be the number of OpenGL function calls rather than the number of triangles rendered. On this latter system, the highest performance was observed when using a map size of 65×65.

Figure 4.7 shows a wire frame rendering of the terrain from figure 4.6, with a screen space error of 1 and 3 pixels.

Our system does have one obvious flaw: In some cases, *e.g.* when viewing the terrain from above, most other algorithms are able to simplify the terrain to just a few triangles. As we are limited by the size and number of GeoMipMaps, our lowest resolution terrain is of a much higher resolution. In practice, however, this is of little importance, as the rendered mesh is still of relatively low resolution, and more importantly, the situations where the problem arises are relatively rare.

While the presented system cannot compete with current state-of-the-art rendering systems [14,15,29,49] performance wise, our system is by design more flexible than most previous systems. In addition to the support for displaying high resolution textures, which for example is not clear how to do with the geometry clipmap algorithm [49], and the terrain editing facilities which to the best of our knowledge has not been handled by any previous system[6], our system capable of handling missing parts of the terrain. If a part of a terrain is missing, the `heights` pointer (see figure 4.1) in the GeoMipMaps covering this area will simply be `null` pointers and those GeoMipMaps will be ignored during rendering. If at some point the data for that part of the terrain becomes available, the affected GeoMipMaps are simply updated with this new data, and will subsequently be treated like all other GeoMipMaps. This is in contrast to having to represent those unspecified areas as flat areas in the terrain. While this certainly is an option, is leaves the user in the blind, as it will not be clear if the areas are unspecified or that part of the terrain just is meant to be flat. And even with this approach, systems such as the BDAM family or the geometry clipmap will not be able to handle the case where the data becomes available during runtime.

The system presented here is currently being used (and have been for almost two years) in the Topos$^{\text{TM}}$ visualization software package [1] (see figure 4.8).

Using the kind of modifications to the terrain discussed in section 4.5, we allow the user to easily change the flow of a stream, or to level a part of the terrain to help visualize a (future) building site etc. While this may be sufficient for some applications, it also reveals the limitations of the underlying geometrical representation of the terrain: With the height map representation of the terrain, we can only raise or lower the terrain, that is, no matter what operator we apply to the geometry, only the y-coordinate of the vertices changes. If more advanced alterations of the geometry were to be applied, we would need

[6]Although some of the previously presented systems, in particular the geometrical mipmap [18] which our system is based upon but also most of the CPU intensive systems [23,43–45], are likely to be able to handle this, most of the recent systems [14,15,29,49] rely on a long preprocessing step to compress the terrain data or generate the simplified tessellation, thereby effectively ruling out performing any kind of runtime alterations of the terrain.

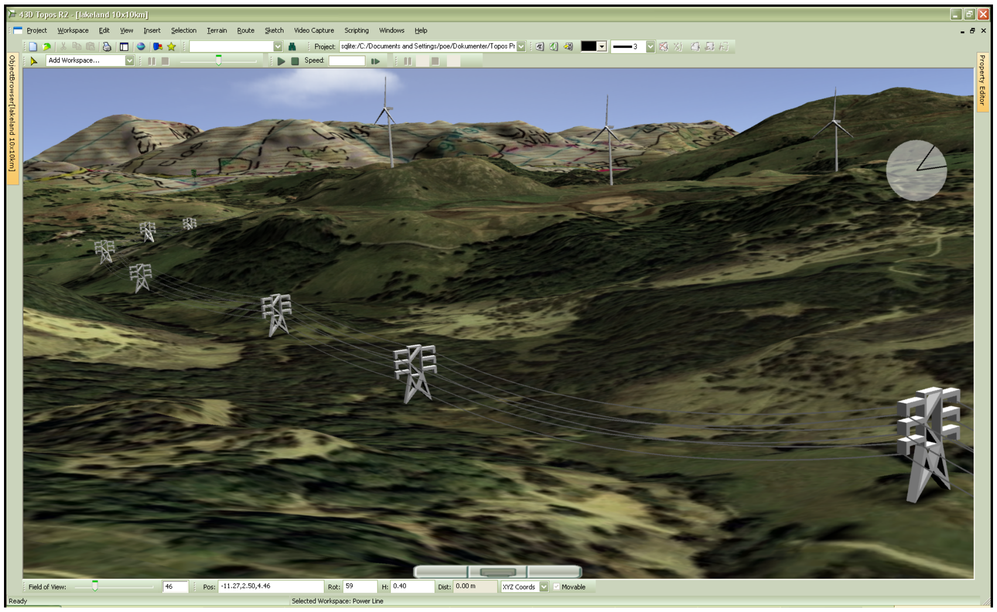

43D Topos R2 - [Lakeland 10x10km]
Project Workspace Edit View Insert Selection Terrain Route Sketch Video Capture Scripting Windows Help
Add Workspace...
Speed:
Property Editor
Field of View:
Pos:
Rot:
Dist:
XYZ Coords
Movable
Ready
Selected Workspace: Power Line

Figure 4.8: Screenshot of the Topos$^{\text{TM}}$ application using an implementation of our terrain rendering algorithm. Reprinted with permission from 43D.

to look at a less restrictive representation of the geometry. While this may not be a reasonable approach in the context of terrain rendering due the benefits of this simple representation in terms of algorithmic simplicity and efficiency, other applications may benefit significantly from a more advanced surface editing paradigm. In the next part of this thesis, we will focus explicitly on this particular issue, but rather than staying within the limited context of terrain rendering, we will look at the problem within a more general setting.

Part III

Geometric Texture Mapping

Chapter 5

Introduction

Motivated by the limited terrain editing operations made possible by the underlying height map representation, we will in the following chapters present a number of more flexible and more general techniques for adding and warping arbitrary *geometric textures* onto surfaces with arbitrary topology. For years, the standard approaches to increase geometric complexity have primarily been 2D texture [9], bump [8], and displacement mapping [16]. These techniques, while capturing a wide range of geometric phenomena, are limited in the types of detail they can represent. Kajiya and Kay [38] realized this early on and introduced volumetric textures to represent more topologically complex structures. Recently, the focus has shifted towards more sophisticated volumetric and geometric texturing approaches in an effort to capture a wider range of complex geometric phenomena [7, 13, 67, 83]. Unlike previous approaches, we can produce topologically connected surfaces with smooth blending and low distortion. Specifically, we present a novel 3D fine-scale explicit and implicit geometry mapping technique based on level sets, interpolation and radial basis functions. While we will assume a level set representation of the base surface, we will present methods for mapping both explicitly and implicitly defined geometric textures. To facilitate our mapping, we parameterize the embedding space of the base level set surface using a novel particle based localized parameterization. We can then warp explicit texture meshes onto this surface at nearly interactive speeds or blend level set representations of the texture to produce high-quality surfaces with smooth transitions.

Our contribution leverages the recent introduction of DT-Grid data structures and algorithms [55] and the large body of level set research to bridge the gap between existing volumetric and explicit geometric mapping techniques. Our general approach uses an implicit level set representation of both the base surface and the texture geometry [61]. This representation allows for robustness to topology changes during the mapping, flexibility when defining the blend of the base and texture geometry, and is amenable to high quality offset surface generation. Additionally, level sets offer a large body of advanced numerical techniques for easily computing surface properties and performing arbitrary deformations. In fact, as has been shown in previous work [53], direct control of blended surface properties is easily achievable with level sets. This high degree of robustness and flexibility, however, comes at the price of increased computational complexity when compared to purely explicit approaches. To address this issue, we have developed a fast *semi-implicit* technique that can conveniently be used for near real-time previewing. It combines an implicit level set representation of the base surface and an explicit polygonal representation of the textures.

We proceed by introducing the most important properties of level sets and implicit surfaces relevant to our work before going into further detail regarding our geometrical texture mapping methods.

Chapter 6

Level Sets and Implicit Surfaces

6.1 Level Set Theory

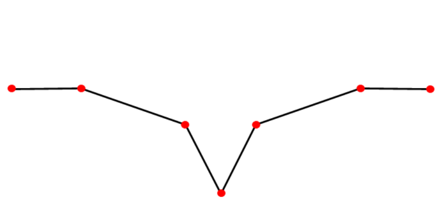
When talking about geometry in computer graphics, people usually think about lines, triangles/polygons, nurbs, subdivision surfaces etc., all of which fall into the category of *explicit surfaces*. Most real-time applications (games etc.) use only points, lines and triangles, as these are the only primitives supported by current graphics hardware.

Although this is an efficient geometry representation for many applications, it does fall short in some situations. One of the most significant shortcomings of the explicit surface representations is related to deforming surfaces. Consider for example a surface as depicted in Figure 6.1(a), represented using a number of line segments. Assuming we were to dilate the surface, simply moving each vertex a given distance along the vertex normal would result in a situation where the surface ends up intersecting itself as depicted in Figure 6.1(b). Obviously the surface is no longer physically realizable, as there is no longer a clear distinction between what is inside the surface and what is outside it. A physically plausible simulation of moving fronts would need to detect and resolve such situations. Furthermore, if the deformation of the surface causes it to change topology, by for instance breaking up into several surfaces or if several surfaces were to merge into one surface, this would also require special handling. Although these situations can, in most cases, be tracked and handled using an explicit surface representation, it is a complex and error prone task.

In contrast, an *implicit* surface representation defines the surface as the isocontour of a Lipschitz continuous function [85] $\phi(\mathbf{x})$. Representing e.g. the zero isocontour, a surface in $\mathbb{R}^n$ is defined as

$$\Omega^0 = \{\mathbf{x} \in \mathbb{R}^n | \phi(\mathbf{x}) = 0\},$$

that is, the set of points $\mathbf{x} \in \mathbb{R}^n$ satisfying the equation $\phi(\mathbf{x}) = 0$. We denote $\phi(\mathbf{x})$ *the embedding function.* It is common to use the zero-isocontour, as this simplifies many aspects of the implicit surface representation. This can be done without loss of generality, as for a given function $\phi(\mathbf{x})$ and an isocontour value $\alpha \in \mathbb{R}$, we can always define a new function $\hat{\phi}(\mathbf{x}) = \phi(\mathbf{x}) - \alpha$, for which the zero-isocontour is identical to the α-isocontour of $\phi(\mathbf{x})$. With this convention,


(a) Interface in 2 dimensions represented using connected lines

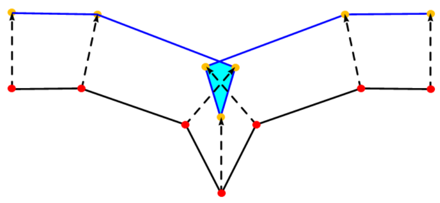

(b) Moving the vertices in the direction of the vertex normal causes the interface to self-intersect

Figure 6.1: Deforming an explicitly represented interface may cause the interface to self-intersect, leaving a region that is at the same time both inside and outside the object.

we can define the *interior* of the geometry as $\Omega^- = \{\mathbf{x} \in \mathbb{R}^n | \phi(\mathbf{x}) < 0\}$ and the *exterior* as $\Omega^+ = \{\mathbf{x} \in \mathbb{R}^n | \phi(\mathbf{x}) > 0\}$. Using an implicit representation of the surface may at first seem inefficient, as we are using a function defined on all of $\mathbb{R}^n$ to represent a surface of dimension $n-1$. However, we will see that this representation has a number of important properties. Using the classification of interior and exterior domain defined above, determining if a given point is inside or outside the volume becomes a check on the sign of the embedding function evaluated at that point. Also, by using a single valued function to define the surface, we are guaranteed a physically realizable closed surface, which means that self intersections are impossible by construction. This is of course due to the fact that a single-valued embedding function by nature cannot at the same time have both positive and negative sign. Furthermore, when using an implicit representation changes in the topology, for example when two water drops collide and merge into one, are handled automatically.

The gradient of an implicit surface is defined as

$$\nabla\phi = \left(\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right) \tag{6.1}$$

The gradient is a vector perpendicular to the isocontours of ϕ, pointing in the direction of increasing function values. Therefore, if $\mathbf{x}$ is a point on the zero isocontour of ϕ, then the gradient points in the same direction as the outward unit surface normal, N. Thus, knowing the gradient, we can define the unit surface normal as $N = (n_x, n_y, n_z) = \frac{\nabla\phi}{|\nabla\phi|}$. The *mean curvature* of the interface is defined in 3D as half the divergence of the normal [60]:

$$\kappa = \frac{1}{2}\nabla \cdot N = \frac{1}{2}\left(\frac{\partial n_x}{\partial x} + \frac{\partial n_y}{\partial y} + \frac{\partial n_z}{\partial z}\right). \tag{6.2}$$

Other operations, such as finding the surface area or the volume of the interior can be found by integrating $\phi(\mathbf{x})w(\mathbf{x})$ over $\mathbb{R}^n$, where $w(\mathbf{x})$ depends on the property we are looking for. For example, if we are interested in finding the volume of the interior, we set

$$w(\mathbf{x}) = \begin{cases} 1 & \text{if } \phi(\mathbf{x}) < 0, \\ 0 & \text{otherwise.} \end{cases}$$

Discrete Representation of Implicit Surfaces

In practice, the implicit surface is rarely represented by an analytic expression such as $\phi(\mathbf{x}) = |\mathbf{x}| - 1$, as it can be extremely hard, if not impossible, to perform any type of deformation of the surface and subsequently find a new analytic expression for the function defining the deformed surface. Instead, the typical approach is to sample the function on a rectilinear grid, and then use interpolation of the values at the grid points to approximate the function value between grid points. In the following, ϕ_{ijk} denotes a lookup in the grid at position i, j, k, corresponding to evaluating $\phi(i\Delta x, j\Delta y, k\Delta z)$. For the sake of clarity, we will suppress indices in upcoming equations if they are constant or not important in that equation. Δx, Δy and Δz denotes the distance between sample points in the x, y and z direction respectively. In the remainder of this thesis, we will follow the assumption that $\Delta x = \Delta y = \Delta z = 1.0$.

The consequence of using a discrete representation of the function is that all of the properties defined above can only be approximated through numerical methods. Depending on the application, the derivatives in equation (6.1) can be approximated using a first order accurate forward difference, ϕ_x^+:

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{i+1} - \phi_i}{\Delta x}, \tag{6.3}$$

a first order accurate backward difference, ϕ_x^-:

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_i - \phi_{i-1}}{\Delta x}, \tag{6.4}$$

or a second order accurate central difference, ϕ_x^0:

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x}. \tag{6.5}$$

Similar equations are used for calculating the derivatives in the y and z directions. The *finite difference* approximation to the derivatives of ϕ in equation (6.3)-(6.5) are derived from a Taylor expansion. Equation (6.3) for example is derived from the Taylor series approximation to ϕ at the point $x + \Delta x$ about x:

$$\begin{aligned} \phi(x + \Delta x, y, z) &= \phi(x, y, z) + \frac{\partial \phi}{\partial x}\Delta x + O(\Delta x^2) && \Downarrow \\ \frac{\partial \phi}{\partial x} &= \frac{\phi(x + \Delta x, y, z) - \phi(x, y, z)}{\Delta x} + O(\Delta x) \end{aligned}$$

Omitting the $O(\Delta x)$ term leads to the approximation in equation (6.3). This approximation is first order accurate, referring to the error introduced by omitting the $O(\Delta x)$ term. In general, an n'th order accurate approximation is an approximation with a $O(\Delta x^n)$ error.

The combination of discretising the implicit function and using numerical approximations introduces a potential problem: If ϕ is not *well behaved*, then the error in the numerical approximations might be rather large. In this context, *well behaved* means that we would like ϕ to be as smooth as possible. Fortunately, as we are only interested in the zero-isocontour of the function, we can freely choose the function defining the surface in any way we please, as long as it preserves the zero isocontour. The *signed distance function* turns out to be a very good choice for representing implicit surfaces. A *distance function* $d(\mathbf{x})$ is defined as $d(\mathbf{x}) = \min(|\mathbf{x} - \mathbf{x_I}|)$ for all $\mathbf{x}_I \in \Omega^0$, that is, a function giving the shortest distance to the surface. A *signed distance function* is a function $\phi(\mathbf{x})$ defined as

$$\phi(\mathbf{x}) = \begin{cases} -d(\mathbf{x}) & \text{if } \mathbf{x} \text{ is in the interior of the geometry,} \\ d(\mathbf{x}) & \text{otherwise.} \end{cases}$$

Another desirable property of a signed distance function is that the length of the gradient is identically one, i.e. $|\nabla\phi| = 1$, except at corners or kinks (jumps/discontinuities in the derivatives) where the gradient is not defined. In fact, this is a defining property of the signed distance function, that is, ϕ is a signed distance function if and only if $|\nabla\phi| = 1$.

Level Set Methods

A deformable surface, $\mathcal{S}(t)$, is defined as a time dependent implicit surface:

$$\mathcal{S}(t) = \{\mathbf{x}(t) \in \mathbb{R}^n | \phi(\mathbf{x}(t), t) = 0\}. \tag{6.6}$$

Differentiating equation (6.6) with respect to t and applying the chain rule, gives the *fundamental level set equation*

$$\phi_t + \nabla\phi \cdot \frac{d\mathbf{x}}{dt} = 0 \tag{6.7}$$

Here, $d\mathbf{x}/dt$ denotes the vector field used to deform the surface. Several numerical schemes for solving this partial differential equation exist, depending on the application and the desired numerical precision. We will go into detail with these schemes in section 6.2. A more detailed discussion of this subject can be found in [60]. As N and $\nabla\phi$ points in the same direction, $T \cdot \nabla\phi = 0$ for any tangent vector T. This implies that for a given velocity field $\mathbf{V}$, the tangential component vanishes. In other words, when specifying a velocity field, we need only worry about the velocity in the normal direction, $\mathbf{V} = V_n N$. The velocity field V_n is often referred to as the *speed function.* If we substitute this into the level set equation, we get:

$$\phi_t + V_n N \cdot \nabla\phi = 0.$$

Furthermore, as $N \cdot \nabla\phi = \frac{\nabla\phi}{|\nabla\phi|}\nabla\phi = \frac{|\nabla\phi|^2}{|\nabla\phi|} = |\nabla\phi|$, we can rewrite equation (6.7) as:

$$\phi_t + V_n|\nabla\phi| = 0. \tag{6.8}$$

An example of a surface deformation providing serious challenges with the explicit representation is morphing. Morphing one surface into another using level sets on the other hand is simple. Given the signed distance representation of two implicit surfaces, ϕ_1 and ϕ_2, morphing from ϕ_1 to ϕ_2 is done as follows: First, we initialize our working volume ϕ to ϕ_1. Then we iteratively solve equation (6.8) with the speed function $V_n = \phi - \phi_2$ to steady state [10, 86]. The only requirement is that all connected components of ϕ_2 overlap with the initial volume, ϕ_1.

Re-Initialization

As mentioned above, using a signed distance function as the embedding function is beneficial. However, even if we start out with a signed distance function, there is no guarantee that it remains a signed distance function following a deformation through evaluation of equation (6.7) or (6.8). In order to keep harvesting the benefits of a signed distance function, we need to reset the embedding function to a signed distance function. Again, as we are only interested in the zero isocontour of the function, performing such a *reinitialization* step is perfectly legal as long as the zero isocontour remains unchanged.

One way to reinitialize the embedding function to a signed distance function is to solve the *reinitialization equation*

$$\phi_t + S(\phi)(|\nabla\phi| - 1) = 0 \tag{6.9}$$

to steady state. When reaching steady state, $\phi_t = 0$, and the equation is reduced to $|\nabla\phi| = 1$ as desired. In equation (6.9), $S(\phi)$ is a sign function, originally taken as 1 in Ω^+, -1 in Ω^- and 0 on the interface. The sign function enables us to work with two different equations, depending on the sign of ϕ at a given location. For points in Ω^+, we solve the equation $\phi_t + |\nabla\phi| = 1$, whereas in Ω^-, we solve the equation $\phi_t - |\nabla\phi| = -1$. Both equations, when solved to steady state yields the result $|\nabla\phi| = 1$, however changing the sign for points in Ω^- ensures that the calculations performed for each grid point uses only information from grid points in the direction of the interface rather than using information from grid points further away from the interface. As a consequence, the zero crossing remains fixed while the distance is propagated away from the zero crossing in each direction, slowly resetting ϕ to a signed distance function. In practice, it turns out that better results are obtained when using a numerically smeared out sign function. Peng et al. [65] suggests that

$$S(\phi) = \frac{\phi}{\sqrt{\phi^2 + |\nabla\phi|^2(\Delta x)2}} \tag{6.10}$$

is a good choice. Equation (6.9) is a Hamilton-Jacobi type partial differential equation, which can be solved using the same numerical techniques used for solving equation (6.7), including higher order spatial and temporal schemes. In other words, if we are willing to sacrifice performance, we can achieve a very high degree of accuracy in our reinitialization step.

The problem with reinitialization through solving equation (6.9) is the performance penalty. Solving the PDE can be quite time consuming, and as it has to be done after every few iterations of the original level set simulation, it quickly becomes the bottle neck of a level set simulation. Alternative techniques do exist, most notably the fast marching method [73] and the more recent fast sweeping method [90]. Both of these methods usually perform significantly faster than the PDE based approach[1], however they are both, in their initially form, only first order accurate. Both methods can be extended to a higher order of accuracy, although to the best of our knowledge, the fast marching method has only been extended to second order accuracy [69, 89]. In our work, we utilize a combination of the first order accurate fast sweeping method whenever appropriate, and a fifth order accurate PDE approach elsewhere.

6.2 Implementing and Representing Level Sets

6.2.1 Solving the Level Set Equations on a Regular Grid

In the previous section, we introduced the implicit surface representation and the level set methods as a powerful representation for dynamic geometry. Sampling the embedding function on a regular grid provided a means for working with complex models without having to derive an analytic expression for the embedding function. We will now go into more details on how to solve the level set equations in this discrete context.

Higher Order Finite Difference Schemes

In the previous section, we introduced the first order accurate forward- and backward-difference and the second order accurate central difference approximation to the derivatives of ϕ (equation (6.3)-(6.5)). While these methods are often sufficiently accurate, more accurate methods are available if required. The *Hamilton-Jacobi Essentially Non Oscillatory* (HJ ENO) finite difference scheme uses Newton polynomial interpolation [39] to approximate ϕ. Using a divided difference table, the smoothest possible polynomial interpolating ϕ is found. The constructed approximating polynomial is then differentiated to get the final approximation to the derivatives of ϕ. Although HJ ENO can be constructed at any order of accuracy, the most commonly used is a third order accurate scheme. In the third order accurate scheme, using the divided difference table to construct the interpolating polynomial, one of exactly three possible polynomials is chosen depending on the behavior of ϕ in the vicinity of the point (i, j, k). Choosing the smoothest possible polynomial, reduces the risk of interpolating across a discontinuity in the derivative thereby avoiding the error that would otherwise have been introduced. Calculating the approximation to $(\phi_x^-)_i$ using this scheme uses a subset of the samples $\{\phi_{i-3}, \phi_{i-2}, \phi_{i-1}, \phi_i, \phi_{i+1}, \phi_{i+2}\}$, depending on the which polynomial is used. Similar, the calculation of $(\phi_x^+)_i$

[1]With the fast sweeping method being the faster of the two. The fast sweeping method has $O(N)$ complexity as opposed to the $O(N \log(N))$ complexity of the fast marching method, where N is the number of grid points.

uses a subset of the samples $\{\phi_{i-2}, \phi_{i-1}, \phi_i, \phi_{i+1}, \phi_{i+2}, \phi_{i+3}\}$. This set of samples that a given finite difference scheme uses is usually referred to as its *stencil*.

By employing a convex combination of the three different interpolating polynomials used for the HJ ENO scheme, the *Hamilton-Jacobi Weighted Essentially Non Oscillatory* (HJ WENO) finite difference scheme is able to approximate the derivatives of ϕ with a fifth order of accuracy in smooth regions using the same stencil as the HJ ENO scheme. Outside smooth regions, the weights are chosen to favor the HJ ENO approximations not interpolating across discontinuities thereby obtaining third order accuracy. For more details on both the HJ ENO and the HJ WENO schemes, we refer to [60].

Temporal Discretization

In order to solve the level set equation (equation (6.7) and equation (6.8)), we need to discretize the equations on our regular three dimensional spatial grid as well as on a one dimensional temporal grid. In practice, the spatial and temporal discretizations are treated differently. The forward-, backward- and central difference methods as well as the HJ ENO and HJ WENO schemes are used for the spatial discretization, but not generally for the temporal discretization. One of the most widely used and most simple methods for temporal discretization of the level set equations is the first order accurate *forward Euler* method. The forward Euler time discretization method is a one sided forward difference in time. Applied to equation (6.8) this yields the following expression:

$$\frac{\phi^{m+1} - \phi^m}{\Delta t} + V_n^m |\nabla \phi^m| = 0,$$

where ϕ^m denotes the values of ϕ at time m, and V_n^m denotes the values of the speed function V_n at time m. To obtain higher orders of accuracy, *Total Variation Diminishing Runge-Kutta* (TVD RK) [77] methods can be applied. The first order accurate TVD RK method is just the forward Euler method. Higher order accurate TVD RK methods are obtained as a convex combination of the initial data and the results from several successive Euler time steps. A second order accurate TVD RK scheme for example is obtained by first applying two successive forward Euler time steps to obtain ϕ^{m+2}. Then, the final solution is obtained by averaging the original values, ϕ^m, and ϕ^{m+2}:

$$\phi^{m+1} = \frac{1}{2}(\phi^m + \phi^{m+2}).$$

Higher order accurate TVD RK schemes proceeds in a similar, yet obviously more complicated, fashion. See [77] or [60] for further details.

Numerical Solutions to Hyperbolic Level Set Equations

Several different numerical schemes are applied in order to solve the level set equations (equation (6.7) and (6.8)), depending on the chosen speed function or velocity field. When the speed function or velocity field does not depend on derivatives of ϕ of higher than first order, these equations are hyperbolic PDEs, known as *Hamilton-Jacobi Equations*. These PDEs have the property that information is propagated in certain directions known as *characteristics*. While the two equations are mathematically identical, their use differs significantly numerically, and thus needs to be treated differently. Equation (6.7) is often used in conjunction with an externally generated velocity field $\mathbf{V}(\mathbf{x}, t)$, that is, we rewrite equation (6.7) as

$$\phi_t + \nabla\phi \cdot \mathbf{V}(\mathbf{x}, t) = 0. \tag{6.11}$$

To solve this equation, we can perform a single forward Euler time step, and use one of the available FD schemes to approximate the derivatives of ϕ. If we expand equation (6.11) around a single spatial point, x_i, and apply the first order forward Euler time step, we get the following equation:

$$\phi_i^{m+1} = \phi_i^m + \Delta t(\mathbf{V}(x_i, m)_x(\phi_x)_i^m + \mathbf{V}(x_i, m)_y(\phi_y)_i^m + \mathbf{V}(x_i, m)_z(\phi_z)_i^m). \tag{6.12}$$

We will only focus on evaluating the $\mathbf{V}(x_i, m)_x(\phi_x)_i^m$ term. The technique we use to approximate this term can be applied independently to the remaining two terms in a dimension by dimension manner. The fact that the information is propagated through each point in space in exactly one direction (the characteristic) implies that we should look in the other direction, known as the *upwind* direction, to determine the future value of ϕ at that point. Consequently, we should use upwind values of ϕ to determine the derivatives of ϕ, rather than downwind values. This means that if $\mathbf{V}(x_i, m)_x > 0$, we should use ϕ_x^- to approximate ϕ_x, and ϕ_x^+ if $\mathbf{V}(x_i, m)_x < 0$.

To ensure *convergence*, that is, to ensure that the correct solution is obtained as $\Delta x \to 0$ and $\Delta t \to 0$, is according to the Lax-Richtmyer theorem [60] equivalent to ensuring *consistency* and *stability*. Both the upwind scheme combined with a forward Euler time step and the schemes outlined below are consistent approximations to the partial differential equations, as the approximation error converges to 0 as $\Delta x \to 0$ and $\Delta t \to 0$. The stability condition ensures that small errors are not amplified over time. This is enforced by restricting the size of the time step taken. In the case of the scheme just outlined, the time step restriction, known as the Courant-Friedrichs-Lewy stability condition (CFL condition), is $\Delta t < \frac{\Delta x}{\max\{|\mathbf{V}|\}}$ [60].

To solve equation (6.8), a somewhat different approach is required. As the equation is non-linear in terms of ϕ, we cannot apply the straightforward upwind scheme used to solve equation (6.11). Instead, we can apply the widely used Gudonov scheme [70] to approximate the squared derivatives:

$$\phi_x^2 = \begin{cases} \max(\phi_x^-, -\phi_x^+, 0)^2 & \text{if } V_n > 0 \\ \max(-\phi_x^-, \phi_x^+, 0)^2 & \text{if } V_n < 0 \\ 0 & \text{otherwise.} \end{cases} \tag{6.13}$$

As with the upwind scheme, the Gudonov scheme is applied independently to the derivatives of each dimension. The estimates calculated using this scheme can then be used to estimate the norm of the gradient, $|\nabla\phi| = \sqrt{\phi_x^2 + \phi_y^2 + \phi_z^2}$.

As with the upwind scheme, we can combine the Gudonov scheme with a first order accurate forward Euler time step, or a higher order TVD RK scheme if desired. Similarly, the derivatives in equation (6.13) are calculated using either first order finite difference schemes or using higher order accurate HJ ENO or HJ WENO schemes. In this case, the CFL condition is a bit more complicated, but a conservative time step restriction is $\Delta t < \frac{\Delta x}{\max\{|V_n|\}d}$, where d is the dimension [60].

Numerical Solutions to Parabolic Level Set Equations

When the speed function depends on derivatives of higher than first order, typically the mean curvature, the level set equations become parabolic equations. These equations have different properties than the hyperbolic equations described above. Unlike the hyperbolic PDEs, a parabolic PDE has no real characteristics. Instead, information flows into each point from (potentially) every direction. Moreover, the propagation speed of this information is (in principle) infinite. Consequently, we need to use the central finite difference scheme, *e.g.* equation (6.5), to estimate the spatial derivatives of ϕ.

As a consequence of the (potentially) infinite propagation speed, a more restrictive time step is required. If for instance $V_n = b\kappa$, where κ is the mean curvature and b is a scaling parameter, then the required time step restriction for a 3 dimensional solution becomes $\Delta t < (\frac{2b}{\Delta x^2} + \frac{2b}{\Delta y^2} + \frac{2b}{\Delta z^2})^{-1} = \frac{\Delta x^2}{6b}$.

6.2.2 Level Set Representations and Narrow Band Methods

While sampling the embedding function on a regular grid may enable us to actually work with implicit surfaces and solve the level set equation, it unfortunately leads to a representation that is prohibitively expensive with respect to memory consumption as well as execution time. Representing a surface in 3-dimensional space sampled on a grid with resolution 256^3 using 4-byte floats to hold the sample values requires a total of 64Mb of memory. Furthermore, deforming a surface using equation (6.7) or (6.8) involves solving the equation on each grid point. Thus, the complexity of the level set simulation is proportional to the size of the embedded volume rather than the size of the actual surface. In the following, we will briefly introduce a number of methods addressing these two limitations of the level set method.

As we have already stated several times, we are only interested in the zero iso-contour of the embedding function. Furthermore, as we will constantly be resetting the embedding function to a signed distance function, solving the level set equation on the entire grid is an unnecessary waste of processing power. This observation has lead to the development of local level set methods, including that of Peng et al. [65] commonly referred to as the *narrow band* method. The narrow band method works by first identifying the grid points for which the absolute value of ϕ is within a given range. These points are classified, see Figure 6.2, as belonging to the *beta tube* if $|\phi(\mathbf{x})| < \beta$, the *gamma tube* if $|\phi(\mathbf{x})| < \gamma$ and the delta tube if $|\phi(\mathbf{x}+\mathbf{y})| < \gamma$ for some $y \leq \Delta x$. The delta tube contains all grid points contained in the gamma tube plus all grid points

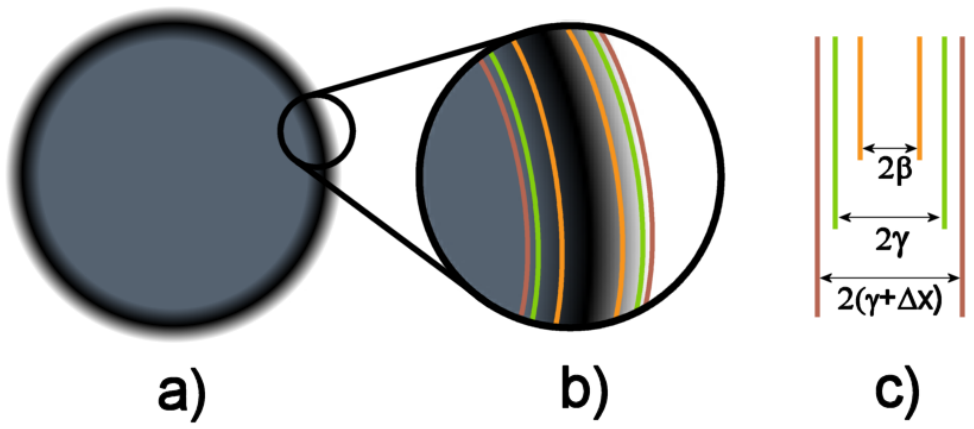


Figure 6.2: Narrow band representation. (a) A signed distance function with values outside the narrow band clamped to $\pm\gamma$. (b) Close up of the narrow band, with the beta tube highlighted in orange, the gamma tube in green and the delta tube in brown. (c) The complete distance range covered by each of the tubes.

not in the gamma tube, but lying right next to one or more points in the gamma tube. The value of ϕ in grid cells outside the delta tube are clamped to γ or $-\gamma$, depending on the sign of ϕ. With this classification of the grid points, we proceed to solve a slightly modified level set equation

$$\phi_t + c(\phi)\nabla\phi \cdot \frac{d\mathbf{x}}{dt} = 0 \tag{6.14}$$

or

$$\phi_t + c(\phi)V_n|\nabla\phi| = 0, \tag{6.15}$$

only on the grid points belonging to the gamma tube. Here, $c(\phi)$ is a cut off function introduced to prevent numerical oscillation at the boundary of the tube:

$$c(\phi) = \begin{cases} 1 & \text{if } |\phi| \leq \beta \\ (|\phi| - \gamma)^2(2|\phi| + \gamma - 3\beta)/(\gamma - \beta)^3 & \text{if } \beta < |\phi| \leq \gamma \\ 0 & \text{if } |\phi| > \gamma \end{cases} \tag{6.16}$$

By limiting the time step taken such that the front moves less than one grid point, we are guaranteed that the gamma tube of ϕ_{i+1} , that is ϕ after time step $i+1$, is contained within the delta tube of ϕ_i. Hence, we need only perform our reinitialization of ϕ on grid points within the delta tube. Finally, we need to reclassify the grid points into the beta, gamma and delta tubes before we are ready to perform another step of the level set simulation. The numerical values of β and γ depends on the numerical scheme used for solving the (modified) level set equation. Typical values for a second or third order correct scheme is $\beta = 2\Delta x, \gamma = 4\Delta x$, and for a fifth order accurate scheme, $\beta = 3\Delta x, \gamma = 6\Delta x$.

The narrow band method reduces the computational complexity to $O(n)$, where n is the number of grid points in the delta tube, but it does not reduce the memory requirements. In fact, it actually increases the amount of memory required, as we now also need to maintain a list of grid points contained within the narrow band (that is, the delta tube), as well as a classification of each of these grid points.

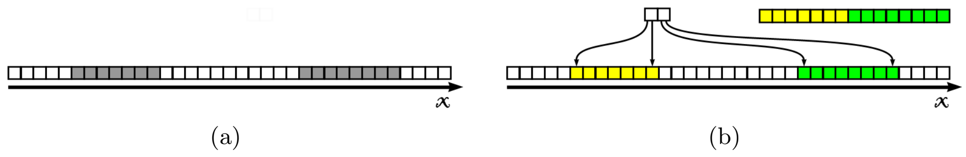
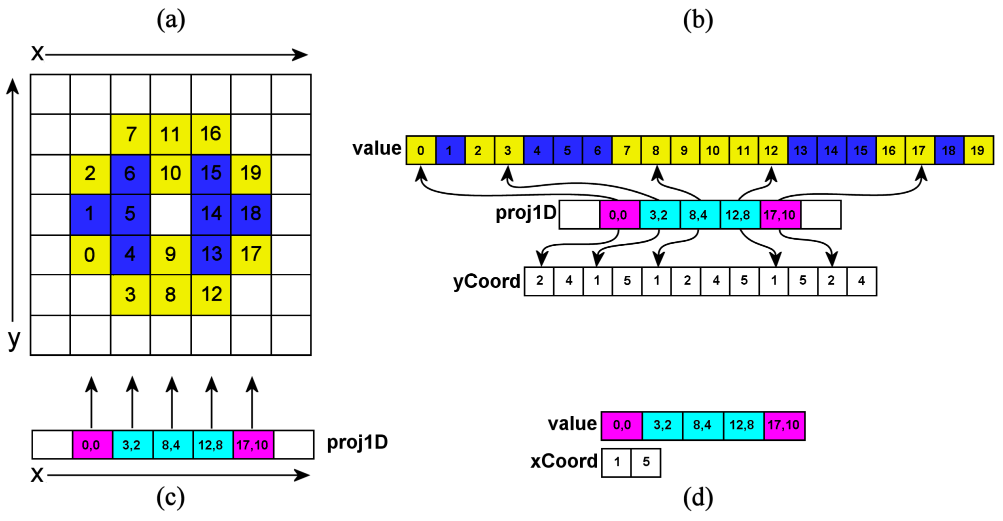


Figure 6.3: 1-Dimensional level set. (a) The grid cells colored gray are the grid cells within the narrow band, and thus the only cells we wish to represent in memory. (b) Maintaining a list of the minimum and maximum x-coordinate of each connected component allows us to compress the values into an array containing only the narrow band values. To be able to distinguish the two connected components in the collapsed array, the first connected component is colored yellow, while the second is green.

Reducing the memory footprint of a level set system has been the focus of attention for several authors in a number of recent papers [36, 37, 48, 56]. Common to these systems is, that they provide data structures enabling a sparse representation of the implicit surface, storing only samples in the regions of interest, which is typically the delta tube required for the narrow band solution presented above. The result is a level set solver with memory and execution time complexity depending on the size of the interface rather than the size of the grid. An in depth discussion of all these systems is out of the scope of this thesis. The interested reader is referred to the respective papers for details. Instead we will give a short description of only one of those data structures, the DT-Grid [56], which is the one we have chosen to use. Our choice is based solely on availability, as all of the above mentioned data structures appear to be equally well suited for our purpose, We have chosen to use the DT-Grid simply because we have an existing implementation readily available.

The DT-Grid is a hierarchical data structure designed to hold a sparse representation of an implicit surface, sampled regularly on a n-dimensional grid. The general idea with the DT-Grid is to store the values in the narrow band along with the information required to retrieve the stored value given a position in n-dimensional space. This is achieved using the following approach: Consider a level set in 1-dimension. Disregarding the values outside the delta tube, the signed distance function can be described as a number of connected components, see Figure 6.3(a). If we store the smallest and largest x-coordinate in a separate list, then we can create an array containing the elements of each connected components stored sequentially: $[\phi_{1,1}, \ldots, \phi_{1,c_1}, \phi_{2,1}, \ldots, \phi_{2,c_2}, \ldots, \phi_{n,1}, \ldots, \phi_{n,c_n}]$, where $\phi_{i,j}$ is the value of the j'th element of the i'th connected component, n is the number of connected components, and c_i is the number of values in the i'th connected component. To extend this into a 2-dimension level set, we first create a boolean projection of the level set onto the x axis, that is, we find the set $\mathcal{X} = \{x | \exists y : \text{grid point } (x, y) \text{ is in the narrow band}\}$, see Figure 6.4. Each



🟨 Min and max y-coordinates of connected components in 2D (**Yellow**).
🟦 Internal y-coordinates of connected components in 2D (**Blue**).
🟪 Min and max x-coordinates of connected components in 1D (**Magenta**).
🟦 Internal x-coordinates of connected components in 1D (**Cyan**).

Figure 6.4: DT-Grid data structure. (a) A 2D grid with the grid points in the narrow band highlighted in yellow and blue. (b) 2D DT-Grid representation of (a). (c) A 1D grid, corresponding to the projection of the 2D grid onto the x axis. (d) 1D DT-Grid corresponding to (c). This 1D DT-Grid forms part of the 2D DT-Grid as depicted in (b). Reprinted with permission from [56].

$x \in \mathcal{X}$ corresponds to a column containing at least one value inside the narrow band, which in turn can be viewed as a 1-dimensional level set and represented using the method described above. Similarly, the set $\mathcal{X}$ can be represented using this compact method, only rather than storing ϕ values, we store pointers to the 1-dimensional DT-Grids representing the corresponding column, see Figure 6.4. Extending this further into 3 or more dimensions is straightforward. For further details, please consult the original article.

6.2.3 Re-sampling

When working with volumes and level set methods, most operations only involves reading and modifying the sampled values. However, sometimes this is not enough, and we need to estimate the function values in between the grid points. Typically, this is done using linear interpolation of the function values at the surrounding grid points using the following formula:

$$\begin{aligned}\tilde{\phi}(i,j) = {} & (\phi(i_i,j_i)(1-i_f) + \phi(i_i+1,j_i)i_f)(1-j_f) \\ & + (\phi(i_i,j_i+1)(1-i_f) + \phi(i_i+1,j_i+1)i_f)j_f,\end{aligned} \tag{6.17}$$

where $\tilde{\phi}(i,j)$ is the interpolated value at position (i,j), i_i is the integer part of i, i_f is the fractional part of i, and $\phi(i_i,j_i)$ is the sampled function value at integer location (i_i,j_i). For reasons of simplicity, the previous example as well as the rest of this discussion will focus on the two dimensional case. Extending this to three or more dimensions is straightforward.

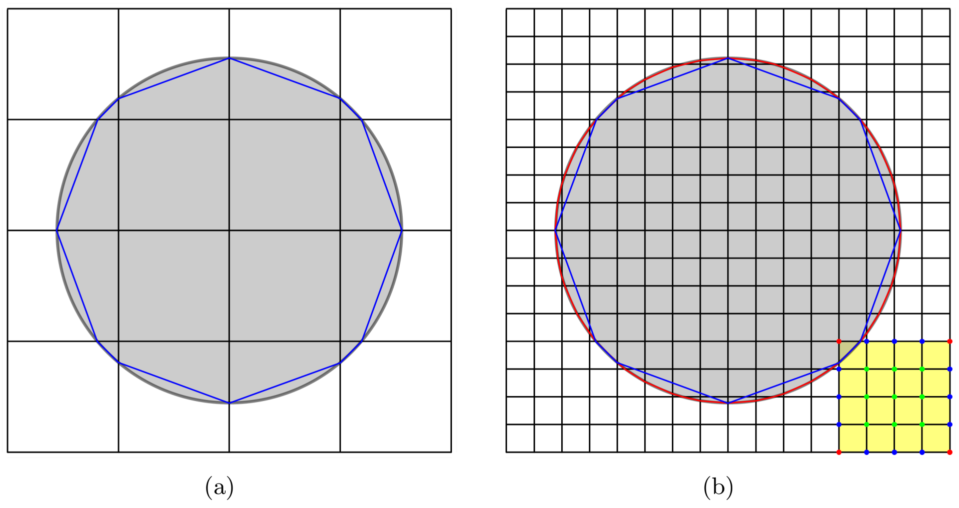


Figure 6.5: (a) Implicit surface sampled on a coarse grid. The blue curve is the linear approximation to the interface generated from the samples. (b) Re-sampling the volume to four times the resolution using linear interpolation. The blue curve is again the linear approximation to the interface generated from the new samples, whereas the red curve is the linear approximation to the interface generated from a re-sampling of the embedding function on the same grid. Note, that the blue curve in (a) and (b) are identical. In the lower right corner of (b), a single voxel from the coarse grid is highlighted in yellow. The red dots are the samples for that voxel taken from the coarse grid. The blue and green dots are the new samples added to the higher resolution grid.

Using linear interpolation is for the most cases sufficient. This is for instance the case when using the marching cubes algorithm to extract a triangle mesh from the iso-surface as described in section 6.4. This seems obvious, as the triangle mesh already is a linear approximation of the iso-surface, hence introducing an extra linear term at the vertices will not produce visible artifacts. This is, however, not the case when re-sampling volumes to a different resolution. If we restrict ourselves to a linear interpolation, re-sampling a volume to a higher resolution will actually result in a lower quality approximation of the embedding function, see figure 6.5. The problem is, that the sample values no longer represent the best possible approximation to the embedding function given the current resolution. Furthermore, surface properties such as gradients, normals, curvature etc. depending on the partial derivatives will be calculated using the linearly interpolated values. Consider the volume in figure 6.5(a). Re-sampling the volume to four times the resolution results in 3 new samples being inserted along the axes between each adjacent pair of samples (blue dots in figure 6.5(b), lower right corner), and 3×3 samples inside the voxel (green dots in figure 6.5(b), lower right corner). Due to the linear interpolation, the three blue points on a horizontal line will have the same partial derivative with respect to the x coordinate, as the rate of change along that line is constant. Similarly, the three blue points on a vertical line will have the same partial derivative with respect to the y coordinate. Finally, the green points, that is, the points added inside a voxel, will all have the same partial derivative with respect to both the x and y coordinates. This is obviously not a desirable result, so a better, higher order, interpolation is required. To derive a higher order interpolation scheme, we observe that the linear interpolation scheme from equation (6.17) is in fact equivalent to a convolution with a separable triangle filter of width 1 [50]:

$$\tilde{\phi}(i,j) = \frac{\sum_{x=-f_{w_i}}^{f_{w_i}} \sum_{y=-f_{w_i}}^{f_{w_i}} w(x-i_f, y-j_f)\phi(i_i+x, j_i+y)}{\sum_{x=-f_{w_i}}^{f_{w_i}} \sum_{y=-f_{w_i}}^{f_{w_i}} w(x-i_f, y-j_f)}, \tag{6.18}$$

where f_{w_i} is the integer width of the filter (if f_w is the width of the filter, then $f_{w_i} = \lceil f_w \rceil$), in this case 1, and w is the filter function:

$$w(x,y) = \begin{cases} (f_w - |x|)(f_w - |y|) & \text{if } (x,y) \in [-f_w, f_w]^2, \\ 0 & \text{else.} \end{cases}$$

This means that we can achieve a higher order interpolation simply by changing the filter function and/or the width of the filter in equation (6.18). When high order interpolation is required, we use a cubic spline filter of width 2 (support width 4). The image in figure 6.9(a) is generated using high order interpolation to locate the intersection point between a ray and the implicit surface. Had we used linear interpolation instead, the result would have been virtually identical to figure 6.9(b). As always, the price we pay for the gained quality is performance. Using a filter of width 2, we need to lookup 16 samples (64 in 3 dimensions) to calculate the new ϕ value. In contrast, linear interpolation requires only 4 samples (8 in 3 dimensions) to calculate the desired value.

6.3 Modeling with Level Sets and Implicit Surfaces

While level set methods and dynamic implicit surfaces have been quite popular for a number of applications like segmentation and physical simulations of water, smoke flowing around obstacles or snow etc., there has been little previous work on implicit models as a modeling tool, one of the strong holds of the explicit surface representation. This is quite a shame, as the implicit representation offers many advantages over the explicit surface representations[2]. One such advantage is the ease with which constructive solid geometry (CSG) operations are performed on implicit surfaces. Performing a CSG operation on polygon meshes, subdivision surfaces or nurbs is a complex and error prone task[3]. On

[2]Note, that we do not in any way advocate that the implicit surface representation should replace the use of an explicit surface representation, instead we see it as a geometric representation that could be used in parallel with explicit representations, similar to the way many modelers already support polygon meshes, nurbs and subdivision surfaces to coexist in a single scene.

[3]Even leading commercial modeling systems such as Maya and 3D Studio Max have difficulties in performing these operations robustly.

the other hand, performing these operations on a pair of implicit surfaces is straightforward: Given two implicit surfaces $\phi_1(\mathbf{x})$ and $\phi_2(\mathbf{x})$, the *union* of the interior regions of ϕ_1 and ϕ_2 is given by $\phi(\mathbf{x}) = \min(\phi_1(\mathbf{x}), \phi_2(\mathbf{x}))$, the *intersection* of ϕ_1 and ϕ_2 is given by $\phi(\mathbf{x}) = \max(\phi_1(\mathbf{x}), \phi_2(\mathbf{x}))$, the complement of ϕ_1 is $\phi(\mathbf{x}) = -\phi_1(\mathbf{x})$, and the region resulting from subtracting ϕ_2 from ϕ_1 is given by $\phi(\mathbf{x}) = \max(\phi_1(\mathbf{x}), -\phi_2(\mathbf{x}))$. These standard CSG operators give sharp edges at the intersection curve of the two input volumes. To produce a softer and more pleasing intersection, Dekkers *et al.* [19] added a *blending function* to the basic CSG operators. With the blending function, the *generalized union* is given by $\phi(\mathbf{x}) = \min(\phi_1(\mathbf{x}), \phi_2(\mathbf{x})) - f_b(|\phi_1 - \phi_2|, n)$, where n is a user defined non negative value that determine the amount of blending ($n = 0$ corresponding to the non blended standard CSG operator). Similar expressions are given for the intersection and subtraction operators. While the choice of the function f_b is free, four desirable properties of the function are given in their paper. These four properties are that the function should be differentiable, it should present an intuitive control over the blending, it should have a limited domain of influence and it should satisfy the Lipschitz condition for $\lambda = 1$, that is, $-1 \leq f_b'(x, n) \leq 0$, for all $x \geq 0$, $n \geq 0$. One function given in the paper, satisfying all four conditions is

$$f_b(x,n) = \begin{cases} n(\frac{x}{n} - \frac{1}{4})^2 & \text{for } x < \frac{n}{4} \\ 0 & \text{for } x \geq \frac{n}{4} \end{cases}$$

This approach generally works quite well, but there are cases where it produces undesirable results. One such example is the union of two volumes that are close together but without actually overlapping each other. Even though the objects do not overlap, applying the generalized CSG union will produce a single, fully connected, surface as long as the two objects are within the blending distance of each other. The CSG operators, with or without blending, are, when used carefully, a very flexible and intuitive tool. This was demonstrated by Wang and Kaufman who developed a complete volume *sculpting* system by combining a set of carefully designed volumes with the standard CSG operators [82].

A more advanced level set surface editing framework was presented in 2002 by Museth et al. [53], introducing several localized surface editing operators defined by speed functions. These speed functions are defined by a combination of three distinct building blocks:

$$\mathcal{F}(\mathbf{x}, \mathbf{n}, \phi) = \mathcal{D}_q(d)\mathcal{C}(\gamma)\mathcal{G}(\gamma), \tag{6.19}$$

where $\mathcal{D}_q(d)$ is a distance based cut-off function depending on the distance d to the geometric structure q, $\mathcal{G}(\gamma)$ is a function depending on some geometric measure γ, derived from the level set surface, and $\mathcal{C}(\gamma)$ is a cut-off function controlling the contribution of $\mathcal{G}(\gamma)$ to the speed function.

Specifically, two of their operators are of great importance to our work. Most important is their CSG operations with automatic localized blending. Following a CSG operation, the authors find the voxels containing the zero crossing from both volumes. These are the voxels containing the *intersection curve* shared by the two volumes. Letting $\mathcal{D}_q(d)$ in equation (6.19) depend on the shortest distance to (a point wise approximation to) this curve, and by letting $\mathcal{G}(\gamma)$ be proportional to the mean curvature of the surface, the result is a localized smoothing of the surface around the intersection of the two initial surfaces. Additionally, restricting the speed function to either only positive values or only negative values, restricts the operator to only add (outwards motion only) or remove (inwards motion only) material. The effect of this operator is demonstrated in figure 6.6. As this method is applied to the surface near the intersection *after* the CSG operation is performed rather than on the embedding function during the CSG operation as in [19], it does not suffer from the problem with nearby surfaces being merged together even if they not intersect. As described in chapter 8, this operator is crucial to us in order to produce a single closed surface of high quality from our mapped geometry.

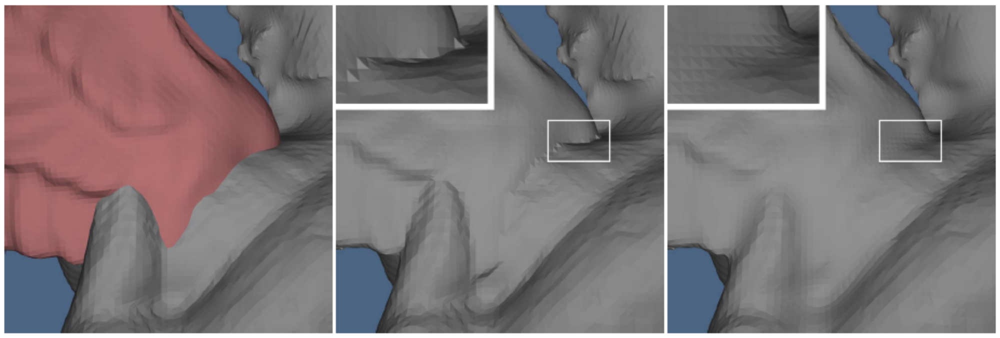


Figure 6.6: Localized blending using mean curvature flow. Left: Closeup of a wing (red) and dragon model to be merged together.Middle: Merging the two models together using a CSG union produces sharp, undesirable creases. The box shows a zoom in on a portion of intersection showing this. Right: Same region after automatic blending based on mean curvature. The blending is constrained to only move outwards. Reprinted with permission from [53].

The second operator that is important to our work is the embossing operator. This operator is implemented through point set attraction or repulsion. Any number of points are placed on or near the surface. Then the distance based cut-off function, $\mathcal{D}_q(d)$, is set to depend on the distance to the nearest point. The geometric cut-off function, $\mathcal{C}(\gamma)$, is set to be non-zero only for points where the surface normal points in the direction of the nearest point (or points away from it if the point is inside the volume), to ensure movement of the surface only in the direction of the point set. Finally, $\mathcal{G}(\gamma)$ is set proportional to ϕ at the nearest point, which forces the interface to stop moving once it reaches one of the points. Although not directly related to our work, this operator has served as a great inspirational source, as we in many ways consider our work an extension to this functionality. Many of our original ideas were inspired by this operator and this framework in general, and although it may not shine through in the final algorithms presented in this thesis, this work has had a great impact on the paths explored during the development of our algorithms. All in all, we consider our work an additional tool to be added to the implicit modeling toolbox initiated by this work.

6.4 Visualizing Level Sets and Implicit Surfaces

Unlike explicit surface representations, such as triangle meshes, implicit surfaces have no natural visual representation. Where a triangle mesh is represented by a list of triangles that are easily displayed, an implicit surface is given by a (possibly discretely sampled) function, whose *kernel* defines the surface. Ultimately, we have two possible approaches to visualizing an implicit surface: We can either approximate it using an explicit representation; or we can try to visualize the surface directly by determining what pixels on the screen it projects onto.

Using an explicit surface representation to approximate an implicit function is usually done using either triangles or points/point-sprites.

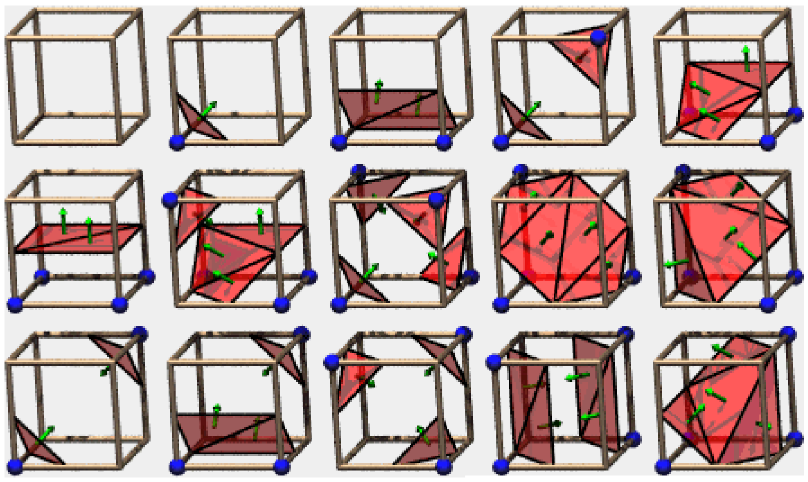


Figure 6.7: Triangulation of the 15 basic cubes used by the marching cubes algorithm.

The most common approach is to use the *Marching Cubes* algorithm [47]. This algorithm divides space into a discrete set of cubes, or *voxels*, which are then processed individually. If the implicit function has all positive or all negative values in the 8 corners of the voxel, then it is either completely outside the surface or completely inside, and no triangles are generated. Otherwise, the voxel is intersected by the surface, and the sign of the values in each of the 8 corners can be used to determine the configuration of triangles for that particular voxel. The values in the 8 corners are then subsequently used to find an improved estimate of the location of the triangle vertices using linear interpolation. In the original paper, the triangulation of different basis voxels were given, see Figure 6.7, leaving the remaining 241 possible configurations to be determined through rotational and complementary symmetry. However, a couple of those base configurations turned out to have a problem with respect to the complementary symmetry: If a, possibly rotated, voxel matches a given base configuration, only with the opposite sign in all corners, then the triangulation used is the same as the one described by that basis configuration, but with the triangles facing the opposite direction. Although this leads to a consistent meshing scheme, it does in some cases lead to holes in the extracted mesh. To fix this, another 8 base configurations were added to replace the *automatically generated* triangulation for the complement of 8 of the original base configurations. This yields a total of 23 base configurations, which are used to generate the remaining 233 configurations. Combined with a narrow band level set representation as described in section 6.1, the marching cubes algorithm is fast, as utilizing the narrow band, we know exactly what cubes will have corner values of different sign, and only these cubes need to be examined. When using the marching cubes algorithm to extract a mesh from an implicit surface, it is important that the function defining the surface is smooth and well behaved. If this is not the case, then the linear interpolation used to find the vertex positions will produce inaccurate results. Again, the signed distance function turns out to be a good choice for representing implicit surfaces. In the more than 20 years that have passed since the invention of the marching cubes algorithms, a number of improvements have been added to the algorithm, primarily intended to improve the quality of the extracted mesh [46, 58]. One of the problems with the marching cubes algorithm (and the signed distance function representation of implicit surfaces) is it's failure to properly handle implicit surfaces with sharp features. The problem arises when a sharp feature (an edge or a corner) is present inside a voxel. The triangulation created by the marching cubes algorithm will only have vertices located on the edges of the voxels, but a proper representation of a sharp feature located inside a voxel would require a vertex to be located inside the voxel rather than on the edge. Kobbelt *et al.* [40] addresses this issue by proposing an extended signed distance representation as well as an extension to the marching cubes algorithm. The *directed distance function* is a three component signed distance function, storing for each grid point the signed distance to the interface in the x, y and z direction. This representation allows for a more accurate estimation of the surface along the edges of the voxels. This is combined with an extended marching cubes algorithm, which follows the approach of the original marching cubes, but additionally, it locates the voxels containing a sharp feature. Then, additional sample points lying on the features are computed and added to the mesh. These *feature points* are estimated based on the local distance values and the gradients. Finally, a small post processing step is required, flipping the edge shared by all triangle pairs where the flipped edge connects two *feature points*. Although the combination of a directed distance function and the extended marching cubes algorithm produces the best results, the two methods are independent and can easily be used without the other. Figure 6.8 shows an example of a mesh extraction performed on a 65^3 volume dataset using the original marching cubes method with a standard signed distance function as well as the improved directed distance function, and the extended marching cubes method on the same two variants of the dataset.

Despite the many improvements to the marching cubes algorithm, there is one *limitation* of the algorithm that was only just solved recently: Shortly after the presentation of the algorithm, the inventors were granted a 20 year patent on the algorithm in USA in 1985. As a result, a number of alternative algorithms, based on similar techniques have appeared through the years, including marching tetrahedra [21] and marching triangles [2, 3].

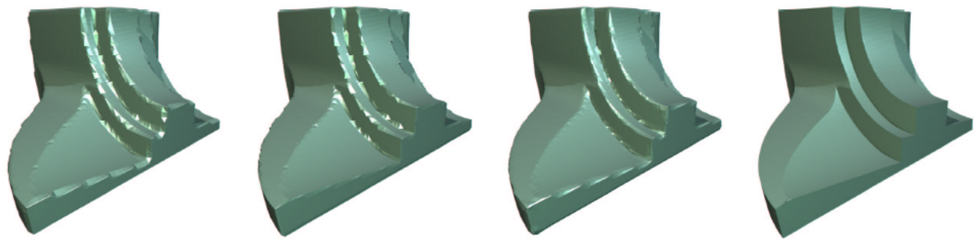


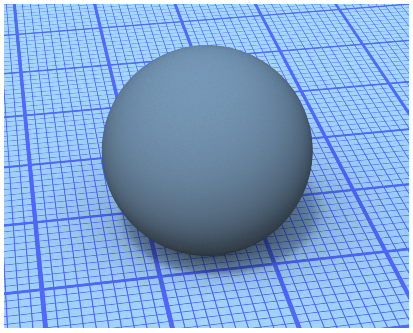
Figure 6.8: Triangulation of a *fandisk* volume using the original marching cubes method (left), using the original marching cubes method on a directed distance function (center left), using the extended marching cubes method (center right), and using the extended marching cubes method on a directed distance function (right). The approximation error to the original polygonal model is below 0.25% when using the extended method on a directed distance function. Reprinted with permission from [40].

Another approach to explicitly approximate an implicit surface is to distribute a large number of particles on the surface. This is done using e.g. the techniques described by Witkin and Heckbert [88]. Each particle is then rendered using a point sprite of a given radius. This approach, usually dubbed *point splatting*, is fast and easy to implement, but has the disadvantage that if the point sprites are either too small or too large, artifacts will appear. As such, this approach is useful only for fast visualization of intermediate results. Alternatively, the particles can be used to generate a triangle mesh as described in [17].

Direct rendering of implicit surfaces is usually done by means of ray tracing or ray casting. For each pixel on the screen, one or more rays are traced from the eye point through that pixel. If these rays intersects the implicit surface, the intersection point is found, and the color of the corresponding pixel is determined from the material assigned to the surface, the light reaching the intersection point from light sources and/or the environment, the surface normal at the intersection point and the ray direction.

Finding the intersection point between an implicit surface, $\phi(\mathbf{x})$, and a ray, $\mathbf{r}(t) = \mathbf{o} + t\vec{d}$, amounts to solving the equation $\phi(\mathbf{r}(t)) = 0$, for $t > 0$ (Solutions to the equation with $t \leq 0$ corresponds to intersections with the ray behind the ray origin, and are thus not considered.). If no solution is found, then the ray does not intersect the surface, whereas one or more solutions mean one or more intersections. If more than one intersection is found, the one with the smallest positive t corresponds to the nearest intersection, which is the one required. If the implicit surface is discretely sampled on a regular grid, we cannot solve this equation analytically, and have to resort to alternative methods. First, we step along the ray, evaluating, at each sampled point along the ray, the function by interpolating (trilinear or higher order) the surrounding grid values. Then, if at any point in this process, the sign of the function changes, then there must exist a point on the line segment between the current point and the previous point, for which the function evaluates to zero. In other words, there must be an intersection between the ray and the implicit surface in between these two points on the ray. Now, all that has to be done is to find the exact location of this intersection point. A simple but efficient method is to use the Secant method for root finding [39] with the t values of latest points as start condition.

Stepping through the volume, looking for the first point of opposite sign can be quite expensive. Fortunately there exists several different approaches to speeding this up, utilizing octrees, nested regular grids [63], KD-Trees [81] etc. to quickly skip over large areas of empty space. If however, the implicit surface is given by a signed distance function, which is the case throughout this thesis, a much simpler and equally efficient approach is what is known as ray leaping. Since the absolute value of $\phi(\mathbf{x})$ is by definition the *shortest* Euclidean distance to the surface, we know that we can safely skip $|\phi(\mathbf{x})|$ units ahead as we are then guaranteed that no intersection can exist between the current point and the new point. This works well until $|\phi(\mathbf{x})|$ is below a certain threshold. At this point, we need to use a more accurate approach to ensure that we don't stall due to a step size virtually identical to zero, nor skip over the interface due to taking too large steps.


(a) Visualizing an implicit surface using volume ray tracing.

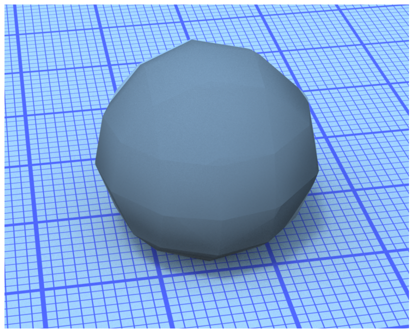

(b) Visualizing an implicit surface using marching cubes.

Figure 6.9: Visualization of an implicit surface. The sphere is represented as a level set sampled on a $20 \times 20 \times 20$ grid, and rendered using direct ray tracing with high order interpolation (a) and through a mesh extracted using the marching cubes algorithm (b). To make the comparison fair, the extracted mesh is ray traced in the same environment as the direct implicit rendering.

Using ray tracing or ray casting to visualize implicit surfaces is usually much slower than using an explicit approximation, but the images generated are typically much more accurate as it does not rely on a polygonal approximation. Instead, an intersection is computed using the trilinearly interpolated sample points or, if so desired, even using a higher order interpolation scheme. Figure 6.9 shows the rendering of an implicit surface using respectively marching cubes and ray tracing. Throughout this thesis, when interacting with implicit surfaces, we will use a triangle mesh extracted using the marching cubes algorithm for rendering as this allows for interactive frame rates. For intermediate results, or when illustrating errors, things that went wrong etc., screen shots, from the application we have developed during this project, will be used. This application uses OpenGL to render one or more triangle meshes extracted using the marching cubes algorithm. For final results on the other hand, the rendering is done using ray tracing, as in this case, the image quality is more important than the render time.

Chapter 7

Related Work

Our work builds on level set, implicit surface modeling as well as volumetric and geometric texture research. A recent body of work proposing various compact data structures and fast algorithms for level set models [36, 48, 56] is critical to our work. Common to all these data structures is that they *uniformly* sample distance values to a surface in a narrow band embedding the surface. This uniform sampling is paramount to perform our smooth blending operations since they amount to solving mean curvature based level set equations. This effectively requires spatial discretization of parabolic partial differential equations which, to the best of our knowledge, cannot be accomplished accurately on non-uniform grids. Consequently, we have not considered adaptive distance fields (*i.e.* "truly adaptive" octrees) [27], though other parts of our texturing pipeline (*e.g.* CSG operations) could potentially benefit from it. Instead we have chosen to base our texture mapping technique on the "Dynamic Tubular Grid" (DT-Grid) presented in [56]. This data structure has been shown to be very CPU and memory efficient and typically allows us to represent level set models of effective resolutions exceeding 1000^3 using less then 100MB.

Much effort has been put into deriving methods for adding textures to unparameterized 3D models, specifically implicit surfaces and level sets, including vector field driven texture synthesis [80] and methods based on parameterizations of *support surfaces* of lower geometric complexity combined with a mapping from the support surface to the actual surface [79, 92]. Common for these methods is a lack of flexibility and user control. Pedersen [64] presented an interactive method to create a parameterization of implicit surfaces by letting the user manually divide the surface into rectangular and triangular *texture patches*. His system works by distributing a set of particles on the surface using a particle system like the one described by Witkin and Heckbert [88]. The surface is then divided into rectangular and triangular patches in the following manner: First three or four particles are selected as corner points for a patch. Then the edges of the patch are formed by approximating geodesics between the corner points. First, a rough approximation to the geodesic is created using Dijkstra's shortest path algorithm [30] on a graph of the particles. This rough approximation is then optimized further to provide a better approximation to the geodesic. The geodesics are then discretely sampled, and a number of u- and v-curves are created inside the patch between the sample points on the curves. The u- and

v-curves are optimized using either an approach similar to the geodesic optimization or 2D flow. Given these internal curves and the edge curves, the patch now has a natural parameterization, and moreover, these curves are also used to create a high resolution triangulation of the patch. Finally, when patches are created for the entire surface, a set of texture *decoration* operations can be performed. The strength of this system is its flexibility. It can be used to apply a texture to the entire surface, or to only a part of the surface, either way, the user is in full control. This method has generally been considered as state of the art since its publication in 1995. Recently, Schmidt et al. presented a local parameterization based on discrete exponential maps [71], producing a simple yet powerful interface applying localized textures (also known as decals) to implicit surfaces. Provided only a local parameterization is required, this method appears to be as flexible as that of Pedersen, but with a significantly more simple and intuitive interface.

Kajiya and Kay introduced the notion of volumetric textures [38]. Their method utilizes volumetric data sampled on a regular grid, and traces rays through a shell volume on a surface. Rays that intersect the shell are transformed to texture space and traced through the sampled data grid. Material properties were constrained across any region. Neyret extends volume textures, allowing the use of multiple different materials in a single region, and objects of different types to be tiled onto a surface [54]. Wang *et al.* present a generalization of displacement maps. For each location in a grid surrounding the base surface, a distance is computed to the geometric texture, called the mesostructure, for some discretization of all directions. Several other variables are precomputed for rendering, including BRDF information and local shadows [83]. Peng *et al.* [66] averaged distance field functions to generate offset surfaces. Then 3D volumes are sliced into 2D textures, and the textures are applied to various levels of the offset surface. The technique allows interactive rendering of the resulting volume.

Fleischer *et al.* propose to use a biologically inspired *cellular texturing* technique to produce organic looking surface details [26]. While producing impressive results, their modeling approach is not very intuitive to use due to a rather complicated underlying biologically motivated simulation engine. Bhat *et al.* demonstrate a volumetric extension of the image analogies technique [33]. This allows them to tile a surface with semi-repeatable patterns at high effective resolution. The patterns do not need to be height fields, and can represent complex structure on the surface [7]. These methods allow for semi-automatic generation of very complex surface geometry, but they appear to add a significant amount of noise to the base geometry (see *e.g.* figure 8 and 9 in the original paper). Furthermore, the vector flow based parameterization leaves the user with little control over the final result.

Recently shell maps [67] generalized the notion of volumetric textures by mapping explicit geometry without converting models into regular grids. Shell maps are invertible mappings between texture space and shell space – the space near an object – that facilitate the transfer of explicit geometry, procedural functions, and scalar fields as fine scale detail near an object. Their method starts by creating an offset surface from the original mesh. This offset surface has the exact same topology and connectivity as the original surface, and is generated by offsetting a copy of the original mesh's vertices a user supplied distance in the direction of the surface normal. In case of a surface self intersection, the involved vertices are moved backwards until the self intersection is avoided. Then, a prism is defined between each corresponding pair of triangles on the original mesh and the offset surface, and each prism is then further divided into three tetrahedrons. Using the s and t texture coordinates from a predefined parameterization of the original mesh, and an r coordinate of 0 for vertices on the original mesh and 1 for vertices on the offset surface, a mapping between texture space and shell space is defined through point location queries coupled with barycentric interpolation on the tetrahedrons. The technique is powerful, but the resulting mappings are only C^0 at tetrahedral boundaries and can create artifacts like the one shown in figure 11.1(b). Furthermore, the mapped geometry and the base mesh do not create a new closed mesh, which can be problematic for applying shaders over the entire surface. The level-set representatin we present complements the explicit geometry representation of Shell Maps by more naturally dealing with sharp discontinuities and changes in topology necessary to generate closed surfaces (when desired). Unlike the method we are presenting, the shell map method requires a global parameterization of the base geometry.

Chapter 8

Geometric Texture Mapping

8.1 Notation and overview

As a prelude to a more detailed presentation of the techniques proposed here, we introduce the following terminology. We use the term geometry interchangeably for both explicit meshes and implicit level sets. Assume we wish to map a geometric texture, A, onto a base surface, B. We shall denote the explicit mesh representation of A as M_A and the implicit level set representation by ϕ_A. The geometric representation of B is always implicit, and will therefore be denoted ϕ_B. The embedding space of A (e.g. defined from its bounding box) will be called *texture space*. The corresponding embedding space of A, after it has been mapped onto B, is called *patch space* (analogous to a portion of "shell space" [67]). The semi-implicit texture mapping then simply works by defining a map of vertices of M_A from texture space to patch space. In contrast, the implicit texture mapping is based on a re-sampling of ϕ_A into patch space which amounts to establishing a map from grid points in patch space to texture space. Thus, both techniques are based on establishing a mapping between the two embedding spaces, but in different directions (see figure 8.1).

We assume that we are given a base surface as a compact level set (*e.g.* a DT-Grid) and a geometric texture either defined by a triangle mesh or as a compact level set surface. If required, conversion between triangle meshes and level sets can be performed using Mauch's fast scan conversion technique [51] or marching cubes [47]. Given this geometry, our system briefly works as follows:

- First, the user manually outlines a patch on the base level set which defines the location of the geometric textures. Given such a patch outline, we then construct a parameterization of the space above the patch. This effectively creates the mapping needed to warp the texture into the space near our base level set surface.

- With the particle based parameterization in place, the user can map the texture mesh onto the base level set at nearly interactive rates using our semi-implicit mapping.

- Alternatively, the user can utilize a higher quality implicit mapping, which maps an implicitly defined geometric texture onto the base surface. The warped implicit texture surface can then be blended with the base surface into a single topologically connected surface with a smooth intersection between the two previously separate surfaces.

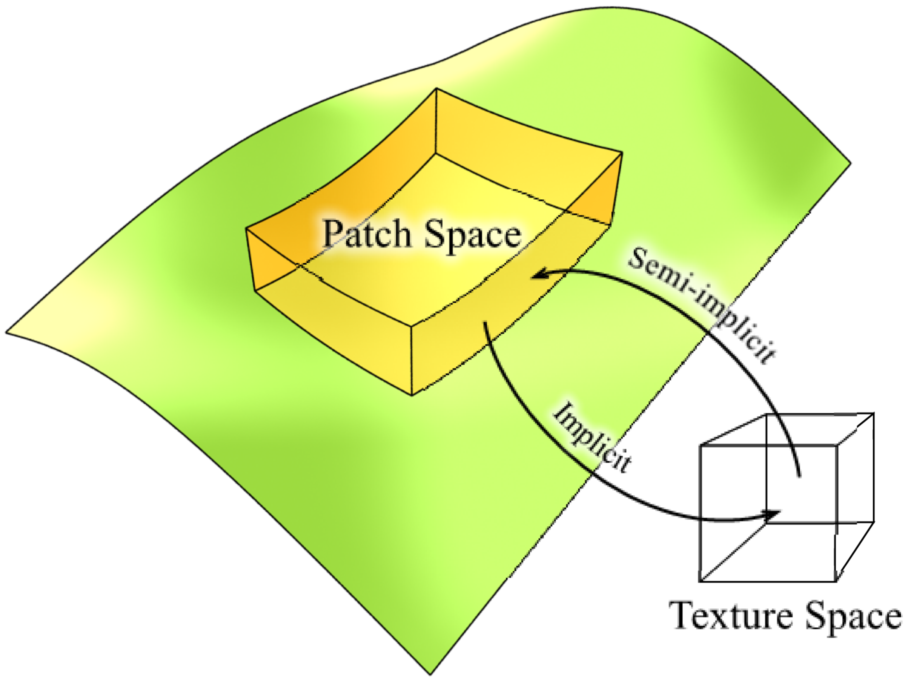


Figure 8.1: Defining the relationship between texture space and patch-space, defined on a portion of the surface. The arrows indicate the mappings performed when applying an implicit geometric texture and when applying an explicit geometric texture.

8.2 Parameterizing Patch Space

While the volumetric parameterization of texture space is assumed known (*e.g.* $u = x, v = y, w = z$), we have to derive the warped parameterization of the corresponding patch space. For this we have developed a number of techniques, based on an initial u, v parameterization of a 2D patch of the base surface and using Lagrangian tracker particles to sweep out u, v, w in the corresponding patch space. However, the distribution of these tracker particles is done in different ways thereby offering distinct features, such as following the base surface faithfully or lowering distortion, for the resulting 3D texture mapping. This flexibility is one of the strengths of our system. In the following we describe our three currently available particle distribution methods as well as their common base.

A common initial step for all our current mapping techniques is the definition and parameterization of a 2D patch on the base surface where the texture is to be applied. We define this patch as a simple control quadrilateral on ϕ_B[1]. Constrained interaction with the vertices, $V_i, i = 1 \ldots 4$, of this control quadrilateral is easily implemented since projections of V_i onto ϕ_B amounts to the closest point transform $V_i - \phi_B(V_i)\nabla\phi_B(V_i)$. This is a consequence of our requirement that the level set, ϕ_B, is represented by a signed distance function. This control quadrilateral is parameterized using a technique similar to Pedersen's [64]. In short, approximate geodesics are first computed between the vertices V. These edges are then subdivided evenly, with a resolution determined by the roughness of the surface[2], and assigned u, v coordinates. Next u, v are swept into the interior of the quadrilateral by means of defining a 2D grid of iso-parametric curves of approximate geodesics connecting the subdivided edges. At each of the grid points of this 2D iso-parametric grid we place a Lagrangian tracker particle, *i.e.* an infinitely small and massless particle, each associated with a unique u, v, w coordinate. In the following we refer to these Lagrangian tracker particles as patch particles or just particles. The position of the patch particles are then optimized to reduce texture distortion. This is achieved by means of a simple constrained mass-spring model [68] where particles on the boundary curves of the patch quadrilateral are fixed and the remaining interior particles are restricted to lie on the base surface.

Surface conforming parameterization: Once the patch particles are generated on the base surface, we propagate the particles along the gradient field of ϕ_B until they reach the desired offset (*i.e.* level of ϕ_B). The w coordinate for the advected particles is then defined to be 1. In the case of the implicit mapping described in section 8.3, it is often necessary to have intermediate layers of particles with $0 < w < 1$ (see section 8.5). This is obtained by distributing a number of particles evenly on the line segment between each advected particle and it's corresponding particle on the surface, using linear interpolation to determine the w value. Figure 8.2 illustrates the particle set distributed for a single patch using this method. Note thateven though ϕ_B is defined as a signed distance function, two particles with the same w coordinate will generally not lie at the same distance away from B (Unless $w = 0$). This is a consequence of the fact that the gradients are strictly speaking not defined at points that have more than one closest point transform to B since here ϕ_B is only C^0. This occurs along the medial-axis of B and numerically manifests itself as $|\nabla\phi_B| \leq 1$ when using central finite difference to compute the gradient. This has the desired feature that although the advected particles might reach other particles, they will never cross paths[3]. As the particles generated by this method are generally not uniformly distributed in patch space, this can lead to

[1]Note that this is not a regular planar quadrilateral since the edges are constrained to lie on the base surface.

[2]As we assume ϕ_B is regularly sampled with $dx = dy = dz$, keeping the sample distance below dx guarantees a sufficient sampling. If, however, the surface is smooth, a lower sample rate is often sufficient.

[3]Numerical roundoff errors and inaccuracies in the finite difference potentially breaks this guarantee, although such particles are still guaranteed to remain close together.

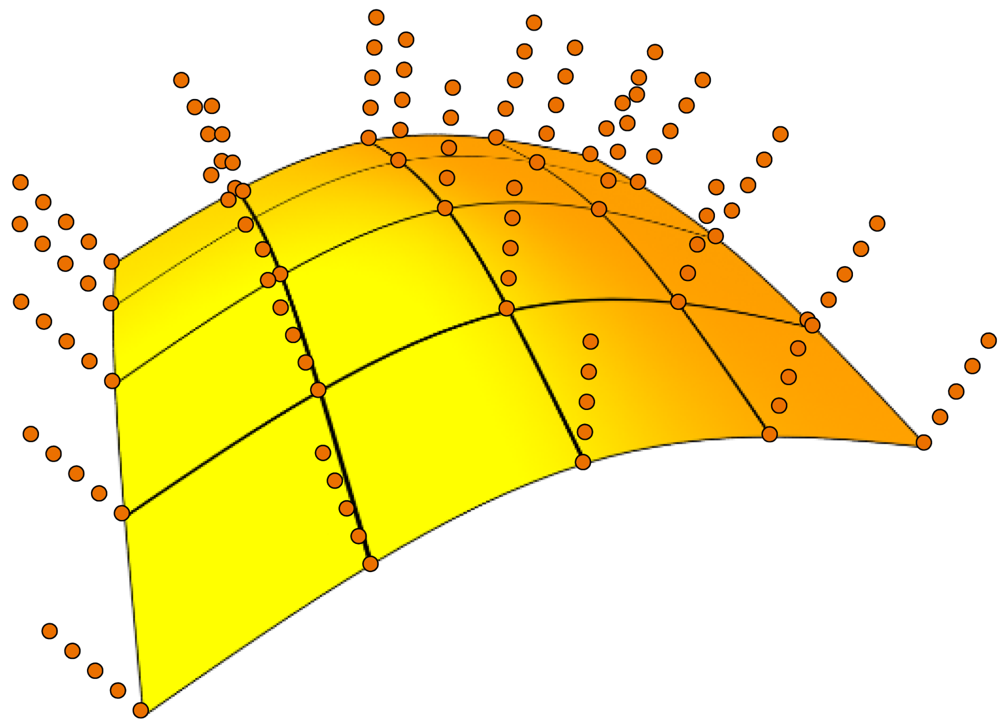


Figure 8.2: The surface conforming parameterization propagates the (u, v) coordinates using a Lagrangian advection method. The particles roughly follow the normal direction, which will spread the parameterization in convex regions and cluster them in concave regions.

significant distortion of the geometric texture. We note, that depending on the application, this may or may not be a desirable feature.

By distributing the tracker particles as just described, we end up with a mapping that is in many ways similar to shell-mapping [67]. Consequently this distribution scheme is hampered by most of the limitations of Shell Map, in particular the sensitivity of the mapping with respect to the curvature of the base surface (See section 8.5). However, one of the main strengths of our method is the flexibility with respect to the method used for distributing the tracker particles. We next present two alternative particle distribution schemes that offer different and improved properties of the resulting geometric texture mapping.

Reduced distortion level set parameterization: The problem with the previous particle distribution method is the (implicit) dependence of the curvature of the base surface. As the tracker particles are advected away from the surface in a direction normal to the surface, small irregularities in the surface can cause severe distortion of the texture due to particles moving closer together in concave regions and away from each other in convex regions. To mend this, we introduce a particle distribution scheme with a stronger focus on the vertices of the user specified control quadrilateral. With this method, these vertices

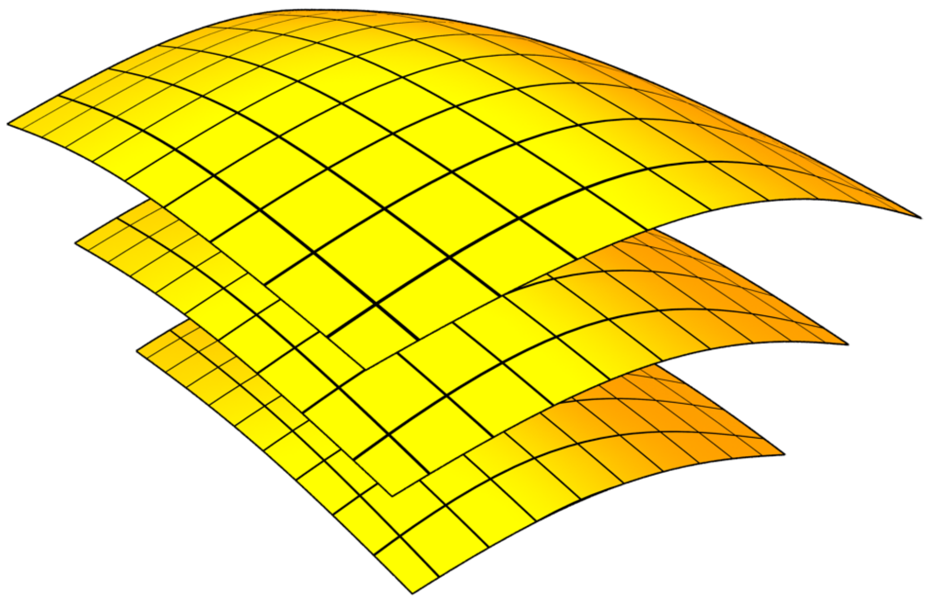


Figure 8.3: The low distortion parameterization can be thought of as uniform layers of an onion. The particles are advected as with the surface conforming parameterization, but then they are relaxed to give each level a uniform parameterization.

are the only particles to be offset along the gradient field of ϕ_B. At regular intervals, derived from the desired offset height and the desired number of particle levels, a new level of particles is created from the four advected control vertices. We do this using the same technique as used for the particles on the surface, only this time we embed it on the ϖ'th level set of ϕ_B, where ϖ is the (fictitious) time during the propagation. The particles at this level are assigned a w value ϖ divided by the desired offset height. The overall result is a uniform parameterization of each discrete level in the patch space, see figure 8.3, leading to geometric texture mappings with significantly less distortion than the first method (see figure 8.5). This method has an additional number of advantages over the first particle distribution method. First, as a new set of particles are generated at the individual levels, the number of particles generated at each level are independent. Thus if the *surface area* of the patch changes with the distance to the base surface, we can adjust the number of particles generated at each level to maintain a desired particle density, thereby guaranteeing a sufficient sampling of each level. Furthermore, we can optionally let the user specify the direction along which each control vertex is offset, rather than forcing it to be in the normal direction. This is feasible as only four particles are offset from the base surface and therefore does not impose a significant extra burden

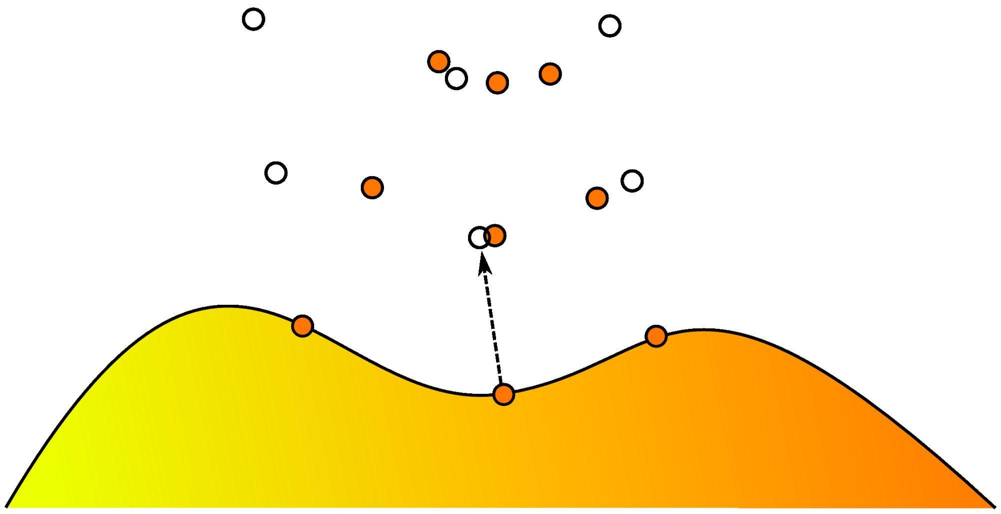


Figure 8.4: Specifying a different direction for the particles to evolve along adds extra flexibility to the parameterization. The white particles are obtained by specifying a custom direction of evolution, parallel to the normal at the center point, at both control vertices.

on the user. The effect of this is depicted in figure 8.4. By allowing the user to specify the offset direction, we add an extra level of control over the final result. This allows, for example, the user to control the distortion of a texture with a large offset in the w direction on a highly curved surface, as seen in figure 8.4. We have used this extra control in several examples in the following sections, most notably in figure 11.1(a).

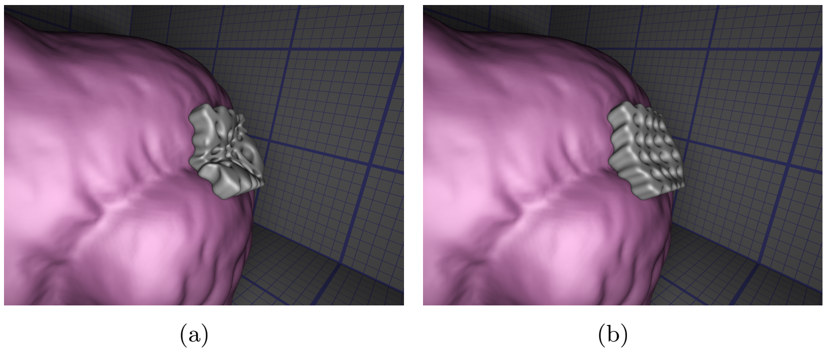


Figure 8.5: Left: Mapping a geometry texture to a bumpy part of the bunny using the surface conforming parameterization. Right: Using the low distortion parameterization.

The difference between the surface conforming and the low distortion parameterizations is illustrated in figure 8.5. This example shows how the mapping based on individually advected particles (figure 8.5(a)) follows the local curvature of the base surface more closely than the mapping based on uniformly distributed particles (figure 8.5(b)), whereas the latter reduces the overall dis-

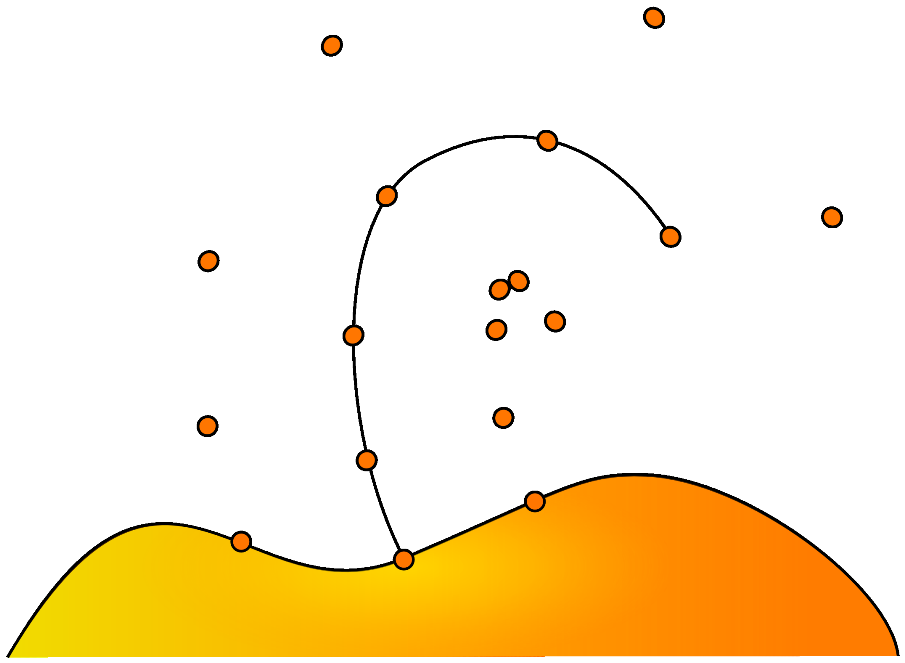


Figure 8.6: Parameterization of a patch (simplified to 2D) using the spline advection paramerization.

tortion of the mapped geometry.

Spline advection: The two particle distribution schemes outlined above both rely on the distance transform of the base surface (*i.e.* the level set ϕ_B) to respectively propagate the particles in the patch space. This effectively means that texture information is propagated in a fixed direction away from the base surface. To add more flexibility we have developed a third parametrization scheme where the particles are propagated along a spline curve originating at the center of the patch. It works as follows: As with the previous distribution schemes, we start by generating the particles on the base surface, assigning u, v-coordinates to each particle. The particles are then propagated in small steps in the direction defined by the spline curve. At each step, the particles are furthermore rotated around the current spline point to align with the tangent of the curve at that point, see Fig. 8.6. As in the previous methods, copies of the particles are saved at regular intervals, and a w-coordinate, derived from the normalized distance traveled along the spline, is assigned to each particle. An example mapping generated with this technique is shown in Fig. 8.7. As the particles move along the spline, it is often desirable to slowly lessen the influence of the surface curvature, see Fig. 8.8. Initially, due to the curvature of the base surface, the particles will be placed at different distances from the plane tangent to the surface at the origin of the spline. With the approach described above, the propagated particles will stay exactly this far away from the plane perpendicular to the tangent of the current point on the spline. Since we can always find this plane, it is however easy to let the particles move closer to the plane such that once the particles are propagated all the way to the end

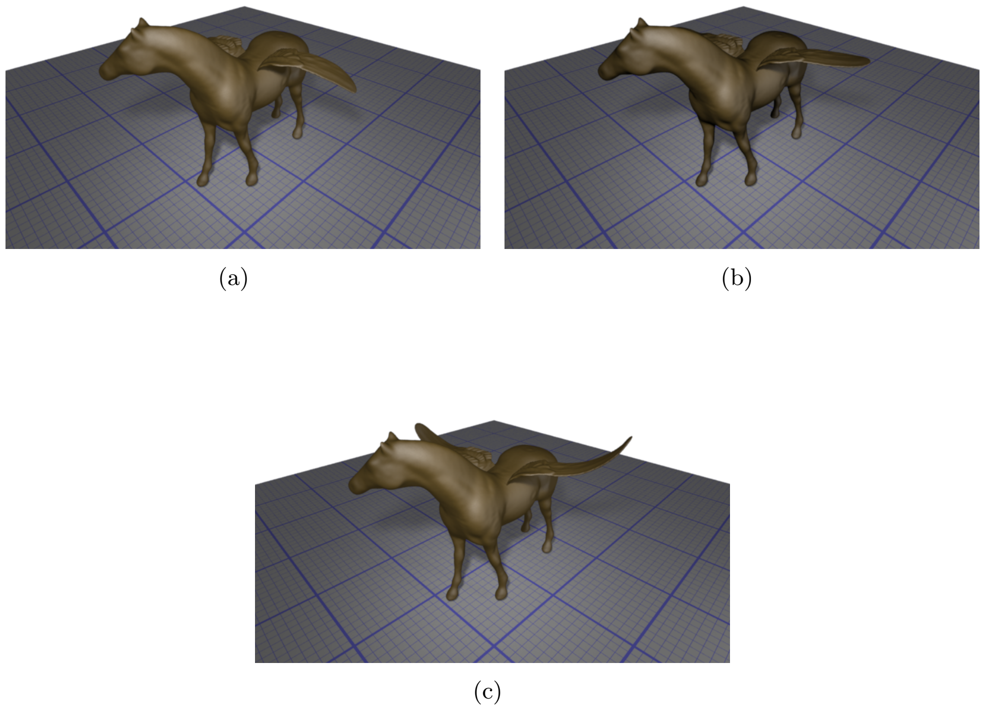


Figure 8.7: The three images show a horse with wings mapped onto it in three different postures using the spline based particle distribution scheme.

of the spline, all particles will lie exactly on this plane.

We note that during the propagation of the particles along the spline curve, care must be taken to avoid particles crossing paths. This would potentially lead to non-monotonic interpolations of the corresponding texture coordinates which in turn result in inconsistent texture mappings. One possible solution to this problem is to treat the advancing particles as small spheres and apply *continuous collision detection* algorithms [32] to ensure that particles do not cross. Continuous collision detection algorithms, while more difficult to implement, offer several advantages over their discrete counterparts. Most notable are their ability to compute the time of first contact versus the discrete approach of simply sampling an object's trajectory and reporting intersections (small, fast moving objects could pass through each other).

As a final remark we note that both the surface conforming- and the low distortion parameterization assume that ϕ_B is defined throughout the patch space. Since we employ a very storage efficient level set representation of ϕ_B, [55], distance information is only stored in a narrow tube of B. Hence, as a prelude to these two methods we first sweep out distances from this narrow tube to the remaining patch space (which is typically a very small sub-space of the bounding volume of B). This has been implemented efficiently using the *fast sweeping* method [90] which has linear time complexity in the number of

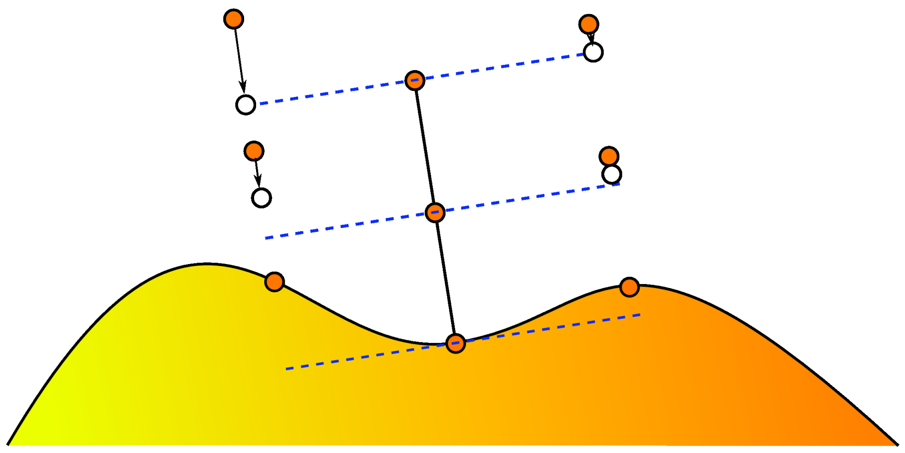


Figure 8.8: The effect of the surface curvature can easily be made to fade away as the particles are advected along the spline. For simplicity, the spline in this figure is just a straight line. Blue dashed lines indicates the plane perpendicular to the spline tangent, and white dots indicate the updated particle positions

voxels in the patch space.

8.3 High-Quality Implicit Mapping

The implicit mapping allows us to warp the geometric texture to follow the curvature of the base surface, and subsequently blend the level set representations of both the warped texture and base surface into a single surface. Our mapping is based on radial basis function interpolation of the texture coordinates associated with the patch particles. The algorithm is as follows. First, we define a regular 3D grid, bounding the region of space spanned by the patch particles, we call this the *embedding volume*. The resolution of this grid is chosen to match the resolution of the grid on which the texture level set is sampled in texture space. Next we define a mapping from patch space to texture space, using radial basis functions to interpolate over our correspondence points between these two spaces. The resulting texture coordinates are then used to resample our texture in patch space. Essentially, for each grid point x_p in the embedding volume, we map it to texture space via the radial basis function interpolation, getting the point x_t. We then use the point x_t to perform an interpolation[4] on the texture volume to get the texture function value $\phi_A(x_t)$, which we assign to the grid point x_p. Once all points in the grid have been updated with a texture function value, the embedding volume will contain a warped instance of the texture geometry.

The method we use for our radial basis function is similar to that of Dinh et al. [20], which is a good candidate because of its robustness with respect to irregularities of the sample points. Furthermore it adds flexibility due to the fact

[4]In our case, this is a trilinear interpolation, but higher order interpolation can of course be used instead, if desired.

that it allows for both strict interpolation as well as a smoother approximation. For the sake of completion we will summarize this technique below.

Assume the patch particles have Cartesian coordinates $\{\mathbf{p}_i, i = 1...n\}$ and texture coordinates $\{k_i, k = u, v, w, i = 1...n\}$, as described in section 8.2. Now we wish to establish a mapping from Cartesian coordinates in patch space to texture coordinates in texture space, $\Phi_{p\to t}$. The key idea is to split the mapping into three independent mappings

$$\Phi_{p\to t}(\mathbf{x}_p) = \mathbf{x}_t \Rightarrow \begin{array}{l} \Phi_{p\to t,u}(\mathbf{x}_p) = x_u \\ \Phi_{p\to t,v}(\mathbf{x}_p) = x_v \\ \Phi_{p\to t,w}(\mathbf{x}_p) = x_w \end{array}$$

with each of the texture mapping functions, $\Phi_{p\to t,k}$, expressed as a weighted sum of *radial basis functions*:

$$\Phi_{p\to t,k}(\mathbf{x}_p) = P_k(\mathbf{x}_\mathbf{p}) + \sum_{i=1}^{n} \omega_{k,i}\varphi(|\mathbf{x}_p - \mathbf{p}_i|), \tag{8.1}$$

where $\varphi : \mathbb{R} \to \mathbb{R}$ is a radially symmetric basis function; n is the number of basis functions; $\mathbf{p}_i$ is the center of the i'th basis; $\omega_{k,i}$ are the weights for the i'th basis for texture coordinate k; and $P_k(\mathbf{x}_p) = \rho_{k,0}x_x + \rho_{k,1}x_y + \rho_{k,2}x_z + \rho_{k,3}$ is a polynomial spanning the null space of the basis function. Similar to [20], we center a basis function at each patch point.

To find the weights and polynomial coefficients, for the k mapping, we apply equation (8.1) to each of the patch points. Since we already have assigned a k coordinate to each patch point, this leads to a linear system of $n+4$ equations with $n+4$ unknowns: To find the weights, $\omega_{k,i}$, and polynomial coefficients, $\rho_{k,j} = \{\rho_{k,0}, \rho_{k,1}, \rho_{k,2}, \rho_{k,3}\}$ for each mapping, $k = \{u, v, w\}$, we apply equation (8.1) to each of the patch points. Since we already have assigned a k coordinate to each patch point, this leads to a linear system of $n+4$ equations with $n+4$ unknowns:

$$\begin{bmatrix} \varphi(|\mathbf{p}_1 - \mathbf{p}_1|) + \lambda_1 & \cdots & \varphi(|\mathbf{p}_1 - \mathbf{p}_n|) & \mathbf{p}_1 & 1 \\ \vdots & & \vdots & \vdots & \vdots \\ \varphi(|\mathbf{p}_n - \mathbf{p}_1|) & \cdots & \varphi(|\mathbf{p}_n - \mathbf{p}_n|) + \lambda_n & \mathbf{p}_n & 1 \\ \mathbf{p}_{1,x} & \cdots & \mathbf{p}_{n,x} & 0 & 0 \\ \mathbf{p}_{1,y} & \cdots & \mathbf{p}_{n,y} & 0 & 0 \\ \mathbf{p}_{1,z} & \cdots & \mathbf{p}_{n,z} & 0 & 0 \\ 1 & \cdots & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \omega_{k,1} \\ \vdots \\ \omega_{k,n} \\ \rho_{k,0} \\ \rho_{k,1} \\ \rho_{k,2} \\ \rho_{k,3} \end{bmatrix} = \begin{bmatrix} k_1 \\ \vdots \\ k_n \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{8.2}$$

This linear system is solved for k and the resulting expansion coefficients $\omega_{k,i}$ are next used in Equation 8.1 to compute u, v, w coordinates on the 3D grid in patch space.

The λ values on the diagonal of the matrix in equation (8.2), allow us to control the smoothness of the mapping. As we have previously mentioned, each particle, $\mathbf{p}_i$ with position $\mathbf{x}_{p,i}$, corresponds to a specific position in texture space, $\mathbf{x}_{t,i}$. Thus, applying equation (8.1) to the position of a given particle should yield its corresponding texture space position, that is, $\Phi_{p\to t}(\mathbf{x}_{p,i}) = \mathbf{x}_{t,i}$. By adding the λ values to equation (8.2), we can relax this correspondence leading to the following inequality: $|\Phi_{p\to t}(\mathbf{x}_{p,i}) - \mathbf{x}_{t,i}| \leq \zeta_i$, where the constant ζ_i is deducted from λ_i. The larger λ_i is, the larger ζ_i will be, also if λ_i is zero then so is ζ_i. As the ζ values increase, the interpolation between the sample values becomes less restricted enabling a smoother interpolation, and thereby also a smoother mapping. For the results in this paper we have typically used two different λ values. Particles on the interface, that is particles with a w value of 0, are assigned small λ values to ensure that the mapping follows the interface closely. These values typically fall in the range 0.001 to 0.01. The remaining points are assigned a larger λ usually between 0.1 and 0.5 to ensure a smoother mapping away from the interface.

Using this mapping involves evaluating equation (8.1) at each grid point in the embedding volume. Thus if n_p is the number of particles and n_g is the number of grid points in the embedding volume, then the execution time for this mapping is $\mathrm{O}(n_g \times n_p)$. Due to using DT-Grids to represent our volumes, the memory required for the mapping is $\mathrm{O}(n_p + n_n)$, where n_n is the number of grid points in the narrow band of the embedding volume of the warped texture[5]. The setup cost is bounded by the linear system of equations in equation (8.2). Solving such a system can be done in time $\mathrm{O}(n_p^3)$ using $\mathrm{O}(n_p^2)$ memory using a standard LU decomposition [39].

8.4 Near Real-Time Semi-Implicit Mapping

Along with the implicit mapping just described, we have also developed a simple and efficient semi-implicit technique that maps an (explicit) polygonal mesh, M_A, onto the implicit base surface, ϕ_B. This mapping is useful either as a faster "preview mode" for our implicit mapping, or as a stand alone method for applying explicitly defined geometry textures onto a given base surface. Where the implicit mapping utilized a mapping from patch space to texture space, using an explicitly defined texture requires a mapping from texture space to patch space. Given such a mapping, we warp the vertices of M_A in texture space into patch space, leaving the mesh connectivity unchanged. This technique as well as the implicit technique can be used in combination with any of the parameterization methods described in section 8.2.

One possible way of defining a mapping from texture space to patch space is to follow the approach taken in the previous section and use radial basis function interpolation of the patch space positions of each particle, with a radial basis function centered at the texture space position of each particle. We would then apply equation (8.1) to each vertex in the mesh to transform the vertices from texture space to patch space. While this definitely would produce a valid mapping, we have chosen a much faster and more simple solution, which makes use of the fact that the patch particles form a *semi-regular* three-dimensional lattice in texture space - see figure 8.9, left. By this we mean that, in texture space, the particles are distributed into regularly spaced levels in the w direction. Each of these levels consists of a two-dimensional regular grid of

[5]In addition to this, we also need memory for the base geometry, the texture geometry and the warped texture geometry. However, to avoid confusion, we have chosen not to include this in the memory cost, as this is not really a consequence of the specific algorithm used for the mapping.

particles, but the number of particles need not be the same at all levels (See figure 8.9 for a two-dimensional example). Since the *texture value* associated with each particle is given by their position in patch space, we can define a mapping $\Phi_{t\rightarrow p}(\mathbf{x}_t) = \mathbf{x}_p$ of a vertex $\mathbf{x}_t = (x_u, x_v, x_w) \in M_A$ as a tri-linear interpolation of the particle texture values. As the number of particles may not be the same at each level in the patch space, we need to apply the interpolation in a specific order: We first interpolate at the two levels located immediately above and below the vertex in texture space, followed by an interpolation in between the levels. Figure 8.9 illustrates this: First the patch space position of the blue dots are obtained from interpolation along the green line segments. Next, we interpolate the values (patch space position) of the blue dots along the yellow line to get the patch space position of the vertex (red dot). Because each particle level form a regular two-dimensional grid, and the levels are uniformly spaced, finding the interpolants is a constant time operation. Thus, calculating the patch space position of a single vertex is also a constant time operation.

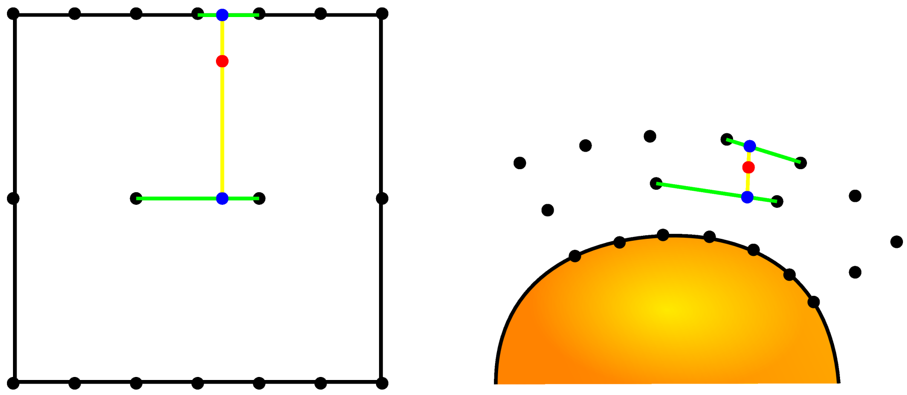


Figure 8.9: Illustrating the semi-implicit mapping on a patch set with a different number of particles at each level. To get the position of a given vertex (red dot) in patch space (right) given it's position in texture space (left), we first interpolate along the green lines to get the patch space position of the blue dots. These are then used to interpolate along the yellow line to get the patch space position of the texture space vertex.

We briefly note that the proposed mapping is somewhat reminiscent of the free-form deformation technique presented by Sederberg *et al.* [72]. The main difference is that our scheme can handle semi-regular samplings and is strictly bounded to the patch space. These properties are very important for our application and are not shared by the higher order interpolation proposed in [72]. Figure 5 in [72] clearly illustrates that geometry is not bounded to the control-polygon which in our application would result in textures mappings that are not explicitly confined to the base surface.

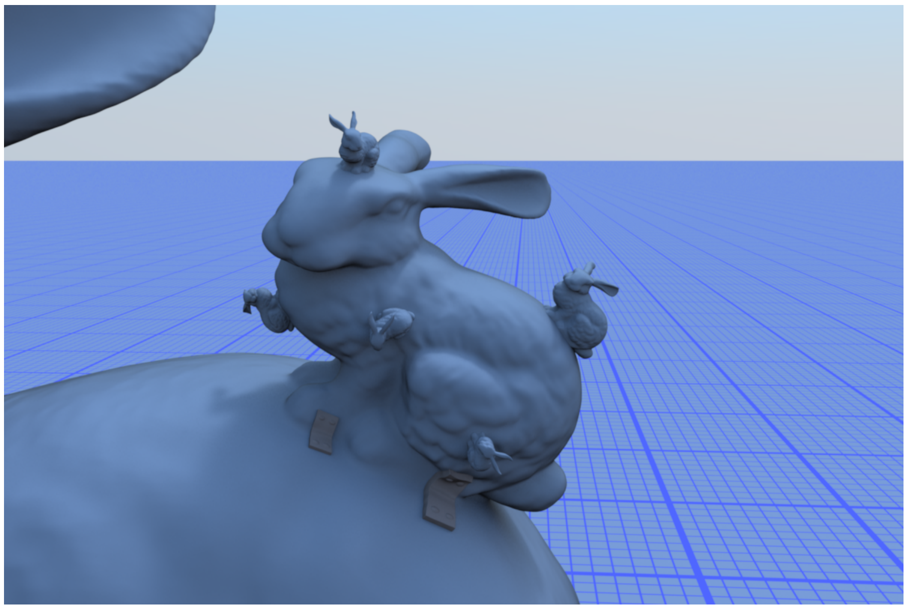


Figure 8.10: Recursive mapping: Mapping a bunny onto another bunny, which then again has a third bunny mapped onto it. The middle bunny is *held onto* the bigger bunny using a couple of metal latches.

8.5 Results and Applications

Since the implicit mapping uses level set representations for both the texture and the base geometry we can easily produce a topologically connected surface by merging the two volumes. This can be achieved with a boolean constructive solid geometry (CSG) union of the two level sets which simply amounts to a min (or max) operation of the distance fields followed by a re-initialization in the resulting narrow band. This however creates a very visible C^1 discontinuity along the intersection. To address this issue, we employ the techniques described in [53] which perform mean curvature based smoothing in the vicinity of the intersection of the two level sets. This approach allows for direct user control of mean curvature, and thus the smoothness, of the resulting volume. Both the merging/CSG union and the smoothing of the intersection are optional operators applied, if desired, once the mapping is completed.

Figure 8.13 shows a torus with several spikes mapped onto it using this technique. Figure 8.13(b) shows a close up of the intersection of the torus and a single spike merged into one without smoothing the intersection. Similarly, figure 8.13(c) shows a closeup of the same part of the torus, only this time the intersection between the torus and the spike has been smoothed.

Merging the base volume and the texture volumes into one volume this way introduces two new problems:

- How do we merge two volumes of different resolution?
- If the base and texture volumes differs significantly in resolution, what should the resolution of the final volume be? If we choose a resolution

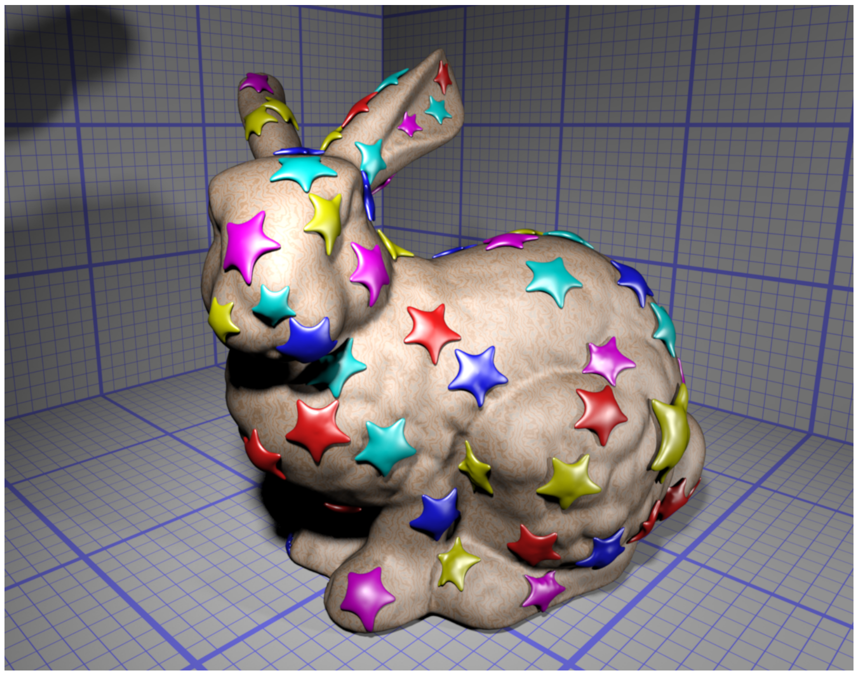


Figure 8.11: Bunny with 55 starfish models added. This figure is inspired by figure 6 in [67]

that is lower than the highest resolution of the volumes to be merged, then we risk loosing some geometric details. Resampling everything at the highest resolution may on the other hand result in a level set that is too large to fit in main memory.

The first problem is solved by re-sampling the volumes to a common resolution. In order to avoid re-sampling artifacts due to linear interpolation, we use the higher order interpolation described in section 6.2. The second problem is a bit harder to address. Typically we re-sample to the resolution of the highest resolution volume, as we prefer having to deal with memory issues rather than loosing geometric detail. In a worst case scenario, where the final volume becomes so large that it does not fit in main memory, we still have the option of using an out-of-core version of the DT-Grid representation [57]. Another viable alternative is to delay the CSG operation, and perform it while rendering. We have implemented such a *multi-volume CSG-on-the-fly* rendering plugin for our ray-tracer, allowing us to represent both the base volume and the warped texture volumes at their optimal resolution while rendering, and still be able to treat them as a single surface. Rendering is performed as described in section 6.4, with the following changes: Whenever we need to evaluate ϕ, we evaluate it on all of the volumes and return the smallest value. This is equivalent to the value we would get, had we performed the CSG operation beforehand. Once an intersection is found we need to find a surface normal for shading purposes. Rather than calculating the normal at the surrounding grid points (using finite difference) and then use tri-linear interpolation to get the final normal, we cal-

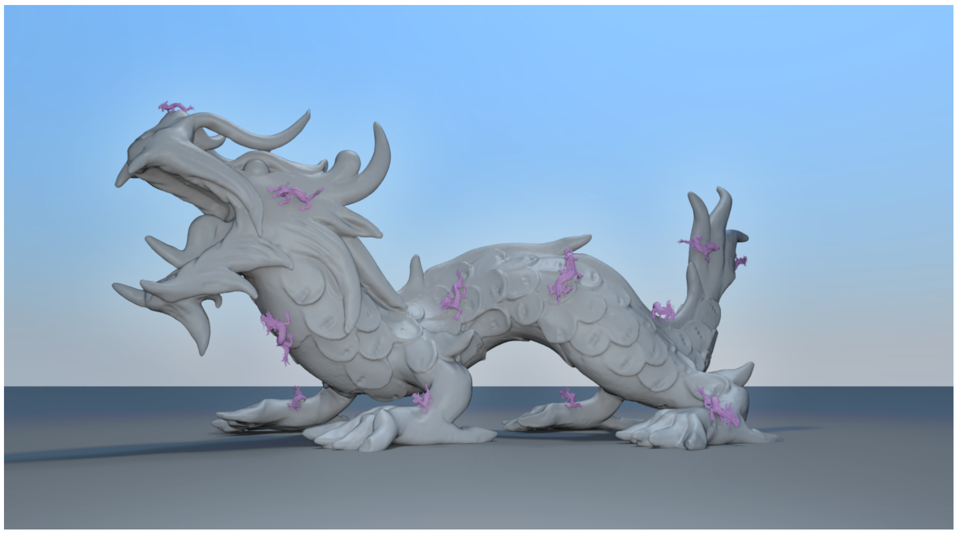
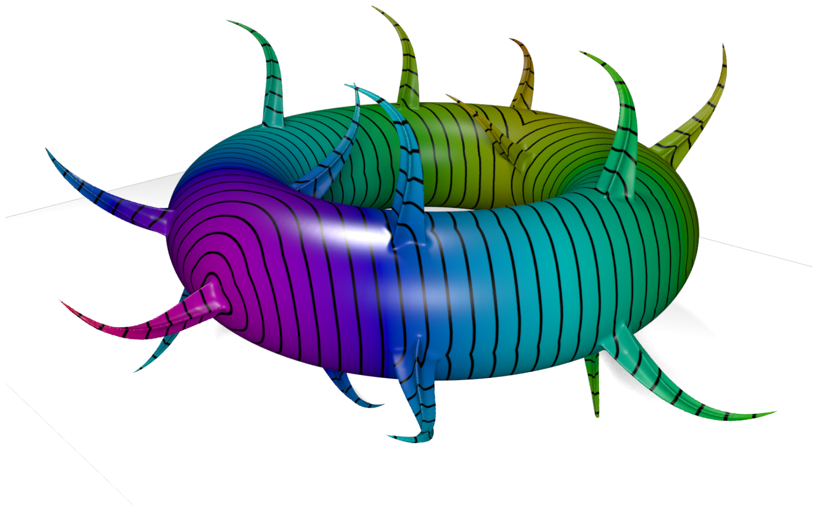
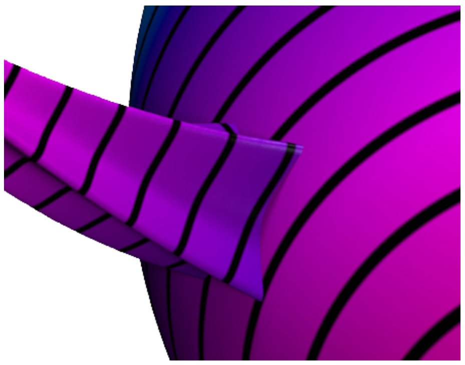


Figure 8.12: Mapping a number of small dragons onto a mother dragon using the proposed technique.

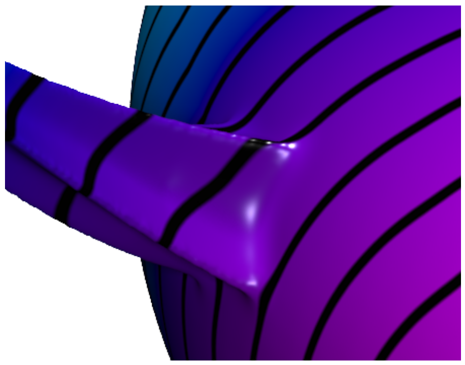
culate it directly at the intersection point using finite difference on interpolated ϕ values. This allows us to use the same evaluation scheme for ϕ as used for finding the intersection point, thereby guaranteeing a consistent shading. To calculate the normal we use a second order central difference scheme (equation 6.2 and 6.5), with a delta value (Δx in equation 6.5) equal to the grid point distance in the highest resolution volume. To avoid the re-sampling artifacts discussed in section 6.2, we need to use a high order interpolation method when calculating the normals. Using tri-linear interpolation for finding the intersection is usually good enough. Figure 8.14 shows an example of our *multi-volume* renderer using high order interpolation for the normals and tri-linear interpolation when finding the intersection point. In figure 8.15, tri-linear interpolation is used not only for finding the intersection point, but also for finding the normal (the base volume is sampled at a resolution that is roughly a factor of ten lower than that of the texture). In figure 8.15(a), the delta value is set to the grid point distance of the highest resolution volume, whereas in figure 8.15(b), the delta value is equal to the grid point distance of the lower resolution volume. In both cases, the rendering artifacts are very obvious. In figure 8.15(a), the base volume appears *flat shaded* due to using normals based on tri-linearly interpolated data, and in figure 8.15(b), the shading of the texture volume is hampered by inaccurate normals resulting from using sampled too far apart in the calculations. When using the CSG-on-the-fly renderer, we obviously cannot perform the curvature based local smoothing. What we can do is to apply a smooth CSG intersection operation as described in [19]. Although applying a smooth CSG operation may not always produce as good results as the localized curvature based smoothing, it does provide a nice alternative when using the CSG-on-the-fly rendering approach.



(a)



(b)



(c)

Figure 8.13: Mapping a number of spikes onto a torus. The closeups of a single spike shows the difference between not applying the CSG union (b) and applying the CSG union as well as the blending of the intersection (c). Notice the discontinuity in the shading in (b), which is a result of the spike and torus being separate geometries. Merging (and blending) the two surfaces resolves the issue (c).

Choosing basis function

In section 8.3, we presented our mapping based on radial basis function interpolation, however nothing was said about what function we use as our radial basis function. In this section, we will make a comparison of some of the most used radial basis functions in order to justify our choice of preferred basis function, which is $\varphi(r) = r$. The basis functions tested are the following:

1. The multi-order Laplacian basis [20]:
   $\varphi(r) = \frac{1}{4\pi\delta^2 r}(1 + \frac{we^{-\sqrt{v}r}}{v-w} - \frac{ve^{-\sqrt{w}r}}{v-w}),$
   $v = \frac{1+\sqrt{1-4\tau^2\delta^2}}{2\tau^2} \qquad w = \frac{1-\sqrt{1-4\tau^2\delta^2}}{2\tau^2}$

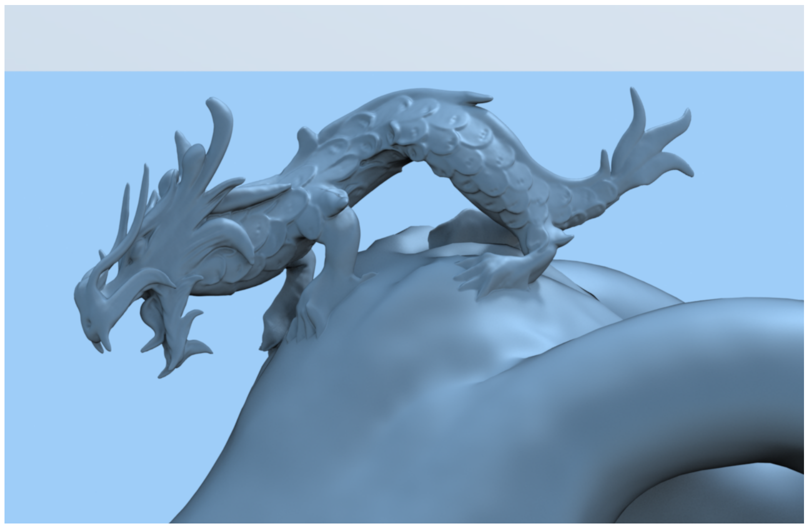
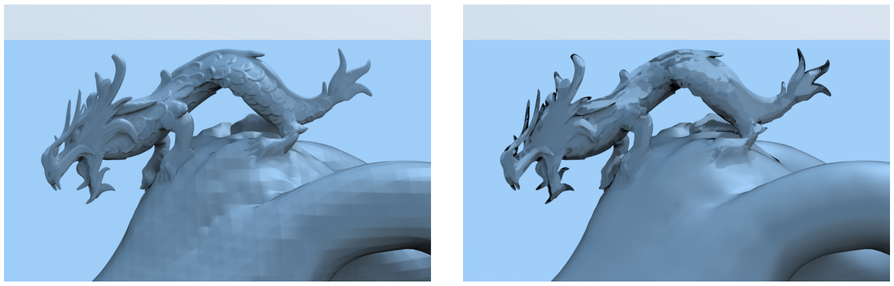


Figure 8.14: On-the-fly CSG union using our *multi-volume* rendering plugin.



(a) Using the smaller resolution grid distance as delta value

(b) Using the larger resolution grid distance as delta value

Figure 8.15: Rendering artifacts due to using linear interpolation when calculating the surface normals on a *multi volume*.

2. The triharmonic spline [5]:
   $\varphi(r) = r^3$
3. The thin-plate spline [87]:
   $\varphi(r) = r^2 \log(r)$
4. The sum of the triharmonic spline and the thin-plate spline:
   $\varphi(r) = r^3 + r^2 \log(r)$
5. The biharmonic spline [5]:
   $\varphi(r) = r$
6. The Gaussian basis [87]:
   $\varphi(r) = e^{-r^2}$

The first basis function has two user controlled parameters δ and τ. These two parameters control the amount of first and third order smoothness respectively, and the balance between them controls the amount of second order

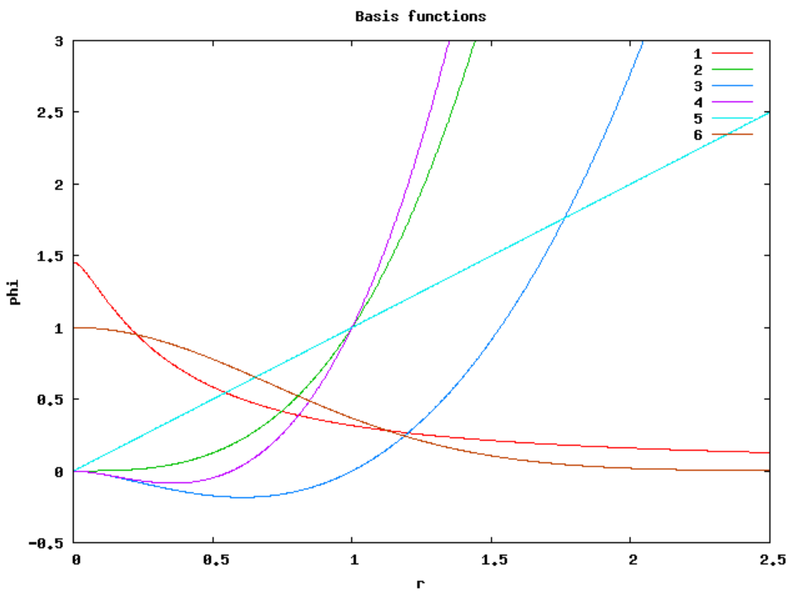
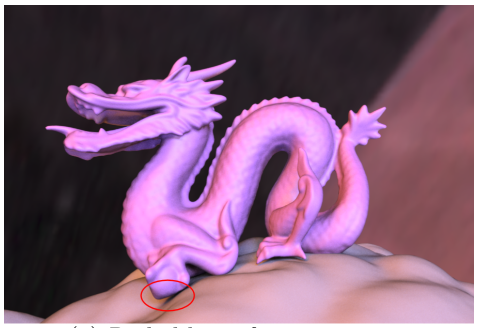
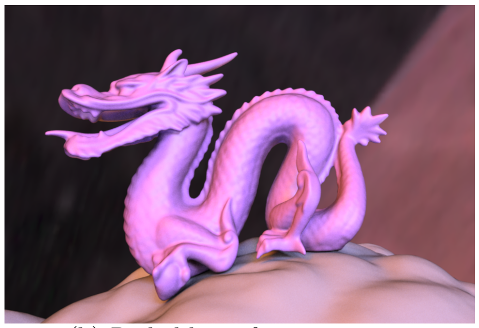


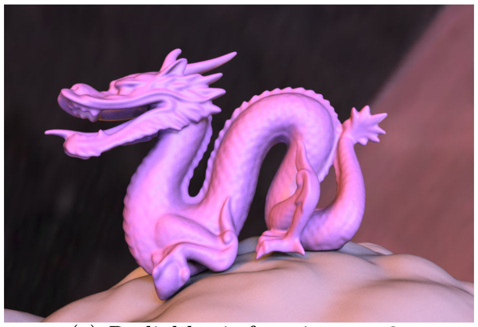
Figure 8.16: Plot of the different basis function tested. Note, that in order to enhance the shape of the first basis function, it has been scaled by a factor of 100.

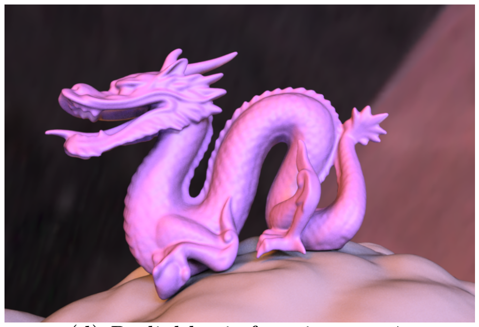
smoothness [20]. For simplicity, we only show test results for one particular configuration, namely $\tau = 0.02$ and $\delta = 5.0$, as these are the parameters with which we have achieved the best results. Figure 8.16 shows a plot of the different basis functions. Note that the values of the first basis function has been scaled by a factor of 100 to enhance the shape of the curve (without this scaling, the curve for that function would appear as a horizontal line). Using the seven different basis functions, we have generated two different mapping examples which we will use to perform a visual comparison of the results produced using the different basis functions. Figure 8.17(a) to figure 8.17(f) shows a dragon mapped onto the side of the Stanford bunny. Apart from figure 8.17(a) and figure 8.17(f), there is little visual difference in the results. Figure 8.17(a) and figure 8.17(f) however appears to have been deformed a little less than actually desired (See *e.g.* the front foot). This is particularly a problem in figure 8.17(f), where the dragon appears not to have been deformed at all. Moreover, the dragon in figure 8.17(f) is disfigured by some dents and holes, including a very obvious hole on the back of its neck and a dent on the back.

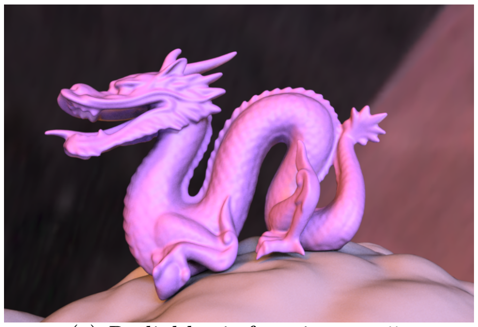
Figure 8.18(a) to figure 8.18(f) show similar examples where another dragon is mapped onto a wavy surface. Again, there is little visual difference between figures 8.18(b) to 8.18(d). There are however some differences in how much it is stretched in the height, which is most noticeable at the back of the dragon. In this example, figure 8.18(e) stands out in that it shows the same quality of the mapping near the surface as seen in figures 8.18(b) to 8.18(d), but without the pronounced stretching of the dragons back. Again, the results produced by the first and last basis function (figure 8.18(a) and figure 8.18(f)) perform significantly worse than the remaining functions. This is particularly visible


(a) Radial basis function no. 1


(b) Radial basis function no. 2


(c) Radial basis function no. 3


(d) Radial basis function no. 4


(e) Radial basis function no. 5

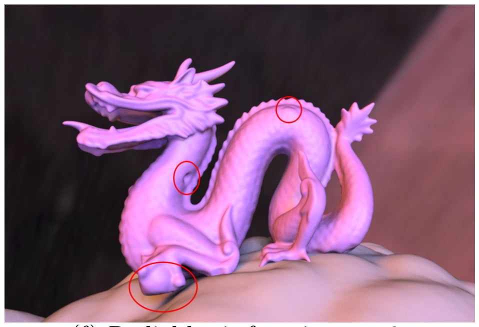
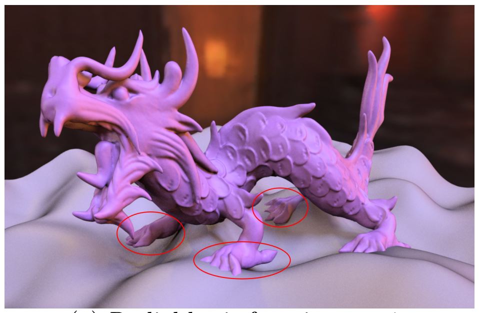

(f) Radial basis function no. 6

Figure 8.17: Mapping a dragon onto the side of the Stanford bunny using the six different radial basis functions. The red circles highlight mapping artifacts with two of the functions.

near the surface, where the dragons feet are not properly warped to follow the base surface. Furthermore, figure 8.18(f) exhibits similar artifacts to those visible in figure 8.17(f), most noticeable on the dragons back right foot. These artifacts along with the overall poor performance of the first and last radial basis functions are a consequence of the rather local nature of these two functions. Both functions have their maximum value at $r = 0$ and tend quickly towards zero as r increases, which means that only nearby particles will have an influence on the mapping. If the particle spacing is to large, which is clearly the case in these examples, then these two functions will not produce satisfactory results.

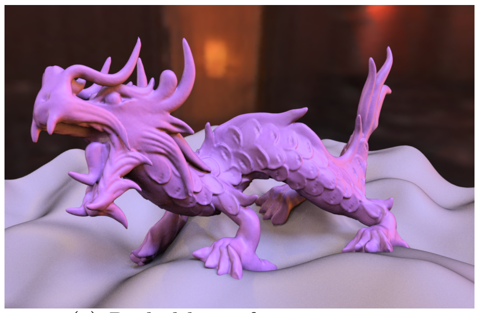
Table 8.1 shows the time taken for performing the mappings for the two test cases. The numbers in table 8.1 clearly show that the fifth basis function performs significantly faster then most of the other functions. The only function


(a) Radial basis function no. 1

(b) Radial basis function no. 2

(c) Radial basis function no. 3

(d) Radial basis function no. 4


(e) Radial basis function no. 5

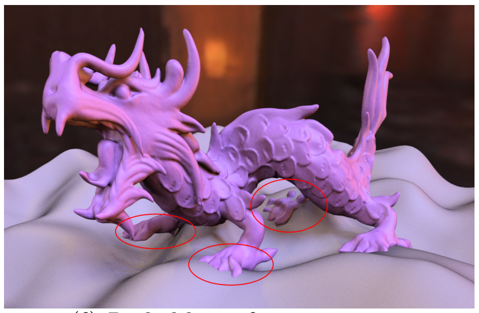

(f) Radial basis function no. 6

Figure 8.18: Mapping a dragon onto a wavy surface using the six different radial basis functions. The red circles highlight mapping artifacts with two of the functions.

Basis function	1	2	3	4	5	6
Test case 1	25219ms	6734ms	21719ms	21390ms	5907ms	36766ms
Test case 2	709818ms	183016ms	598433ms	618814ms	160911ms	1595677ms

Table 8.1: Mapping time for the two radial basis function test cases.

remotely close to the fifth basis function, performance wise, is the second basis function. As the basis functions are evaluated once per particle per grid point, it comes as no surprise that the complexity of the chosen basis function has a significant impact on the performance of the entire system. From the above we observe that the fifth basis function produces visual results as good as, and sometimes even better then the other functions. As it furthermore performs significantly faster than most of the other functions, it seems only logical to choose that function as the preferred basis function.

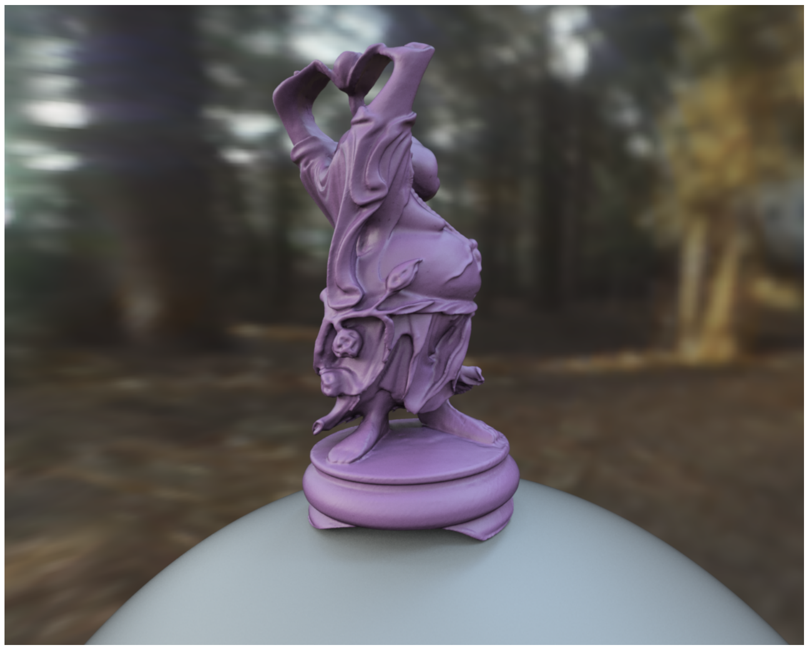


Figure 8.19: Adding custom transformations to the patch set. Prior to performing the mapping, a 180° twist transformation has been applied to the patch.

Benefits of using several patch levels

As we have already hinted in section 8.2, we often prefer to use more than two levels of particles for our parameterizations. At first, it may appear that only two levels are required: One level at the surface, and one level at the desired offset distance. There are however two important reasons for introducing a number of intermediate levels. First of all, having more than two levels adds flexibility to the parameterization. The spline based parameterization for example would not be of any use if only the start and end levels were used. Similarly, by using multiple levels of particles, we can perform various other transformations of the patch prior to using it for our geometrical texture mapping. This is demonstrated in figure 8.19, where a 180° *twist* transformation has been applied to the patch before using it to map the Buddha statue onto the base surface. Using only the top and bottom levels, the mapping becomes a linear interpolation between the top and bottom layer unable to produce the desired result as shown in figure 8.20(b).

The second reason for using more than two levels applies only to the implicit mapping. As the implicit mapping is based on a radial basis function interpolation approximating a smooth function, it is important that we have enough samples of the function we are trying to approximate. If we do not provide enough samples of the function, then the approximation to the function becomes unpredictable. This is demonstrated in figure 8.21, which shows a mapping of the dragon onto a bumpy surface. Using 6 levels of particles, the warped dragon has the desired shape, where in contrast, when only 2 levels are used, the warped dragon exhibits an undesirable stretching.

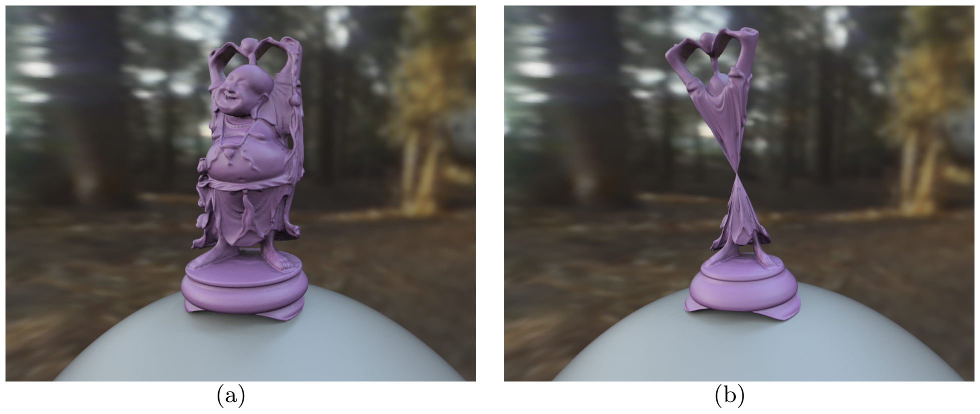


Figure 8.20: (a)The Buddha statue mapped onto a sphere using the standard reduced distortion parameterization as in figure 8.19, but without the twist transformation applied. (b)The result of performing the same mapping as in figure 8.19, but using only two particle levels.

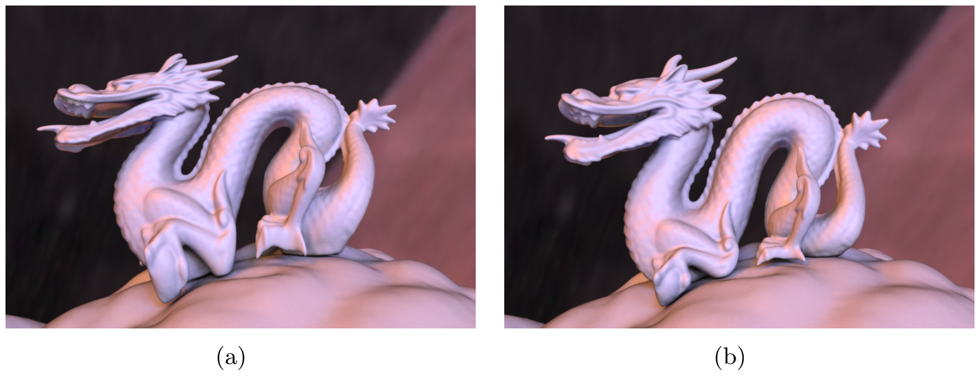


Figure 8.21: Comparison between using only two layers of particles (a) and using more layers (b). A total of 6 layers of particles were used to generate the mapping in (b); both (a) and (b) were generated using the implicit mapping. Notice the uneven stretching of the dragon in (a). In contrast, the dragon in (b) retains a shape faithful to its original shape.

Chapter 9

Shell Map Hybrid

9.1 Introduction

In the previous chapter, we presented a method, based on radial basis function interpolation, for mapping implicit geometric textures onto an implicitly defined base surface. Although the presented method is stable and produces high quality results, it fails to impress when it comes to execution time. The execution time for the actual mapping is $\mathrm{O}(n_g \times n_p)$, where n_p is the number of particles and n_g is the number of grid points in the embedding volume. This quickly becomes a problem when using high resolution geometric textures, leading to a large number of grid points in the warped texture, or when the base geometry is very detailed, in which case a large number of particles are needed to sample the patch space. This led us to the development of a new mapping inspired by the shell map algorithm [67], using a combination of the approach presented in chapter 8 and the shell map method to obtain a faster implicit-on-implicit mapping.

9.2 Shell Map Hybrid

The idea is to generate two levels of particles for each patch, one at the surface, and one at the maximum desired offset height. Following the approach of [67] using the first particle level as the base surface and the second level as the offset surface, we can *fill* the space between the two levels with tetrahedrons. This allows us to perform the patch-space to texture-space mapping using a fast barycentric interpolation rather than the significantly slower radial basis function interpolation.

In order to facilitate the construction of the tetrahedrons, we need to enforce a single additional constraint on the particle sets: Each level must contain the same number of particles in the same topological configuration (number of particles in the u direction and number of particles in the v direction). Of the different particle generation methods presented in chapter 8, only the reduced distortion parameterization needs to be altered to comply with this new constraint: Once the particles on the surface have been generated, we lock the number of points for consecutive particle levels to that of the first level. This has the obvious drawback that the sampling rate will decrease away from the surface in convex regions. A more correct approach is to determine the largest number of particles needed in the u and v direction for all of the levels needed, lock the patch sizes to these numbers, and regenerate all levels. This, on the other hand, results in a higher sampling rate than required in concave regions. Although this results in more particles, it is still to be preferred over a potential under-sampling of the patch space.

If we form a triangulation of the first particle level, and an identical (index wise) triangulation of the second particle level, each pair of corresponding triangles constitutes the caps of a prism, and the union of all the prisms defined by each triangle pair completely covers the patch space without overlapping. As the sides of the prisms will not in general be planar, the prisms will not generally be convex either. This rules out using the general barycentric interpolation techniques [84] on the prisms. Therefore, following [67], we split each prism into three tetrahedrons, that are easily interpolated using barycentric coordinates. To avoid interpolation artifacts between adjacent prisms, the tetrahedrons need to be created in a way that guarantees a proper alignment, that is, if a face is shared by two prisms, creating the tetrahedrons needs to be done in a way that ensures that the face is split along the same diagonal for both prisms.

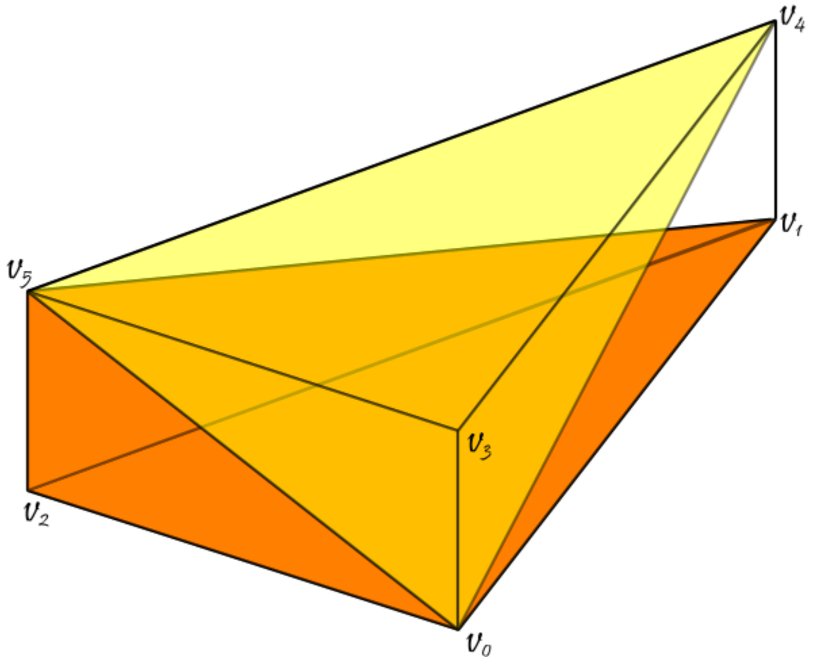


Figure 9.1: Relationship between prism vertices and the created tetrahedrons.

For a prism $\mathcal{P}$ with particle indices v_0, v_1 and v_2 at the lower particle level, and particle indices v_3, v_4 and v_5 at the upper level, see Figure 9.1, we rearrange the order of the indices such that $v_0 < v_1 < v_2$. The remaining indices are reordered as well, so that v_3 remains above v_0, v_4 remains above v_1 and v_5 remains above $v2$. If for example $v_0 = 12, v_1 = 5, v_2 = 14, v_3 = 34, v_4 = 27$

and $v_5 = 36$, then we reorder the order of the vertices such that $v_0 = 5, v_1 = 12, v_2 = 14, v_3 = 27, v_4 = 34$ and $v_5 = 36$. Note, that due to the way we create the particles and prisms, we always have that $v_3 = v_0 + c$, $v_4 = v_1 + c$ and $v_5 = v_2 + c$, where c is the number of particles per level. With the reordered indices, we create the following three tetrahedrons [34]: $\mathcal{T}(v_0, v_1, v_2, v_5)$, $\mathcal{T}(v_0, v_1, v_4, v_5)$ and $\mathcal{T}(v_0, v_3, v_4, v_5)$. The combination of this ordering and a consistent generation of the tetrahedrons guarantees that faces shared by two prisms will always be split along the same diagonal when creating tetrahedrons for those two prisms. This follows from the fact that the diagonal is always created between the vertex with the highest index and the vertex with the lowest index.

Once we have a set of tetrahedrons, we are ready to define our mapping from patch space to texture space (or vice versa). Since the vertices of the tetrahedrons correspond to the patch particles, we have a texture space coordinate associated with each corner of a given tetrahedron. Assuming we know that a given point, p, lies inside the tetrahedron $\mathcal{T}$, we can find the texture space coordinate of p using barycentric interpolation of the texture space coordinates assigned to each of $\mathcal{T}$'s corners.

The only problem remaining is how to quickly locate the tetrahedron containing a given point. Using the brute force method of testing against all tetrahedrons in turn leads to an $\mathrm{O}(n_g \times n_p)$ execution time similar to the radial basis function interpolation based approach we are trying to improve upon. To do this, we need a way to quickly zero in on the correct tetrahedron without having to check all tetrahedrons. As this test is in fact not much different from the ray-primitive intersection tests performed in ray-tracing, it is natural to look at some of the data structures used to speed up the ray-primitive intersection testing. We have chosen to use a bounding volume hierarchy with axis aligned bounding boxes [76], as it provides a good balance between ease of implementation, construction cost and lookup cost. Constructing a bounding volume hierarchy can be done in time $\mathrm{O}(n \log n)$ in the number of elements, which in our case is the number of tetrahedrons, or equivalently the number of particles. Creating the tetrahedrons takes linear time, which leads to the total setup cost being $\mathrm{O}(n_p \log n_p)$. Although the lookup cost remains linear in the number of tetrahedrons (and thus the number of particles), this is the *worst case* scenario, that is, when all tetrahedrons overlap. If however, the tetrahedrons are well distributed with little or no overlap, then the *expected* lookup time is $\mathrm{O}(\log n_p)$. Fortunately, the way we generate the tetrahedrons guarantees a reasonably uniform distribution of non-overlapping tetrahedrons, which in turn means that with this optimization, the *expected* execution time is reduced to $\mathrm{O}(n_g \times \log n_p)$. To create the bounding volume hierarchy, we only need the tetrahedrons and their bounding boxes. Hence, creating the bounding volume hierarchy is done using $\mathrm{O}(n_p)$ memory. Similarly, as the particles and the bounding volume hierarchy contains all the information we need for the mapping, this is performed using $\mathrm{O}(n_p)$ memory.

While the original shell map method [67] used a single offset surface, and thereby always had only a single layer of prisms, we have, for several reasons, preferred using several layers of particles for the previous two methods presented. Similarly, with this method, we allow additional layers of prisms to be generated if more than two layers of particles exists. For a patch set with n particle levels, we construct $n-1$ levels of prisms. Although introducing more than one layer of prisms does not in general improve the quality of the mapping, there are two other important benefits:

1. By introducing more levels, the generated tetrahedrons become smaller and more regular shaped, yielding a smaller, more quadratic shaped bounding box. This in turn means that fewer bounding boxes will overlap, reducing the number of actual point-in-tetrahedron tests that need to be performed, thereby speeding up the lookup operation. Although more tetrahedrons are created, the use of a bounding volume hierarchy ensures that this has a smaller influence on performance than the reduction in point-in-tetrahedron tests.

2. More layers mean more flexibility. One of the key strengths of the methods presented so far is the flexibility with respect to the distribution of the patch particles, however without introducing support for multiple layers, particle distribution methods such as the spline based method would not be supported.

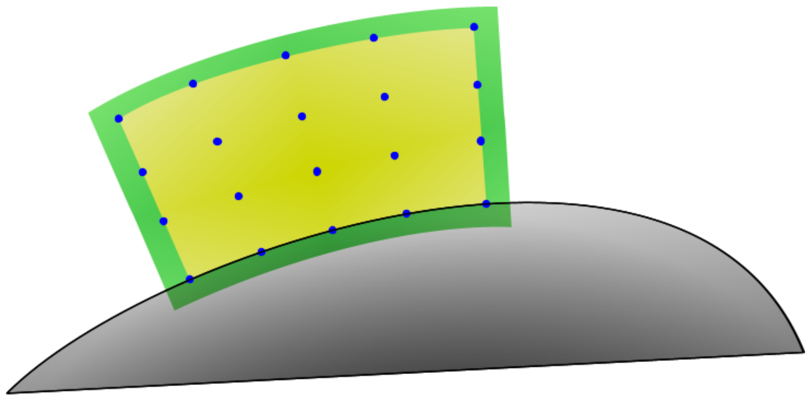


Figure 9.2: To avoid artifacts along the edge of the mapped region (yellow area), we need to extend the mapping to cover a larger region (green area).

A final detail regarding this method is that since we are using implicit geometry as our texture, we need to take extra care to re-sample the warped texture volume, not only in the area of interest, but also in the region surrounding the area of interest, as illustrated in Figure 9.2. This is necessary, as we would otherwise not have the required samples needed for reconstructing the geometry at the border of the area of interest, leading to artifacts such as those visible in figure 9.3. This problem is not unique for this method, however due to the nature of the radial basis function interpolation, which interpolates to perfectly valid values even outside the mapping region, the problem is automatically handled when using the radial basis function based mapping. The problem with the Shell Map inspired mapping is that once we have a sample point outside the region defined by the particles there will be no tetrahedron containing that point. The result is an undefined texture coordinate for that point, which we map to a constant ϕ value of γ, that is, we force the points to be outside the implicit surface and outside the narrow band. We fix this problem by adding an extra layer of particles below the surface. Particles in this layer are assigned a negative w coordinate. Additionally, an extra layer is added *above* the patch set with a w value higher than 1.0 assigned to the particles. Similar *borders* are added in the u and v directions.

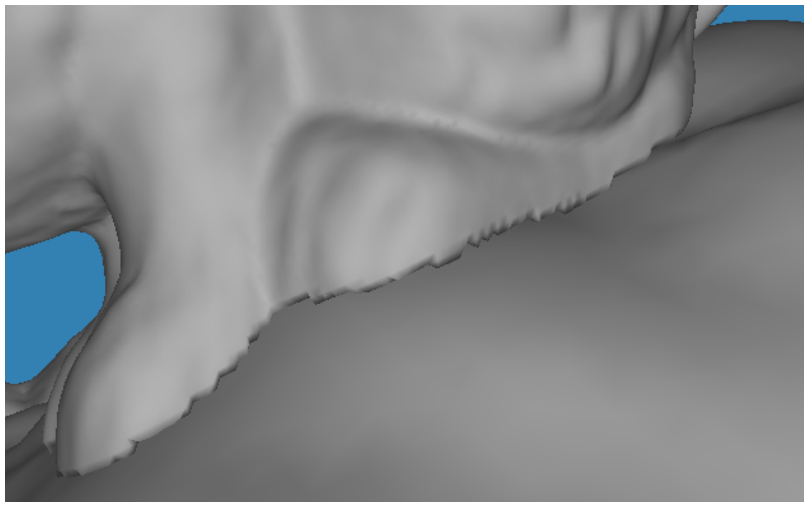


Figure 9.3: Aliasing artifacts along the intersection of the base geometry and the texture geometry as a result of resampling the texture geometry only within the actual area of interest.

9.3 Results and Discussion

In this section, we will present some results produced using the Shell Map Hybrid method, and we will evaluate them against results produced by the two previous methods. Our primary test case will be the mapping of the Stanford bunny onto another copy of the same bunny, and the mapping of a dragon onto the nose of a larger dragon depicted in Figure 9.4.

As with the semi-implicit mapping, the Shell Map hybrid mapping is based on linear interpolation, which means that it is only C^0, with a (possible) discontinuity in the derivative along the sides of each of the tetrahedrons. Whether or not these discontinuities are visible is highly dependent on the magnitude of the change in the derivative. The larger the change in the derivative, the more visible. Obviously, the kind of artifacts resulting from this are unwanted. Fortunately there is a very simple approach to reducing these artifacts: This issue is not much different from trying to represent a smooth curved surface using triangles: The more/smaller triangles you use to approximate the surface, the smoother it will appear. This same approach works well in our case as well; by increasing the number of tetrahedrons/decreasing the size of the tetrahedrons, we effectively decrease the magnitude of the change in the derivatives along the

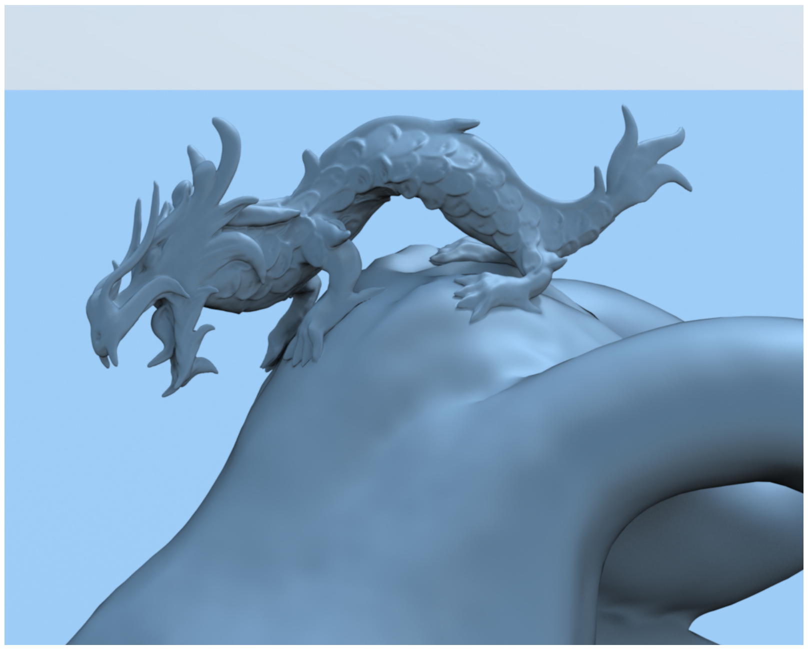
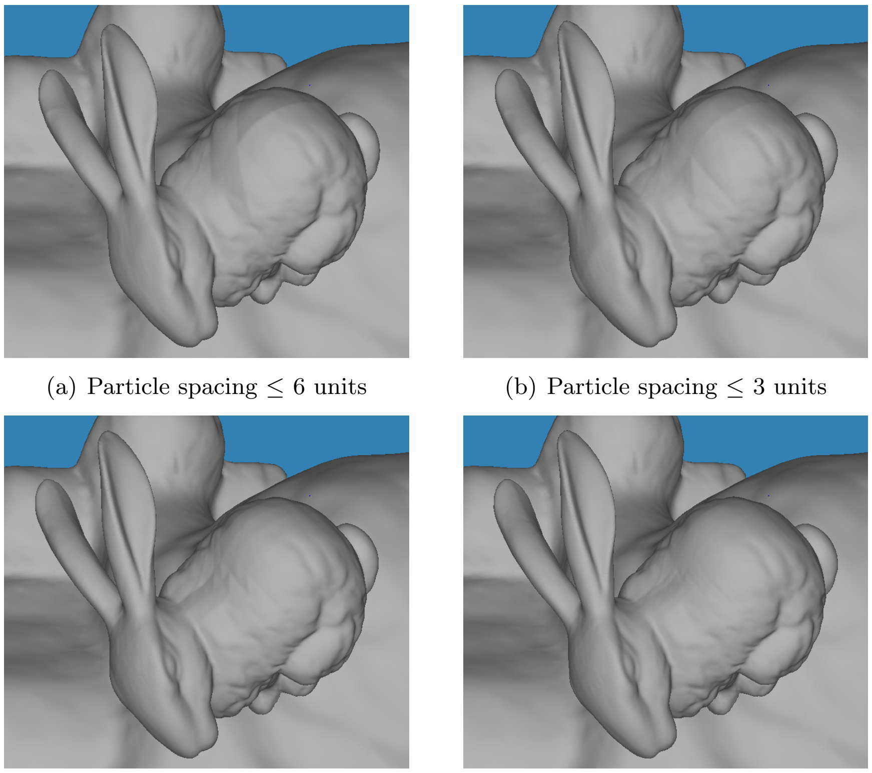


Figure 9.4: Baby dragon mapped onto nose of mother dragon using the Shell Map Hybrid method.

sides of the tetrahedrons, thereby reducing the visual effect of the discontinuities in the derivatives. The only remaining questions are how small the largest distance between the particles should be in order to achieve the desired results, and what influence this increased particle count has on the performance of the mapping. Figure 9.5 shows four examples of the bunny mapped onto another copy of the bunny. In these four images, the particle spacing in the u and v directions were kept below 6 units, 3 units, 1.5 units and 0.75 units in terms of the grid spacing on the base volume. The particle spacing in the w direction was chosen to roughly match the u and v spacing to get as square bounding boxes for the prisms and tetrahedrons as possible. Table 9.1 lists the total number of particles used for generating each image, the time spent on setting up acceleration data structures etc., the time spent on the actual mapping, and the total time spent on setup and mapping. In all four cases, approximately 2.8 point-in-tetrahedron tests were performed on average per grid point in the warped texture volume, a volume of size $189 \times 193 \times 203$. With a low particle density, linear artifacts are visible in the warped texture. As expected, these artifacts become less and less visible as we increase the particle density. Once the particles are restricted to be no further than 0.75 units away from each other, all artifacts are as good as gone. As can be seen from table 9.1, performing a mapping based on a maximum particle spacing of 6 is, in this example, only approximately 34% faster than the same mapping based on a maximum particle spacing of 0.75. Thus, the performance penalty is rather low compared to the gain in quality. All our other sample images created using this method were created using a maximum particle spacing between 1.0 and 0.75.



(a) Particle spacing ≤ 6 units

(b) Particle spacing ≤ 3 units

(c) Particle spacing ≤ 1.5 units

(d) Particle spacing ≤ 0.75 units

Figure 9.5: The influence of the particle density on the result of the Shell Map Hybrid method. With a low particle density, linear interpolation artifacts are visible, most notably on the back of the bunny. With a maximum particle distance below 1.5, only a few artifacts are noticeable, and once the particle distance is kept below 0.75, there are virtually no visible artifacts left.

Particle spacing	Particle count	Setup time	Mapping time	Total time
6	480	47ms	9750ms	9797ms
3	2926	78ms	10797ms	10875ms
1.5	13194	485ms	11671ms	12156ms
0.75	52896	2266ms	12641ms	14907ms

Table 9.1: Patch size and timings for mapping with different particle densities.

A similar test was performed in order to test the performance impact of using several levels of particles in order to get more regular shaped tetrahedrons. Using the same mapping scenario as above, the bunny was mapped onto the other bunny, using two different patch sets. Both patch sets cover the same area and have a maximum particle distance in the u and v directions below 0.75. The only difference is the number of particle levels used, which for the first case is 29 and for the second case only 6 levels. Using 29 levels, a total of 52896 particles were used. Creating the bounding volume hierarchy took 2.2s, while performing the mapping/warping took 12.6s. The total time spent on the mapping was then 14.9s, using approximately 2.8 point-in-tetrahedron tests per grid point. With 6 particle levels 10944 particles were used. This reduced the time spent on creating the bounding volume hierarchy to 0.4s, however due to an increase in the point-in-tetrahedron tests up to approximately 7.1 tests per grid point, the mapping time increased to 26704 giving a total time of 27.1s. Thus, as expected, by increasing the number of levels, the bounding volumes used in the bounding volume hierarchy become more tight fitting giving a much more effective tree and consequently a significantly more efficient mapping.

As with the radial basis interpolation based approach, comparing the Shell Map hybrid method to the semi-implicit method is somewhat misleading due to the fundamentally different texture representation. The two different geometric representations have different properties, resulting in a number of differences in the final result. Most notably, the semi-implicit method does not allow the warped texture to be merged with the base volume into a single piece of geometry. Due to the direct representation of the surface, however, the complexity of the algorithm is significantly reduced (as we showed in section 8.4, the complexity is $O(n_v)$, where n_v is the number of vertices in the mesh). Performing the same mapping as displayed in Figure 9.5, using the particle set with a maximum distance of 0.75 and a mesh with 35947 vertices, the semi-implicit mapping needs only 62ms as opposed to the 14907ms used by the shell map hybrid method. A more fair comparison between this method and the semi-implicit method would be to compare the semi-implicit method to a semi-implicit version of the Shell Map hybrid, that is, a version using the tetrahedron interpolation to map explicit geometry rather than implicit geometry, which is basically the original Shell Map using our parameterizations. As this mapping would have to be from texture space to patch space, we can use the regularity of the particle positions in texture space just as we did with the semi-implicit mapping, leading to a $O(n_v)$ algorithm[1]. This mapping should be just as fast as the semi-implicit mapping (it is essentially Shell Mapping), however we have not had the time to implement this and are therefore unable to verify it.

Comparing the Shell Map hybrid method to the radial basis function interpolating method is quite reasonable, as both methods use implicit geometry as texture. Clearly, the radial basis function interpolation based mapping produces higher quality results when compared to the barycentric interpolation of the Shell Map hybrid, but the latter has other advantages. Most important is the performance. Clearly, the expected running time of $O(n_p \log n_p)$ of this method is superior to the $O(n_p \times n_p)$ running time for the radial basis interpolation method. Table 9.2 shows a performance comparison between the two different methods. As expected, these figures clearly show, that the Shell Map hybrid outperforms the radial basis function method. In both cases, the Shell Map hybrid is approximately a factor of 7 faster than the other method.

Apart from being significantly faster, there are a few other advantages with the Shell Map hybrid. First of all, although it requires significantly more particles to produce good results, it still has a (potentially) lower memory footprint

[1]Note, that the bounding volume hierarchy or a similar data structure is not needed for this mapping.

Scene	SMH Particles	SMH Exec. Time	RBF Particles	RBF Exec. Time	Speedup
Bunny Scene (Fig. 9.5)	52896	14907ms	361	102828ms	6.90
Dragon Scene (Fig. 9.4)	3822	55609ms	248	406156ms	7.30

Table 9.2: Performance comparison between the Shell Map hybrid mapping (SMH) and the radial basis function interpolation based mapping (RBF). Note, that the number of particles used in the examples depends on the roughness of the surface, the size (area) of the patch and the height of the patch. The number of particles is chosen such that the two different mappings produce results of roughly identical visual quality.

than the radial basis function method. This is due to the fact that the latter method uses $O(n^2)$ memory as opposed to the $O(n)$ memory usage of the Shell Map hybrid. Secondly, this mapping is bijective. This means that we can implement a ray tracer performing the mapping while tracing the rays, similar to the renderer described in the Shell Map paper [67]. The advantage of this is that if the same texture is used several times, we would only need to keep the original texture in memory, rather than say 20 warped copies. Unfortunately, with this approach, we will no longer be able to perform the smooth blending between the base surface and the textures, and as we would have to perform the mapping over and over, the performance is likely to suffer.

Chapter 10

Geometric Texture Animation

One of the most important properties of level sets and implicit surfaces is the ease with which they can dynamically deform. A great part of this strength stems from the fact that changes in topology requires no special treatment, as the implicit representation automatically resolves this. Similarly, artifacts such as surface self intersections that would, if not handled correctly, lead to a physically impossible surface are also resolved automatically by the implicit representation. With this in mind, it seems natural to try to extend our geometrical texture mapping to *dynamic* implicit surfaces. In this chapter, we present our preliminary work on geometrical texture mapping on *dynamic* implicit surfaces.

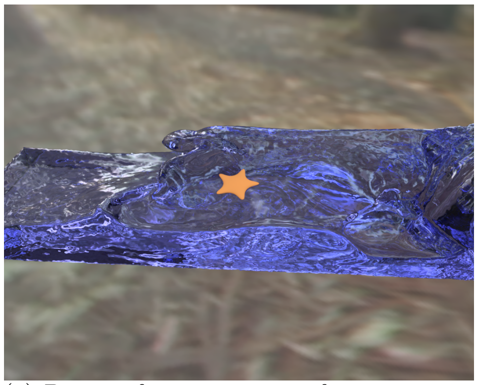
We divide our work on animations into three categories: Animating textures, base surface animations and key frame animation of the patch.

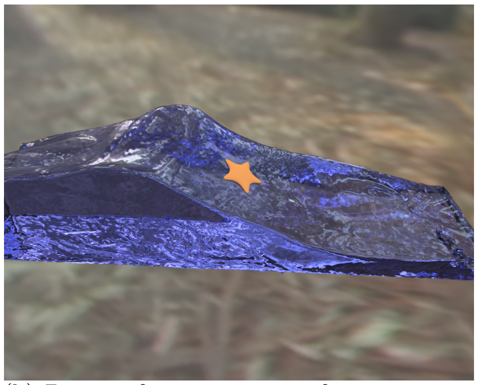
Animating textures Using animated textures is by far the simplest case of the three. As the base geometry and texture geometry are independent, the only thing tying the two together is the particle based parameterization. As such, using an animated texture simply corresponds to mapping different textures (different frames of the texture animation) onto the same base using the same particle set.

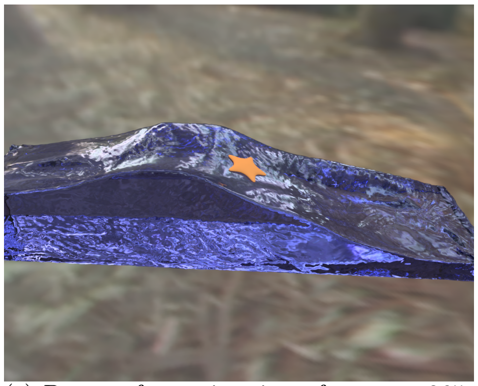
Base surface animation The term *base surface animation* covers the case where the base surface is deforming, and where the applied texture *follows* the deformation of the surface. One of the challenges in this case lies in defining the desired effect: How do we expect the geometric texture to respond to the deformation of the base surface? We have chosen to let the texture follow the movement of the surface in the direction of the surface normal. With the signed distance function representation, this is done using the following approach: First, the patch corners for a desired parameterization is defined on the initial surface. Then, during the animation of the base surface, and after each iteration, these four particles defining the patch are moved along with the surface, thereby creating the base for a new parameterization for the current surface. Moving the particles along with the surface is achieved by projecting the particles onto the closest point on the animated surface. With a signed distance function, this is done using the following formula:

$$\tilde{\mathbf{p}} = \mathbf{p} - \frac{\nabla\phi(\mathbf{p})}{|\nabla\phi(\mathbf{p})|}\phi(\mathbf{p}).$$

We then use the four translated particles to automatically create a unique parameterization for each frame in the animation of the base surface. An example of this is depicted in figure 10.1.


(a) Base surface animations, frame nr. 210


(b) Base surface animations, frame nr. 260


(c) Base surface animations, frame nr. 265

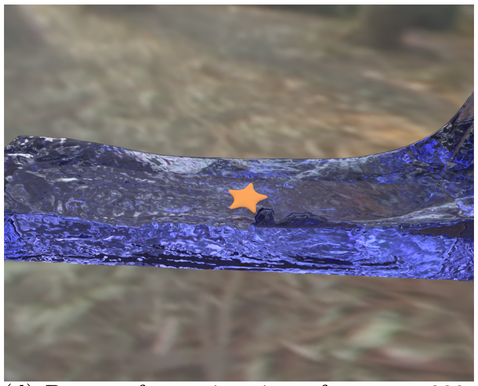

(d) Base surface animations, frame nr. 290

Figure 10.1: Example of a geometric texture mapped onto an animated base surface.

Key frame patch animation The third type of animation we have investigated is key frame based animation of the patch on a static base surface. Our current approach supports scaling and translation of the patch using the following approach: Assuming we are given two distinct patch definitions, that is, the corner particles defining the two patches, we wish to perform a smooth transition from the first patch set to the second. Starting with the corner particles $p_{s1}, \ldots, p_{s_n}$ for the source patch and the corresponding particles $p_{d1}, \ldots, p_{dn}$ for the destination patch, we create the (approximated) geodesic curves (using the same approach as in section 8.2) $c_1, \ldots, c_n$ between the corresponding points of the two patch sets. Thus, $c_i(t)$ denotes the (approximated) geodesic curve between the two points p_{si} and p_{di}, with $c_i(0) = p_{si}$ and $c_i(1) = p_{di}$. We can then calculate the position of the corner particles for the animated patch at frame f_c of f_{max} as the position along the individual curves at the time $t = \frac{f_c}{f_{max}}$, as depicted in figure 10.2.

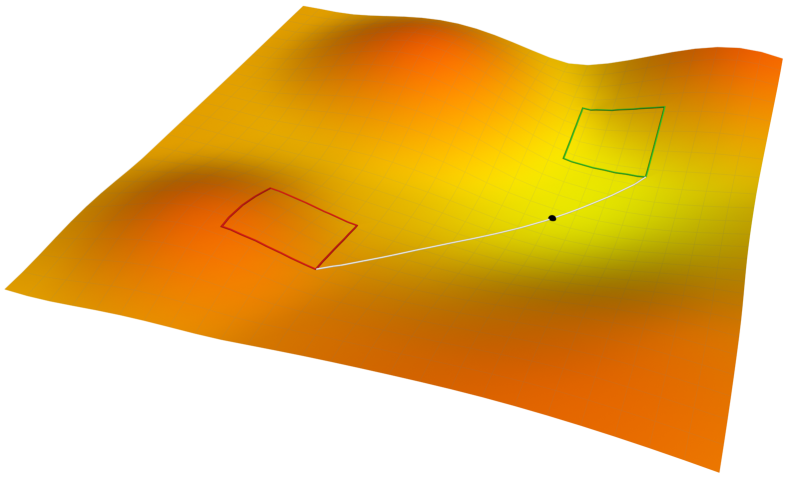


Figure 10.2: Interpolation of a single patch corner between two patches. The white curve represents the geodesic between the two corresponding corner points of the source patch (green square) and the destination patch (red patch). The black dot represents the interpolated position of the corner point at $t = \frac{1}{3}$.

Bringing it all together While the three animation methods presented above may provide a set of useful tools for some applications, they are still quite limited with respect to what can be achieved. It is not until we start combining the different methods that we start to see the full potential in these methods. Consider for example a snake zig-zagging across a rough surface. This can be animated using a key frame animation of the snake's path combined with a texture animation of the snake's zig-zag movement. Or alternatively, a water droplet, running down a balloon being inflated. This latter example would require a combination of the base animation method for animating the balloon, a key frame animation of the droplet's path down the balloon, and perhaps even a texture animation of the droplet itself to add a bit of extra realism. The problem with this example is that combining the key frame animation with a base animation introduces a new challenge: How do we maintain the geodesics curves on the base surface when it is being animated? For the key frame animation, we require a source and target patch that we can interpolate between. When the base surface is animated, we want to specify the source patch on the first frame of the base animation, and the target patch on the last frame to have full control over the final position of the patch. We cannot generally expect to be able to specify the destination patch on the initial base surface and then have it end up at the desired position on the final base surface. Instead, we need to specify the patch on the final base surface, and then run the animation backwards in order to calculate all the intermediate positions for the destination patch. Thus, to combine a key frame animation with a base surface animation we use the following algorithm:

1. Load base surface and source patch corners
2. Animate base surface one iteration at a time. For each intermediate step:
   (a) Update position of source patch corners
   (b) Save current patch and volume
3. Load destination patch corners
4. Step backwards through animation. For each intermediate volume:
   (a) Load current volume
   (b) Update position of destination patch corners
   (c) Save current patch

Then, to generate the i'th frame, we load the volume, source and destination patch corresponding to frame i. We then use this as input to a standard key frame animation (as described above), evaluating the intermediate patch at time $i/(n_f - 1)$, where n_f is the total number of frames in the animation. The result is the corner particles defining the patch for the ith frame of the animation.

10.1 Results

We have used the above described methods to produce some *proof of concept* animations. Figure 10.1 shows four still frames from an animation using the base animation method on a fluid simulation. Figure 10.3 shows four still frames, superimposed into one image, from an animation of the dragon moving across the back of the Stanford bunny, created using the key frame animation technique (the dragon itself is not animated). Finally, figure 10.4 shows four still frames from an animation of the Stanford bunny morphing into the Utah teapot. During the animation, a starfish is moving from a user defined position on the bunny to a user defined position on the teapot. This animation is created using the combination of the key frame animation and base animation described above.

While the examples in figures 10.1, 10.3 and 10.4 show some of the potential of the presented methods, there are still a number of issues needing to be resolved before they will be truly useful in general: First of all, the particles defining the corners of the patch, or both the source and destination patch in the case of a key frame animation, need to be at the same topologically connected component of the surface. If the source and destination patch resides on two disconnected components, then there is no way we can create the

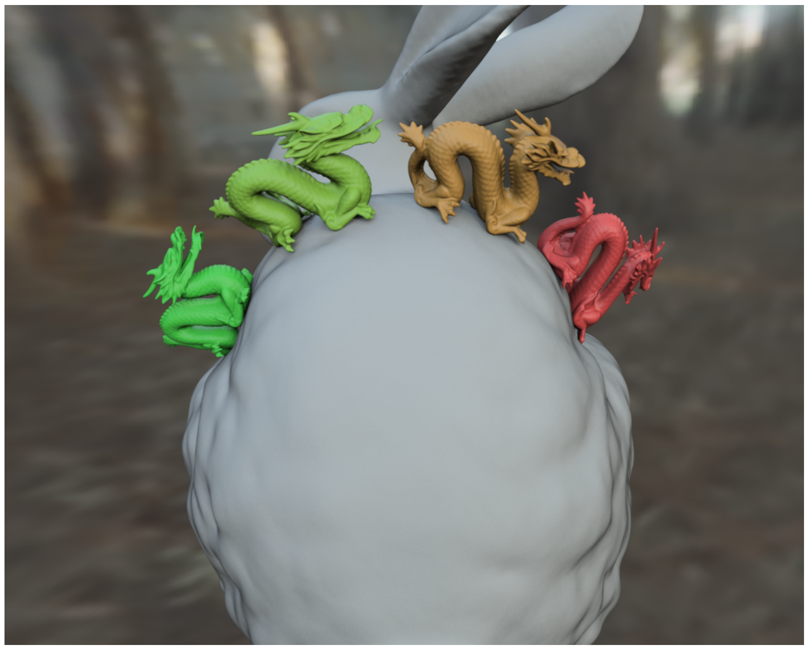
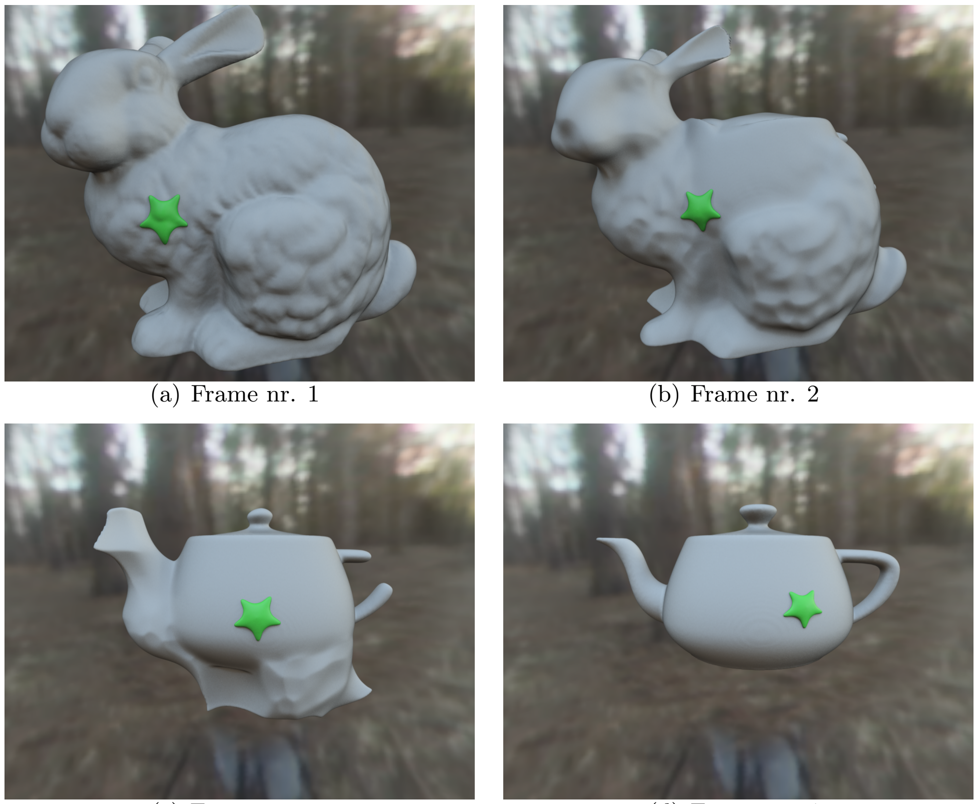


Figure 10.3: Key frame animation of a dragon moving across the back of the Stanford bunny. The green dragon on the left is the initial position of the dragon, that is the dragon at time $t = 0.0$. Then, the next dragon is at the position for time $t = 0.33$, followed by $t = 0.66$, and finally, the red dragon is the final frame, that is $t = 1.0$.

geodesics between the corresponding corner particles on the source and destination patches. Without the geodesics, we cannot perform the interpolation between the two patches, which is fundamental to that technique. Similarly for the base surface animation technique, if somehow the particles defining the patch end up at different connected components, we cannot create the required geodesics between the corner particles. This means that we cannot generate the parameterization that is crucial to our mappings. Even if the corner particles remain on the same connected component, there is still the risk that one or more of the particles, while following the movement of the surface, is moved far away from the other particles, as illustrated in figure 10.5. While this does not invalidate the patch as such, as we will still be able to generate the required geodesics, the end result is still a severely distorted patch, leading to a severely distorted texture. Another issue lies in the use of several geodesics (one per corner particle pair) for key frame animations rather than one geodesic for the entire patch: Consider for example the base surface being a cylinder with the source patch at the front of the cylinder, and the destination patch at the exact other side, each patch defined by four corner particles. In this case, two of the geodesics will run left around the cylinder while the other two will run to the right. This means that rather than moving the patch, this animation will result in the patch being stretched around the cylinder. At a certain point during the animation, the shortest path between the particles will no longer be around the



(a) Frame nr. 1

(b) Frame nr. 2

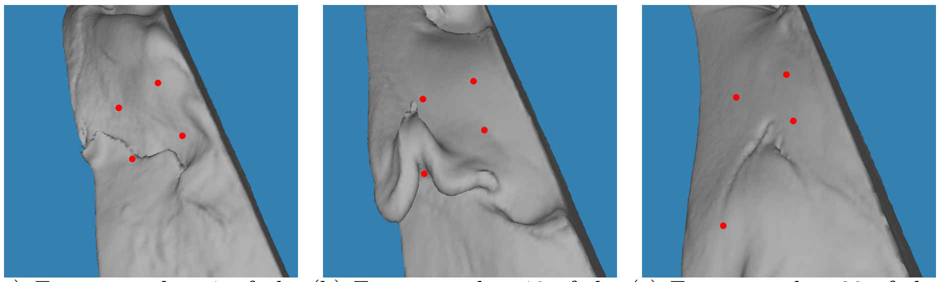
(c) Frame nr. 3

(d) Frame nr. 4

Figure 10.4: Combined base surface and key frame animation example. While the base surface is morphing from a bunny into a teapot, the green starfish is moving slowly across the morphing base surface from left to right.

front of the cylinder but around the back. At this point, the patch will jump to the back of the cylinder and start to shrink into its final shape. Finally, there are minor issues such as the lack of proper support for rotating the patch in a key frame animation.

Although the issues just described limit the usefulness of these animation techniques, we do believe that if we can resolve them there is definitely a great potential in these techniques. We are currently working on different strategies for solving all of the above mentioned issues except from the issue with the source and destination patch being on different connected components. This includes a new parameterization which is based on the exponential map [12, 71]. The exponential map is a local parameterization based on *geodesic polar coordinates*. This mapping takes geodesic curves originating from a seed point $\mathbf{p}$ to straight vectors originating from $\mathbf{p}$ in the tangent plane to the surface at $\mathbf{p}$. Thus, as opposed to the four particles currently needed, only a single seed point and a direction vector is needed for this mapping. The idea is then to use the exponential map to drive the distribution of the particles. With only a single particle and a direction vector defining the patch, we no longer have any problems with the defining particles drifting in different directions.



(a) Frame number 1 of the fluid simulation base animation

(b) Frame number 10 of the fluid simulation base animation

(c) Frame number 20 of the fluid simulation base animation

Figure 10.5: Illustration of the problem with particles drifting far away from the rest of the patch. In (a), we see the initial position of the four corner particles (red dots). (b) shows a wave on the base geometry rolling over one of the particles, thereby forcing it to move in a different direction then the other particles. (c) shows the position of the particles once the wave has passed. Note how one of the particles has moved away from the other particles.

Furthermore, with this parameterization, it will be much easier to support key frame animations including rotations of the patch. Unfortunately, we do not yet have a working implementation of an exponential map based parameterization. As for the issue with the source and destination patch residing on different connected components, we are not sure if this case makes sense anyway, as it would require the texture to jump from one connected component to another.

Chapter 11

Applications and Evaluation of Geometric Texture Mapping

In the previous chapters we have presented a framework for geometrical texture mapping of implicit surfaces. Our system uses a particle based parameterization, for which we have presented three different distribution techniques emphasizing the flexibility of our parameterization technique. Along with our parameterization, we have presented three different mapping methods based on our particle parameterization. We list some of the properties of these three mapping methods in table 11.1

	Semi-Implicit	Rbf interpolation	Shell Map Hybrid
Texture Geometry	Explicit	Implicit	Implicit
Initialization cost (time)	O(1)	O(n_p^3)	O($n_p \log n_p$) (O(1))
Setup cost (memory)	O(n_p)	O(n_p^2)	O(n_p) (O(n_p))
Running cost (time)	O(n_v)	O($n_g \times n_p$)	O($n_g \times \log n_p$) (O(n_v))
Running cost (memory)	O(n_p)	O(n_p)	O(n_p) (O(n_p))
Mapping type	Linear	High order	Linear

Table 11.1: Comparison of the three mapping methods presented in this thesis. In the above table, n_p is the number of particles, n_v is the number of mesh vertices, n_g is the number of grid points in the warped volume and n_n is the number of grid points in the narrow band of the warped volume. Values en parenthesis in the Shell Map Hybrid column represents the (not implemented) explicit version as mentioned in chapter 9.

The methods presented here are in some ways similar to those presented by Porumbescu *et al.* in their Shell Map paper [67]. There are however some fundamental differences between their work and ours. Firstly, we have chosen to use level sets to represent the base geometry, whereas they are using triangle meshes. Secondly, our method uses a local parameterization, while Shell Maps requires a full global parameterization of the entire mesh. Needles to say, these differences obviously impact the results produced by the two different methods. Most notably is the flexibility offered by the implicit representation, and the improved quality of the produced mappings. For example, with the implicit approach, we have the option of generating a single topologically connected and smooth surface from the base geometry and all the applied geometric textures

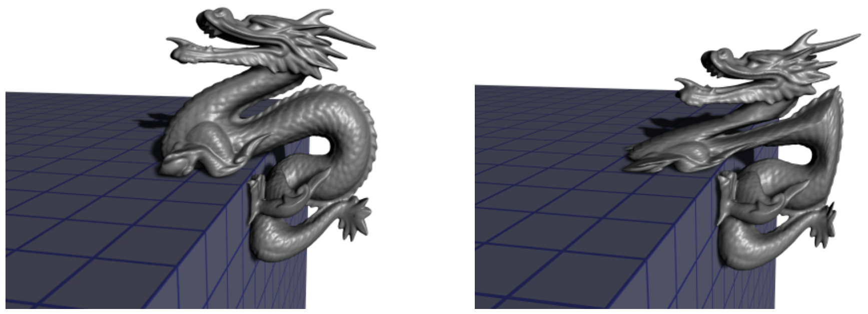


Figure 11.1: Mapping a dragon onto a sharp corner using (a) our new geometric texture mapping technique (semi-implicit mapping), and (b) using the technique of [67]. Both mappings were done in less than one second.

(see *e.g.* figure 8.13). Additionally we do not have to worry about self intersecting geometry, not even if different textures intersect each other when mapped onto the base surface. In the following, we will go into details with some of the advantages of choosing an implicit or semi-implicit approach over an explicit approach.

Figure 11.1 shows an example of mapping an explicit geometric texture (a triangle mesh) onto an object with a sharp edge. The parameterization used by the shell mapping technique [67] combines the base surface with an offset surface generated by offsetting the vertices in the base mesh a given distance in the normal direction. Consequently, discontinuities in the base mesh, such as the sharp edge in figure 11.1 will also be present in the offset surface and will consequently influence the entire parameterization. As a result, the object mapped using the shell mapping technique, figure 11.1(b), becomes severely distorted. Our reduced distortion parameterization on the other hand, guarantees a uniform distribution of the patch particles. Consequently our mapping, figure 11.1(a), produces a smooth result, even across such sharp edges. Although the distortion minimization technique presented in [91] can help reduce the distortion in the case of shell mapping, it cannot completely resolve the problem due to the linear interpolation in shell space. The only way to completely resolve this problem is to generate a smoother offset surface, which is exactly what our approach does. As for the performance of the two techniques, both mappings were done in roughly the same time, which is in less than one second.

Using the signed distance function to generate our *offset surface* presents a number of advantages. First of all, the further we move away from the base surface, the smoother the offset surface becomes. This means that the influence of high frequency details in the base geometry reduces the further we move away from the surface, yielding smoother looking results, as seen *e.g.* in figure 11.1(a). Also, most similar systems using explicit geometry representations are limited to rather small offsets. The problem is that the vertices are offset in a fixed direction. Consequently, if a vertex is about to collide with a different part of the

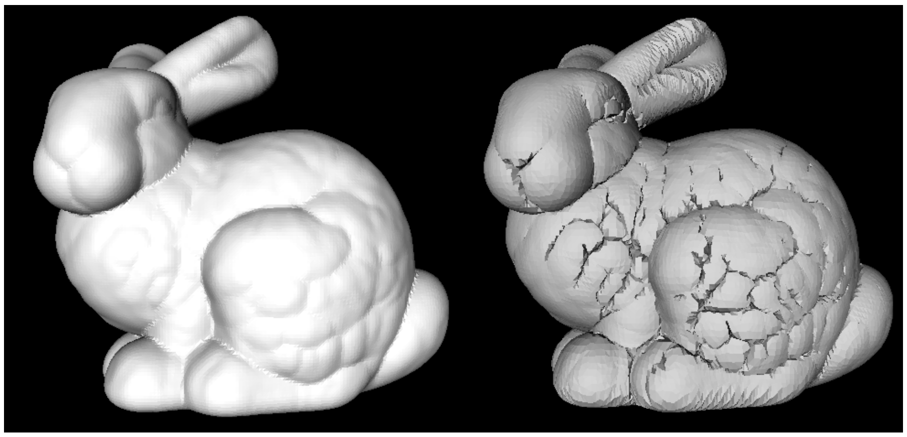


Figure 11.2: The offset on the left is a natural result of the level-set's implicit representation. On the right we see an explicit polygonal offset (note the degeneracies in concave regions) using the method proposed in Shell Maps [67].

offset surface, then that vertex cannot be moved any further. This is illustrated in figure 11.2, which shows the offset surfaces generated for the bunny model using level sets and the method used for Shell Maps. The creases we observe in the offset surface generated with the explicit approach are due to vertices being locked prematurely to avoid self intersections. This problem is not present in the offset surface generated using the level set approach. The small bunnies in figure 8.10 is an example of mappings using larger offsets (although our method is capable of handling significantly larger offsets). In addition to being significantly more stable, generating the offset surface using level set techniques is also a much simpler task than using the explicit technique. With the explicit technique, self intersections and aliasing are issues that has to be dealt with manually. In contrast, with the level set approach this is handled by solving equation (6.8) with $V_n = 1$.

As we have already mentioned in section 8.5, using an implicit surface as base- and texture-geometry allows us to perform a smoothly blended CSG union to get a single connected surface, smoothed in the regions of the intersection of the two original volumes. Performing CSG operations on explicit geometry on the other hand is very hard, and due to the finite numerical precision used, not all cases can be handled correctly. To further elaborate on the advantage this presents, we note that apart from adding geometry using a CSG union, we can also do a CSG difference. Performing a CSG difference between the base and the texture results in the texture being subtracted from (or carved out of) the base volume. This is illustrated in figure 11.3, where a number of stars are carved out of a teapot.

As always, everything has its price: In our case, the price we have to pay for all the benefits just described is speed. Warping implicit surfaces is much slower than warping explicit geometry. There are basically two reasons for this difference in performance. The first reason is, that we need to calculate a value for each grid point in the embedding volume of the warped texture, and that the

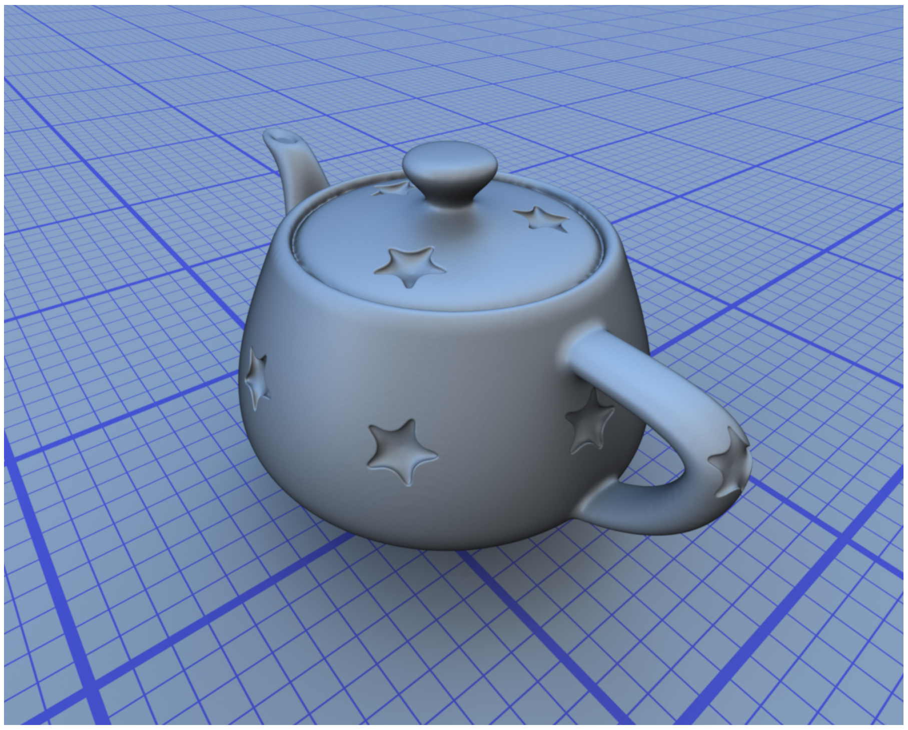


Figure 11.3: Teapot with star shaped cut-outs created using our mapping and CSG subtraction.

embedding volume typically has significantly more grid points than a explicit representation has vertices. Even though we are using a memory efficient narrow band representation, we still need to calculate the value for all grid points in the full embedded volume before we can determine whether or not those values should be stored in the compact representation. Thus, the total number of *points* needing to be mapped from one space to the other is typically higher for the implicit mapping than for the explicit mapping. The second reason that the implicit mapping is slower is, that we need to perform the mapping from patch space to texture space rather than the other way around. As the sampling of texture space is a regular sampling, we can exploit this to create a very fast mapping from texture space to patch space. The sampling of patch space on the other hand is by no means regular, and we must thus apply a more generic interpolation scheme. Consequently, we have not been able to optimize this method to the same degree as we could with the scheme used for the mapping from texture space (for example, finding the nearest sample point in texture space can be achieved in constant time, whereas the same operation in patch space requires logarithmic time in the number of sample points). Considering these two issues, we cannot expect the implicit mapping to compete with the Shell Map method, or our semi-implicit mapping which is just as fast as the Shell Map method, in terms of execution time. We do, however, believe that the improved result, the possibility to generate a single topologically connected surface and the overall flexibility of our system more than compensates for this.

As a part of the geometric texture mapping methods presented in this thesis, we have presented three different particle based parameterizations of the patch space, and three different mapping methods used to warp a geometric texture to follow the base surface as dictated by the parameterization. While these different methods help to illustrate some of the flexibility of the proposed methods, they also pose a potential problem: What method should be used when? Of the three methods, the semi-implicit method stands out because it is based on an explicit representation of the texture geometry. Obviously, if the texture for some reason cannot be converted into an implicit representation, then the semi-implicit method is the only option. Similarly, if the response time is an issue and a single connected surface is not required, we also recommend using the semi-implicit approach. This could for instance be the case in the modeling phase, where the semi-implicit mapping could be used until a satisfactory result is obtained, at which point the final high quality model could be generated using one of the implicit methods. The choice between the two implicit methods is a bit less obvious. Clearly, the Shell Map hybrid is faster than the radial basis interpolation variant, and if the particle spacing in the parameterization is kept small enough, the results are in most cases of a very good quality. Overall, however, the radial basis interpolation based mapping does produce higher quality mappings due to the higher order interpolation used. In particular, there are cases where this higher order interpolation is a great advantage: Consider mapping a texture onto a surface with many relatively large bumps using the reduced distortion parameterization. Due to the high sampling density required for the Shell Map hybrid, these bumps will propagate through the texture and influence the warping of the texture even away from the surface (although some of the effect will wear off away from the surface due to the use of the signed distance function to generate the particles). In contrast, due to the higher order interpolation of the radial basis function interpolation, a significantly lower sample density can be used away from the surface. As a result, the bumps in the surface will have a reduced influence on the mapping away from the surface. In fact, by changing the sample density, we can control to which extend small features in the base surface affects the mapping away from the surface.

As already pointed out, the biggest limitations of the presented methods is the performance of the implicit methods. Most of the examples shown so far has taken from a couple of seconds (*e.g.* the stars added to the bunny in figure 8.11 and removed from the teapot in figure 11.3) and up to several minutes (*e.g.* the baby dragons in figure 8.12) to complete per mapped texture. While some of the time spent can be ascribed to the fact that the current implementation is a *proof of concept* implementation that lacks full optimization, we cannot claim this to be fully responsible for the performance. By introducing the Shell Map hybrid, we managed to reduce the problem somewhat, although there is still plenty of room for improvements. Fortunately, we have several ideas of how to improve on the overall performance of our system. First of all, we believe that we can speed up the Shell Map hybrid method by taking advantage of the coherence in tetrahedron lookups. Whenever we are to locate the tetrahedron containing a given point, we can speed up the process by looking at the previously tested point. As these two points are close in world-space, the new point will either be in the same tetrahedron as the previous one, or in a tetrahedron close by. If we add neighbor pointers to the faces of each tetrahedron, we can step from the previous tetrahedron in the direction of the new point until we find the actual tetrahedron containing this point. As the previous and current points are close in world space, we will in most cases step through no more than one or two tetrahedrons. This essentially means that we can reduce most of the lookups from logarithmic to constant time. There are special cases where this lookup cannot be used, *e.g.* if no previous tetrahedron was found, however, for most grid points, this method should be applicable. Another possible optimization stems from the fact that the calculations performed per grid point in the warped volume are independent of each other. This means that both mapping methods are perfect candidates for a multi threaded implementation allowing it to utilize multiple processors or multiple processor cores. Also, the calculations required for the radial basis function based mapping is completely identical for all grid points, which means that another evident optimization would be to utilize the SIMD instruction set available on most current processors. Finally, we have recently become aware of a the Aranz FastRBF$^{\text{TM}}$ engine [5], which according to the company's website is able to perform radial basis function interpolation significantly faster than with our current implementation. With their RBF engine, the complexity of the radial basis function mapping reduces from $\text{O}(n_g \times n_p)$ to $\text{O}(n_p \log n_p + n_g)$. We will return to this in the future work chapter.

Part IV

Conclusion and Future Work

Chapter 12

Future Work

We have several suggestions for improvements and ideas for future work, both with respect to the terrain rendering algorithms presented in part II and the geometrical texture mapping methods presented in part III, and we will now discuss them in that order.

Regarding the terrain rendering, we have two areas that we would like to further improve in the future: improved texture management; and better support for out-of-core terrain rendering.

To handle textures more efficiently, we plan to investigate a paging system inspired by those described in [14, 22], although in a slightly simplified version. This entails keeping the texturing system as is, but in the cull phase we will ensure that only the textures that are actually needed (or are likely to be needed within the next few frames) reside in texture memory. An alternative we would like to investigate is to cut down the textures size such that each texture is applied only to a single GeoMipMap. This would allow for a much finer control over the texture caching and pre-fetching.

To allow for rendering of terrains with a memory footprint larger than the available main memory, the first step is to implement the compression scheme discussed in section 4.6. If this does not reduce the memory footprint sufficiently, a paging system would be required to load the GeoMipMaps that are not currently loaded, but are likely to be needed within the next couple of frames. Similarly, GeoMipMaps that are not in use, and not expected to be used soon either would have to be *unloaded* in order to free up system memory.

Regarding the geometrical texture mapping, our primary focus for the future is to further improve the performance of the implicit mapping methods as well as adding new features to our flexible system.

We are currently looking at two different approaches for speeding up the implicit mappings. The first is to optimize the code rather than the actual algorithms. As previously mentioned, our current implementation is an experimental *proof of concept* implementation, meaning that it has been developed with a strong focus on flexibility rather than speed. If we choose to remove some of that flexibility, *e.g.* by restricting ourselves to choosing the radial basis function at compile time rather than at run time, we would gain some significant performance improvements. Furthermore, as mentioned in chapter 11, most of the heavy calculations are great candidates for a multi threaded implementation allowing us to utilize multiple processors or multiple processor cores. Similarly, utilizing the SIMD or a similar instruction set available on most current processors would also allow for a significant increase in performance. The other approach for increasing the performance is to look at reducing the complexity of the algorithms. We have already, in chapter 11, presented our idea for improving the complexity for the shell map hybrid: by utilizing knowledge of the previous point-in-tetrahedron lookup, most lookups can be performed in constant time, thereby, in practice, reducing the overall complexity from $\mathrm{O}(n_g \times \log n_p)$ to $\mathrm{O}(n_g)$. At the same time, we are investigating possible ways of improving the performance of the radial basis function based mapping. Our first idea is to use a radial basis function with local support rather than the current global support. This way, we will only need to sum over the particles within a certain distance rather than over all particles. We then intend to place the particles in a spatial data structure such as a kD-Tree or a grid to be able to locate the particles within this distance in sub-linear time. Another idea is to use a similar data structure to let us quickly locate the nearest particle. If the texture coordinate associated with that particle maps to a point in the texture volume outside the narrow band, we may be able to conclude that the current point in patch space will also map to a point outside the narrow band. If this is true, it will not be necessary to perform the full radial basis function interpolation for this point, which again would provide a significant speedup to the mapping. Finally, as briefly mentioned in chapter 11, we have just recently become aware of the Aranz FastRBF$^{\mathrm{TM}}$ engine [5]. According to the company website, utilizing this radial basis function engine rather than a standard radial basis interpolation engine such as the one we are currently using would reduce the time complexity of our radial basis mapping from $\mathrm{O}(n_g \times n_p)$ to $\mathrm{O}(n_p \log n_p + n_g)$. Obviously, we are quite keen on experimenting with this engine in practice.

One of the interesting aspects of the geometrical texture mapping methods in terms of further development is the animation toolbox presented in chapter 10. As mentioned previously, there are a couple of issues with our current methods for animating geometric textures. Most of these issues are a consequence of the fact that there is currently no constraints on the patch corners. Thus, there is nothing that keeps the corner particles from drifting in different directions, thereby distorting the patch and consequently also the texture. Rather than trying to introduce some kind of spatial relationship between these corner particles, which most likely would significantly increase the complexity of the system, we are currently investigating the potential of exponential maps [12,71]. As mentioned in chapter 10, the exponential map is defined by a single seed point and a direction vector. Thus, if we employ a parameterization based on this mapping, we avoid the problem with the defining particles drifting in different directions, as there will only be one defining particle. Furthermore, with this parameterization, it will also be possible to support key frame animations including rotations of the patch, as this will reduce to a (quaternion based) key frame animation of the direction vector. Finally, by employing a parameterization on the exponential maps, we believe that the system will become even simpler and more intuitive to use, as the interface for the exponential maps is indeed very simple and intuitive.

This does not mean that we intend to retire our current corner based approach, as we still believe that is has its own advantages. Specifically, it offers a great deal of flexibility, which is something we plan on taking further advantage of. One such idea is to add more user control over the bounding curves generated between the patch corners by adding a number of control points on the generated curves, and then let the user manually adjust these points to produce the desired curves.

As mentioned in section 8.2, our system is very flexible with respect to the specific methods used to distribute the patch particles. We have already presented three different methods, and we believe that there is still a great possibility to develop additional methods, including the exponential map based method we are currently working on.

One issue that we have not looked at in great detail is the distortion of the warped texture geometry. We optimized our mappings to minimize the 2D distortion of the particles on the surface, and in the case of the reduced distortion parameterization, this was also done at the remaining particle levels. We did, however, not try to minimize the overall 3D texture distortion for any of the parameterizations. We believe that we will be able to adapt the distortion metric presented by Zhou *et al.* [91] to also work with our parameterizations. The method they presented performed an optimization on the texture coordinates associated with vertices of the offset surface used for shell mapping.

Finally, we are interested in looking into how the parameterizations and mapping methods can be adapted to also work with a base geometry specified as a triangle mesh. In particular, we are interested in using a height field as our base geometry, as it would allow us to easily add the kind of details to a terrain discussed in section 4.7. Obviously, this would mean that we would sacrifice some of the features of our methods such as the ability to produce a single smooth topologically connected surface. We do, however, believe that *e.g.* the spline based parameterization combined with the semi-implicit mapping would be a valuable tool for adding trees etc. to a bumpy landscape.

Chapter 13

Conclusion

The focus of this thesis has been on developing flexible and effective algorithms for various kinds of geometric texturing, both in terms of terrain visualization and in terms of geometric texture mapping.

First, we have presented a flexible framework for real-time rendering of large textured terrains. This system stands apart from other systems in that not only does it support real time rendering of large textured terrains, it also abides to a more unusual requirement: it is possible to alter the terrain on-the-fly with virtually no impact on performance. By carefully designing the data layout, and using vertex programs to optimize data storage, less than 2.5 bytes is stored in memory per vertex. While our algorithm is not as fast as the current state-of-the-art algorithms, it is significantly more flexible than previous systems: we support arbitrarily large textures, on-the-fly altering of the terrain and even partial specification of the terrain with on-the-fly updating of the terrain if additional data should become available. Additionally, the presented methods are simple and easy to implement. It is currently being used (and have been for almost two years) in the Topos^{TM} visualization software package [1], that apart from being a commercial product also forms the backbone of a major incident overview demonstrator prototype on the Palcom research project [62, 74].

Second, we have presented robust and flexible techniques for warping and blending (or subtracting) geometric details, in the form of a geometric texture, onto level set surfaces. Our current approach is based on the use of implicit geometry, which makes it easy to merge the base and texture geometry into a single topologically connected object as well as robustly smoothing the intersection between the base and texture geometry guaranteeing a smooth surface with smooth normals. Furthermore, our mapping employs a flexible particle based parameterization. As the parameterization is characterized by the distribution of the particles, we can easily change the parameterization by changing the way the particles are distributed. To demonstrate this flexibility, we have presented three different methods for distributing the particles, including a method that reduces the overall texture distortion. Additionally, a number of interesting effects can be achieved by applying different transformations to the patch particles. One example of this was shown in section 8.5 (figure 8.19).

Although the semi-explicit mapping proposed in this thesis is very fast, the radial basis function based implicit mapping can be slow. The problem is that the speed of the implicit mapping depends, not only on the size of the volume it is being mapped into, but also on the total number of particles defining the parameterization. Although the improved results compared to *e.g.* the Shell Map technique [67] and the possibility to generate a single topologically connected surfaces should well compensate for this, we are currently considering different approached to increase the speed of the mapping without compromising the high quality provided by the radial basis function interpolation. We have already presented an alternative approach based on the tetrahedronization of shell space introduced in the Shell Map paper by Porumbescu *et al.* [67]. As shown in chapter 9, this *Shell Map hybrid* has a lower run time complexity, and in the tests we have performed it was roughly a factor of 7 faster than the radial basis function method. The trade-offs are that the barycentric interpolation used causes interpolation discontinuities, reducing the mapping to C^0 - just as the original Shell Map method [67]. Furthermore, the extended control of the smoothness of the mapping provided by the λ-values in the radial basis function weight calculation is not available for the shell map hybrid. Thus, the two mappings each have their advantages: while the Shell Map hybrid method is the faster of the two, the radial basis function method produces results of higher quality.

Finally, we have presented a set of animation *building blocks*, allowing our geometric texture mapping techniques to be applied in an animation context as well. Although it can be argued that the presented building blocks have not fully matured yet, they show the potential of the presented methods.

We have developed a very powerful and flexible toolbox for geometric texturing and stimulating conversations with people in the movie industry makes us confident that our techniques can be useful in actual production environments.

In general, the idea of using level sets and implicit surfaces as a tool for surface modeling and editing has been introduced in several previous publications([28, 53, 82]). The methods presented here adds to this already existing level set modeling toolbox and will, hopefully, help promoting level sets as a viable modeling tool to coexist with current explicit geometrical representations in future modeling systems.

Bibliography

[1] 43D. Topos. http://www.43d.com/topos.php.

[2] S. Akkouche and E. Galin. Adaptive implicit surface polygonization using marching triangles. In D. Duke and R. Scopigno, editors, *Computer Graphics Forum*, volume 20(2), pages 67–80. Blackwell Publishing, 2001.

[3] S. Akkouche and E. Galin. Adaptive implicit surface polygonization using marching triangles. *Comput. Graph. Forum*, 20(2):67–80, 2001.

[4] Alias. Maya.
http://www.alias.com/eng/products-services/maya/index.shtml.

[5] Aranz. Aranz - applied research associates nz ltd.
http://www.aranz.com/research/modelling/theory/rbffaq.html.

[6] U. Assarsson and T. Möller. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools*, 5(1):9–22, 2000.

[7] P. Bhat, S. Ingram, and G. Turk. Geometric texture synthesis by example. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 41–44, New York, NY, USA, 2004. ACM Press.

[8] J. F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 286–292. ACM Press, 1978.

[9] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *ACM Communications*, 19(10):542–547, 1976.

[10] D. E. Breen, S. Mauch, and R. T. Whitaker. 3D scan conversion of CSG models into distance volumes. In *IEEE Symposium on Volume Visualization*, pages 7–14, 1998.

[11] M. Büscher. Vision in motion. *Environment and Planning A*, 38(2):281–299, 2006.

[12] M. P. D. Carmo. *Differential Geometry of Curves and Surfaces.* Prentice Hall, 1976.

[13] Y. Chen, X. Tong, J. Wang, S. Lin, B. Guo, and H.-Y. Shum. Shell texture functions. *ACM Transactions on Graphics*, 23(3):343–353, August 2004.

[14] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. BDAM — Batched Dynamic Adaptive Meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514, Sept. 2003.

[15] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Planet–sized batched dynamic adaptive meshes (P-BDAM). In *Proceedings IEEE Visualization*, pages 147–155, Conference held in Seattle, WA, USA, Oct. 2003. IEEE Computer Society Press.

[16] R. L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press.

[17] P. Crossno and E. Angel. Isosurface extraction using particle systems. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 495–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[18] W. H. de Boer. Fast terrain rendering using geometrical mipmapping. http://www.flipcode.com/articles/article_geomipmaps.pdf.

[19] D. Dekkers, K. van Overveld, and R. Golsteijn. Combining csg modeling with soft blending using lipschitz-based implicit surfaces. *Vis. Comput.*, 20(6):380–391, 2004.

[20] H. Q. Dinh, G. Turk, and G. Slabaugh. Reconstructing surfaces by volumetric regularization using radial basis functions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(10):1358–1371, 2002.

[21] A. Doi and A. Koide. An efficient method of triangulating equivalued surfaces by using tetrahedral cells. pages 214–224, 1991.

[22] J. Döllner, K. Baumann, and K. Hinrichs. Texturing techniques for terrain visualization. In *Proc. of the 11th Ann. IEEE Visualization Conference (Vis) 2000*, pages 227–234, 2000.

[23] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In R. Yagel and H. Hagen, editors, *IEEE Visualization 9́7*, pages 81–88. IEEE, Nov. 1997.

[24] D. H. Eberly. *3D Game Engine Design*. Morgan Kaufmann, 2000. ISBN 1558605932.

[25] W. Evans, D. Kirkpatrick, and G. Townsend. Right triangular irregular networks. Technical report, Tucson, AZ, USA, 1997.

[26] K. W. Fleischer, D. H. Laidlaw, B. L. Currin, and A. H. Barr. Cellular texture generation. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 239–248, New York, NY, USA, 1995. ACM Press.

[27] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 249–254. ACM, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[28] T. A. Galyean and J. F. Hughes. Sculpting: an interactive volumetric modeling technique. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 267–274, New York, NY, USA, 1991. ACM Press.

[29] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli. C-bdam - compressed batched dynamic adaptive meshes for terrain rendering, sep 2006. To appear in Eurographics 2006 conference proceedings.

[30] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* Wiley, 2005.

[31] Google. Google earth. http://earth.google.com/.

[32] S. Hadap, D. Eberle, P. Volino, M. C. Lin, S. Redon, and C. Ericson. Collision detection and proximity queries. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 15, New York, NY, USA, 2004. ACM Press.

[33] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin. Image analogies. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340, New York, NY, USA, 2001. ACM Press.

[34] J. Hirche, A. Ehlert, S. Guthe, and M. Doggett. Hardware accelerated per-pixel displacement mapping. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 153–158, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[35] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[36] B. Houston, M. Nielsen, C. Batty, O. Nilsson, and K. Museth. Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation. *ACM Transactions on Graphics*, 25(1):1–24, 2006.

[37] B. Houston, M. Wiebe, and C. Batty. RLE sparse level sets. In *Proceedings of the SIGGRAPH 2004 Conference on Sketches & Applications*. ACM, ACM Press, 2004.

[38] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 271–280, July 1989.

[39] D. Kincaid and W. Cheney. *Numerical analysis: mathematics of scientific computing.* Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1991.

[40] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature sensitive surface extraction from volume data. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 15–22, Aug. 2001.

[41] B. D. Larsen and N. J. Christensen. Real-time terrain rendering using smooth hardware optimized level of detail. In *Journal of WSCG, Vol.11, No.1.* EuroGraphics, Feb. 2003.

[42] J. Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*, Washington, DC, USA, 2002. IEEE Computer Society.

[43] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Real-time continuous level of detail rendering of height fields. *Proceedings of SIGGRAPH'96*, pages 109–118, 1996.

[44] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 363–371, Washington, DC, USA, 2001. IEEE Computer Society.

[45] P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[46] A. Lopes and K. Brodlie. Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics*, 09(1):16–29, 2003.

[47] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algoritm. *Computer graphics*, 21(4):163–168, July 1987.

[48] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics*, 23(3), Aug. 2004.

[49] F. Losasso and H. Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics*, 23(3):769–776, Aug. 2004.

[50] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 100–107, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[51] S. Mauch. *Efficient Algorithms for Solving Static Hamilton-Jacobi Equations.* PhD thesis, California Institute of Technology, 2003.

[52] T. Möller and E. Haines. *Real-Time Rendering.* A K Peters, Ltd., 1999.

[53] K. Museth, D. E. Breen, R. T. Whitaker, and A. H. Barr. Level set surface editing operators. *ACM Transaction on Graphics (Proc. SIGGRAPH)*, 21(3):330–338, July 2002.

[54] F. Neyret. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55–70, jan-mar 1998.

[55] M. B. Nielsen and K. Museth. Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets. *Linköping Electronic Articles in Computer and Information Science*, 9(001):ISSN 1401–9841, 2004. Accepted for publication in *Journal of Scientific Computing* January 26, 2005.

[56] M. B. Nielsen and K. Museth. Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets. *Journal of Scientific Computing*, ISSN: 0885-7474, 2006.

[57] M. B. Nielsen, O. Nilsson, A. Söderström, and K. Museth. Out-of-core and compressed level set methods. *ACM Transactions on Graphics, to appear.*

[58] G. M. Nielson and B. Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 83–91, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[59] OpenGL Architecture Review Board.
opengl vertex buffer object extension.
http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt.

[60] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces.* Springer, Berlin, 2002.

[61] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[62] Palcom.
http://www.ist-palcom.org/application-areas/major-incidents.

[63] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *IEEE Visualization '98*, pages 233–238. IEEE, 1998.

[64] H. K. Pedersen. Decorating implicit surfaces. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 291–300, New York, NY, USA, 1995. ACM Press.

[65] D. Peng, B. Merriman, S. Osher, H. Zhao, and M. Kang. A PDE-based fast local level set method. *J. Comput. Phys.*, 155(2):410–438, 1999.

[66] J. Peng, D. Kristjansson, and D. Zorin. Interactive modeling of topologically complex geometric detail. *ACM Trans. Graph.*, 23(3):635–643, 2004.

[67] S. D. Porumbescu, B. C. Budge, L. Feng, and K. I. Joy. Shell maps. In *ACM SIGGRAPH*, volume 24, pages 626–633. ACM, 2005.

[68] X. Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In W. A. Davis and P. Prusinkiewicz, editors, *Graphics Interface '95*, pages 147–154. Canadian Human-Computer Communications Society, 1995.

[69] J. Rickett and S. Fomel. A second-order fast marching eikonal solver.

[70] E. Rouy and A. Tourin. A viscosity solutions approach to shape-from-shading. *SIAM J. Numer. Anal.*, 29(3):867–884, 1992.

[71] R. Schmidt, C. Grimm, and B. Wyvill. Interactive decal compositing with discrete exponential maps. *ACM Trans. Graph.*, 25(3):605–613, 2006.

[72] T. W. Sederberg and S. R. Parry. Free-form deformation of solid geometric models. *ACM SIGGGRAPH Computer Graphics*, 20(4):151–160, 1986.

[73] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. of the National Academy of Sciences of the USA*, 93(4):1591–1595, February 1996.

[74] D. Shapiro, M. Büscher, P. Mogensen, M. Christensen, and P. Ørbæk. Spatial computing and ambient collaborative environments for design and construction. In *Proc. 3rd International Conference on Innovation in Architecture, Enagineering and Construction (AEC2005)*, June 2005.

[75] P. Shirley. *Fundamentals of Computer Graphics.* AK Peters, 2002. ISBN 156881-124-1.

[76] P. Shirley and R. K. Morley. *Realistic Ray Tracing.* A K Peters, Ltd., 2003.

[77] C. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock capturing schemes. *J. Comput. Phys.*, 77:439–471, 1988.

[78] C. C. Tanner, C. J. Migdal, and M. T. Jones. The clipmap: A virtual mipmap. In M. Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 151–158. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

[79] M. Tarini, K. Hormann, P. Cignoni, and C. Montani. Polycube-maps. *ACM Trans. Graph.*, 23(3):853–860, 2004.

[80] G. Turk. Texture synthesis on surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 347–354, New York, NY, USA, 2001. ACM Press.

[81] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.-P. Seidel. Faster isosurface ray tracing using implicit KD-trees. *IEEE Trans. Vis. Comput. Graph*, 11(5):562–572, 2005.

[82] S. W. Wang and A. E. Kaufman. Volume sculpting. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 151–ff., New York, NY, USA, 1995. ACM Press.

[83] X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum. Generalized displacement maps. In H. W. Jensen and A. Keller, editors, *Eurographics Symposium on Rendering*, 2004.

[84] J. Warren, S. Schaefer, A. N. Hirani, and M. Desbrun. Barycentric coordinates for convex sets. Technical report, Rice University, 2004.

[85] E. Weisstein. Wolfram mathworld. http://mathworld.wolfram.com.

[86] R. T. Whitaker and D. Breen. Level-set models for the deformation of solid objects. In *Eurographics Workshop on Implicit Surfaces*, pages 19–35, June 1998.

[87] Wikipedia.
radial basis function.
http://en.wikipedia.org/wiki/Radial_basis_function#RBF_types.

[88] A. P. Witkin and P. S. Heckbert. Using particles to sample and control implicit surfaces, June 07 1995.

[89] Y.-T. Zhang, H.-K. Zhao, and J. Qian. High order fast sweeping methods for static hamilton-jacobi equations.

[90] H. Zhao. Fast sweeping method for eikonal equations. *Mathematics of Computation*, 74:603–627, 2004.

[91] K. Zhou, X. Huang, X. Wang, Y. Tong, M. Desbrun, B. Guo, and H.-Y. Shum. Mesh quilting for geometric texture synthesis. *ACM Trans. Graph.*, 25(3):690–697, 2006.

[92] R. Zonenschein, J. Gomes, L. Velho, L. de Figueiredo, M. Tigges, and B. Wyvill. Texturing composite deformable implicit objects.

Appendix A

Real-Time Visualization of Large Textured Terrains

Real-Time Visualization of Large Textured Terrains

Anders Brodersen*
University of Aarhus

Abstract

In this paper, we present a framework for real-time rendering of large scale terrains with texture maps larger than what the graphics hardware can display in a single texture. The presented system is compact and efficient, yet very simple and easy to implement.

Keywords: terrain rendering, level of detail, texturing

1 Introduction

Real-time visualization of large terrains has been an active area of research for more than a decade. In the past few years, as a natural result of the constantly increasing capabilities of modern graphics hardware, the focus has turned from CPU intensive algorithms, where level of detail is determined per triangle, to the more GPU intensive algorithms, where the level of detail is determined for a set of triangles at a time. The result is a much simpler and faster level of detail calculation at the cost of a somewhat increased triangle count.

In most cases, however, the important issue of how to add a texture map to these large terrains have been ignored. The problem arises because modern graphics hardware is limited to displaying textures of sizes up to 2048×2048 or 4096×4096, which for most applications (particularly games) is more than sufficient. However, for a terrain covering an area of 80km^2, such a texture would only provide one texel per 20 or 40 meters!

In this paper, we demonstrate how a geometric mipmap based terrain engine can be adapted to efficiently render large scale terrains and at the same time allow textures larger than what the graphics hardware is capable of displaying using a single texture. By combining carefully designed data structures with the use of vertex programs, the proposed system requires less than 2.5 bytes per vertex.

2 Related Work

Algorithms for interactive rendering of height fields are typically divided into two categories: Algorithms that take advantage of the regular structure of the datasets to create an efficient hierarchical representation of the terrains, which is then used for run-time by progressive mesh refinement or simplification[Duchaineau et al. 1997; Lindstrom and Pascucci 2002]; and algorithms based on a more general unconstrained triangulation of the terrain, such as the BDAM algorithm[Cignoni et al. 2003] and the view dependent progressive meshes presented by Hoppe[Hoppe 1998].

*rip@daimi.au.dk, Åbogade 34, 8200 Århus N, Denmark

Inspired by the massive evolution in consumer graphics hardware, a new group of algorithms has started to appear. Taking advantage of the highly increased triangle throughput of modern graphics hardware, existing algorithms have been modified[Levenberg 2002; Cignoni et al. 2003] and new algorithms have been invented[Losasso and Hoppe 2004; de Boer 2000]. What these new algorithms have in common is that they utilize far simpler algorithms which, at the cost of rendering more triangles than required to achieve the desired mesh quality, has a much lower CPU overhead. In other words, most of the work is shifted from the CPU to the GPU.

Although texture mapping is an important aspect of terrain rendering, most papers touch only very briefly the subject. The standard approach for the systems that do handle large textures is to partition the texture into tiles, binding each tile to a certain part of the terrain[Hoppe 1998]. In some cases the textures are arranged into a pyramidal structure to facilitate texture level of detail along with the geometric level of detail[Döllner et al. 2000; Cignoni et al. 2003; Hua et al. 2004]. In all cases, the texture handling is tightly coupled with the geometrical level of detail algorithm; the only truly general approach is the clip-map[Tanner et al. 1998], which requires special hardware. The approach presented in this paper is also based on texture tiling, but with texture level of detail currently limited to standard hardware controlled mipmapping. Extending this system with texture management as in [Döllner et al. 2000] and [Cignoni et al. 2003] can easily be done with no significant changes to the rest of the system.

3 Data Structures and Memory Layout

The terrain engine presented here has been implemented as part of a commercial program, requiring more than just fast rendering of the terrain. One requirement is the ability to render the terrain with a texture larger than the textures displayable by current hardware. Another requirement is that preprocessing of the data should be limited to no more than a few minutes.

In designing a system that satisfies all of our requirements, we have turned to the GeoMipMap algorithm[de Boer 2000], where the terrain is divided into smaller patches, called GeoMipMaps, of size $(2^n + 1) \times (2^n + 1)$. The original GeoMipMap algorithm is clearly designed for smaller scale terrains used in games, but we have extended it to be suitable also for rendering larger terrains.

Our system uses a 3 level data structure:

- At the bottom level we have a 2D array of *GeoMipMap* structures, containing the heights of the vertices in that particular patch as well as some information needed for the level of detail algorithm. The class declaration is listed in figure 1.

- At the mid level, we have a 2D array of the *MapBlock* structure. This structure allows us to control the texture mapping. Each MapBlock controls $n \times m$ GeoMipMaps as well as one or more textures, used for decorating all of the controlled GeoMipMaps. Like [Döllner

```
class GeoMipMap
{
   //Current level of detail
   unsigned short LoD;
   //delta max for each detail level
   unsigned short *deltaMax;
   unsigned short *heights;
   VboElement *vboElm;
   //Bounding box
   unsigned short ymin, ymax;
};
```

Figure 1: Class Declaration of GeoMipMap. The VboElement entry is described in section 5.

et al. 2000], textures can be combined using any of the blending operators supported by OpenGL.

- Finally, at the top level we have a single *Terrain* structure, which forms the interface between the terrain engine and the rest of the system. The Terrain structure also holds all data that can be reused between different MapBlocks or GeoMipMaps.

Storing the heights in the GeoMipMaps instead of having one large 2-dimensional array means that heights along the shared edge of two GeoMipMaps need to be stored in both GeoMipMaps. However, as we will see in the next sections, the benefits of this layout makes this a very small price to pay.

4 Level of Detail

The level of detail algorithm used in our system is based on the geometrical mipmapping algorithm presented by de Boer[de Boer 2000]. We follow the convention from [de Boer 2000] that the y coordinate of our vertices represents the height of the terrain.

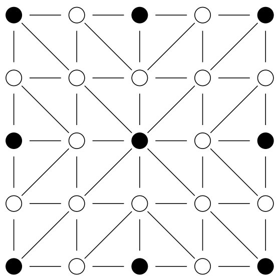


Figure 2: Mesh layout for a map size of 5. The black circles are the vertices used for lower detail level mesh (level 1). Both the white and black circles are used for the highest resolution mesh (level 0).

The terrain is subdivided into a number of smaller patches, called GeoMipMaps, of size $(2^n+1)\times(2^n+1)$ samples (typically either 17×17 or 33×33), which is then rendered at full resolution, every second vertex only, every fourth vertex only, etc., depending on the desired level of detail. In other words, the desired level of detail is determined for the entire GeoMipMap, making the refinement process both simple and efficient. Changing from one GeoMipMap level to the next simply amounts to removing every second vertex in both directions, thus reducing the number of vertices from $(2^n+1)\times(2^n+1)$ to $(2^{n-1}+1)\times(2^{n-1}+1)$, which is depicted in figure 2. At creation time we calculate, for each level of each GeoMipMap, the maximum geometrical error caused by changing from level 0 (highest resolution) to that level. This is the largest vertical distance between a vertex in the original mesh and the triangulation of the current level. Selection is then done at run-time, given the current view parameters and the world space bounding box of a GeoMipMap, by calculating the maximal geometrical error allowed inside that bounding box, with respect to a user supplied threshold. This value is then compared to the error values for the GeoMipMap, and the lowest detail level with a maximum error below this value is chosen to be rendered.

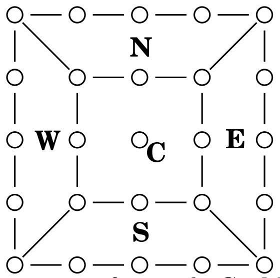


Figure 3: The 5 regions of a single GeoMipMap implicitly defined to simplify the task of avoiding cracks in the mesh. Note that when no more than 3x3 vertices remain, the center region is empty.

One last issue with the level of detail algorithm is how T-vertices and the resulting cracks in the mesh are avoided. We divide each GeoMipMap into 5 separate regions, see figure 3, where tessellation of the center region is based entirely on the currently selected level of detail. The tessellation of the four border regions are based on the currently selected level of detail, and the level of detail of the neighbor GeoMipMap sharing an edge with that region. If the neighbor is at a higher resolution, we add the missing vertices to the shared edge to ensure a consistent tessellation between GeoMipMaps. This is demonstrated in figure 4. This is the opposite approach of [de Boer 2000] and [Larsen and Christensen 2003], where vertices are removed rather than added. Removing vertices instead of adding them results in fewer triangles to render, but at the cost of removing triangles with a potential geometrical error larger than the calculated maximum. We believe that providing a tessellation of a (potentially) lower quality than implied by the user specified threshold is an ill design choice, and therefore prefer the slightly increased triangle count.

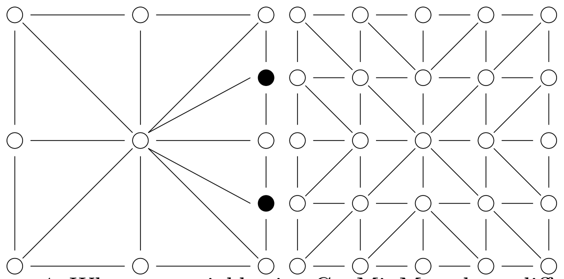


Figure 4: When two neighboring GeoMipMaps have different resolution, extra vertices (black dots) are added to the lower resolution GeoMipMaps representation of the shared edge, making the two edges identical.

4.1 Keeping Memory Usage at a Minimum

An important benefit from using a regular grid height map for terrain visualization is that the x and z coordinates can easily be calculated at runtime, and thus need not be stored. To avoid recalculating the x and z coordinate whenever we need to draw a triangle, we take advantage of the fact that each map has its own copy of the y coordinates making up that map. By precalculating the x and z coordinates of *one* map, we can reuse these coordinates for all other maps, by simply translating the vertices to the position of the new map relative to the original map.

In practice, this means that we store the x and z coordinates of the lower left block in one array, and the y coordinates for each of the $n \times m$ maps in separate arrays. When rendering, we pass in the x and z coordinates using the 0th OpenGL vertex attribute array, and the y values using the 1th vertex attribute array. We then use a simple vertex program to assemble this into a full vertex.

Using the GeoMipMap structure listed in figure 1, the amount of memory needed to store a single GeoMipMap of size 17×17 is no more than 604 bytes. The MapBlock structure is a lightweight structure containing nothing more than a compressed bounding box (2 shorts), two indices, one or more texture ids and a list of visible maps, thus it contributes little to the overall memory consumption. As the top level Terrain structure is never instantiated more than once, even with all the indices stored (see section 5), the total memory consumption for even moderate size terrains[1] is less than 2.5 bytes per sample.

5 Efficient Rendering

Vertex Buffer Object Management In order to achieve high frame rates, it is important that we have all the needed vertex data in graphics memory. However due to the large memory requirements of the textures combined with the additional use of graphics resources by the application itself (that is, textures and vertex data for other geometry displayed along with the terrain), it is equally important that we do not waste resources on parts of the terrain that are not drawn.

For this reason we have implemented a manager for OpenGL vertex buffer objects. Each GeoMipMap stores a pointer to a VBOElement, which is a wrapper around an OpenGL vertex buffer object (VBO). VBOElements are managed using a standard *least recently used* approach. Whenever we are about to render a GeoMipMap with no VBOElement assigned, we revoke the least recently used, and assign it to the GeoMipMap.

An important detail for this to work properly is that no VBO is allowed to be used more than once per frame. Should a VBO be used twice in the same frame, then the least recently used sharing approach will often cause most if not all VBOs to be updated, effectively killing performance. To avoid this, we keep track of how many VBOs are used each frame. If at any one time we have used all VBOs in our queue, and a GeoMipMap makes a request for a VBOElement, a new one is immediately created instead, growing the queue to fit the current requirements. If the number of VBOElements in use is less than the current size of the queue, we slowly shrink the queue[2] in order to reclaim graphics resources when possible.

Pre-calculated indices For efficiency, we pre-calculate the indices for all possible level of detail configurations for a GeoMipMap and its 4 neighbors, which for GeoMipMaps of size 17×17 results in 354 different sets of indices. This takes up 88896 bytes of memory (58718 after being converted to tristrips), which are turned into triangle strips, and stored in a single VBO. A more memory efficient approach would be to separately render each of the sections of the five section GeoMipMap layout presented in section 3. This reduces the total memory requirement for the (stripified) indices, to a mere 4338 bytes (for a 17x17 GeoMipMap) but at the cost of having five draw calls per GeoMipMap instead of one. Using maps of size 17×17, we have seen a performance increase of up to 7% when using only one draw call and therefore recommend using that approach. However, for map sizes larger than 17×17, the memory needed to store the indices may become a problem, and drawing the five regions separately may then be advisable.

[1]The larger the terrain, the smaller is the influence of the single Terrain structure

[2]By removing at most one element per frame

```
RENDER-TERRAIN()
 1  visibleMapBlocks.clear();
 2  for each mb in mapBlocks
 3  do if mb is visible
 4        then visibleMapBlocks.add(mb);
 5             for each gmm in mb.geoMipMaps
 6             do if gmm is visible
 7                   then mb.visibleGeoMipMaps.add(gmm);
 8                        gmm.calculateDesiredLoD();
 9  for each mb in visibleMapBlocks
10  do mb.setupTextures();
11     mb.updateVertexProgramState();
12     for each gmm in mb.visibleGeoMipMaps
13     do gmm.getNeighbourLoDs(&n, &s, &e, &w);
14        gmm.setupVertexArrays(n, s, e, w);
15        gmm.updateVertexProgramState();
16        renderPatch();
```

Figure 5: Pseudo-code describing the rendering procedure.

Frustum Culling View frustum culling in performed using the optimized two point axis-aligned bounding box/plane intersection test with masking by Assarsson and Möller[Assarsson and Möller 2000]. Culling is performed on the 3 level layout presented in section 3. Because the bounding boxed of all three levels are aligned to the same coordinate system, the n-vertices and p-vertices are the same for all structures, and therefore need only be found once per frame. As a result, frustum culling using the existing data structures is very fast, without introducing any additional data structures.

Texture Handling Handling textures of arbitrary size is done by cutting the textures in to smaller sub-textures, each sub-texture being small enough to be displayable by the graphics hardware. The subdivision of the texture is done in a way that each sub-texture fits exactly $n \times m$ GeoMipMaps. Each sub-texture is then assigned to a MapBlock controlling exactly the $n \times m$ GeoMipMaps covered by the sub-texture. During the cull phase, each visible GeoMipMap is added to a visibility list of the corresponding MapBlock. The visible GeoMipMaps are then rendered, one MapBlock at a time, thus requiring the textures of that MapBlock to be bound only once per frame.

Splitting textures is done in an offline step for image textures that are independent of the actual terrain, while textures, such as light maps, that are tightly coupled to the geometry are generated and subdivided at runtime.

An outline of the rendering loop, including culling, level of detail calculations etc. is depicted in figure 5.

6 Results

The implementation has been tested on a laptop computer powered by an Intel Pentium M 1.5GHz processor with one gigabyte of memory and an nVidia GeForceFx 5650 Go chip with 128MB dedicated graphics memory.

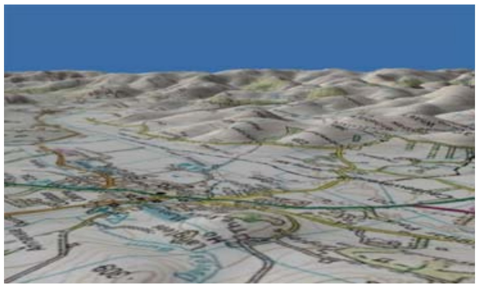


Figure 6: View of the Broad-Law terrain

Our test terrain is an area in Scotland known as Broad Law. The terrain is made up of 8193×8193 height samples, and rendered with a 8192×8192 RGB image tiled into 8×8 textures of size 1024×1024 and a 1024×1024 light map also tiled into 8×8 sub textures, see figure 6.

With a screen space error, τ, of one and a GeoMipMap size of 17×17, the terrain is rendered at on average 70 frames per second at a rate of up to 35MΔ/sec.

It is our experience that the best performance is obtained when using GeoMipMaps of size 17×17, at which point the balance between the number of draw calls and number of unnecessary triangles drawn seems to be optimal. This correlates with the results obtained by Larsen and Christensen[Larsen and Christensen 2003].

Figure 7 shows a wire frame rendering of the terrain from figure 6, with a screen space error of 3 pixels.

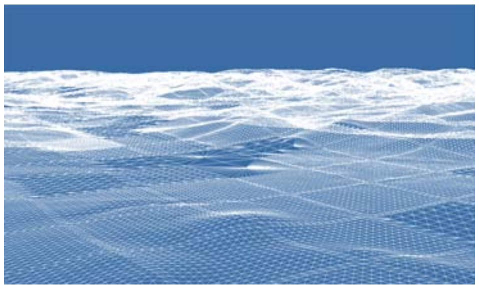


Figure 7: Wire frame rendering of terrain with $\tau = 3$ pixel.

7 Future Work

We have two issues that we would like to address in the future: Improved texture management and better support for terrains larger than what fits in main memory.

To handle textures more efficiently, we plan to investigate a system inspired by those described in [Döllner et al. 2000; Cignoni et al. 2003], although in a slightly simplified version. This entails keeping the texturing system as is, but in the cull phase we will ensure that only the textures that are actually needed or are likely to be needed within the next few frames, reside in texture memory, and that unneeded levels of the mipmap remain unspecified.

As for terrains larger than what fits in main memory, the first step is to implement the compression scheme similar to that of [Losasso and Hoppe 2004]. Another option is to use memory mapped files and then leave the rest to the operating system, as done f.ex. in [Lindstrom and Pascucci 2002].

8 Conclusion

We have presented a framework for real-time rendering of large terrains with texture maps larger than what the graphics hardware can display in a single texture. By carefully designing the data layout, and using vertex programs to allow reusing of as much data as possible, less than 2.5 bytes is stored in memory per vertex.

Finally, the presented system is simple, efficient and easy to implement.

9 Acknowledgments

This work was done in collaboration with 43D (www.43d.com). A special thanks goes to Peter Ørbæk of 43D for a lot of invaluable help and discussions. Also a great thanks to Ken Museth and Christina Brodersen for proof reading and comments to the paper.

References

Assarsson, U., and Möller, T. 2000. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools 5*, 1, 9–22.

Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. 2003. BDAM — Batched Dynamic Adaptive Meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3 (Sept.), 505–514.

de Boer, W. H., 2000. Fast terrain rendering using geometrical mipmapping. http://www.flipcode.com/articles/article_-geomipmaps.pdf.

Döllner, J., Baumann, K., and Hinrichs, K. 2000. Texturing techniques for terrain visualization. In *Proc. of the 11th Ann. IEEE Visualization Conference (Vis) 2000*, 227–234.

Duchaineau, M. A., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B. 1997. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization ´97*, R. Yagel and H. Hagen, Eds., IEEE, 81–88.

Eberly, D. H. 2000. *3D Game Engine Design*. Morgan Kaufmann. ISBN 1558605932.

Hoppe, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, Los Alamitos, CA, USA, 35–42.

Hua, W., Zhang, H., Lu, Y., Bao, H., and Peng, Q. 2004. Huge texture mapping for real-time visualization of large-scale terrain. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM Press, New York, NY, USA, 154–157.

Larsen, B. D., and Christensen, N. J. 2003. Real-time terrain rendering using smooth hardware optimized level of detail. In *Journal of WSCG, Vol.11, No.1*, EuroGraphics.

Levenberg, J. 2002. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*, IEEE Computer Society, Washington, DC, USA.

Lindstrom, P., and Pascucci, V. 2002. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics 8*, 3, 239–254.

Losasso, F., and Hoppe, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics 23*, 3 (Aug.), 769–776.

Tanner, C. C., Migdal, C. J., and Jones, M. T. 1998. The clipmap: A virtual mipmap. In *SIGGRAPH 98 Conference Proceedings*, Addison Wesley, M. Cohen, Ed., Annual Conference Series, ACM SIGGRAPH, 151–158. ISBN 0-89791-999-8.

Appendix B

Geometric Texture mapping using Level Sets

Geometric Texturing Using Level Sets

Anders Brodersen, Ken Museth, Serban Porumbescu and Brian Budge

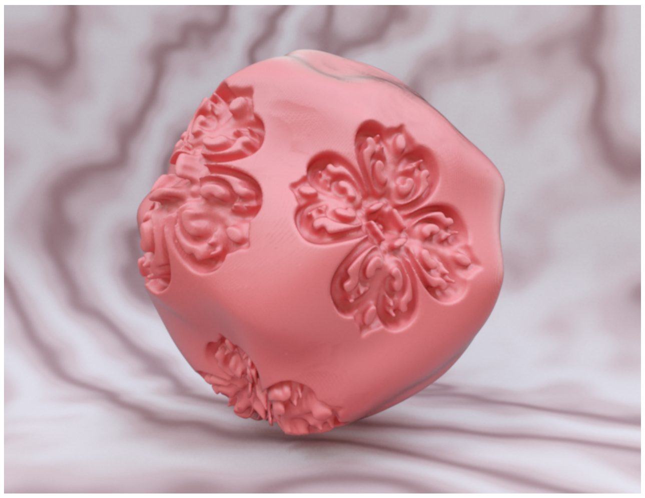


Fig. 1. Carving patterns into an irregular surface by subtracting a geometric texture using the proposed technique.

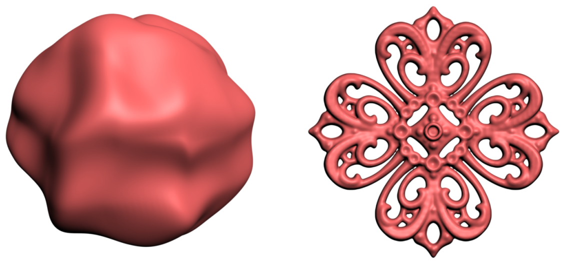
***Abstract*— We present techniques for warping and blending (or subtracting) geometric textures onto surfaces represented by high resolution level sets. The geometric texture itself can be represented either explicitly as a polygonal mesh or implicitly as a level set. Unlike previous approaches, we can produce topologically connected surfaces with smooth blending and low distortion. Specifically, we offer two different solutions to the problem of adding fine-scale geometric detail to surfaces. Both solutions assume a level set representation of the base surface which is easily achieved by means of a mesh-to-level-set scan conversion. To facilitate our mapping, we parameterize the embedding space of the base level set surface using fast particle advection. We can then warp explicit texture meshes onto this surface at nearly interactive speeds or blend level set representations of the texture to produce high-quality surfaces with smooth transitions.**

***Index Terms*— Geometric texture mapping, parameterization, implicit surfaces, volume texturing, geometric modeling.**

I. Introduction

WE present a novel 3D fine-scale explicit and implicit geometry mapping technique based on level sets, interpolation, and radial basis functions. Our work is motivated by the need to easily model fine geometric detail in a convenient fashion. For years, the standard approaches to increase geometric complexity have primarily been 2D texture [1], bump [2], and displacement mapping [3]. These techniques, while capturing a wide range of geometric phenomena, are limited in the types of detail they can represent. Kajiya and Kay [4] realized this early on and introduced volumetric textures to represent more topologically complex structures. Recently, the focus has shifted towards more sophisticated volumetric and geometric texturing approaches in an effort to capture a wider range of complex geometric phenomena [5]–[8].

Anders Brodersen is with University of Aarhus, Denmark.
Ken Museth is with both Linköping University, Sweden and University of Aarhus, Denmark.
Serban Porumbescu and Brian Budge are with University of California, Davis.



(a) (b)

Fig. 2. Base geometry (left) and texture geometry (right) used to created Fig. (1).

Our contribution leverages the recent introduction of DT-Grid data structures and algorithms [9] and the large body of level set research to bridge the gap between existing volumetric and explicit geometric mapping techniques. This is achieved by providing a fast geometric mapping suitable for modeling and previewing and an implicit mapping approach that complements existing explicit mapping techniques (*e.g.* [8]) by generating closed, smoothly blended surfaces. Our general approach uses an implicit level set representation of both the base surface and the texture geometry [10]–[12]. This representation allows for robustness to topology changes during the mapping, flexibility when defining the blend of the base and texture geometry, and is amenable to high quality offset surface generation (see Fig. 15 for a comparison of implicit versus explicit offset surfaces). Additionally, level sets offer a large body of advanced numerical techniques for easily computing surface properties and performing arbitrary deformations. In fact, as has been shown in previous work [13], direct control of blended surface properties is easily achievable with level sets. This high degree of robustness and flexibility, however, comes at the price of increased computational complexity when compared to purely explicit approaches. To address this issue, we have also developed a fast *semi-implicit* technique that can conveniently be used for near real-time previewing. It combines an implicit level set representation of the base surface and an explicit polygonal representation of the textures.

We assume that we are given a base surface as a compact level set and a geometric texture either defined by a triangle mesh or as a compact level set surface. If required, conversion between triangle meshes and level sets can be performed using a fast scan conversion technique [14] or marching cubes [15]. Given this geometry, our system briefly works as follows:

- First, the user manually outlines a patch on the base level set which defines the location of the geometric textures. Given such a patch outline, we then construct a parameterization of the space above the patch. This effectively creates the mapping needed to warp the texture into the space near our base level set surface. The process of defining the patch and the creation of the parameterization is described in section II.
- Section III then presents a simple procedure that maps the texture mesh onto the base level set at nearly interactive rates.
- Alternatively, to produce a single topologically connected surface with smooth blending between the texture and base, the user can utilize a higher quality implicit mapping. This is the topic of section IV.

A. Contributions

The techniques presented in this paper include:

Implicit geometry mapping with smooth blending. We complement existing explicit geometry mapping techniques by using an implicit approach which smoothly blends mapped geometry to create closed surfaces suitable for rendering and various surface property computation. We do this using compact level set representations of the the base and texture surfaces. It is the first general texture space to shell space mapping technique utilizing implicits that we are aware of.

Fast semi-implicit geometry mapping. We also introduce a near real-time *semi-implicit* mapping approach that combines an implicit level set representation of the base surface with an explicit polygonal representation of mapped textures. This technique is useful as a preview tool (prior to implicit mapping) and as a stand alone method for mapping explicit geometry.

Flexible volumetric parameterization. We compute a low distortion parameterization with a minimum of user-interaction. Our parameterization is not dependent on prior surface texture coordinates. Instead, it is based on a local parameterization generated on the fly, using a simple and easy to use point and click interface. Furthermore, our parameterization is characterized by the distribution of a set of particles, but is independent of the algorithm used to distribute these particles. This means that the particles can be distributed in a number of different ways, allowing for a vast number of unique mapping effects. Finally we include results from a simple free-form variation of our mapping technique where the texture warping is derived and controlled by a deformable spline curve.

B. Related Work

Our work builds on level set, implicit surface modeling, and volumetric and geometric texture research. A recent body of work proposing various compact data structures and fast algorithms for level set models [9], [16], [17] is critical to our work. Common to all these data structures is that they *uniformly* sample distance values to a surface in a narrow band embedding the surface. This uniform sampling is paramount to perform our smooth blending operations since they amount to solving mean curvature based level set equations. This effectively requires spatial discretization of parabolic partial differential equations which, to the best of our knowledge, cannot be accomplished accurately on non-uniform grids. Consequently, we have not considered adaptive distance fields (*i.e.* "truly adaptive" octrees) [18], though other parts of our texturing pipeline (*e.g.* CSG operations) could potentially benefit from it. Instead we have chosen to base our texture mapping technique on the "Dynamic Tubular Grid" (DT-Grid) presented in [9]. This data structure has been shown to be very CPU and memory efficient and allows us to represent level set models of effective resolutions exceeding 1000^3 using less then 100MB.

Much effort has been put into deriving methods for adding textures to un-parameterized 3D models, specifically implicit surfaces and level sets, including vector field driven texture synthesis [19] and methods based on parameterizations of support surfaces of lower geometric complexity [20], [21]. Common for these methods is a lack of flexibility and user control. Pedersen [22] presented an interactive method to create a parameterization of implicit surfaces by letting the user manually divide the surface into rectangular and triangular *texture patches*. This method has generally been considered as state of the art since its publication in 1995. Recently, Schmidt et al. presented a local parameterization based on discrete exponential maps [23], producing a simple yet powerful interface for texturing implicit surfaces, provided only a local parameterization is required.

Kajiya and Kay introduced the notion of volumetric textures [4]. Their method utilizes volumetric data sampled on a regular grid, and traces rays through a shell volume on a surface. Rays that intersect the shell are transformed to texture space and traced through the sampled data grid. Material properties were constrained across any region. Neyret extends volume textures, allowing the use of multiple different materials in a single region, and objects of different types to be tiled onto a surface [24]. Wang *et al.* present a generalization of displacement maps. For each location in a grid surrounding the base surface, a distance is computed to the geometric texture, called the mesostructure, for some discretization of all directions. Several other variables are precomputed for rendering, including BRDF information and local shadows [6]. Peng *et al.* [25] averaged distance field functions to generate offset surfaces. Then 3D volumes are sliced into 2D textures, and the textures are applied to various levels of the offset surface. The technique allows interactive rendering of the resultant volume. All of these techniques map geometry by first 3D scan converting models into a regular grid which leads to data storage and aliasing related issues.

Fleischer *et al.* propose to use a cellular texturing technique to produce organic looking surface details [26]. While producing impressive results, their modeling approach is not very intuitive due to a rather complicated underlying biologically motivated simulation engine. Bhat *et al.* demonstrate a volumetric extension of the image analogies technique [7]. This allows them to tile a surface with semi-repeatable patterns at high effective resolution. The patterns do not need to be height fields, and can represent complex structures on the surface.

Recently Shell Maps [8] generalized the notion of volumetric textures by mapping explicit geometry without converting models into regular grids. Shell maps are invertible mappings between texture space and shell space – the space near an object – that facilitate the transfer of explicit geometry, procedural functions, and scalar fields as fine scale detail near an object. The approach generates a correspondence between texture space and shell space via a tetrahedral tiling. Point location queries coupled with barycentric mappings between corresponding tetrahedra are used to transform objects between spaces. The technique is powerful, but the resultant mappings are only C^0 continuous at tetrahedral boundaries and can create artifacts like the one shown in Fig. 12(b). Furthermore, the mapped geometry and the base mesh do not create a new closed mesh, which can be problematic for applying shaders over the entire surface. The level set approach presented in this paper complements the explicit geometry representation of Shell Maps by more naturally dealing with sharp discontinuities and changes in topology necessary to generate closed surfaces (when desired).

We present a novel technique for the mapping of geometry onto surfaces. While sharing some conceptual similarities with other methods that map 2D textures (*e.g.* images, bump or displacement maps) and 3D textures (*i.e.* volumetric and geometric) onto meshes there are some significant differences. Our technique can map explicit geometry, but can also treat geometry implicitly which allows us to create closed continuous meshes (topological 2-manifolds). This nice property allows us to define important surface properties like normals and curvatures on the resulting surface. The method requires no surface-wide parameterization, and our local parameterization only requires the user to select the region where they want to map their geometry.

C. Notation

As a prelude to a more detailed discussion of our techniques we shall briefly introduce some terminology. In this paper the term geometry is used interchangeably for both explicit meshes and implicit level sets. Assume we wish to map a geometric texture, A, onto a base surface, B. We shall denote the explicit mesh representation of A as M_A and the implicit level set representation by ϕ_A. The geometric representation of B is always implicit, and will therefore be denoted ϕ_B. The embedding space of A (*e.g.* defined from its bounding box) will be called *texture space*. The corresponding embedding space of A, after it has been mapped onto B, is called *patch space* (analogous to a portion of “shell space” in [8]). The semi-implicit texture mapping then simply works by defining a map of vertices of M_A from texture space to patch space.

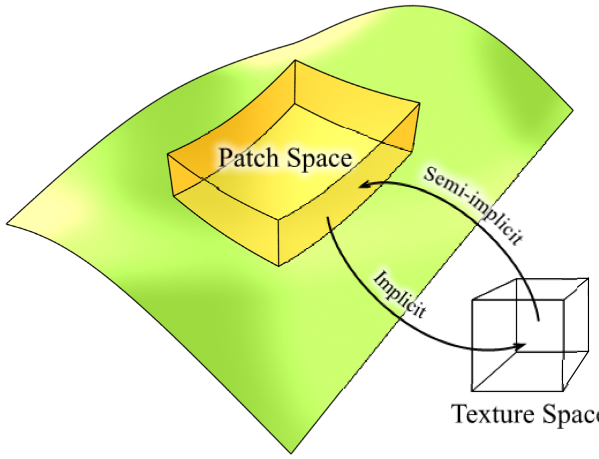


Fig. 3. The **semi-implicit** method uses a direct correspondence between grid points in patch space and texture space. For a given point $\mathbf{x_t}$ the corresponding 8 surrounding points in texture space are found. Weights are computed from these 8 points, and the weights are applied in a trilinear interpolation in patch space. The **implicit** method uses the correspondence between points in patch space and texture space to solve for the weights of a radial basis function. Patch space can then be sampled with $\mathbf{x_p}$s, finding corresponding points in texture space using the radial basis function. Finally, a trilinear interpolation on the texture volume is used to find the distance value.

In contrast, the implicit texture mapping is based on a resampling of ϕ_A into patch space which amounts to establishing a map from grid points in patch space to texture space. Thus, both techniques are based on establishing a mapping between the two embedding spaces, but in different directions (see Fig. 3).

II. Parameterizing patch space

While the volumetric parameterization of texture space is assumed known (*e.g.* $u = x, v = y, w = z$), we have to derive the warped parameterization of the corresponding patch space. For this we have developed a number of techniques, based on an initial u, v parameterization of a 2D patch of the base surface and using Lagrangian tracker particles to sweep out u, v, w in the corresponding patch space. The specific distribution of these tracker particles is created in different ways thereby offering distinct features, such as following the base surface faithfully or lowering distortion, for the resulting 3D texture mapping. This flexibility is one of the strengths of our system. In the following we describe the common base for all our (current) particle distribution methods.

A common initial step for all our current mapping techniques is the definition and parameterization of a 2D patch on the base surface where the texture is to be applied. We define this patch as a simple control quadrilateral[1] on ϕ_B. Constrained interaction with the vertices, $V_i, i = 1 \ldots 4$, of this control quadrilateral is easily implemented since projections of V_i onto ϕ_B amounts to the closest point transform $V_i - \phi_B(V_i)\nabla\phi_B(V_i)$. This is a consequence of our requirement that the level set,

[1]Note that this is not a regular planar quadrilateral since the edges are constrained to lie on the base surface.

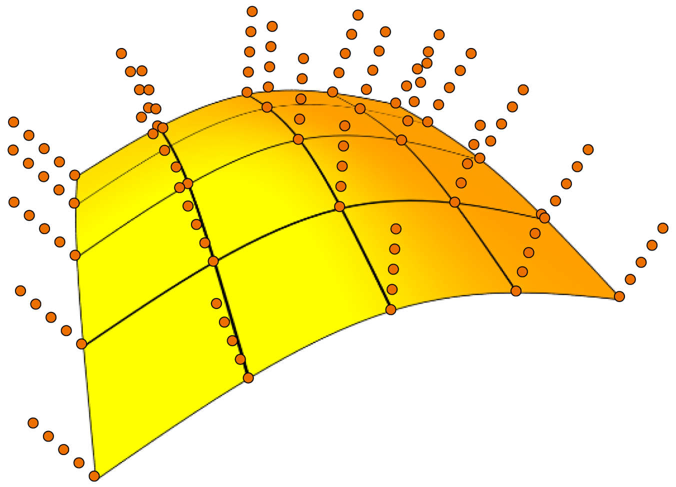


Fig. 4. The surface conforming parameterization propagates the (u, v) texture coordinates using a Lagrangian advection method. The particles roughly follow the normal direction, and the time of arrival is used as the third texture coordinate w.

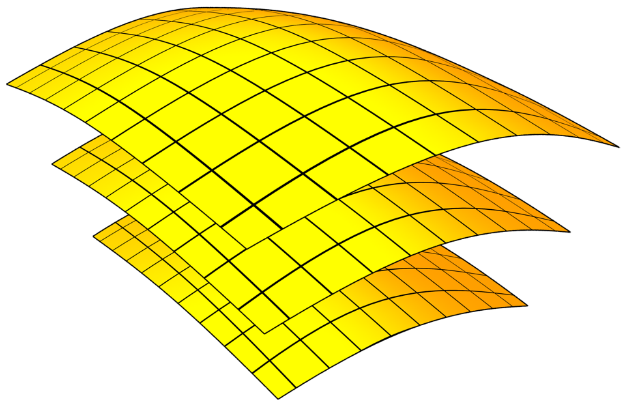


Fig. 5. The low distortion parameterization can be thought of as uniform layers of an onion. The particles are advected as with the surface conforming parameterization, but then they are relaxed to give each level a uniform parameterization.

ϕ_B, is represented by a signed distance function. This control quadrilateral is parameterized using a technique similar to Pedersen's [22]. In short, approximate geodesics are first computed between the vertices V_1 and V_2, V_2 and V_3, V_3 and V_4, and V_1 and V_4. These edges are then subdivided evenly, with a resolution determined by the roughness of the surface[2], and assigned u, v coordinates. Next u, v are swept into the interior of the quadrilateral by means of defining a 2D grid of iso-parametric curves of approximate geodesics connecting the subdivided edges, each curve corresponding to a unique u or v value. At each of the grid points of this 2D iso-parametric grid we place a Lagrangian tracker particle, *i.e.* an infinitely small and massless particle, each associated with a unique u, v, w coordinate. The u and v values are obtained from the two curves intersecting at that point, and the w value is set to zero. In the following we refer to these Lagrangian tracker particles as patch particles or just particles. The position of the patch particles are then optimized to reduce texture distortion. This is achieved by means of a simple constrained mass-spring model [27] where particles on the boundary curves of the patch quadrilateral are fixed and the remaining interior particles are restricted to lie on the base surface.

Surface conforming parameterization: Once the patch particles are generated on the base surface, we propagate the particles along the gradient field of ϕ_B until they reach a desired offset (*i.e.* level of ϕ_B). The w texture coordinate for the advected particles is then defined to be 1. In the case of the implicit mapping described in section IV, it is often necessary to have intermediate layers of particles with $0 < w < 1$. This is obtained by distributing a number of particles evenly on the line segment between each advected particle and it's corresponding particle on the surface, using linear interpolation to determine the w value. Fig. (4) illustrates the particle set distributed for a single patch using this method. Note thateven though ϕ_B is defined as a signed distance function, two particles with the same w coordinate will generally not lie at the same distance away from B (Unless $w = 0$). This is a consequence of the fact that the gradients are strictly speaking not defined at points that have more than one closest point transform to B since here ϕ_B is only C^0. This occurs along the medial-axis of B and numerically manifests itself as $|\nabla\phi_B| \leq 1$ when using central finite difference to compute the gradient. This has the desired feature that although the advected particles might reach other particles, they will never cross paths[3]. As the particles generated by this method are generally not uniformly distributed in patch space, this can lead to significant distortion of the geometric texture. We note, that depending on the application, this may or may not be a desirable feature.

By distributing the tracker particles as outline above, we end up with a mapping that essentially resembles shell-mapping [8]. Consequently this distribution scheme is hampered by most of the limitations of Shell Map, in particular the sensitivity of the mapping with respect to the curvature of the base surface (See section V). However, one of the main strengths of our method is the flexibility with respect to the distributing the tracker particles. We next present two alternative particle distribution schemes that offer different and improved properties of the resulting geometric texture mapping.

Reduced distortion level set parameterization: The problem with the previous particle distribution method is the (implicit) dependence of the curvature of the base surface. As the tracker particles are advected away from the surface in a direction

[2]As we assume ϕ_B is regularly sampled with $dx = dy = dz$, keeping the sample distance below dx guarantees a sufficient sampling. If, however, the surface is smooth, a lower sample rate is often sufficient.

[3]Numerical roundoff errors and inaccuracies in the finite difference potentially breaks this guarantee, although such particles are still guaranteed to remain close together.

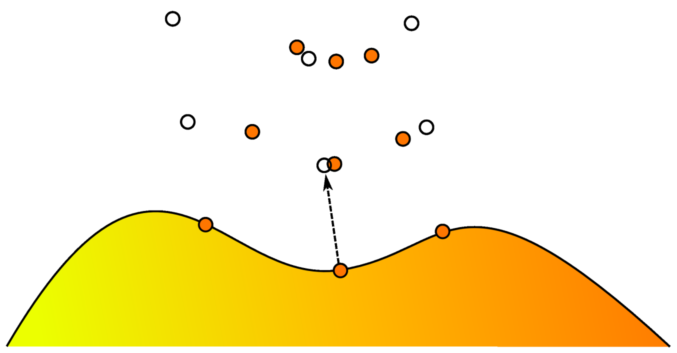


Fig. 6. Specifying a different direction for the particles to evolve along adds extra flexibility to the parameterization. The white particles are obtained by specifying a custom direction of evolution, parallel to the normal at the center point, at both control vertices. The orange particles correspond to the surface conforming distribution.

normal to the surface, small irregularities in the surface can cause severe distortion of the texture. This is due to the fact that particles will typically move closer together in concave regions and away from each other in convex regions. To mend this, we introduce a particle distribution scheme with a stronger focus on the vertices of the user specified control quadrilateral. With this method, these vertices are the only particles to be offset along the gradient field of ϕ_B. At regular intervals, derived from the desired offset height and the desired number of particle levels, a new level of particles is created from the four advected control vertices. We do this using the same technique as used for the particles on the surface, only this time we embed it on the ϖ'th level set of ϕ_B, where ϖ is the (fictitious) time during the propagation. The particles at this level are assigned a w value ϖ divided by the desired offset height. The overall result is a uniform parameterization of each discrete level in the patch space, see Fig. 5, leading to geometric texture mappings with significantly less distortion. This method has an additional number of advantages over the first particle distribution method. First, as a new set of particles are generated at the individual levels, the number of particles at each level are independent. Thus, if the *surface area* of the patch changes with the distance to the base surface, we can adjust the number of particles generated at each level to maintain a desired particle density, thereby enabling a sufficient sampling of each level. Furthermore, we can optionally let the user specify the direction along which each control vertex is offset, rather than forcing it to be in the normal direction. The effect of this is depicted in Fig. (6). By allowing the user to specify the offset direction, we add an extra level of control over the final result. This allows, for example, the user to control the distortion of a texture with a large offset in the w direction on a highly curved surface, as seen in Fig. (6). We have used this extra control in several examples in the following sections, most notably in Fig. (12(a)).

The difference between the (initial) surface conforming parametrization and the new low distortion parameterization is illustrated in Fig. (7). This example clearly shows how the mapping based on individually advected particles (Fig. (7(a))) follows the local curvature of the base surface more closely than the mapping based on uniformly distributed particles (Fig. (7(b))), whereas the latter reduces the overall distortion of the mapped geometry.

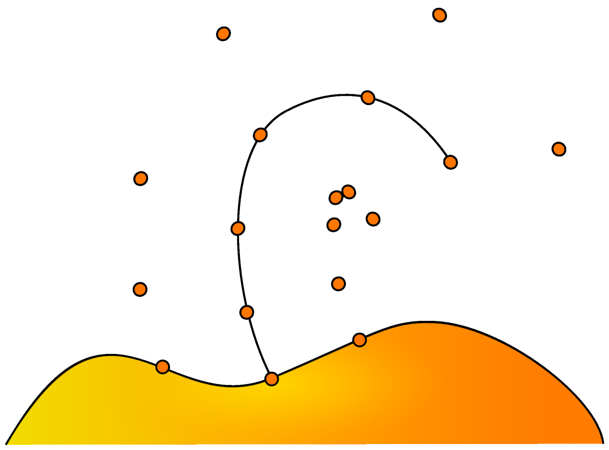


Fig. 8. Parameterization of a patch (simplified to 2D) using the spline advection paramerization. Note how the particles *move* faster or slower depending on the curvature of the spline-curve.

Spline advection: The two particle distribution schemes outlined above both rely on the distance transform of the base surface (*i.e.* the level set ϕ_B) to respectively propagate the particles in the patch space. This effectively means that texture information is propagated in a fixed direction away from the base surface. To add more flexibility we have developed a third parametrization scheme where the particles are propagated along a spline curve originating at the center of the patch. It works as follows: As with the previous distribution schemes, we start by generating the particles on the base surface, assigning u, v-coordinates to each particle. The particles are then propagated in small steps in the direction defined by the spline curve. At each step, the particles are furthermore rotated around the current spline point to align with the tangent of the curve at that point, see Fig. 8. As in the previous methods, copies of the particles are saved at regular intervals, and a w-coordinate, derived from the normalized distance traveled along the spline, is assigned to each particle. An example mapping generated with this technique is shown in Fig. 9.

We note that during the propagation of the particles along the spline curve, care must be taken to avoid particles crossing paths. This would potentially lead to non-monotonic interpolations of the corresponding texture coordinates which in turn result in inconsistent texture mappings. One possible solution to this problem is to treat the advancing particles as small spheres and apply *continuous collision detection* algorithms [28] to ensure that particles do not cross. Continuous collision detection algorithms, while more difficult to implement, offer several advantages over their discrete counterparts. Most notable are their ability to compute the time of first contact versus the discrete approach of simply sampling an object's trajectory and reporting intersections (small, fast moving objects could pass through each other).

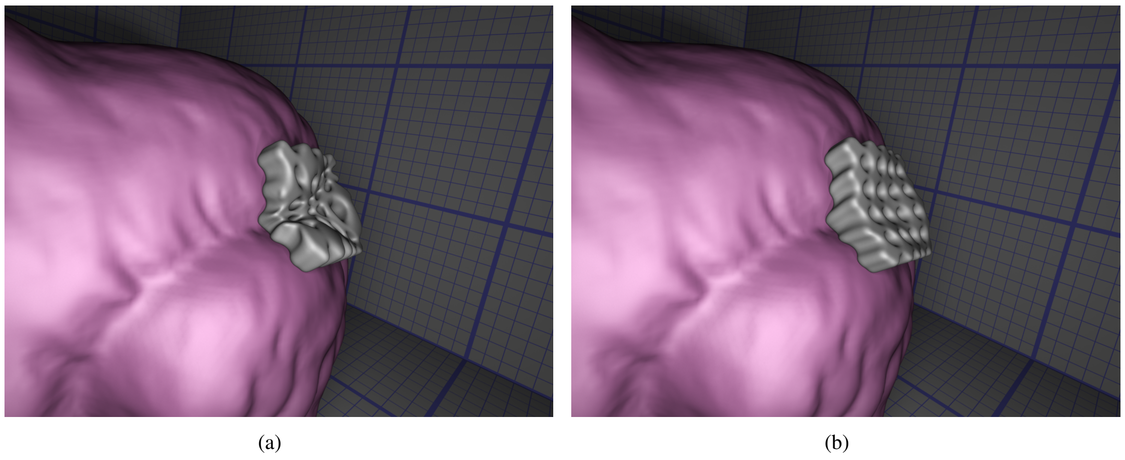


Fig. 7. Left: Mapping a geometry texture to a bumpy part of the bunny using the surface conforming parameterization. Right: and using the low distortion parameterization.

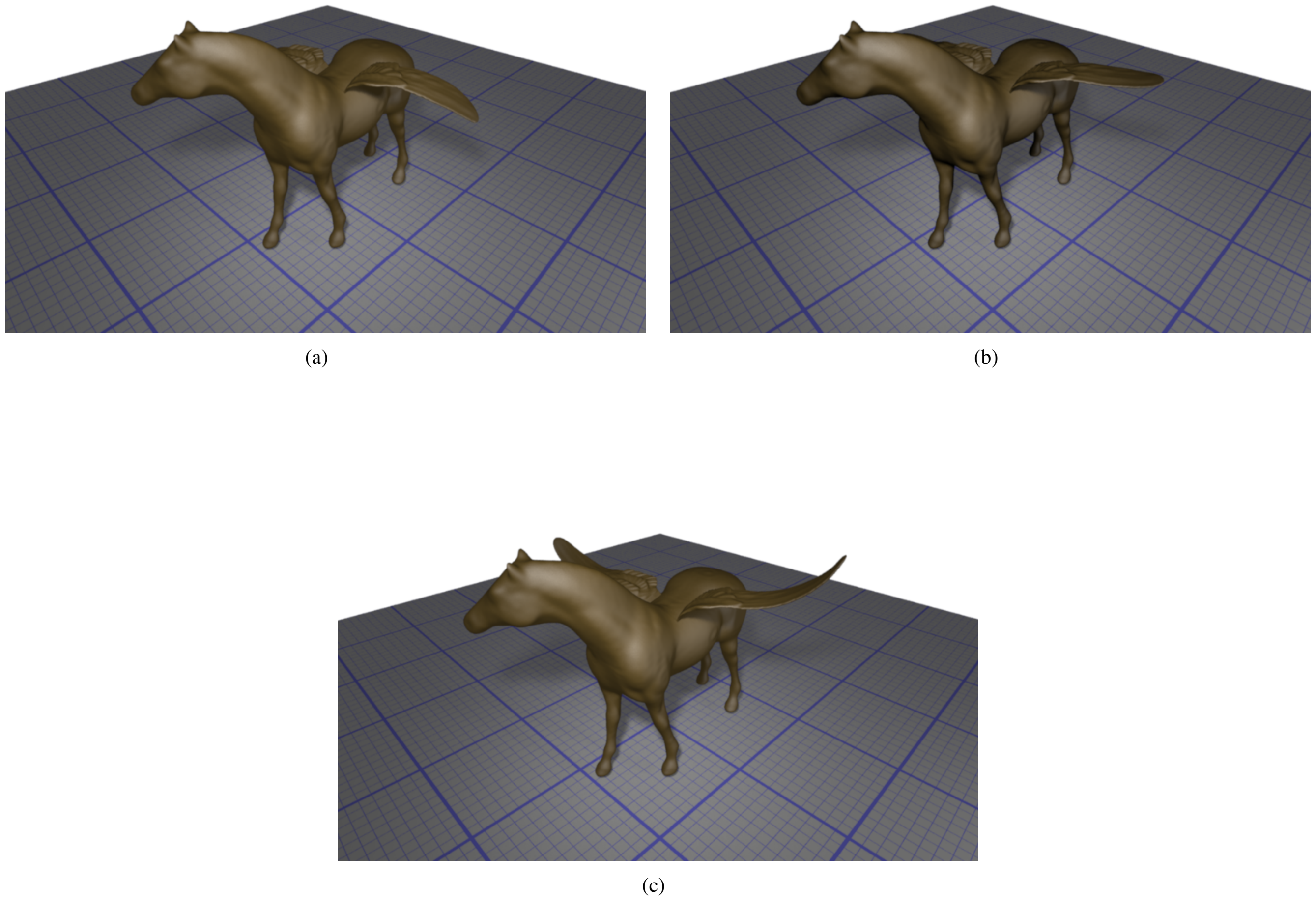


Fig. 9. The three images show a horse with wings mapped onto it in three different postures using the spline based particle distribution scheme.

As a final remark we note that both the surface conforming and the low distortion parameterization assume that ϕ_B is defined throughout the patch space. Since we employ a very storage efficient level set representation of ϕ_B, [9], distance information is only stored in a narrow tube of B. Hence, as a prelude to the parameterization methods outline above we first sweep out distances from this narrow tube to the remaining patch space (which is typically a very small sub-space of the bounding volume of B). This has been implemented very efficiently using the *fast sweeping* method [29] which has linear time complexity in the number of voxels in the patch space.

III. Near real-time semi-implicit mapping

We have developed a simple and efficient semi-implicit technique which can be used as a "preview mode" for our implicit mapping, to be described in the next section. The semi-implicit method maps an (explicit) polygonal mesh, M_A, onto the implicit base surface, ϕ_B, by warping the vertices of M_A in texture space into patch space using fast tri-linear interpolation. The mesh connectivity is left unchanged. This technique, as well as the implicit technique, can be used in combination with any of the parameterization methods described in section II.

The semi-implicit mapping makes use of the fact that the patch particles form a *semi-regular* three-dimensional lattice in texture space - see Fig. (10), left. By this we mean that, in texture space, the particles are distributed into regularly spaced levels in the w direction. Each of these levels consists of a two-dimensional regular grid of particles, but the number of particles need not be the same at all levels (See Fig. (10) for a two-dimensional example). Since the *texture value* associated with each particle is given by their position in patch space, we can define a mapping $\Phi_{t\rightarrow p}(\mathbf{x}_t) = \mathbf{x}_p$ of a vertex $\mathbf{x}_t = (x_u, x_v, x_w) \in M_A$ as a tri-linear interpolation of the particle texture values. As the number of particles may not be the same at each level in the patch space, we need to apply the interpolation in a specific order: We first interpolate at the two levels located immediately above and below the vertex in texture space, followed by an interpolation in between the levels. Fig. (10) illustrates this: First the patch space position of the blue dots are obtained from interpolation along the green line segments. Next, we interpolate the values (patch space position) of the blue dots along the yellow line to get the patch space position of the vertex (red dot). Because each particle level form a regular two-dimensional grid, and the levels are uniformly spaced, finding the interpolants is a constant time operation. Thus, calculating the patch space position of a single vertex is also a constant time operation.

We briefly note that the proposed mapping is somewhat reminiscent of the free-form deformation technique presented by Sederberg *et al.* [30]. The main difference is that our scheme can handle semi-regular samplings and is strictly bounded to the patch space. These properties are very important for our application and are not shared by the higher order interpolation proposed in [30]. Figure 5 in [30] clearly illustrates that geometry is not bounded to the control-polygon which in our application would result in textures mappings that are not explicitly confined to the base surface.

IV. High-quality implicit mapping

The implicit mapping allows us to warp and subsequently blend level set representations of both the geometric texture

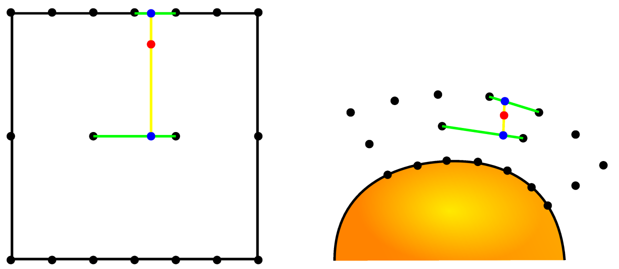


Fig. 10. Illustrating the semi-implicit mapping on a patch set with a different number of particles at each level. To get the position of a given vertex (red dot) in patch space (right) given it's position in texture space (left), we first interpolate along the green lines to get the patch space position of the blue dots. These are then used to interpolate along the yellow line to get the patch space position of the texture space vertex.

and base surface. We use radial basis functions to perform our mapping. The algorithm is as follows. First, we define a regular 3D grid, bounding the region of space spanned by the patch particles. We call this the *embedding volume*. The resolution of this grid is chosen to match the resolution of the grid on which the texture level set is sampled in texture space. Next we define a mapping from the patch space into the texture space by means of radial basis function interpolation. This essentially allow us to resample our texture geometry in patch space. More specifically, for each grid point x_p in the embedding volume, we map it to texture space via the radial basis function, resulting in the point x_t. We then use the point x_t to perform an interpolation[4] on the texture volume, thereby getting the desired distance value. Once all points in the grid are assigned a distance value, the embedding volume will contain a warped instance of the texture geometry.

The method we use for our radial basis function is similar to that of Dinh et al. [31], which is a good candidate because of its robustness with respect to irregularities of the sample points. Furthermore, it adds flexibility due to the fact that it allows for both strict interpolation as well as approximation, simply by varying a parameter (λ_i). For the sake of completeness we will summarize this technique below.

Assume the patch particles have Cartesian coordinates $\{\mathbf{p}_i, i = 1...n\}$ and texture coordinates $\{k_i, k = u, v, w, i = 1...n\}$, as described in section II. Now we wish to establish a mapping from Cartesian coordinates in patch space to texture coordinates in texture space, $\Phi_{p\rightarrow t}$. The key idea is to split the mapping into three independent mappings

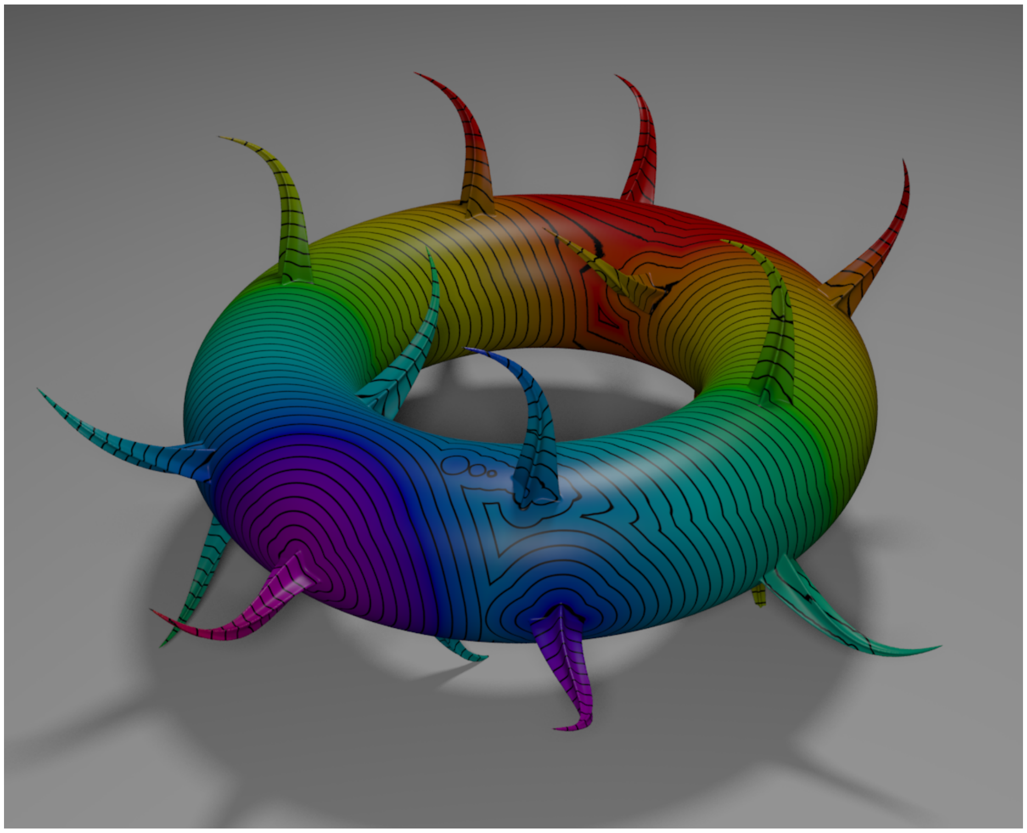
$$\Phi_{p\rightarrow t}(\mathbf{x}_p) = \mathbf{x}_t \Rightarrow \begin{array}{l} \Phi_{p\rightarrow t,u}(\mathbf{x}_p) = x_u \\ \Phi_{p\rightarrow t,v}(\mathbf{x}_p) = x_v \\ \Phi_{p\rightarrow t,w}(\mathbf{x}_p) = x_w \end{array}$$

with each of the texture mapping functions, $\Phi_{p\rightarrow t,k}$, expressed as a sum of weighted *radial basis functions*:

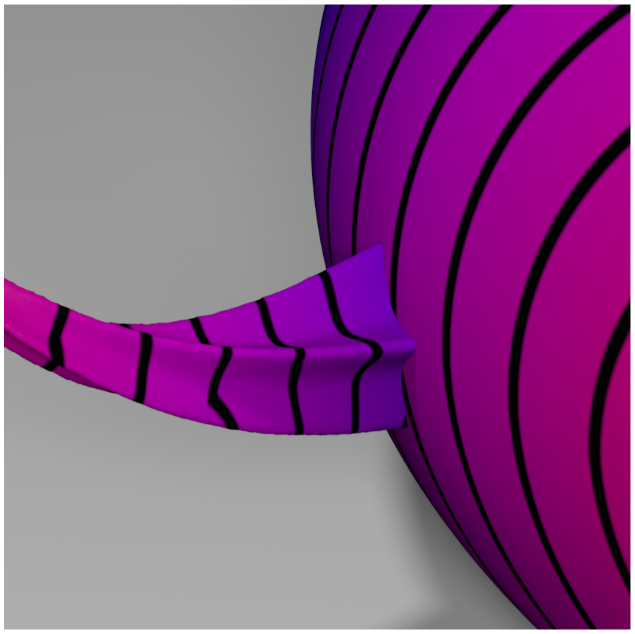
$$\Phi_{p\rightarrow t,k}(\mathbf{x}_p) = P_k(\mathbf{x_p}) + \sum_{i=1}^{n} \omega_{k,i}\varphi(|\mathbf{x}_p - \mathbf{p}_i|), \tag{1}$$

where $\varphi(\mathbf{x}_p)$ is a radially symmetric basis function; n is the number of basis functions; $\mathbf{p}_i$ is the center of the i'th basis;

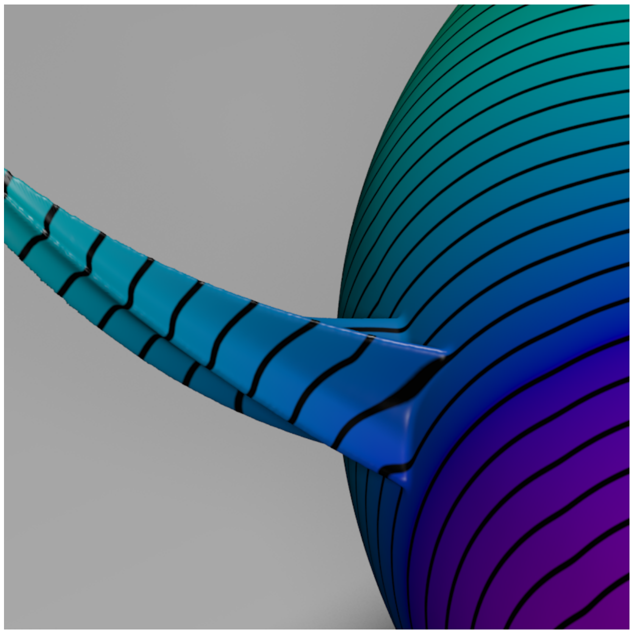
[4]We typically employ tri-linear, an occasionally tri-cubic, interpolation, but essentially any bounded interpolation scheme can be used.



(a)



(b)



(c)

Fig. 11. Mapping a number of spikes onto a torus. The closeups of a single spike shows the difference between not applying the CSG union (b) and applying the CSG union as well as the blending of the intersection (c). Notice the discontinuity in the shading in (b), which is a result of the spike and torus being separate geometries. Merging (and blending) the two surfaces resolves the issue (c).

$\omega_{k,i}$ are the weights for the i'th basis for texture coordinate k; and $P_k(\mathbf{x}_p) = \rho_{k,0}x_x + \rho_{k,1}x_y + \rho_{k,2}x_z + \rho_{k,3}$ is a polynomial spanning the null space of the basis function. Similar to [31], we center a basis function at each patch point.

To find the weights, $\omega_{k,i}$, and polynomial coefficients, $\rho_{k,j} = \{\rho_{k,0}, \rho_{k,1}, \rho_{k,2}, \rho_{k,3}\}$ for each mapping, $k = \{u, v, w\}$, we apply Equation 1 to each of the patch points. Since we already have assigned a k coordinate to each patch point, this leads to a

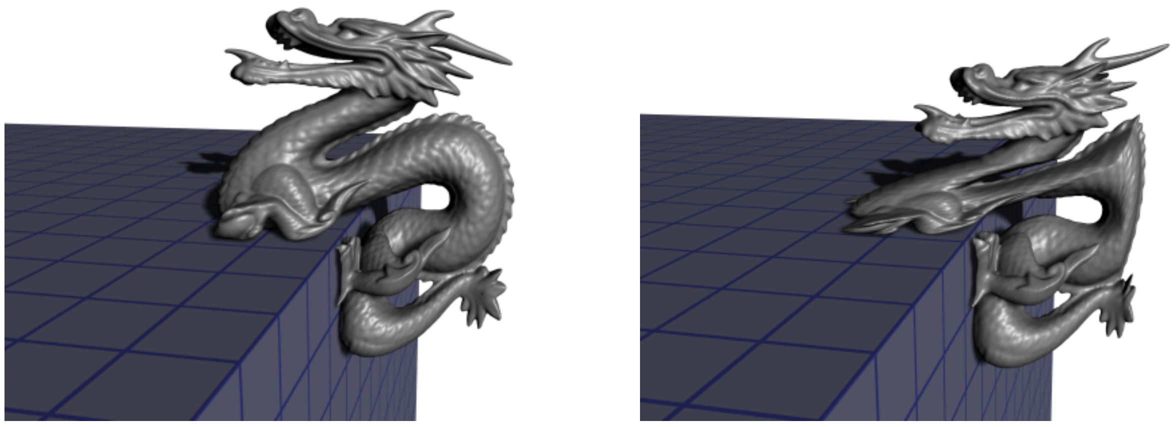


Fig. 12. Mapping a dragon onto a sharp corner using (a) our new geometric texture mapping technique (semi-implicit mapping), and (b) using the technique of [8]. Both mappings were done in less than one second.

linear system of $n+4$ equations with $n+4$ unknowns:

$$\begin{bmatrix} \varphi(|\mathbf{p}_1-\mathbf{p}_1|)+\lambda_1 & \cdots & \varphi(|\mathbf{p}_1-\mathbf{p}_n|) & \mathbf{p}_1 & 1 \\ \vdots & & \vdots & \vdots & \vdots \\ \varphi(|\mathbf{p}_n-\mathbf{p}_1|) & \cdots & \varphi(|\mathbf{p}_n-\mathbf{p}_n|)+\lambda_n & \mathbf{p}_n & 1 \\ \mathbf{p}_{1,x} & \cdots & \mathbf{p}_{n,x} & 0 & 0 \\ \mathbf{p}_{1,y} & \cdots & \mathbf{p}_{n,y} & 0 & 0 \\ \mathbf{p}_{1,z} & \cdots & \mathbf{p}_{n,z} & 0 & 0 \\ 1 & \cdots & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \omega_{k,1} \\ \vdots \\ \omega_{k,n} \\ \rho_{k,0} \\ \rho_{k,1} \\ \rho_{k,2} \\ \rho_{k,3} \end{bmatrix} = \begin{bmatrix} k_1 \\ \vdots \\ k_n \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2}$$

After solving this linear system for each $k=\{u,v,w\}$ the resulting $\{\omega_{k,i},\rho_{k,j}\}$ and are next back-substituted into Equation 1 to compute u,v,w coordinates on the 3D grid in patch space. The resampled texture level set is then simply computed by interpolation in the texture space.

The λ values on the diagonal of the matrix in Equation 2, allow us to control the smoothness of the mapping. As previously mentioned, each particle, $\mathbf{p}_i$ with position $\mathbf{x}_{p,i}$, maps to a specific position in the texture space, $\mathbf{x}_{t,i}$. By adding the λ_i values to Equation 2, we can relax this correspondence leading to the following inequality: $|\Phi_{p\to t}(\mathbf{x}_{p,i})-\mathbf{x}_{t,i}|\leq\zeta_i$, where the constant ζ_i is deducted from λ_i. The larger λ_i is, the larger ζ_i will be. Also, if λ_i is zero then so is ζ_i. As the ζ values increase, the interpolation between the sample values becomes less restricted enabling a smoother interpolation, and thereby also a smoother mapping. For the results in this paper we have typically used two different λ values. Particles on the interface (*i.e.* particles with $w=0$) are assigned small λ values to ensure that the mapping follows the interface closely. These values typically fall in the range 0.001 to 0.01. The remaining points are assigned a larger λ usually between 0.1 and 0.5 to ensure a smoother mapping away from the interface.

Since the implicit mapping uses level set representations for both the texture and the base geometry, we can easily produce a single topologically connected surface by merging and blending the two volumes. This can be achieved with boolean (CSG) operations like union or difference of the two level sets. This in turn simply amounts to a min/max operation of the distance fields followed by a re-initialization in the resulting narrow band. However, the result of boolean CSG operations typically create very visible C^0 discontinuities along the intersection seam. To further address this, we employ the techniques described in [13] that performs localized mean curvature based smoothing in the vicinity of the intersection of the two level sets. This approach allows for direct user control of mean curvature, and thus the smoothness, of the resulting volume. Both the merging/CSG union and the smoothing of the intersection are optional operators applied, if desired, once the mapping is completed. Due to numerical issues, we cannot guarantee that the base surface and the texture will match up exactly. Thus, to ensure a sufficient overlap between the two surfaces required to get a nice blending, we *push* the texture slightly downwards by adding a small offset to the w texture coordinate.

Fig. 11 shows a torus with several spikes mapped onto it using this technique. Figs. 11(b) and 11(c) shows a close up of the intersection of the torus and a single spike, one with the merging and blending performed, Fig. 11(c), and one without, Fig. 11(b). The shading is based on surface distance to the top of the lower left spike. As the objects in Fig. (11(b)) have not been merged into one, the shading breaks down in this example, resulting in a very obvious transition between base and texture geometry.

V. Results and applications

Fig. 12 shows an example of mapping a geometric texture onto an object with a sharp edge. Due to the underlying parameterization of *shell space*, which is based on an offset surface generated by offsetting the base mesh vertices in the direction of the vertex normals, the object mapped using the shell mapping technique of [8], Fig. 12(b), is severely distorted. As our technique allows a guaranteed uniform distribution of the patch points, our mapping, Fig. 12(a), guarantees a smooth mapping, even across such sharp edges. Although the distortion minimization technique presented in [32] can help reduce the distortion in the case of shell mapping, it cannot completely resolve the problem due to the linear interpolation in shell space. The only way to completely resolve this problem is to generate a smoother offset surface, which is

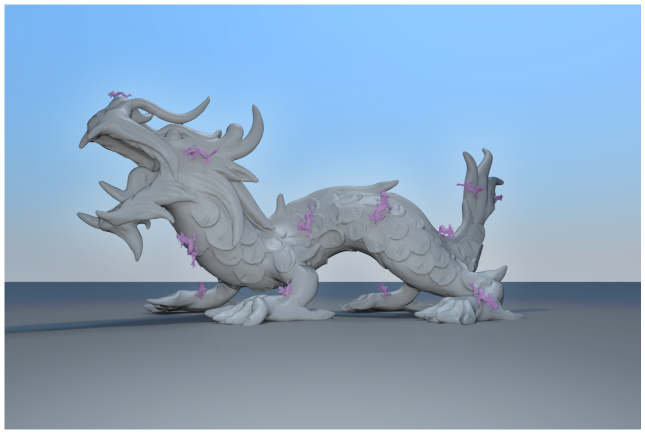


Fig. 13. Mapping a number of small dragons onto a mother dragon using the proposed technique.

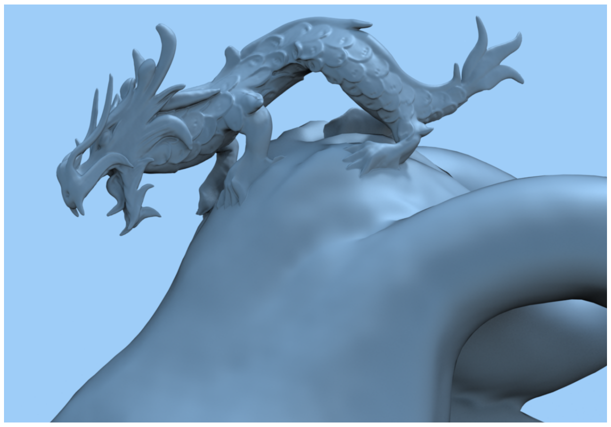


Fig. 14. Closeup of a single one of the baby dragons from Fig. 13. Notice the high level of detail and topological change that results from blending.

exactly what our approach does. As for the performance of the two techniques, both mappings were done in roughly the same time, which is in less than one second.

One major problem with using regularly sampled implicit surfaces is the memory requirements of the 3-dimensional grid, which imposes a problematic limit at high and useful resolutions. This is the primary reason for using the DT-grid, which allows us to use significantly higher volume resolutions. The large dragon in Fig. 13 has an effective resolution of 512x244x350 and all of the 12 *baby* dragons are made using the same resolution.

Although the two mapping schemes presented in sections III and IV produce almost visually identical results in many cases, they are in many ways very different methods offering a different set of features in addition to the obvious difference in the geometric representation of the texture. The most important feature of the semi-implicit method is its speed. Whereas the time complexity of the implicit method scales with the number of particles times the number of voxels in the embedding volume, the semi-implicit mapping is linear in the number of vertices on the texture geometry. The dragon in Fig. 12(a) contains more than 400,000 vertices and was mapped in less than a second using the semi-implicit method. While the semi-implicit method is often capable of producing good results relatively fast, the implicit method offers some distinct benefits. Most importantly, since both the texture and base surface are represented using level sets we can readily produce a simple topologically connected surface by means CSG operations - either prior to a mesh extraction or alternatively during direct ray-casting. Furthermore, we can apply a smoothing operation (see [13]) on the intersection of the base surface and the warped texture, if a smooth intersection with continuous normals is desired. Another advantage of the radial basis function interpolation is that it is significantly less sensitive to the distribution of the patch points. If the base surface has many high frequency features, these features will directly influence the result of an explicit mapping. One the other hand, the implicit mapping allows for direct control of the smoothness through the parameters λ_i entering the linear system in equation (2). By increasing λ_i the implicit mapping will retain the ability to produce a smooth mapping, while still following for lower frequency features of the base surface. Also, while the semi-implicit mapping is only C_0, the implicit mapping allows for multiple orders of continuity, although the exact order is determined by the chosen basis function. In our tests, we have seen the best results when using $f(r) = r$ as our basis function. Still, the implicit mapping is much slower than semi-implicit scheme. Mapping a single model takes from 20-30 seconds for the two latches in Fig. 16, using 280 particles and an embedding volume of 4 million voxels for the small latch, and 660 particles and 5.6 million voxels for the larger. Mapping times vary between 4-5 minutes per baby dragon in Fig. 13 (and Fig. 14) using 3-400 particles and an embedding volume of 20-30 million voxels.

Another benefit of our implicit approach is we can easily map new objects onto previously mapped objects. Fig. 16 shows two latches and several bunnies mapped onto a base surface and a previously mapped bunny. To achieve a similar result, Shell Maps would have to fuse the two bunnies together, generate new uv coordinates, and finally create a new offset surface.

Using the signed distance of the level set function for generating *offset surfaces* offers several advantages. First of all, the further we move away from the base surface, the smoother the offset surface becomes. This means that the influence of high frequency details in the base geometry decreases away from the surface, resulting in smoother looking results, as shown in Fig. 12(a). Previous approaches have employed explicit geometry representations which can lead to problematic self-intersections of the dilated offset surfaces. Consequently these methods have been limited to rather small offsets which in turn only allows for the mapping of small geometric textures. This self-intersection problem is illustrated in Fig. 15, where two offset surfaces are generated from the bunny model using, respectively, level sets and the technique presented in Shell Maps [8]. It should be evident from this simple example that our current method is significantly more robust with surface offsets. The small bunnies in Fig. 16 is an example

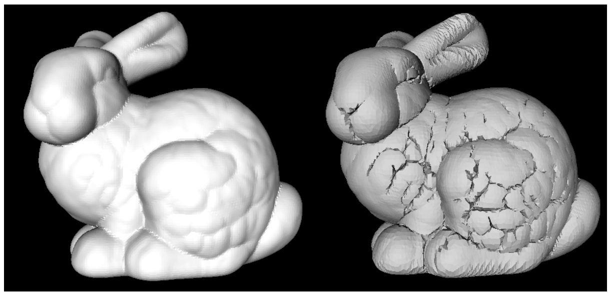


Fig. 15. The offset on the left is a natural result of the level-set's implicit representation. On the right we see an explicit polygonal offset (note the degeneracies in concave regions) using the method proposed in Shell Maps [8].

of mappings using larger offsets (although our method allows for even larger offsets).

VI. Conclusions and Future Work

We have presented fast and flexible techniques for warping and blending (or subtracting) geometric details, in the form of a geometric texture, onto level set surfaces. These techniques are similar in nature to the shell mapping technique, though we have eliminated some of the limitations of the shell mapping approach. Our current approach is based on using implicit geometry, which makes it easy to merge the base and texture geometry into a single topologically connected object as well as smoothing the intersection between the base and texture geometry guaranteeing a smooth surface with smooth normals. Furthermore, our mapping employs a flexible particle based parameterization. As the parameterization is characterized by the distribution of the particles, we can change the parameterization by changing the way the particles are distributed. To demonstrate this flexibility, we have presented three different methods for distributing the particles, including a method that reduces the overall texture distortion.

Although the semi-explicit mapping proposed in this paper is very fast, the implicit mapping is rather slow. The problem is that the speed of the implicit mapping depends, not only on the size of the volume it is being mapped into, but also on the total number of particles defining the parameterization. We are currently considering different approached to address this issue. One idea is to replace the current global radial basis functions with functions that have only local support. Another interesting approach would be to only re-sample the level set of the geometric texture in a local neighborhood of its surface. However, this idea is far from simple and we have currently not been able to device a robust algorithm.

Another interesting idea for future work is to replace the 2D parametrization technique of Pedersen [22] with discrete exponential maps of [23]. The latter approach seems more intuitive and simpler to use from an artist's point of view[5].

[5]For a single patch, our current approach requires the user to place four particles on the surface whereas an approach based on discrete exponential maps (DEM) would require only two. Furthermore, changing the size or orientation of the patch requires us to change the position of all four particles, as opposed to only moving a single point around with the DEM approach.

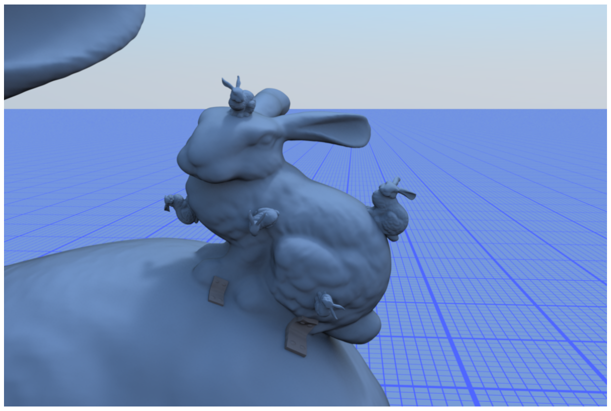


Fig. 16. Recursive mapping: Mapping a bunny onto another bunny, which then again has several smaller bunnies mapped onto it. The middle bunny is "held onto" the bigger bunny using a couple of metal latches that are in fact texture mapped cuboids.

Though we do not claim to have developed flawless techniques for geometric texturing, interactions with people in the visual effects industry confirm our believe that the methods presented in this paper are indeed very useful.

Acknowledgment

We would like to thank Michael Nielsen and other members of the Graphics Group at Linköping University for allowing us to use their software - in particular the DT-Grid implementation. Additional thanks goes to Ola Nilsson for assistance with rendering and Louis Feng for help generating illustrations. Finally we would like to acknowledge our reviews for many good comments that helped to improve our paper.

References

[1] J. F. Blinn and M. E. Newell, "Texture and reflection in computer generated images," *ACM Communications*, vol. 19, no. 10, pp. 542–547, 1976.

[2] J. F. Blinn, "Simulation of wrinkled surfaces," in *Computer Graphics (SIGGRAPH '78 Proceedings)*. ACM Press, 1978, pp. 286–292.

[3] R. L. Cook, "Shade trees," in *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM Press, 1984, pp. 223–231.

[4] J. T. Kajiya and T. L. Kay, "Rendering fur with three dimensional textures," in *Computer Graphics (Proceedings of SIGGRAPH 89)*, vol. 23, no. 3, July 1989, pp. 271–280.

[5] Y. Chen, X. Tong, J. Wang, S. Lin, B. Guo, and H.-Y. Shum, "Shell texture functions," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 343–353, August 2004.

[6] X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum, "Generalized displacement maps," in *Eurographics Symposium on Rendering*, H. W. Jensen and A. Keller, Eds., 2004.

[7] P. Bhat, S. Ingram, and G. Turk, "Geometric texture synthesis by example," in *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. New York, NY, USA: ACM Press, 2004, pp. 41–44.

[8] S. D. Porumbescu, B. C. Budge, L. Feng, and K. I. Joy, "Shell maps," in *ACM SIGGRAPH*, vol. 24, no. 3. ACM, 2005, pp. 626–633.

[9] M. B. Nielsen and K. Museth, “Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets,” *Journal of Scientific Computing*, vol. 26, no. 3, pp. 261–299, 2006, (submitted November, 2004; accepted January, 2005).

[10] S. Osher and J. A. Sethian, “Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations,” *Journal of Computational Physics*, vol. 79, pp. 12–49, 1988. [Online]. Available: citeseer.nj.nec.com/osher88fronts.html

[11] J. A. Sethian, *Level Set Methods and Fast Marching Methods*, 2nd ed. Cambridge, UK: Cambridge University Press, 1999.

[12] S. Osher and R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*. Berlin: Springer, 2002.

[13] K. Museth, D. Breen, R. Whitaker, and A. Barr, “Level set surface editing operators,” *ACM Trans. on Graphics (Proc. SIGGRAPH)*, vol. 21, no. 3, pp. 330–338, July 2002.

[14] S. Mauch, “Efficient algorithms for solving static hamilton-jacobi equations,” Ph.D. dissertation, California Institute of Technology, 2003.

[15] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3D surface construction algoritm,” *Computer graphics*, vol. 21, no. 4, pp. 163–168, July 1987.

[16] F. Losasso, F. Gibou, and R. Fedkiw, “Simulating water and smoke with an octree data structure,” *ACM Transactions on Graphics*, vol. 23, no. 3, Aug. 2004.

[17] B. Houston, M. Nielsen, C. Batty, O. Nilsson, and K. Museth, “Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation,” *ACM Transactions on Graphics*, vol. 25, no. 1, pp. 1–24, 2006.

[18] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones, “Adaptively sampled distance fields: A general representation of shape for computer graphics,” in *Proceedings of SIGGRAPH 2000*, ser. Computer Graphics Proceedings, Annual Conference Series, ACM. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000, pp. 249–254. [Online]. Available: citeseer.nj.nec.com/frisken00adaptively.html

[19] G. Turk, “Texture synthesis on surfaces,” in *SIGGRAPH ’01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM Press, 2001, pp. 347–354.

[20] M. Tarini, K. Hormann, P. Cignoni, and C. Montani, “Polycube-maps,” *ACM Trans. Graph.*, vol. 23, no. 3, pp. 853–860, 2004.

[21] R. Zonenschein, J. Gomes, L. Velho, L. de Figueiredo, M. Tigges, and B. Wyvill, “Texturing composite deformable implicit objects.” [Online]. Available: http://citeseer.ist.psu.edu/zonenschein98texturing.html

[22] H. K. Pedersen, “Decorating implicit surfaces,” in *SIGGRAPH ’95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM Press, 1995, pp. 291–300.

[23] R. Schmidt, C. Grimm, and B. Wyvill, “Interactive decal compositing with discrete exponential maps,” *ACM Trans. Graph.*, vol. 25, no. 3, pp. 605–613, 2006.

[24] F. Neyret, “Modeling, animating, and rendering complex scenes using volumetric textures,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 55–70, 1998.

[25] J. Peng, D. Kristjansson, and D. Zorin, “Interactive modeling of topologically complex geometric detail,” *ACM Trans. Graph.*, vol. 23, no. 3, pp. 635–643, 2004.

[26] K. W. Fleischer, D. H. Laidlaw, B. L. Currin, and A. H. Barr, “Cellular texture generation,” in *SIGGRAPH ’95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM Press, 1995, pp. 239–248.

[27] X. Provot, “Deformation constraints in a mass-spring model to describe rigid cloth behavior,” in *Graphics Interface ’95*, W. A. Davis and P. Prusinkiewicz, Eds. Canadian Human-Computer Communications Society, 1995, pp. 147–154. [Online]. Available: citeseer.ist.psu.edu/provot96deformation.html

[28] S. Hadap, D. Eberle, P. Volino, M. C. Lin, S. Redon, and C. Ericson, “Collision detection and proximity queries,” in *SIGGRAPH ’04: ACM SIGGRAPH 2004 Course Notes*. New York, NY, USA: ACM Press, 2004, p. 15.

[29] H. Zhao, “Fast sweeping method for eikonal equations,” *Mathematics of Computation*, vol. 74, pp. 603–627, 2004.

[30] T. W. Sederberg and S. R. Parry, “Free-form deformation of solid geometric models,” *ACM SIGGGRAPH Computer Graphics*, vol. 20, no. 4, pp. 151–160, 1986.

[31] H. Q. Dinh, G. Turk, and G. Slabaugh, “Reconstructing surfaces by volumetric regularization using radial basis functions,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 10, pp. 1358–1371, 2002.

[32] K. Zhou, X. Huang, X. Wang, Y. Tong, M. Desbrun, B. Guo, and H.-Y. Shum, “Mesh quilting for geometric texture synthesis,” *ACM Trans. Graph.*, vol. 25, no. 3, pp. 690–697, 2006.