# A New Method for Texture Mapping Point-Based Models

**Linbo Bai**

**A Thesis**

**in**

**The Department**

**of**

**Computer Science and Software Engineering**

**Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada**

**December 2006**

*Your file Votre référence*
*ISBN: 978-0-494-28949-5*
*Our file Notre référence*
*ISBN: 978-0-494-28949-5*

# Abstract

## A New Method for Texture Mapping Point-Based Models

Linbo Bai

In recent years, point-sampled geometry is becoming ubiquitous in graphics and geometric information processing. From a computer graphics point of view, the first major challenge in point-based geometry is to render high quality realistic images. A commonly used technique in realistic rendering is texture mapping, which essentially adds surface and/or material property detail in the final stages of rendering the image. The primary focus of the research reported in this thesis is to find suitable solutions to directly map one or more textures onto the surface of geometry represented by points without explicitly converting the surface to polygon mesh or to another geometric surface representation.

Parameterization is the most important step required for adding texture onto the surfaces of objects. In this thesis, a global parameterization method is developed by using level set methods to evolve the concerned surface to a surface with implied parameterization, say a sphere. By tracking the point samples to their final destinations on the sphere, a polar coordinate is assigned to every point in the original model. The user then chooses a few anchor points that map into one or more texture images to yield a simple and flexible procedure to map texture images onto the surface of a point-based model. The method has been implemented using MATLAB and C++ and tested on a number of point-based models.

# Acknowledgement

My sincere thanks to my supervisor Professor S. P. Mudur. His advice and guidance have been invaluable and, his enthusiasm and scientific knowledge both motivating and inspiring. I believe there would have been no possibility of me finishing this thesis without his stimulating suggestions and encouragement.

Very special thanks to Mr. Sushil Bhakar, a doctoral student of my supervisor. It has been a great experience to work with him and share his knowledge in many interesting discussions.

Many thanks to the people in the 3D graphics team. It is a very nice experience to work with them.

Thanks to the computer graphics laboratory of ETH Zurich for sharing their wonderful software: Pointshop3D, it provided me a flexible analysis tool. Thanks to Ian M.Mitchell, who shared the wonderful toolbox of level set methods which provided me with a lot of inspiration.

Finally I want to express my special thanks to my wife Ping and my two sons. It is just because of their love and encouragement that I could finish this work.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# *Chapter 1: Introduction*

A major challenge in the field of computer graphics is to produce a computer-generated realistic image of a digitally represented complex object. Photorealistic [1] rendering is the general name for computational techniques that address this challenge. Traditionally, the surfaces of objects are represented geometrically, typically using suitable algebraic equations. Complex objects however require a piecewise surface representation. The most common representation used is the piecewise linear form, which essentially amounts to approximating a curved object's surface by a polygonal mesh. Given a digital model's information consisting of a geometric representation of the objects, material properties for the objects' surfaces, details of lighting the objects and a camera like view specification, rendering then becomes the computational process of converting this data into an image for a display surface, usually the monitor screen. Rendering of complex models to generate highly realistic images can be computation intensive. Hence various shading techniques have been developed as short cuts to obtain suitable approximations of the realistic image. Another commonly used technique in realistic rendering is texture mapping, which essentially adds surface and/or material property detail in the final stages of rendering the image. With rapid developments in 3D scanners, it has become relatively easy to obtain detailed geometric and color information of the surface of an object in the form of dense point samples. An emerging approach, known by the name of point based graphics, is to handle the surface geometry of a complex object directly as point samples, without explicitly converting the surface into a

[1] Photorealism refers to a style of painting that resembles photography in its meticulous attention to realistic detail.

polygonal mesh or any other piecewise surface form. Clearly one can represent almost all kinds of complex objects, limited only by density of sampling and data sizes due to the large number of samples. While a slew of new techniques have been developed for efficient rendering of point-based models as continuous surfaces, realistic rendering techniques have yet to evolve. In particular, as we shall show in this thesis, it is difficult to map texture onto such discrete representations. Hence the primary objective of the research reported in this thesis is to investigate techniques for mapping one or more textures (provided in the form of digital images) onto the surface of an object represented by point clouds.

## *1.1 Computer graphics rendering pipeline*

Rendering of three-dimensional objects primarily involves the conversion of objects in a three-dimensional scene into a two-dimensional image on the display. In the most popular approach used to date, the computations required to carry out this conversion are best described in the form of a pipeline, as shown in Figure 1.1.

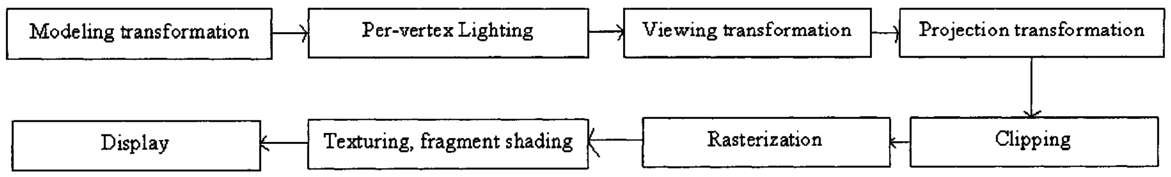


**Figure 1.1**: Computer graphics rendering pipeline

The first step is to model the scene made up of different 3D objects, composed using modeling primitives. Modeling transformations, such as translation and rotation, convert the different modeling primitives from their local coordinate space to the scene space. Then light source based computations are carried out. The viewing and projection transformations convert the 3D scene space geometry into 2D image space. The geometric primitives that fall outside of the viewing volume, as specified by the view parameters, will be not visible and are discarded. Rasterization is the process by which the 2D image-space representation of the scene is converted into raster format and the initial pixel values are determined. At the texturing and fragment shading stage, color computation using texture, illumination, shading etc. are performed. Finally, the pixels are displayed on the monitor screen.

If there is no texture mapping, every object can be drawn either in a solid color, or smoothly shaded between the colors at its vertices. Then if we want to draw a large brick wall, each brick must be drawn as a separate rectangle, and even then the bricks may appear too smooth and regular to be realistic.

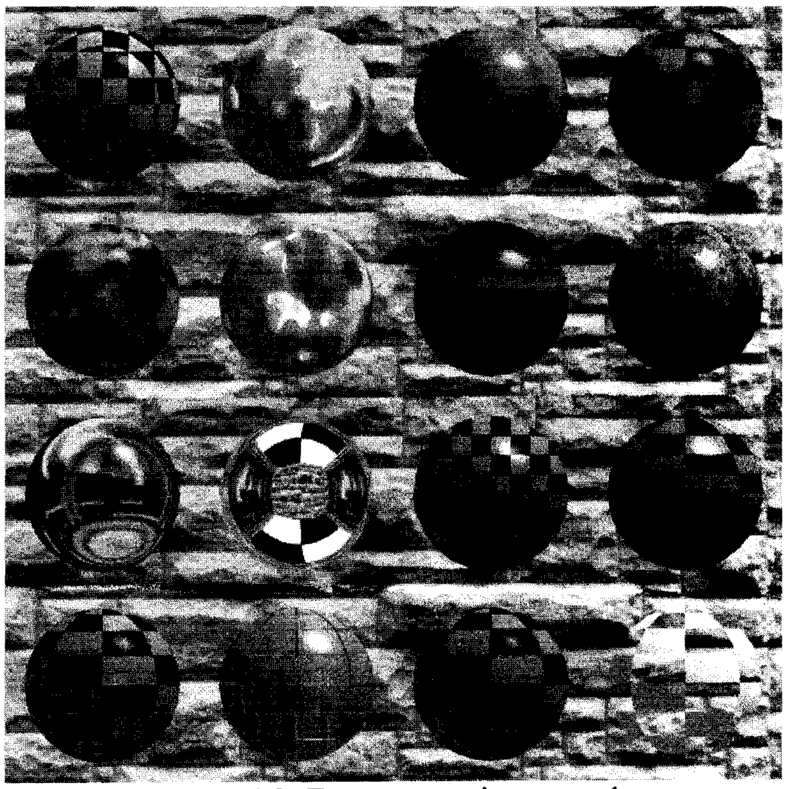


**Figure 1.2:** Texture mapping example
(Figure source: http://www.sli.unimelb.edu.au/envis/texture.html)

In order to generate realistic image, texture mapping is essential. Texture can represent any variation in the surface attributes like color, surface normal, transparency, surface displacement, etc. If the computer generated images are able to capture these complex details they would look more realistic. See, for example, the background as well as the textured spheres in Figure 1.2.

Texture mapping an image on to a 3D object is like gift-wrapping the 3D object with the "texture image sheet". Texture mapping is a kind of shading technique for image synthesis[1] and it also can be defined as the mapping of a function onto a surface in 3D [2]. Mathematically, it is the process of transforming a texture onto a surface. For different modeling methods, texture mapping methods are different.

## *1.2 Modeling methods*

In 3D computer graphics, polygonal modeling is the most used approach for modeling objects by representing or approximating their surfaces using polygons. The most popular kind of polygon used is a triangle.

Triangle meshes are thus the most common surface representation in many computer graphics applications. Triangle meshes are very flexible, since surfaces of any shape and topology can be represented by a single mesh without the need to satisfy complicated inter-patch smoothness conditions. The simplicity of triangle primitive (simplex) allows for easier and more efficient geometry generation and geometry processing algorithms. This is very evident in modern day interactive graphics, where the highly optimized graphics hardware is able to process and render several millions of triangles per second.

NURBS[3], short for non-uniform, rational B-spline, is also a mathematical model used in computer graphics for generating and representing curves and surfaces. This form is popular in computer aided design and engineering applications, wherein geometric precision and continuity are mandatory properties.

## *1.3 Traditional texture mapping*

In this section, we shall use the triangle mesh as the modeling method to introduce texture mapping.

A 2D texture map is simply a rectangular array of data, for example, data, which defines the variations in color, reflectance of the surface, or a function representing the variations of surface normal[4]. Let us consider the image shown in Figure 1.3, to be used as a texture. If we sample this image into a $128 \times 256$ color array, then we can assign each array element a texture coordinate $(u, v)$, where $0 <= u, v <= 1$.

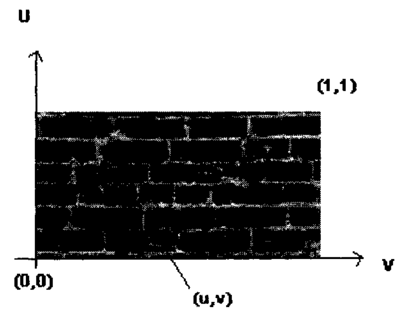


**Figure 1.3:** 2D brick texture

When rendering a surface with a texture, there are two mappings and three spaces concerned: the mapping from texture space to the 3D object space and from 3D object space to 2D screen space, as shown in Figure 1-4. The first mapping is the parameterization of the object surface to build the correspondence between texture space and 3D object space that is to assign a unique texture coordinate to a specific physical point on the object's surface. This is a bijective mapping. Another mapping is called projection, which is determined by the modeling and viewing transformation and is used by the other stages shown in Figure 1.1.

Because triangle meshes maintain topology information, it is straightforward to do parameterization. Texture coordinates are assigned to each vertex. Given the texture coordinates for the vertices of a triangle, the texture coordinates for any point of the surface inside the triangle can be obtained using linear interpolation. The same method applies to a polygon mesh. More complex piecewise surface representations, like NURBS, have their inherent parameterization that can be used as texture coordinates.

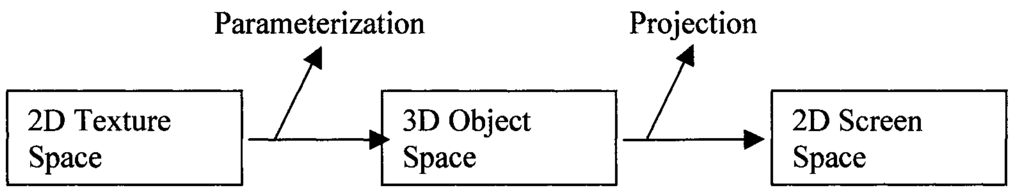


**Figure 1.4:** The pipeline to render textured surface

It is well established that texture mapping adds much detail to a scene to make it look realistic, while requiring only a modest increase in rendering time.[1]

## *1.4 Point-based graphics*

Even though the triangle mesh is the most popular modeling method, it has limitations and disadvantages in some special applications, particularly for deformable shapes and when complex shapes have to be represented accurately leading to an explosion in the number of triangles needed. Most algorithms working on triangle meshes require maintaining consistency of topological information. As a consequence, manifold-extraction or topology cleanup steps are necessary for mesh generation methods. Maintaining the topological consistency throughout the mesh-processing pipeline makes these algorithms sometimes significantly more complicated. Dynamic mesh connectivity is one example where frequent topology changes occur because the mesh is locally restructured in order to avoid too much stretching after extreme deformations.

The overhead of managing, processing, and manipulating large triangle-mesh connectivity information has led many researchers to question the future utility of the triangle as the fundamental graphics primitive [5].

In recent years point-based geometry has gained increasing attention as an alternative surface representation, wherein the object's surface is simply represented using an appropriate sample of surface points. There are mainly three kinds of topics in this field:

- **Acquisition**

  This is the process to generate the cloud of points. 3D photography [6] and 3D scanning are widely used techniques to obtain a digital representation of physical models.

- **Processing and modeling**

  In order to do further processing, it is needed to reconstruct the object's surface from the cloud of points. Approximating a cloud of points by a surface is a well-researched area [7]. Similarly, fitting a point cloud with a triangle mesh, piecewise NURBS or quadric mesh has also been extensively researched [8]. However, the main objective of point-based graphics is to be able to carry out all required processing and rendering directly with sampled points, without explicitly pre-processing the point cloud into a surface representation. There are a number ways proposed to dynamically reconstruct in a localized manner, the object's surface from sample points by looking only at a neighboring set of samples [9].

- **Rendering**

  This stage is to display a geometric model on a screen or printing device.

Most modern day 3D digital photography and 3D scanning systems acquire both the geometry and appearance of complex, real-world objects. These techniques generate huge volumes of point samples [6]. Levoy and Whitted[10] were the first to propose the use of point as a display primitive and a universal meta-primitive for object representation. Subsequently various researchers have presented rendering methods using point primitives [11,12,13,14]. Pfister and Zwicker proposed using surface elements (surfels) as the rendering primitives in [15]. Surfels are point primitives without explicit connectivity and have position, texture color, normal, and other attributes. An interactive system for point-based editing – Pointshop3D – was introduced by M.Zwicker et al. [16].

Because point-based representation neither has to store nor maintain globally consistent topological information, it is more flexible compared to triangle meshes when it comes to handling highly complex or dynamically changing shapes.

## *1.5 Texture mapping on point-based graphics*

Because point-based representation does not maintain global topological information, the traditional parameterization method cannot be used and new mapping methods, especially new parameterization methods, need to be designed.

Zwicker et al. in [16] have proposed the technique of minimum distortion parameterization of surface patches by defining a discrete version of the objective function. In Pointshop3D, users can display and edit the point-based models, and they are allowed to select a surface patch to do texture mapping. The texture mapping quality depends on the correspondence of feature points and the parameterization is not guaranteed to be bijective. Other methods have also been proposed. These include the methods by Haitao Zhang using point parameterization and point neighborhood matching methods[17], and by Alexa et al. [18] using direction fields. Another popular way to map texture onto point based geometry is to generate triangular mesh and apply existing methods for the meshes. Martin Wicke introduces an algorithm to convert point-sampled objects to textured meshes in [19]. They use Cocone and Tight Cocone algorithms [20] [21] to generate triangles, and simplify the triangulation by merging closed triangular regions. Then EWA splatting algorithm [11] is used to do patch texture mapping. In [22], Guo XiaoHu etc. use Global Conformal Parameterization method under the assumption that the point clouds are uniformly and well sampled, but as we will see later their global parameterization is not well suited for texture mapping. All the above methods are reviewed in detail in the next chapter covering both advantages and disadvantages of individual methods.

## 1.6 Research objectives and methods

This thesis is primarily concerned with texture mapping 2D images onto 3D closed objects represented by point samples. So the objective of this thesis is as follows:

- To find a solution to globally parameterize the surface without explicitly converting the point cloud into a triangle mesh or any other surface representation such as NURBS.
- To explore the suitability of this global parameterization method to texture map 2D image on to 3D object's surface
- To set up a test environment using public domain point based rendering software, PointShop3D, for carrying out experiments with texture mapping of point based models.
- To design and develop graphics software programs implementing the proposed global parameterization solution and to handle texture mapping of point-based model represented in SFL format, which is used in Pointshop3D.

Traditionally in order to do global parameterization, we need to know the connectivity and topology information of the object. As pointed out earlier, point-based models do not have such information available. But if we know the object is a sphere, or cube or other such well-studied surface, there is implied topology information that can be used to provide us with a suitable parameterization. This could work if we can create a correspondence between the surface of object and another such surface, say, sphere. That means we need to build a mapping from the object's surface to the sphere's surface. Level set methods [23] [24] provide us powerful tools to evolve the surface of any complex object. The methodology we have followed is the following.

- **Convert the SFL format to implicit function, for example, signed distance function to represent the surface of object**
- **Using level set methods, evolving the surface to surface of sphere, tracking the movement of surface points and keeping consistent neighborhood for each point.**
- **Use spherical parameterization and linear interpolation method for the final texture mapping**
- **Create texture mapped point based model in SFL format for Pointshop3D software to render the texture-mapped surface.**

## *1.7 Thesis outline*

The rest of this thesis is organized as follows:

- ***Chapter 2 : A Comprehensive Review of Surface Parameterization for Point Models***

In this chapter, the parameterization methods for texture mapping point-based models currently in use are introduced, and their advantages and disadvantages are discussed. Then the motivation for our proposed method is presented.

- ***Chapter 3: Background on Surface Representation Techniques***

  In this chapter, different methods of surface representation, implicit function, explicit function and signed distance function (SDF), to represent the curve or surface are described, and some basic geometry variables are introduced. The SFL file format is described and finally, a method for converting SFL model to SDF is presented.

- ***Chapter 4: Global Parameterization of Point Based Models by Evolving the Interface***

  In this chapter, the motion of interface (points on the object surface separating interior and exterior) is first defined and two common solutions for evolving the interface: explicit technique and implicit technique are introduced. Following this, the application of level set in our method is introduced.

- ***Chapter 5: Numerical Implementation***

  In this chapter, the grid creation, the approximation method to partial derivative and curvature, speeding, and re-initialization are introduced. Finally, we briefly describe significant aspects of the software we have developed.

- ***Chapter 6: Conclusion and Future Work***

  In this chapter, the conclusion is drawn and some aspects that could be further improved are listed.

- ***Appendix A:***

  Basic data structure used in our software in code format is appended.

# *Chapter 2: A Comprehensive Review of Surface Parameterization for Point Models*

With the development of point-based graphics, and especially with the introduction of point as the universal geometric display primitive, many researchers have started to devise new techniques for mapping texture on to the point-based objects. As we have seen in the previous chapter, a key issue in texture mapping an image onto a 3D object's surface is that of deriving a suitable surface parameterization that provides a bijective map between the texture coordinates in 2D and the surface in 3D. In this chapter, surface parameterization methods, particularly, methods for point-based models are introduced in detail and the main motivation for the method proposed in our research is also presented.

## *2.1 Surface parameterization*

Let us recall that surface parameterization is needed to define a correspondence between texture space in 2D, say with coordinates u and v, and 3D object space, say with coordinates x, y, z. In short, we need the mapping $(u,v) \to (x,y,z)$.

A naive method to build such mapping is to indicate for every point on the object's surface a unique texture coordinate pair. However, for a continuous surface this is not practical. The most common approach is to indicate several surface points with texture coordinates such that both the surface points and the texture points form closed (polygonal) regions in their respective spaces, and other texture coordinates are computed based on the given information. For example, if we did this specification for 3 points, then we would have a triangular patch on the surface of the object, and a corresponding triangle in texture space, thus providing a one-to-one mapping between surface points within the triangular patch and the color values within the texture space triangle.

The ideal parameterization is the bijective mapping and conformal mapping used in texture mapping [29]. Bijective mapping means the map is one to one and inversive, like the mapping X in Figure 2.1. For any point in set A there is only one corresponding point in the set B; inversely, for every point in set B, we can find one and only one corresponding point in set A.

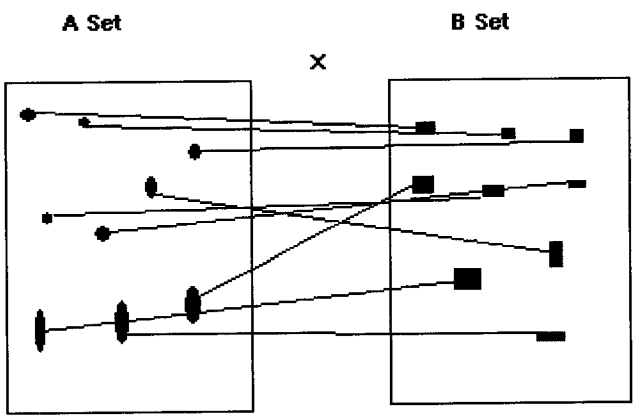


**Figure 2.1**: Illustration of bijective mapping

Conformal mapping is also called angle-preserving mapping. If the mapping is from surface S to $S^*$, the mapping is conformal if the angle of intersection of every pair of intersecting arcs on $S^*$ is the same as that of the corresponding pre-images on S at the corresponding point. In this way, we say there is no distortion caused by parameterization. Many techniques have been developed to compute conformal parameterizations [26] [27]. But, most parameterization methods are unable to avoid some distortion [25].

In general, parameterizing an arbitrary 3-D shape to map into a 2D texture space is very complex. The most popular method is to create a triangle mesh to approximate the shape. Because the geometric connectivity information in a triangle mesh is completely known, the parameterization is easy and can be done using piecewise linear interpolation functions. In Figure 2.2, we are given a 2D object space triangle with vertices A, B, and

C, and corresponding texture coordinates as $(u_1, v_1)$ $(u_2, v_2)$and $(u_3, v_3)$. Then our goal is to get $(u, v)$of any point P in the object space. In Cartesian coordinates system, we have following equation:

$$P = aA + bB + (1 - a - b)C \qquad (2\text{-}1)$$

It is easy to solve this equation to get the coefficients $a$ and $b$. Then using the following equation we can get u and v:

$$u = au_1 + bu_2 + (1 - a - b)u_3$$
$$v = av_1 + bv_2 + (1 - a - b)v_3 \qquad (2\text{-}2)$$

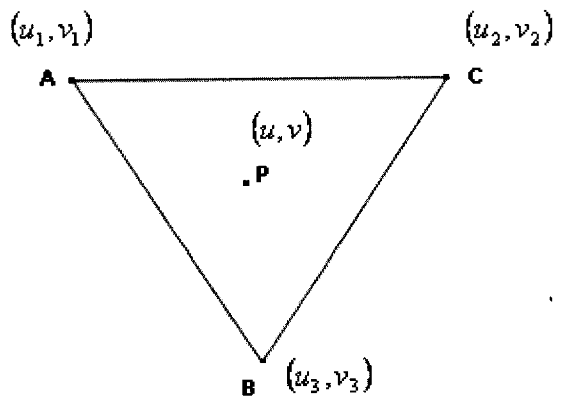


**Figure 2.2**: Interpolation of texture coordinates of P within the triangle ABC

However, for point-based models, lack of continuity of surface makes it difficult to do parameterization, because we have no topological information to use. In the following sections, we discuss a number of parameterization methods proposed for point-based models and described in the literature.

## 2.2 *Related surface parameterization methods*

### 2.2.1 *Direction Fields over Point-Sampled Geometry*

In [18] Alexa et al. present an algorithm to use direction fields over point-sampled geometry to do parameterization.

In this method the first step is to specify initial directions. This specification of initial direction can be done by user so as to position an appropriate number of discontinuities in the direction field, like sources and sinks, explicitly with the point set. The initial directions can be also assigned by computing surface features such as principal curvature (Figure 2.3 a).

After this initialization, a number of iterations will be performed in which a point is selected and its direction is modified based on its neighborhood until the expected smoothness of direction fields is found. The new direction is calculated as the sum of the directions of all neighborhood points weighted using the Gaussian coefficients.

The calculated directions can be used to generate texture information for the point set surface. First one initial point is assigned a random color from the texture. Then a point with the shortest distance to the initial point is selected. The color for this new point is calculated by generating a texture neighborhood of the colors already generated and by finding the point in the original texture that has the smallest color difference compared to the neighborhood. This step is repeated for all other points.

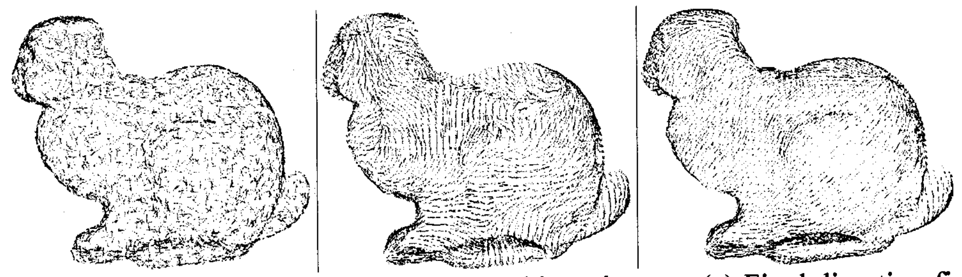

(a) Initial direction fields (b) after several iterations (c) Final direction fields

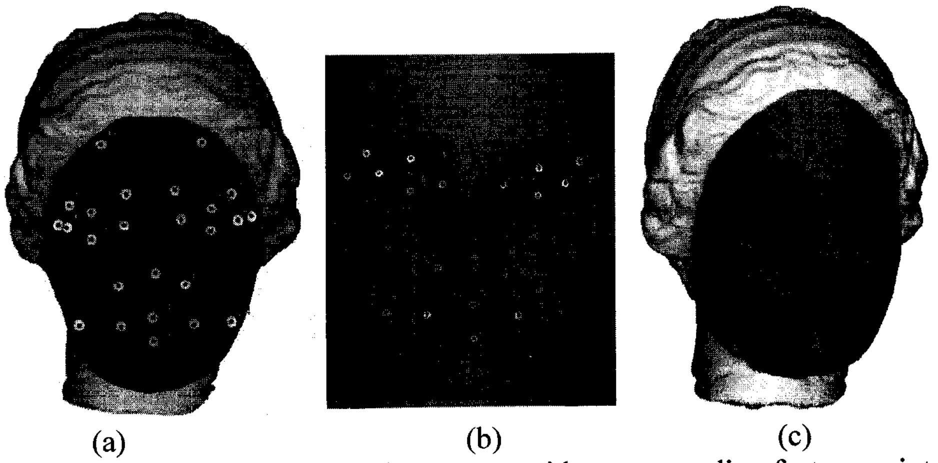
**Figure 2.3**: Direction fields computation procedure (Figure source [18])

In this method, initial direction fields completely influence the parameterization quality. Incorrectly specified initial directions result in poor texture mapping. Thus, considerable user skill and intervention are required to properly position the sources and sinks so that a desired quality of textured image can be obtained.

### *2.2.2 Pointshop3D*

Zwicker et al. have introduced the public domain Pointshop3D software in [16]. Pointshop3D is a software system for interactive shape and appearance editing of 3D point-sampled geometry. Similar to 2D pixel editors, say, like Photoshop®, it provides conventional editing tools and functions, such as patch selection, eraser, patch parameterization, paint brushes etc.

The algorithm which Pointshop3D uses for texture mapping is known as minimum distortion parameterization. Before parameterization, using a dedicated selection tool the user has to select a patch on the surface and specify several feature points (Figure 2.4). Then the user also has to specify corresponding feature points on the texture.



(a) Patch selection and feature points.(b) Texture with corresponding feature points.
(c) texture mapping result
**Figure 2.4**: Texture mapping in Pointshop3D (Figure source [16])

The algorithm is based on an objective function derived from [28] and is briefly described below.

Suppose a continuous parameterized surface patch $X_S$ is defined by a one-to-one mapping: $X:[0,1]\times[0,1]\rightarrow X_S \in IR^3$. Each point $\mathbf{u}$ = $(u,v)^T$ in $[0,1]\times[0,1]$ represents a point $\mathbf{x} = (x,y,z)^T$ on the surface:

$$\mathbf{u} \in [0,1]\times[0,1] \Rightarrow X(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix} = x \in X_S \qquad (2\text{-}3)$$

The minimum distortion means that the parameterization optimally adapts to the geometry of the surface. Additionally, the user is able to specify a set M of point correspondences between the surface points $x_j$ and texture points $p_j$, to control the mapping. Then the objective function is:

$$C(X) = \sum_{j \in M} \{X(p_j) - x_j\}^2 + \varepsilon \int_{\Omega} r(u)du \qquad (2\text{-}4)$$

Where
$$r(u) = \int_{\theta} \left( \frac{\partial^2}{\partial r^2} X_u(\theta, r) \right)^2 d\theta \qquad (2\text{-}5)$$

And
$$X_u(\theta, r) = X\left( u + r \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} \right) \qquad (2\text{-}6)$$

By computing the minimum of function (2-4) the parameterization can be approximated by a function.

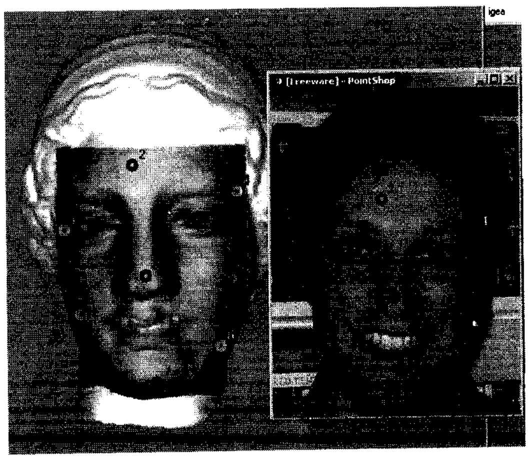


**Figure 2.5**: Texture mapping results in Pointshop3D with few of feature points

If all points lie in a plane, and at least three or more points obeying an affine mapping are given as fitting constraints, the resulting parameterization will be an affine mapping. However, this algorithm does not guarantee a bijective mapping. This algorithm can not deal with patches of high curvature as it will generate large distortion. And the quality of texture mapping is very dependant on rather careful subdivision of surface into patches by the user, and also specification of appropriate point correspondences between surface points and texture. For example in Figure 2.5, if only a few constraint points are indicated, the quality of texture mapped image is considerably poorer.

### 2.2.3 *Fast Hybrid Approach for Texturing Point Models*

Haitao Zhang et al. present a hybrid technique based on point parameterization method and neighborhood matching method to map texture onto point models [17].

The point parameterization method uses the basic idea of flattening the point model's surface into one or more 2D patches under a certain distortion criteria; and the image texture is then mapped onto these patches. Lastly, alpha blending is used to minimize the discontinuity in the gaps between the patches.

The second method is based on neighborhood matching where a color is assigned to each point by searching the best match within an irregular neighborhood. It uses the k-neighborhood[29] method, that is, it chooses k number of closest points to make up the neighborhood.

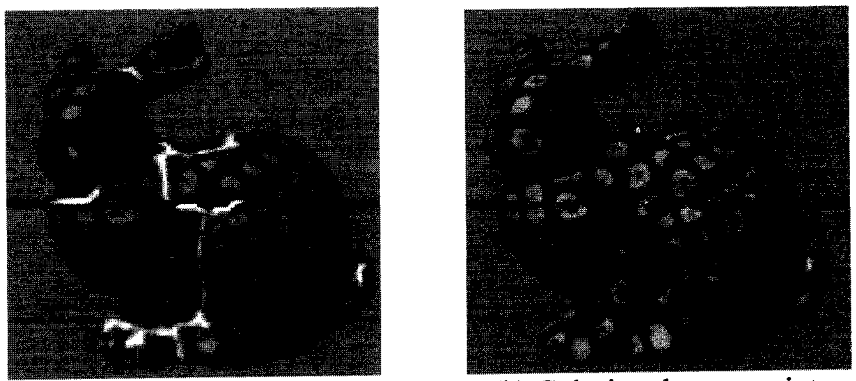

(a) After patch mapping (b) Coloring the gap points

**Figure 2.6**: The hybrid method procedure for texture mapping (Figure source [17])

The hybrid technique applies the point parameterization method first to texture the patch points, followed by the point neighborhood matching method for coloring the uncolored gap points (*Cf* Figure 2.6).

For this method, when doing the first step, the quality of the resulting textured image is decided mainly by how well the distortion criteria are defined. If the criteria are loose, the distortion will be bigger; otherwise, more patches will be created and the discontinuity will be increased. Thus guaranteeing a good texture map again requires considerable skill and experimentation.

## *2.2.4 Global Conformal Parameterization*

In [22] Guo et al. present a global conformal parameterization method to do meshless thin-shell simulation. This method is derived from the corresponding methods used on mesh surface which can deal with objects of any genus [30]. The basic idea is to find holomorphic 1-form of the surface: $(w_1, w_2)$. $w_1$ and $w_2$ are zero curl and zero divergence; in addition, $w_1$ and $w_2$ are conjugate to each other. After getting the holomorphic 1-form, the surface can be mapped to the parametric plane by integration.

As can be seen in Figure 2.7, the bunny model is of genus 0. Three points, two at the ear tips and one at the bottom, are selected as points on the opened boundary. The fair Morse function is used to analyze the topology information of the point-based model, and the holomorphic 1-form can be obtained based on the boundary and topology information. Then many little surface patches are created and represented by $(w_1, w_2)$ as shown in Figure 2.7 (c) shows.

As the patches represented by $(w_1, w_2)$ are automatically created, it is difficult to create the correspondence between texture points and the surface points. In addition, different boundaries will generate different parameterization. Further, it assumes the point surface is sufficiently and regularly sampled. However, as is well known and also discussed by the authors in their paper, the sampling issue is far from trivial.

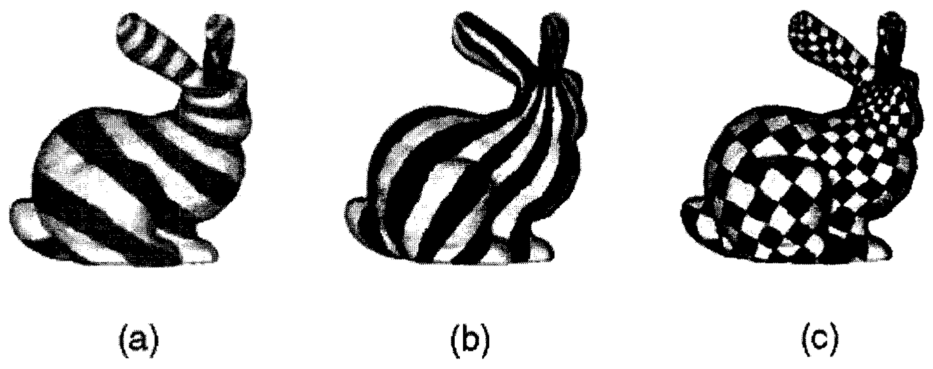


**Figure 2.7**: The global conformal parameterization of the bunny model (Figure source [22])

### *2.2.5 Comparison*

A comparative analysis of the above four methods is provided in Table 2.1.

| **Methods** | **Advantages** | **Disadvantages** |
|---|---|---|
| Direction Fields | Smoothness in parameterization | Very dependent on the initialization and the positions of sinks/sources; |
| Pointshop3D | Easy to use; high quality in a small patch; fast | Bigger patches lead to larger distortion; the mapping quality is dependent on the number of feature points; discontinuities exist between patches |
| Hybrid method | No discontinuity between patches | Very dependent on the distortion criteria; |
| Global conformal parameterization | Global parameterization of surface | Needs well sampled model and difficult to create the correspondence between 2D texture and 3D surface |

**Table 2.1**: The comparison of different parameterization methods

## *2.3 Motivation of our parameterization method*

In summary, when we consider the above methods, the main issue with the first three parameterization methods discussed above is that the parameterization is performed locally with user defined patches or user defined discontinuities in the case of direction fields. Delineating patches is entirely the responsibility of the user. Similarly defining sources and sinks, and giving the initial direction field are entirely the responsibility of users. All these require considerable manual skill, care and intervention, in order for the texture mapped rendering to produce acceptable results. The main problem is that the correspondence map between a patch and texture space are independent of other patch-texture maps. Unless adequate care is taken, discontinuities in texture at shared/disjoint boundaries of patches can be very disconcerting. While global parameterization can help avoid these kinds of problems, Guo's conformal global parameterization is unsuitable for texture mapping purposes, as it breaks up the planar parametric domain into small patches based on flatness and assigning suitable texture coordinates to each of these small automatically created patches would be a nightmare.

However Guo's method inspires us to find a global parameterization method which is more suited to texture mapping.

There is no explicit geometric connectivity information in point-based representations. However, if the object is a well known surface for which the parameterization of any point on the surface is implied, like, say a sphere in 3D or a circle in 2D, and the center position and radius are known, then even if the model is represented just by sample points we have enough geometric information to do global parameterization. In this case, a global bijective mapping can be created. This suggests to us that evolving the given object's surface and converging its surface to the surface of a sphere will enable us to easily use this geometric property. Of course, the evolution should keep the geometric distribution of points within acceptable distortion.

Level set theory provides us powerful tools to evolve an object into a desirable target surface. Level Set Methods [31][32] use a numerical technique which can track the evolution of interfaces. This technique has a wide range of applications, including problems in fluid mechanics, combustion, and manufacturing of computer chips, computer animation, image processing, etc. For detailed description of the level set method that we have used, please refer to Chapter 4.

Therefore, in our method for global parameterization, we try to evolve the target object's surface to another surface of a well known object. Then, using the implied geometric information we do global parameterization and use it for texture mapping an image onto the surface of a point based model.

# *Chapter 3: Background on Surface Representation Techniques*

In order to be able to correctly deal with point based models, say to process and render the object, a suitable representation for the object's bounding surface is essential, even if this surface is not explicitly computed and stored as part of the object's data structure. There are two common approaches used to represent the bounding surface: explicit and implicit functions. In this chapter we will introduce and compare these two approaches from the point of view of their suitability for various processing and rendering operations on point-based models. In particular, signed distance function as a kind of implicit function is the one that we have used in our work as it has many desirable properties. This will also be discussed in detail, including an algorithm for computing a discrete representation for point-based models.

We would like to recall that in our system, we use the Pointshop3D software to render 3D surface represented by points. Pointshop3D uses a file format known as SFL format to store the point-based models. We too have chosen the point model in SFL format as the input of our implementation. Hence, towards the end of this chapter we introduce the SFL format and the process for transformation of SFL to discrete signed distance function.

## *3.1 Explicit and implicit function forms*

In this thesis we shall concern ourselves only with closed objects. The bounding curve or surface of every closed object divides the 2D space or 3D volume into two parts: an interior portion and an exterior portion. The surface separating the interior from the exterior can also be called an interface. For example, in Figure 3.1, we use a 2D interface, a curve, to describe this; and this holds for a surface in 3D space. The circle divides the 2D space into two portions $\Omega^+$ and $\Omega^-$.

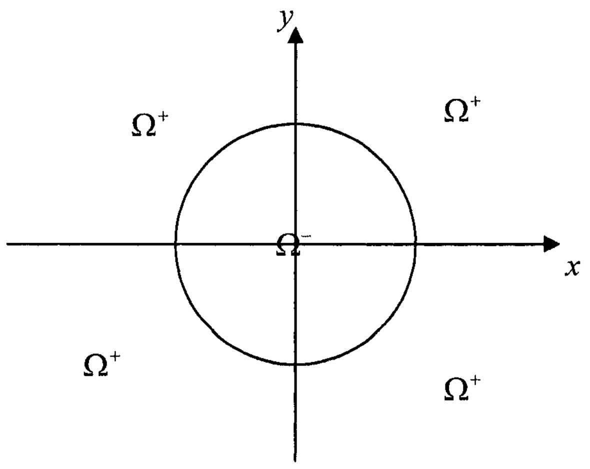


**Figure 3.1:** 2D interface

In order to numerically represent the two portions, a function $\phi$ needs to be defined. Normally we can have following definition:

If $\phi(x,y) < 0$, then point $(x,y)$ belongs to interior potion $\Omega^-$;
If $\phi(x,y) > 0$, then point $(x,y)$ belongs to exterior potion $\Omega^+$;
For $\phi(x,y) = 0$, the point $(x,y)$ belongs to the interface $\Omega$.

There are two ways to define the interface: explicit function and implicit function.

### *3.1.1 Explicit function*

To represent an interface with explicit function, it is need to specify all the points on the interface. The common way is to use parametric functions for each of the coordinates.

For the circle in Figure 3.1, the curve's explicit function is given by the two equations:

$$\begin{aligned} x &= r * \cos(\theta) \\ y &= r * \sin(\theta) \qquad (3\text{-}1) \end{aligned}$$

Where, $\theta$ is the polar coordinate within the real number range given by $(0, 2\pi)$, and $r$ is the radius of the circle.

Similarly, for a sphere surface, an interface in 3D, the explicit function can be defined:

$$\begin{aligned} x &= r * \cos(\alpha) * \cos(\theta) \\ y &= r * \cos(\alpha) * \sin(\theta) \\ z &= r * \sin(\alpha) \end{aligned} \qquad (3\text{-}2)$$

Where, $\alpha$ and $\theta$ are the polar coordinates with ranges given by $(0, \pi)$ and $(0, 2\pi)$ respectively, and $r$ is the radius of the sphere.

### *3.1.2 Implicit function*

Alternatively, the interface can also be represented in the implicit function form. For the circle in Figure 3.1, we define one such function as:

$$\phi(\vec{x}) = x^2 + y^2 - r^2 \qquad (3\text{-}3)$$

Where, $r$ can be any positive number.

We define the zero isocontour as the interface, which means each $\vec{x}$ denoting a point in the space that satisfies $\phi(\vec{x}) = 0$ belongs to the interface. For unit circle's interface, r is equal to 1.

Similarly, for a sphere surface, the implicit function can be defined as:

$$\phi(\vec{x}) = x^2 + y^2 + z^2 - r^2 \qquad (3\text{-}4)$$

And the interface can be defined as the zero isocontour.

Here, there is no loss of generality defining zero isocontour as the interface. The reason is as follows. If we wish to choose $\alpha$ isocontour as the interface:

$$\phi(\vec{x}) = \alpha \qquad (3\text{-}5)$$

Then a new function can be defined: $\tilde{\phi}(\vec{x}) = \phi(\vec{x}) - \alpha$ (3-6)

So the zero isocontour of $\tilde{\phi}(\vec{x})$ is identical to the $\alpha$ isocontour of $\phi(\vec{x})$.

From the implicit function, the interior and exterior portion can also be defined. The interior points consist of $\vec{x}$ which satisfy

$$\phi(\vec{x}) < 0 \qquad (3\text{-}7)$$

The exterior portion is composed of points $\vec{x}$ which satisfy

$$\phi(\vec{x}) > 0 \qquad (3\text{-}8)$$

Based on above introduction, we can say that the explicit function is of one-dimension less than the space represented. For example, the explicit function of curve depends only on one parameter, and the explicit function of 3D surface depends on two parameters. On the other hand, the implicit function used to represent the interface has the same dimension as the space concerned. In a later chapter, we will see that by implicit function we have more global geometric information available than the explicit function, and this is very important in the evolution of the object's surface using level set method.

## *3.2 Discrete Forms for Interface Representation*

Usually it is difficult to give an implicit function or explicit function for any general curve or surface of complicated shape. Hence, where acceptable, it is more convenient to approximate the interface using discrete representations.

First let us consider the interface discretization for an explicit function. Consider equation (3-1); the continuous parameter $\theta$ in this equation can be discretized into a finite number, say, $n$ points:

$$\theta_0 < \theta_1 ... \theta_{i-1} < \theta_i < \theta_{i+1} < ... < \theta_n$$

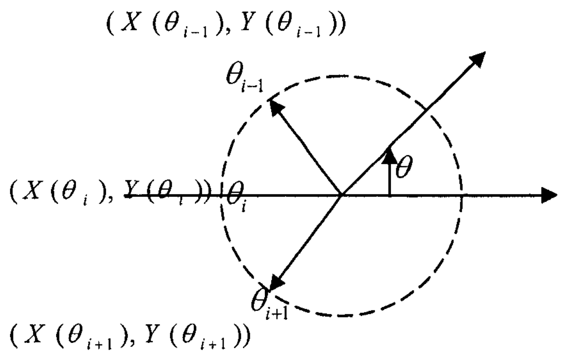
Then the interface (3-1) is approximated by $\vec{x}_0 .. \vec{x}_1 .. \vec{x}_{i-1} .. \vec{x}_i .. \vec{x}_{i+1} .. \vec{x}_n$, where

$$\vec{x}_i = (x, y) = (r * \cos(\theta_i), r * \sin(\theta_i))$$

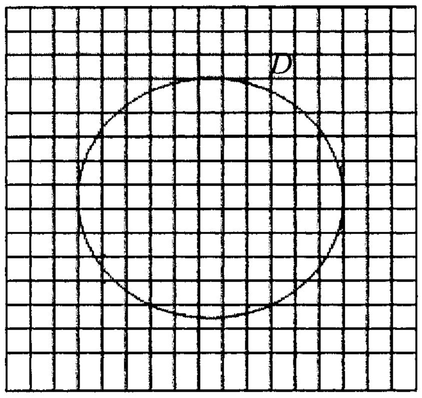
As the number of discrete points in the parameter space is increased, so is the accuracy of approximation of the two dimensional curve. The explicit discretization of a circle is shown in Figure 3.2 (a).

The implicit representation given in equation (3-3) can be stored with a discretization as well, except now, one need to discretize all of $\Re^2$, which is impractical, since it is unbounded. Instead, we discretize a bounded subdomain $D \subset \Re^2$. Within this domain, we choose a finite set of points $(x_i, y_i)$ for $i = 1..N$ to discretely approximate the implicit function. The set of points is called a grid.

There are many ways of choosing the points in a grid, and these lead to a number of different types of grids, for example, unstructured, adaptive, curvilinear grids. The most popular grids are Cartesian grids defined as $\{(x_i, y_j) | 1 \le i \le m, 1 \le j \le n\}$. In our system, in order to be able to use a simple data structure to store the grid point's value, we use uniform Cartesian grid. This is shown in Figure 3-2 (b).


$(X(\theta_{i-1}), Y(\theta_{i-1}))$
$\theta_{i-1}$
$\theta$
$(X(\theta_i), Y(\theta_i))$
$\theta_i$
$\theta_{i+1}$
$(X(\theta_{i+1}), Y(\theta_{i+1}))$

(a) Explicit discretization
To approximate the circle using the finite number of values computed based on discrete $\theta$



(b) Implicit discretization
The grid nodes inside the circle have negative level value; grid nodes outside have positive value; a grid node is on the interface if its level value equals 0

**Figure 3.2:** Explicit and implicit discretization

Similarly, for equations (3-2) and (3-4), the discretization can be done, except that the explicit function is discretized in two-dimensional parameter space and the implicit function is discretized in three dimensions.

In both explicit and implicit discretization, we do not have the complete and exact location of the interface in the domain space. Instead, they both only give information at some sample points. All other points on the interface have to be computed from the given sample points. Usually this is done by using a suitable combination of searching and interpolation.

Even though implicit representation requires one-higher dimension space, we will see that for our purposes, it is easier to work with than explicit representation. This is because when using explicit discretization the connectivity of all the sample points needs to be explicitly recorded and maintained. This is easy for the two-spatial dimensional explicit discretization, because the connectivity is implied by the order of the parameter values. However, it is not at all straightforward to record and maintain the connectivity in three spatial dimensions. We need to choose an optimal number of points on the two-dimensional surface and record their connectivity. When the exact surface and its connectivity are known, it is simple to tile the surface with triangles whose vertices lie on the interface and whose edges indicate connectivity. On the other hand, if the connectivity is not known, it can be quite difficult to determine. In fact, even some of the most popular algorithms for this purpose can produce surprisingly inaccurate surface representations, for example, inconsistent orientation, incorrect topology or surfaces with holes.

However, for the implicit function's discretization, with adequate resolution, the connectivity does not need to be considered, because the grid implies the connectivity information in the whole space.

Especially, for dynamic objects, for example, like splashing water, the surface moves around. The connectivity is not a "one-time" issue like that of explicit discretization.

Instead, it needs to be resolved over and over again every time pieces. Furthermore, when dealing with the evolution of interface with a sharp corner, merging the explicit representation makes the evolution process very complex, and can result in unacceptable errors. Therefore, in our specific case, implicit representation and discretization has been preferred over explicit function.

## *3.3 Geometric properties from implicit representation*

Given an implicit function there are a number of powerful geometric tools that are useful in our application.

Firstly, let us recall that we have designated the zero isocontour as the interface. Hence we can easily determine which side of the interface a point is on just by looking for the local sign of $\phi$. That is, $\vec{x}$ is inside the interface if $\phi(\vec{x}) < 0$, otherwise it is outside the interface.

Secondly, implicit function makes both simple Boolean operations and more advanced constructive solid geometry operations easy to apply. If $\phi_1$ and $\phi_2$ are different implicit functions, then $\phi(\vec{x}) = \min(\phi_1(\vec{x}), \phi_2(\vec{x}))$ is the implicit function representing the union of the interior regions of $\phi_1$ and $\phi_2$. Similarly, $\phi(\vec{x}) = \max(\phi_1(\vec{x}), \phi_2(\vec{x}))$ is the implicit function representing the intersection of the interior regions of $\phi_1$ and $\phi_2$.

Thirdly, the gradient and surface normal at any point of the interface given by implicit function have the following definition.

Gradient $\nabla\phi$:

$$\nabla\phi = \left(\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right) \qquad (3\text{-}9)$$

That is the gradient consists of the directional derivative in each dimension. For the gradient under two dimensions, its definition is easily obtained by just removing the z direction derivative.

The gradient $\nabla\phi$ is perpendicular to the isocontours of $\phi$ and points in the direction of increasing $\phi$. Therefore, if a gradient is computed at a point on the interface, it has the same direction with local unit outward normal at that point. Then we can induce the definition of local unit normal $\vec{N}$ for interface points:

$$\vec{N} = \frac{\nabla\phi}{|\nabla\phi|} \qquad (3\text{-}10)$$

For discrete implicit function, the approximation to the gradient and normal can be given easily. On our Cartesian grid, the derivatives in equation (3-9) need to be approximated. There are the following three basic ways:

- First-order accurate forward difference

$$(\phi_x^+)_i = \frac{\partial\phi}{\partial x} = \frac{\phi_{i+1} - \phi_i}{\Delta x} \qquad (3\text{-}11)$$

- First-order accurate backward difference

$$(\phi_x^-)_i = \frac{\partial\phi}{\partial x} = \frac{\phi_i - \phi_{i-1}}{\Delta x} \qquad (3\text{-}12)$$

- Second-order accurate central difference

$$(\phi_x^0)_i = \frac{\partial\phi}{\partial x} = \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} \qquad (3\text{-}13)$$

For the directional derivative in other two directions, the equations have same format. Using such approximations can cause some errors in the computation of new location of the interface surface. But if the implicit function is smooth enough, these approximations are within acceptable errors and the surface position has just a small perturbation [33].

Another important geometric property for the interface surface is the mean curvature, which means the divergence of the normal $\vec{N} = (n_1, n_2, n_3)$ ,

$$k = \nabla \cdot \vec{N} = \frac{\partial n_1}{\partial x} + \frac{\partial n_2}{\partial y} + \frac{\partial n_3}{\partial z} \qquad (3\text{-}14)$$

So that $k > 0$ for convex region, $k < 0$ for concave region, $k = 0$ for a plane (Figure 3.3).

Substituting equation (3-10) into equation (3-14) gives:

$$k = \nabla \cdot \left( \frac{\nabla \phi}{|\nabla \phi|} \right) \qquad (3\text{-}15)$$

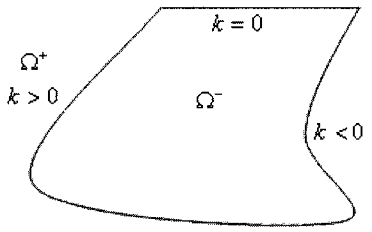


**Figure 3.3**: the mean curvature feature

Using directional derivative we can write equation (3-15) as follows:

In two dimensions: $k = \left(\phi_{xx}\phi_y^2 - 2\phi_y\phi_x\phi_{xy} + \phi_{yy}\phi_x^2\right)/|\nabla\phi|^3$ (3-16)

And in three dimensions:

$$k = \left(\phi_x^2\phi_{yy} - 2\phi_x\phi_y\phi_{xy} + \phi_y^2\phi_{xx} + \phi_x^2\phi_{zz} - 2\phi_x\phi_z\phi_{xz} + \phi_z^2\phi_{xx} + \phi_y^2\phi_{zz} - 2\phi_y\phi_z\phi_{yz} + \phi_z^2\phi_{yy}\right)/|\nabla\phi|^3$$
(3-17)

Where, the first order partial derivative can be computed using equation (3-11) – (3-13).

The second partial derivative of $\phi$ in the $x$ direction is given by

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} \qquad (3\text{-}18)$$

The second partial derivative $\phi_{xy}$ is computed based on $\phi_x$ and $\phi_y$, and here we give the equation using central difference

$$\phi_{xy} = \frac{\phi_x^{j+1} - 2\phi_x^j + \phi_x^{j-1}}{2\Delta y} \qquad (3\text{-}19)$$

Other second-order derivatives have the similar format as in equation (3-18) and (3-19).

## *3.4 Signed distance function (SDF)*

Because of the advantages of using implicit functions, discussed above, it is an appropriate representation method for a dynamic object's surface. Furthermore, the smoothness of the implicit function is a desirable property in sampling the function and using numerical approximations. By choosing a suitable implicit function, it not only simplifies the equation but also reduces the computation cost. In addition, it can increase the accuracy of computation. The signed distance function is one good choice.

A distance function $d(\vec{x})$ is defined as

$$d(\vec{x}) = \min(|\vec{x} - \vec{x}_I|) \qquad \text{for all } \vec{x}_I \in \Omega \qquad (3\text{-}20)$$

Geometrically, the distance can be constructed as follows. If $\vec{x} \in \Omega$, then $d(\vec{x}) = 0$. Otherwise, for a given point $\vec{x}$, find the point on the interface closest to $\vec{x}$, and label this point $\vec{x}_C$. Then $d(\vec{x}) = |\vec{x} - \vec{x}_C|$.

For a given point $\vec{x}$, if the closest interface point is $\vec{x}_C$, the line segment from $\vec{x}$ to $\vec{x}_C$ is the shortest path from $\vec{x}$ to the interface. Any local deviation from this line segment increases the distance from the interface. In other words, the path from $\vec{x}$ to $\vec{x}_C$ is the path of steepest descent for the function d. Furthermore, since d is the Euclidean distance, we get

$$|\nabla d| = 1 \qquad (3\text{-}21)$$

This is intuitive in the sense that moving twice as close to the interface gives a value of d that is half as big.

A signed distance function is an implicit function $\phi$ as follows

$$|\phi(\vec{x})| = d(\vec{x}) \quad \text{for all } \vec{x}$$

And $$\phi(\vec{x}) = -d(\vec{x}) \quad \text{for } \vec{x} \text{ in interior portions} \qquad (3\text{-}22)$$

$$\phi(\vec{x}) = d(\vec{x}) \quad \text{for } \vec{x} \text{ in exterior portions}$$

Similarly, we have

$$|\nabla\phi| = 1 \qquad (3\text{-}23)$$

However, equations (3-21) and (3-23) are only true for any $\vec{x}$ as long as there is a unique closest point $\vec{x}_C$. That is (3-21) and (3-23) are true except at points that are equidistant from (at least) two distinct points on the interface. Unfortunately, such equidistant points can exist, making (3-21) and (3-23) only true only in a general sense.

*Equations that are true in a general sense can be used in numerical approximations as long as they fail in a graceful way that does not cause an overall deterioration of the numerical method [31].* This is a general and powerful guideline for any numerical approach. More important, if the failure of an equation that is true in a general sense causes overall degradation of the numerical method, then many times a special-case approach can be devised to fix the problem.

As mentioned in previous chapters, in order to improve the accuracy of the computation of some geometry variables, the smoothness of the implicit function is important. It turns out that signed distance functions have a good smoothness except at some kinks which can be numerically smeared. So, signed distance function is still a good choice to represent the interface.

By using equation (3-23) we can simplify many of the formulae. For example, equation (3-10) simplifies to

$$\vec{N} = \nabla\phi \qquad (3\text{-}24)$$

Equation (3-16) simplifies to

$$k = \Delta\phi = \phi_{xx} + \phi_{yy} + \phi_{zz} \qquad (3\text{-}25)$$

Representing the interface via signed distance function also simplifies the level set evolution, as we shall see in the next chapter.

## *3.5 SFL file format [33]*

The SFL file format has been introduced to provide a versatile file format to import and export point based objects and scenes into Pointshop3D.

SFL file is a binary format file. An SFL file is a container for scenes containing multiple surfel objects, each with its own transformation matrix. The SFL format is extensible. Each object can have different surfel attributes, and the associated attributes can be arbitrarily extended.

Following is a short description of the structure of an SFL file:

- An SFL file consists of a *header*, containing general information about the scene. In this part, the author, application and it's title, time etc. can be defined.
- Next is the surfel set lists, which contain *surfel sets* of the scene in a sequential list.
- A surfel set corresponds to a surfel object in Pointshop3D. Each surfel set can have its own set of surfel attributes.
- The SFL library allows storing multiple *resolutions* per surfel set. This feature is not yet used by Pointshop3D. Presently one only stores the ``default resolution" for each surfel set.
- Each resolution holds an array of *surfels*.

Because SFL file is a binary file, it cannot be read or written using normal text editing software. Pointshop3D provides APIs to read or write SFL file. For the detailed commands please refer the Pointshop3D website. [33]

The attributes for each surfel include the following:

- Position: 3D coordinates for each point sample

- Normal: the Normal vector for each point sample
- Radius: the radius for the surfel disk, center is at the position of the point sample
- Diffuse color
- Texture Coordinates
- ... and others which we do not list as they are not at present relevant to our work

## 3.6 *Conversion from SFL to SDF*

As discussed, it is a good choice to use signed distance function to represent the interface. We need to find a solution to convert the interface sample points in an SFL file which forms the input into the corresponding SDF.

In our solution, there are two steps to generate SDF for corresponding SFL

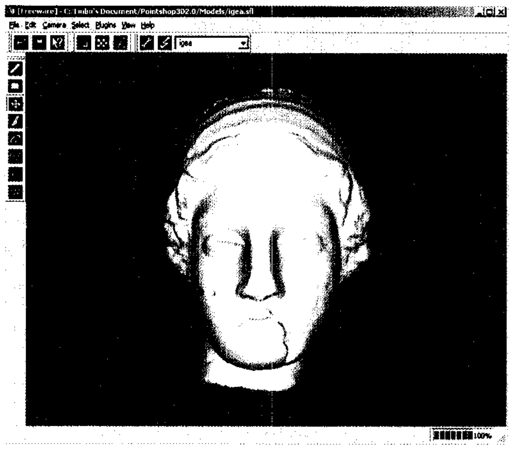
- Read SFL and generate two data files including position and normal information separately
- Compute a uniformly divided bounded range (rectangular box) for the entire point set.; compute the signed distance to the sampled interface contained in the above data for each grid node.

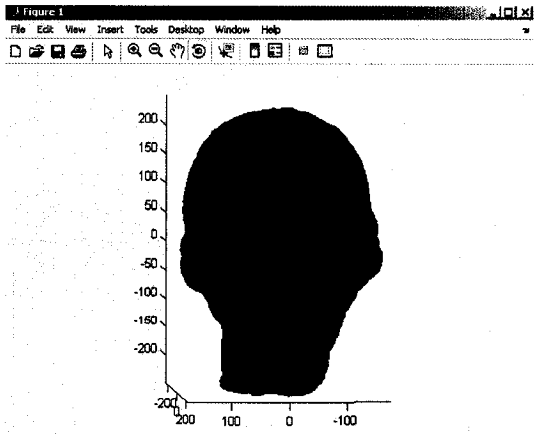
SFL file is a binary format file and it only can be read by the API provided by Pointshop3D library [34]. So, a specific function sfl2text is written to achieve this. The SFL file is first opened to enable browsing of all the surfel lists; then for each surfel the position information and normal information are extracted; next the position and normal information are saved into separate data files. A portion of sfl2text function is shown below:

```
for (i=0; i<numSurfels; i++)  // numSurfels is the total number of surfels in current file
        { ....
        in->beginSurfel()
        TempSplat splat;
        in->readSurfelPosition3(splat.x, splat.y, splat.z);
        in->readSurfelNormal3(splat.nx, splat.ny, splat.nz);
        in->endSurfel();
       data_file << splat.x <<"  " << splat.y << "  " <<splat.z << "\n";
       normal_file << splat.nx <<"  " << splat.ny << "  " <<splat.nz << "\n";
       ...
    }
```

In order to use signed distance function to represent the interface, we first need to create a grid whose size is decided by the object's size. The maximum width in X, Y and Z dimension will be computed and used to decide the grid range. Then the grid will be uniformly divided. The last, also the most important, step is to find the closest interface point got from the position data file for each grid node and compute the distance. To find the closest interface point is a time-consuming task, and as can be seen from the literature, there has been a lot of research carried out on this [35][36][37]. Here I use the Matlab program developed in our group to do the conversion.

Figure 3.4, (a) shows the rendering of Igea model represented using SFL in Pointshop3D software, and Figure 3.4 (b) shows the same model represented using SDF in Matlab.


(a) The Igea model in SFL format and displayed in Pointshop3D



(b) The Igea model in SDF format and displayed in Matlab

**Figure 3.4:** the SFL format and SDF format of Igea model

# *Chapter 4: Global Parameterization of Point Based Models by Evolving the Interface*

## *Introduction*

In order to do global parameterization for texture mapping on point-based model, as was discussed in Chapter 2, we will first evolve the interface of the object to the interface of another specific target object, such as a sphere. The 2D polar coordinate of every point on the sphere surface is distinct and easy to compute. Thus every point on the interface of the given point based object can now be assigned a 2D parametric coordinate values.

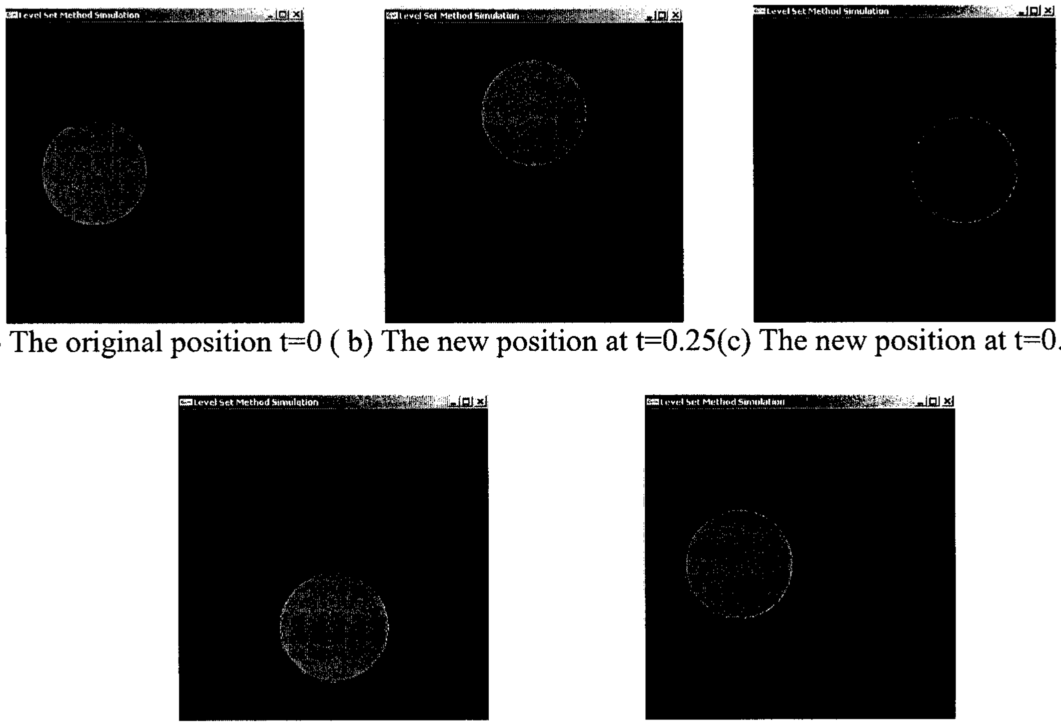
Level set methods provide us powerful tools to evolve the interface. The level set technique was devised by Osher and Sethian in [38] as a simple and versatile method for computing and analyzing the motion of an interface in two or three dimensions, such as closed curve in 2D or surface in 3D that bounds a region. The goal is to compute subsequent motions of interface under a velocity that can depend on position, time, the geometry of the interface or external physics.

In the first section of this chapter we shall introduce the motion of an object's interface. Next, level set methods are introduced. The use of level set method in the task of texture mapping is then described. This is followed by description of two solutions to track the movement of the interface.

## *4.1 Motion of interface*

The processes of evolving and tracking the interface of an object are part of many problems in science and physical engineering. We are mainly concerned with closed interfaces: curves in two dimensions and surfaces in three dimensions. Usually a closed interface divides the space into two portions: interior and exterior. The evolution of interface means the changes in these two portions; the changes may be in position, size, shape etc. The motion of the interface usually is driven by giving each interface point a velocity in time dimension.

There are many kinds of motion of the interface. For example, the convection of interface, which maintains the interior shape and size, (*Cf* Figure 4.1); the expanding of interface in which only the velocity in normal direction is considered, (*cf* Figure 4.2).



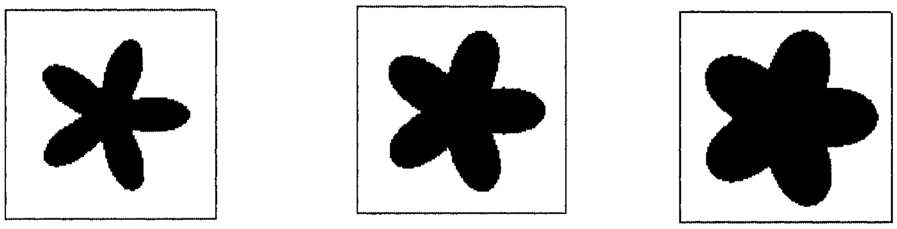
(a) The original position t=0 ( b) The new position at t=0.25(c) The new position at t=0. 5

(d) The new position at t=0.75 (e) The new position at t=1.0

**Figure 4.1:** The interface convection without change in shape of the interface; (a) and (e) are at the same position.

In a variety of physical phenomena, one wants to track the motion of the interface with speed dependant on some geometric property, such as the curvature. Two well-known examples are crystal growth [39] [40][41] and flame propagation[42][43]. This kind of motion is very popular and it is also helpful in our method. This will be discussed in detail in subsequent sections. Figure 4.3 shows the motion of a star shaped interface in 2D, based on curvature.

In our method, we will combine constant velocity motion in normal direction with curvature dependant velocity motion in normal direction.


(a) t =0 (b) t= 3 (c) t =6

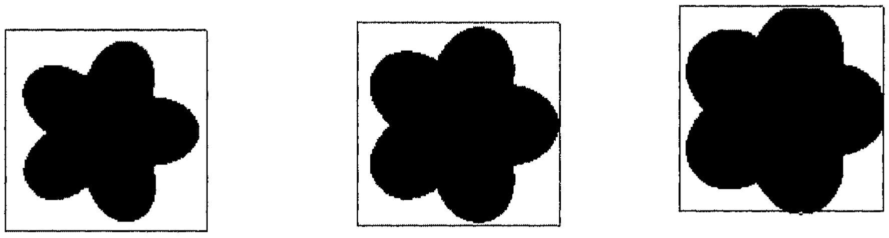

(d) t= 9 (e) t =12 (f) t= 15

**Figure 4.2:** The motion in normal direction

(a) The original star interface (b) The interface at time= 0.3 (c) The interface at time= 0.7

(d) The interface at time= 1.0 (e) The interface at time= 2.0 (f) The interface at time= 4.0

**Figure 4.3:** The motion in normal direction based on curvature

## 4.2 Methods to evolve the interface

As discussed in last chapter, there are two popular approaches to represent the closed interface: explicit and implicit. Correspondingly, there are two types of numerical algorithms employed in the solution of evolving and tracking the interface.

### 4.2.1 Explicit techniques: parameterize the moving interface

Suppose we use the parameterized representation of an interface as the backbone of a numerical algorithm. We choose finite number of buoys and the line between buoys represents the curve in two dimensions, (*Cf* Figure 4.4). Similarly, we can choose finite number of buoys in 3D and combine triangles by choosing three buoys as vertices to represent the surface. This method is also known as the marker particles [44].

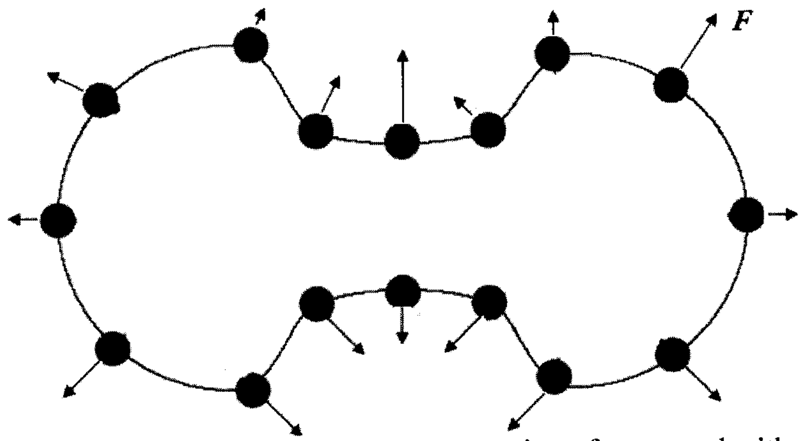


**Figure 4.4:** The discrete explicit representation of curve and with velocity F

Suppose the velocity of each buoy is given as $\vec{v}(\vec{x}_i)$, $\vec{x}_i$ for $i = 1..N$ are points on the interface curve. Then we want to track the interface points' new positions at a later time under the given velocity. The simplest way is to solve the ordinary differential equation (ODE) given by:

$$\frac{d\vec{x}}{dt} = \vec{v}(\vec{x}) \qquad (4\text{-}1)$$

This is the Lagrangian formulation of the interface evolution. The evolution of interface can be approximated by moving of these buoys and the maintaining the connectivity (line segment or the triangle). This is not so hard to accomplish if the connectivity des not change and the surface elements are not distorted too much.

However, there are the following known problems when using explicit technique to evolve the interface [33][46][45]:

- The discreterization parameterization has to been done repeatedly.

  When evolving the interface, any trivial change in velocity field values can cause large distortion of boundary elements, and the accuracy of the method can deteriorate quickly if we continue to use the current buoys and their current connectivity. A remedy is to stop the advancement periodically, re-walk along the curve or surface to drop new buoys and determine their connectivity. However, doing this for a propagating interface, especially a surface in three dimensions, is not at all an easy task.

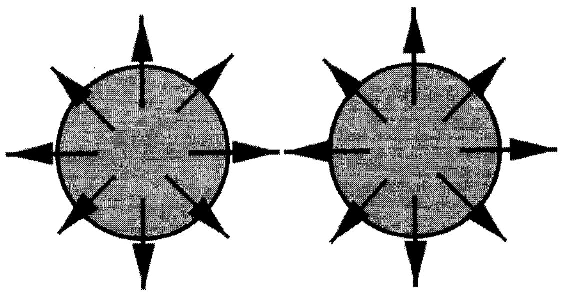
- For the most physically popular motion under curvature, it is difficult.

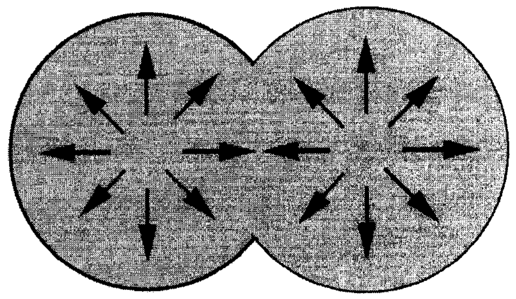
  Many physics phenomena, such as burning flame and the moving wave front, can be simulated by the motion of interface dependent on curvature. As the interface is represented explicitly, there is not enough geometric information to compute curvature because only the geometric information of the buoys is available.

- Topological changes, such as merging curves, require special treatment.

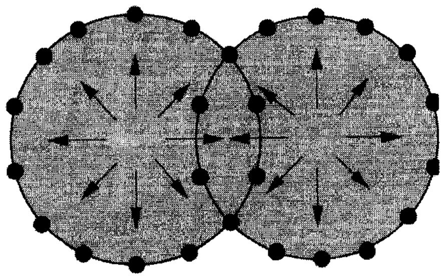
  A more serious problem comes when an evolving boundary attempts to change its topology. In the Figure 4.5, there are two separate circular flames, each burning outwards at a constant speed.

Before merging, the independent evolution of each of the two flames can be predicted easily. As the two separate flames burn together, the evolving interfaces merge into a single propagating front. However, the evolution procedure will run into a real trouble: the two sets of buoys located inside the burned region must somehow be removed if we want to track the true edge of the expanding flame. Trying to systematically determine which buoys to remove is a complex task.





(a) Original flames (b) The merging of two flames



(c) The two pairs of bouys inside the burning range should be removed

**Figure 4.5:** The evolution of two separate burning flames

### *4.2.2 Implicit techniques: level set methods*

The explicit representation represents the N dimensional interface in N dimensions. In contrast, implicit representation embeds the N dimensional interface in N+1 dimensions. That is, a curve is represented as the interface of two 2D space portions, and a surface is represented as the interface of two 3D space portions. As mentioned earlier, in this case, we can get all the global geometric information that we need and it is also easy to compute gradient, curvature and other geometric quantities.

The most popular numerical technique for evolution of implicit representation is level set method, which adds dynamics to implicit interfaces. The first idea that started off the level set popularity was the Hamilton-Jacobi approach to numerical solutions of a time-dependent equation for a moving implicit surface. It was first done in the seminal work of Osher and Sethian.[38]

In the level set method, the interface is represented implicitly by the zero level set of a function, $\phi(\vec{x}) = 0$. Note that $\phi$ is defined for all $\vec{x}$, and it is not just for the points on the boundary. When we consider the evolution of interface, we designate the level set function as time-dependent, so we add one time variable into the function $\phi(\vec{x},t)$. Figure 4.6 shows, for time $t = 0$, the interface as $\phi(0) = \phi(\vec{x},0) = z = 0$. If the interface evolves with given velocity, at time t the interface is $\phi(t) = \phi(\vec{x},t) = z = 0$.

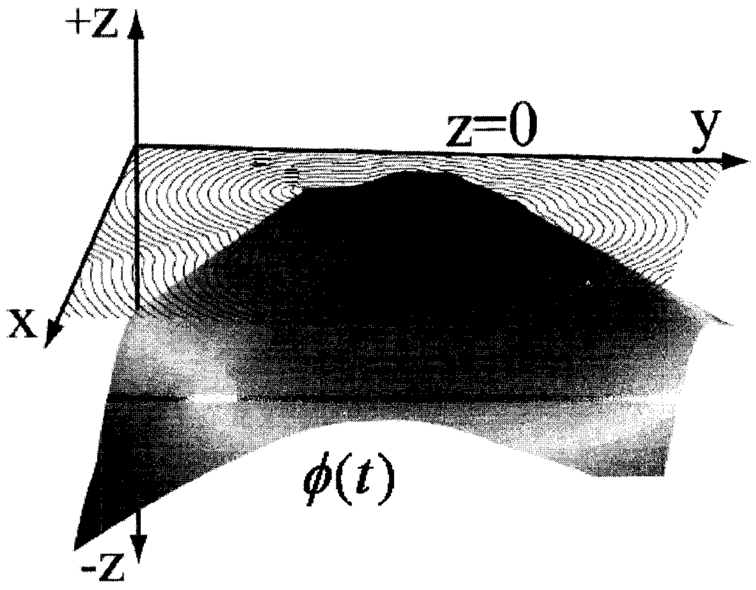


**Figure 4.6:** Curve is defined as the zero level set and embedded in two-dimensional space

Next, we give the mathematical definition of level set method and of the equations that can be used for different motions.

## Common Convection Equation

Level set methods are a collection of numerical algorithms for solving a particular class of partial differential equations (PDE) [47]. Assume we have level set function $\phi(\vec{x})$, and the interface is the set with $\phi(\vec{x},0) = 0$. And suppose for each point in the space, not only on the interface, we have velocity vector $\vec{V}(\vec{x},t) = (u,v,w,t)$, $t$ is the time factor. Then, we can get the following simple convection equation.

$$\phi_t + \vec{V} \cdot \nabla\phi = 0 \qquad (4\text{-}2)$$

where the $t$ subscript denotes a temporal partial derivative of level set function and $\nabla$ is the gradient operator. Substituting equation (3-9) into equation (4-2), we get

$$\phi_t + u\phi_x + v\phi_y + w\phi_z = 0 \qquad (4\text{-}3)$$

Equation (4-2) is a partial differential equation (PDE) which defines the motion of the interface given by $\phi(\vec{x},t) = 0$. By solving this equation, we can get the new level set function at time $\Delta t$, and the interface can be extracted out by restricting $\phi(\vec{x},\Delta t) = 0$. Equation (4-2) is an Eulerian formulation of the interface evolution, since the interface is captured by the implicit function $\phi(\vec{x})$, as opposed to being traced by interface elements as was done in the equation (4-1). Equation (4-2) is referred to as the level set equation.

As shown by the convection in Figure 4.1, equation (4-2) is used and the velocity is given as $\vec{V}(\vec{x}) = (-2\pi x, 2\pi y)$.

To represent $\phi$ in a finite form and to solve the above equation numerically, we can use a grid within a fixed range to discretize the space. A common choice is a simple uniform Cartesian grid; quadtree and octree representations can be used for higher efficiency [48]. The numeric implementation is described in the next section.

## Motion in the Normal Direction Equation

In this part we give the level set equation for the motion under an externally generated constant velocity field in the normal direction. The velocity is defined by $\vec{V}(\vec{x},t) = a\vec{N}$, where $a$ is a constant. Putting this velocity into equation (4-2), we get

$$\phi_t + a\vec{N} \cdot \nabla\phi = 0 \qquad (4\text{-}4)$$

Then substituting equation (3-10) into equation (4-4), give us:

$$\phi_t + a\,|\nabla\phi| = 0 \qquad (4\text{-}5)$$

The constant $a$ can be of either sign. When $a > 0$ the interface moves in the normal direction, and when $a < 0$ the interface moves opposite to the normal direction. For example, see Figure 4.2, where $a = 0.02$.

## Motion based on Mean Curvature Equation

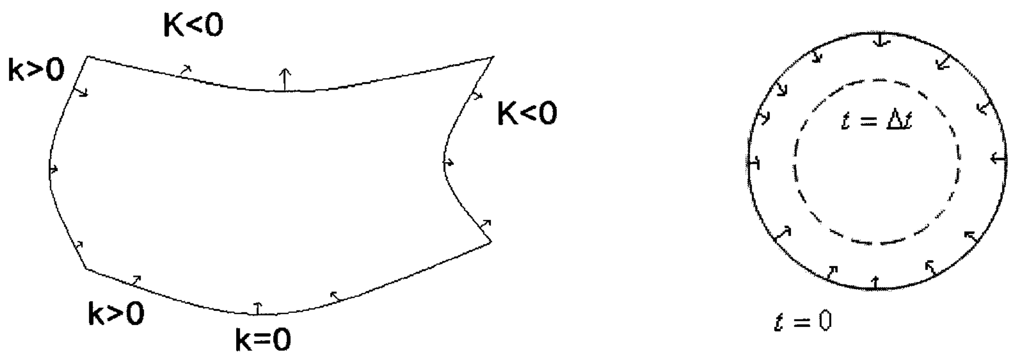
As we discussed before, there are many physics phenomena, such as the burning flame, in which the interface's movement is dependent on the local curvature.

Generally we consider the interface motion in the normal direction and the velocity is proportional to the mean curvature. For example, $\vec{V}(\vec{x},t) = -bk\vec{N}$, where b is a positive constant and the minus sign is chosen so that convex parts move in and concave parts move out. See Figure 4.7 (a). Let us recall that in Figure 4.3 we have already seen motion of star interface dependent on curvature.

If the original interface is itself a circle (in 2D) or sphere (in 3D) and it is evolved depending on the curvature in normal direction, then the interface will remain as circle or sphere respectively, as can be seen in Figure 4.7 (b). This is quite intuitive, as we know that the circle and sphere have a constant curvature everywhere on the interface.

In [49], Grayson proved that every simple closed curve collapses smoothly to a single point, without crossing over itself, and there is the remarkable theorem: no matter how complicated or convoluted a curve might be, it quickly relaxes itself into a circular object and shrinks down to a point under the motion based on curvature (*cf* Figures 4.3 and 4.8). This theorem has been the primary inspiration for our research on texture mapping on two counts:

1) The target is a circular object, and
2) During evolving there is no intersection between the interface points. But this is more complicated in three dimensions, and we will discuss it in next section.



(a) The sign of k decides the direction of velocity

(b) The circle remains as circle with the motion based on curvature

**Figure 4.7:** Velocity $\vec{V}(\vec{x},t) = -bk\vec{N}$, b is a positive number

Here we continue our discussion of the level set motion dependent on curvature. The velocity field for motion by mean curvature contains a component in the normal direction only, that is the tangential component is identically zero. Inserting the new velocity into equation (4-2), we get:

$$\phi_t + (-bk\vec{N}) \cdot \nabla\phi = 0 \qquad (4\text{-}6)$$

And based on equation (3-10), we have $\vec{N} = \frac{\nabla\phi}{|\nabla\phi|}$; and inserting this into (4-6) yields:

$$\phi_t + (-bk)|\nabla\phi| = 0 \qquad (4\text{-}7)$$

Equation (4-7) is the level set function for the evolution dependent on curvature in the normal direction.

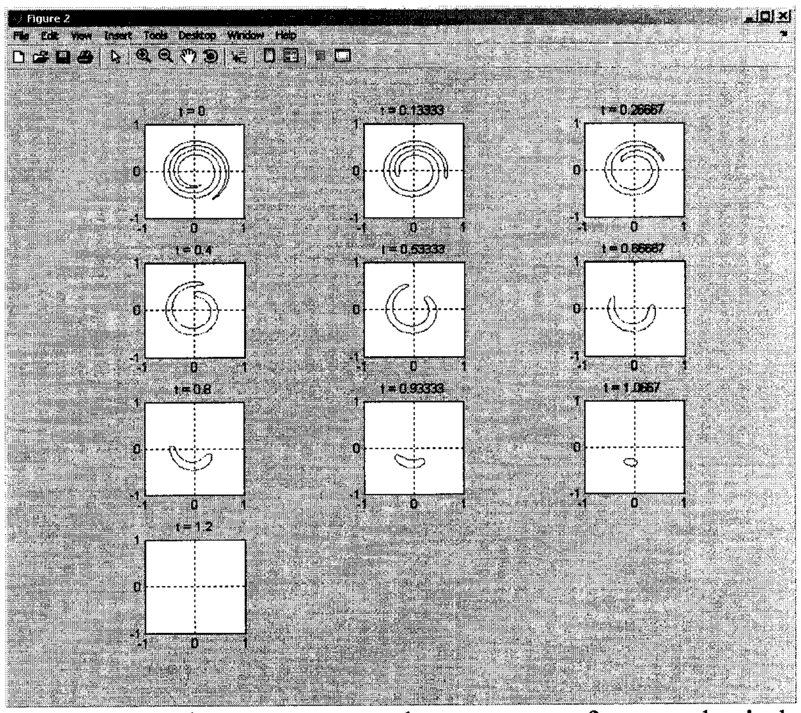


**Figure 4.8:** The movement under curvature of a wound spiral

## Corresponding Level Set Equation Using SDF

As discussed earlier in Chapter 3, signed distance function (SDF) is an implicit representation with many good geometric properties. The level set method does not mandate $\phi$ to be the signed distance function. However, the numerical approximations are inaccurate if $\phi$ has large variations in the gradient. Signed distance function has a very good property in this regard: $|\nabla\phi| = 1$ everywhere. Therefore, if possible it is a good idea to use SDF to represent the interface, because SDF not only improves the computation accuracy, but also simplifies the level set equation.

If $\phi$ is a signed distance function, then equation (4-5) will reduce to

$$\phi_t = -a \qquad (4\text{-}8)$$

The value of $\phi$ either increases or decreases depending on the sign of a. Equation (4-8) is much easier than (4-5). In addition, by initially using signed distance function as the implicit function, after each evolution step it has to continue to be a signed distance function, that means if $|\nabla\phi^n| = 1$, then $|\nabla\phi^{n+1}| = 1$.

By using signed distance function, equation (4-7) becomes

$$\phi_t = bk \qquad (4\text{-}9)$$

However, after each evolution step, $\phi$ will not remain as SDF.

For general evolution, after each step the original implicit function can not be kept as SDF, so we need to do frequent initialization to keep $\phi$ close to signed distance function, which is called reinitialization procedure and will be introduced in a later section..

## 4.3 The procedure to do texture mapping on point-based model

In order to apply the level set methods in our method, first let us review our planned procedure to do texture mapping on point-based model. (*Cf* Figure 4.9)

### 4.3.1 The input and data structure

As discussed in Chapter 1 and Chapter 3, the main input of our system is the point based model represented by SFL file. That means the input is only a set of discrete interface points with different geometric quantities, such as position, normal, the surfel's radius, etc.

In order to use level set methods to move the interface of model, first we have to create the implicit representation of the model. As mentioned before, we use signed distance function to represent the interface. Numerically we only need to compute the signed distance of grid node to the interface, without the need to construct a function [38].

In order to generate the signed distance representation from the SFL file, we use the uniform grid to embed the interface; two-dimensional grid for curve as shown in Figure 4.10 and similarly a three-dimensional grid for surface. In Figure 4.10, suppose $(x', y')$ is a point of the model from the SFL file, and $(x_i, y_j)$ is the $(i, j)$ grid node. Here, we assume that the computational range D of grid is big enough to cover all the interface points, and further that we have $\Delta x = x_i - x_{i-1} = x_{i+1} - x_i$, the same as $\Delta y$.

As we know, the level set method for implicit function representation does not directly track the movement of interface point $(x', y')$; instead it updates the level set of each grid node $\phi(\vec{x})$, where $\vec{x} = (x_i, y_j)$. So we need a separate data structure to store the interface points and the grid array.

In our method, the interface is stored as a vector and the basic element is the interface point. Each interface point has not only positional information, but also other information such as the corresponding position on circle or sphere, the texture ID, texture coordinates. See the appendix A for more detailed information on this aspect.

For the grid, in the C++ implementation, a class is created because it not only stores the grid node distance value but also needs to do other computations. See appendix A for details.

Other inputs are the textures that will be mapped onto the surface. Again, we simply use a two-dimensional array to store the 2D image texture, and up to 8 texture image arrays identified by their ID are organized in a vector structure.

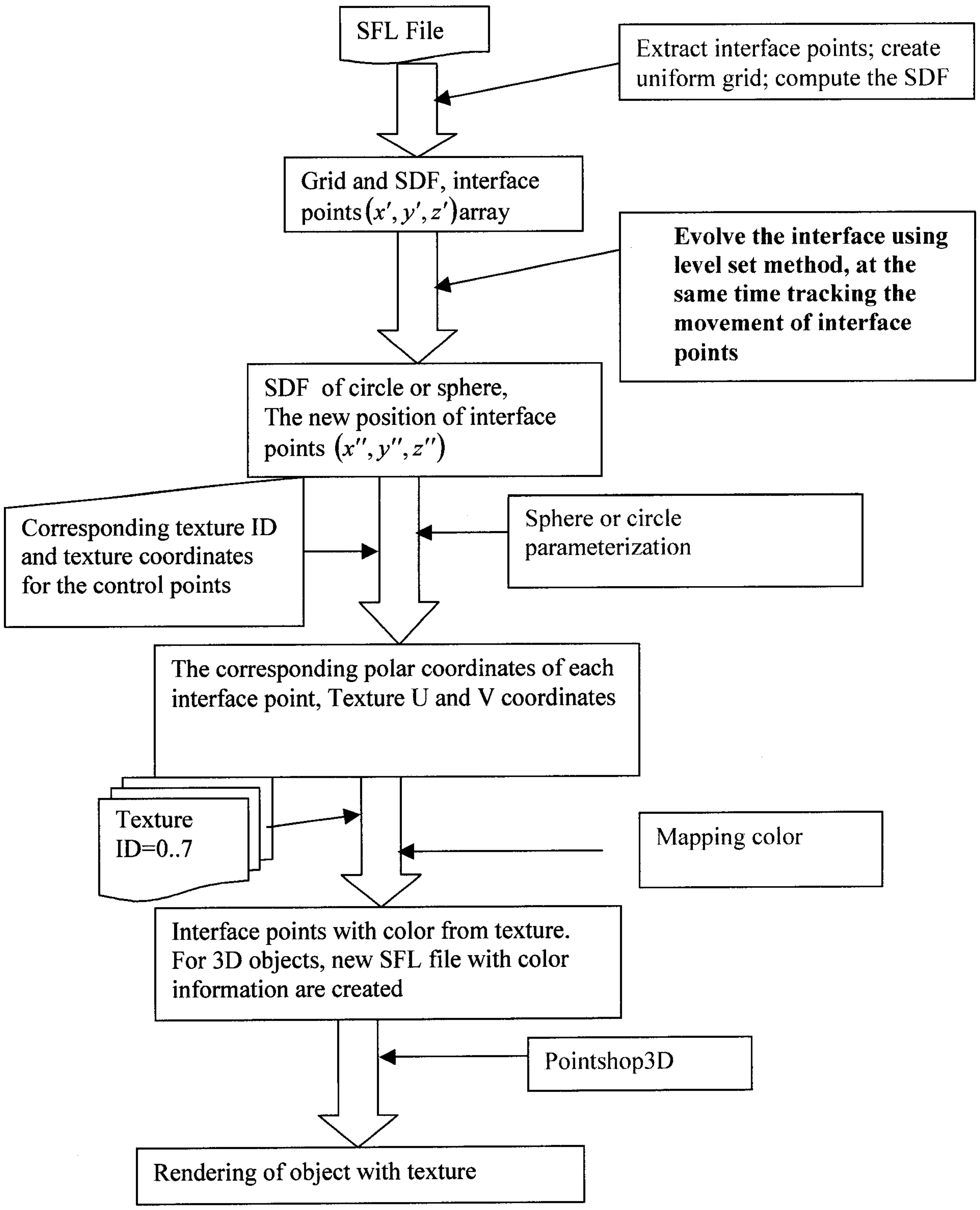

SFL File
Extract interface points; create uniform grid; compute the SDF
Grid and SDF, interface points $(x', y', z')$ array
**Evolve the interface using level set method, at the same time tracking the movement of interface points**
SDF of circle or sphere, The new position of interface points $(x'', y'', z'')$
Corresponding texture ID and texture coordinates for the control points
Sphere or circle parameterization
The corresponding polar coordinates of each interface point, Texture U and V coordinates
Texture ID=0..7
Mapping color
Interface points with color from texture. For 3D objects, new SFL file with color information are created
Pointshop3D
Rendering of object with texture

**Figure 4.9:** The texture mapping procedure

### 4.3.2 Functions

In the flow chart (*Cf* Figure 4.9) there are in total 4 functions to accomplish the texture mapping.

1. **Extract interface points; create uniform grid; compute the SDF**

   The input for this procedure is the SFL file and output is the vector of interface points with different properties, the grid object with the signed distance array for total grid nodes. As already explained earlier, the SFL file is first read and the point related data transferred into two files: position file and normal file; at the same time the interface vector is also created. Then, to create the signed distances for each grid node, we run the Matlab program which accepts the position and normal file as the input.

2. **Evolve the interface and track the movement of interface points**

   This is the most important part and will be presented in detail in next section.

3. **Sphere or circle parameterization**

   In this procedure the Cartesian coordinates of interface point are converted to corresponding polar coordinates.

   In Figure 4.11, suppose $(x', y', z')$ is a point on a sphere surface, and its corresponding polar coordinate is $(\theta, \phi)$. Then, there are following equations:

$$\begin{aligned} & z' = r * \cos(\theta) + z_c \\ \text{And} \quad & x' = r * \sin(\theta) * \cos(\phi) + x_c \qquad (4\text{-}10) \\ \text{And} \quad & y' = r * \sin(\theta) * \sin(\phi) + y_c \end{aligned}$$

   Where r is the radius of the sphere, $(x_c, y_c, z_c)$ is the center of the sphere. $\theta$ is the angle between vector $(x', y', z')$ and Z axis, and $\theta \in [0, \pi]$. $(x'', y'', z'')$ is the projected point on X-Y plane, and $\phi$ is the angle between the vector $(x'', y'', z'')$ and X axis, and $\phi \in [0,2\pi]$. Both $\theta$ and $\phi$ are continuous. For two-dimensional texture these polar coordinates can be used to compute the texture coordinates $(u,v) = \left(\frac{\theta}{\pi}, \frac{\phi}{2\pi}\right)$.

## 4. Mapping color

In this procedure, for each interface point we can get the corresponding color in the texture by computing its texture coordinate.

Here we use a two dimensional image as a sample to describe this procedure. (Figure 4.12) The image size is $M \times N$, and the coordinate of pixel in the image is $(u,v)$ where $u \in [0,1]$ and $v \in [0,1]$.

In the last procedure we get the unique polar coordinates $(\theta,\phi)$ for every interface point. Easily we can create the relation between the polar coordinate and texture coordinate using following equation:

$$u = \theta / \pi$$

And $$v = \phi / 2\pi \qquad (4\text{-}11)$$

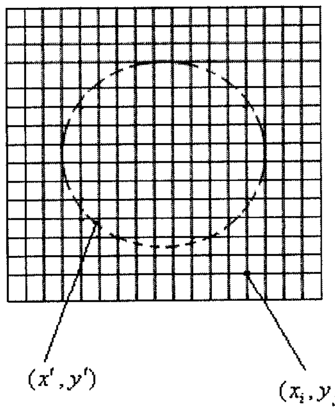


**Figure 4.10:** The grid for circle model

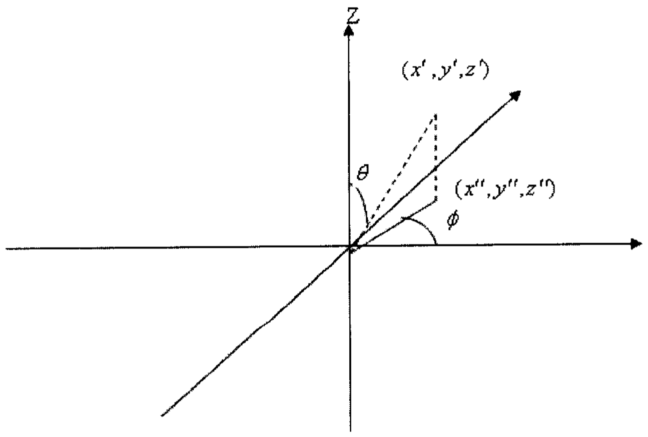


**Figure 4.11:** Cartesian coordinates to polar coordinates

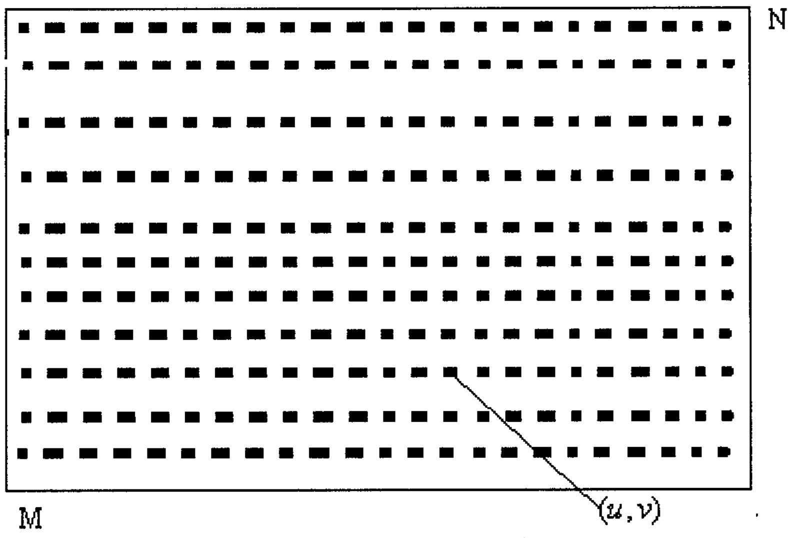


**Figure 4-12:** $M \times N$ 2D texture image

### 4.3.3 Output

The output of our system is different for 2D and 3D interfaces. For the 2D interface, curve, we just store the interface points with mapped color information in an array, and this is displayed directly by drawing the separate points. See the example in Figure 4.13; the left picture is the curve with mapped color, and the right is the texture.

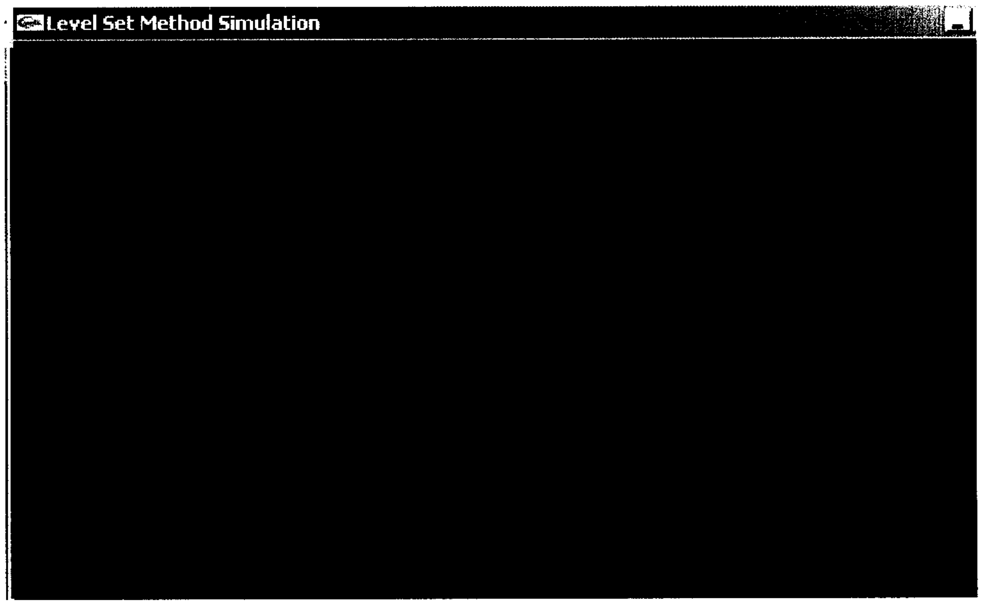
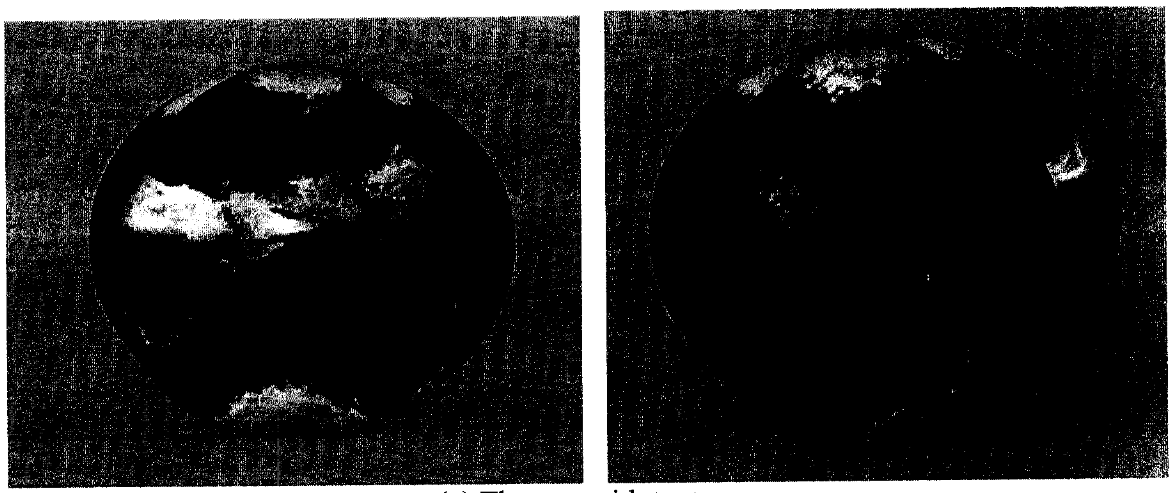


**Figure 4.13:** The texture mapping result for a 2D star

For 3D interface, we use the Pointshop3D software to render the surface. Before rendering we need to create new corresponding SFL files. In our program a dedicated data structure is used to store the interface points and their evolving information, so that it is easy to transfer the interface points to SFL file with the computed texture coordinates and mapped color. Figure 4.14 is a texture mapping result rendered in Pointshop3D.



(a) The star with texture

(b) Texture Image

**Figure 4.14:** The 3D star surface rendered in Pointshop3D

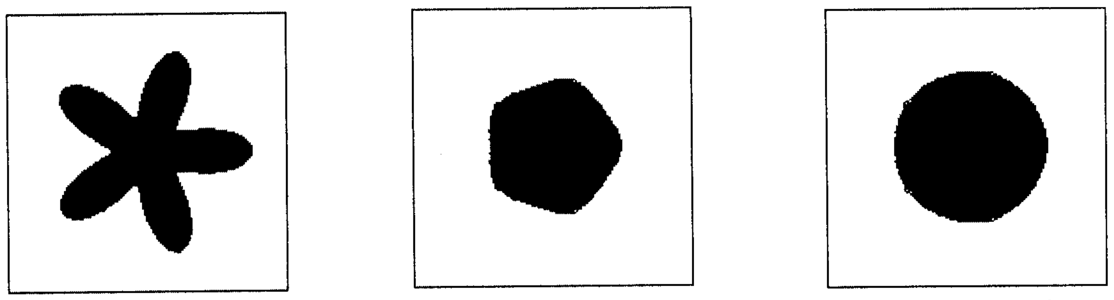
## 4.4 *The use of level set methods in texture mapping*

Level set methods have proven to be successful in tracking, modeling and simulating the motion of dynamic surfaces in various fields including graphics, image processing, computational fluid dynamics and many others. However, in order to use this method in our texture mapping, some changes have to be done; specifically some restraints have to be added.

### *4.4.1 Evolving the interface*

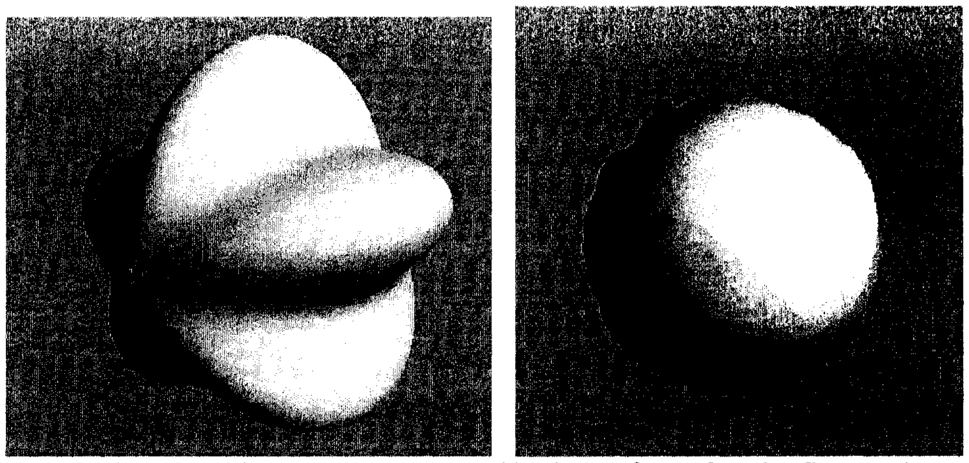
In our method the key step is to evolve the interface of any shape to a circle or sphere. We divide this process into two steps: evolve the interface dependent on local curvature in the normal direction until the curvature at each interface point is not negative, which is called the first evolution; evolve the interface in the normal direction to a bounding circle or sphere, which is called the second evolution. Please see Figure 4.15 and Figure 4.16.

In what follows, these two evolutions will be discussed separately.



(a) Original star (b) The interface with non negative curvature (c) The interface of bounding circle

**Figure 4.15:** 2D star evolve process


a) The original star b) The surface after the first evolution

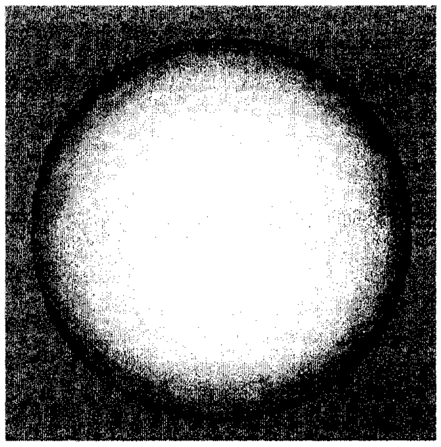

c) The surface after the second evolution

**Figure 4.16:** 3D star evolve process

### *4.4.2 The first evolution*

From Figure 4.3 and Figure 4.8, we know that even if the curve is very complicated, when we evolve the interface dependent on curvature using level set method, and under a certain condition – CFL condition, which is used to control how fast the interface moves at each time step and is discussed in detail later – there will be no self intersection and the curve tends to become a circle or collapse to a point. This in turn gives rise to another question: can we only evolve the interface dependent on curvature to reach a circle or sphere?

For the curve in two dimensions, this is possible. As we know the curvature of each point on the circle remains constant when evolving dependent on curvature. So if we want to only use the first evolution to reach our goal, the stop condition should be that the curvature of every interface point is the same. However, there are problems with this termination condition:

- The curve may collapse and disappear as a point before the stop condition is satisfied. For example, as can be seen in Figure 4.8, the spiral is still not a circle at time=1, but it disappears as a point at time=1.2.
- It needs a long time and the radius will be very small. For example in Figure 4.3, at time=1, each point on the interface has nonnegative curvature, but only at time=3.6 it fulfils the constant curvature condition. It usually has a very small radius compared with the bounding circle.

For the surfaces in three dimensions, the evolution is more complicated. First let us look at the definition of curvature. Standing at a point on a surface, the curvature of any path depends on the direction we travel. For example, standing in the center of a horse's saddle, one curvature is positive since it bends up, while another one bends down and hence is negative. Usually, we are concerned with motion under the mean curvature, which is the average of the biggest and smallest curvature. Figure 4.17, shows the evolution of the surface of dumbbell under mean curvature, and the middle handle has the same property with the saddle in curvature.

In Figure 4.17, because the handle is rather narrow, the dumbbell surface will break into two separate parts at some time with the evolution dependent on curvature. This is because, the mean curvature at the middle becomes positive, and it shrinks quickly until it becomes a point. Eventually, each part will evolve to a sphere surface and finally shrink to a point. However, in order to do texture mapping, especially when we want to build the global parameterization, this kind of breaking should be avoided.

Hence only using the first evolution based on curvature has the following problems. Usually it takes a lot of computational time to evolve up to the expected circle or sphere, and secondly, the worst result is that the closed surface may break into several separate parts. In order to address the above problems, we set a new stop condition to the first evolution. When the mean curvature of each interface point is not negative, we stop the first evolution. This way we can avoid the break into separate parts as shown for the dumbbell in Figure 4.18.

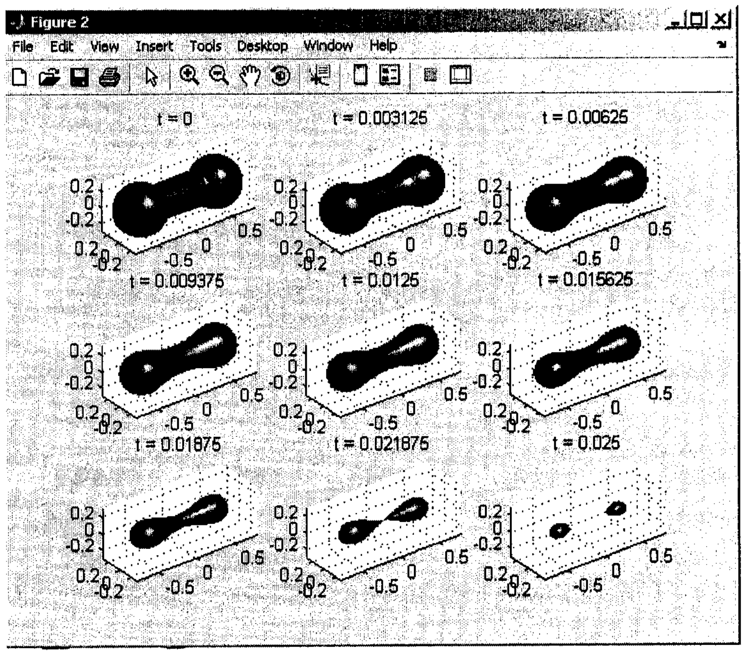


**Figure 4.17:** The dumbbell is evolved under mean curvature; with increasing time the dumbbell breaks into two separate parts.

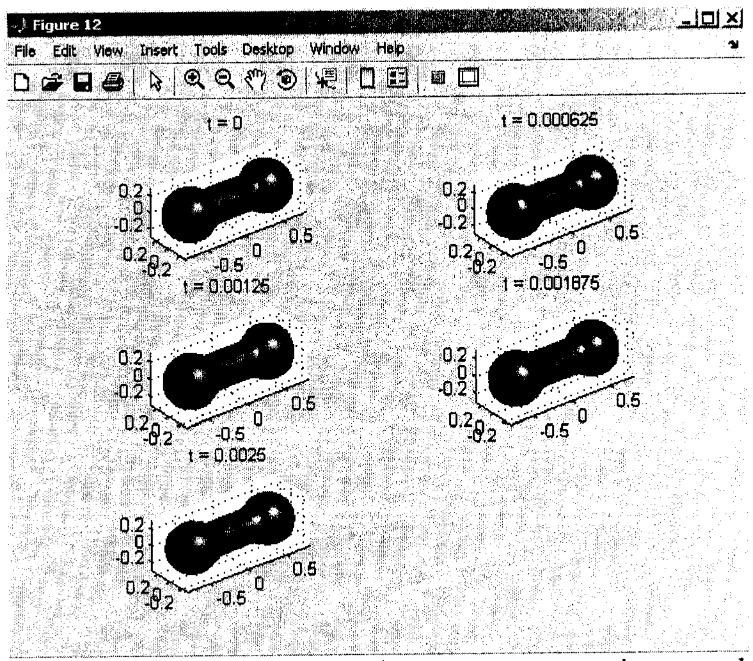


**Figure 4.18:** The dumbbell evolution under mean curvature is stopped when mean curvature is not negative everywhere.

### 4.4.3 The Second Evolution

There are two ways to do the second evolution. First method is to evolve the interface in the normal direction using level set method as described in Section 4.2.2, that is to evolve the interface with a constant velocity in the normal direction. Another way is to cast a ray from the interface point and the ray has the same direction with the normal at that point; then to compute the intersection point of the ray with the bounding circle or sphere, *cf* Figure 4.19. The bounding circle is decided by the maximum and minimum value of x and y coordinates of the interface points.

It may appear that we can track the new position on the bounding circle or sphere by directly using the second evolution without the first evolution. Unfortunately, it is not practicable. Below we will use the level set method to describe the problem.

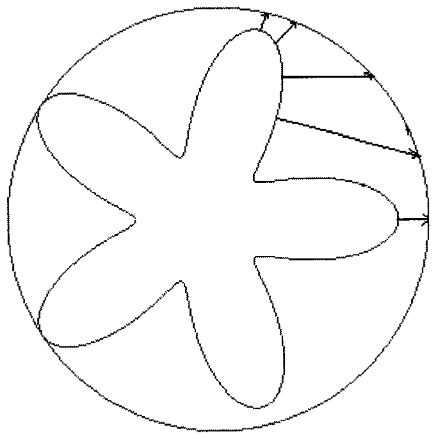


**Figure 4.19:** To cast a ray from the interface point in the normal direction to intersect with the circle

As described in Section 4.2.2, if we move the interface with constant velocity in normal direction, even without using the first evolution method, the interface can directly reach a circle or sphere. In some cases this is ok. For example, in the cases where the original interface is convex everywhere. However, not every interface has this good feature, for example the star. If we only take the second evolution, some interface points will intersect each other. For example Figure 4.20 shows an example where the interface tracking process used only second evolution.

We can see that the interface points in the red rectangle will merge or will be lost. This is not correct. Figure 4-21 gives the reason. When evolving in the normal direction A and B points must intersect at some time; this intersection will cause a major problem in tracking the interface and subsequently in texture mapping. Even if we choose a special method to track the movements of A and B, get their new position A' and B', there remains the unacceptable error, which is that the order of A' and B' is different from the order of A and B.

If we use the new position of these points to parameterize the curve or surface, the texture will be mapped wrongly, *cf* Figure 4.22 (a).

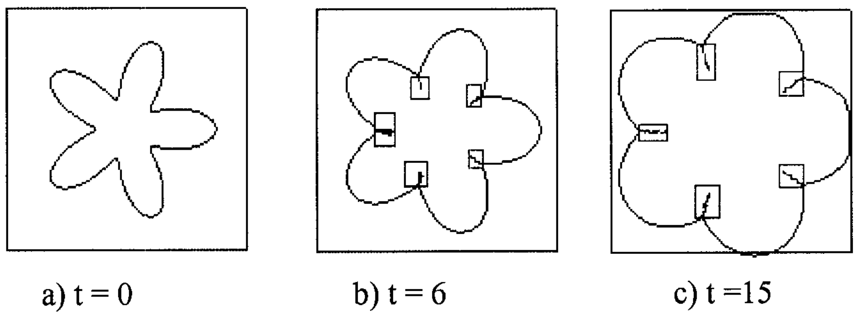


**Figure 4.20:** Using only second evolution the interface points in the red boxes cannot be evolved and tracked correctly for texture mapping.

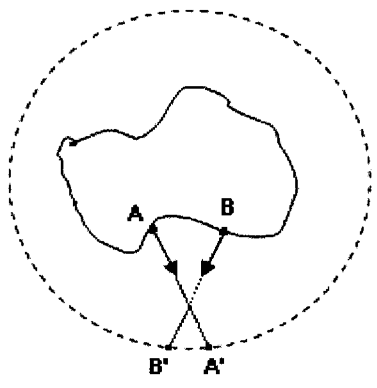
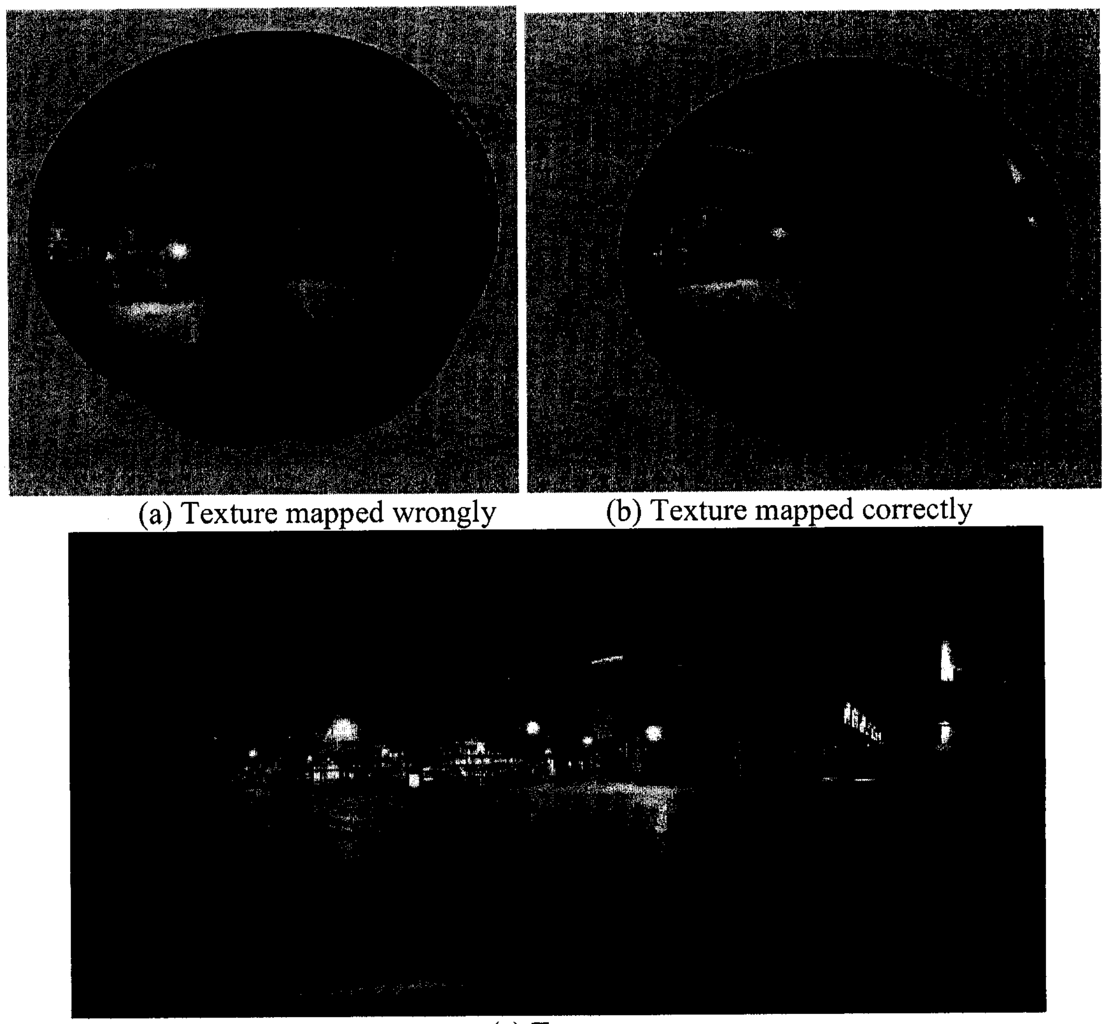


**Figure 4.21:** The normal at A and B intersect

### 4.4.4 First evolution plus second evolution

As described in Section 4.3.1, our goal is to evolve the interface to a circle or sphere, and from the discussion in the previous two sections, it is clear that we can not reach this goal only using either first or second evolution. So in order to avoid the problems discussed above we divide the evolution process into two steps: use the first evolution to get the approximation to circle or sphere, for example, Figure 4.13(b) and Figure 4.14(b). Then we expand the interface to the bounding circle or sphere using the second evolution.

The geometric meaning of curvature is the divergence of normal. If all the curvatures of points on the interface are not negative, it means that for 2D interface, curve, there is no concave potion, and for the 3D interface, surface, the rays in the normal direction of each surface point will not intersect outside the interface. With this condition, using the second evolution to reach the bounding interface, will not cause the parameter reversal problem described in section 4.4.3, *cf* Figure 4.23.



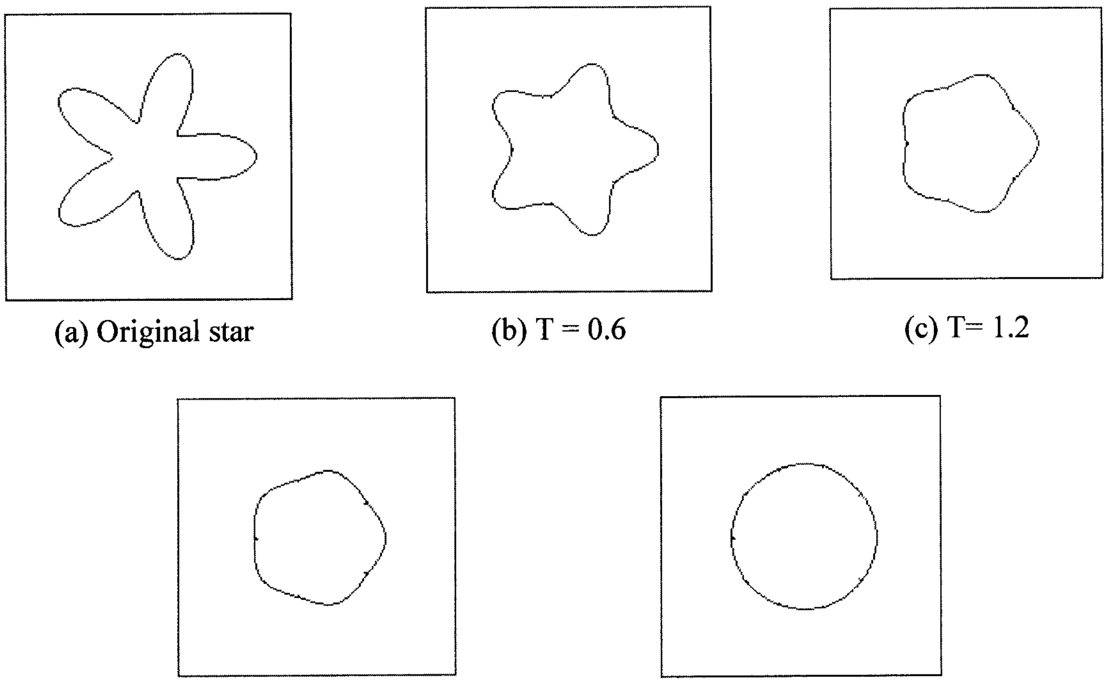
(a) Texture mapped wrongly (b) Texture mapped correctly

(c) Texture

**Figure 4.22:** The texture mapped wrongly only using second evolution

For the second evolution, we have two choices: directly casting a ray in the normal direction to compute the intersection point with bounding interface or evolving in the normal direction with constant velocity using level set method. These two methods have almost the same results. Ray casting is a one step computation and hence takes less time.

However, there are situations in which evolving the interface has advantages. In the case of dynamic objects, we can track movement of interface points so that it is possible to get the deformation information of objects with texture, *cf* Figure 4.24.



(a) Original star (b) T = 0.6 (c) T= 1.2

**(d) T =1.5, when K>=0** **(e) The target circle**

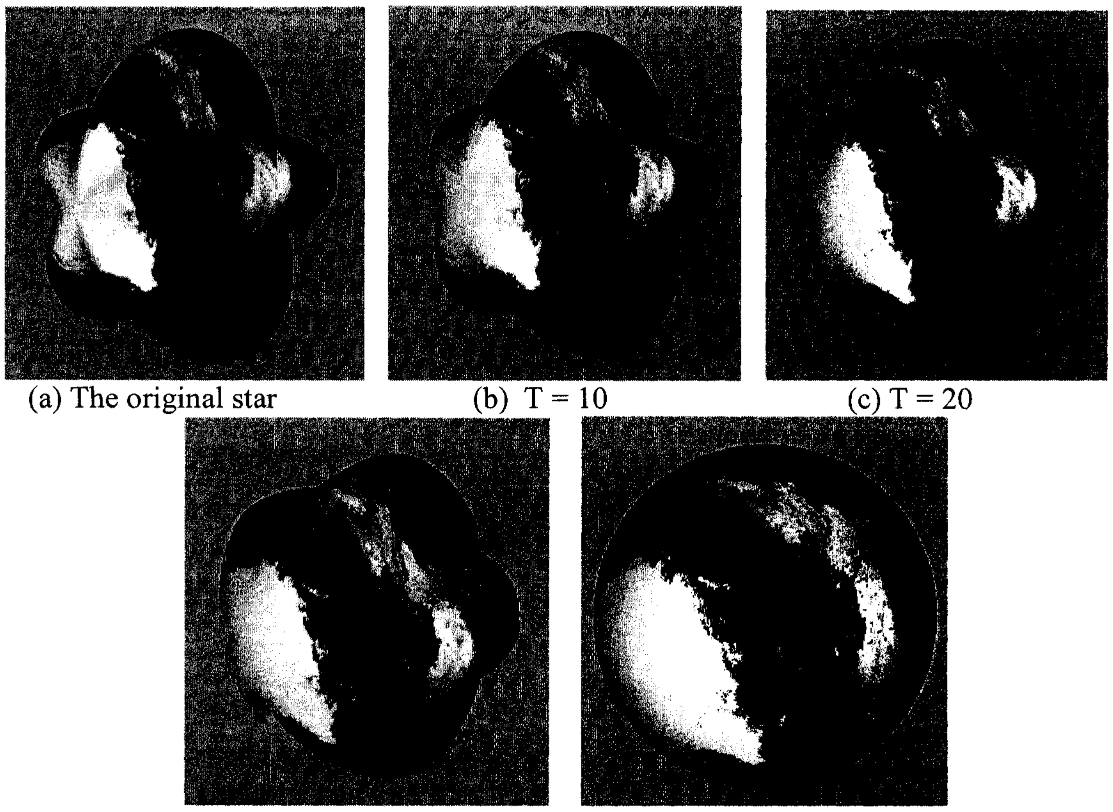
**Figure 4.23:** The evolution of star under the first and second evolution

### 4.4.5 Tracking the movement of interface points

Level set methods don't explicitly track the movement of interface points. However, in order to accomplish texture mapping, we have to assign a texture coordinate for which we must know where the original interface point is on the sphere. So we need to track the movement of every point sample of the point based model. For this we compute the new position of the interface points separately at each time step. The velocity of the interface point is decided by the velocities of the grid nodes surrounding it.

First, we calculate the velocities of the grid nodes surrounding the interface point. In the data structure of interface point, there is one important field $(grid\_X, grid\_Y, grid\_Z)$

which stores the grid index to identify the grid cell within which the interface is located, *cf* Figure 4.25.


(a) The original star (b) T = 10 (c) T = 20

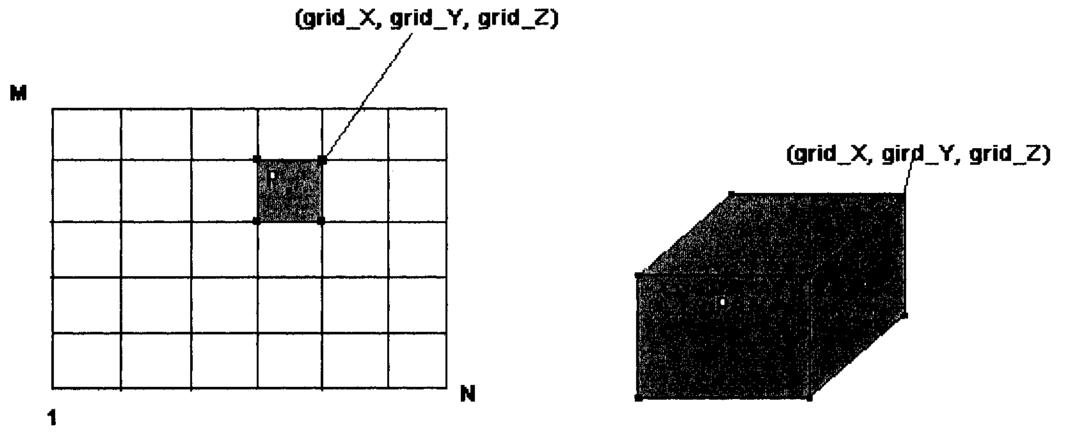
(d) T = 30 (e) target Sphere with texture

**Figure 4.24:** The deformation of objects with texture

$(grid\_X, grid\_Y, grid\_Z)$ decides which grid cell the interface point P belongs to. For example, in Figure 4.25 (a), the velocity of P can be interpolated by the four vertices of the green gird cell. On the other hand, in three dimensions, the velocity of P is decided by the eight vertices of the green box cell, *cf* Figure 4-25 (b).

Based on the level set equation, we know that there is a velocity for each grid node. For example, if we evolve the interface in the normal direction dependent on the curvature, the velocity is:

$$\vec{V} = -bk\vec{N} = -bk\frac{(\phi_x, \phi_y, \phi_z)}{|\nabla\phi|} \qquad (4\text{-}12)$$



(a) 2D grid cell and interface point P (b) 3D grid cell and interface point P

**Figure 4.25:** The interface point P in grid cell

Then for the velocity in X dimension we get:

$$\vec{V}_x = -bk\frac{\phi_x}{|\nabla\phi|} \qquad (4\text{-}13)$$

For the Y and Z dimension, there are similar equations.

Thus the velocity of each grid node can be calculated by computing the curvature, normal and gradient.

After getting the velocities of the vertices of the grid cell, we can use the bi-linear interpolation (in 2D) and tri-linear interpolation (in 3D) to calculate the velocity $\vec{V}_i$. Bi-linear interpolation and tri-linear interpolation are the extension of linear interpolation for interpolating functions of two or three variables. [50] Then the new position of interface point can be computed using the following equation:

$$P' = \begin{Bmatrix} P.x + \vec{V}_i(x) * \Delta t \\ P.y + \vec{V}_i(y) * \Delta t \\ P.z + \vec{V}_i(z) * \Delta t \end{Bmatrix} \qquad (4\text{-}14)$$

Where $\Delta t$ is the change of time of each evolution step.

## 4.5 Texture mapping

After evolving the interface to the circle or sphere, we can easily parameterize the circle or sphere interface using functions described in 4.3.2. Then, the correspondence between some physical points on the interface and the texture coordinates can be given to finish the texture mapping.

In this part we will use the three dimensional interface and two dimensional texture to describe our method for texture mapping point sampled surfaces.

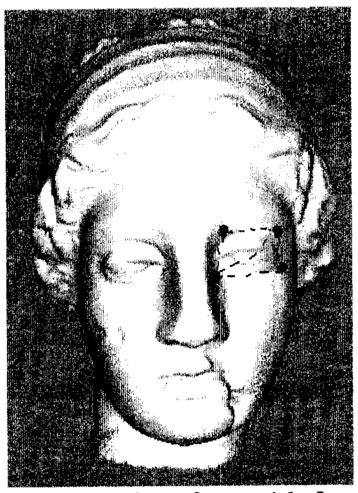
### 4.5.1 Indicate texture coordinates using anchor vertices

We use three or more anchor vertices on the surface to indicate the correspondence between the surface points and texture coordinates. For example, in Figure 4.26, we can just indicate the texture coordinates of four points on the eye. Then the texture is mapped as shown in Figure 4.26 (c).

### 4.5.2 Calculating texture coordinates

After creating the correspondence, we need to calculate the texture coordinates of all the surface points. For this we have adopted a simple technique of tessellating the texture region using triangulation, so that we can use simple bi-linear interpolation in texture coordinate space, *cf* Figure 4.26 (b). Other interpolation techniques, say for example, radial basis functions could also be used.

As we know after evolving the interface to sphere, there is a corresponding sphere point $(x_o, y_o, z_o) \to (x_s, y_s, z_s)$ for each surface point, and for each sphere point there is a unique polar coordinate: $(x_s, y_s, z_s) \to (\theta, \phi)$. By combining these two mappings, we can have a map from the surface points to polar coordinates: X: $(x_o, y_o, z_o) \to (\theta, \phi)$.


(a) The original surface with four points

(b) The texture

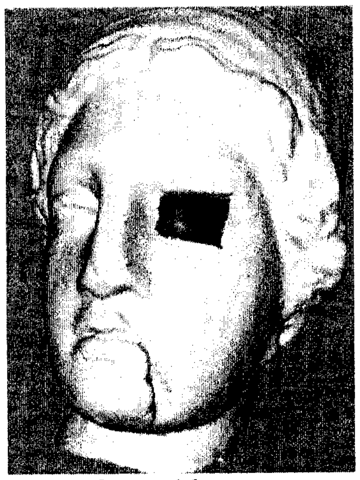

(c) The surface with texture mapped

**Figure 4.26:** Using anchor points to map texture

Using the above X map we can transfer the issue to check whether a point is in the texture triangle in two dimensions. We use the cross products method [51] to do the check and the procedure is as follows.

In Figure 4.27, we have a triangle with three vertices A, B, and C. Lines AB, BC and CA are three edges. And each edge splits the space in half and one of those halves is entirely outside the triangle. For example, if a point is inside the triangle ABC it must be below AB and left of BC and right of AC. If any one of these tests fails we can return false.

Suppose there is a point $P$ outside the triangle, if we take the cross product of [B-A] and [P-A], you'll get a vector pointing out of the screen. On the other hand, if you take the cross product of [B-A] and [p'-A] you'll get a vector pointing into the screen. From Figure 4.27 we can see that [B-A] cross [p'-A] points in the same direction as [B-A] cross [C-A], so we say p' may be inside the triangle. Then we need to test p' with the other lines as well. *If the point was on the same side of AB as C and is also on the same side of BC as A and on the same side of CA as B, then it is in the triangle.*

If we found the point p' is inside the triangle, we can use bi-linear interpolation to interpolate the texture coordinate $(u,v)$ of the point.

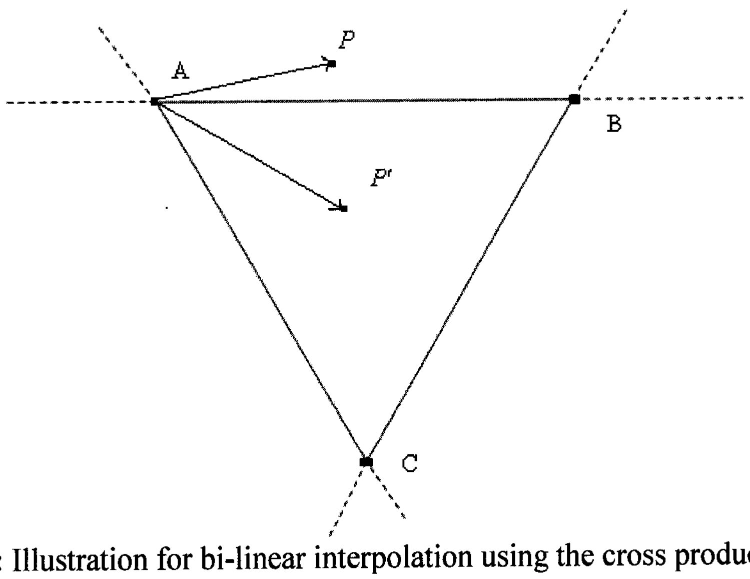


**Figure 4.27:** Illustration for bi-linear interpolation using the cross product method

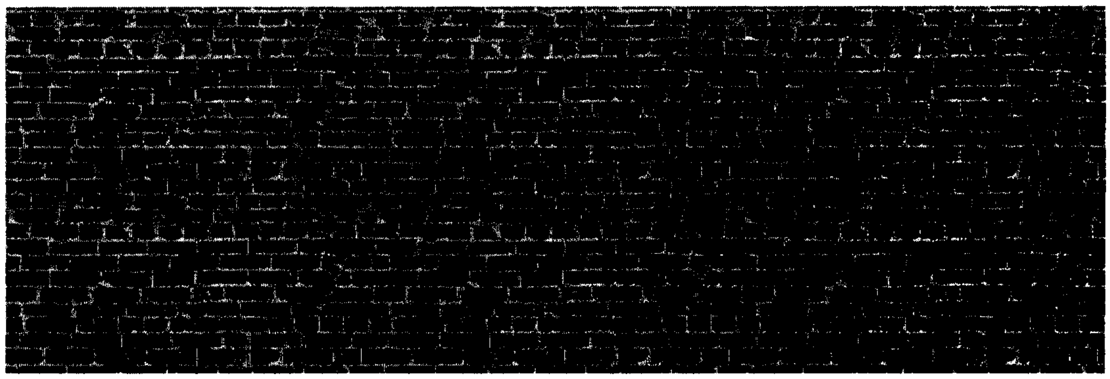
### *4.5.3 Mapping color*

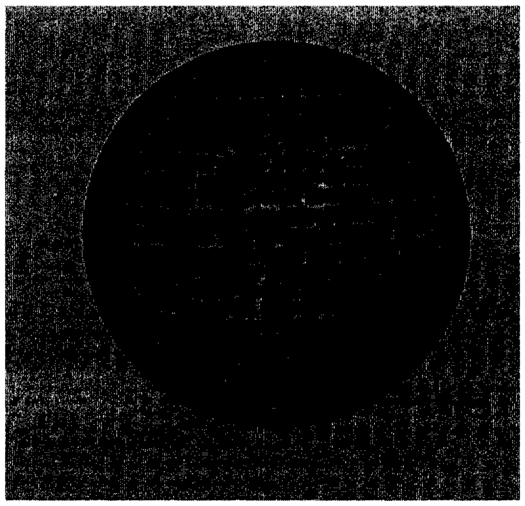
When indicating the texture coordinates for the anchor vertices, the texture ID is also assigned. In our system, the texture ID can be from 0 to 7, that is totally 8 textures can be imported at the same time.

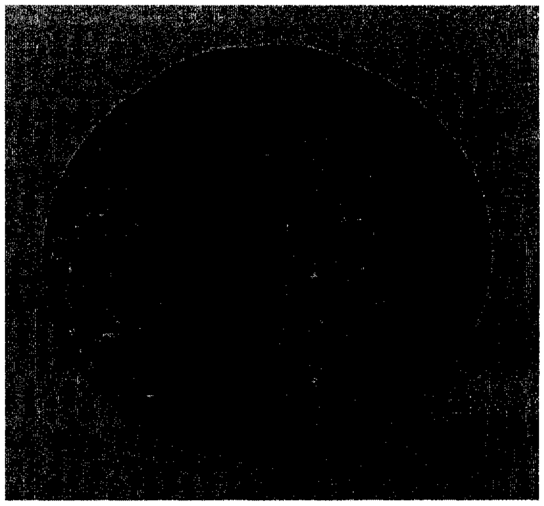
Based on the texture ID and the computed texture coordinates $(u,v)$, we can easily extract the color and write the color information into the interface point structure.

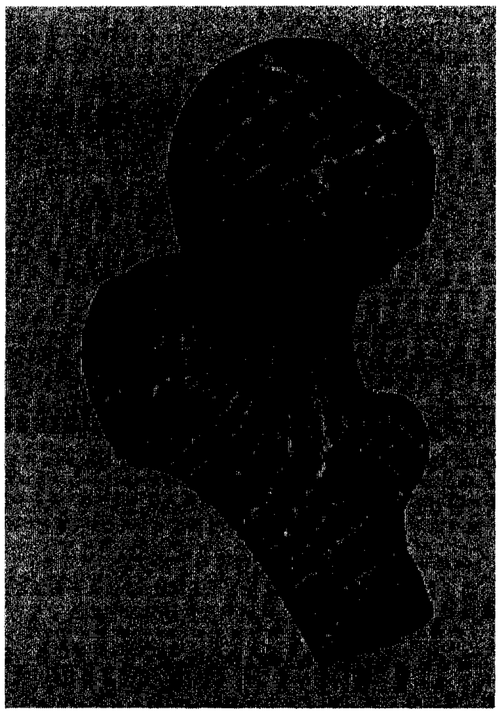
## ***4.6 Texture Mapping results and examples***

In this part we show some texture mapping results.


(a) Brick texture


(b) Sphere with brick texture


(c) 3D star with brick texture


(d) Balljoint with brick texture

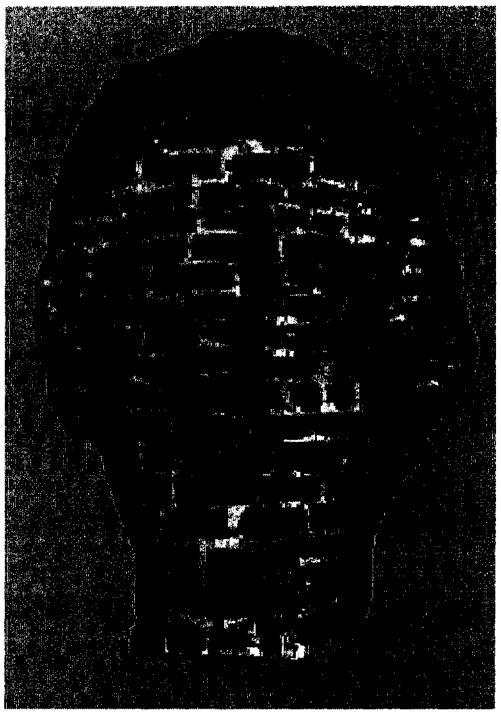

(e) Igea model with brick texture

**Figure 4.28:** Different point based models with brick texture

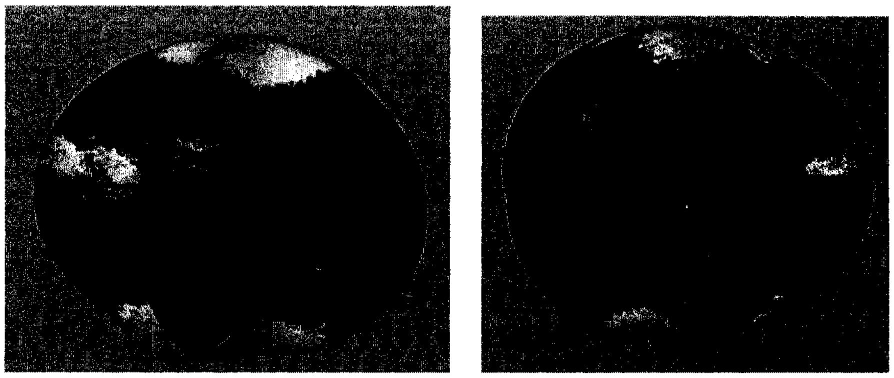


**Figure 4.29:** 3D star with earth texture

# Chapter 5: Numeric Implementation

In the previous chapter we introduced the basic theory and algorithms used in our method to do texture mapping on point-based geometry. In this chapter, significant issues related to the software implementation of the method are discussed.

## 5.1 Grid

One key issue for evolving interface using level set methods is to represent the interface with implicit function. Further to facilitate numerical implementation of level set based interface evolution, the implicit function should be discretized. There are many ways to discretize the implicit function. We have chosen uniform Cartesian grids to discretize the implicit function. In this way we can use the simple array[1] data structure to store the grid information.

The uniform Cartesian grid is defined as:

2D grid $\{(x_i, y_j) \mid 1 \le i \le M, 1 \le j \le N\}$

Or 3D grid $\{(x_i, y_j, z_k) \mid 1 \le i \le M, 1 \le j \le N, 1 \le k \le P\}$

where $(x_i, y_j, z_k)$ is the computational coordinates of the $(i, j, k)$ grid node

and $\Delta x = x_i - x_{i-1} = x_{i+1} - x_i \quad \Delta y = y_j - y_{j-1} = y_{j+1} - y_j \quad \Delta z = z_j - z_{j-1} = z_{j+1} - z_j$ .

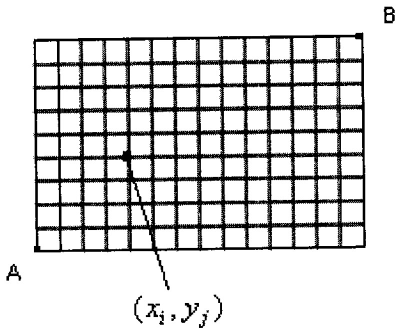
$M$, $N$ and $P$ indicate the number of grid nodes in X, Y and Z dimension respectively.

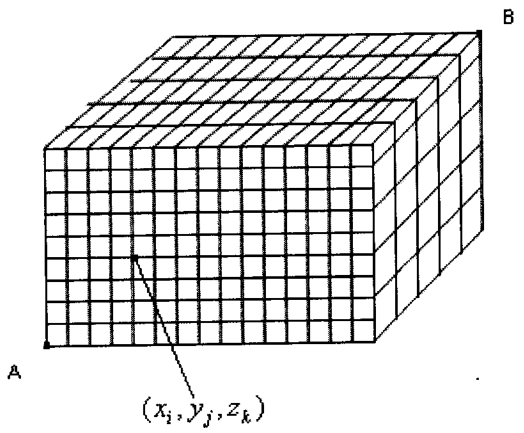
In our software, a basic grid class Target_grid and two sub classes, TwoD_Grid and ThreeD_Grid, are created to store the 2D and 3D grid information. For the detailed structure, please refer to Appendix A. It may be noted that 2D interfaces and grids have

[1] A hierarchical data structure such as octree could have been used to reduce memory requirements. However, since the goal of this development was more to carry out the numerical implementation and validate the applicability of our method than to create a memory efficient software implementation, we decided to keep the design simple to debug and test.

been implemented by us primarily to test out our method's applicability and also to produce explanatory illustrations as already seen in this thesis.

The key feature of grids is the range of the grid which is decided by the coordinates of A and B points in Figure 5-1. Other grid quantities include the dimension of grid, the gap $\Delta x$, $\Delta y$ and $\Delta z$ information. The grid class also provides member functions to calculate the coordinates of each grid node identified by the node index $(i, j, k)$.





(a) Two-dimensional grid (b) Three-dimensional grid

**Figure 5.1:** Two-dimensional and three-dimensional grid

In Figure 5.1, given the coordinates of point A and the gaps $\Delta x$, $\Delta y$ and $\Delta z$, we can compute the coordinate of any grid node $(i, j, k)$ as follows:

$$(x_i, y_j) = (A.x + i * \Delta x, A.y + j * \Delta y)$$
$$(x_i, y_j, z_k) = (A.x + i * \Delta x, A.y + j * \Delta y, A.z + k * \Delta z)$$

The grid size can be set manually or can be derived from the SFL file. When converting the SFL file into point data file and surface normal file (refer section 3.6), the maximum and minimum value in each dimension can be got. Then we can use these values to decide A and B: $A = (\min\_X, \min\_Y, \min\_Z)$ and

$B = (\max\_X, \max\_Y, \max\_Z)$. The number of grid elements is chosen such that it is smaller than the minimum radius for any surfel. Heuristically we have chosen it as 0.7 times the radius. Alternatively this number is also allowed to be set by the user considering computer system's memory.

As we know, using implicit function to represent interface, in most cases the interface point will be located somewhere within a grid cell. But using level set, we have only the computed information at the grid nodes, for example, the velocity, normal etc. Hence, to get the velocity or normal of the interface point, we will need to do interpolation using the values at vertices of the corresponding grid cell. Bilinear or trilinear interpolation methods are commonly used.

## 5.2 Partial derivative

The computation of partial derivative plays a key role in the level set methods. Below the approximation methods to the spatial and temporal partial derivatives are discussed.

### 5.2.1 Temporal approximation

In equation (4-2), $\phi_t$ is the derivative of function $\phi$ in time direction. Practical experience suggests that level set methods are more sensitive to spatial accuracy than to temporal approximation. So we approximate the temporal derivative using the first-order accurate forward Euler method as follows:

$$\phi_t = \frac{\phi^{n+1} - \phi^n}{\Delta t} \qquad (5\text{-}1)$$

Where $\phi^n$ is the function value at $t = n$, $\Delta t$ is a time step.

The forward Euler method is of first-order accuracy, and if we want higher order accuracy for the temporal approximation to the derivative, we can use the total variation diminishing (TVD) Runge-Kutta (RK) methods proposed by Shu and Osher in [52].

In our system, two functions have been created to yield the temporal derivative:

- odecfl1(...)

  This function is used to approximate the temporal derivative using equation (5-1) with first-order accuracy.

- odecfl2(...)

  This function approximates the temporal derivative in second-order accuracy, and uses following equations:

$$\frac{\phi^{n+1}-\phi^{n}}{\Delta t}+\vec{V}^{n}\cdot\nabla\phi^{n}=0 \qquad (5\text{-}2)$$

$$\frac{\phi^{n+2}-\phi^{n+1}}{\Delta t}+\vec{V}^{n+1}\cdot\nabla\phi^{n+1}=0 \qquad (5\text{-}3)$$

Then average of the above two results is taken to get:

$$\phi^{n+1}=\frac{1}{2}\phi^{n}+\frac{1}{2}\phi^{n+2} \qquad (5\text{-}4)$$

### *5.2.2 Spatial partial derivative*

When computing equations (4-2) and (4-3), we need the partial derivative in each direction. Equations (3-11), (3-12) and (3-13) give three ways to compute the derivative. Naively, one might evaluate the spatial derivatives in a straightforward manner using those equations. Unfortunately, randomly choosing an equation will fail. Instead upwind differencing method will be used to choose the correct equation to get accurate and stable computation of partial derivative.

For simplicity, we use the one-dimensional version of equation (4-2) as an example to describe upwind differencing method.

$$\phi_t^n + u^n \phi_x^n = 0 \qquad (5\text{-}5)$$

Where the sign of $u^n$ indicates whether the values of $\phi$ are moving to the right or to the left. Given a grid point $x_i$, then equation (5-5) can be rewritten as:

$$(\phi_t)_i^n + u_i^n (\phi_x)_i^n = 0 \qquad (5\text{-}6)$$

Where $(\phi_t)_i^n$ denotes the temporal derivative of $\phi$ at the point $x_i$ at time $n$ and $(\phi_x)_i^n$ denotes the spatial derivative of $\phi$ at the point $x_i$ at time $n$. If $u_i^n > 0$, the values of $\phi$ are moving from the left to right, and the method of characteristics tells us to look to the left of $x_i$ to determine what value of $\phi$ will land on the point $x_i$ at the end of a time step, leading us to use equation (3-12) $(\phi_x^-)_i$ to approximate the derivative. Similarly, if $u_i^n < 0$, equation (3-11) $(\phi_x^+)_i$ should be used to approximate the derivative. This method of choosing an approximation to the spatial derivatives based on the sign of $u$ is known as upwind differencing.

In the above discussion we use equation (3-11) or (3-12) to approximate the derivative, which is of first-order accuracy. The higher order accurate approximation to the spatial derivative can be realized by using the Hamilton-Jacobi of essentially non-oscillatory (ENO) and weighted ENO (WENO) methods, discussed below.

The idea ENO polynomial interpolation of data for the numerical solution of conservation laws was first introduced by Harten etc. in [53]. Then it was improved further by Shu and Osher in [52]. The HJ ENO method allows one to extend first-order accurate upwind differencing to higher-order spatial accuracy.

In [54] Liu et al proposed a WENO that takes a convex combination of the three ENO approximations.

In our system, we realized the first-order and second-order accurate ENO method to approximate the spatial derivative: *UpwindFirstFirst* and *UpwindFirstENO2*. For the second-order accurate approximation, it uses the Newton polynomial interpolation to find $\phi$ and then differentiates it to get $\phi_x$. The zeroth divided difference of $\phi$ is defined by

$$D_i^0\phi = \phi_i \qquad (5\text{-}7)$$

The first divided difference is defined midway between grid nodes as

$$D_{i+1/2}^1\phi = \frac{D_{i+1}^0\phi - D_i^0\phi}{\Delta x} \qquad (5\text{-}8)$$

The second divided difference is defined as: $D_i^2\phi = \dfrac{D_{i+1/2}^1\phi - D_{i-1/2}^1\phi}{2\Delta x}$ (5-9)

The divided differences are used to reconstruct a polynomial of the form

$$\phi(x) = Q_0(x) + Q_1(x) + Q_2(x) \qquad (5\text{-}10)$$

Where $Q_0(x) = c$ is a constant

$$Q_1(x) = (D_{k+1/2}^1\phi)(x - x_i) \qquad (5\text{-}11)$$

$$Q_2(x) = c(x - x_k)(x - x_{k+1}) \qquad (5\text{-}12)$$

Equation (5-10) can be differentiated to compute the spatial derivative $(\phi_x^+)_i$ and $(\phi_x^-)_i$ as follows:

$$\phi_x(x_i) = Q_1'(x_i) + Q_2'(x_i) \qquad (5\text{-}13)$$

$$Q_1'(x_i) = D_{k+1/2}^1\phi$$

To define $(\phi_x^-)_i$, we use equation (5-10) starting with $k = i - 1$, and to define $(\phi_x^+)_i$, we start with $k = i$.

Equation (5-12) is a little complex. If $|D_k^2\phi| \le |D_{k+1}^2\phi|$, we set $c = D_k^2\phi$ otherwise we set $c = D_{k+1}^2\phi$. Then we get

$$Q_2'(x_i) = c(2(i-k)-1)\Delta x \qquad (5\text{-}14)$$

This gives us the second order approximation to the spatial derivative.

### 5.2.3 CFL condition

The combination of the forward Euler time discretization with the upwind difference scheme is a consistent finite difference approximation to the partial differential equation (4-2), since the approximation error converges to zero as $\Delta t \to 0$ and $\Delta x \to 0$. However, this approximation is convergent if and only if it is both consistent and stable. Stability guarantees that small errors in the approximation are not amplified as the solution is marched forward in time.

In our system we use the Courant-Friedreichs-Lewy condition (CFL condition) to enforce the stability. This condition asserts that the numerical waves should propagate at least as fast as the physical waves. This means that the numerical wave speed, $\Delta x / \Delta t$, must be faster than the physical wave speed $|u|$. This leads to the CFL time step restriction as:

$$\Delta t < \frac{\Delta x}{\max\{|u|\}} \qquad (5\text{-}13)$$

Where, $\max\{|u|\}$ is the largest value of $|u|$ over the entire Cartesian grid. In reality we only need to choose the largest value of $|u|$ at the interface. Usually we give a CFL number $\alpha$ and $0 < \alpha < 1$. Then from equation (5-13) we get

$$\Delta t \frac{\max\{|u|\}}{\Delta x} = \alpha \qquad (5\text{-}14)$$

If $\alpha$ is close to 0, then each time step will be very small resulting in slow evolution; on the other hand if $\alpha$ is close to 1, there is high risk of it not being stable. So in our software we set $\alpha = 0.5$ to restrict the time step $\Delta t$.

## 5.3 Curvature computation

In this part we will discuss the approximation to the curvature which is important when evolving the interface dependent on curvature.

From the definitions in equations (3-16) and (3-17) of curvature, we know that the level set equation (4-7) of motion dependent on curvature is a parabolic equation. The derivative can not be approximated by the upwind difference method, because the value of $\phi$ at next time step is not decided by the previous value in a direction; instead it depends on the previous values in every direction. So the central differencing method, given for example, in equation (3-13), has to be used.

For the second order derivative, equations (3-18) and (3-19) are used. We can then substitute all the first order and second derivative values into equation (3-16) or (3-17) to compute the curvature. And the computed curvature will be of second order accuracy.

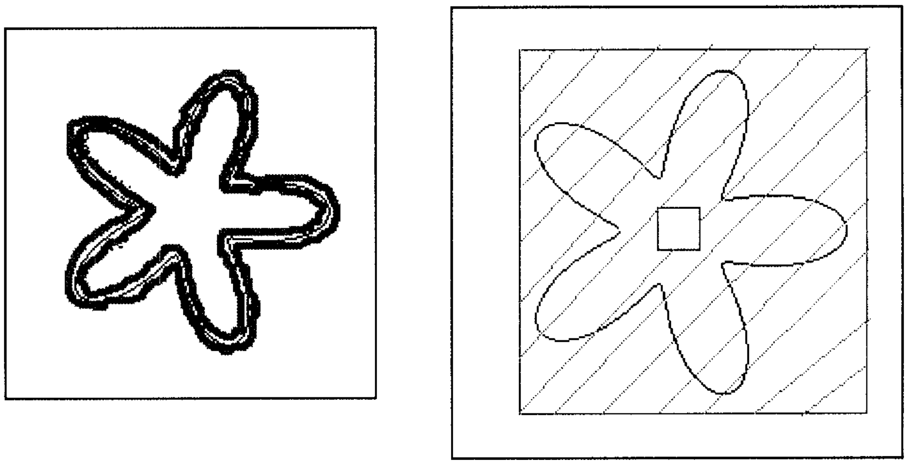
In our software, *curvature_second* function is used to compute curvature, and *compute_first_Second_de* function is used to compute the first and second order derivatives.

## 5.4 Accelerating the computations

Using level set methods to evolve the interface, the time cost can be a bottleneck for texture mapping, because at each time step, we have to compute the partial derivatives for each grid node of the whole grid and then modify the level value $\phi$ at each grid node.

For example, for the three dimensional grid with the size of $101 \times 101 \times 101$, at each time step, we need to compute the three partial derivatives in three dimensions. That is, we have to call the function to compute derivatives $3 \times 101 \times 101 \times 101$ times. After that we have to change the level values at each of the $101 \times 101 \times 101$ grid nodes.

However, in reality we are only concerned with the movement of the interface or a narrow band around the interface, and we do not need the evolution of level value of the points far off the interface. So if we can limit the computation to a band range which is close to the interface, we can save time and speed up the evolution. There has been quite a lot of research on speeding up the level set method.[55][56][57]. The common idea is to localize the computational range. In [55] and [56] they both use a tube to localize the range as shown in Figure 5.2 (a) and the main difference is only in the actual method used to create the tube. The width of tube usually is about five to six grid distances on the each side of the interface.



(a) The tube close to the interface is used to speed

(b) The external box (Red line) and Internal box (green line) is used to speed

**Figure 5.2:** Using tube or bounding box to limit the computational range

By using the tube the computation cost can be reduced greatly to speed the evolution. In order to use the tube a special algorithm and data structure needs to be used to generate the tube, after each evolution.

In our case, we demonstrate that our method can also be speeded up by using two bounding boxes, external and internal bounding boxes as shown in Figure 5.2 (b). The external box is decided by the minimum and maximum value of the interface points' coordinates, and the coordinates of the left-bottom vertex is $(\min(x) - 5\Delta x, \min(y) - 5\Delta y)$, the coordinates of the right-top vertex is $(\max(x) + 5\Delta x, \max(y) + 5\Delta y)$. Similarly, the three dimensional external bounding box can be generated.

The internal bounding box is defined as the box within which there are no points up to five grid distances off the interface in the inward direction.

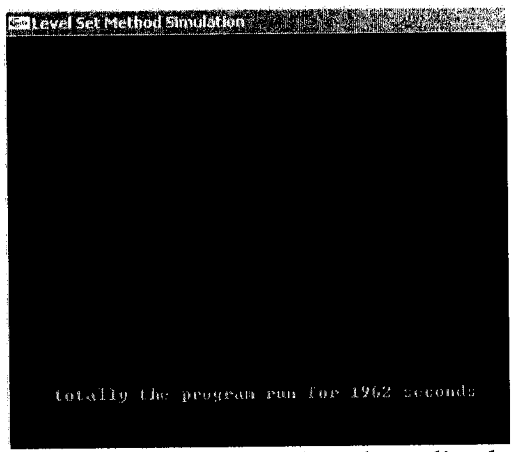
Then, for the grid points enclosed between these two bounding boxes we can easily use two-dimensional or three-dimensional array to store all the level values.

In order to keep the interface always between the two bounding boxes, the bounding boxes need to be updated as the interface evolves. There are several options we can choose to update the bounding boxes. First, update each box after each time step; this solution is easy to program and safe; however, it adds more computational cost towards update of the bounding boxes. Second, a bounding box can be updated whenever we find that the interface is close to the edge of a box. This solution also needs to spend some time to estimate the position of the interface. Third we can update the bounding box after fixed number of time steps. In our system we choose the third solution and update the box after five time steps. Because each time step is limited by the CFL condition and we have chosen the CFL number as 0.5 we can be sure that after five steps the interface will still lie between the two bounding boxes.

Figure 5.3 shows the evolution time of the 2D star. Figure 5.3 (a) shows that it takes 1962 seconds to complete the evolution without bounding box. Figure 5.3 (b) shows it that takes 1036 seconds with only the external box and Figure 5.3 (c) shows that it accelerates further to take only 976 seconds to complete evolution with two boxes. Table 5-1 shows that this technique accelerates computation for different models.

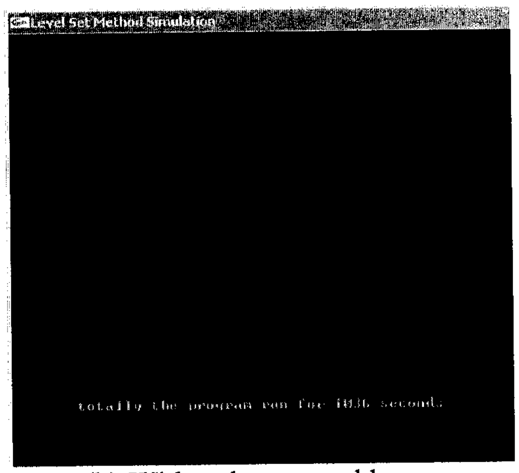
| **Model** | **Without bounding box** | **External bounding box** | **External and internal bounding box** |
|---|---|---|---|
| 2D star | 1962s | 1036s | 976s |
| 3D star | 6846s | 3287s | 3011s |
| 3D Igea model | 13063s | 8341s | 7853s |
| 3D balljoint model | 18178s | 10451s | 9983s |

Table 5-1: The accelerating result using bounding boxes



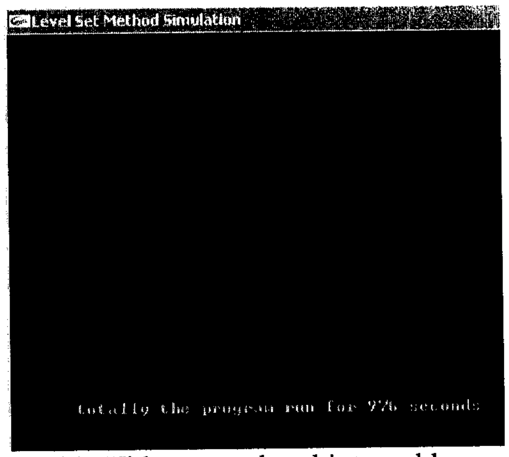
```
totally the program run for 1962 seconds
```

(a)Evolution of 2D star without bounding box



```
totally the program run for 1036 seconds
```

(b) With only external box



```
totally the program run for 976 seconds
```

(c) With external and internal box

**Figure 5.3:** Accelerating evolution computations using bounding box

## 5.5 Re-initialization

In this part we will discuss the method use to ensure that the implicit function remains as the signed distance function during the evolution process.

As discussed in Chapter 4, if the interface is represented by a signed distance function, the accuracy of calculation of partial derivative and curvature can be improved. So in our software we have chosen the implicit function as being the signed distance function.

For every kind of evolution, if we use SDF, we can use the simplified equations, such as equation (4-8) or (4-9). However, after each time step, the implicit function may not remain as SDF. So if we still want to use the simplified equation through out the evolution process then, we must re-initialize the function as SDF. This is a time-consuming procedure.

If we use the standard equations, such as equations (4-3), (4-5) or (4-7), we do not need to keep the implicit function as SDF. However, the accuracy of calculating partial derivative or curvature can not be guaranteed.

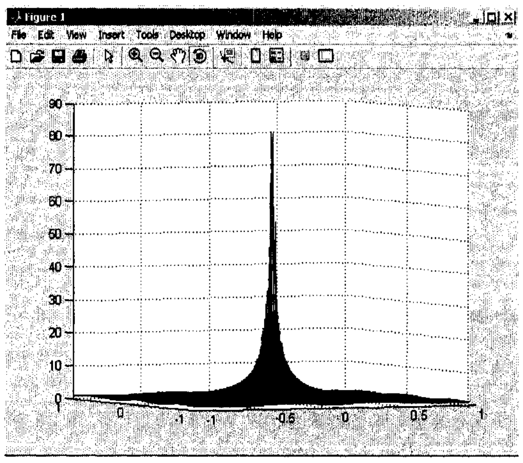
As a compromise we can choose SDF to represent the geometry, but use the common equations to evolve the interface. In this way, we do not need to initialize the implicit function in each step. Instead we re-initialize the function after every fixed number of steps.

Given the initial function $\phi$, we can use the crossing time method [31] to convert it to SDF. Based on the feature $|\nabla\phi| = 1$ of SDF, there is the following re-initialization equation:

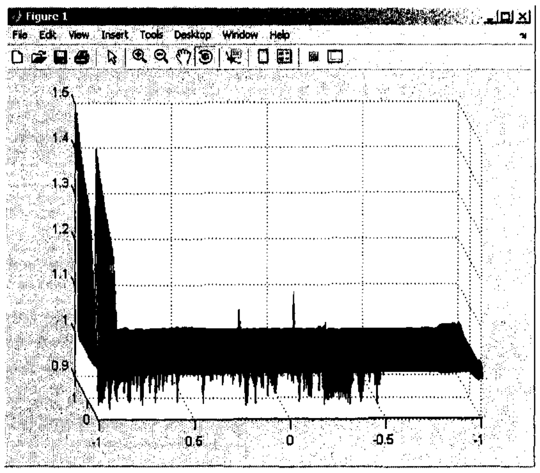
$$\phi_t + S(\phi_0)(|\nabla\phi| - 1) = 0 \qquad (5\text{-}15)$$

where $$S(\phi_0) = \frac{\phi_0}{\sqrt{\phi^2 + |\nabla\phi|^2 (\Delta x)^2}}$$

Level value of $\phi$ will be changed at each step, and when the value of $\phi$ is stable, the re-initialization can stop.



(a) The gradient distribution before re-initialization



(b) The gradient distribution after re-initialization

**Figure 5.4:** Gradient distribution comparison

We choose the 2D star model to show the effect of re-initialization. Figure 5.4 displays the gradient distribution of the original function and the SDF. From Figure 5.4 (a) we can see that in the center the gradient changes very quickly. On the other hand, Figure 5.4 (b) shows that, except for the edge of the grid, the gradients of all the grid nodes are around 1.0.

## 5.6 Software usage

In this part, the steps to use our software are briefly described.

### *5.6.1 Initialization:*

The first step is to initialize the startup environment, for example, to create the interface points array, the grid, and the level value array. There are three ways to do the initialization.

- From SFL file and the initial signed distance function

The SFL file is used to create the interface points array. Some features of the points, such as position, initial normal, surfel's radius, can be got from the SFL file. The converted SDF is analyzed to create the grid and level value array for each grid node.

- From the SFL file and the evolved signed distance function

To evolve the interface using level set methods is a time-consuming procedure, so in order save time, the evolved signed distance function and the grid information can be saved as a text file, in a preprocessing phase.

Next time if we want to restart mapping texture on the interface, we can directly read the evolved SDF to initialize the software. By this way, we don't need to evolve the interface again.

- User-defined interface and signed distance function

Sometimes we can create interface by ourselves. What we need to do is just save the interface points in one text file and provides other required information. The corresponding SDF also needs to be created.

### *5.6.2 Evolve*

Our software also provides other kinds of evolution procedures, for example, the convection with external velocity and the motion in normal direction with given velocity. So the first step is to choose the evolution type. The evolution type can be CONVECTION, NORMAL, CURVATURE, and TEXTUREMAP.

As we described before, there are many different ways to approximate the partial derivative and curvature, and the different ways have different order accuracy. The accuracy can be identified as: LOW, MIDDLE, HIGH, and VERY HIGH.

If the accuracy is LOW, we use the first-order accuracy in temporal and spatial derivatives' computation.

If the accuracy is MIDDLE, we use the first-order accuracy in temporal approximation, and the second-order accuracy in spatial derivatives' computation.

If the accuracy is HIGH or VERY HIGH, the temporal approximation will be of second-order accuracy, the spatial derivative's approximation is of third or fifth order accuracy.

### *5.6.3 Mapping texture*

In order to map texture on to the interface, first the textures should be imported into the memory to create texture array. The function is as:

*model->import_image*(Texture ID , "file name");

The texture ID can be any number between 0 and 7, and the file name is the name of a BMP file.

Then we can identify the some anchor points and assign them with texture coordinates to control the mapping of texture. If we don't do this identification, the zeroth texture with default texture coordinates will automatically be mapped onto the interface.

### *5.6.4 Output*

For the two dimensional interface, the result is displayed by our program directly on the screen. On the other hand, for 3D data, a modified SFL file is created of the point based model with mapped texture, and the SFL file can be rendered using Pointshop3D software.

### *5.6.5 Examples*

In this part we will use the Igea model to describe the texture mapping procedure.

- Input

  The Igea model represented in SFL model(Cf. Figure 5.5 (a)). The sfl2txt function and matlab program will be launched to generate the signed distance function for Igea model.

- Evolving interface

  The signed distance function and SFL file are read into software system, and the interface is evolved to a sphere's surface.

- Global Parameterization

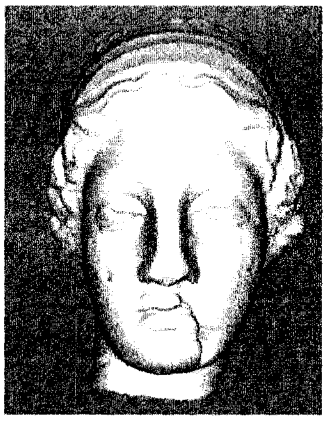
  By default there is a texture, whose texture ID is equal to 0, to create the global parameterization automatically. The anchor points can be indicated. (CF Figure 5.5 (c)).

- Different texture can be mapped on the model at the same time.

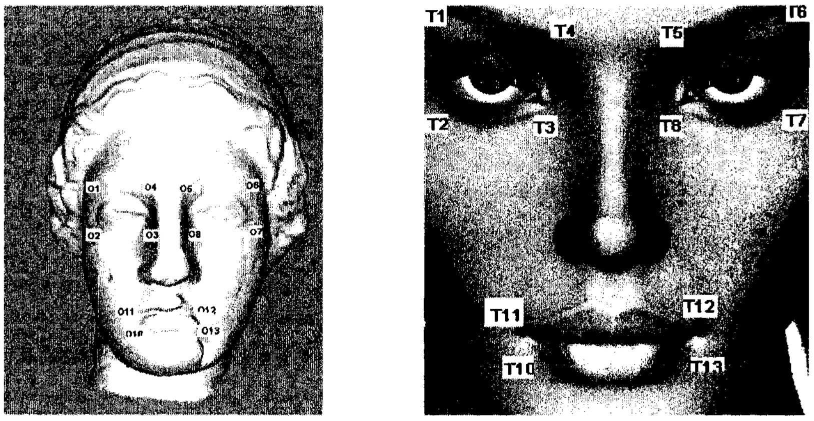
  For example, we want use another eye texture (CF Figure 5.5(b)) to map the eye patch.

- The rendering of objects with mapped texture

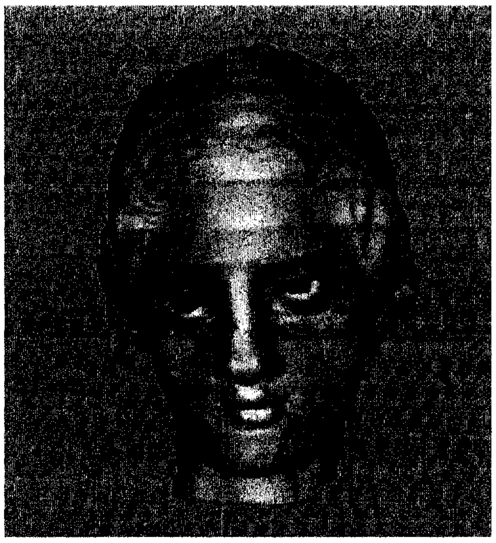
  Pointshop3D is used to render the objects. Figure 5.5(d) shows the results of the parameterization in Figure 5.5 (c). Figure 5.5 (e) shows the different eye texture.


(a) The original Igea model

(b) Another eye texture



(c) Define the anchor points on the texture and original model


(d) The result with texture in (c)

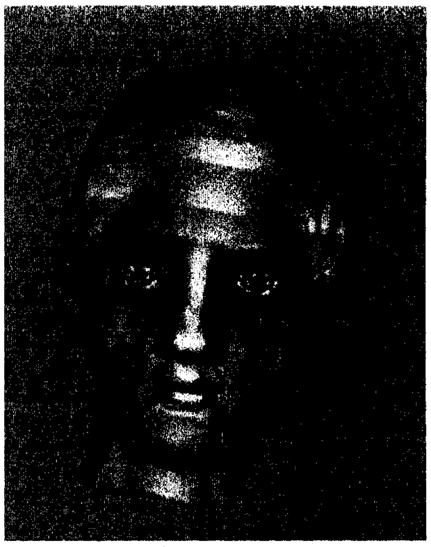

(e) The result with different eye texture in (b)

**Figure 5.5:** The texture mapping procedure with Igea model

# *Chapter 6: Conclusion and Future Work*

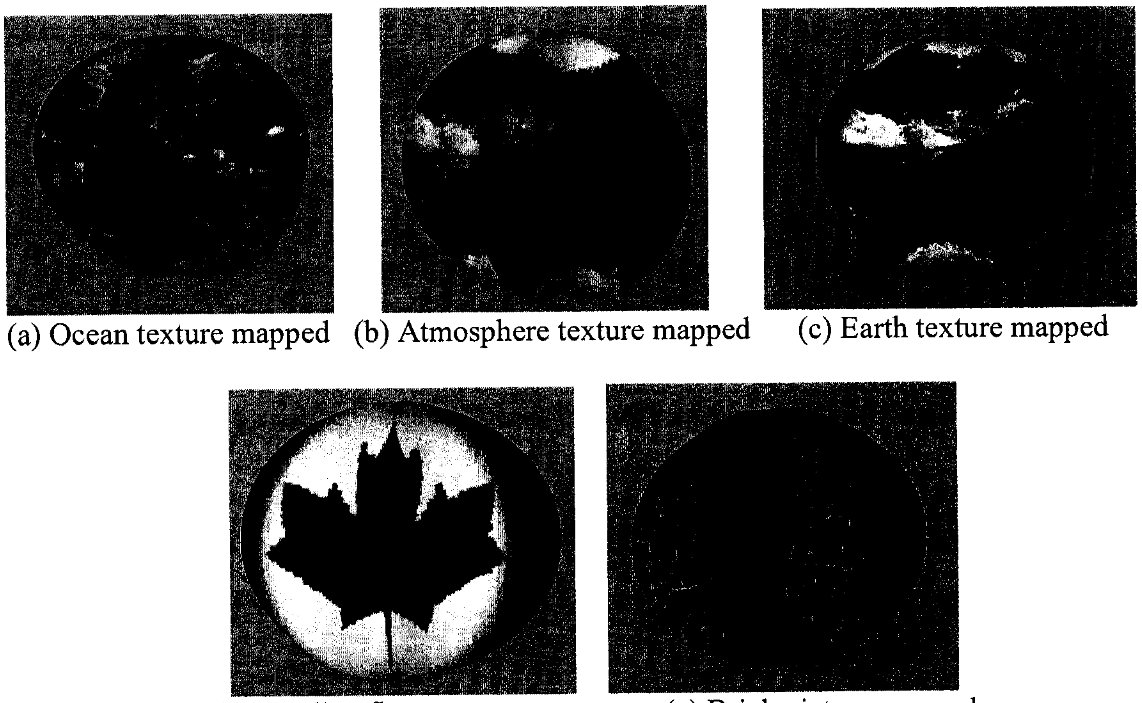
This chapter concludes the thesis with a summary of the main contributions, and some ideas for future research.

## *6.1 Principal Contributions*

The primary focus of the research reported in this thesis has been on techniques to map image based texture onto surfaces of point-based models with reduced human intervention. Based on an exhaustive research of available techniques and experience based on experimentation, we came to the decision that a global parameterization method is desirable. This is based on the observation that earlier techniques mandated break-up of the original surface into patches. In this context, the following contributions have been made in this thesis:

1. A solution has been proposed to globally parameterize the surface of point-based geometry without explicitly converting the surface to triangle mesh or other continuous surface representations. The proposed solution uses level set methods to evolve the surface to another surface with implied 2D parameterization and hence every point sample of the point based model can be assigned distinct 2D coordinates using polar coordinates.
2. While the process of evolving the surface of a point based model is computationally expensive, and difficult to carry out in real time except for small models, this process can be carried out in a pre-processing phase. The global parameterization obtained for the points are stored along with the model and used in every other texture mapping operation. For example, in Figure 6.1, the 3D star model has been mapped with different textures using the pre-processed global parameterization.

3. A procedure to do the conversion between SFL file and SDF has been created.
4. For experimenting and testing the results of the proposed texture mapping procedure, a software implementation was successfully completed. The software's input can be SFL, SDF or user-generated level set value. The output is SFL file which can be displayed directly using Pointshop3D software. Results from this software have been demonstrated in earlier chapters.


(a) Ocean texture mapped (b) Atmosphere texture mapped (c) Earth texture mapped

(d) Canadian flag as texture (e) Brick picture mapped

**Figure 6.1:** Different textures mapped onto the star model

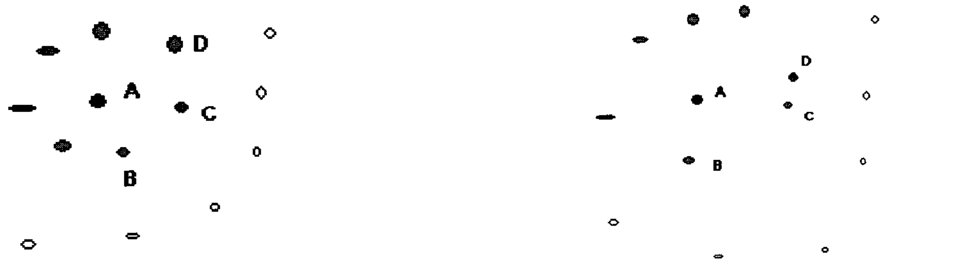
## *6.2 Future work*

In this research we have proposed a new solution based on level set techniques to derive a global parameterization that is then used to map texture onto a closed point-based object. However, there is still more work needed before this method can be widely used in practice.

The slow speed of evolving the interface is a bottleneck. The speed depends on the number of interface points which is determined by the object surface sampling rates, the grid size and resolution, and the accuracy to which the global parameterization is required. We have implemented a simple acceleration technique based on inner and outer bounding boxes. This was done so as to keep the data structure and computer implementation simple. However, a more complex inner/outer bounding shape closely matching the interface may reduce the running time further. So in the future this method should be incorporated in our software.

As described in Chapter 4, we can indicate the texture ID and assign texture coordinates to a set of anchor points to control texture mapping. It is better to integrate our system into Pointshop3D as a plug-in, so that anchor points and corresponding texture image coordinates can be identified interactively using a mouse. In this way, the 3D textured surface also can be displayed as part of the process of interactively carrying out texture mapping.

Another issue is about the analysis of the distortion when doing parameterization. In our system, we can keep the persistence of the neighborhood of points, but can not guarantee the distance ratio between points. For example, in Figure 6.2 (a) originally the red point A has seven green points as its neighborhood. In Figure 6.2 (b) these seven green points are still the neighbors of A, but the ratio of distance DC and BC is changed. This causes distortion in the resulting texture on the surface and a good solution needs further research.



a) Original neighborhood b) Evolved neighborhood

**Figure 6.2:** Illustration for texture distortion geometry

# References

[1] P.S.Heckbert. ***Fundamentals of Texture Mapping and Image Wrapping.*** Master thesis

[2] P.S. Heckbert. ***Survey of Texture Mapping***. IEEE Computer Graphics and Applications, pages 56-67, Nov. 1996.

[3] Les Piegl, ***On NURBS: a Survey***. IEEE Computer graphics and Application, Vol. 11, Pages: 55-71, 1991

[4] K.Perlin. ***An Image Synthesizer.*** Computer Graphics, (SIGGRAPH '85 Proceedings), Pages 287 – 296, July 1985

[5] M. Gross. ***Are points the better graphics primitives?*** Computer Graphics Forum, Vol. 20(3), 2001.

[6] Sainz, M. Pajarola, etc. ***A simple approach for point-based object capturing and rendering.*** IEEE Computer Graphics & Applications, Vol. 24(4) Page24-33, 2004

[7] Holger Wendland, ***Scattered Data Approximation***. Cambridge University Press, 2006

[8] C.Stoll, Z.Karni, etc., ***Template Deformation for Point Cloud Fitting***. Point Based Graphics 2006

[9] Leif Kobbelt and Mario Botsch, ***A Survey of Point-Based Techniques in Computer Graphics***. 2004

[10] Levoy M., Whitted T.: ***The use of Points as a display primitiv***e. Tech. Rep. 85-022, Computer Science Department, University of North Carolina, January, 1985

[11] M.Zwicker, H.Pfister, J.Van Baar, and M Gross. ***Surface Splatting.*** In SIGGRAPH 2001 Proceedings, Pages 371 – 378. August 2001

[12] J. P. Grossman and W. J. Dally. ***Point sample rendering.*** Rendering Techniques '98, Eurographics, Pages 181–192. 1998.

[13] A. Kalaiah and A. Varshney. ***Differential Point Rendering.*** Rendering Techniques '01, Springer Verlag, Pages 139-150, 2001.

[14] S. Rusinkiewicz and M. Levoy. ***QSplat: A multiresolution point rendering system for large meshes.*** *Siggraph 2000, Computer Graphics Proceedings,*pages 343–352. 2000.

[15] H.Pfister, M.Zwicker, J.van Baar, and M Gross. ***Surfels: Surface Elements as Rendering Primitives.*** In Computer Graphics, SIGGRAPH 2000 Proceedings, Pages 335-342. July 2000.

[16] M.Zwicker, M.Pauly, O. Knoll, and M Gross. ***Pointshop 3D: An Interactive System for Point-Based Surface Editing.*** In SIGGRAPH Proceedings, pages 322–329, July 2002

[17] Zh.Haitao, Q.Feng, and K.Arie. ***Fast Hybrid Approach for Texturing Point ModelsComputer Graphics Forum***, Vol. 23, Pages 715 – 725, 2004

[18] M.Alexa, T.Klug etc. ***Direction Fields over Point-Sampled Geometry.*** In SIGRAPH2001, pages417 – 424, Aug 2001

[19] W.Martin, O.Sandro and M.Gross. ***Conversion of Point-Sampled Models to Textured Meshes***. Proceedings of SIGGRAPH05, 2005

[20] N. Amenta, S.Choi etc. ***A Simple Algorithm for Homeomorphic Surface Reconstruction***. Intl. J. Comp. Geom and Appl. Vol. 12, Pages 125 – 141. 2002

[21] T. Dey, S.Goswami: Tight ***Cocone: A Water-tight Surface Reconstructor***. J.Computing. Inf. Sci. and Engin, Vol.3, pages 302 – 307, 2003

[22] Guo XiaoHu, etc. ***Meshless Thin-Shell Simulation Based on Global Conformal Parameterization***. IEEE Transactions on Visualization and Computer Graphics. Vol.12 No.3. Pages 375 – 385, May 2006

[23] Stanley Osher and Ronald Fedkiw. ***Level set Methods and Dynamic Implicit Surfaces.*** Springer Press, 2002.

[24] J.A. Sethian ***Level Set Methods and Fast Marching methods***, Cambridge University Press, 1999

[25] Floater, M.S., Hormann, K., ***Surface parameterization: a tutorial and survey***. In: Multiresolution in Geometric Modelling. Springer, Berlin. 2004

[26] B.Levy, S.Petitjean, N.Ray, and J.Maillot. ***Least squares conformal maps for automatic texture atlas generation***. In SIGGRAPH02, Pages 362 – 371, 2002

[27] M.Eck, T.Derose, and T.Duchamp et al. ***Multiresolution Analysis of Arbitrary Meshes***. ACM Siggraph Conf.Proc., pages 173 – 182, 1995

[28] B.Levy. ***Constrained texture mapping for polygonal meshes***. In SIGRAPH2001, pages417 – 424, Aug 2001

[29] Mark Pauly, ***Point primitives for Interactive Modeling and Processing of 3D Geometry***. Doctoral thesis, Federal Institute of Technology of Zurich, 2003

[30] Gu, Xiangfeng, Yao. ***Global Conformal Surface Parameterization***. Geometry Processing. Pages 127 – 137, 2003

[31] Stanley Osher and Ronald Fedkiw. ***Level set Methods and Dynamic Implicit Surfaces.*** Springer Press, 2002.

[32] J.A. Sethian ***Level Set Methods and Fast Marching methods***, Cambridge University Press, 1999

[33] SFL Format website: http://graphics.ethz.ch/pointshop3d/sfldoc/html/pages.html

[34] http://graphics.ethz.ch/pointshop3d/

[35] Yen-his Richard Tsai, ***Rapid and Accurate Computation of the Distance Function Using Grids***. Journal of Computational Physics 178, pages 175 – 195

[36] David E. Breen, S.Mauch, R.T.Whitaker, ***3D Scan Conversion of CSG Models into Distance Volumes***, Proceedings of the 1988 IEEE symposium on volume visualization, pages 7 – 14, 1998

[37] Mauch, Sean. 2000 (September). ***A Fast Algorithm for Computing the Closest Point and Distance Function.*** Tech. rept. CalTech. unpublished.

[38] Osher, S. and Sethian, J.A., ***Fronts propagating with curvature dependent speed: Algorithms based on Hamilton-Jacobi Formulations.*** J. comput.phys, pages 12-49 (1988)

[39] J.S. Langer, ***Instabilities and pattern formation in crystal growth***, Rev. mod Phys, Vol. 52, Pages 1-28, 1980

[40] J.S. Langer & H.Muller-Krumbhaar, ***Mode selection in a dendritelike nonlinear system***, Phys Rev, A 27, pages 499 – 514, 1983

[41] Pamplin, B.R., ***Crystal Growth***. New York, Pergammon Press, 1975

[42] Frankel, ML and Sivashinsky, ***The effect of viscosity on hydrodynamic stability of a plane flame front***. Comb. Sci. Tech., 29, pages. 207 - 224 (1982).

[43] Markstein, G.H., ***Non-Steady Flame Propagation***. Pergammon Press, MacMillan Company, New York, 1964

[44] Zabusky, N. & Overman, E. ***Tangential Regularization of contour dynamical algorithms.*** J. Comput. Phys. Vol. 52, Page 351–374, 1983.

[45] Per-Olof Persson, ***the level set Method lecture notes***. Massachusetts Institute of Technology Cambridge.

[46] J.A. Sethian, ***Level Set Methods: An act of violence***, *American Scientific*, 1997

[47] Ian M.Mitchell, ***A toolbox of level set methods version 1.1***. University of British Columbia.

[48] J.Strain, ***Tree Methods for Moving Interfaces*** Journal of Computational Physics, vol. 151. Pages 616 – 648, May 1999

[49] Grayson, M., ***The heat equation shrinks embedded plane curves to round points***. J.Diff. Geom. Vol.29, Pages 285 - 314, 1987

[50] http://en.wikipedia.org/wiki/Bilinear_interpolation

[51] http://www.blackpawn.com/texts/pointinpoly/default.html

[52] Shu, C.W. and Osher, S., Efficient ***Implementation of Essentially Non-Oscillatory Shock Capturing Schemes II*** J. Comput. Phys. Pages 32 – 78, 1989

[53] Harten, A. Egquist, B., Osher, S., and Chakravarthy, S., ***Uniformly High order Accurate Essentially Non-Oscillatory Schemes. III***, J.Comput. Phys. Vol. 71, Pages 231 -303, 1987

[54] Liu, X.-D, Osher, S., and Chan, T., ***Weighted Essentially Non-Oscillatory Schemes***. J. Comput. Phys, Vol.115, Pages 202 – 212, 1994

[55] Danping Peng, Barry Merriman, etc. ***A PDE-Based Fast local level set method***. Journal of computational physics, Vol. 155, Pages 410-438, 1999

[56] David A. and James. A.Sethian, ***A fast level set method for propagating interfaces***. Journal of Computational Physics, Vol. 118, Pages 269-277, 1995

[57] Chohong Min, ***Local level set method in high dimension and codimension,*** Journal of Computational Physics, Vol. 200, Pages 368-382, 2004

# Appendix A: Data Structure

In this appendix, we list the codes that define some basic data structure used in our software.

1. **Interface Point:** this structure define the data structure for the interface points

```
struct Interface_point
{
    //! The original coordinates which will be decide first
    float ori_X, ori_Y, ori_Z;

    //! the coordinates of the interface points after the evolve
    float cur_X, cur_Y, cur_Z;

    //! the texture ID, ID can be from 0 to 7
    int text_ID;

    //! The texture coordinates
    float text_U, text_V;

    //! The grid position corresponding to the interface points
    int grid_X, grid_Y, grid_Z;

    //! The normal
    float Normal_X, Normal_Y, Normal_Z;

    //! The current Normal
    float cur_nor_X, cur_nor_Y, cur_nor_Z;

    //! The color of the interface point
    float Color_r, Color_g, Color_b;

    //! The radius used to render using Pointshop, which is decide by the grid size
    float radius;
};
```

2. **Grid:** this is the base class for the grid

```
class Target_Grid
{
public:
    //! Default constructor
    Target_Grid()
    {
    }
    //! Destructor
    /* Here the destructor is defined as virtual function, because this class will work
as base class */
```

```
        virtual ~Target_Grid()
        {
        }

        //! To set grid parameters
        virtual void set_grid_parameters(grid_coordinate left, grid_coordinate right, float
    x_step, float y_step, float z_step=0);

        //! To return the grid coordinates based on the grid index
        virtual grid_coordinate get_coordinate(int X_index, int Y_index, int
    Z_index=0)=0;

        //! The two corner to define the size of grid array
        grid_coordinate left_bottom_front, right_top_back;

        //! To save the gap in each dimension
        float dx, dy, dz;

        /**
         * To save the grid points number in each dimension
         */
        Int_node N;
    };
class TwoD_Grid : public Target_Grid
{
        ...
};

class ThreeD_Grid : public Target_Grid
{
        ...
};
```