Automatic 3D Model Creation with Velocity-based Surface Deformations

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Risto F. Rangel-Kuoppa

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

The virtual worlds of Computer Graphics are populated by geometric objects, called models. Researchers have addressed the problem of synthesizing models automatically. Traditional modeling approaches often require a user to guide the synthesis process and to look after the geometry being synthesized, but user attention is expensive, and reducing user interaction is therefore desirable.

I present a scheme for the automatic creation of geometry by deforming surfaces. My scheme includes a novel surface representation; it is an explicit representation consisting of points and edges, but it is not a traditional polygonal mesh. The novel surface representation is paired with a resampling policy to control the surface density and its evolution during deformation. The surface deforms with velocities assigned to its points through a set of deformation operators. Deformation operators avoid the manual computation and assignment of velocities, the operators allow a user to interactively assign velocities with minimal effort. Additionally, Petri nets are used to automatically deform a surface by mimicking a user assigning deformation operators. Furthermore, I present an algorithm to translate from the novel surface representations to a polygonal mesh.

I demonstrate the utility of my model generation scheme with a gallery of models created automatically. The scheme's surface representation and resampling policy enables a surface to deform without requiring a user to control the deformation; self-intersections and hole creation are automatically prevented. The generated models show that my scheme is well suited to create organic-like models, whose surfaces have smooth transitions between surface features, but can also produce other kinds of models. My scheme allows a user to automatically generate varied instances of richly detailed models with minimal user interaction.

Acknowledgements

My beloved parents, without whom I would not be the person I am today and who are the best parents I could have. My beloved fiancée, Idalia, for being with me and accepting to share our lives together.

My supervisor, David Mould, who has taught me so many things and for his big patience towards my almost-hard-wired brain. To the staff of the course GSR 989 "Introduction to University Teaching", Ron Marken, Alec Aitken, Tereigh Ewert-Bauer, and Kim West, for the best course I have ever attended in regards to teaching and academic professional life. To Jim Greer for helping me find my true research interests. To the instructors of courses I attended: Eric Neufeld, Mark Eramian, Ralph Deters, Derek Eager, Michael Horsch, and Dwight Makaroff. To the faculty and students of the IMG group that shared knowledge, experiences, and friendship.

To the staff of the Computer Science Department for their great support in "keeping the things running". In particular, to Jan Thompson and Lori Kettel for being lovely friends and for their impecable jobs.

To CONACyT (Mexican Council for Science and Technology) for their funding and support of my PhD studies, and UAM (Universidad Autnoma Metropolitana-Mexico) for their support in my PhD studies.

To all the others that I have mistakenly forgotten to mention here.

Gracias a todos!

I dedicate my thesis to all people who believe the best teacher is the one who teaches how to think and not what to think.

I also dedicate my thesis to all those who can laugh at the following dialog knowing true wealthiness is achieved by knowledge and self achievements:

Bart: [after they watch a film] "I was so bored I cut the pony tail off the guy in front of us". [holds pony tail to his head].

Bart: "Look at me, I'm a grad student. I'm 30 years old and I made $600 last year".

Marge: "Bart!, don't make fun of grad students. They've just made a terrible life choice".

Simpson episode *Home Away from Homer*, Season 16 No. 20.

Contents

List of Tables

List of Figures

List of Abbreviations

NURBS	Non-Uniform Rational B-Spline
norm	Normalization of a vector
cross	Cross product of two vectors
dot	Dot product of two vectors
avg	Average of a series of vectors

Chapter 1

Introduction

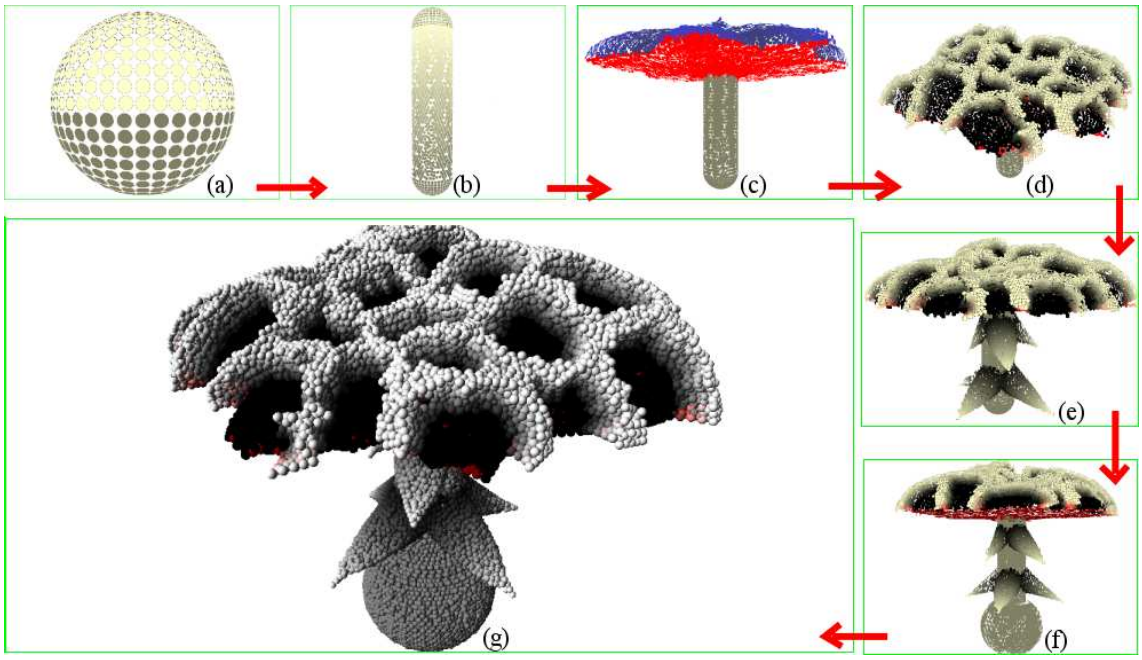

Figure 1.1: Steps to automatically create rich and detailed models.

1.1 Motivation

Most Computer Graphics experts recognize that the three major areas of the field are Modeling, Rendering, and Animation [86]. The Modeling area covers the different ways to specify and store in a computer the shape and appearance of objects. In Computer Graphics, the term "model" refers to the representation of an object's surface, such as a polygonal mesh. The Rendering area encompasses the different techniques to create images containing models; the problems solved within the area vary from projecting 3D models onto 2D planes to computing the color of each point in the scene. Finally, the Animation area encloses the techniques to create the illusion of motion in a sequence of images.

In Computer Graphics, different techniques for the creation of models have been reported. Existing techniques support both manual and automatic model creation. Manual model creation consists of a user defining each individual element of the model. Automatic Model Creation consists of an algorithmic procedure to define the elements of the model. Both approaches have their advantages and disadvantages. Manual creation generates precise and richly detailed models, but requires a lot of time. Automatic approach facilitates the fast generation of models, but they often lack in variety. My research aims at the Automatic Model Creation area. I have chosen this area because of my interest in reducing the time to create 3D models. Additionally, the technology has reached a state in which direct creation of models is easily supported, allowing the exploration of elaborated techniques for automatic model creation.

My research focuses on the automatic creation of organic-like models. I have chosen organic-like models because of their great demand in Computer Graphics and a personal interest. I consider organic-like models as those that often lack sharp edges, have smooth surface transitions, and where surface properties (such as color) vary slightly between surface regions. For example, a frog has smooth variations on its skin, it lacks sharp edges, and it has colored regions on the skin.

Historically, Computer Graphics researchers have used nature as a guide to create models. Researchers have used different descriptions of nature to recreate the appearance of nature's objects. In my case, I use deformations to mimic the surface changes of organic objects when they grow into a specific shape. I have chosen deformations because I consider them to be intuitive for a user/designer to describe changes on a surface.

Deformations simplify the automatic creation of varied models. A complex model can be seen as the result of a deformation process; nature contains a myriad of complex objects that result from deformations. If the deformation process can be broke down into basic deformations, one can achieve a great variety of models by rearranging the basic deformations. Furthermore, the parameterization of the basic deformations increases the models variety.

This thesis describes my research in the model creation area. I have provided an Automatic Model Creation approach for organic-like models. My approach for Automatic Model Creation consists in deforming a simple initial surface, and shaping it into a desired model. The deformations are assigned with a minimum of user interaction and can be parameterized; that is, deformations can be used to create different instances of the same type of model by simply changing some parameters. Also, the deformations can be applied to create either local 3D detail on the model surface or large shape changes. Figure 1.1 illustrates my Automatic Model Creation approach; Figure 1.1.a shows a simple initial surface (a sphere); the initial surface deforms creating a mushroom-like shape, then the surface deforms at a lower scale to create surface details, and finally the model is enhanced by applying a deformation with different parameters that creates thorn-like details and a bulbous root (see Figure 1.1.e-f).

My Automatic Model Creation approach involves solving several problems. The first problem is the need of a Surface Representation that renders as a continuous surface. Second, the specification of deformations must be intuitive for a user/designer. Third, the surface must present a regular surface so that deformation can be applied easily. Fourth, the model creation process must be simple and have a consistent behavior throughout the deformations and must not be difficult to maintain (technically). Fifth, deformations should be able to create large scale shape changes as well as small surface details. Sixth, deformations must be applied automatically in order to fulfill the goal of automatically creating models. Finally, the resulting models must be easy to use by other applications. I have solved all these problems during my research, and their solutions are explained in detail throughout the rest of this thesis. Next, I provide an explanation of my contributions:

1. I solve the problem of a surface representation that renders as a continuous surface with my proposed surface representation. My surface representation is a hybrid between pure-point surfaces and polygonal meshes. Therefore I follow a surface-based modeling approach instead of solid modeling as others in Computer Graphics. I have named it as "Volipoc" from "**Vo**latile-**Li**nked **Po**int **C**loud". Volipoc consists of a point of clouds and volatile edges that link pairs of points, although a continuous surface is represented. The continuous surface is computed by interpolating between edges of the surface representation.

2. I specify deformations in terms of velocities. Velocities are easily understood by a user/designer and are easily assigned to the surface representation; each point of the surface representation has a velocity associated with it.

3. I kept the surface point density constant with my resampling mechanism in order to facilitate assignment of velocities. My resampling consists of creating or deleting points if neighboring points are too far or too close, respectively. Neighborhood is defined by the volatile edges. Volatile edges are also created or deleted during resampling.

4. I restricted topology changes because such changes complicate the design of sequences of deformations. My resampling process is designed to prevent new holes from appearing in surfaces, and to make it difficult for existing holes to be removed. If a model maintains the same topology during a sequence of deformation processes, each deformation can be designed independently. The user can be confident that a deformation operation will not be applied to a model with different topology than what it was designed for. Although I recognize that topology changes may be desirable in some cases (as when creating a window on a model). I chose to start by avoiding topology changes. Topology changes were prevented by avoiding the creation of holes on the restricted surface and the blend of surface segments. I used a space-partitioning data structure to detect imminent blends. Hole creation is avoided with

the resampling mechanism. Although under particular arrangements of deformations, existing holes can be filled in. However, new holes can never be created.

5. I defined a set of deformation operators capable of creating local changes on the surface as well as shape changes. Deformation operators are a set of parameterizable procedures that select points from the model and assign velocities to points. The operators can assign deformations (through velocities assigned to surface points) at both local and global scales; that is, deformations can affect the whole surface or just a local zone.

6. I provide a high-level abstraction to automatically distribute deformation operators by mimicking a user/designer behavior. In particular, I used Petri nets to simulate the application of deformation operators at different stages of the model creation process.

7. I designed an algorithm that translates a model in Volipoc to a polygonal mesh representation. My algorithm does not make an approximation of the surface; instead, it chooses the edges and points that form a polygonal mesh.

I have integrated the aforementioned contributions into a single software system. The software ("Vebam" for **Ve**locity **B**ased **a**utomatic **M**odeling) reflects the framework for Automatic Model Creation that constitutes my research. That is, I have devised the surface representation, the surface behavior, the deformations specifications, and the deformations management to best support the generation of models whose surfaces are created by their deformation and that have smooth transitions along the surface as well as sharing surface properties along regions.

Even though my framework can be used as a universal modeler, it is best suited for the generation of organic-like models. This design makes it difficult to generate other types of models such as those containing sharp edges; however, it is still possible to create them. The sharp edges will present a smoothed transition in the boundary; however, this transition is less noticeable as the scale of the model increases.

1.2 Problem Statement

Computer Graphics is a field born from the combination of computing tools and the desire to replicate the visual beauty of objects. Nature contains many complex and beautiful objects, as well as beautifully complex objects. From bacteria to insects, from algae to cacti, from gems to mountains, from mice to whales, all vary in complexity and beauty. We humans try to understand and replicate in our computers the objects that surrounds us in nature. In particular, the Computer Graphics field deals with the rendering and animation of objects' models, as well as the creation of models.

Computer Graphics models and their creation techniques are affected by the type of modeling and the type of representations. Models can be of the solid or surface types, and representations can be of the explicit or implicit types. Solid models can tell if a point in space is either inside or outside the model, while surface models can not answer this. As for the types of representations, an explicit representation lists all the primitives that constitute the model, while an implicit representation provides a way to compute the model constituents.

Models are in great demand in Computer Graphics independently of their type and representation. Everything from simple geometric shapes to complex models are needed for different purposes in Computer Graphics. Since it is impossible to know all possible models that would ever be needed, the Computer Graphics field requires tools and techniques for the creation of models. The current thesis contains my contribution to the area of Modeling in Computer Graphics. My contribution consists of automatic model creation with an explicit surface representation.

Manual model creation is costly because of the repetition of similar tasks and the non-reusability of resources. Computer Assisted Design (CAD) tools support the processes to create models, and CAD tools can, arguably, be used to create any possible model. For example, dinosaur models in movies have been created with CAD tools, the models usually require hundreds of hours of human effort. Some people might add that any CAD tools' deficiency can be overcome with three-dimensional scanners, but consider the time and tasks needed to produce or obtain a model or a real-life object (to be scanned). For example, a sculptor would invest a lot of hours in chiseling a marble statue (Michelangelo's David took 5 years for completion, from 1500 to 1504). CAD/scanner models are expensive to create despite available tools. The cost can be reduced by repeating similar steps. For example, steps involved in creating a dinosaur's tooth would certainly repeat between teeth due to the model's similarity.

My research lies in the Automatic Model Creation area. This area covers the techniques to automatically create models. Automatic Model Creation requires mainly computational power. This area receives many contributions every year, which reflects the enormous resources invested in Computer Graphics [50]. The contributions vary in style, but their approaches are either synthesis or composition-based. The synthesis process consists of directly generating the components of the model's representation. The composition-based approach assembles existing models into a new one. Chapter 2 contains a more detailed review of the different approaches for creating models and their appearances.

Deformations are a good way to specify a process for creating models. A myriad of real-world detailed objects can be seen as the results of a deformation. For example, a plant is the result of a seed deforming into a stem, which later deforms by growing branches, whose ends deform into leaves. Another example is fruit. Pieces of fruit can be seen as branches' buds deforming into a shaped fruit such as an apple, banana, or strawberry. A last example would be thorns. These are

the result of a local deformation on the surface where they grow.

Despite great advancements in procedural techniques, Automatic Model Creation is still an area under construction. I venture to say that the Automatic Model Creation area will have a never-ending flow of contributions; my main argument is that there is no possibility to create a universal Automatic Model Creation procedure because it is impossible to foresee the human imagination, and since the Automatic Model Creation consists of automating repetitive tasks for creating one particular type of models, each procedure to create a model is naturally limited (if not impeded completely) to create different models. Therefore, the automatic creation of particular types of models requires particular procedures and since there is an infinite number of types of models then there is an infinite way to automatically create them. We just have to design these procedures and improve them.

Displacements are a good way to describe deformations. A deformation is a surface displacement; when the displacement is measured over time then the notion of velocity is natural (a velocity defines a change of position over time). Therefore, real-world processes are mimicked by using velocities to express deformations. Velocities are a good way to describe deformations because we are habituated to see the displacements from some natural phenomenon occurring over time. For example, most people see the growth of an apple as a surface expanding at a certain velocity rather than a cellular reproduction.

Models can be created by deforming an initial surface. Velocities are a good description of deformations over time; therefore, models can be created by assigning to a surface velocities that deform it. Such process involves the factors of Surface Representation and Deformation Specification.

The surface representation describes how the model is represented internally in the computer. Examples of surface representations are: set of points in space with an interpolating rule, scalar fields distributed on a 3D grid, and sets of polygons (polygonal meshes) whose vertices are defined in 3D space. The polygonal mesh is the most common surface representation used in Computer Graphics. Other examples of surface representation used in Computer Graphics are described in Chapter 2. In Chapter 3, I present a novel surface representation used to automatically create models.

The Deformation Specification establishes how the velocities are linked to the model. The Deformation Specification is linked to the surface representation because deformations have to be specified in terms of the model itself. For example, a polygonal mesh can be deformed if the velocities are specified as displacements of its vertices over time.

An explicit surface representation such as a polygonal mesh can make the Automatic Model Creation process easier. Explicit surface representations are always used during rendering (z-buffer technique), while other surface representations are converted to an explicit representation before rendering. Therefore, an explicit surface representation simplifies the Automatic Model Creation

because there is no need for a conversion between the rendered shape and the surface representation of the model.

I used an explicit surface representation in my research. I chose an explicit surface representation because the vast majority of graphics hardware (PC level) implement the z-buffer instead of the ray-tracing rendering algorithm. The z-buffer algorithm is best suited for explicit surface representations than the ray-tracing algorithm that serves better for implicit surfaces. On the other hand I chose a surface representation rather than the solid representation because the capability of surface representation to handle open surfaces, patches. Also I chose surfaces because of the model generation I followed (velocity-based deformations). Deformations do not require querying for a point in space being inside or outside a model. Therefore a surface representation avoided the overhead of handling solid representations. Furthermore the solid representation already have a big presence in Computer Graphics.

The model creation process using an explicit surface representation must control the amount of components used by the surface representation and ensure that the surface does not self-intersect. These factors are known as the spatial density of components and the surface self-intersection respectively. Spatial density of components refers to the number of components per space unit, e.g., the number of vertices of a polygonal mesh. Surface self-intersection indicates that a region of the surface crosses another, e.g., two intersecting triangles of a polygonal mesh. The density factor is important for Automatic Model Creation because it sets the lowest scale at which deformations should be specified. Deformations can be specified at any scale, but the surface's density governs which deformations are actually reflected on the surface; deformations specified at a smaller scale than the surface's density would not be noticed simply because there are no surface elements to reflect the changes. Additionally, the self-intersection factor is important to Automatic Model Creation because no self-intersecting surfaces exist in the real world. A geometric surface's spatial density and self-intersecting condition need to be considered for Automatic Model Creation because they set limits on what deformations can be used while still obtaining a valid surface. In this text, changes in spatial density and self-intersections are referred to as irregularities.

One of my goals is to present a surface such that deformations can be applied without concern for the surface state: in particular, without worrying about self-intersections or changes on the surface density. The surface density irregularity indicates that the separation between surface elements varies excessively, which is a problem because the separations make it difficult to deform the surface. The self-intersection irregularity defines an impossible surface for a real 3D object. These two irregularities are counterproductive toward my Automatic Model Creation goals because they have to stop the deformation process to provide maintenance to the surface or manually monitor the deformation to prevent issues. For the automatic creation of models I implemented an automatic resampling mechanism to provide a quasi-regular surface density. Additionally, I

implemented a surface evolution control to prevent self-intersections.

Models' deformations have a scope of effect depending on the extent of the deformation. Three levels exist: surface, local, and global. Surface level indicates direct control over the surface basic elements; that is, control is achieved by directly assigning velocities to each surface element. For example, a modeler may want to adjust the location of some vertices in a polygonal mesh to achieve a specific effect on the model's surface. The local level indicates direct control over a subset of the overall surface; i.e., the control is achieved by specifying the deformation in terms of the surface subset and not the individual points constituting the subsets. Note that direct control at the local level implies indirect control at the surface level; that is, deformations are not specified in terms of individual points, even though each point receives a velocity. For example, the creation of individual bumps on a flat surface needs only the information of the bumps' centers, but the geometric properties of the surfaces around the centers are also changed. Finally, the global level indicates direct control of the whole surface. Scaling a whole mesh is an example of direct control at global level. Automatic Model Creation avoids direct control at surface level due to its automation goal. However, it is desirable that Automatic Model Creation supports the specification of deformations at local and global levels.

The Automatic Model Creation process is enhanced by the capability to manage repetitive deformations and to parameterize them. A parameterized deformation is a deformation whose behavior is controlled by a set of values. The behavior of a deformation is defined by the velocities that implement the deformation, and these velocities are computed based on the parameters. Many models can be seen as the result of repetitive deformations. For example, reptile skin can be seen as a surface with several bumps and scales distributed on it.

If deformations can be parameterized then they support the creation of diverse yet similar detail. For example, the reptile's scales can be seen as similar bumps that differ slightly on their radiuses, heights, and colors. An Automatic Model Creation system should allow a user to parameterize repetitive deformations and manage their application, allowing the creation of similar models with similar detail. For example, the parameterization of a local deformation to create a thorn-like detail would allow the creation of different thorn types. Furthermore, the management of such deformation would enable the application of different thorns to different models, thus creating complex models such as thorned apples or thorned bananas.

Deformation management for Automatic Model Creation requires the ability to apply deformations and also to stop them. While simple shapes can easily be generated for few deformations, complex shapes (with several convex and concave regions) require a highly complex initial deformation. An easier way to obtain complex shapes is to compose several simple deformations and apply them at different stages of the model creation. Furthermore, stopping the deformations also facilitates the model creation since a designer does not have to worry about prolonged deformations.

Note that this differs from other surface deformation techniques such as Level Set Methods [83] in which deformations are set in a "let-go" approach; that is, the deformations are set in an initial state and the system is allowed to evolve autonomously without further user interaction.

Level Set Methods have been traditionally used to simulate the evolution of surfaces. The applications usually consist of a set of initial conditions and the result is the evolution of the surface and possibly its final state. That is, the Level Set Methods are usually used in an initial value problem simulation. In my case, Volipoc is used to support the Automatic Model Creation process; that is, the design of the surface representation avoids having users/modelers deal with surface correction problems (density) and allows the specification of deformations in an intuitive manner (velocities). In other words, Level Set Methods are difficult to stop and control occurring deformations. I have tried to provide such control within my framework.

In Vebam, model creation can be described in terms of states and transitions. States and transitions are used in combination to start, stop, and modify deformations at different stages of model creation. For example, a thorn-like surface detail can be obtained by following the next sequence of states and transitions: state "No deformation", transition "Thorn-generating deformation starts", state "On-going deformation", transition "5 seconds elapsed", state "No deformation". The previous sequence depicts in terms of states and transitions the process to create a thorn-like detail on a surface.

Petri nets are mechanism to describe state/transition systems [78]; and these can be used to manage deformations. A Petri net is a bipartite ("State" and "Transition" nodes) oriented graph with tokens associated with "State" nodes. The marking reflects occurrences of states, and the tokens are moved around the net by the transition nodes. By associating deformations with "State" nodes one can achieve Automatic Model Creation through deformation. Chapter 5 contains a thorough description of my use of Petri Nets for Automatic Model Creation, and Chapter 7 shows the results.

1.3 Solution Outline

A useful Automatic Model Creation scheme would include: velocity-based surface deformations, an explicit surface representation (that does not create irregularities during resampling), local and global deformation specification, and management of deformations application. Nevertheless, two questions remain. First, how do I create such a scheme? Second, how do I demonstrate its utility?

The rest of this document describes an Automatic Model Creation scheme containing the aforementioned features. The scheme's main goal is to enable the creation of richly detailed 3D models through deformations. The scheme encompasses specifications for a novel surface representation, velocity-based surface deformations, surface resampling, surface evolution, deformation application,

deformation management, time evolution, surface rendering, and surface conversion. Additionally, I implement the scheme concepts in a software tool, Vebam (Velocity-Based automatic model generator), and I used Vebam to demonstrate the utility of the scheme by automatically generating models.

The scheme's features are described next, detailed explanation of these are provided throughout the rest of this document:

- Surface Representation - I have defined a novel surface representation for the scheme. The representation consists of oriented points in space and volatile edges. The oriented points are surface samples with information associated with them: a position, a velocity, an orientation, and a color. The rendering of the surface uses the color and orientation information, while the surface deformation uses the velocity information. The volatile edges are used to explicitly define the shape of the surface (joining points), and they are volatile to provide flexibility for deformations and control the point cloud density. (Note, I use the term volatile because edges can rapidly appear and disappear during the resampling of the surface, as described in Chapter 4).

- Velocity-based Surface Deformation - I use velocities to specify deformations on Volipoc. The scheme computes points displacements based on their velocity and a time step.

- Time Step Generation - Time steps are generated by a clock. The clock is also used to synchronize the states of the model generation manager. Also, the clock is used to interrupt deformations and allow user feedback on the deformation process.

- Resampling - I provide a resampling technique for Volipoc. The resampling consists of tracking the lengths of volatile edges, and eliminating them depending if they are too short or too long. A point spawn or points fusion accompanies the edge elimination. Freeing the user from correcting any surface irregularities and enables the continuous application of deformations. A grid partitioning of space sped the resampling process, the grid reduces the number of comparisons made between points.

- Hole prevention - I control the surface evolution in a way that no holes nor blends appear. The control consists of preventing points to get too close to each others and fuse surface regions. The grid used in the resampling is also used to control the fusing of points. Holes are prevented by the resampling mechanism.

- Deformation Specification - I use a set of operators to associate velocities with points. The operators can assign velocities at local and global levels. Once points have velocities associated with them, then the surface evolves in time by computing the points displacements according to their velocities and by resampling the surface.

- Deformation Management - For truly automatic model creation, I provide a mechanism to automatically apply and stop deformations. This mechanism consists of a state engine that controls the start, duration, and stop of operators.

- Surface Rendering - I use two different rendering techniques for Volipoc: Surfel+Wireframe and sphere-based rendering. Surfel+Wireframe rendering uses surfels for points [68] and simple wireframe for the volatile edges. I consider this rendering style to reflect the important visual information of the surface representation, that is, the positions of points, their orientation, and the shape of the surface defined by the volatile edges. The second rendering approach, sphere-based, consists of using spheres instead of surfels as the rendering primitive. This approach facilitates the appreciation of the surface shape.

- Surface Conversion - I provide an algorithm that translates from Volipoc to a polygonal mesh. The algorithm uses the existing edges and points to create a polygonal mesh. That is, it selects the existing triangles that form a tessellated mesh of triangles. Note that this approach differs from surface reconstruction algorithms that make an approximation of the surface; my approach only uses existing information of the surface representation and does not create (interpolate) any new information.

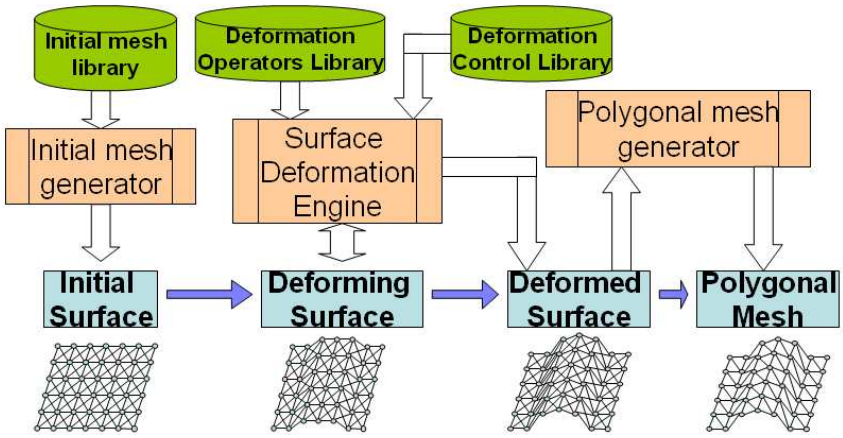

Figure 1.2: Overall scheme flow and model creation stages (blue blocks from left to right). The stages consists of: create initial surface, interactively and/or automatically deform the surface, and create the polygonal mesh).

Figure 1.1 shows an example of the creation of a model with Vebam. The first step is setting the initial surface (Figure 1.1.a); in this case, a sphere. The upper half of the sphere moves upward to form the model in Figure 1.1.b: a cylinder with round end caps. Figure 1.1.c shows a cap-mushroom-like structure formed by deforming the upper edge of the cylinder. Figure 1.2.d shows an intricate surface detail generated with a deformation operator (these are explained in Chapter

5). Figure 1.1.e shows thorn-like surface details that are deformation at a higher scale than those of Figure 1.1.d, but generated with the same approach (deformation operators). Figure 1.1.f shows a bulbous root created by deforming the lower edge of the initial cylinder. Finally, Figure 1.1.g shows the final model automatically generated with Vebam; the model is rendered using small spheres as rendering primitives to better appreciate the shape. It is with this example and the results presented in Chapter 7 that I demonstrate the utility of my Automatic Model Creation deformation framework and its implementation in Vebam.

The overall operation of my scheme is shown in Figure 1.2. For the creation of a model, a user starts by specifying an initial surface; the surface can be a previously created model or a predefined initial mesh. The next step is to specify the surface deformation. The specification can be done by using a predefined sequence of deformation operators. The second way is better suited for prototyping deformation effects.

I have implemented the aforementioned features and scheme components in the "Vebam" software. Vebam includes a rendering engine which contains a combination of known rendering techniques, chosen to display the surface's relevant information: connectivity, shape, and orientation. I implemented the rendering engine and Vebam's modules in C++ using the Direct3D v.9.0a API in the Win32 platform.

1.4 Contributions Summary

The main contribution of my research is a scheme for the automatic creation of models through composition of deformations. My scheme specifies how to obtain richly detailed models by deforming an initial surface; the deformations are specified in terms of velocities. The scheme consists of a series of specifications that all together contribute to the goal of automatic model creation. Each specification solves a particular problem of Automatic Model Creation, and each is the result of careful testing and design. The specifications define first an explicit surface representation for the models, second a resampling policy for the surface, third local and global deformation specifications, and fifth deformation management. Some of the specifications are contributions to Computer Graphics by themselves, such as the designs for the surface representation and its resampling.

The designs of the surface representation and resampling behaviors are the result of a common goal: support the Automatic Model Creation with inexpensive computation of surface deformations. Both designs are tightly linked to each other and depend on each other to fulfill their goals; yet, I present them as independent contributions for simplicity. Also, similar contributions are reported separately in Computer Graphics; for example, different resampling techniques have been reported for the same surface representation. Therefore, I avoid confusion by labeling the surface representation and resampling designs as "minor contributions."

The first minor contribution is a specification of a novel surface representation. Its design solves the problems of using velocity-based deformations and using an explicit surface representation. Volipoc is a hybrid between point-based surfaces and polygonal meshes; its design exploits the pure-point surfaces' advantage of not having restrictive linkages between surface elements and the polygonal meshes' advantage of having a continuous surface. Velocities are directly assigned to the surface's points in order to deform the surface. A surface representation like mine has not been reported in Computer Graphics.

My resampling specification keeps a constant surface density and avoids surface self-intersections. The surface density is controlled by computing new points of the surface when it stretches or compresses. A new point is created when two points sharing an edge move apart; the points that move apart are considered as the "spawning" points. The new point's position, velocity, and color are computed using the same properties of the "spawning" points. The normal is computed so that the surface presents a smooth normal transition (see Chapter 4).

The space-partitioning structure prevents the surface from blending with other sections of the surface. The blending is prevented by stopping a surface's points from trying to move into an area where the resampling would create edges that change the surface topology. Detailed discussion of this process is presented in Chapter 4.

The control of surface displacements and hole appearances provides topology preservation as a byproduct. Reported techniques to preserve topology involve the computation of intersecting segments of the surface; the new position of the surface is rejected if an intersection occurs. My approach is similar in that a point's new position is computed and if it is in a forbidden region then the movement is prevented. In this way, surface intersections are avoided by controlling the points displacements. Details of this process are presented in Chapter 5.

My resampling mechanism differs from the resampling of polygonal meshes in that I am not restricted to obtaining a tessellated mesh. On the other hand, the resampling mechanism of pure point surfaces does not use inter-point connectivity to resample the surface, as mine does. My resampling mechanism uses inter-point information and creates a non-tessellated mesh; therefore, I provided a novel way for density control of an explicit surface representation. Different resampling techniques have been reported in Computer Graphics but none for my novel surface representation.

In addition to my novel surface representation and resampling technique, my Automatic Model Creation scheme includes an original solution to the problem of specifying local and global deformations. The specification uses two types of operators: Selection Operators and Velocity Operators. Selection Operators specify the selection of surface elements; Selection Operators use any geometric property of the surface and its elements. Velocity Operators specify how velocities are computed for a given set of surface elements; velocities can be computed in any way including the use of geometric properties of the surface and its elements.

My Automatic Model Creation scheme solves the problem of managing repetitive deformations. I used pseudo-Petri nets to control the operators being applied to the surface. Petri nets model systems with states and transitions; I defined modifications to the Petri nets (hence the pseudo-Petri nets labeling) so they can be used to control deformations of surfaces. My modifications include two new ways to fire transitions (changes of state) and the association of the operators with Petri nets' states; these extensions to Petri nets, and the application of Petri nets to control the deformation of surfaces, have not been reported previously in Computer Graphics.

Furthermore, my contribution to Automatic Model Creation provides an innovative conversion algorithm, transforming from Volipoc to the more common representation of polygonal meshes. This algorithm has some conceptual similarities to the ball-pivoting surface reconstruction algorithm, but differs in the surface representation (mine includes linkage between points).

Summarizing, my main contribution to Computer Graphics is the whole Automatic Model Creation scheme. My contribution fulfills the goal of automatically creating rich and detailed models through velocity-based deformations. My main contribution encompasses two minor contributions, a couple of original solutions, and an innovation. The two minor contributions are a novel surface representation and its resampling technique. The original solutions answer the problems of how to specify local and global deformations and how to manage deformations. The innovation consists of a way to translate from Volipoc to a polygonal mesh. Finally, I demonstrate the utility of my scheme and its components by their implementation on the Vebam software and its application to automatically create visually rich models.

The rest of this document is organized as follows. Chapter 2 reviews key results related to content generation in Computer Graphics. Chapter 3 explains the details of my proposed surface representation. Chapter 4 explains the behaviors associated with Volipoc; in particular, the resampling and control of the surface evolution are explained. Chapter 5 explains how deformations are specified and controlled for my particular surface representation. Chapter 6 describes additional problems solved throughout my research that complement my initial contribution. Chapter 7 shows different models obtained within my Automatic Model Creation framework; steps and parameters used are specified also. I conclude with Chapter 8 and give some suggestions for future work. Appendix A explains how the outermost shape of Volipoc is computed. Appendix B contains definitions for the terminology used within this document. Finally, Appendix C gives a brief introduction to Vebam.

Chapter 2

Content Generation in Computer Graphics

The capability of computers to process huge amounts of data allows them to deal with the complex processes of rendering Computer Graphics scenes. The same capabilities are used to create content for Computer Graphics. This chapter reviews the state of the art in content generation. I grouped the reviewed research by shared features, with the advantages and disadvantages of similar techniques described in the review.

For reviewing the model generation area of Computer Graphics, I classified the contribution into two types: Interactive and Procedural. An *Interactive* contribution is when a user creates content; the contribution focuses mainly on explaining the different tools and approaches to support the user in the creation process. A *Procedural* contribution is when a set of parameterizable instructions creates content; the contribution focuses on explaining the procedure that automatically generates the content.

An automatic process is a relative concept. Some user interaction is always present whether a contribution is interactive or procedural. The interaction level depends on the problems solved in the contribution. In procedural contributions the interaction can be as simple as specifying parameters and as complex as building a library of sample content. However, if the decisions during the content creation process are made mainly by a user then I say that such contribution is of the manual type. Similarly as the process requires less and less user input (decision making) then the process becomes more automatic. Unfortunately, diversity of content types decreases when more decisions are taken automatically, even though the time to create content of the same type is decreased. Figure 2.1 sketches this concept; the vertical axis describes how much user input is required and increases from bottom to top, while the horizontal axis describes the automaticity level, and increases from left to right. For example, a CAD tool can be used to create virtually any imaginable content; however, a software for automatic creation of buildings may not be the best option to create tiger-like 3D models, but it can easily create several buildings with different features in a batch process.

Contributions containing both interactive and procedural components are referred to as *semi-automatic* contributions. The difference between procedural and semiautomatic contributions is that in a semiautomatic contribution a user has more control over the decision-making steps. A

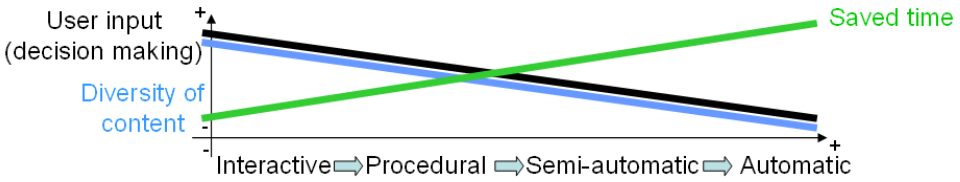

Figure 2.1: Content generation approaches and user decision level.

procedural contribution does not allow a user to make any decisions at all when the content is being synthesized. I use the term *content* indiscriminately to identify both the geometries and textures of Computer Graphics models (see Figure 2.2). Since the goal of this project is Automatic Model Creation, I am interested in the procedural content generation, but for completeness I am reviewing the interactive approach also.

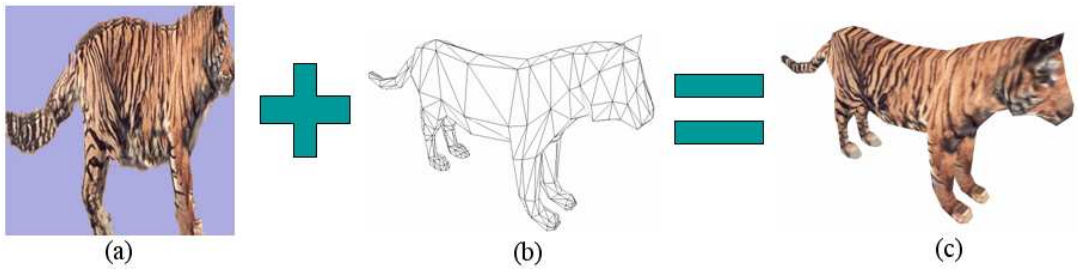

Figure 2.2: Examples of content in Computer Graphics: (a) A texture represented with a bitmap, and (b) A geometry represented with a polygonal mesh. Both types of content are used to create richly visual models (c).

2.1 Types and representations of models

In the real world objects are defined by the particles that constitute the objects. As examples we have: beaches are formed by grains of sand, smoke is formed by floating ashes, flesh is formed by cells, and rivers are formed by water molecules. Each particle could be modeled in a computer in order to obtain a computer model of the real object. However, Computer Graphics models do not need all that information. The Computer Graphics models require the information used in synthesizing and manipulating visual content.

Computer Graphics models can be of two types: surface and solid. The surface type describes the edge of the real-world object being modeled. On the other hand, solid models (sometimes called volumetric) define the space occupied by the model. This definition is usually specified by being able to classify whether a point in space is inside or outside of the model. Examples of surface models are: polygonal meshes [24, 86], simplex meshes [17], point-based surfaces [30, 43, 65, 81, 109, 91],

and normal meshes [32]. Examples of solid models are: isosurfaces [103], level sets [83], blobs [102], distance-based volumes [47, 27, 14], bounded volumes [1, 14], and metaballs [7].

The differences between surface and solid models are not only about the information they reflect (boundary vs. volume), but they also differ in the benefits each type of model provide. In the following, we compare some of the advantages and disadvantages of solid and surface model representations.

Surface models can be rendered with the z-buffer technique without making any processing to the surface information. On the other hand rendering solid models requires first computing their boundary. Boundary computation also affects how surface and solid models behave in detecting collisions. Collision detection is more complicated with surface models than with solid models. Surface models requires additional processing to identify if a point, ray, or vector is inside or the model. This is not the case with solid models; testing whether a point, ray, or vector is inside them is easily solved by the model definition of a volume.

Other advantages and disadvantages related to surface and solid modeling exist. However, some of this may be more goal-oriented. For example, solid modeling exploits the reusability of existing content with CSG (Constructive Solid Geometry), which can not be applied directly to surface models. On the other hand, surface models can easily be deformed to form intricate surface details, which is not easy with solid models. This last was one reason why Volipoc is a surface representation; to simplify the creation intricate surface details by deforming it.

Although deformation is not in widespread use, ease of deformation is a consideration for me because my modeling operations are deformations. Surface models are very simple to deform because the deformation is only expressed and only deals with surface elements. On the other hand solid models require deformations expressed in terms of volumes, which happens to be more difficult to handle than surface deformations because of the higher dimension (a volume is of higher dimension than a surface).

Models are generally represented in two ways: explicitly (sometimes called parametric) and implicitly. Implicit surfaces are defined by a function (implicit function) that establishes an algebraic relation between the function's variables instead of defining variables in terms of the other variables (explicit function). For example, $R(x, y) = 0$ is an implicit function and $y = f(x)$ is an explicit function. An implicit surface is the set of points that satisfies the implicit function. An explicit, parametric, surface is the set of points defined by the independent and dependent variable of the explicit function that defines the surface.

The notion of an implicit surface was first introduced in Computer Graphics by Ricci [80]. He created an interactive program to manipulate mathematical functions that represent volumes. The function's parameters could be changed interactively, thus showing the manipulation of the implicit surfaces. Additionally, Ricci provided a set of boolean operators for the volumes, thus providing

the basics for Constructive Solid Geometry (CSG). CSG is a technique to create complex models by using boolean operators that combine existing (often simple) models.

The notion of implicit surfaces was also initially used by Blinn [7], Wyvill et al. [103], and Bloomenthal and Wyvill [10]. Blinn defined a surface as the summation of individual Gaussian distributions. Blynn modeled electron density maps (molecules) but the concept was easily retaken in other Computer Graphics endeavors such as noise generation [104]. Similar to Blinn, Wyvill defined an implicit surface by summing functions and provided an algorithm to transform from his implicit surface to an explicit polygonal mesh. Finally, Bloomenthal and Wyvill associated implicit functions with skeletal elements to facilitate the interactive manipulation of models. Other implicit representations used in Computer Graphics are: isosurfaces [103], distance-based volumes [47, 27, 14], and metaballs [7].

As for the notion of creating models, the field of Computer Graphics has seen the creation of different tools since the early stages of the field. These tools have covered the manipulation of models independently of their representation. For example, interactive tools to manipulate polygonal meshes and NURBS [69, 24, 8, 92] surfaces were reported. I consider that due to the the difficulty for most designers to understand surfaces in terms of functions and comprehend the effects of manipulating function parameters, the most common graphics hardware is designed to render explicit surfaces (z-Buffer technique).

Despite of not having the same presence in the consumer world as explicit surfaces, implicit surfaces continued evolving. Different techniques to manipulate implicit surfaces were reported. Museth et al. [56] provided a set of editing operators for Level Set-based models. Wyvill et al. [102] designed BlobTree, which extends the notion of CSG by adding other blending and warping operators to the boolean operators. Finally, Pasko et al. [64] created F-rep which is a system that uses R-functions (function that changes sign if and only if an argument changes sign) to define models and operations for manipulating them.

An implicit surface commonly used in Computer Graphics is the iso-surface, often referred to as Level Sets (Osher and Sethian [62]). Level sets are the combination of isosurfaces and the techniques to manipulate the isosurface. An isosurface is the set of points that evaluates an implicit function equal to a constant, e.g., $\{x, y, z | f(x, y, z) = c\}$. The isosurface, implicit surface, is often computed from a scalar field, which in turn is specified in terms of differential equations. If the differential equations are affected then the shape of the isosurface changes.

Level sets have been used for interactive content generation by Museth et al. [56] and Lawrence and Funkhouser [42]. Museth provided a set of editing operators for level sets. The operators included Constructive Solid Geometry, Morphological (dilation and erosion), blending, smoothing and sharpening, and displacement maps. Lawrence and Funkhouser modified the gradient field to change the shape of the Level Set. The previous examples show that Level Sets have been used for

creating models, although they have been used mainly for simulations.

Volipoc is an explicit surface representation; therefore, the rest of the literature review focuses mostly on explicit representations of surfaces. However, implicit representations are referenced as seem convenient to denote a concept in model creation. The reader is directed to Bloomenthal [9] for a compendium of implicit surfaces and their applications.

The types and representation of models used in Computer Graphics affect the model creation process and its capabilities, sometimes providing advantages and sometimes disadvantages. Independently of type and representation the model creation process can be automatized. The automation process varies in the level at which a user need to intervene. The extremes of the automation process are: interactive and automatic. These are used to review the generation processes in the next two sections.

2.2 Interactive Content Generation

The following sections describe what I consider are the main contributions for interactive content generation. I have classified the contributions as direct (Section 2.2.1) and composition-based content creation (Section 2.2.2).

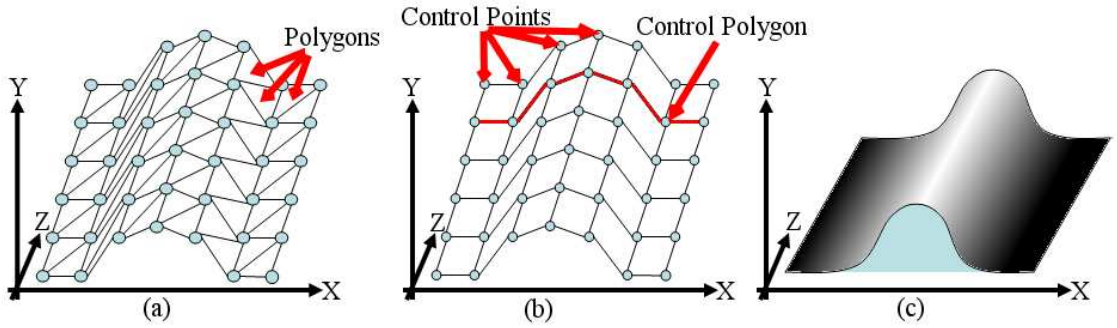

Figure 2.3: (a) Polygonal mesh surface representation; triangles are the constituting polygons. (b) NURB surface representation; control points form control polygons that are used to define the surface (the real surface is computed as an interpolation of the control polygons)). (c) Surface approximation of the surfaces represented in (a) and (b).

Interactive content generation contributions provide great CAD tools and techniques. These focus on supporting and facilitating the creativity of artists to create content. Sometimes, they even increase the artists productivity by eliminating redundant operations. However, they cannot substitute for artists when deciding how to create the content.

Artists have seen the potential of reusing existing content. This allows us to classify the interactive content generation in two types: direct content generation and composition-based content

generation. Direct content generation means that an artist manipulates content primitives to create new content without using previously existing content. On the other hand Composition-based content generation means that the user combines pre-existing content.

2.2.1 Interactive Direct Content Generation

Since the creation of Bézier curves [3, 4] in the 1970s and the later introduction of NURBS [25] in the 1980s, the concept of interactively altering a model's surface to create a new model has been present in Computer Graphics. The most common surface representation in Computer Graphics today is the polygonal mesh. Research related to mesh deformation addressed topics about mesh quality. In particular, the problems of subdivision [12, 51, 52, 54, 76, 89], smoothing [13, 19, 33], decimation [82, 85], and mesh congruency [31, 49, 89] were addressed.

The problems of subdivision, smoothing, decimation, and mesh congruency each address a particular need when deforming the meshes. Mesh subdivision aims to add detail to a polygonal mesh. The detail is added directly to the mesh by breaking its polygons into smaller polygons. On the other hand, decimation reduces the number of surface elements required to represent a surface detail. The basic idea is to represent the same surface features with less primitives in a given region. Smoothing aims to reduce noise that can appear as high-frequency variations. Finally, the mesh congruency problem appears when the mesh self-intersects; that is, two or more polygons intersect. This is solved by modifying the mesh's vertices and edges until the self-intersection disappears.

Many of the concepts introduced by the aforementioned contributions are found in commercial CAD tools; many applications have built-in tools for decimation, smoothing, and detecting self-intersecting meshes. The active research related to mesh deformation has focussed on decimation and smoothing. Yim et al. [105] aligned adjacent triangles' normals, and Ohtake [60] used the dual of a mesh to create new vertices and refine the original mesh. Recently, decimation approaches have been reported by some people: Garland and Heckbert [29], Hoppe [36], Shaffer and Garland [84], and Williams et al. [99]. Garland and Heckbert manipulated pairs of vertices based on a distance function without requiring them to be linked by an edge as in the traditional approaches, which only considered vertices that were linked by an edge. Hoppe used edge collapsing for decimation. Shaffer and Garland used additional data structures (quantization grid and BSP-Tree) for mesh simplification. Williams et al. used a measure of visual appeal rather than mathematical properties to simplify a model's mesh.

The earliest interactive contributions focused on polygonal meshes, but polygonal meshes are not ideal for all situations. Triangles with large edges may appear during editing; these are not desirable because they difficult the continuous deformation of the mesh. Also self-intersecting surfaces can be created during mesh editing. These problems have been addressed by researchers, but these solutions still have some drawbacks. To deal with self-intersecting surfaces Snel et al. [89]

used edge collapsing and splitting, but the mesh could be modified so that the model shape would be completely different. Lobregt and Viergever [49] presented an expensive solution using triangle normals (each normal required to be computed during interaction with the user). And Gueziec [31] used the Euclidean distance between adjacent triangles with the same problem of having a costly solution as Lobregt and Viergever. The thin triangles problem was addressed by Brakke [11], who used equiangulation to replace pairs of triangles with a pair whose angles were more similar, but sometimes the process made no change on the mesh. For example, if the adjacent triangles were of the same proportions then the equiangulation would result in a parallelogram with the shared edge flipped to the other diagonal.

Interactive direct content generation contributions date from the early stages of Computer Graphics when the basic surface representations (NURBS and polygonal meshes) were introduced. Usually, a brief period of publications starts when a new surface representation is reported; some of the publications address the interactive content generation problem. In addition to NURBS and polygonal meshes, other surface representations have been reported in Computer Graphics for content generation.

The contributions that generate content by directly creating the elements of the surface address diverse problems; for the case of NURBS and polygonal meshes, the problems addressed are subdivision, smoothing, decimation, and self-intersection. Later, new contributions addressing more intricate problems were reported. For the scope of this document, I am interested in the problems related to content generation. Other contributions where content is created directly have been reported and use surface representations other than NURBS and polygonal meshes.

A point-based surface representation is an alternative to polygonal meshes; the basic idea is to use points without connection between them to represent a surface, i.e., vertices without edges. Point-based surfaces were introduced by Levoy and Whitted [43] in 1985. Since then, other researchers have provided similar ideas. Pfister et al. [68] used surfels (sizable and oriented points in space) as rendering primitives, and Rusinkiewicz and Levoy [81] used a point sampled surface with different rendering primitives. Both contributions used a hierarchical structure to implement frustrum culling and resolution control. They improved the rendering performance by reducing the number of point rendered and demonstrated that resolution could be controlled without modifying the underlying surface.

From the interactive content generation contributions point of view, there have been different contributions for point-based surfaces. Zwicker et al. [109] introduced Pointshop3D, a tool used to edit point-sampled models. Pauly et al. [65] expanded Pointshop3D with several algorithms for deforming point-based models. The main advantage of point-based surfaces is that large edges are not a problem (there are no triangles defining the surface), nor is surface self intersection. On the other hand, the main problem of point-based surfaces is that the spatial density of the points varies

when the surface expands or shrinks, therefore requiring a user to create new point samples. The creation of these new points is not simple since the lack of inter-point linkage makes it difficult to guarantee that a point is created on the surface of the model.

Additional examples of point-based surfaces used in Computer Graphics are provided by Szelisky and Tonnesen [91] and by Keiser et al. [39]. Szelisky and Tonnesen provided an oriented particle surface representation, their resampling and force-based deformation produce uncontrollable surface displacements. Keiser et al. used oriented-points also, but their surface deformation includes a simulation of the model's internal volume restricting the deformations.

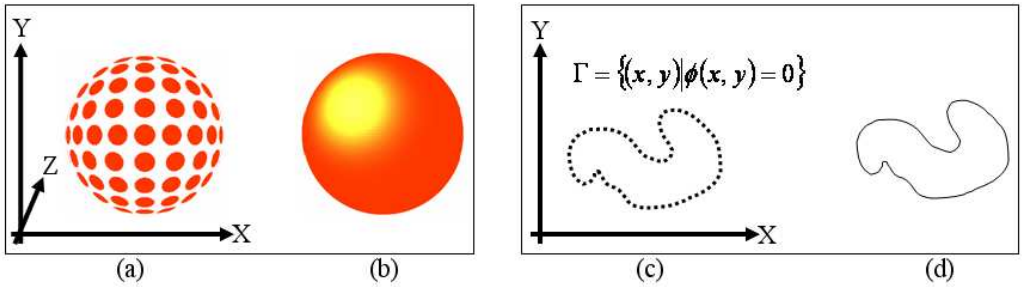

Figure 2.4: (a) A point-based representation of a sphere, and (b) Surface represented in (a). (c) Level Set representation of a curve, and (d) Shape represented in (c).

Rendering continuous surfaces from a cloud of points has been addressed in different ways in Computer Graphics. Rusinkiewicz and Levoy [81] used different primitives (square, circle, and diffused circles) per point. The primitives sizes were adjusted such that no holes on the surface would appear when rendering the point cloud. Zwicker et al. [109] used circles of different sizes. Pauly et al. [65] also used circles that deform into ellipses when the surface deforms also. Levoy and Whitted [43] used a one point per pixel approach. Grossman and Dally [30] used points also, but filled gaps by resampling the image. Pfister et al. [68] used an a tuple with shape and shade attributes for rendering but did not used them for surface deformations as is my case. Recently, Sun and Chang [40] proposed the use of octagonal splats.

Other surface representations are available in Computer Graphics, but are not as common as those described above. Examples of additional surface representations are Normal Meshes and Adaptively Sampled Distance Fields.

Normal Meshes were introduced by Guskov et al. [32], who suggest that locally smooth surfaces can be described as a single scalar height function over a tangent plane. The computed tangent planes surround a 3D model; the model is defined by the planes' coordinates and a height function. The height function describes the surface beneath the planes.

Frisken et al. [27] used Adaptively Sampled Distance Fields; they test whether a point (vertex)

is inside or outside the mesh structure. The test makes use of the implicit surface governing the mesh, that is, the test evaluates the value of the surface to determine if a point is inside or outside. Also, they use the test to embed surface details and edit vertices.

The aforementioned surface representations are original ways to describe surfaces, but for the Automatic Model Creation point of view, the use of such representations may cause the inconvenience of additional computations for transforming between the rendered surface and its implicit representation. Additionally, interactivity may be difficult because a user would need to think at implicit surface definition (mathematical functions).

Furthermore, manual creation of complex models would be a daunting task since each model would require the individual specification of it surface elements. To overcome such difficulties, researchers have provided ways to use existing content to compose new models.

Some interactive content generation contributions use abstraction layers to edit models. An abstraction layer is a simplified version of the model, for example, the skeletal representation of a model is an abstraction of it. Interactive contributions using model abstraction have three main steps: original-to-abstract conversion, abstract editing, and abstract-to-original conversion. An example of using a model's abstraction is provided by Nealen et al. [58]; they change a model's shape by modifying regions on its surface. The region selection uses a hand-drawn silhouette; a second silhouette is used to define the target region. Their system computes the transformation of the points in the source and target silhouettes and uses that transformation to modify the vertices of the original region.

Similarly, Llamas et al. [47] created a shape deforming tool. Their tool uses a pivot point defined by a user; the pivot point is a model's surface point. The pivot is used as a reference for all the user's transformation commands, which consists of stretching and compressing deformations. A metaphor to describe this process is to have a rubber-like model floating in space that a user stretches and compresses to change its shape. The original model is deformed by applying the recorded transformations.

Deforming a model's shape through its abstraction is mainly used for animation rather than content creation. Animation techniques are obtained when the model's abstractions are used to change the position of model segments. Zhou et al. [107] used a graph to represent the volume of a model. The graph is interactively edited to achieve a desired shape. The transformations applied to the modified graph are then translated to the original model. Ju et al. [38] deform a control structure and use it to modify the orientation of a model's limbs. The control structure is a simplification of the model's mesh; the control structure segments the model limbs. The control structure's sections are transformed (reoriented or scaled), then translated to the original mesh. The conversion is done by interpolating the transformation parameters through mean value coordinates. Lipman et al. [46] translated a polygonal mesh to a Discrete Form representation.

The representation is edited to obtain deformations on the original mesh. A First Discrete Form represents a mesh through a graph representing the relations between the mesh vertices, edges, faces, and their geometric values. They deform the graph by applying constraints to the equations they use to define the mesh. Finally, the deformation is applied to the original polygonal mesh by translating back from the Discrete Form's representation.

A model's abstraction simplifies the application of repetitive transformations. However, the conversion between original and abstract representations may produce loss of information; such loss jeopardizes the desired transformation effects.

The main advantage of interactively editing a surface representation is the accuracy provided to the user to do whatever he/she wishes to the model. On the other hand, the main disadvantage for interactively creating content is that manual processes may be time consuming, repetitive, and prone to error. Such problems can be dealt with by using preexisting content to compose new content, as described next.

2.2.2 Interactive Composition-Based Content Generation

The creation of a new model is not accomplished only by modifying existing models, but also by using segments of existing models to compose new models. This approach requires techniques to define segments and seamlessly assemble them. A seamless assembly has no obvious transition between its segments.

There are two main approaches to making a seamless composition of existing content components. The first is to modify the surface elements and match the boundaries where the components will join, and the second approach is to create a patch to join the components. Each of these approaches can be applied according to the surface representation used. For example, for polygonal meshes the first approach would require modifying the vertex positions and possibly creating new edges between the boundaries of the components, while the second approach would require creating a mesh that fits both components' boundaries.

Figure 2.5 illustrates the two composition approaches. Figure 2.5.a show two polygonal meshes to be composed into a new one; the problem of matching borders arises when composing diverse content. Two main approaches for solving it are: (a) create a patch compatible with the borders of the content (Figure 2.5.b), and (b) modify the border(s) so they can be joined (Figure 2.5.c).

Funkhouser et al. [28] provided an example of the the composition approach. They allow a user to cut and paste polygonal mesh segments. Segments of different models are attached to each other, following the modify-boundaries approach. The segment contours are attached by remeshing their closest vertices. Additionally, Funkhouser et al. define a descriptor for polygonal meshes; the descriptor is a measure of how far points from one surface need to be moved in order to match the shape of the other surface under comparison.

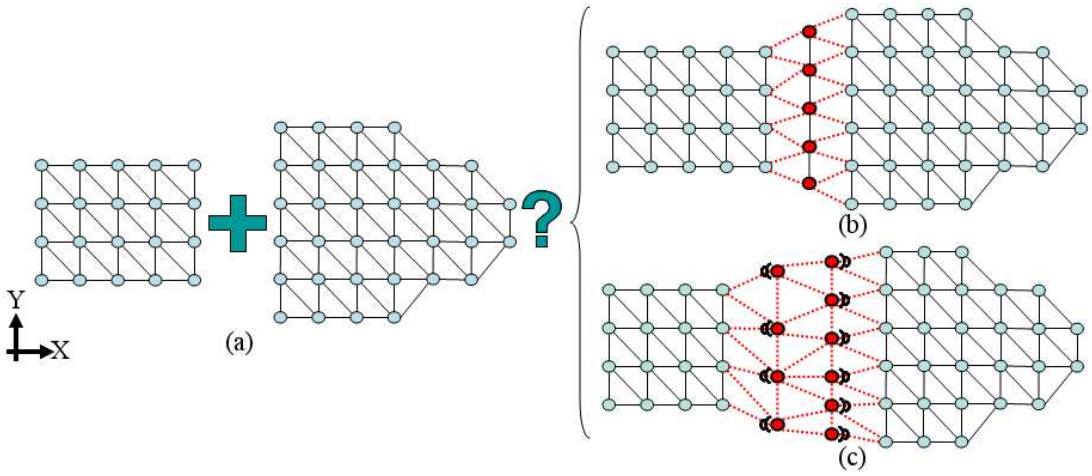

Figure 2.5: Composition problem (a) and its solution approaches: (b) Create patch between components, and (c) Modify component borders to fit each other.

An early attempt similar to Funkhouser et al. was made by Allan et al. [2]. They provided a system for deforming a polygonal mesh by distributing variations on the surface around a specific vertex. Also similar to Funkhouser, Yu et al. [106] define a way for segmenting a model, but with a different integration technique; Funkhouser modifies the vertices to fit boundaries while Yu et al. regenerate the meshes between boundaries, following the patch-boundaries approach. Yu et al. used a gradient field to manipulate (automatically, without user interaction) the vertex positions of model segments during attachment obtaining seamless attachments. Another approach to make seamless blends is presented by Adams et al. [1], who used Boolean operations to compose surfel-based models. They partition space and label voxels as inside or outside a model, and the Boolean operators use the labeling to decide which surfels include in the composed model. Their algorithm resamples ambiguous surfels in the space region where the operands intersect; the resampling consists of replacing surfels that intersect other surfels (creating the ambiguity) with smaller ones. A similar technique is reported by Biermann et al. [6], who provide a technique for cutting-and-pasting surface details. Biermann's technique allows a user to select a model's region to be copied. These selected regions are transformed into a surface segment without detail; the segment is used to define the area in the target surface where the detail will be pasted. The detail from the source region is matched to the target region by mapping both regions into a plane. Last, the planar representation of the target's region detail is applied in the final surface, thus, *pasting* the detail from the source region.

Depending on the application, the composition-based contribution may not require the user to make a seamless join between components. This is often the case of L-Systems, where the goal is to obtain a complex branching structure. L-Systems results are often rendered as intersecting

segments of components since the goal is the branching structure and not the seamless composition of its elements.

Some of the contributions have components of different approaches. Ijiri et al. [37] created a flower modeling interface. Their system allows a user to create or select existing flower components (composition approach) such as petals, leaves, pistils, and stamens. Additionally, the components' surface is modified to achieve the desired shape (surface deformation), but Ijiri et al. generate the plant structure with traditional L-Systems. In other words, Ijiri et al. produced their results using both the composition and content-modification approaches.

Composition-Based Content generation has several advantages. An advantage is that the time and effort spent in previously created content, as well as the embedded intellectual property, are reexploited. Another advantage is that the time to create new content is reduced when the pool of precreated objects has a considerable size. The composition approach requires good descriptions of surfaces so that they can be used to index elements for a pool of objects. A good indexing system must consider the shape, orientation, and surface characteristics of the segment the user is looking for; the indexing must also deal with scaling effects.

Interactive contributions have a significant presence in the Computer Graphics content creation area. These contributions are mainly used to demonstrate and introduce a new technique. Recent interactive contributions address new surface representations and are used to demonstrate cut-and-paste techniques. These contributions show the flexibility and capabilities users have for creating new content; the creation is done by either deforming or using components of existing content. Key aspects to consider in interactive contributions are the content source and modeling operators. There are two types of content source: self-made and preexisting. Self-made content must be created from scratch. On the other hand, preexisting content is used in interactive contributions in two ways: editing and composition. Editing preexisting content contributions involves surface deformation problems. Composing new models with preexisting content involves segmenting and blending problems.

Interactive contributions provide a set of operators to users. Some interactive contributions have a plethora of operators to deform and/or edit new content; others are limited to the problem they solve. In general, the categories of operators addressed in the contributions are: Constructive Solid Geometry, Morphological, and Volume blending.

The interactive contributions do not well support repetitive tasks, which can be time consuming and prone to error. An alternative to creating content where repetitive tasks are involved is to use a procedural approach. A review of procedural content generation contributions is presented in the next section.

2.3 Procedural Content Generation

Procedural content generation consists of creating content through a series of predefined steps; the steps' application is controlled only through initial parameters and predefined algorithms. The steps or procedures involved in the content creation use the initial parameters to know how to synthesize new content.

Two classic examples of procedural content generation are fractals and Voronoi regions. Fractals are particularly attractive in procedural generation because they are aesthetically appealing and because complex structures can be generated with few parameters. Oppenheimer [61] created complex plant-like structures with fractals, Smith [87] created mountain-like structures, and Ebert et al. [20] used fractals to create sea-like surfaces and clouds. Voronoi regions [95] have been widely used in Computer Graphics and computational geometry. The main application of Voronoi regions in Computer Graphics is to partition a surface into regions. Voronoi regions are also used to aid in other synthesis processes. Worley [101] used them as a basis function for texture synthesis, Hausner [34] used them for distributing tiles so that they would not occlude each other and still be packed regularly, and Mould [55] was inspired by them to simulate region growing for surface crack rendering.

Another classic way to generate content in Computer Graphics is by using noise. The concept of noise does not refer to just random noise, but refers to variations. Noise has been widely used in Computer Graphics to add variety to procedurally synthesized content while remaining spatially coherent.

2.3.1 Noise and Procedural Synthesis

The main goal of content synthesis is to create new instances of a content class. If the processes that generate content are repeated without a single variation in their parameters, then the instances of the content class will be the same; therefore, variation must be included in the processes. Variation is a very important concept in procedural content creation. A properly tuned variation methodology will allow the creation of content similar enough between the instances as to identify them as belonging to the same class, yet different from each other. Variation is usually implemented as a form of noise. Some contributions have a variational methodology tightly tied to the problem they are solving but others use more common techniques. One of the most commonly used techniques for variation is Perlin noise [66]. Perlin noise is computed by interpolating between precomputed gradients uniformly distributed on a grid.

Perlin noise has been improved recently; Perlin [67] suggested a new interpolation scheme in 2002. The new interpolation function has degree five and has zero first and second derivatives at each grid node (the grid cells are one unit long). The zero derivatives mean that no discontinuities will

be present in adjacent value regions when derivative operators are applied. For example, normals (which are the result of a derivative operator) will transition smoothly without discontinuities between grid cells; therefore, the shading will not have abrupt changes. The other modification suggested by Perlin is to improve the noise evaluation performance. He suggested to use a small set of fixed gradient directions to distribute on the domain. The fixed gradients are such that the contribution of each gradient to the noise evaluation is simply done by sums and no multiplication is involved, therefore reducing the computational costs.

Perlin noise has been used to synthesize models with characteristics that seem to be non-uniform, random, and with smooth transitions between variations. However, this appeal is only apparent since Perlin noise is a cyclic function (it repeats over regular domain intervals). Among the synthesis applications of Perlin noise are: landscapes, clouds, textures, solid textures, and animations (see Ebert et al. [20] for examples).

Perlin noise is not the only source for random variations in Computer Graphics; for example, Worley [101] extends the concept of noise for textures by defining a set of new basis functions used to evaluate the noise. His noise functions take into account the proximity of control points distributed in the space in order to achieve new texture styles.

Another notable way to compute noise is that of Wyvill and Novins [104]. They computed a smooth-transitioned noise by summing functions. They increased the noise quality by distributing the function centers such that they are more densely and evenly packed than a fixed axis-alignment.

Using Perlin noise to create landscapes suggests the procedural creation of 3D content. Landscapes are created with Perlin noise as height fields. A height field is a 2D array with a height value per cell. The height values are computed with a noise function. Height values have also been used to synthesize planet surfaces where the height values are mapped to a spherical surface instead of a plane. The aforementioned examples show how content can be created by computing displacements (height values) on a surface; however, I do not consider such an approach 3D content creation. In this document, 3D content creation is the arrangement of elements created in a 3D domain and not just displacements from a surface, as described in the next section.

Perlin noise has several characteristics that contribute to its popularity. Among those characteristics is its scalability. This is an important feature because it changes the scale of the noise function with minimum effort, and since many natural things are fractal (have various levels of self-similarity) the scalability feature makes Perlin noise suitable to model all these things. Instead of creating several different random number generators for each level of detail, one can easily add noise functions at different scales.

Procedural content involves generating both geometries and textures. The concept of texture as a bidimensional parametric patch mappable to a 3D surface was introduced by Catmull [12] in 1978. Textures are used in Computer Graphics to mimic detailed surfaces without requiring

the geometry to represent such detail. In other words, textures simplified the creation of detailed surfaces as well as reduced the resources to store (memory) and render (computing time) surfaces. Hand-making textures is a time consuming problem. Texture creation or texture synthesis is a wide field in Computer Graphics and receives significant contributions yearly.

The goal of this project is to synthesize geometry rather than textures (as bitmaps), but texture synthesis has a big presence in Computer Graphics and some of its techniques have been extrapolated to synthesize geometry. Because of this and for completeness purposes, I review selected research related to texture synthesis in the next section.

2.3.2 Texture Synthesis

Texture synthesis is a broad area of Computer Graphics. The earlier techniques focused on creating textures as a result of some simulation that "painted" the textures. New techniques makes use of existing textures to synthesize new ones. These techniques compose texture elements (texels) into new textures. Additionally, different techniques have been designed to preserve important features (such as high-frequency detail) of the textures.

Figure 2.6 summarizes the main approaches for synthesizing textures. First, textures are created by modifying parameters of mathematical equations (Figure 2.6.a). These are constructed to create patterns based on the equations properties, such as periodicity, attenuation, and scaling. Second, textures are the visualization of simulation processes (Figure 2.6.b). Third, textures are obtained by modifying parameters on a multi-scale synthesis process. The multi-scale representations are usually obtained during an analysis preprocessing (Figure 2.6.c). Finally, textures are obtained by copying texels or blocks of texels from a sample texture and pasting them on a canvas to synthesize new textures (Figure 2.6.d). This technique needs metrics to both select and place texels (or blocks of texels) from the source and placed them on the canvas, respectively. When copying blocks of texels, some researchers complement this technique with a methodology to adjust boundaries between pasted blocks.

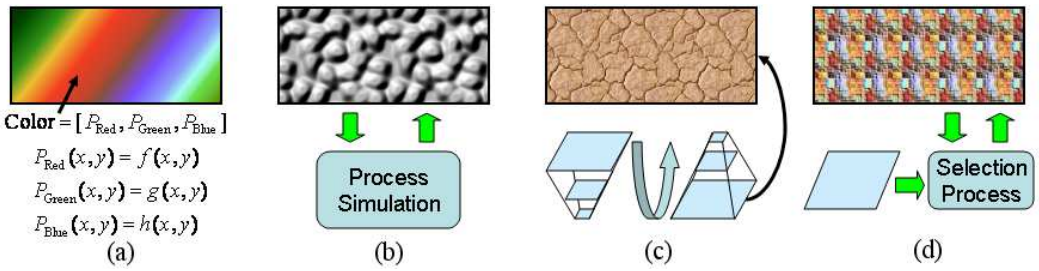

Figure 2.6: Approaches to synthesize textures: (a) Direct computation of patterns, (b) Simulation, (c) Multiscale processing, (d) Copying patches of existing samples.

Procedural texture synthesis creates new textures by using texture samples (patches) or by directly synthesizing individual texels. Individual texels are usually created to form some specific pattern. The patterns have been expressed in different ways by different authors. Perlin [66] and Peachey [15] independently defined the concept of "solid texture" by defining a texture through functions with 3D space as their domain.

Another approach of synthesizing textures is to create a texture such that its texels can be reused several times and still look as one single texture. Neyret and Cani [59] used existing textures to compose new ones. They reported a method to create tilable texture patches (see Figure 2.7).

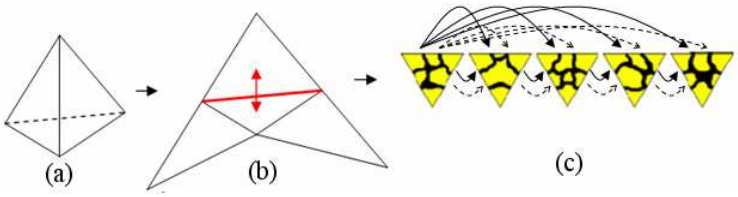

Figure 2.7: Tilable texture principle: (a) The surface is a tesselated arrangement of polygons, (b) Textures coherence worries only about the border between the polygons, and (c) Texture patches are created so that they share similar borders.

One way to create complex patterns in textures is to simulate processes that generate such patterns. Turk [93] and Witkin and Kass [100] simulate reaction-diffusion processes to create stripes and circular patterns on textures. Similarly Walter et al. [96] created stripes and spots on textures by assuming that texels were created from an original cell (i.e., they are "clones"); see Figure 2.8. The simulation of a natural process to create textures expands the texture synthesis field. Yet, some of nature's patterns are created by unknown processes.

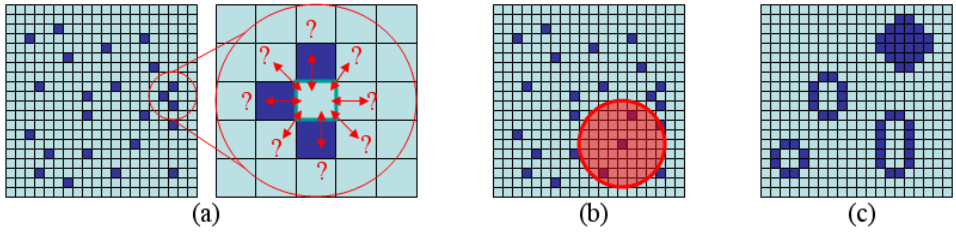

Figure 2.8: Reaction-diffusion simulation for texture patterns generation: (a) Turk [93] considered the interaction between adjacent elements of the texture, (b) Walter et al. [96] used the radius of effect, (c) Both approaches generated spot- and stripe-like patterns.

Some texture patterns are difficult to represent as a composition of implicit functions or simulations. Another approach to create textures is to reassemble pieces of an input texture to synthesize similar textures. A common approach [70, 35, 16] is to make a pyramidal analysis-synthesis of the

sample texture. In the analysis stage, the texture is reduced in scale. A pyramid-like stack of scaled down textures is built; at each step, descriptors of texture properties are computed. In the synthesis stage the process is inverted, scaling up from a simple texture (the pyramid's peak) to a bigger one; the descriptors of the analysis stage are used to synthesize a new texture at each pyramid level (see Figure 2.9). Different descriptors have been used with this technique. Portilla and Simoncelli [70] used statistics to summarize the relations between samples of the original texture, Heeger and Bergen [35] used color histograms, and DeBonet [16] used edge and line filters to preserve high frequency information. The previous examples show how pyramid-like structures help to preserve some texture features at different resolutions. However, texture features can be preserved without a multi-scale structure.

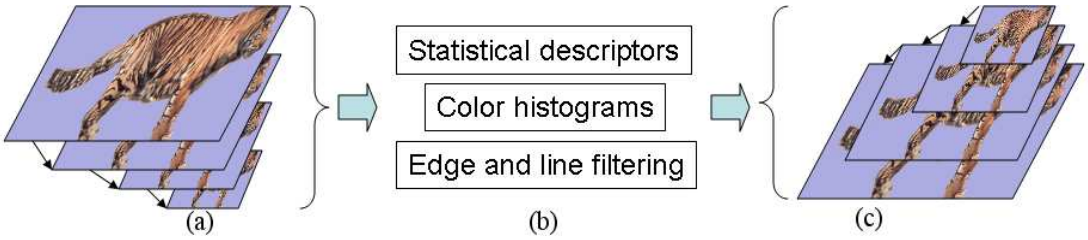

Figure 2.9: Pyramid-like texture synthesis consist of two main steps: (a) Analysis, and (c) Synthesis. The analysis use different ways (b) to describe the texels per pyramid level at the analysis process, which are used back in the synthesis process.

Another approach to preserving the structure of textures is to copy patches of the sample texture to create a new texture (see Figure 2.10). The patches can be as small as a single pixel in the sample texture. Efros and Leung [21] synthesize new textures by copying an initial set of pixels from the original sample; new pixels are added to pixels already copied in a spiral order. The criterion for choosing a pixel is that it belongs to an area in the sample texture *similar* enough to the area surrounding the new pixel in the synthesized texture. Similarly, Wei and Levoy [98] synthesize textures by adding new pixels chosen from the sample texture. The synthesized texture is initialized with random noise. Their algorithm consists of changing each pixel of the random noise for a pixel of the original texture. An alternative to copying pixels from sample textures is to copy subsets of adjacent pixels and patches from the sample texture. Such a technique was introduced by Efros and Freeman [22] and Liang et al. [44]. Later such an approach was refined by Nealen and Alexa [57]; their approach was to copy patches as large as possible in order to reproduce the texture's global structure. Their patch size is determined by an error measuring the discrepancies between the overlapping regions of the copied patches.

A characteristic of sample based synthesis is the need of an initial pool of content to exploit in order to synthesize more. However, new techniques allow to use an initial source as a sample and then synthesize according to this. This technique has the issue that an initial source must exist in

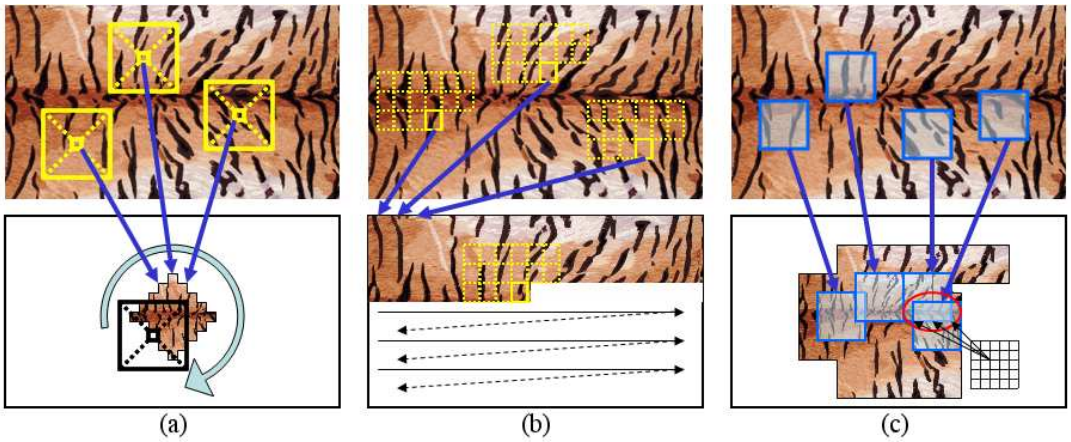

Figure 2.10: Texture synthesis by copying patches: (a) Efros and Leung [21] copy pixels based on a neighborhood measure, (b) Wei and Levoy [98] copy pixels based only on the pixels already copied, and (c) Nealen and Alexa [57] copy full patches and adjust overlapping texels.

order to synthesize more of its type. The source texture must contain all features desired in the synthesized textures.

A problem with copying patches from a sample texture is that the patches may not be tilable, and therefore artifacts in the synthesized texture appear. One way to solve this problem [16] is by building the texture from a low scale representation to a high-scale representation and inherit the high-frequency structure from level to level, i.e., make use of a pyramid representation as described previously. Nealen and Alexa [57] used another approach; they adjusted the pixels of the overlapping regions so that each will have valid neighbors according to the sample texture. The adjustment consists of re-synthesizing invalid boundary pixels. A pixel is marked as invalid if it has not been synthesized previously, and an error measure exceeds a user-defined error value. The error measure is computed with the color components of the original pixel and the pixel synthesized in a previous step. The re-synthesis process uses a mask created with the invalid pixel's boundary; the mask is used to choose the best pixel from the sample texture and replace it with the invalid pixel. Re-synthesizing invalid pixels creates seamlessly tilable patches that are used to synthesize new textures.

2.3.3 3D Synthesis

Three-dimensional synthesis is similar to the area of texture synthesis. Many of the techniques are inspired by those of texture synthesis. Three-dimensional synthesis started by arranging 3D primitives in space to create 3D structures. Some approaches create the whole 3D structure in space or add detail to existing structures. Different techniques solve different problems related to

the composition of 3D elements. Their solutions vary from the proper scaling and positioning of the elements to modifying the boundaries of the elements.

Procedural geometry synthesis involves the creation of 3D models by specifying the position and orientation of the 3D content components. A technique widely used to procedurally generate 3D models is the L-Systems, which are parallel rewriting grammars in which grammar symbols represent a model component. L-Systems were introduced by Lindenmayer [45] as an attempt to model developing biological systems. For example, when modeling plants the symbols represent leaves, flowers, fruits, and stems. The rewriting rules of L-Systems are used to replace symbols in strings (see Figure 2.11). A string is a model's structure representation (abstraction), and the rewriting rules are the model's structure modifications. Prusinkiewicz et al. [75] were the first to assume that simulating the developmental process of a plant creates a model that realistically resembles the real one.

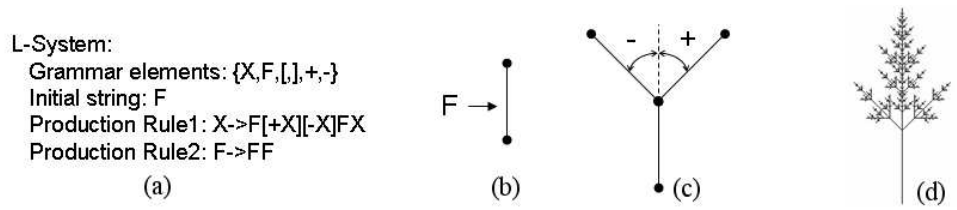

Figure 2.11: (a) Components of an L-system. (b) Some of the grammar elements are replaced with rendering primitives; for this case, all 'F's' are replaced with a straight line. (c) Some of the grammar elements define changes on the rendering process; for this case, '+' and '-' alter the line orientations by 45 degrees. (d) Example of the branching structure obtained after five applications of the production rules. Images taken from [87]

.

L-Systems have been expanded throughout the years. Prusinkiewicz et al. [73] introduced the concept of differential L-Systems (dL-Systems) in an attempt to facilitate smooth animations of growing plant models based on L-Systems. Later, Prusinkiewicz et al. [74] extended L-Systems to incorporate environmental information. In this case the application of rewriting rules is controlled by environmental variables. A further improvement was presented by Mech and Prusinkiewicz [53], who presented the open L-Systems (oL-System). In oL-Systems the control of rewriting rules in a particular model not only includes environmental variables, it also considers the model's state (represented by a string) and the state of other models in the simulation. Similarly, Renton et al. [79] made use of observational models to define probabilistic functions that govern the rewriting rules. The probabilistic functions specify how likely a rule is to be applied based on some simulation parameters. Recently, Parish et al. [63] used L-Systems to synthesize cities. L-Systems are used to create the building geometry and distribution along the streets; the building distribution takes

into consideration local and global restrictions.

L-Systems have many extensions that in the end they seem to resemble a state system with conditioned transitions rather than just a state engine. Early uses of L-systems showed great results in creating branching structures, but they were too perfect, they lacked the "accidents" that occur in nature. This triggered the generation of stochastic L-systems, where production rules were executed by a stochastic function. Later, these functions were linked to simulation processes to obtain more control over the production rules without losing the natural look of random variations. In other words, the decision whether to execute or not a production rule took into consideration the state of the system, which is equivalent to having states (execution of a production rule) governed by transitions (events on the system).

Approaches using grammatical representations have the advantage of providing complex branching structures, but each model class requires a proper grammatical rule design. It is worth noting that grammatical representations work on a discrete domain, i.e., the results will be clearly defined for each of the production-rules application (to the string), but intermediate forms are not defined between production rules.

L-Systems have great acceptance in Computer Graphics, but other techniques to synthesize 3D content have also been used. Viennot et al. [94] used a matrix representation of binary branching structures, in which matrices are linearly combined to synthesize branching structures. Smith et al. [88] synthesize bracket-based structures by optimizing the forces represented on the structure. Their force system considers the mass of the structure itself as well as the mass of additional constructions. Their results include railroad bridges, bridges, and an Eiffel tower, each with additional constructions like highway lanes and observational decks.

Another approach to procedurally generated 3D models is to synthesize detail on the model surfaces. Fleischer et al. [23] created models with richly detailed surfaces. Their technique distributes, over a model's surface, a set of geometry instances representing thorns and scales (see Figure 2.12.c). The distribution follows the simulation of cell interactions over the surface; each cell is a representation of a surface segment and some associated value relevant to the simulation.

Porumbescu et al. [71] procedurally generate 3D models; their technique is known as shell mapping. Shell mapping is based on a function that maps a three-dimensional volume to a surface. The mapping creates a distribution of bounded volumes, along the surface; the prisms are stretched and squeezed according to the surface shape where they are mapped. The prisms are later replaced with geometry models that are deformed according to the prisms they are replacing (see Figure 2.12.d). The global effect of shell mapping is the creation of new models with richly detailed surfaces; the surface detail consists of geometries deformed to follow the surface contour.

Other approaches to creating 3D content procedurally have been inspired by texture synthesis techniques. Some of them use the sample-based approach, in which existing elements of 3D models

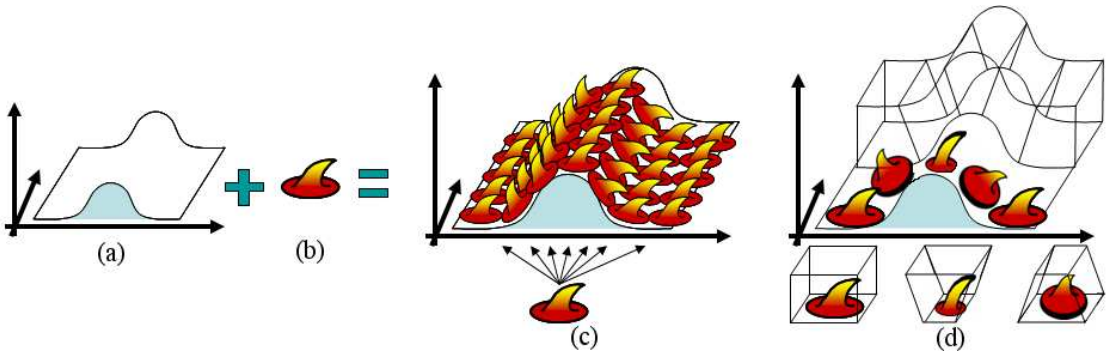

Figure 2.12: Approaches to detail synthesis through geometry composition. (a) Surface to add detail. (b) Geometry representing the detail. (c) Fleischer et al. [23] distribute detail instances over the surface. (d) Porumbescu et al. [71] distribute deformed volumes over the surface; the volumes transformations are later mapped to the detail instances.

are used as primitives to create new models. Lagae [41] used distance fields to represent 3D models and synthesize new ones by copying similar 3D segments, a technique very similar to the techniques of Wei and Levoy [98] and Efros and Leung [21] for texture synthesis. Similarly, Bhat et al. [5] extended the same concepts to work on volumetric models. They used vector fields as descriptors of the model surfaces; the descriptors are used to find matches on the sample geometry.

Another approach is to manipulate the description of the model. New models are created by manipulating the description of the initial model, instead of using primitives to compose new models. Cutler et al. [14] manipulated signed distance fields. The manipulation is implemented through a set of operators and material representations implemented in their own scripting language. Sumner et al. [90] analyze deformed meshes to replicate their deformation. Their technique makes use of inverse kinematics to analyze a mesh in two different poses; the parameters obtained are later used to synthesize any similar pose for the same mesh.

The approaches for synthesizing 3D content that are inspired in 2D procedural synthesis techniques have some requirements to fulfill (see Figure 2.13.a-f). To properly apply a sample-based approach for procedural 3D content generation, one is required to provide the following: proper geometry representation, similarity definition, definition of the geometry basic elements, and a blending technique. The geometry representation establishes the terms in which the other elements are defined (similarity definition, basic elements definition, and blending technique); it also affects how the techniques for texture synthesis are reused. The similarity definition is required to measure the elements of the geometry. Element measurement is needed to identify which element is the best to use during the synthesis process. Measuring an element cannot be done without a description of the elements geometry. The blending technique is required to seamlessly put together the geometries' basic elements in the synthesized geometry.

Figure 2.13 illustrates the elements of the composition approach for geometry synthesis. First, a suitable surface representation is required for the model components; examples are pure-point surfaces, polygonal meshes, and level sets (Figure 2.13.a) Second, a descriptor of the model components is required to measure how similar are the components (Figure 2.13.b). Third, the model components require to be defined; that is, which building blocks are used to compose new models (Figure 2.13.c). Fourth, a way to join together components is required to assemble building block between them (Figure 2.13.d). Finally, the model generation by composition often uses a hierarchical representation of the models to be created to specify the dependencies between components (Figure 2.13.e).

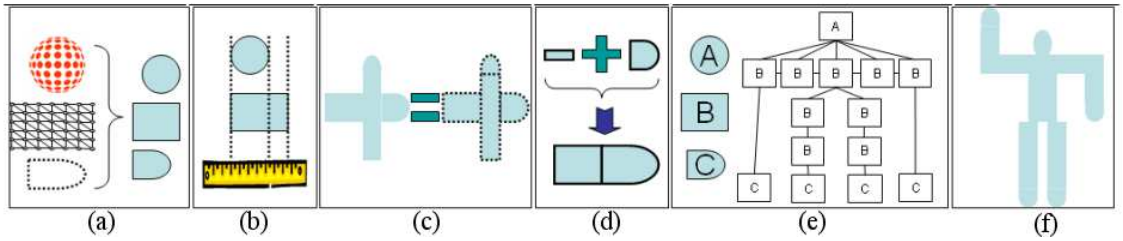

Figure 2.13: Elements of the composition approach for geometry synthesis: (a) Surface representation, (b) Similarity metric, (c) Model primitives definition, (d) Blending specification, (e) Inter-component relation, and (f) Composed model

Some 3D synthesis approaches are more problem specific but they have some general requirements also. A common requirement found in the reviewed contributions is the need to provide an inter-component relation ([87, 97, 94, 73, 74, 53, 72, 75]). The inter-component relation is needed to provide a synthesis reference; the reference is required to locate and orient the elements of the models during synthesis. Approaches not involving grammar representations or not mimicking a 2D synthesis approach are not as common in the field.

Different techniques and particular contributions have been reviewed in this chapter noting their main advantages and disadvantages. The Computer Graphics community has interest for techniques to automatically create content for Computer Graphics, for both textures and geometry types.

Several approaches have been addressed by different authors. The approaches vary from CAD to automatic generation. Despite its name, the automation approach always requires a minimum of user interaction. The interaction ranges from parameter specification to partial generation of content. The automatic approach allows a user to generate huge amounts of content but lacks variety in the type of content. That is, different classes of content can not be generated with a process that generates different instances of a single class.

The review of the field has showed me different approaches for content generation. The tex-

ture generation results educated me on how to deal with surface inter-element interaction and manipulation. Additionally, these results also helped me to visualize how two-dimensional fields (textures) can be used to modify three-dimensional structures (geometry). Furthermore, I realized of the power and simplicity of the composition from the grammar-based content generation results. Finally, I extrapolated the need for a stage-controlled automatic generation approach from the modifications reported to grammar-based approaches.

The different results of the automatic content creation area inspired me. My research fits in the automatic content creation area, 3D content in particular. The 3D synthesis is an active area in Computer Graphics; the composition approach has received several contributions, but the content modification has also not been addressed often. I think this happens because such approaches have the problem of how to control the modifications of existing models to create new ones and because current model representations require high maintenance if modified.

I have addressed the two main problems of modifying 3D content. First, I generated new content by controlling deformations of existing models, I controlled the deformations using a state-transition system which is the trend of structural synthesis. Second, I supported the application of deformations by defining a novel way to represent 3D models that requires less maintenance if modified; I specify deformations by composing velocities.

Also, the literature review allowed me to perceive the tradeoff between automation and diversity of content. I focused on automatic generation of geometry for biological-like surfaces because of their demand in Computer Graphics and because they are costly to produce. I decided this because of the complexity on such surfaces. Even though, my research can be used to automatically generate any class of geometries, it suits best the biological-like surfaces.

Additionally, my research has elements of interactivity and procedural content generation. However, these are mainly features to demonstrate my research and visualize results. I describe the rest of my research in the following chapters.

I consider my contribution to be an original and significant one since I am addressing a problem that commonly appears and also because in my literature review I could not find my ideas implemented. I found research that is similar to mine, and in some cases, inspired me for my research. The most similar research I found is that of Lawrence and Funkhouser [42], Szelisky and Tonnesen [91], and Keiser et al. [39].

Lawrence and Funkhouser [42] modified existing models by deforming their surfaces. They used velocity-based deformations. Their velocity reflected surface displacements on the surface normal direction, constant displacement in an arbitrary direction, and proportional to the surface mean curvature. The velocities are interactively mapped on the model's surface using a painting metaphor. They used a three-channel color field that is painted on surfaces. This paint encodes the velocities that are associated with the surface elements, which move in space according to

their velocities, deforming the surface. They used two surface representations for their research: polygonal meshes and level sets. They resampled the surface previous to its deformation when using polygonal meshes to avoid aliasing problems, their resampling consisted of subdividing the polygonal mesh where high frequency paint has been used. On the other hand they did not have to provide a resampling mechanism when using level sets, but they had to use a volumetric paint to specify deformations; this type of paint is a three-dimensional field of paint-coded velocities. They presented various models generated by interactively painting velocities on their surfaces to deform them.

Lawrence and Funkhouser provided a great interface for surface deformation. A user can easily modify a model's surface by deforming it, and the deformation specification is simple and straightforward. However, they did not explore alternatives for the velocity specification, for example, a velocity whose intensity varies according to the surface height would require a user to manually estimate the height. Furthermore, any sequence of deformations to create a particular model requires a user to interactively paint them. Moreover, their system is subject to the disadvantages of using polygonal meshes or level sets. Polygonal meshes may self-intersect and are prone to have large density variations despite of resampling. The level sets are too slow for interactive use. Additionally, they did not prevent the surface from self-intersecting when using polygonal meshes.

I addressed the drawbacks of Lawrence and Funkhouser in my research. Even though we use velocities to specify deformations, my velocity specification is more general than theirs since velocities can be synthesized with information obtained from the model's surface in my research. I defined a hybrid surface representation instead of using polygonal meshes or level sets. Volipoc is an explicit surface representation that updates at interactive frame rates. Additionally, I paired Volipoc with a resampling mechanism and deformation control; these allow me to deform models without worrying about topology changes (including self-intersections) while avoiding aliasing problems by keeping a quasi-uniform surface density. Finally, I automate the model creation by mimicking the interactive assignment of deformations with a finite state machine.

Szelisky and Tonnesen [91] also created models by deforming their surfaces. They simulated elastic surfaces that deform to create new models. They simulated elasticity with an inter-particle force system that pulls and pushes particles until an inter-particle energy is minimized. The modeling is done by using operators that compress and stretch a surface as well as "cut" it; "cutting" a surface was done by applying a force that makes the inter-particle force exceed a threshold, thus separating the particles. Their particle definition include several physical properties as well as an orientation in space.

Szelisky and Tonnesen's approach is a great tool to simulate the effects of deforming different materials. Similarly to Lawrence and Funkhouser they did not address the automation of the model generation. Additionally, the energy minimization process makes the particles keep moving

until the energy has been dissipated, which could lead to unwanted deformations during the model creation.

I addressed the drawbacks of Szelisky and Tonnesen in my research. I provide a more direct interaction with the surface so that the user can more straightforward design deformations to produce the desired outcome. Additionally, I provide an additional abstraction layer that automates the model creation process.

Another similar recent research result is that of Keiser et al. [39]. Even though Keiser et al. also used an oriented-points surface representation, their surface representation simulates elasticity and viscosity through the minimization of forces between adjacent surface elements. Their adjacency definition is different from mine since the numbers of neighbors to be used in their inter-particle simulations is limited and is not recomputed every simulation step (as happens in my case). They also consider particles to simulate the behavior of the models internal volume, which also affects the behavior of the surface, thus, making their surface behavior completely different from mine.

The composition generation approach may be confused with the structure generation approach. The composition generation approach aims to create new models through adding or replacing existing parts of a model; while the structure generation approach manipulates models' structural representations to create new ones. Some research contribute in both approaches, for example, Ijiri et al. [37] report a software tool for floral model creation. Their tool allows the creation of different L-System-based flower models, which would be considered as a contribution in the structure generation approach, but their tool also allows the replacement of the models' components (pistils, petals, stems) by others stored in a database, thus, a contribution in the composition approach. The composition approach for content creation has the big advantage of using content that has already been created. A disadvantage of this approach is the need for properly segmenting a model, although this activity is usually left to a user. Also, when a large amount of models is available, a system to properly index them in a database is required. Another disadvantage of this approach is the need for algorithms to properly modify or complete the models' parts so they blend together properly. One way to deal with the previous problems is to create or *grow* models' elements from the surface of a initial model. Such approach would require the capability to deform a model's surface and is the approach I have undertaken as described in the next chapter, the state of the art review is described in the following section.

Chapter 3

Surface Representation

My Automatic Model Creation scheme uses a new surface representation that contains the advantages of pure-point surfaces and those of polygonal meshes. I call the new surface representation Volipoc. Pure-points surfaces do not require to handle connectivity; that is, points can be added or removed from the surface without worrying about leaving points unconnected to the surface. A disadvantage is that the surfaces are not continuous; therefore, pure-point surfaces require an interpolation method to complete the gaps between points, and when points are removed from or added to the surface, the interpolation method may drastically change the surface shape. On the other hand, polygonal meshes have the advantage of representing continuous surfaces without holes in it (the interpolation rule is perfectly defined). However, they have the disadvantage of not well handling deformations on the surface, especially when the deformation involves stretching and compressing the surface.

My Automatic Model Creation approach considers the generation of models by deforming surfaces. I did not use polygonal meshes because the cost of controlling connectivity in large meshes is too high. On the other side, I did not use pure-point surfaces because I also wanted a continuous surface for the user/designer. Pure-point surfaces also have another disadvantage: surface details may be lost when rendered as a continuous surface. Because of this, I designed my own surface representation for my Automatic Model Creation goal.

Volipoc defines a model in terms of points in space and links between points. A model's surface S is a pair consisting of two sets (see Figure 3.1). The first set, PS, contains oriented points of the model's surface. For deformation purposes, each point has a velocity vector associated with it. The second set, LS, contains edges (pairs of points from the first set). My representation differs from the traditional polygonal mesh representation by not being restricted to form a surface of tesselated triangles. Also, my representation differs from pure-point surfaces by establishing a linkage between points.

The polygonal mesh representation is not used because it may generate large edges during stretching/compressing deformations. I wanted to avoid the additional processing required to control these long/thin triangles. On the other hand, pure-point surface representations could not be used either because resampling may lead portions of the surface to blend together. That is, an

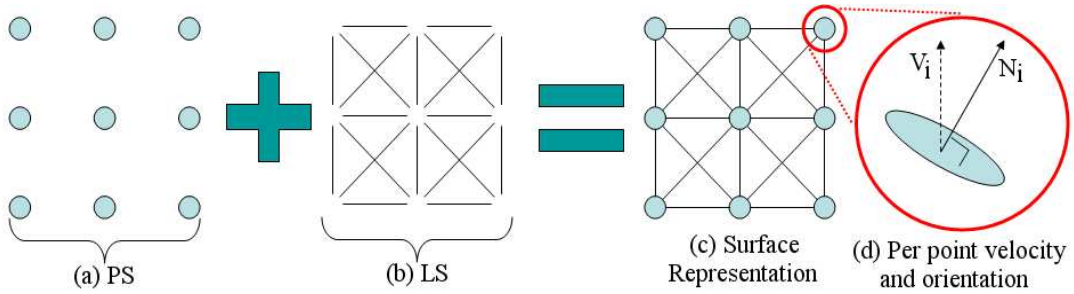

Figure 3.1: (a) A set PS of oriented points in space, (b) A set LS of links, (c) The links from LS are used to define the shape of a surface by linking points in PS, and (d) Each point has its own velocity and orientation.

interpolation rule would require properly distinguishing between points that are too close to each other but belong to different sections of the surface and those that are not too close. My new surface representation is a hybrid between polygonal and pure-point surfaces.

Large edges are a problem for surface deformations because they cause the surface not to deform uniformly. Figure 3.2 shows how a surface with large edges does not deform uniformly. Figure 3.2.a-b show two meshes with uniform and non-uniform densities respectively; the first mesh has uniform triangles while the other has several edges that are distinctively larger than the others. Figure 3.2.c shows a desired shape for the meshes, drawn in the plane X-Y orthogonal to the plane X-Z of Figures 3.2.a-b. Figures 3.2.d-e show the resulting meshes after deforming to obtain the shape of Figure 3.2.c. Note how the large edges of the second mesh produced a surface that does not reflect the intended surface of Figure 3.2.c.

Volipoc avoids the generation of long edges with a resampling mechanism. It adjusts the surface density by adding or removing points from the surface. This occurs when points move far or close respectively, that is, when long edges are detected (see Section 4.1).

I designed Volipoc to facilitate surface deformations. In this project, surfaces deform to create new three-dimensional models. The surface representation is explained in detail throughout the rest of this chapter. Section 3.1 explains the critical elements of the surface representation. Section 3.4 an acceleration using a grid. Section 3.3 explains the behavior of Volipoc under deformation. Section 3.5 summarizes the surface representation contribution.

3.1 Surface Elements

For Automatic Model Creation purposes I need a surface capable of supporting arbitrary deformations. As I explained in Chapter 1, such a surface would be a point-based surface. Point-based surfaces have the advantage of not presenting irregularities when deformed; therefore, the additional

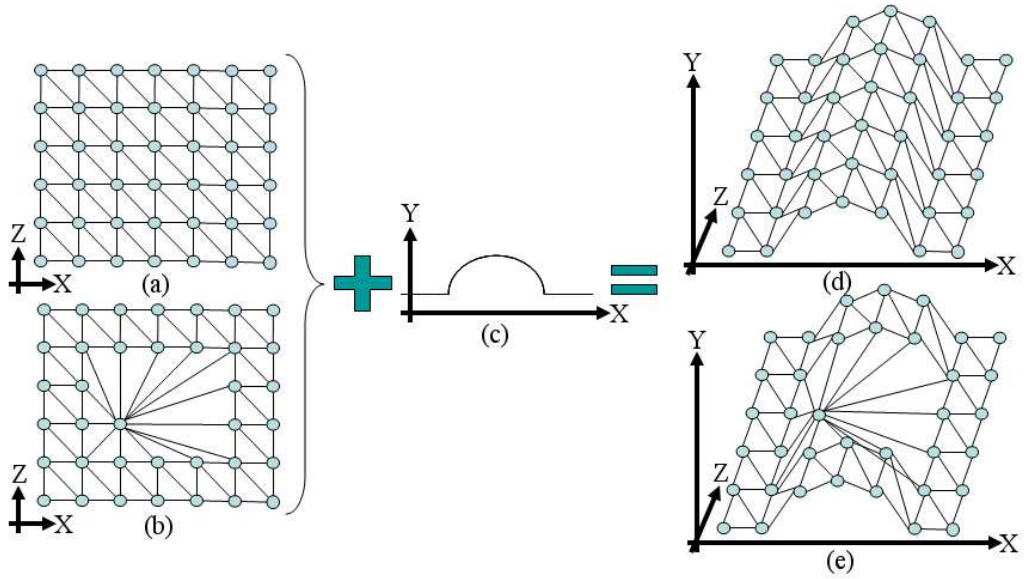

Figure 3.2: Example of a deforming polygonal mesh with uniform (a) and non-uniform densities (b). When a deformation (c) is applied, the uniform mesh (d) reflects better the intended deformation than the non-uniform mesh (e).

computation to correct such irregularities is avoided. Additionally, I want a surface that correctly reflects the intended deformations and does not require interrupting the deformation process; such features are important for this project since I aim for the automatic creation of models.

3.1.1 Point Cloud

The first key element of Volipoc is a set of points in space. I define the set PS as the set of point samples from the surface.

Each point p has a set of associated properties. A point $p_i \in PS, i \in (1, ..., |PS|)$ is a tuple of the form $p_i = \left\{\overrightarrow{P_i}, R_i, G_i, B_i, \overrightarrow{N_i}, \overrightarrow{V_i}\right\}$, where $\overrightarrow{P_i} = \{x_i, y_i, z_i\}$ are the coordinates of the point in space, R_i, G_i, B_i are color components, $\overrightarrow{N_i} = \{n_{xi}, n_{yi}, n_{zi}\}$ are components of a normal vector, and $\overrightarrow{V_i} = \{v_{xi}, v_{yi}, v_{zi}\}$ are components of a velocity. Each point is an "oriented point" since it has a normal (orientation) $\overrightarrow{N_i} = \{n_{xi}, n_{yi}, n_{zi}\}$ and an associated velocity $\overrightarrow{V_i} = \{v_{xi}, v_{yi}, v_{zi}\}$. Finally, each point also has a color represented by its red (R_i), green (G_i) and blue (B_i) components ($R_i, G_i, B_i \in [0, 1]$). Figure 3.3.a illustrates the elements of the tuple, and Figure 3.3.b is a snapshot taken from Vebam displaying four points. Note that the oriented points are rendered with a circle primitive; the colored arrows indicate their velocities and the red lines indicate their orientation (normal).

To support interaction, the points are rendered with a flat circle in Vebam. The circle has a normal in the same direction as the orientation of the point ($\overrightarrow{N_i}$), and its color is the same as the

point (R_i, G_i, B_i). The circle diameter is set to one unit but can be changed from the user interface. In Chapter 6, I present additional rendering primitives and discuss their effects.

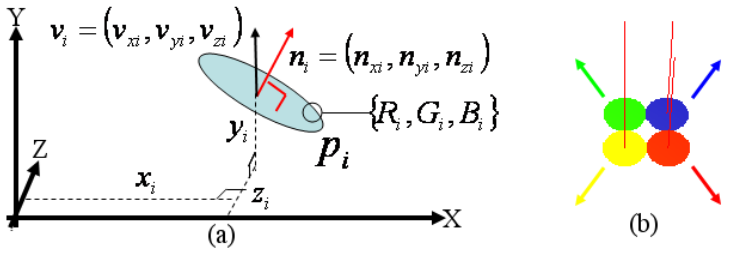

Figure 3.3: (a) Each point p_i has an associated velocity v_i, a normal n_i, and a color represented by its chromatic components (R_i, G_i, B_i). (b) Snapshot taken from Vebam showing four oriented points (colored arrows indicate their velocities).

In addition to its three-dimensional location, a point has an associated normal; points with an associated normal are commonly known as "oriented points" in Computer Graphics. The normal is important for shading a rendered object. I include the normal information in the tuple to avoid the post-processing computation of surface normals.

The velocity information of points is used to deform the surface. The surfaces' points have an associated velocity $\overrightarrow{V_i}$; the velocity is used to compute the new position of a point when the surface is deforming. Details of the displacement computation are presented in Section 3.3.

The color components (R_i, G_i, B_i) of a point are used in the rendering process. The color components are equivalent to the color of the material used to render the surface [24]. I use the color components to visually identify regions of a surface. Additionally, I use the components to express the magnitude of velocities; that is, the color intensity is directly proportional to the velocity of the point.

Additional information can be associated with each point, depending on the future application of the models. For example, texture coordinates could be useful since most models nowadays have textures to improve visual quality.

Pure point surfaces have the disadvantage of not forming continuous surfaces. When rendering a pure-point surface, holes appear on the surface unless each pixel has a point projected onto it [43]. Approaches to solve this problem are:

- Interpolation of the surface. This approach uses the points to construct a surface that exactly matches the points.
- Extrapolation of the surface. This approach obtains a surface that approximates the point cloud.

I facilitated the computation of a continuous surface by adding edges to Volipoc. My approach

is an interpolating approach; that is, I reconstruct a surface that matches exactly the points of the point cloud. The computation of a continuous surface from Volipoc is explained in Section 3.2; the computation is based on selecting from all possible triangles in the surface representation the outermost triangles, thus defining the shape of the surface (see Appendix A); Appendix A contains the steps to compute a continuous surface from Volipoc. I follow this approach because one of the goals of my research (as explained in Chapter 1) is to present to a user a direct mapping between the modeled surface and the rendered one. Additionally, the edges are an essential element during the resampling of the surface.

3.1.2 Volatile Edges

The second key element of Volipoc are links between points or "edges". These links are interconnections between two points of the surface. I define the set LS as the set of links of the surface. For example, figure 3.4.a shows a snapshot from Vebam where four points ($PS = \{p_1, p_2, p_3, p_4\}$) define a square surface; the links in this case are $LS = \{\overline{p_1p_2}, \overline{p_1p_3}, \overline{p_1p_4}, \overline{p_2p_3}, \overline{p_2p_4}, \overline{p_3p_4}\}$

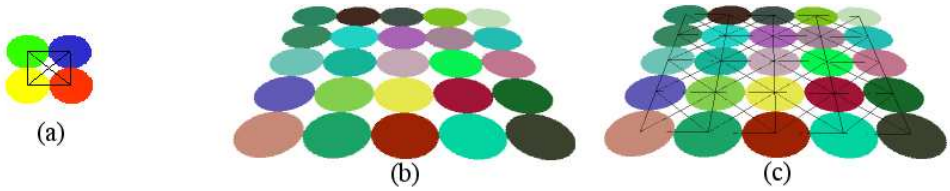

Figure 3.4: (a) Snapshot from Vebam with the same points of Figure 3.3.b but rendering edges instead of normals. (b) Snapshot of a flat rectangular surface taken from Vebam. (c) Same surface but with edges rendered.

Although Volipoc is not a polygonal mesh, a surface within Volipoc does contain polygons (triangles) defined by the points and edges. However, the polygons are not a tesselation of the surface because triangles may overlap or intersect (see Figure 3.4.c).

Edges could have been avoided in the surface representation but they support the goal of automatically creating models without interrupting the deformations for surface repair. Several techniques exist to transform from a point cloud to a polygonal mesh, but such techniques make an approximation of the surface. My edge usage allows me to have an explicit representation of the surface. Additionally, edges reduce the processing required to compute a continuous surface by avoiding the need of applying remeshing operations to interpolated the surface (see Appendix A). The edges are already part of a meshing process; therefore, their reutilization avoids additional computation when computing the continuous surface (or converting to a polygonal mesh).

The size and contents of sets LS and PS can change over time. As stated in Chapter 1, one of

the goals of this research is to provide a framework for Automatic Model Creation; in particular, I want to automate the processes that maintain the surface density. The surface density applies directly to the amount of elements in PS and LS. The processes that change the elements of PS and LS are described in Chapter 4; such processes control the density of the points in space, and set and remove edges between points depending on the deformation of the surface.

3.2 Surface Reconstruction

Different types of surfaces exist in Computer Graphics. I classify them in two types: explicit and implicit. I use this classification to help introduce the concepts related to surface behavior presented in Chapter 4. An explicit representation defines explicitly a surface in terms of three-dimensional coordinates. An implicit representation defines implicitly a surface; that is some portions of the surface must be computed. For example, polygonal meshes and pure-point surface are explicit representations of surfaces.

Explicit representations are often tied with a rendering primitive to use during rendering, researchers have tested different rendering primitives for various reasons. The reasons vary from the rendering time to quality of the image rendered. For example, a sphere renders smoother edges and does not require orientation but does not tile as well as hexagons. Point surfaces are often rendered with circles or squares as rendering primitives. A circle can either be a fan stripe of triangles or a texture with transparent pixels on rectangular polygonal mesh. Either case requires more resources than just a square that can be rendered with two triangles and no texture. On the other hand, rendering with squares may show artifacts (squares' pointy corners) on the edges of surfaces, but they may look better using circles because of their smooth border.

I render the surface representation's point cloud at two levels: interactive and continuous. My interactive rendering uses a simplified surfel rendering [65]; that is, I render points as oriented circles (see Figure 3.3.b). The other level, continuous rendering, reconstructs a continuous surface using the points and edges of the surface representation.

3.2.1 Continuous Surface Computation

Volipoc is an explicit surface representation, and it is a hybrid between a pure-point based surface and a polygonal mesh. Volipoc uses points and edges to define a shape. Volipoc is neither a pure-point based surface nor a polygonal mesh. Volipoc is not a pure-point based surface because it contains edges. On the other hand, Volipoc is not a polygonal mesh because it is not constrained to be a tessellation of polygons. Figure 3.5 illustrates the previous concepts; Figure 3.5.a shows a surface represented by a set of points and a set edges, Figure 3.5.b zooms in the surface, and Figure 3.5.c shows two overlapping polygons making Volipoc not a polygonal mesh.

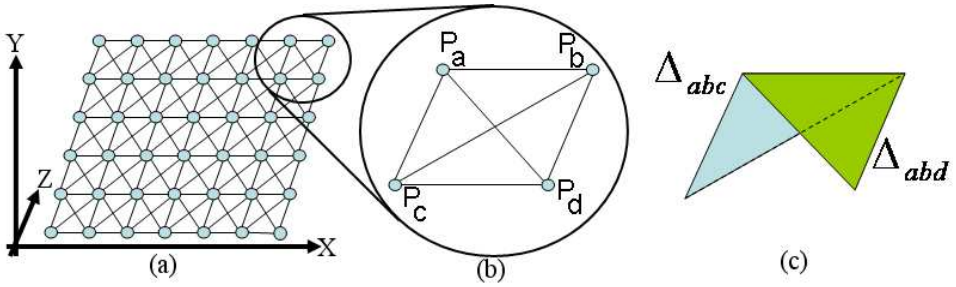

Figure 3.5: (a) A flat surface using Volipoc, (b) Detail of the surface construction, and (c) Overlapping polygons.

Volipoc describes continuous surfaces despite not being a polygonal mesh. The surface defined by Volipoc is that of the outermost shape. Appendix A contains the procedure for computing the outermost shape from Volipoc. The "outermost" notion defines sides of the surface and is represented by normals associated with points. Figure 3.5 shows a surface defined with Volipoc.

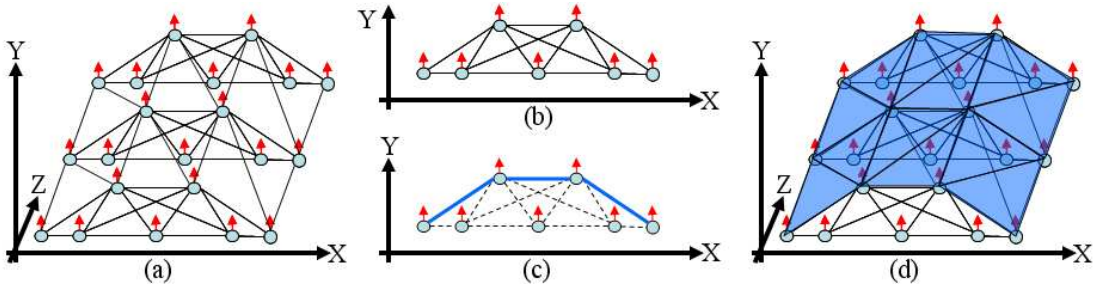

Figure 3.6: (a) Surface represented with Volipoc (red arrows indicate the orientation of points), (b) Cross section of surface, (c) Outermost shape in bold blue lines, and (d) Outermost surface shaded in blue.

A continuous surface is embedded within Volipoc. The continuous surface consists of triangles formed with the points and edges of Volipoc. Since many triangles can exist and some may overlap each other, then the best triangles need to be chosen. The best triangles are the outermost triangles of the surface because these represent what the user/designer sees and has assigned velocities to. In other words, the best triangles are those that are not "behind" others. One triangle is behind another depending on their orientation, which is computed with the orientation of its points. Basically, one triangle is behind another if they both rotate in the same direction around their shared edge, as explained in Appendix A. Furthermore, I used the concept of outermost triangles in an algorithm that converts from Volipoc to a polygonal mesh (see Chapter 6).

3.3 Surface Deformation

The goal of this project is the automatic creation of models for Computer Graphics. The models are created by deforming an initial surface. The deformations are specified in terms of velocities associated with the surface points.

Surface velocities can be generated either automatically or semiautomatically. I use a sophisticated system to produce velocities interactively and procedurally. Both parts allow a user/designer to apply local and global velocities. While the interactive part allows the prototyping and fine tuning of model creation, the procedural part allows automatic assignment of velocities, thus obtaining the Automatic Model Creation. Velocity assignment at surface level is not supported because it does not provide any advantage towards the Automatic Model Creation goal. Note that interactive application of velocities to deform surfaces was previously investigated by Lawrence and Funkhouser [42].

A surface deforms in time by undergoing displacement in different areas. The displacements are obtained by moving the points according to their velocities. Figure 3.7.a shows point p_i whose position (x_i, y_i, z_i) is updated after $\Delta t = 0.1$ seconds to $x_i^{'} = v_{xi} * \Delta t + x_i, y_i^{'} = v_{yi} * \Delta t + y_i, z_i^{'} = v_{zi} * \Delta t + z_i$ (Figure 3.7.b). Figure 3.7.c-f shows snapshots taken from Vebam of the surface presented in 3.3; the snapshots shows the evolution of the surface at every two timesteps.

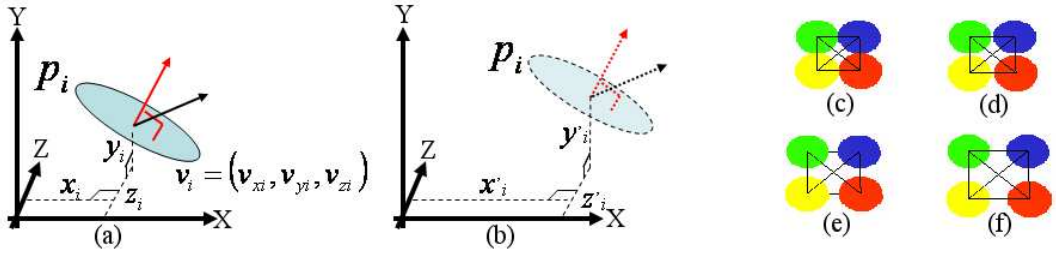

Figure 3.7: (a) and (b) Points' positions are updated according to their velocities and a time step. (c)-(f) Surface from Figure 3.3 deforming at 0.2 seconds intervals.

For simplicity, and since my research focuses on automatically generating 3D models rather than animating them, I chose a constant timestep. I found that a timestep of 0.1 seconds is good enough to make interactive deformations (with surfaces of less than 10,000 points) that appear as smooth deformations. A clock generator produces the time steps; the clock generator is also known as the "Time Handler". Section 4.1 contains a detailed explanation of the Time Handler and its interaction with the other modules of Vebam; for the surface deformation issue, the Time Handler is simply a clock.

Other authors have used velocities to deform surfaces. For example, Level Set Methods [83] track surfaces that vary in time; the surfaces are the zero level sets of functions that change over

time. Another example is that of Lawrence and Funkhouser [42] who used the velocities to deform surfaces; the velocities are painted interactively on the surface by a user.

A deforming surface may stretch or compress. The velocities of surface points can make them move apart from each other or move closer. Therefore, the surface density changes with the deformation. For example, Figure 3.7.c-f shows a surface expanding; the four points have velocities that move them away from each other changing the surface density. As stated in Chapter 1, the surface deformation is simplified when the density is uniform along the surface; assigning velocities to a uniformly distributed set of surface elements is easier than if they were scattered non-uniformly. The surface density is controlled by an automatic resampling mechanism, it uses the distance between points to either remove an edge (when points are too far apart) or create a new point or blend the points into a new one (when points are too close).

3.4 Acceleration using Grid

The surface representation uses a grid that partitions the space where the surface resides. The grid is an arrangement of neighboring cubes aligned with the spatial axes (see Figure 3.8.a).

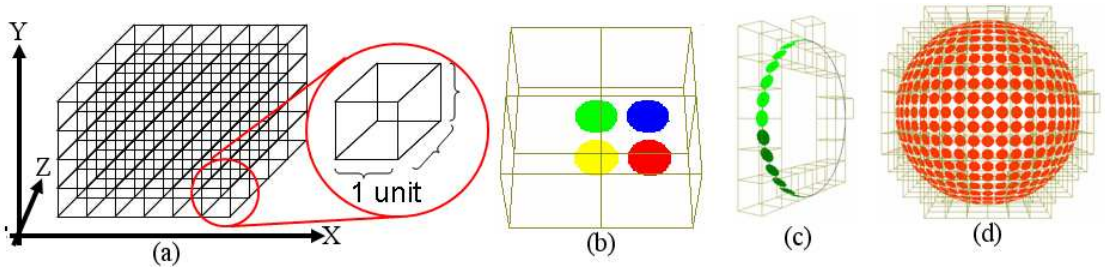

Figure 3.8: (a) Grid, (b) Surface from Figure 3.3.c and the grid cells where the points are contained, (c) A "circle" surface in 3D and the grid cells containing the points, and (d) A spherical surface and the grid cells containing the points.

At the implementation level, each point of the surface knows in which grid cell it resides, and each cell knows what points it contains. This information allows me to implement queries in space; the queries ask about the presence of other points in a cell or surrounding cells. This queries are used to resample the surface and maintain a quasi-uniform point density.

The cell-point information is updated when points move between cells. When a point moves outside a cell, the point and cell links are updated accordingly. That is, the original cell releases the point reference, the receiving cell registers the moving point, and the point updates its cell references.

More than one point can reside within the same cell. There is no restriction to having more than one point in a cell. Points are added or deleted from cells only during the control of the

surface density. The grid is used to improve the control performance but the density is controlled with a different mechanism. Density is controlled by adding or removing points to the surface when it stretches or compresses. The stretch/compression is detected through the length of the edges linking pairs of points, as will be explained in Chapter 4.

Other space partitioning techniques have been used in Computer Graphics; however, a fixed grid is enough for the problems of improving performance and avoiding topology changes. These problems can be solved by any data structure which provides fast querying of adjacent space regions. Even though other structures such as octrees present similar features, they consume more computational resources when updating them than a simple grid. An octree may require updating (blending cells or creating new ones) when the number of points in a cell changes from changes to or from zero while the grid never requires such updates. For example, Figure 3.9 shows a case where the octree has more links between neighboring points than a grid partition. Figure 3.9.a shows that point P_3 has as neighboring points (based on adjacent and diagonal cell connection) four points: P_1, P_2, P_4, and P_5, but Figure 3.9.b shows P_3 has less neighbors in a grid partition: P_2 and P_3.

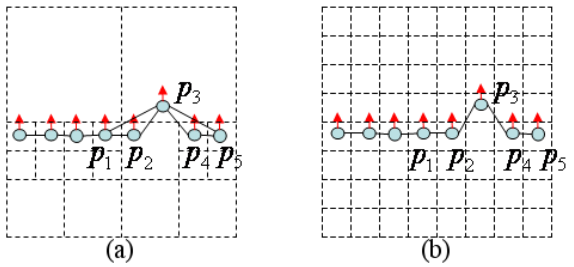

Figure 3.9: (a) An octree space partition may provide more comparisons between neighboring points than (b) A regular grid space partition.

The octree can produce more comparisons because of the irregular size of adjacent and diagonal cells. An octree cell may have more neighbors than a grid cell (see Figure 3.9), and since the number of comparisons between neighboring cells is done by comparing the points in the cells then the octree partition may produce more comparisons. One could argue that an octree could be modified so that only one point exists per cell and therefore all neighboring cells (containing points) would be of the same size and have the same number of neighboring cells. However, such modification would be equivalent to a grid with a dynamic space partition (space is partitioned on demand) but with an increased complexity in its maintenance. Since my interest is to reduce the number of comparisons then I have chosen to use the grid partition because the worst-case number of comparisons is less than that of the octree (see Figure 3.9).

The Surface Evolution Control could have been implemented without the grid; however, such implementation would have been more expensive (computationally speaking) than with the grid.

During the Surface Evolution Control it would have been costly to compare each point against all the other points on the surface, and each point would have been evaluated at every step of the surface simulation. That is, having the total amount of surface points $|PS|$ then the number of comparisons $NumComparisons$ required for the Surface Evolution Control would be $O = (|PS|^2)$, and these comparisons would have been made every step of the simulation. However, with my proposed Surface Evolution Control using the surface representation grid, the number of comparisons is $NumComparisons = n_i + n_j \; i, j \in (0, ..., |PS|)$, $i \neq j$ is the sum of points in the cells (n_i and n_j) involved in the comparison, $n_i + n_j << |PS|^2$ for surfaces with a large number of points. Figure 3.10 illustrate the difference in the number of comparisons between my Surface Evolution Control using the grid and an Surface Evolution Control without using the grid.

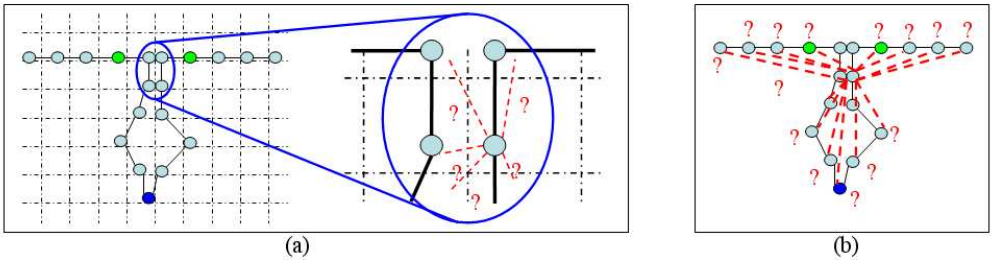

Figure 3.10: The number of comparison in my Surface Evolution Control (a) is notoriously less than without using the grid. (b) The latter case requires comparison of each point in the surface with all the others.

The maximum speed of any point is one unit per timestep. When a set of velocities have been assigned to points on a surface, all are scaled so that the maximum velocity of any point is one unit per timestep. This restriction is implemented so that a point can move only into an adjacent grid cell in one timestep, facilitating grid updates.

The grid also affects the velocity of the points. The maximum velocity of a point is one. This limit avoids the creation of holes on the surface, and thus avoids topology changes. If a point moves more than 1 unit per timestep then the neighboring relation between adjacent points would be lost; neighboring relations or edges are maintained only between points that are within the same cell or with those in adjacent cell. The edges are updated during the resampling process, as will be explained in the next Chapter.

3.5 Summary

In this Chapter, I have presented the surface representation I use to create 3D models through surface deformation. The surface representation basic elements are points and volatile edges. The points have associated information to specify deformations and render the models. The information

consists of an orientation in space (normal), color components, and velocity. The other basic element of the surface representation, volatile edges, are edges that link pairs of points and are used to define the shape of the model. The edges are volatile because they are created and destroyed for automatically resampling the surface during deformations.

Volipoc models a continuous surface despite not being a polygonal mesh. A continuous surface can be obtained with the point cloud and the volatile edges. Furthermore, a polygonal mesh can be obtained from the surface without any surface approximation due to meshing operations obtained from the resampling process.

Volipoc contains two main elements: a set PS of points and a set LS of edges between points. In addition to its position in space, points have the properties of orientation, velocity, and color. Each property is used to support the rendering (color and orientation) and deformation (velocity) processes. The edges support the processes of surface computation and density control, as explained in Appendix A and Chapter 4 respectively.

Volipoc deforms by having velocities associated with the surface representation points. The velocities displace the points through space at no more than one unit per time step. This restriction is needed to avoid topological changes in the surface, and it is linked to a space partitioning structure used for performance improvement.

A 3D grid partitions the space where the models surface deform. The partition reduces the number of comparisons done during the resampling process. This process avoids changes in surface topology.

I have presented a novel surface representation that is a hybrid between polygonal meshes and pure-point surfaces. Volipoc has the advantages of not having restrictive linkages between surface elements and the polygonal meshes' advantages of having a solid surface formation. I designed Volipoc to support surface deformations; these deformations are the basis to create models which is the main goal of this project.

Chapter 4

Surface Behavior

In nature, the creation of many complex objects can be seen as the result of expanding and shrinking forces that deform the surfaces of objects. For example, for a human viewer, coral may be interpreted as a growing, expanding surface. Similarly, quartz can be seen as a growing crystal. Another example are canyons. A human viewer can see canyons as a shrinking terrain. The previous examples show how complex surfaces can be seen as the result of a surface deformation.

I use deformations to create models. The deformations are specified as velocities assigned to points describing the models' surfaces. The surface shape changes when the points' positions are updated by integrating their velocities.

When generating 3D models for Computer Graphics, deformations may provoke two types of problems. The first is big variations in the density of surface components, changing the distance between them (see Figure 4.1), which leads to rendering difficulties. The second problem is topology changes (see Figure 4.2), which may make difficult to track the surface on time. This problem arises when the topology of the model's surface changes. Topology changes when holes appear on a surface and when segments blend together.

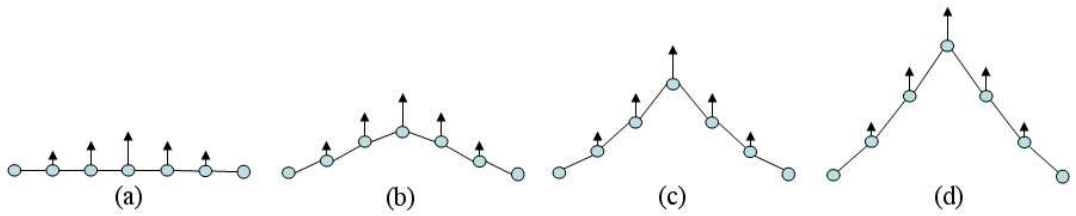

Figure 4.1: Velocity based deformation example: (a), (b), (c), and (d) show a deforming surface whose points' positions are updated at constant timesteps according to the velocities associated with the points and changing the spatial density of the points.

In Section 4.1, I propose an automatic resampling scheme to control the surface density. The scheme consists of resampling neighboring points that move apart from each other (or move closer). Two points are their respective neighbors if and only if a volatile edge exists between them in the surface representation. The volatile edges are used to measure the distance between neighboring

Figure 4.2: Example of topology change due to deformations. (a) Cross section of surface. (b) The surface starts deforming, and (c) the surface continues deforming. (d) The surface topology changes (a hole is created) due to a deformation.

points; if the distance is bigger or smaller than specific values (see Section 4.1) then the edge is destroyed, a new point is created, and new neighboring edges are created.

On the other hand, I prevent topology changes (hole generation and surface blends) with the scheme described in Section 4.2. In general terms, the scheme consists of preventing the resampling from occurring between surface segments if that would produce a topology change. In order to do this, I used the 3D grid of the surface representation to control the resampling and to stop points from producing self-intersections or blends.

4.1 Resampling

Resampling is the first key problem for my Automatic Model Creation. Resampling aims at keeping the density of the surface points within a specific range. Resampling handles the surface density changes when neighboring points move apart and close each other.

In my Automatic Model Creation scheme, an initial surface is deformed to create a model. Since the deformation of a surface can be understood as a displacement over time, deformations are specified in terms of velocities, associated with surface elements. In Volipoc, each point of the surface (element of PS) has a velocity associated with it; the velocities have an arbitrary direction and do not necessarily follow the point's orientation. These velocities are used to compute the point's new position in space at each time step of the surface evolution in time (see Figure 4.1); the Time Handler provides the time step, as explained in the next section.

Time Handler

The time handler is the source of time steps for the system integrator. The integrator is the system component I used to solve the velocity equations that move the surface points. These equations are a simple initial value problem. The equations describe the new position of points based on their velocity, their current position and a timestep.

The new points position makes the surface shape change. The process of modifying the surface shape by updating its points' positions is referred to as "surface simulation" or "surface evolution" in this text. When all points on the surface have had their positions updated then it is said that

the surface has completed a "simulation step" or "evolution step".

Another important function of the time handler is to stop the surface evolution at a user's request. This feature allows the user/designer to modify the model deformation, to load a new model, or to save the current model; therefore, the user is able to direct the model creation as desired. Note that this supports interactive creation of model. Later, this same support is used to automatize the model creation process by mimicking a user/designer with a finite state machine.

Another important function of the Time Handler is to set the moment at which state changes in the Automatic Model Creation process are computed. At the end of each time step the conditions associated with state transitions are evaluated. If the condition is validated then the transition attempts a change of state. The Time Handler functions are shown in Figure 4.3.

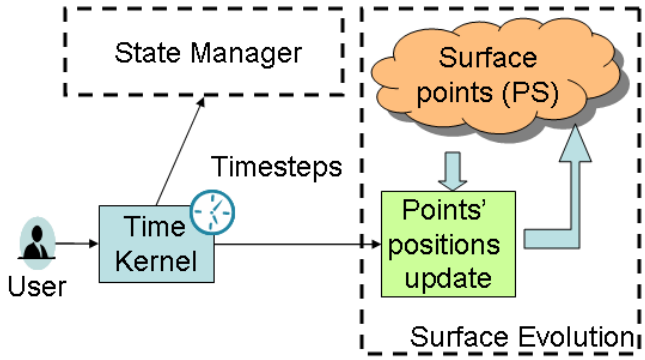

Figure 4.3: The Time Handler provides the timesteps used to update the positions of the surface points and those used by the State Manager (explained in section 5.1).

4.1.1 Resampling Operations

Resampling occurs with two types of operations: spawn and fuse. The spawning operation creates new points to fill gaps formed between linked points. On the other hand, the fuse operation substitutes a pair of linked points that moved too close to each other with a single point.

The two resampling operations produce a coherent surface after the resampling process. They avoid perforations on the surface when the surface expands or stretches. The resampling mechanism maintains a continuous surface. The resampling is always active during deformation; therefore, no segment could detach from the rest of the surface.

Figure 4.4 shows the main steps of the resampling process. Figure 4.4.a shows two initial points with individual velocities associated with them. In 4.4.b the points move, increasing the distance between them, until in Figure 4.4.c the distance increases beyond the unit, triggering the resampling process. Figure 4.4.d shows how the spawned point (green) queries its neighborhood to establish its links to other points, the outcome of resampling. The neighborhood computation is improved

by only using the cells adjacent to the cell of the spawned point.

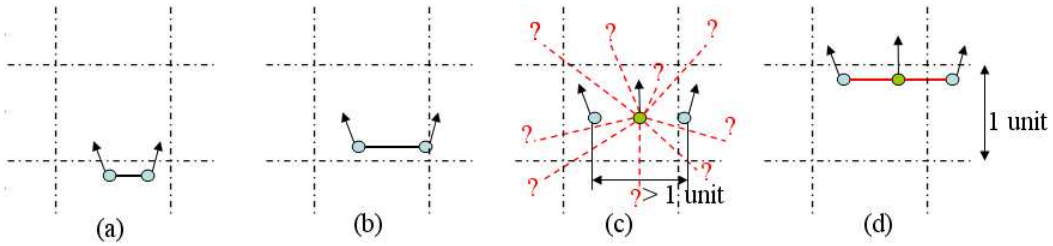

Figure 4.4: Resampling process: (a) Initial condition, (b) Resampling condition, (c) Resampling queries, and (d) Resampling completed.

A surface's edges are used to resample the surface, spawning new points or fusing existing ones. New points are created if an edge (in LS) has a length of more than one unit after the positions of the edge have been computed. New points are added to PS, and their position and velocity are linearly interpolated from the spawning points. The spawning points are removed from LS. On the other hand, if an edge has a length less than half a unit, then the edge's points are removed from PS. A new point replaces the pair of points in PS. The new point's properties are obtained from linear interpolation of the properties of the spawning points.

New points query the surrounding space for neighboring points. If the distance between the new point and another point is less than one unit then a new link between points is added to LS; the new pair consists of the new point and the neighboring point.

To accelerate neighborhood calculations, I employ a space-partitioning structure in the form of three-dimensional grid; each cell has sides of one unit length. The cells hold the surface's points if they are within the cells' boundaries. The grid cells limit the computation to the points in the cell containing the new point and adjacent cells.

Vebam's implementation of the resampling is straightforward. It has no limitation on the number of points per cell. Its only limit is the size of the grid, which varies depending on the available memory. The resampling mechanisms guarantee a quasi-uniform density of surface points. The distance between any two given points is in the range (0.5, 1.0] units.

4.1.2 Normal Computation

When the surface is resampled, the normal of the points created during resampling must reflect, as accurately as possible, the normal of the surface they represent. For example, Figure 4.5.a shows the correct normal of a spawned point when the spawning points move in the same line defined by the spawning points and when the points move on any other direction (Figure 4.5.b).

When a point spawns by the resampling process, most of its parameters can be computed as the linear interpolation of the same parameters of the spawning points. However, the linear

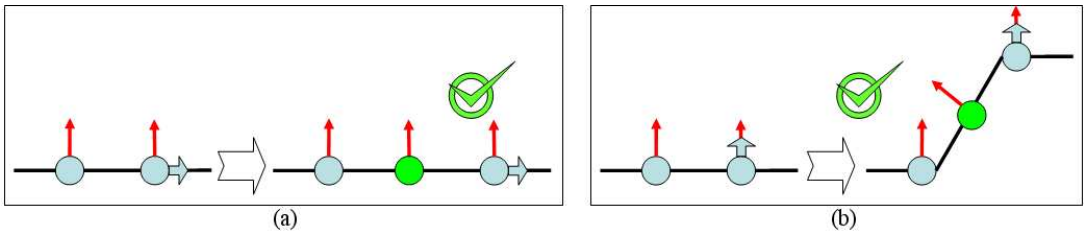

Figure 4.5: Correct normal of a point spawned (green) by resampling of the surface.

interpolation of the spawning points' normals does not provide an accurate normal computation. For example, Figure 4.6.a shows that the linear interpolation computes a proper normal when the spawning points move along the surface, but this is not the case when the movement is different (see Figure 4.6.b).

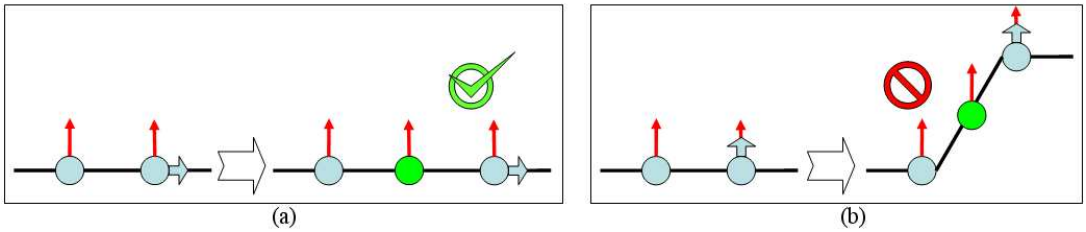

Figure 4.6: (a) The normal of a spawned point (green) can be computed by linearly interpolating the normal of the spawning points if the displacement of the spawning point is perpendicular to the resulting normal (a), but the normal is wrongly computed if the displacement is at any other angle.

I compute the new point's normal with the plane between the spawning points. First, I define plane P_l (Figure 4.7) whose normal is the difference between the spawning points positions, $p_{\text{diff}} = \text{norm}(p_j - p_i)$, where $\text{norm}(p_k)$ is p_k normalized. Next, I compute normal N_{le} by linearly interpolating the spawning points' normals. Finally, the normal of the new point is computed by projecting N_{le} on the plane P_l, i.e., $N_{new} = \text{norm}(N_{\text{le}} - c * p_{\text{diff}})$ where $c = N_{\text{le}} \cdot p_{\text{diff}}$. I consider my normal computation to actually reflect the transition of the surface normal for the points displacements since it reflects the normal of the local plane, which is not the case for the linear interpolation of the point's normals.

4.1.3 Surface Resolution

The resolution of the surface is not limited to the grid resolution. That is, the surface can have details at a smaller scale than the grid resolution, as illustrated in Figure 4.8.

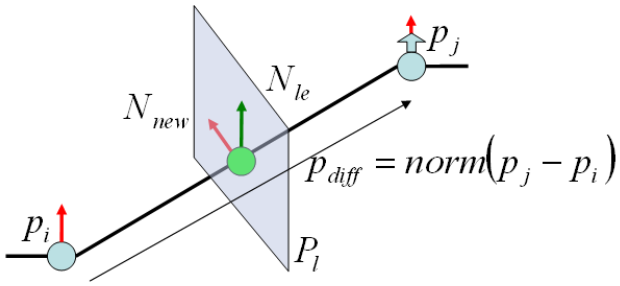

Figure 4.7: The proper normal of a spawned point (green) is computed as the projection of the linear interpolation of the normals of the spawning points on the plane equidistant to the spawning points.

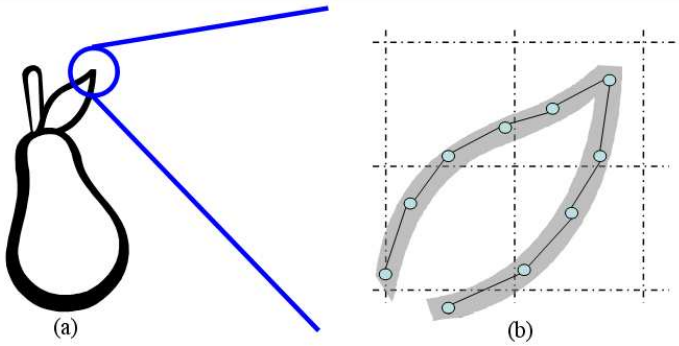

Figure 4.8: The surface resolution is not the grid resolution. (a) Intended shape to represent. (b) Detail of the shape; note that detail can be represented at a smaller resolution than the grid resolution.

The surface resolution, the minimum detail that can be represented, is defined by the resampling radii rather than by the grid resolution. The resampling minimum radius sets the minimum distance between any two neighboring points; therefore, it is not possible to define a detail with points closer than the resampling minimum distance. Figure 4.9 illustrates this concept. Figure 4.9.a shows the detail to be represented and Figure 4.9.b shows the detail using Volipoc. However, it is not possible to represent this detail if the resampling minimum radius r_{min} forces a fusion between points, as shown in Figure 4.9.c. On the contrary, if r_{min} is such that no points fuse together then the surface detail is represented without any problem, as shown in Figure 4.9.d.

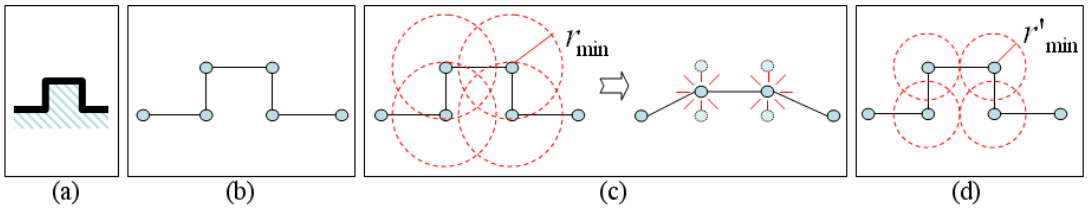

Figure 4.9: Surface resolution is limited by the resampling minimum radius and not by the grid resolution. (a) Surface detail. (b) Detail represented with Volipoc. (c) The detail can not be represented if the resampling minimum radius r_{min} produce a fusion between points. (d) Details can be easily represented as long as they do not require points to be closer than the resampling minimum radius r'_{min}.

Surface details may be lost if the surface shrinks too much. That is, the shape of a surface may change if its deformation fuses points. The loss of detail occurs when sharp edges are lost on the surface. In other words, details are lost when high-frequency surface changes are removed due to the fusion of points. Figure 4.10 illustrates this concept. Figure 4.10.a shows an initial shrinking surface, with velocities toward its geometric center. Figure 4.10.b shows the same surface with points (dark blue) resulting from the fusion of the initial surface, these points are now assigned velocities to expand the surface. Finally, Figure 4.10.c shows the resulting expanded surface, whose shape is not the same as the original one due to the loss of details during the shrinking process.

The loss of detail only happens in shrinking deformations; loss of detail does not happen during expanding deformations. Loss of detail should not occur in practical use of the system, since the deformations I expect from a user/designer are mostly growing deformations.

My scheme is designed with two main user/designer behaviors in mind: a user will start from a basic model and the user designs a model based on its current state. This is similar to the Constructive Solid Geometry approach, which requires the generation of very simple basic models from which complex models are obtained. These result from using models as operands of boolean operators.

Expanding deformations do not eliminate existing surface details. Since the expanding deformations trigger resampling operations on the surface, no point would be removed from the surface (no

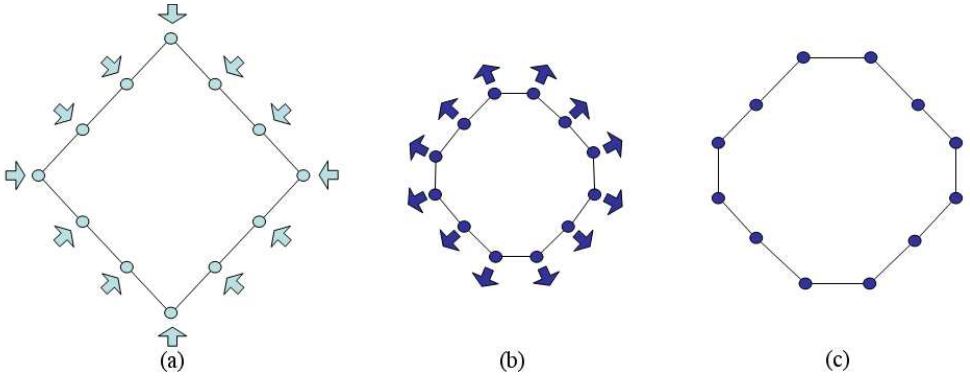

Figure 4.10: Surface detail loss: (a) Shrinking initial surface, (b) Surface after fusion of points (velocities are changed so the surface will expand), and (c) Final surface with details lost.

detail would be lost). The only problem of expanding operations would be the oversampling of the shape; that is, a model is represented with more points than needed. This issue is not important for two main reasons:

- Early decimation to reduce oversampling the surface may be a wasted effort. Since the deformation results from a user/designer's creativity, the surface decimation would be useless if the decimation would be reversed when a new deformation is applied to the decimated zone. Additionally, not having a decimated surface matches the goal of always presenting a homogeneous surface to confidently apply deformations.

- Decimation of the surface does not contribute to the automatic model creation. The goal of my research is the automatic creation of models. The optimization of the surface representation is not within the scope of my research. That is, obtaining a better representation (using less resources) to represent the models is not critical for the automation of the model generation process.

Lost details can be retrieved if the resampling/fusion operations are recorded. With the recorded operations, one can easily roll back all the resampling/fusion operations, obtaining the original surface. This is not addressed because the expected deformations are mostly expanding deformations. Therefore, I do not expect deformation that would eliminate surface details.

4.2 Surface Evolution Control

In nature the evolution of a surface is controlled with a set of different mechanisms that reflect the nature of the phenomenon. For example, chemical compounds can control the replication of cells while other compounds control the specialization of cells defining individual organs. However, for

the goal of Automatic Model Creation, I was not interested in creating an accurate simulation of a natural process, but I need a control of the surface that avoids the complications of deforming surfaces. The complication of topology changes is addressed in this section.

The surface evolution control specifies how the surface moves in space. It avoids surface intersections and surface blends, preventing topology changes. The control uses the same space partitioning structure that is part of the surface representation (see Chapter 3).

The surface evolution control restricts the points' displacements using the state of adjacent cells. A point is allowed to move to an adjacent cell if one of two conditions is satisfied: the cell is empty, or the cell contains a point linked to a point in the original cell. The last condition means that an edge between a point in the same cell as the moving point and a point in the destination cell has to exist in LS. For example, imagine a point A residing within cell number 1 whose velocity makes it move into cell 2: the movement is valid only if at least one edge between points in cell 1 and 2 already exists. Figure 4.11 shows the cases when a point is stopped as part of the surface evolution control.

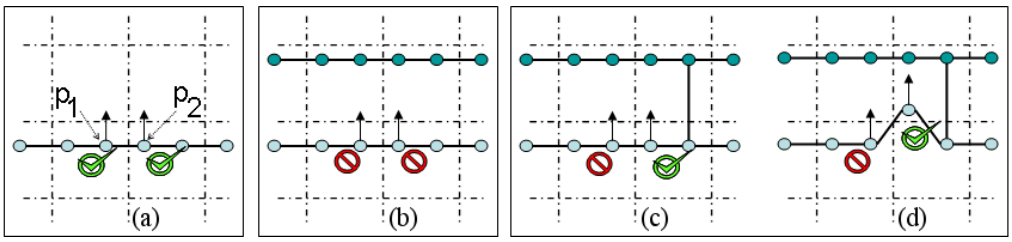

Figure 4.11: Surface evolution control: (a) Point P1 and Point P2 are allowed to move to the adjacent cells because they are empty, (b) P1 and P2 cannot move because the destination cells are occupied by points with no link to the cells of P1 and P2, (c) P2 is allowed to enter the target cell because of a link between cells, and (d) P2 continues to move within the cell because it already resides in it.

My surface evolution control avoids any blending of surface segments that would change the surface topology. The simple rules that control the surface evolution reflect the two main principles to avoid the topology changes: avoid the fusion between points that are not already linked together, and avoid the generation of holes on the surface. Existing holes may be eliminated. Note that topology will not change as long as the initial surface does not contain holes already. For example, an initial surfaces like a torus may undergo topology changes.

The first principle avoids the blend between surface segments. Another way to appreciate this principle is by interpreting the surface as a graph. In such interpretation, the points would be nodes and the volatile edges would be links between points. Therefore, the principle would consist of avoiding linking nodes that do not already share a neighbor. Linking nodes that are not neighbors is equivalent to having a segment of the surface blend with another one. Figures 4.12 and 4.13 show

an example of the Surface Evolution Control first principle.

Figure 4.12.a shows an initial surface where the red arrows are the orientation of the points and the clear-blue thick arrows are their velocities. The initial surface shows a case similar to that of Figure 4.2 where the surface changes its topology. However, my Surface Evolution Control prevents the topology from changing. Figure 4.2.b shows the same surface after a timestep: the points with velocities have moved. Figures 4.2.c-d show the state of the surface after two more steps; some of the points have spawned new points (green) because of the resampling and some fused into new points (dark-blue). Recall that the resampling mechanism was explained in detail in Section 4.1.

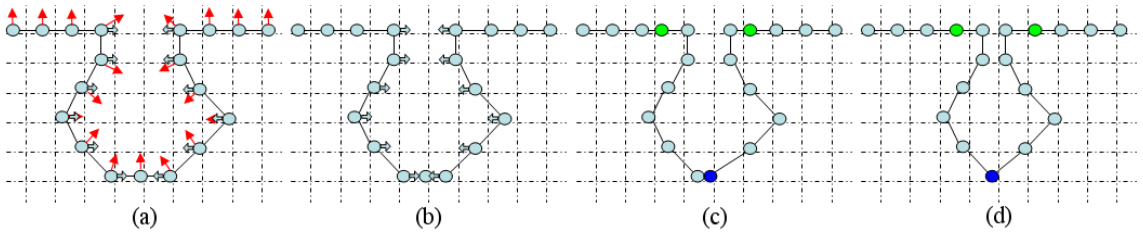

Figure 4.12: Example of a surface with potential changes in topology. The Surface Evolution Control avoids the changes.

Figure 4.13 shows the continuing deformation of the surface from Figure 4.12.d. In Figure 4.13.a the points between the resampling-spawned points (green) and their neighbors have stopped moving because of the Surface Evolution Control mechanism. The same happens to the points that subsequently approach each other, as shown in Figure 4.13.b-d.

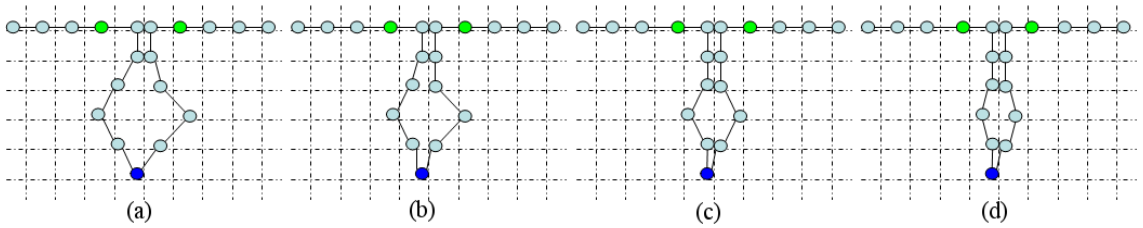

Figure 4.13: Succesive steps of Figure 4.12.

All points end stopping their approach avoiding a blend of the surface that would change the initial topology (see Figure 4.14). Note that, despite the resampling scheme the Surface Evolution Control prevents the surface from blending; therefore, the initial topology is preserved. Figures 4.12 to 4.14 show how the Surface Evolution Control avoids the creation of holes on the surface.

The number of comparisons required for the Surface Evolution Control is small compared to the number of surface points. As explained above, the Surface Evolution Control consists of allowing the movement of one point from its current grid cell to an adjacent one. The movement depends on

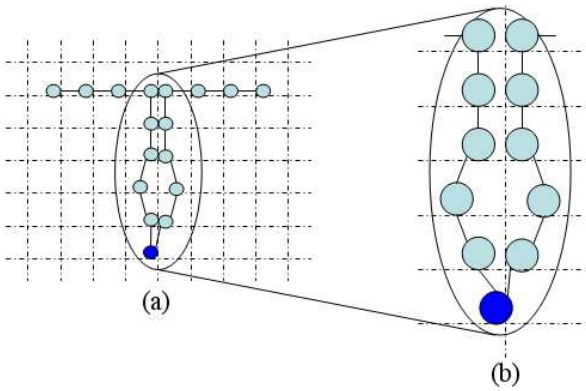

Figure 4.14: Close up of Figure 4.13.d. Note that the Surface Evolution Control has prevented the change in topology by stopping points that would resample with non-neighboring points.

the occupancy of the adjacent cell and the links between points in the cells. This operation requires the computer to compare points in adjacent grid cells; the number of comparisons is bounded by the number of points in the adjacent cells. This number is expected to be a lot smaller than the total amount of points in the surface.

The computation of my Surface Evolution Control is bounded by the maximum number of points in a cell n_M. Therefore, the maximum number of comparison would be ${n_M}^2$ (worst case scenario). On the other side, the best case scenario is when one of the cells contains no points, in which case no comparison is done.

The maximum number of points in a cell, n_M, is determined by the resampling radii. These radii, as explained in Section 4.1, determine the distance range at which resampling is triggered: in other words, how close and how far two neighboring points can be. The radii I used (0.5 and 1.0) define the maximum number of points in a cell. Independently of their particular values, the maximum number of points in a cell will always be bounded by a constant. Therefore, the number of comparisons done for deforming the surface with the grid partitioning will always be much smaller (constant bounded) than without the grid (square).

The constraints on surface evolution prevent the surface from self-intersecting or blending, but introduce a minor visual artifact: a point may stop at an arbitrary distance from the cell boundary. The points will stop at random distances because the point's velocity may make it move into the next cell at any moment. Which may trigger the condition that stops the point. Figure 4.15 illustrates the visual artifact phenomenon. Figure 4.15.a shows segments of a surface that are moving toward each other (similar to Figure 4.2) and Figure 4.12.a shows a visual artifact that occurs by arbitrarily stopping points. Figure 4.15.b shows that points were stopped by the Surface Evolution Control because their next position would have triggered a change in topology.

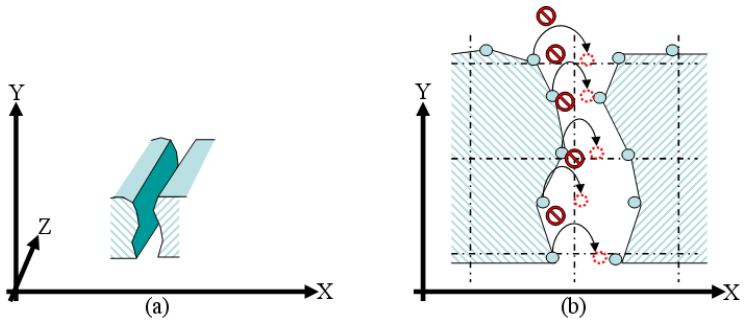

Figure 4.15: Visual artifact illustration: (a) Segments of a surface moving toward each other and (b) Points of the segments stopped because their next position would change the surface topology.

The visual artifact does not harm the Automatic Model Creation goal. I do not expect the visual artifact to appear often because the most of the deformations are going to be designed to create rich visual details and not occluded details. Also, any visual artifact should not correspond to a big percentage of the final model.

The maximum distance at which any non-linked points can stop from each other is in the range (0.0, 1.0], which is insignificant compared to the estimated size of the models. Models are expected to have dimensions of several dozens of grid units, as shown in Chapter 7.

The size of the surface in Vebam's implementation is limited only by the size of the grid. This depends on the available memory. The implementation allows the system to identify and stop any segment of the surface that is going to intersect or blend.

4.3 Summary

The behavior associated with Volipoc defines how the surface changes automatically to fulfill the Automatic Model Creation goals. In particular, my surface behavior provides a surface with quasi-uniform density and prevent topology changes. I achieve quasi-uniform density by resampling the surface. I prevent changes in topology by avoiding the surface blending with itself. This process is also supported by the resampling that prevents the generation of holes.

The resampling mechanism consists of adding or removing new points on the surface when it stretches or compresses. Resampling is triggered only if two points move away from each other and they are linked by a volatile edge. These points are referred to as "spawning points". The resampling process includes removing the volatile edge between spawning points and creating new edges by querying the neighborhood of the spawning points. The properties of new points are computed by linearly interpolating the properties of the points that triggered the resampling; the only exception is the normal property.

I compute the normal of new points with the following approach. Instead of simply using the linear interpolation of the spawning points normals, I project the linearly interpolated normal of the spawning points normals on the plane between the spawning points. The plane is midway to the spawning points and is perpendicular to the edge between spawning points.

The resampling mechanism and Surface Evolution Control allows a user to freely and confidently apply deformations to the surface. This simplifies the work of the user/designer since that person does not need to worry nor interrupt the deformation to take care of the surface. The resampling controls the surface density and prevents the generation of holes. The Surface Evolution Control prevent the surface from blend with itself. The resampling and Surface Evolution Control together prevent the surface to change topology.

Another way to see the resampling is by interpreting the surface representation as a graph representing the surface. In such case, the resampling mechanism guarantees that the graph does not change into disjoint graphs. This simplifies using the surface with other algorithms. In particular, the flooding algorithms that traverse complete graphs to compute/assign properties of the points. Using the graph to traverse the points surface is the approach I used to deform the surface by assigning velocities, as explained in Chapter 5.

Chapter 5

Deformation Specification and Control

In nature, an object's form is the result of the underlying rules of the phenomenon that creates the object. Therefore, the shape of the object can be modified when one manipulates the causes of the phenomenon. For example, the size and number of branches on seaweeds depend on the nutrients and sun incidence where they grow. If one manipulates the nutrients and sun incidence then one can manipulate the shape of the seaweed. Computer Graphics has had a lot of contributions where nature has been the inspiration to create models. Computer Graphics researchers often use the rules (or a simplified version) of natural phenomena to reproduce models.

My research aims at the Automatic Model Creation of organic-like models in general; therefore, I have not bound myself to replicate a particular natural phenomenon. Instead, I required a mechanism capable of controlling the deformation of an object so it can deform into a desired shape.

The particular problems to achieve Automatic Model Creation for Volipoc (Chapter 3) and its behavior (Chapter 4) are:

- Assignment of velocities to surfaces points.
- Controlling the velocities throughout the different creation stages.

I assigned velocities to surface points with Deformation Operators. Deformation Operators are a set of parameterizable procedures that select points from the surface and assign them velocities. I classify deformation operators into two types: Selection Operators and Velocity Operators. Selection Operators are procedures that select a subset of points from the surface. Velocity Operators are procedures that assign a velocity to selected points. For simplicity, sometimes Selection Operators and Velocity Operators can be implemented as a single deformation operator. Section 5.1 explains in detail how deformations are specified by using Selection and Velocity Operators.

Deformation Operators are not enough to automatically create models, especially complex ones. Deformation Operators are best suited to creating individual details on the surface. Designing the selection of points and velocities assignment for one particular detail is straightforward. However, designing the selection of points and velocities assignment for a more complex deformation is not an easy task.

I devised another abstraction layer of the Automatic Model Creation scheme to facilitate the task of selecting surface points and assigning them velocities: this layer is called the Operator Management layer. Operator Management stands above the Deformation Operators (see Figure 5.1). I designed the layer to use the Deformation Operators to create complex organic-like models by composing particular details. The layer consists of a finite state machine where states and transitions reflect the model creation stages.

I think having conditioned generation of content reflects not only the trend of structural model generation (such as L-systems), but it also reflects what happens in nature. One can appreciate the creation of models in nature as a staged process; for example, cactus have complex surfaces where thorns are repeated along the surfaces. Section 5.2 describes in detail how I use a finite state machine, in particular a Petri Net, to apply deformations to the surface; the deformations are applied according to the state of the Petri Net.

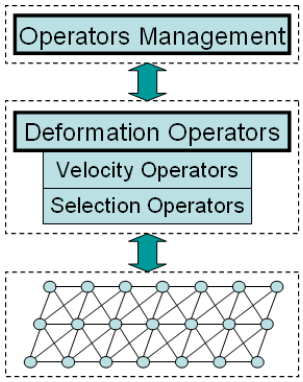

Figure 5.1: Layers of my Automatic Model Creation scheme: First, abstraction of the surface (surface representation), second, Deformation Operators that I use to deform the surface, and third, Management of Deformation Operators so that complex models are created by composing simple deformations.

The velocities are assigned with two types of operators: selection and velocity. I also provide simple examples of these types of operators to illustrate their functionality. Additionally, I present other more complex operators that generate rich details on a surface. Finally, I explain how the model generation process involves programming the application of operators in a finite state machine.

5.1 Deformation Specification

The question of how to assign velocities to a model's surface is answered in this Section. I have mentioned already that a model deforms in time because of the velocities associated with the surfaces' points. The velocities are assigned to points with the help of two types of operators:

Selection and Velocity. The Selection Operators implement algorithms to choose points from the model. The Velocity Operators compute the magnitude and direction of the velocities. Velocity Operators are used to individually assign velocities instead of having a user/designer assign each of them manually. The Selection Operators and Velocity Operators are used to specify deformations at local and global levels.

Separation between Selection and Velocity Operators facilitates the creation of Deformation Operators. Such separation has two advantages. First, it allows us to focus on the specific task. Second, it provides more options for varied deformations by combining different Selection Operators and Velocity Operators. However, more experienced users/designers may be interested in joining the selection of points and velocity assignment.

The effect of Selection and Velocity Operators is reflected between deformation steps. As mentioned earlier, the surface deforms over time. However, the effects of the Selection Operator and Velocity Operator happen immediately when the operator is applied. The deformation of the surface occurs when the points' position is updated every simulation step.

5.1.1 Selection Operators

Selection Operators choose a subset of points from the model. An algorithm implemented in the operator selects the points. The algorithm can query geometric properties of the model's surface: top-most point, 2-neighbors of a point, or center of mass, for example. Figure 5.2 shows an example of a Selection Operator that selects points around an initial point; the points are selected if their projection onto the plane defined by the initial point (with its position and orientation) is closer than 1.5 units to the initial point.

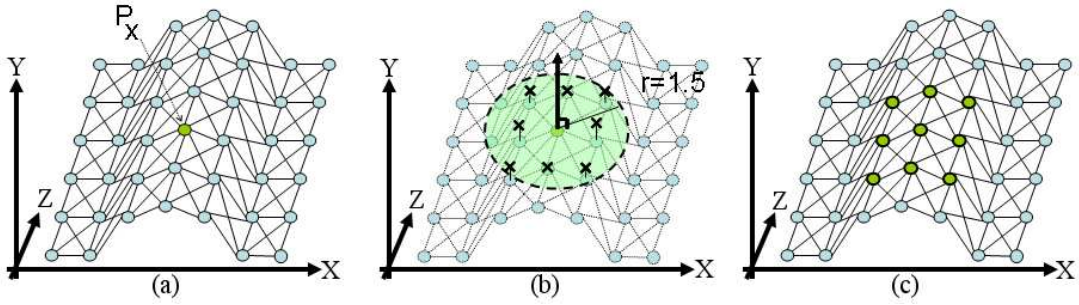

Figure 5.2: Selection Operator example: (a) Initial surface and point P_x (operator's initial point), (b) Projection of near points on the plane defined by P_x's position and orientation, and (c) Selected points (in green) project on the plane at a distance less or equal than r=1.5 units.

Selection Operators in Vebam are directly implemented in Vebam's source code. As stated previously, Vebam is a software tool that demonstrates the utility of the specifications of the

Automatic Model Creation scheme of this project. Therefore, Vebam is not designed to support the creation of the operators by a user. Instead, it is designed to demonstrate their utility for Automatic Model Creation. Next, I give a detailed example of a Selection Operator. The operator selects regions of a model's surface. The selected regions follow a Voronoi-like distribution on the surface; hence the name of the operator: "Voronoi Regionalization".

VoronoiRegionalization(m,t) labels all of a model's points as belonging to one of m possible sets. The sets are the equivalent of Voronoi-like regions on the model's surface. The labeling is done with a variant of Lloyd's algorithm [48], which inserts and removes points from the regions depending on their proximity to the region's center. This process is repeated t times. I have used Lloyd's Algorithm to show how a Selection Operator can easily be used with my Volipoc for selecting points.

I denote the ith point of the model's surface by p_i. The principal internal data structure is the surface representation of the model; that is, the whole set of n points linked by volatile edges, which can be seen as a graph representation of the surface. I denote that point p_i belongs to region j by $r_i = j$, $1 \leq j \leq m$. Each region has a centroid c_j, which is the average position of points in region j.

The result of this algorithm is the labeling of all the model's points. The resulting regions are equivalent to Voronoi regions on the surface. This labeling can later be used as the input for another operator. The following prosecode [108] describes how **VoronoiRegionalization**(m,t) can be implemented.

1. (Initialize region centers) Randomly select m different points as the initial region centers, i.e., $c_1 = 1, c_2 = 2, \ldots c_m = m$

2. (Clear labels) Clear all labels so that no point is assigned to any region. Set $r_x \leftarrow 0$, $x = [1, \ldots, m]$. Note that the regions are counted from 1 to m, considering 0 as a non-existing region or a null label.

3. (Label first point) Find the closest point to center of region j, $r_x \leftarrow j$ if GeometricDistance(c_j,p_j)= smaller GeometricDistance(c_j,p_x), $\forall j \leq m$.

4. (Grow regions) Label as j all the points that are linked to a point in region j and that have a null label. For all j, $r_x \leftarrow j$ if $r_x = 0$ and $r_o = j \mid p_x$ and p_o are linked by a volatile edge.

5. (Repeat) Repeat step 4 until all points have been assigned to a region.

6. (Compute region centers) Compute the geometric center of all regions by computing the average sum of the position with the same level, $c_j \leftarrow \sum p_m \mid r_m = j$

7. (Repeat) Repeat steps 2 through 6 t times.

Figure 5.3 shows an example of **VoronoiRegionalization**(*7,10*). The initial surface is a flat square of 2500 points. The surface is rendered with oriented circles, one per point and with same direction (see Figure 5.3.l). Each color indicates a Voronoi region; if the region is null then the color is black. The inverse-colored points close to the center of the regions are the points closer to the region centroid. Figure 5.3.a shows the initial surface with seven random region seeds. Figure 5.3.b to Figure 5.3.k shows the evolution of the points labeling as the algorithm executes for ten passes.

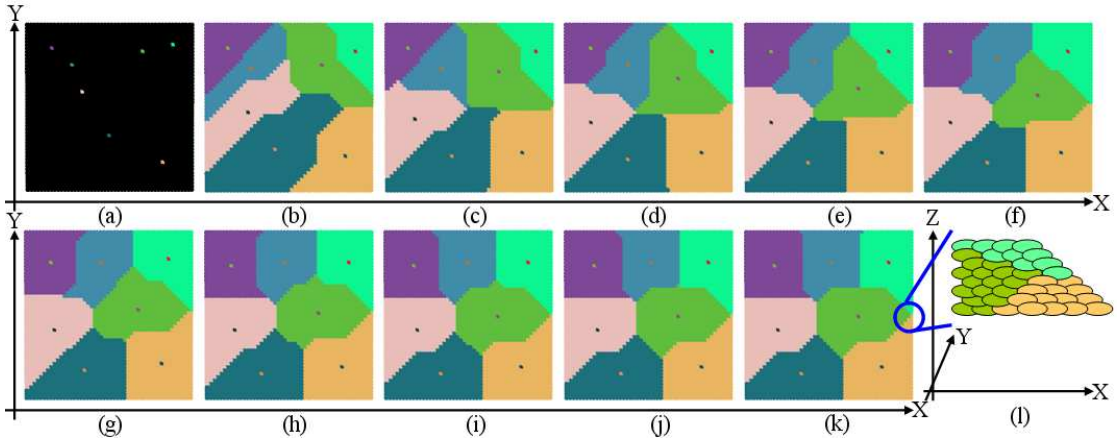

Figure 5.3: Example of **VoronoiRegionalization**(*7,10*).

Selection Operators have the advantage of isolating the task at hand: selecting points from the surface. The results (selected points) can later be used by other types of operators to assign velocities to the points. Isolating the task makes it simpler for a user to focus on only one type of task. However, Deformation Operators can be designed to select and assign velocities in the same operation, Section 5.1.3 contains examples of such combined operators.

Selection Operators can use any type of abstraction to select points. In the previous example I specified the Selection Operator in the most basic and direct way, that is, I used the surface representation primitives (edges and points) and their properties. However, any other information could be used instead, whether geometry, color, or anything else.

From the implementation point of view Selection Operators are implemented directly in Vebam code. Vebam's reusability could be improved by adding support to create deformation operators without recompiling the whole code, although it would not provide any advantage from the point of view of demonstrating the utility of my Automatic Model Creation framework, which is one of my goals. Similar tools in the field (e.g. Pointshop3D [109]) have improved reusability by supporting plugin creation.

5.1.2 Velocity Operators

Deforming a surface requires more than just selecting points: their velocities have to be assigned. Velocity Operators assign velocities to selected points in a procedural way. Velocity Operators are designed to assign velocities to a group of selected points; the velocities are computed to deform the surface according to a user/designed wishes. Usually, Velocity Operators will only require a single point in addition to the parameters of the velocities-computing procedure.

Velocity Operators compute a velocity (magnitude and direction), and assign it to points. If necessary, Velocity Operators can query any geometric information of the surface. For example, an operator could compute a point's velocity based on the distance of the point and the surface's centroid. Figure 5.4 shows an example of a Velocity Operator.

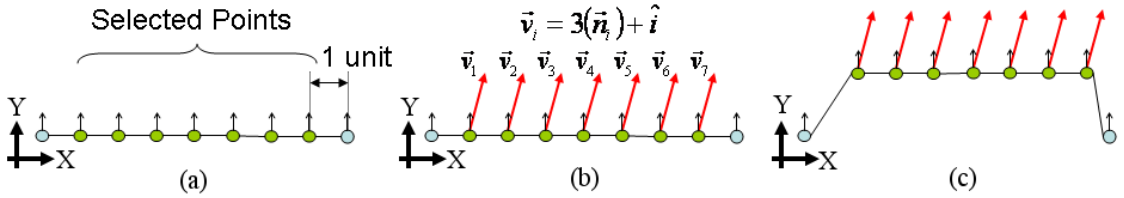

Figure 5.4: Velocity Operator example: (a) Initial surface showing the links and normals of points, (b) Velocities computation of the selected points with $\vec{v}_i = 3(\vec{n}_i + \hat{i})$ where $\vec{n}_i$ is the normal of point i, and (c) The displacement experienced by the points due to their new velocity.

Next, I describe an example of a Velocity Operator named ScaledUnitaryGaussian. The operator computes velocities and assigns them to points in a region. The velocities generate a Bump-like deformation on the surface, where the bump is shaped like a Gaussian.

ScaledUnitaryGaussian(p,σ, d) computes velocities for a set of points centered at point p. The points must be linked to point p and closer than d units. The velocities' intensities drop off following a Gaussian-like distribution of the form $e^{-(\frac{\text{dist}(p,p_i)^2}{\sigma^2})}$, where σ is a standard deviation and $\text{dist}(p, p_i)$ is the distance between p and the point under evaluation p_i.

The principal data structure is the surface representation of the model; that is, the whole set of n points linked by volatile edges (see Chapter 3), which can be seen as a graph representation of the surface. An additional data structure is a queue that holds references to points. I denote the points in the front and back of the queue by p_f and p_b respectively. I denote the point being processed (whose velocity is being computed) by p_p and its neighbors as p_n. Finally, I denote the velocity of point i by v_i.

The result of this algorithm is the velocities assigned to the points closer than d units to point p. The points must be adjacent by point p and all points within the path must be closer than d units to point p, as shown in Figure 5.5.

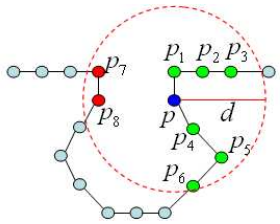

Figure 5.5: Closer and reachable points. Points $p_1 \ldots p_8$ are all closer than d units to point p, but only $p_1 \ldots p_6$ are either directly reachable from point p without traversing points that are not within d unitos form p.

The following prosecode describes how **ScaledUnitaryGaussian**(p,σ, d) can be implemented.

1. (Initialize queue) Push point p in front of LILO queue.

2. (Get next processing point) Pop point from the back of queue and use it as the processing point, $p_p \leftarrow p_b$.

3. (Compute velocity) Compute the velocity of the processing point as a Gaussian drop off from point p and the standard deviation parameter (σ), $v_p \leftarrow n_p \cdot e^{-\frac{dist(p,p_p)^2}{2 \cdot \sigma^2}}$, where $dist(p, p_p)$ is the distance between point p and the processing point p_p; note that this is not the operator's parameter d. The resulting velocity is in the same direction as the point's normal magnitude.

4. (Select next points) Push in front of the queue the points linked (with a volatile edge) to point p if they satisfy all the following conditions:

   - The points have not been processed previously. This condition allows the loop to terminate.
   - The distance between p and its neighbor p_n is less than d units, that is, if $dist(p, p_n) < d$. This condition guarantees that the operator only modifies the velocities of points closer than d units to point p.

5. (Repeat) Repeat steps 2-4 until queue is empty. This cycle repeats until all points that are closer than d units to point p and are adjacent have received a velocity.

Figure 5.6 is an example of **ScaledUnitaryGaussian**(p, 2.25, *8.0*). Figure 5.6.a shows the initial surface consisting of 1600 points distributed over 20X20 grid cells; the coloration is only to help appreciate the surface deformation. Figure 5.6.b shows the effects of applying **ScaledUnitaryGaussian**(p, 2.25, *8.0*) where p is the surface center. I modified the operator so that it changes the points' colors according to their velocity; the color intensity is directly proportional to the velocity. Figure 5.6.c shows the surface after three seconds of simulation, and Figure 5.6.d

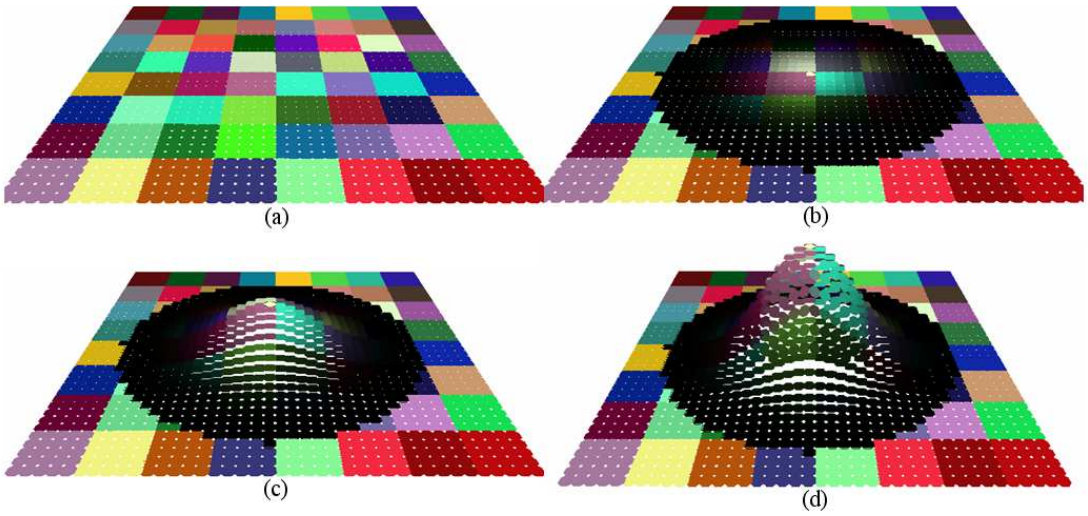

Figure 5.6: Example of **ScaledUnitaryGaussian**(p, σ, d) .

shows the result after six seconds; in the last case, resampling has started to create new points due to surface expansion, as described in Chapter 4.

ScaledUnitaryGaussian(p,σ, d) is an example of how Velocity Operators can be used to assign velocities to surface points and deform the surface. Velocity Operators can be used to create content; the variety of content will depend on the Velocity Operator design, where a more parameterizable design leads to a higher variety.

Simple Velocity Operators and Selection Operators can be combined to generate complex models. For example, I combined **ScaledUnitaryGaussian** and **VoronoiRegionalization** to create bumps on a surface. I used the center of the Voronoi region as point p and limited the velocity modification to points of the same Voronoi region as p. To do this, I simply included a condition to verify if the point is labeled with the same region as the region center.

Figure 5.7 shows the result of combining **ScaledUnitaryGaussian** and **VoronoiRegionalization**. Figure 5.7.a shows the labeling obtained from applying **VoronoiRegionalization**(*17*,*6*). Note that the operator made six passes of the algorithm; hence, some of the regions are concave. If the operator would have made more passes then the regions would tend to be convex. I used the operator with only six passes to visualize a surface with concave regions. Figure 5.7.b is the result of applying **ScaledUnitaryGaussian**(p_i,2.25, *8*), i=1...17, where p_i is the center of the ith Voronoi region. Figure 5.7.c and Figure 5.7.d show the surface after evolving two and four seconds respectively.

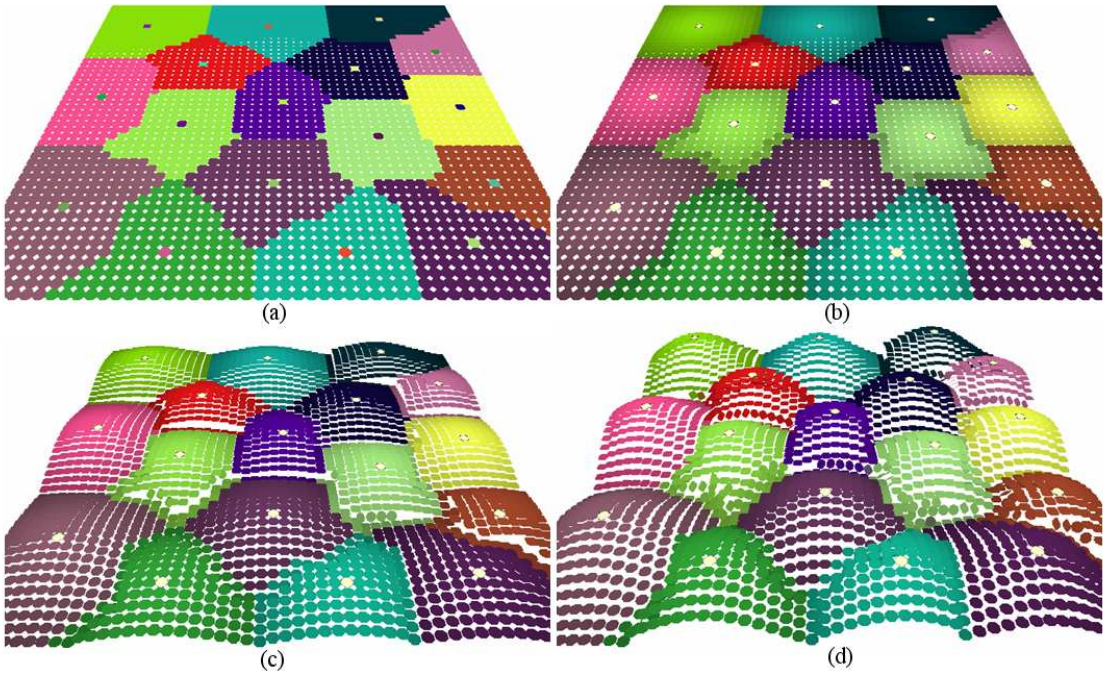

Figure 5.7: Combining simple Selection Operators and Velocity Operators can lead to complex models with minimal user interaction.

5.1.3 Additional Deformation Operators

Next I present additional deformation operators that I created to generate different types of surface details: thorns, craters, and cracks. Additionally, a global operator is included: roughening. This operator roughens an existing surface.

Thorn Operator

The Thorn generator operator will generate thin thorn-like details on the surface. The thorn-like details are defined by their base radius, thinness, trend direction and growing speed, as shown in Fig. 5.8.

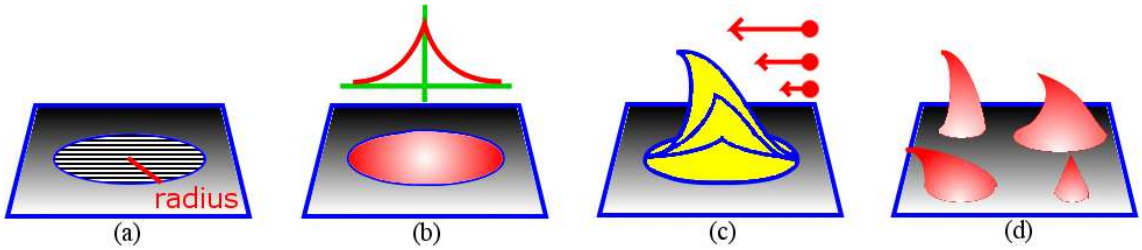

Figure 5.8: Thorn operator and its defining elements.

The thorn's base has a circular shape (Fig. 5.8.a), but it may be bounded by another operator's

result. A similar example is in Chapter 5. In that example, the results of a Voronoi regionalization operator are used to bind Gaussian velocity distribution operators.

The Thorn operator's thinness parameter is used to assign velocities to the surface points in the operator's base region (Fig. 5.8.b). The velocities follow a distribution of the form

$$v(r) = (1 - r)e^{-rt} \tag{5.1}$$

The distribution form is centered in the circular region and is used in a domain between 0 and 1 while its range is between 0 and the region's radius R, i.e. $v(r) : [0, R] \rightarrow [0, 1]$. Figure 5.9 shows the distribution evaluated with $R = 1$ and different values of the thorn's thinness factor t.

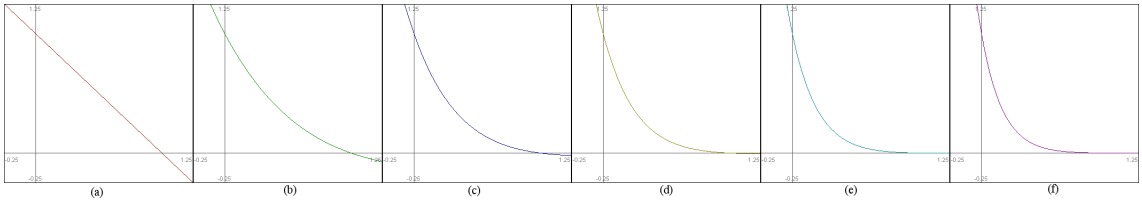

Figure 5.9: Equation 5.1 evaluated with R=1 and: (a) $t = 0$, (b) $t = 1$, (c) $t = 2$, (d) $t = 3$, (e) $t = 4$, (f) $t = 5$.

The trend direction parameter defines the thorn's growing direction. It is implemented as a set of velocities pulling surface points in a certain direction, thus modifying its displacement every simulation step. The pulling velocities increase with the point's distance from its original location. The growing effect I expect to get from these pulling velocities is shown in Fig.5.8.c. Note: techniques to create surface details are well known and straight thorns can be created with such techniques but they cannot create details with concave sections. The Thorn operator described here is able to create straight and curved (with concave sections) thorns because of the trend direction parameter.

The Thorn operator's last parameter, growing speed, allows the scaling of the deforming velocities to obtain different thorn heights. With this parameter and the previous ones (radius, thinness, and trend), the Thorn operator will be capable of creating a great variety of custom thorns; see Fig. 5.8.d for the estimated results.

Figure 5.10 shows the effect of the operator. Figure 5.10.a shows an initial flat surface where different instances of the operators have been applied; the black circles indicate the area of effect of each operator. Figure 5.10.b shows the surface after several seconds of deformation. Figure 5.10.c-d show the surface transformed into polygonal meshes with different textures.

Crater Operator

The Cratering operator assigns velocities to a surface region so that it deforms and creates a crater-like detail; that is, a surface depression surrounded by a wall with varying heights. The

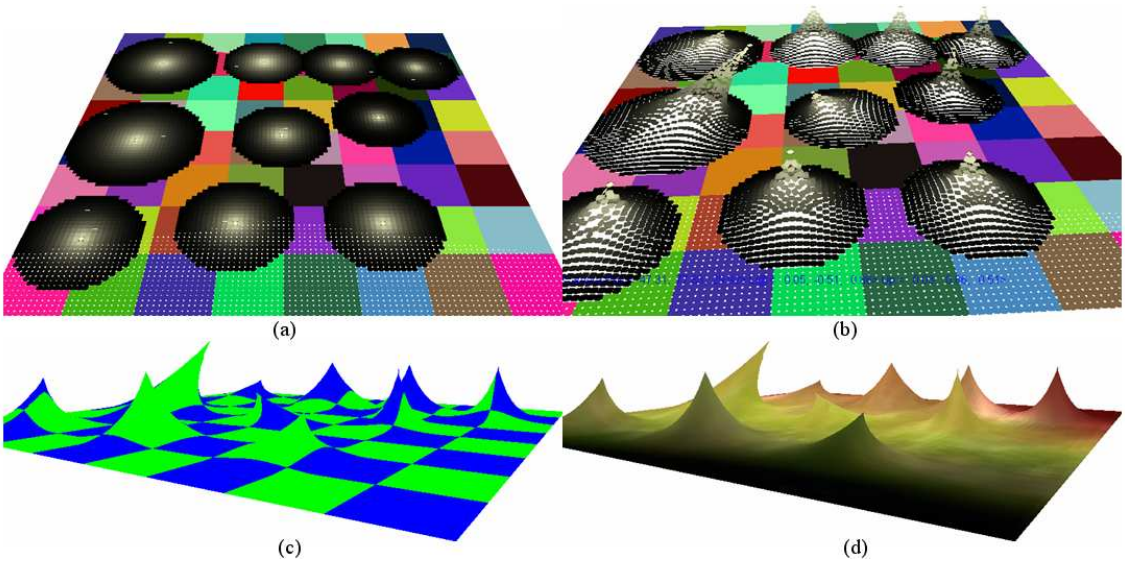

Figure 5.10: Use of the Thorn operator.

operator uses several parameters for different crater features: base radius (R), base roundness, peaks' numbers, and peaks' radiuses. The first parameter, base radius, specifies the radius of the crater. The second parameter, base roundness, indicates how round the region's shape is. The third parameter, peaks' numbers, indicates the number of peaks. The fourth and last parameter, peaks' radiuses, specify the radius per peak, peak generation is explained ahead. Figure 5.11 sketches the Cratering operator and its parameters.

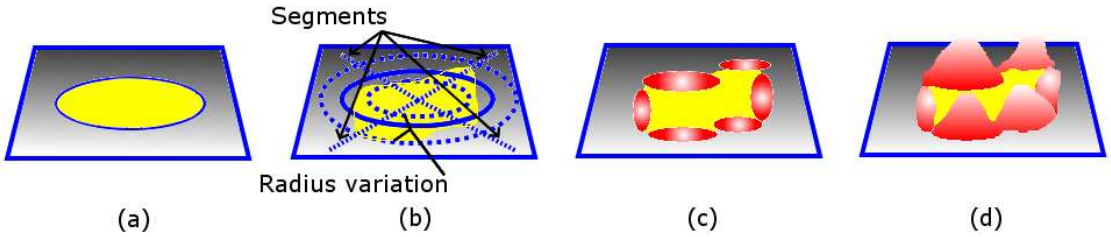

Figure 5.11: Cratering operator parameters: (a) Base radius, (b) Roundness, (c) Peaks. Estimated result (d).

The crater's base roundness describes how much the crater's base follows a circular shape. It is specified by two values: R_v, radius variation, and S, the number of base segments. The radius variation indicates the maximum radius percentage by which base shape can vary. The base segments number specifies the number of inflection points the base's shape will have. To compute the base, the initial circular base is divided in S slices of same size (Fig. 5.12.a). A random distance in the range $\left[R - \left(R \times \frac{R_v}{100}\right), R + \left(R \times \frac{R_v}{100}\right)\right]$ is computed at each of the segment's edges; these distances mark the inflection points (D_i, $i = 1, 2, ..., S$) of the base's final shape (Fig. 5.12). The inflection points are used to evaluate the function $d(a)$ which defines the distance between the

base's edge and its original center (Fig. 5.12.c):

$$d(a) = \left(\left(\frac{\cos\left(\left(\frac{a-a_i}{|a_{i+1}-a_i|} \right) \times \pi + O(i) \right)}{2} \right) \times |D_i - D_{i+1}| \right) + R - \left(R \times \frac{R_v}{100} \right) + D_i \qquad (5.2)$$

$$O(i) = \begin{cases} \pi & \text{if } D_i > D_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

where $a, a_i, a_{i+1} \in [0, 2\pi)$ are the degrees of a point in the circular base and the degrees of the inflection points D_i, D_{i+1} respectively; see Fig. 5.12.b. The indices work in a circular way, i.e. if $i = S$ then $i+1 \leftarrow 1$. Equation 5.2 computes the distance defining the modified crater's base shape; see Fig. 5.12.c and d.

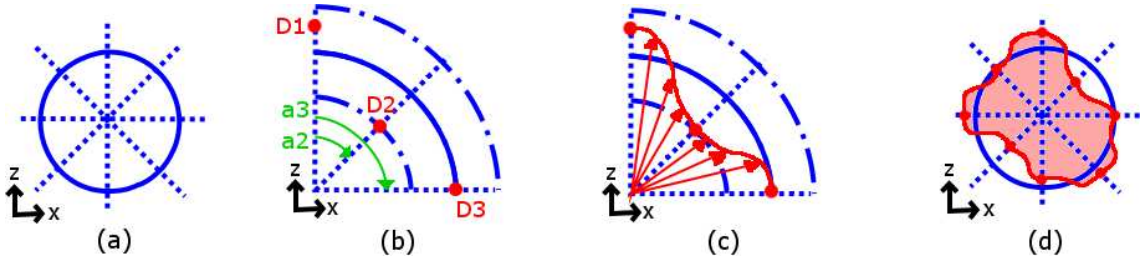

Figure 5.12: Crater's base shape computation.

The crater's wall is created by growing peaks on the crater's base edge. The wall is specified by the number of peaks and their radius, height and scattering balance. The peaks are randomly distributed along the crater's base edge. Each peak consists of a region whose velocities are defined by equation 5.3. If a peak region intersects an already existing region, the points in the intersection are assigned the highest velocity. The cosine form of the equation guarantees a smooth transition between the peak's top and base.

$$v(x,z) = h \times \cos\left(\frac{d_r \times \pi}{r_p + s \times \left(\frac{d_r - (d_c - r_p)}{r_p} \right)} \right) \qquad (5.3)$$

In equation 5.3, h is the peaks height, d_r is the distance of the peak's region point to the crater's circular base center, r_p is the peak's radius, d_c is the distance of the peak's center to the crater's circular base center, and s is a asymmetry parameter (Fig. 5.13.a).

The asymmetry parameter is used to create the effect of having a steeper slope in the peak's region closer to the crater's center, as real crater do (Fig. 5.13.b). Figure 5.14 shows the effect produced by the asymmetry parameter.

With the previously described behavior, the Cratering operator assigns velocities to form a crater-like detail on the surface. The crater will consist of an irregular crown-like formation with randomly distributed peaks, each with a steeper slope on the inner wall, as sketched in Fig. 5.13.c.

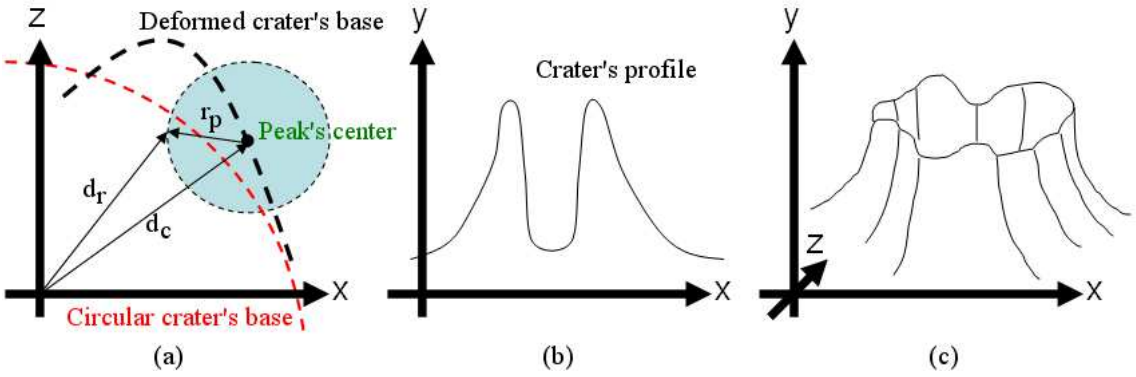

Figure 5.13: Crater wall's peak generation.

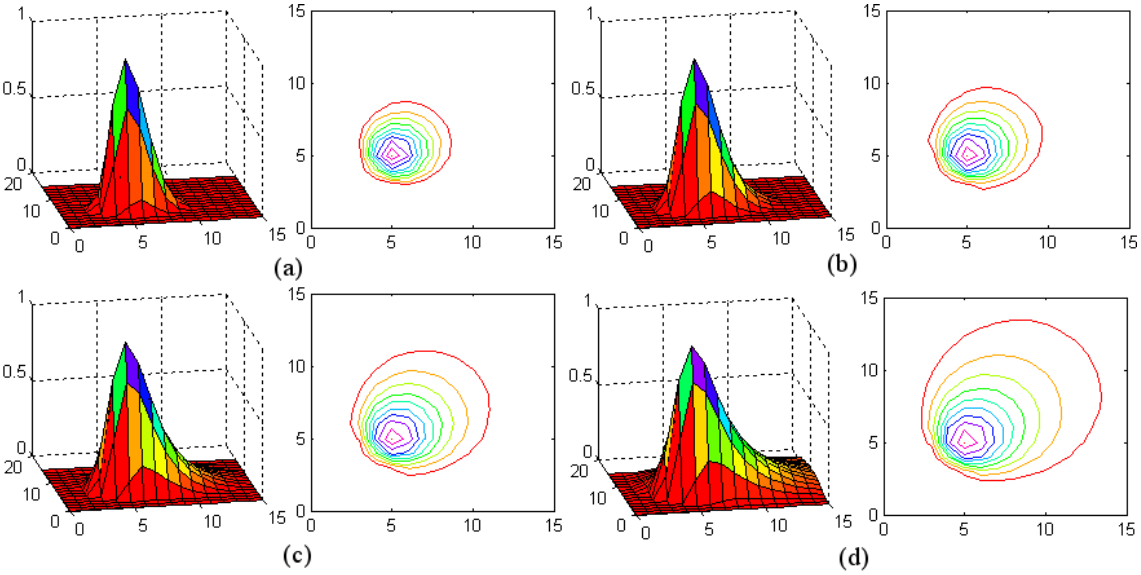

Figure 5.14: Asymmetry effect; peak center is at (5,5), peak radius is 2 and crater's center is (0,0). (a) s = 1.0, (b) s = 1.25, (c) s = 1.5, (d) s = 1.75.

Figure 5.15 shows how the Cratering operator generates a Crater-like surface detail. Figure 5.15.a shows an initial flat surface where an instance of the operator has been applied; the black region indicates the area of effect of each operator. Figure 5.15.b shows the surface after several seconds of deformation. Figure 5.15.c-d show the surface transformed into polygonal meshes with different textures.

Roughening Operator

As its name indicates, the Roughening operator changes a model's surface to one with irregularities, protuberancies and ridges. This operator is based on the idea of adding higher frequency detail to a surface. For example, Fig.5.16.a models a surface ($f(x) = c$) to which some noise ($l(x) = cos(x)$) is added. The resulting surface (Fig.5.16.b) has the original shape plus the irregularities introduced by

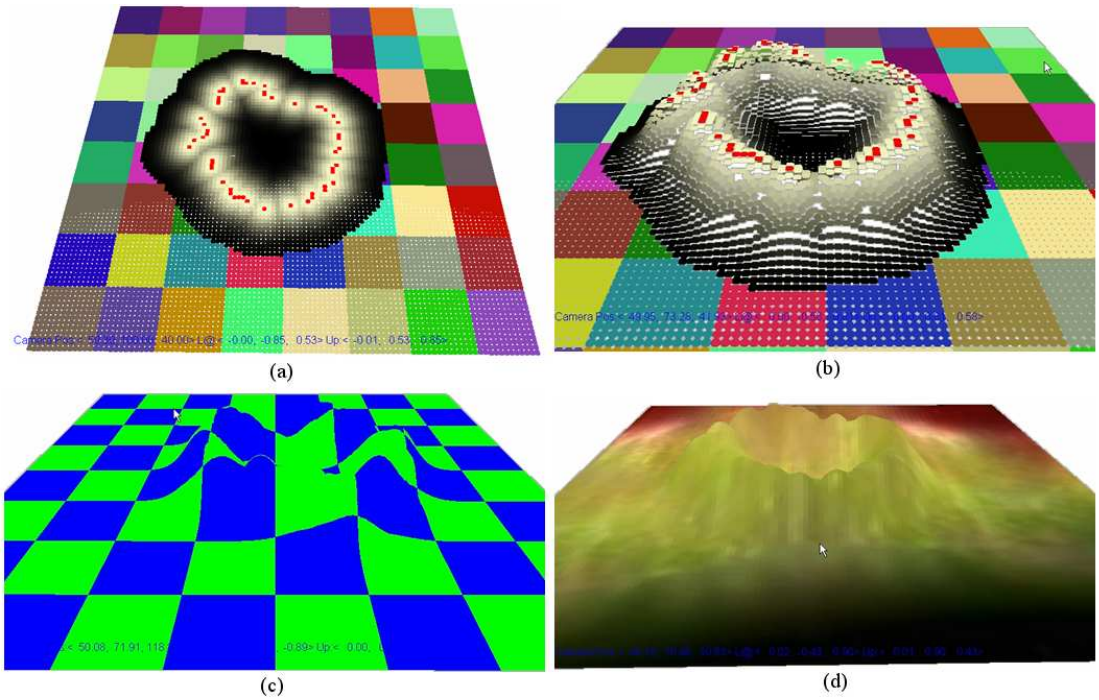

Figure 5.15: Use of the Cratering operator.

the noise. When higher frequency noise is added, the surface presents protuberancies (Fig.5.16.c). Furthermore, adding even higher frequency noise increases the surface roughness (Fig.5.16.d).

The Roughening operator does not add random noise to the surface since this would create a highly irregular surface, which is not the goal of this operator. The operator's goal is to roughen a surface but with smooth transitions between the details. For this purpose, the operator executes three steps of adding noise to the surface. The noise frequency increases at each step.

The *noise* added by the operator is computed with the velocities of cosine-like bumps distributed on the surface. The bumps are distributed following the results of the **VoronoiRegionalization** operator.

The Voronoi regions are created with n_v random seeds. The regions' velocities distribution is different from the examples previously cited; the regions used in the Roughening operator are cosine bumps instead of following a Gaussian velocity distribution. The cosine bumps' velocities are computed as follows:

$$v_i(x,z) = h_i \times \left(\frac{\cos\left(\frac{d_p}{d_b} \times \pi\right)}{2} + 0.5 \right) \tag{5.4}$$

where v_i is the velocity of the surface element, h_i is the maximum height for the bumps at step i, d_p is the distance between the evaluating point and the regions centroid, and d_b is the distance between the regions centroid and its border in the direction of the point under evaluation (Fig.5.17.a). Equation 5.4 guarantees that the normal at the bump's peaks and Voronoi regions

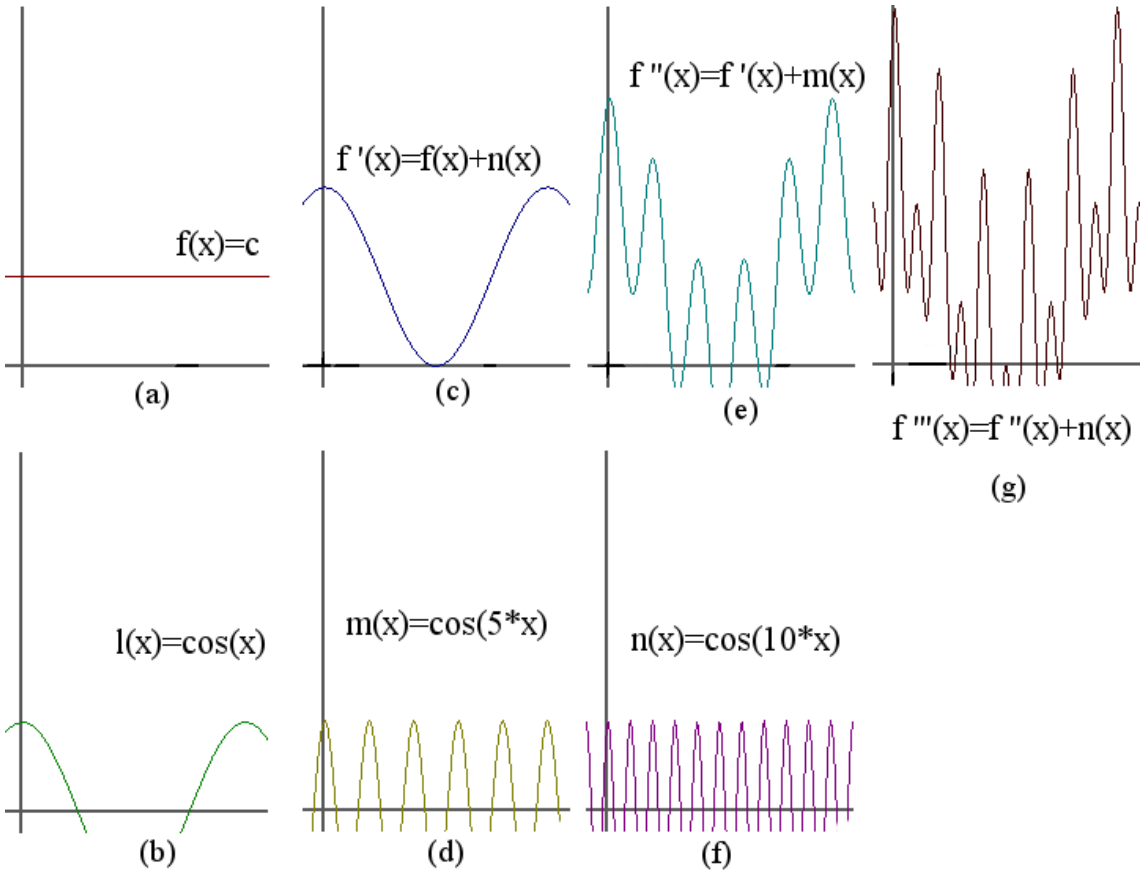

Figure 5.16: Surface roughening by adding high frequency noise.

intersection is perpendicular to the surface, thus guaranteeing a smooth transition between bumps (Fig. 5.17.b); transitions have C^1 continuity. The *smoothness* measure is the same as that defined by Foley et al. [24]: (Geometric continuity) "For two curves to join smoothly, we require only that their tangent-vector match; we do not require that their magnitudes match".

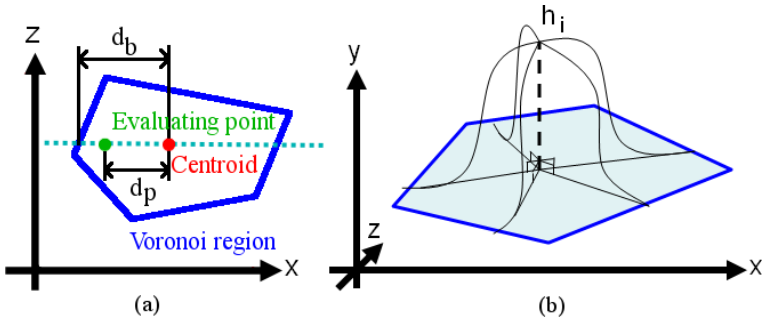

Figure 5.17: Cosine-based bump computation.

The *noise* frequency increases in each of the operator steps. The frequency increment is modeled by increasing the number of Voronoi regions used to generate the bumps. The additional Voronoi regions are defined by a multiplier parameter m_v. For example, for n_v=5 and m_v=3, m_v would be

15 and 45 for the second and third steps respectively. The height h_i also varies from step to step; the variation is proportional to m_v and is given by: $h_{i+1} = \frac{h_i}{m_v}$. Figure 5.18 shows the sketches of a surface for $n_v = 5$ and $m_v = 3$, during step 1, 2, and 3 (Fig. 5.18.a, b, and c)

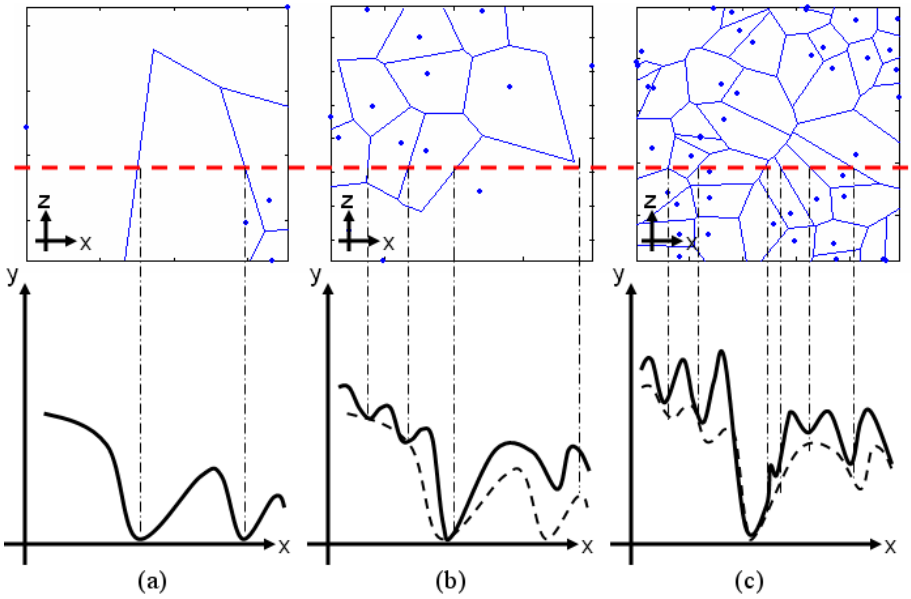

Figure 5.18: Sketches of the Roughening operator steps and its effects.

Summarizing, the Roughening operator parameters are: number of initial Voronoi regions n_v, multiplier m_v, and initial bump's height h_1.

Figure 5.19 shows the effects of the Roughening operator. Figure 5.19.a shows an initial flat surface where an instance of the operator has been applied. Figure 5.19.b shows the surface after several seconds of deformation. Figure 5.19.c show the surface transformed into polygonal meshes with a checkerboard texture.

Crack Operator

The Cracking operator selects surface points in a crack-like pattern. These points are assigned velocities that generate a crack detail on the surface. The operator parameters are: crack seed, cracking threshold $c_t \in [0, 1]$, operator radius c_r, and maximum crack width c_w. The operator consists of four stages: weight assignment, tree creation, crack creation, and crack widening.

The algorithm of the cracking operator is akin to Kruskal's algorithm, that is, it finds a minimum spanning tree for the surface. The surface is interpreted as a weighted graph; points are nodes and volatile edges are assigned a weight. The first stage of the operator, weight assignment, associates a weight in the range $[0, 1]$ with each link between neighboring points in the operator's area. If the link's weight is bigger than the specified threshold, $w_i > c_t$, then that link is a valid path for the final crack (Fig. 5.20.a). The next stage, tree creation, creates a tree path from the seed point

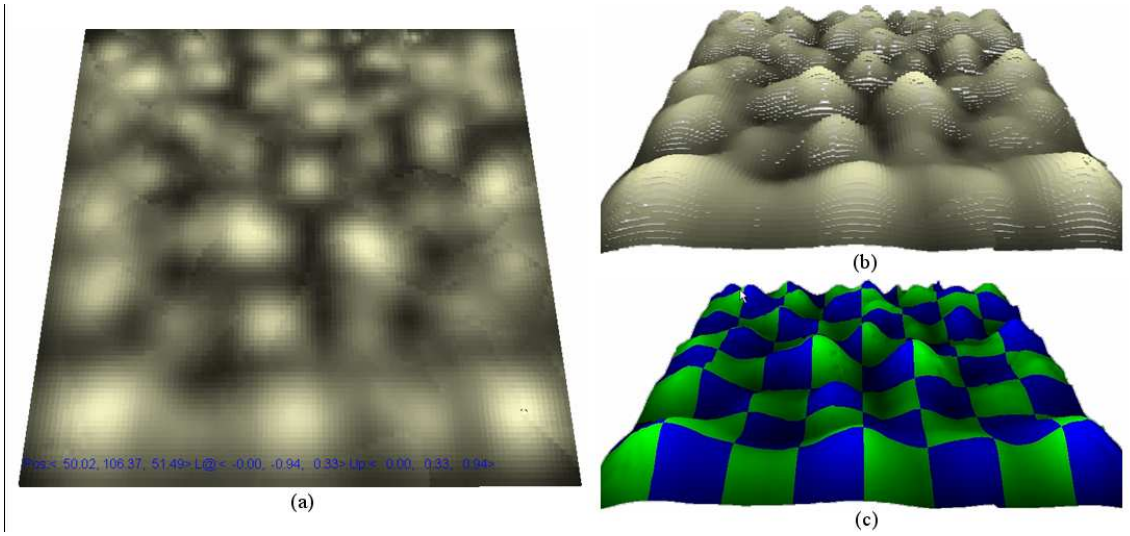

Figure 5.19: Use of the Roughening operator.

by traversing all the neighboring links whose weight is bigger than the threshold. The links are added to the tree if they do not close a loop and if they are adjacent to a point already in the tree, as shown in Fig. 5.20.b. In order to create the final crack structure, the tree is refined in the final stage. The refinement process consists of traversing the tree eliminating certain paths; every time a node with two or more bifurcations is found, the one with less tree depth is discarded. The previous process is shown in Fig.5.20.c. Note: The sketches shown in Fig.5.20 have been made on a rectangular application area; for the proposed operator, the area will be circular.

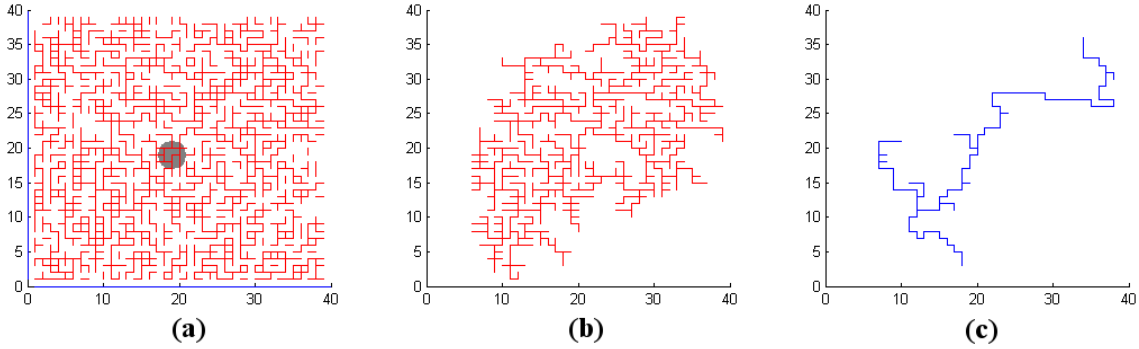

Figure 5.20: Three stages of the cracking operator.

Once the crack path has been created, the crack is widened with a dilation process. The number of dilation operations applied is limited by the parameter c_w, which decreases proportionally according to the distance from the crack segment to its center. After the dilation process has finished, velocities are assigned to the crack's points (Fig.5.21.b); these velocities will deform the surface by collapsing inwardly (contrary to the surface normals), thus simulating a crack on it (Fig. 5.21.c).

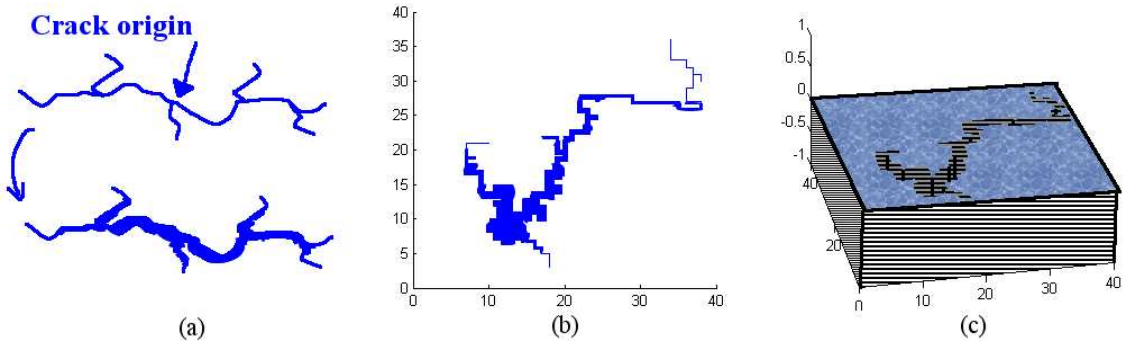

Figure 5.21: (a) Crack widening, (b) Widening of crack in Fig. 5.20.c and (c) Expected effect.

Figure 5.22 shows an example of the Cracking operator. Figure 5.22.a shows an initial flat surface where three instances of the operator have been applied. Figure 5.22.b shows the surface after several seconds of deformation. Figure 5.22.c show the surface transformed into polygonal meshes with different textures.

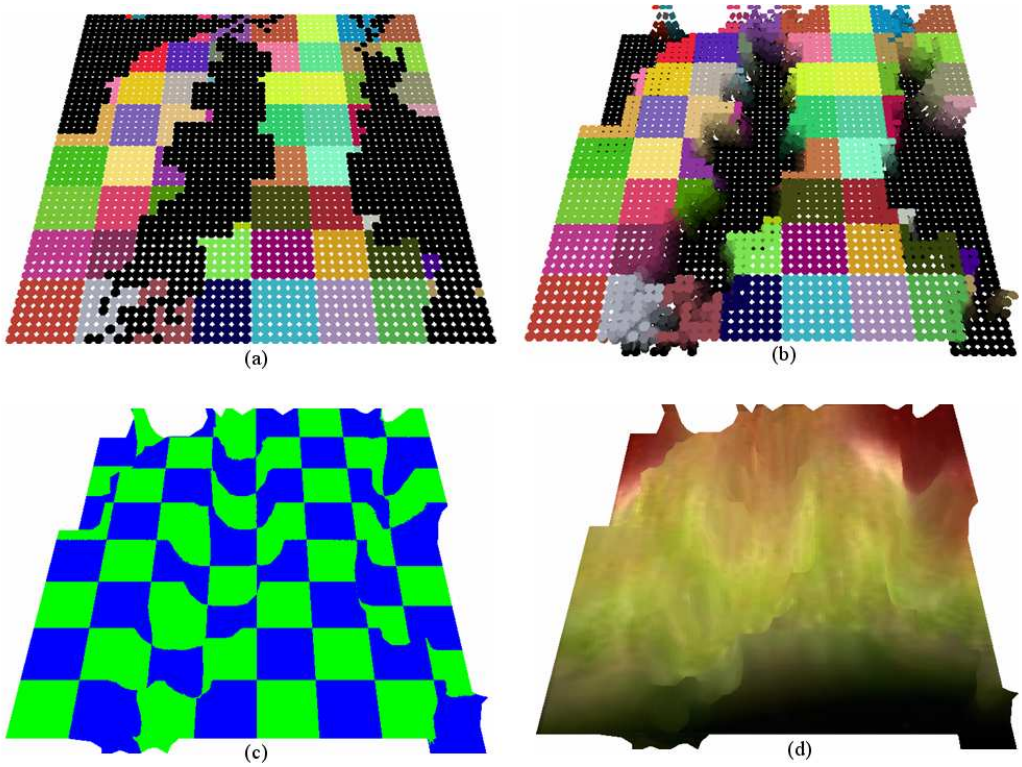

Figure 5.22: Use of the Cracking operator.

I have shown how deformation operators can be used to obtain a great variety of models and reduce the user/designer interaction by procedurally assigning velocities to points. However, complex models may be difficult to create with a single application of operators. For example, a tree model is composed of a trunk, branches, leaves, and roots, each with different surface details. The

task of creating a model with such shape and detail may seem a daunting task. However, if the model is seen at different stages it may not be so difficult to create the model.

Nature suggests guidelines to create models. For the tree model example, some guidelines could be the growing stages of a tree (root deepening, trunk and branches elongating, leaves growing, fruit maturing). Such guidelines show that different parts of the objects are created at different times. For example, the fruit on a tree usually do not mature until the trunk and branches have grown to a certain size. I have followed the idea of creating model details at different stages.

I expanded my Automatic Model Creation scheme so that operators can be applied automatically at different stages of the model creation. I used a finite state machine to control the application of operators. The next section contains the details for controlling the application of deformation operators.

Even though just having a library of operators would be very useful for the creation of different models, a user would still need to apply them individually. I further provided an Operator Application Manager to control which operators are applied and when and where they are applied. The Operator Application Manager is explained in detail in the next section.

5.2 Management of Operators

In order to reduce the user/designer interaction, I included in my scheme a way to apply deformation operators. An Operator Application Manager reflects the control of operators in my scheme. The Operator Application Manager controls the operators through a finite state machine.

The operator management uses Velocity Operators and Selection Operators, defining when to apply them (during the Time Handler steps), where to apply them, and how long to apply them.

The features I required for the Operator Application Manager are:

1. Capability of starting deformations at different moments. Deformations should be applicable at different moments of the model creation because it is difficult to design a single deformation that would create a model with diverse detail.

2. Capability of stopping deformations at different moments. Deformation Operators are easier to design when they do not have to be designed for a never-ending execution.

3. Capability of starting and stopping deformations based on the state of the model. This capability allows the user/designer to design simple deformations for creating models without worrying about deformations distortion.

To satisfy the aforementioned conditions, I decided to use a finite state machine where the deformations are associated with states. Also, I needed a way to trigger changes of states from

within the specification of the state machine. I used a known finite state machine that has these properties: Petri Nets.

A Petri net [78] is a representation of a discrete distributed system. Petri nets model systems whose states are atomic; the system can be in one or more states and can even have more than one instance of each state. Additionally, Petri nets define the relation between states and the conditions required to change states. Graphically speaking a Petri net is a directed bipartite graph; its constituting nodes are states and transitions. Figure 5.23 shows the visual components of Petri nets and a couple of examples.

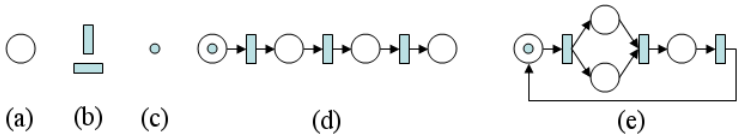

Figure 5.23: Petri nets visual components: (a) States, (b) Transitions (horizontal or vertical rectangles), and (c) Tokens (Drawn inside a state to indicate an instance of that state is occurring). The graphs in (d) and (e) are examples of Petri nets. Note that they are bipartite oriented graphs.

Petri nets are not indispensable for managing the application of operators. Actually, any finite state machine could have been used. However, I chose Petri nets because of its clear differentiation and control over states and transitions. Additionally, I envisioned using certain Petri net's properties to enhance the Automatic Model Creation process. In particular, Reachability and Boundedness were properties of interest. Reachability indicates if a given state can be reached given a set of active states. This property allows a user/designer know if a given deformation will be applied or not to the model under creation. Boundedness indicates if more than certain number of state instances can be active at a given time. This property allows a user/designer to know if a given deformation will be applied a certain number of times or not.

I used Petri nets for management of operators. Petri nets' states indicate the application of one or more operators at the same time or a consecutive application of operators. Operators are associated with the states; if a state is active then its operators are applied to the surface. Operators are applied in tandem by separating them into individual states and then connecting them linearly. The state changes are triggered by the transitions, which are fired every time a simulation step is completed. Note that firing a transition does not always make a state change; a state changes if the transition is fired and the state conditions are fulfilled. For my Automatic Model Creation framework, Petri net transitions are associated with queries of geometric properties of the model, so that when these queries give positive results then the transition is fired.

Although in general Petri nets can have more than one state active, the Petri nets used in this project are limited to having only one state active. The management of the operators with a

single-state engine avoids overwhelming the users by forcing them to design parallel deformations. In the addition to the previous restriction, the Petri Net's transitions used are "soft transitions." Soft transitions, in Petri Net theory, are transitions that can fire (change the net state) if at least one state previous to the transition is the current net state. Finally, I include the concept of time transitions: transitions that will attempt to fire at each step of the Time Handler after a certain number of steps have passed.

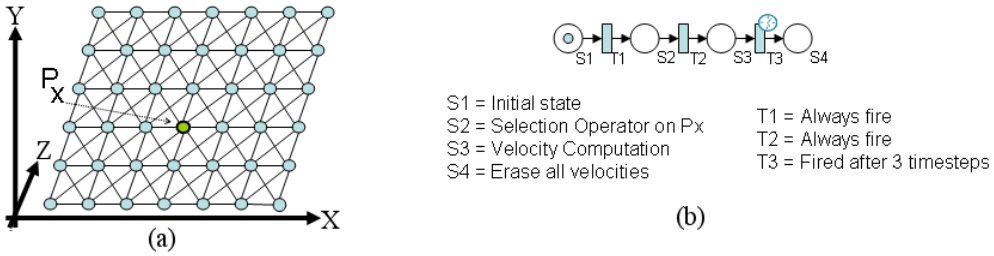

Figure 5.24: (a) Initial surface for the Petri Net in (b). (b) A Petri Net that creates a detail on a surface.

Figure 5.24.b shows a Petri Net which creates a prism-tower detail on a surface. State S1 is the initial state of the Petri Net; no operator is applied. State S2 applies the Selection Operator shown in Figure 5.2. State S3 applies a Velocity Operator that assigns a velocity in the same direction as the normal of the point. The surface evolves for three time steps and then all velocities are erased. Transitions T1 and T2 attempt to fire at each timestep, but they do not change the state of the Petri Net unless there is an active state (represented by a token in the input state of the transitions) previous to the transition; on the other hand, transition T3 is a timed transition and will fire after three timesteps have passed and there is an active state previous to the transition. Figure 5.25 shows the execution of the Petri Net shown in figure 5.24 and its effects on a surface.

Vebam's implementation of Petri nets is straightforward. The Petri nets used in Vebam are directly implemented in the source code. Each state contains a list of operators to be applied, and each transition contains queries geometric properties.

5.3 Summary

The problem of automatically assigning velocities to surface points is solved with Deformation Operators. I classified Deformation Operators into two types: Selection Operators and Velocity Operators. Selection Operators determine which points are going to be used during the deformation. Velocity Operators compute the intensity and direction of the velocities that will be assigned to points.

Separating deformations in Selection Operators and Velocity Operators is not mandatory. How-

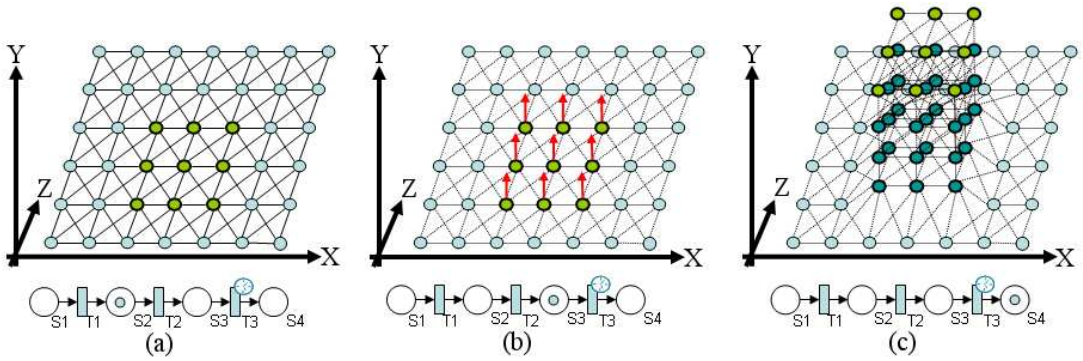

Figure 5.25: (a) State S2 executes Selection Operator. (b) State S3 assigns velocities to selected points (green points). (c) All velocities are erased at state S4 after four timesteps; during these the selected points moved upward (dark blue points).

ever, such separations facilitates the design of the operators. Also, the separation increases the variety of models by combining operators.

Velocity and Selection Operators support the Automatic Model Creation process by providing a way to simplify the selection of points and the velocity computation. However, the operators bay themselves still need a user/designer to apply to the surface. To release the user/designer from such task, I provided a way to automate the management the operators application.

With operator management, the Automatic Model Creation process is fully supported. The abstraction layer provided by the management of operators minimizes the inputs required from a user to automatically create a model. A user is only required to specify the operators' parameters (if any) to create an instance of a specific model. The Petri nets automatically apply operators; the states of the Petri Net have operators associated with them and the operators are applied to the surface when the state becomes the active state of the Petri Net. The state changes for operator management are evaluated at every time step.

Petri nets are created by user/designers. The user orchestrates the deformations that define a class of models by setting the states and transition of a Petri net. The user associate deformation operators with the states and specify the queries for each transition.

The Petri Nets for managing the operators may not be needed per se, however, they enormously facilitate the generation of complex models. It uses the divide-and-conquer approach allowing the composition of deformations and by enabling and disabling them at different stages of the model creation process.

Chapter 6

Secondary Automatic Model Creation Problems

The previous chapters describe how to specify deformations, and how to automate the application of deformation operators. In this Chapter, I present a couple of problems which are not part of the Automatic Model Creation problem per se, but which had to be solved.

These secondary Automatic Model Creation problems are, first, the conversion from Volipoc to polygonal meshes, and second, the visualization of Volipoc. The conversion problem consists of converting from Volipoc to the polygonal mesh representation. The visualization problem consists of showing the relevant information of the surface representation to a user/designer.

To solve the first problem, I designed an algorithm that uses the existing edges and points on the surface and selects the best ones that create a polygonal mesh. My algorithm is not a surface reconstruction algorithm: That is, my algorithm does not approximate a surface with the information in Volipoc, but selects a subset of the existing elements of a model's surface.

My visualization solution uses known rendering techniques to display a surface described within Volipoc. The techniques used are wireframe rendering and surfel rendering. Wireframe rendering is used to render the volatile edges between points as well as the point's normals. Surfel rendering is used to render the surface points; surfel rendering consists of rendering an ellipse with the same color and orientation as the point.

In addition to the rendering techniques, I enabled in Vebam some rendering options to enhance the visualization. These features are surfel culling and edge culling. Surfel culling works the same way as backface culling of polygonal meshes. That is, points facing the viewer are not rendered. Edge culling consists of deciding whether or not to render an edge. My edge culling approach consists of rendering an edge if and only if both of its edge points face the viewer.

In addition to wireframe and surfel rendering, I tested another rendering technique: using spheres instead of surfels. The spheres' colors are the same as the points. I found that this rendering technique is another good way to appreciate the surface's shape.

6.1 Volipoc to Polygonal Mesh Conversion

In Chapter 3, I mentioned that the surface to take into account in Volipoc is the outermost surface. I used this same concept as the basis of my conversion algorithm. My algorithm chooses the edges of the surface such that the outermost polygonal mesh is created.

The surface conversion algorithm converts a model in Volipoc to a polygonal mesh. The conversion algorithm follows the project's premise of keeping as close as possible the user's interpretation of the surface and what is being rendered, by not interpolating nor computing any new information. Instead, it selects the best triangles to create a water-tight mesh. The best triangles are the outermost triangles that do not intersect or overlap final triangles. The candidate triangles are the possible triangles that can be formed from an edge (pairs of points); note that all points on the surface are included in the polygonal mesh (a point in Volipoc is a vertex in the resulting polygonal mesh). The edges used are the edges of the tree graph of the model. The tree graph is a spanning tree whose root is the top-most point. The final triangles are the triangles chosen during previous analysis of edges. Figure 6.1 shows the main steps to translate the surface in figure 5.24.a.

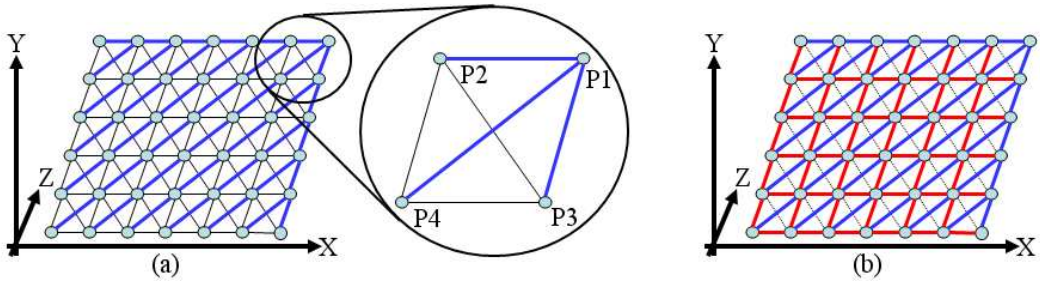

Figure 6.1: Main steps of the surface conversion algorithm: (a) Tree graph of surface (blue edges); all possible triangles for tree graph edges $\overline{P1P2}$, $\overline{P1P3}$, and $\overline{P1P4}$ are the triangles $\triangle P1P2P4$, $\triangle P1P3P4$, and $\triangle P1P2P3$. (b) Resulting polygonal mesh (points with blue and red edges); triangle $\triangle P1P2P3$ was not selected because it overlapped with triangles $\triangle P1P2P4$ and $\triangle P1P3P4$.

The algorithm works in two phases: tree creation and edge selection. The first phase uses Volipoc as a graph representation of the surface. The second phase uses the geometric properties of the surface representation elements to construct the polygonal mesh. I denote the selection of the edges by a labeling indicating whether a volatile edge belongs to the polygonal mesh or not. I use two labels in the algorithm: TREE-BRANCH and CLOSING-LOOP. TREE-BRANCH indicates that the edge is part of the tree-graph representation of the surface. CLOSING-LOOP indicates that the edge closes a loop with a TREE-BRANCH edge.

The principal data structure used in this algorithm is the set of points and volatile edges of Volipoc; initially, none of the points and edges are labeled in any way. Additionally, I use a queue

q for traversing the points throughout the algorithm. I denote by p_{Top} the highest point, p_f the point at the front of q, by p_p a reference to the point being processed, by e_i the ith edge that links the point being processed (p_p) with any other point, and by t_i the ith edge that is not labeled as TREE-BRANCH and links p_p with any other point. The result of this algorithm is a labeling of points. The labeling indicates all edges that form the triangles of the final polygonal mesh. The following prosecode describes the conversion algorithm:

1. Phase 1 - Tree creation. This phase organizes the points into a tree. The root of the tree is the top-most point of the surface (see Figure 6.1.a).

   (a) (Initialize graph-traversing cycle) Insert p_{Top} in q.

   (b) (Obtain next point to process) Make $p_p \leftarrow p_f$, remove p_f from q.

   (c) (Build tree)
   If ((e_i is not labeled) AND ($e_i \neq p_p$ is not labeled))
   Then label e_i as TREE-BRANCH
   This step adds to the tree (through labeling edges) the edges that do not close loops and avoids adding edges that are already in the tree.

   (d) (Continue) Push in the front of q all points neighbor of p_p that are not labeled. This step guarantees that all the points of the model's surface will be processed to build the tree.

   (e) (Repeat) Repeat steps 1.a to 1.d until q is empty.

2. Phase 2 - Edge selection. This phase selects edges that are not part of the tree, thus generating the triangles of the resulting polygonal mesh. The selected edges are those that generate the outermost triangles of the polygonal mesh. The computation of the outermost triangles includes the outermost triangles computed in the previous step, which is why Phase 2 requires that the root of the tree-graph from Phase 1 is the top-most point.

   (a) (Initialize tree-traversing cycle) Push in q all points neighbor of p_{Top} (order is not important).

   (b) (Obtain next point to process) Make $p_p \leftarrow p_f$, remove p_f from q.

   (c) (Select loop-closing edges) For all t_i, label t_i as CLOSING-LOOP if the t_i generates one or two outermost triangles (Appendix A contains the computation of outermost triangles). This step starts generating the polygonal mesh by closing loops. Note that there may be more than one edge that generates one or two triangles, which edge is not important as long as the triangles are the outermost ones (the condition of an edge generating at least one triangle is needed to process open surfaces).

(d) (Repeat) Repeat steps 2.b to 2.c until q is empty.

(e) (Return) The algorithm ends once all the tree-graph has been traversed. The result of the algorithm is a labeling indicating the edges constituting a polygonal mesh. These edges are those labeled as TREE-BRANCH and CLOSING-LOOP.

Note that the computation of the outermost triangles that have the top-most point as one of their vertices makes those triangles as the top-most triangles of the surface. This condition helps that the outermost triangles computed subsequently will be the outermost triangles of the surface. Appendix A contains the process to compute outermost triangles.

One could argue that the best edge to select would be the one that generates the most regular triangles (which was one of the motivations for a new surface representation, avoid large edges). However, since the surface already keeps a quasi-uniform density (see Chapter 3); hence, there is no need to solve the ambiguity between edges for this algorithm in particular.

Figure 6.2 shows different steps of the aforementioned algorithm when applied to a flat open surface. Figure 6.2.a shows the initial surface; a flat square of 25 points. The edges and points are not equivalent to a polygonal mesh since there are overlapping and intersecting triangles. Figures 6.2.b-c show the first two passes of Phase 1; the blue edges are labeled as TREE-BRANCH. Figure 6.2.d shows the tree-graph resulting from Phase 1. Figures 6.2.e-g show the first three passes of Phase 2. The red edges are the edges labeled as CLOSING-LOOP. Figure 6.2.h shows the resulting polygonal mesh (green triangles) obtained at the end of the algorithm, the mesh's triangles are formed by the edges labeled as TREE-BRANCH and CLOSING-LOOP.

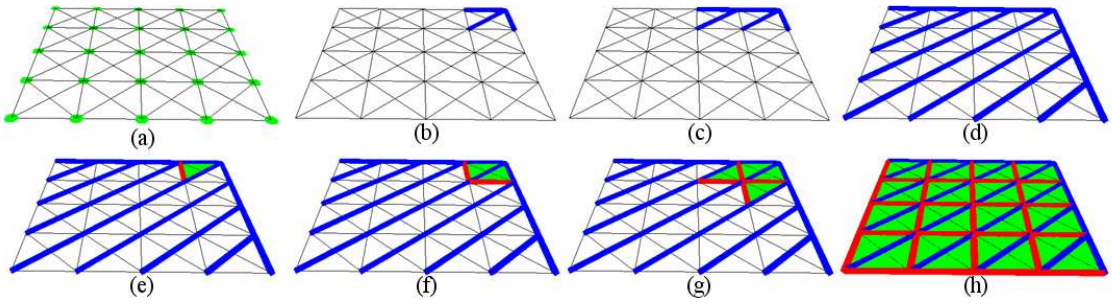

Figure 6.2: Steps of the conversion from Volipoc to a polygonal mesh representation of a flat open surface.

Figure 6.3 shows the same cases as Figure 6.2 but with a closed-surface sphere. Note that the TREE-BRANCH and CLOSING-LOOP edges are not culled during rendering (Figure 6.3.b-g). Images in Figure 6.2 and Figure 6.3 are snapshots of the Vebam software.

Vebam's implementation of the surface conversion algorithm is straightforward. The algorithm implementation creates the tree graph by traversing the points in PS with edges' information. The

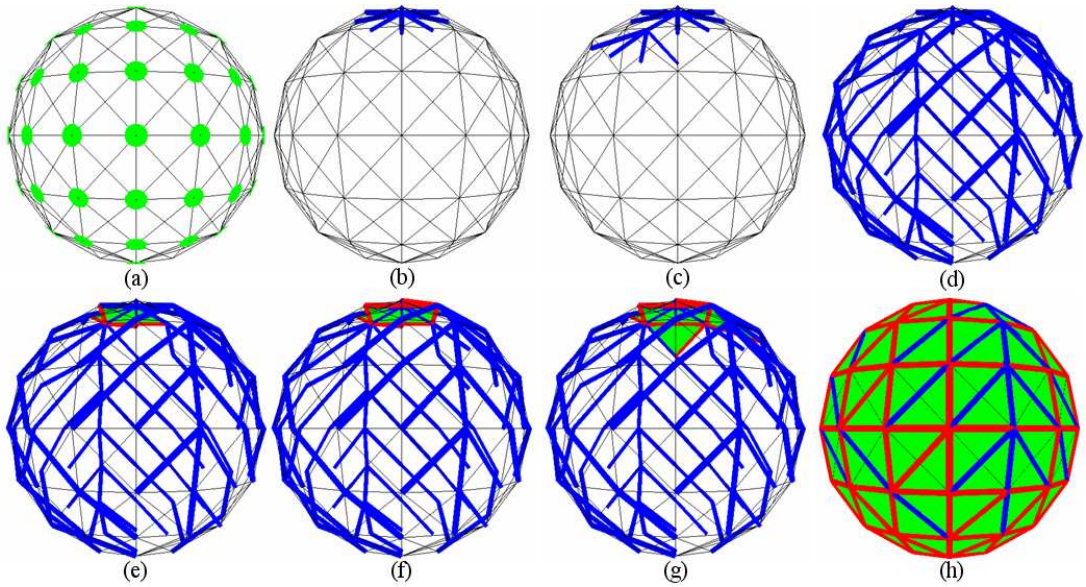

Figure 6.3: Steps of the conversion from Volipoc to a polygonal mesh representation of a closed spherical surface.

sets PS and LS are used to construct all possible triangles sharing each edge. The best triangles create the polygonal mesh object. The object also uses the additional information of points: normal, color, and texture coordinates. Vebam's interface also includes the option to select or load a specific texture to be used on the polygonal mesh. Once the conversion process is finished, the resulting mesh can be rendered.

Generating a polygonal mesh from a point cloud is not a new problem. This problem has been addressed in different ways in Computer Graphics. Most of them make an interpolation of the surface using point samples. This differs from my approach because I obtain a polygonal mesh without creating any new information for the polygonal mesh. Although, the points of Volipoc could also be used with any other surface-reconstruction algorithm.

Even though the Automatic Model Creation goal does not require a surface-conversion algorithm, I decided to pursue this goal for completeness. Besides, the surface-conversion algorithm demonstrates that a polygonal mesh can be obtained without more remeshing operations than those already executed during resampling.

My surface-conversion algorithm selects arbitrarily between possible triangles. This may cause that preferred triangles (more equilateral) are not selected and any triangle can be chosen instead. This may seem to contradict my initial justification for creating my own surface representation avoiding long edges on triangles (see Chapter 3). This is not the case because excessively long edges cannot appear due to the resampling process. Therefore, arbitrarily selecting any pair of outermost triangles does not affect the quality of the polygonal mesh.

The polygonal mesh obtained from my surface-conversion algorithm may require some decimation post-processing. Since the size of triangles depends on the grid size, long flat segments on a surface may use more triangles than needed to represent the segments. Therefore, a decimation process could be used to reduce the number of triangles and still have the same surface. However, such process focuses more on enhancing the quality of the polygonal mesh and does not influence the Automatic Model Creation goal.

6.2 Surface Rendering

The Automatic Model Creation is greatly facilitated when a user can visualize the surface as it evolves in time. Such visualization allows a user to interrupt and replan the deformation accordingly until a desired deformation is achieved; in other words, the visualization is required for supporting debugging and interaction purposes.

In this Section I explain the different approaches I undertook to visualize the surface representation relevant information. Even though the main goal of my research is the automatic creation of 3D models, these models have to be rendered to properly appreciate the results. Since I used a new surface representation, I had to define a way to visualize my models.

One problem I faced was deciding which rendering techniques I could use to visualize the surface information. For this, I determined what is the relevant information of Volipoc. I decided that such information would be the surface points and their visual attributes (orientation and color) as well as the surface shape. The surface shape is defined by the volatile edges linking points together.

I considered that a proper visualization technique would include the point samples of the surface, their orientation, and their color, but I also considered important the shape of the surface defined by the volatile edges of Volipoc.

I used known rendering techniques to visualize Volipoc's relevant information. I used surfel rendering for points and wireframe rendering for edges. Also, I tested changing the rendering primitive in surfel rendering, using spheres instead of flat circles.

Volipoc can be rendered neither as a traditional polygonal mesh nor as a pure-point surface. Even if the links between points on my surface are used to define triangles, the surface can not be rendered as a polygonal mesh because it is not a tessellation of triangles (triangles intersecting or occluding each other appear). Even though the surface can be rendered as a pure-point surface, important information (edges) would be omitted. The rendering is done using wireframe for edges and pure-point rendering for points.

6.2.1 Surfel Rendering

Surfel rendering is a very simple technique. It consists of using a circle to render oriented points from a surface (see Figure 6.4). The circle is centered at the position of the point and has the same orientation and color as the point. The circle can be rendered with different shading options; however, the shading does not need to be complex. In particular, I implemented in Vebam two shading options: with and without specular reflection. Finally, circle culling is done by testing its normal against the viewer direction.

Figure 6.4 shows the surfel rendering features. The first row, Figures 6.4.a-d, show a flat square with 225 points; the circle diameter varies from 0.6 to 1.2 grid units in 0.2 increments. The next row, Figures 6.4.e-i, show a sphere represented with 90 points and same diameter changes as in the previous row. Note that this is a sphere rendered with surfels and not points rendered with sphere primitives. Figures 6.4.j-n repeat the same cases but with specular shading enabled. Finally, Figure 6.4.o shows a sphere with 13314 points rendered with a circle diameter of 1.0 grid units, Figure 6.4.p show the same sphere with specular shading enabled. Figure 6.4.q-r are the same as the previous two cases but with a circle diameter of 1.2 grid units. All images shown in Figure 6.4 are snapshots from the Vebam software.

6.2.2 Wireframe Rendering

Like surfel rendering, wireframe rendering is also a very simple technique. It consists of using a line to render edges. Line colors are predefined or obtained from the points. I considered that simple black lines were enough to describe (visually) the shape of a model with Volipoc. Pure wireframe rendering does not include any culling operation; however, culling helps the viewer by only displaying the surface facing the viewer. With culling, a viewer does not have to mentally separate the shape of the surface closer to the camera from any other surface behind.

I implemented wireframe culling within the wireframe rendering. If an edge has at least one point oriented toward the camera then the edge is rendered; otherwise, it is not.

Figure 6.5 shows some examples of wireframe rendering in Vebam. Figure 6.5.a shows an initial flat surface of 25 points. Figure 6.5.b shows the edges rendered with wireframe rendering. Figure 6.5.c shows the same surface but with a smaller surfel radius. Figure 6.5.d shows a sphere of 90 points with its edges rendered and culling in operation. Finally, Figure 6.5.e shows the same sphere but without any culling.

6.2.3 Sphere Rendering

In addition to the surfel rendering technique, I tested with different rendering primitives, spheres instead of surfels. Spheres have the advantage of not requiring orientation as surfels do; therefore,

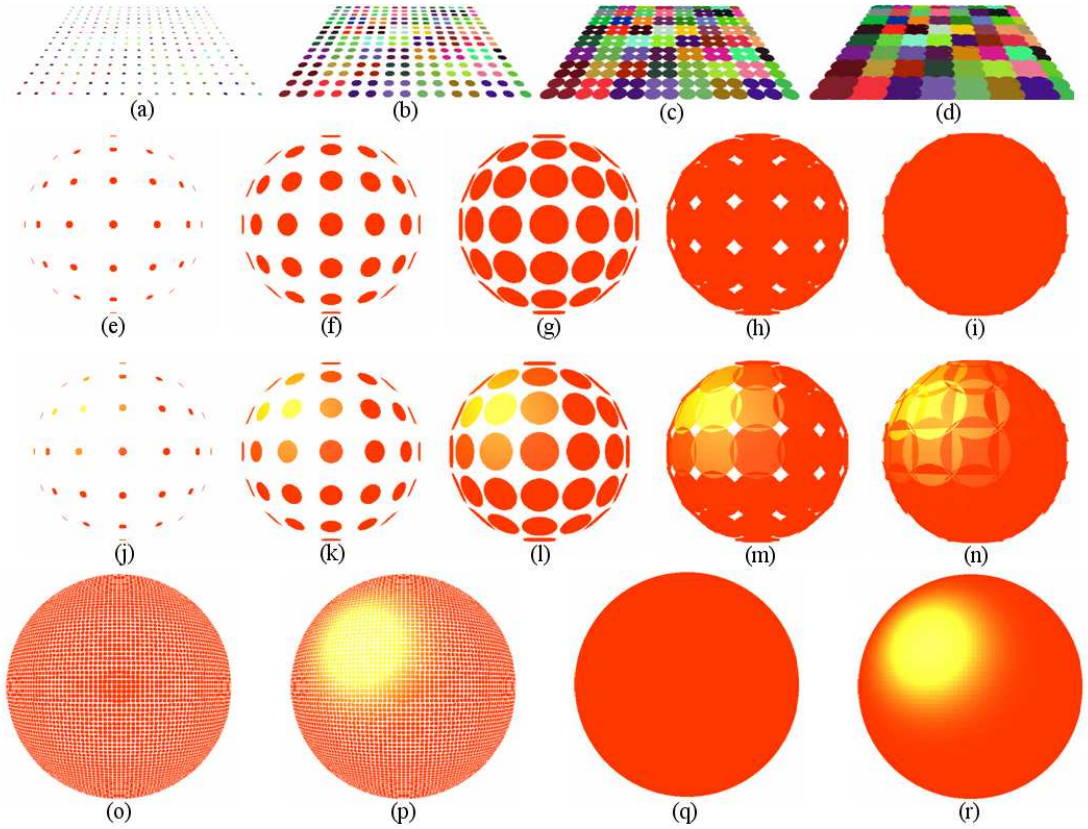

Figure 6.4: Surfel rendering.

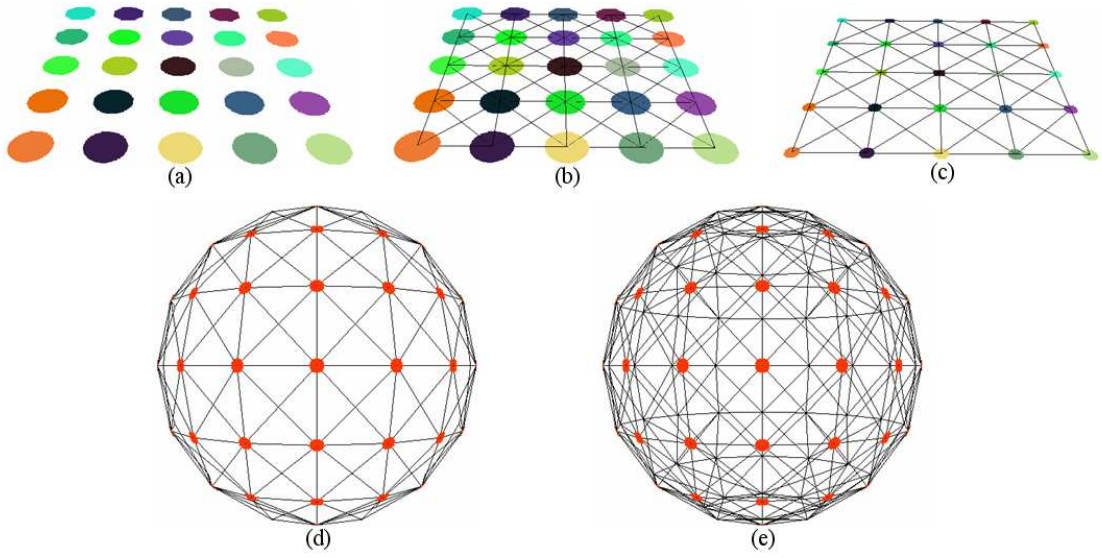

Figure 6.5: Wireframe rendering.

computation to orient the primitive is not needed. I tested with two shadings of the spheres: normal and pure-emissive. Normal shading consists of having ambient, diffuse, specular, and emissive light components added to the surface color. Pure-emissive consists of only adding an emissive component. Finally, the sphere's diameter can be modified in the same way as the surfel's diameter.

Figure 6.5 shows models with Volipoc and sphere rendering shading modes. The left column contains models rendered with surfel rendering. The center column contains models rendered with spheres in the pure-emissive shading mode. Finally, the right column contains models rendered with sphere rendering in normal shading mode, the sphere rendering enhances the appreciation of surface planar variations. Examples of additional models and their renderings are shown in Chapter 7.

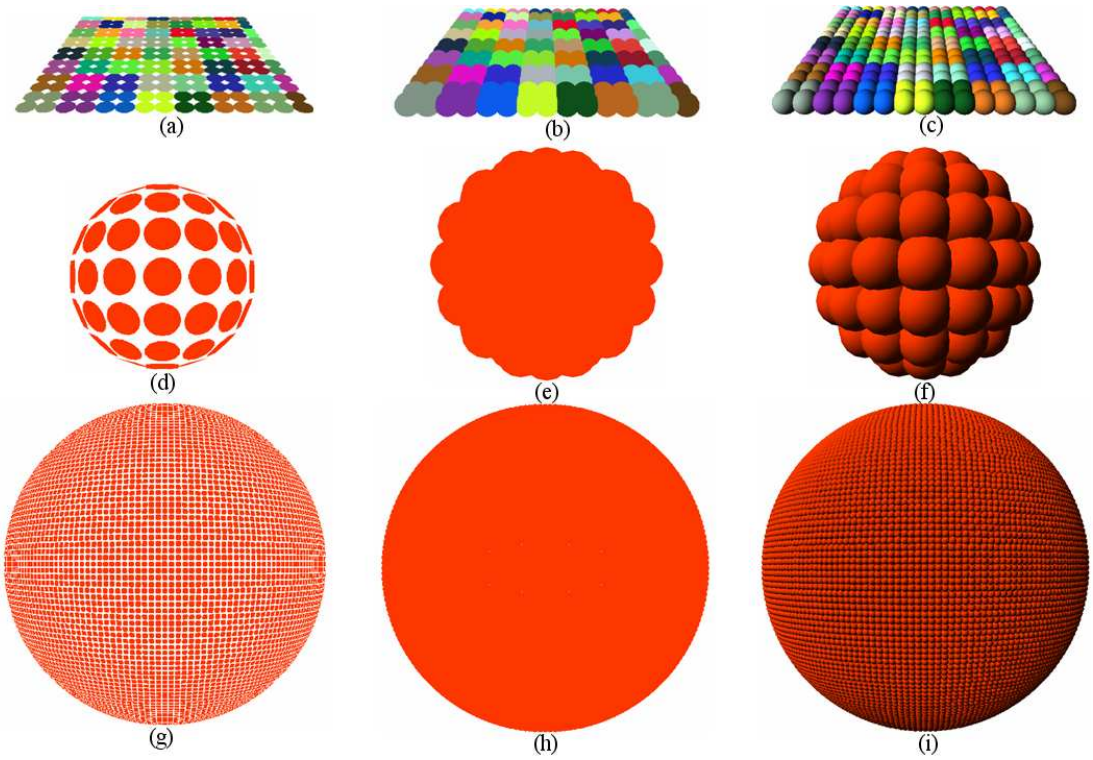

Figure 6.6: Sphere rendering.

6.2.4 Additional Rendering Features

In addition to the rendering techniques explained in the previous sections, I added some additional rendering features to Vebam: texture mapping and flat, Gourad, and Phong shading. These are well known techniques in Computer Graphics. These techniques are available for both the polygonal mesh obtained resulting from my conversion algorithm and the polygonal mesh obtained from rendering all possible triangles (including overlapping and intersecting triangles). For the second

type of polygonal mesh, since all possible triangles are used for rendering, then some triangles may overlap or intersect. This may cause z-buffer-related artifacts to appear on the rendered image but would not affect the surface construction.

Additionally, I implemented texture mapping by adding coordinate textures to the surface representation points. These coordinates are linearly interpolated in resampling operations. Figure 6.7 illustrates the additional rendering features on two models. The first model is a flat rectangle of 900 points. The second model is the result of deforming the first one with **ScaledUnitary-Gaussian**(p,2.25, *8.0*), where p is the point closer to the centroid of the surface. Figures 6.7.b and 6.7.f show the models rendered with flat shading. Figures 6.7.c and g show them with Gourad shading (Phong shading did not show any visual difference). Finally, Figures 6.7.d and 6.7.h show the models with the texture coordinates of a test texture.

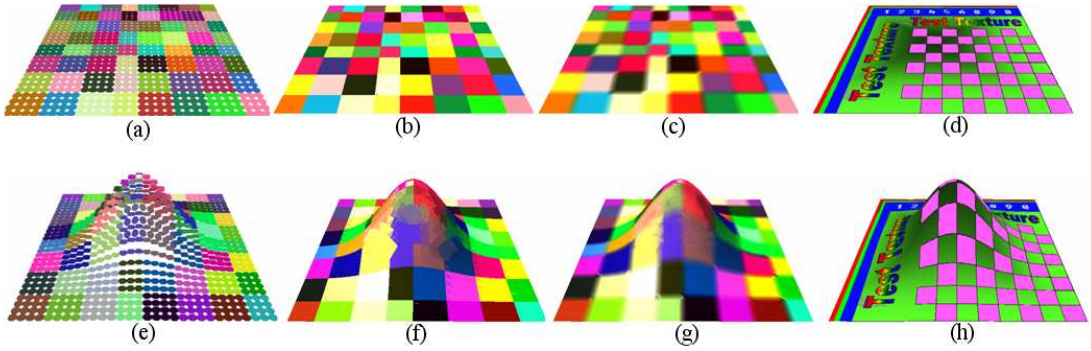

Figure 6.7: Additional Rendering Features.

6.3 Summary

I have presented solutions for problems related to the automatic model creation goal. The problems I addressed are the conversion of a model from Volipoc to the polygonal mesh representation and the visualization of Volipoc.

My conversion algorithm consists in using the existing edges of the model to build a polygonal mesh without interpolating or computing any new information. This means that the polygonal mesh will be the exact result of the deformed surface and not an approximation. My algorithm demonstrates that a polygonal mesh can be obtained from Volipoc without using remeshing operations other than those done automatically during resampling. My surface conversion algorithm only uses existing surface information; some may see the conversion algorithm as a filtering process in which only the best edges are chosen.

As for the rendering features, I tested different rendering options for visualizing Volipoc. I used rendering options that are common in Computer Graphics. I used surfel+wireframe rendering,

all-possible-triangles polygonal mesh rendering, and different shading options. I chose these options because they have been used by other researchers in surface representations similar to mine. Additionally, I thought my rendering options display the relevant information of Volipoc (points and volatile edges).

Chapter 7

Results

In this Chapter, I present a set of models that were automatically created with Vebam, intended to demonstrate the different features of the modeling framework. Note that all the figures in the present Chapter, unless otherwise specified, are snapshots from the Vebam software.

I separated the models into two types: semi-automatically generated and automatically generated. The first type are those models that I generated by applying deformation operators interactively. On the other hand, the automatically generated models were generated using an Operator Application Manager to automatically apply the deformation operators. As for the semi-automatically generated models, sometimes the initial surfaces are deformed in different stages to achieve certain effects. All specific stages and operators used to generate a model are specified in detail in a per-model basis. The algorithms of the deformation operators are explained in detail in Section 5.1.3 respectively.

I completely implemented the "Vebam" software from scratch in C++ on Win32. Since my specifications involve novel concepts, it is better to make an original implementation, providing flexibility to the process of testing the specifications and redesigning them as necessary. Vebam's goal is to demonstrate the utility of the scheme specifications; therefore, some features are limited to an in-code implementation rather than an implementation supporting full user interaction. Nevertheless, the design of Vebam allows a user to input parameters without having to recompile Vebam.

The first implementation of Vebam and the two basic operators (VoronoiRegionalization and ScaledUnitaryGaussian) were published in at the "Fifth International Workshop on Computer Graphics and Geometric Modeling, CGGM 2006", see [77]. The paper also contained some of the mushroom-like models.

The first Section of this Chapter contains the following subsections:

- Semi-automatically Generated Models - These are basic models that I generated with a simple deformation on an initial surface. Also, I added a simple feature to differentiate between types of models. These models aim to demonstrate that Volipoc properly supports stretching and compressing deformations and that the user/designer is freed from giving maintenance to the surface.

- Model Parameterization - These models are of the same class (same deformations) as the previous ones, but created with different parameters. The goal of these models is to demonstrate how easily one can create different instances of the same type of models by simply manipulating existing deformations. Additionally, the utility of deformation operators is also demonstrated with these models.

- Automatically Generated Models - These models are created with a Petri net that manages the deformations. These models demonstrate that complex and varied models can be achieved by delegating the repetitive tasks of model generation to the deformation manager.

- Artistic Models - I created these models to exemplify how easy richly detailed models can be created with all the components of my Automatic Model Creation framework.

- Summary - Summarizes the results presented in this section.

7.1 Semi-automatically Generated Models

In order to demonstrate the utility of deformation application and deformation operators, I created a library of fruit-like models. The types of models I created are: tomatoes, pears, strawberries, bananas, apples, and mushrooms.

7.1.1 Tomato-like Models

I created tomato-like models from an initial sphere. Figure 7.1 shows sketches of the steps to create the tomato-like models. The first step (Figure 7.1.a) was to apply a deformation composed by two types of velocities. The first velocity is in the same direction as the points normals and with a constant velocity, which makes the sphere grow uniformly. The second velocity was obtained by applying the **ScaledUnitaryGaussian** on the upper half of the sphere. The operator is applied following a circle centered at the top-most point of the sphere. This operation makes the top-most part of the sphere grow more slowly than its surroundings, thus, generating a concave region (see Figure 7.1.b). Finally, I applied individual **ScaledUnitaryGaussian** operators on the upper-half of the model to create bumps as those presented in real tomatoes.

Figure 7.2 shows images taken from Vebam during the creation of a tomato-like models. The image contains two rows: the upper row shows snapshots of Vebam where the models are rendered with surfels and the other row shows snapshots where the models are sphere-based renderings of the model. In both cases, the color of the points (surfels and spheres) are directly proportional to their velocities magnitudes. In some cases, the color of a particular point or points was changed to highlight a particular feature.

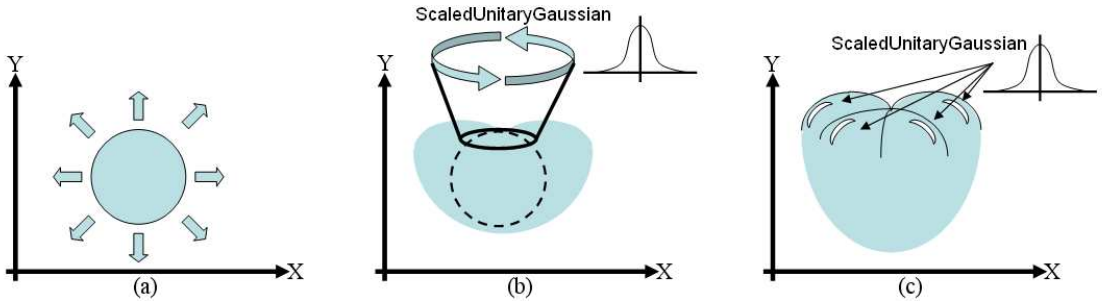

Figure 7.1: Tomato-like model generation steps.

Figure 7.2.a shows the initial sphere and the velocities (as color intensities) of the first step to create the tomato. The red point is the top-most point of the sphere. The initial sphere contains 899 points. After 10 seconds the model has deformed to that shown in Figure 7.2.b, which has 1747 points. Figure 7.2.c shows the velocities assigned in step 2, no further deformation has occurred). Figure 7.2.d shows the model at 10 seconds after; soft protuberances have started to appear on the upper half of the model. At this point the model has 2338 points.

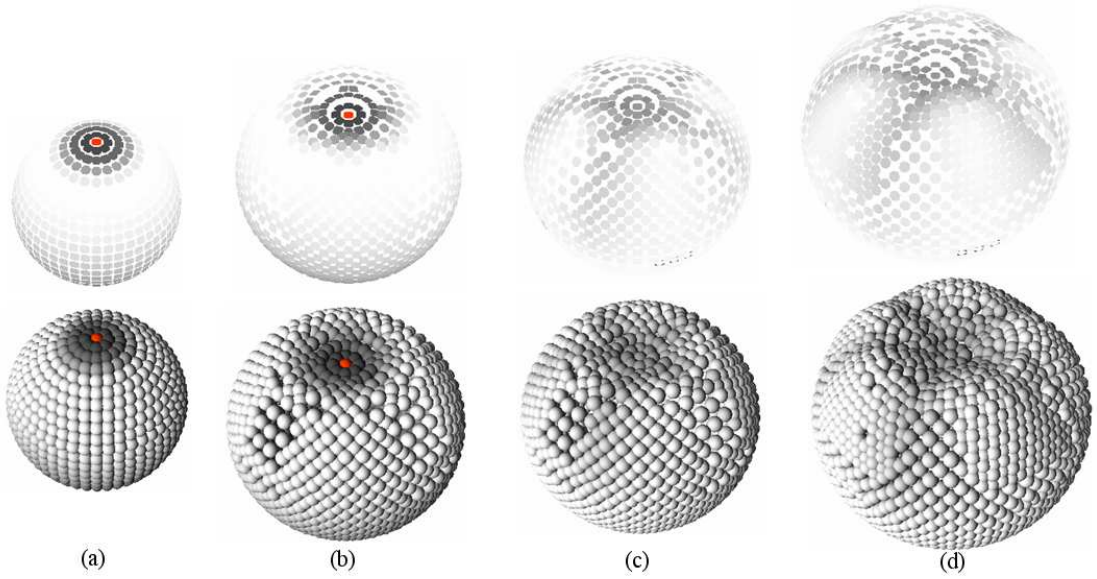

Figure 7.2: Tomato model at different generation steps.

This first model, tomato model, shows how easily a user can deform an initial surface and obtain a desired shape. The tomato top shows how varying velocities (through a deformation operator) simplifies the model generation. Instead of individual point velocity assignment, a user just has to apply one deformation operator per top bump.

Figure 7.3.a shows the model after 10 additional seconds; that is, 20 seconds after the second

step to create the tomato. Finally, Figure 7.3.b shows the final model. This final model is rendered as both surfels-edges (up) and a polygonal mesh (down). This polygonal mesh is obtained by applying my conversion algorithm; additionally, a red color texture was applied to the polygonal mesh.

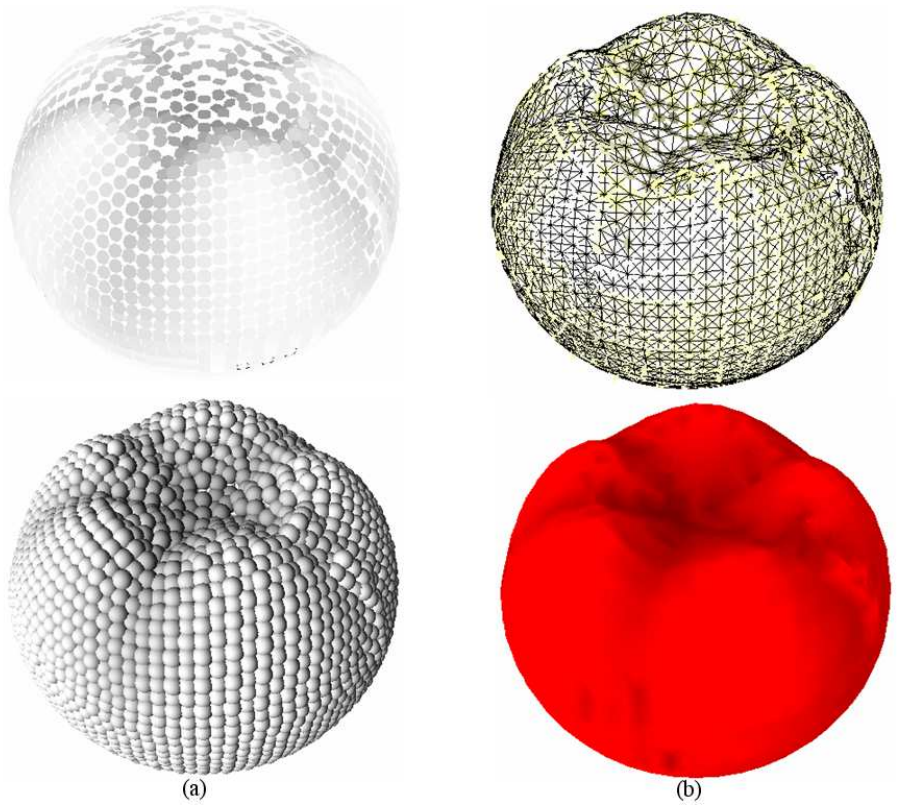

Figure 7.3: Tomato model at different generation steps and final model.

Figure 7.3 shows one final tomato model. In column a, the model shows how the surface has kept a quasi-uniform density. Additionally, the shape of the model is enhanced by the color intensities of the last deformations applied, the bump generation deformations. In column b, we can appreciate that despite the irregularities on the surface the initial surface keeps a regular connectivity, that despite not being a polygonal mesh, it does not cause a problem to obtain the polygonal mesh shown in the bottom-right corner.

7.1.2 Pear-like Models

Similar to the tomato models, I created pear-like models from an initial sphere. First, I apply an inflating deformation to the lower half of the sphere (see Figure 7.4.a). An inflating deformation consists of velocities with the same direction as the points normals. At the same time, I use the **ScaledUnitaryGaussian** operator as with the tomato model, but there is a difference: I subtract

the velocity intensity provided by the operator. This makes the sphere deform into a pear-like model (see Figure 7.4.b). Finally, I bend the upper half of the pear so it does not look too symmetric (see Figure 7.4.c). I made the bending by adding a horizontal velocity component to the points of the upper half of the model.

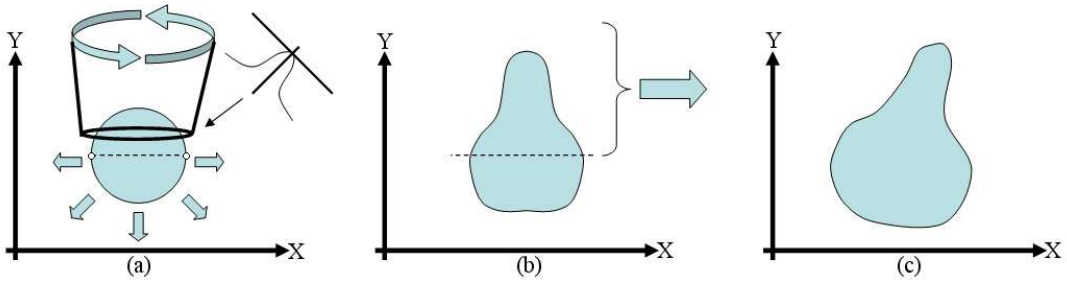

Figure 7.4: Pear model steps.

Figure 7.5 contains snapshots of a pear model at different stages of its creation. Column (a) shows the initial sphere consisting of 606 points. As with the tomato model, the points color intensities are directly proportional to their velocities. The lower half of the sphere (blue) has an inflating deformation, the upper half has the velocities computed by the 1-**ScaledUnitaryGaussian** factor, as explained before. Column (b) shows the surface after 10 seconds; it contains 1766 points now. At this point, I added the velocities to bend the upper half of the model. Column (c) shows the surface after additional 10 seconds, when it contains 2685 points. This same model is shown in column (d); the upper image is the polygonal mesh while the lower image is the same polygonal mesh with a pear-like texture.

The pear is an interesting model. As opposed to the tomato model, the pear model has concave regions that are generated by a deformation whose velocities' intensity are subtracted from the unit). Also, the pear model exemplifies how different segments of the surface can be deformed without affecting each other. For example, the bottom half of the pear is inflated while the upper half is pulled to the right.

7.1.3 Strawberry-like Models

The next model I created was a strawberry-like model. Figure 7.6 shows the steps for creating this type of models. First, I created a concave region similar to that of the tomato model. That is, I applied a **ScaledUnitaryGaussian** operator in a circular pattern at top of the sphere (Figure 7.6.a). After that, I applied a "pulling" deformation to the lower half of the sphere (Figure 7.6.a). This deformation consists of downward velocities scaled to one; the maximum is at the center of the sphere while the minimum at its edges. This deformation makes the lower half squeeze downwards (Figure 7.6.c).

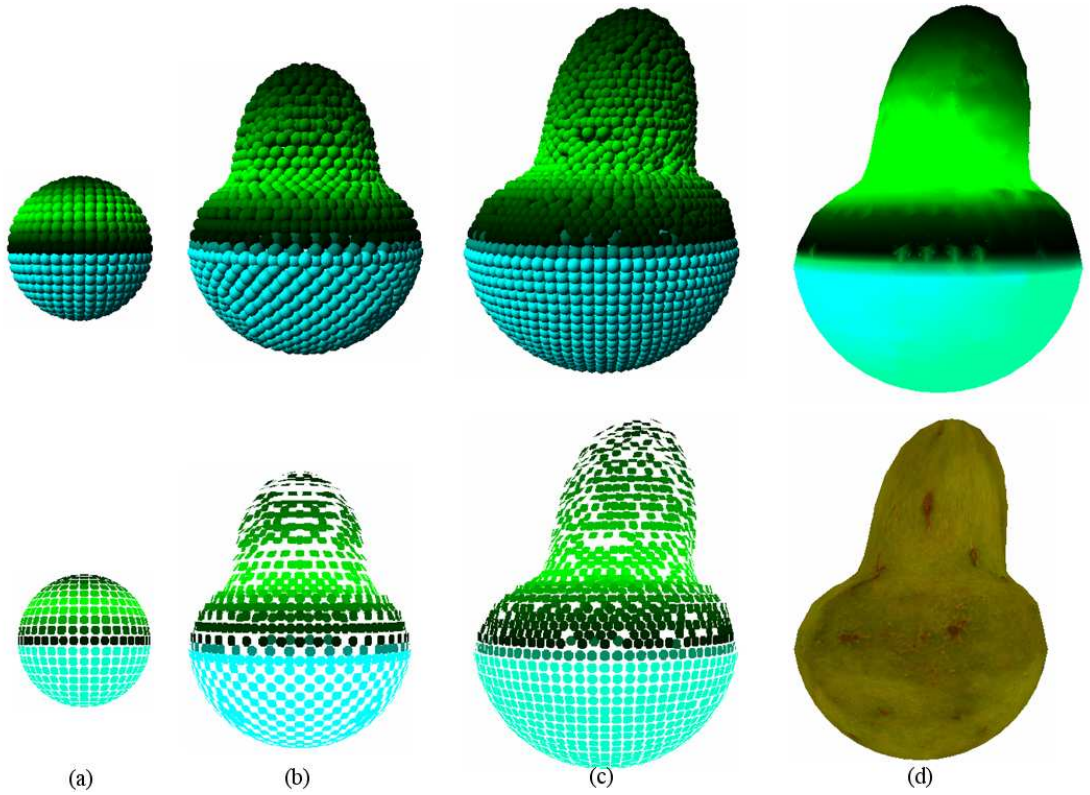

Figure 7.5: Pear model at different stages of its creation.

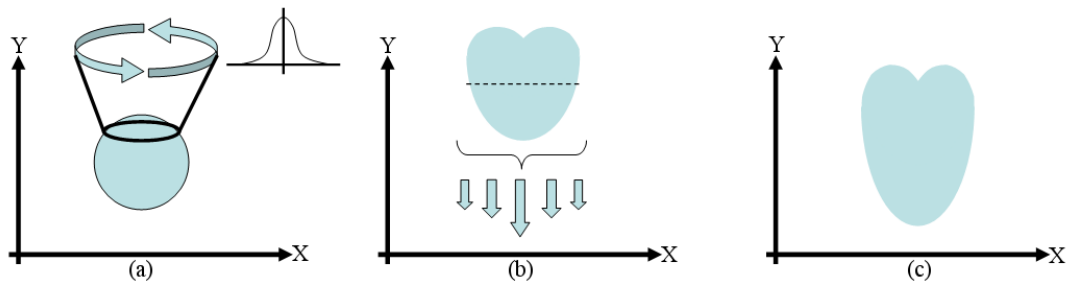

Figure 7.6: Strawberry model steps.

Figure 7.7 shows a strawberry model I created with the aforementioned method. Figure 7.7.a shows the initial sphere of 606 points and the initial velocities (as color intensities). Figure 7.7.b shows the model after five seconds. It has deformed into a surface of 930 points. At this point, I applied the downward-pulling velocities for the second stage of the model creation. After five more seconds, the model deformed into that shown in Figure 7.7.c. At this moment, the model contains 1342 points. Finally, Figure 7.7.d shows the polygonal mesh obtained from the final model; the model is rendered with (below) and without (above) a strawberry texture.

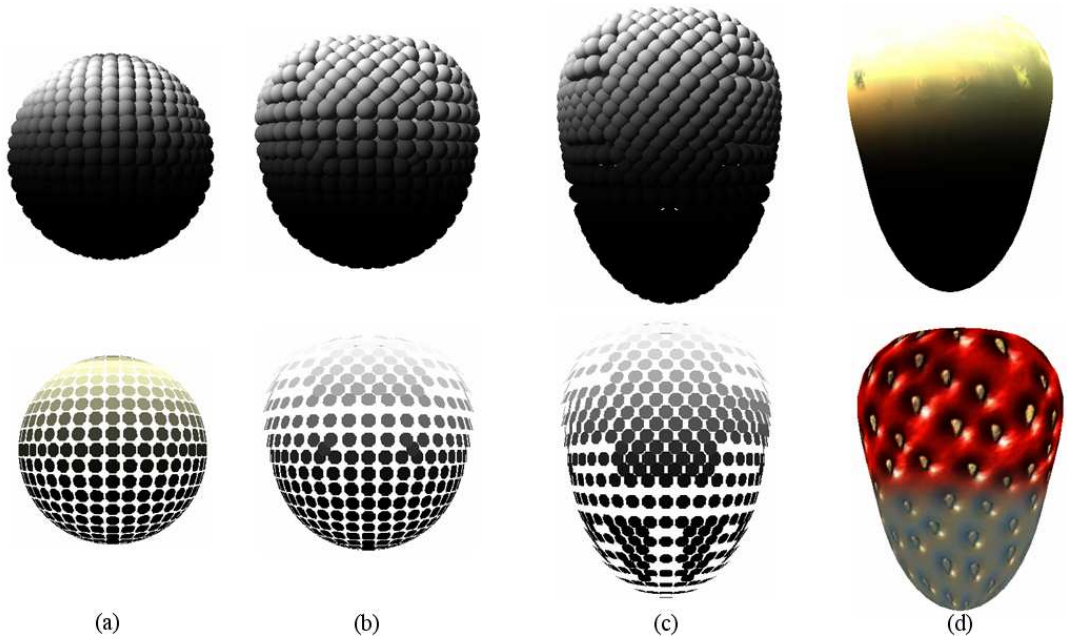

Figure 7.7: Strawberry model at different stages of its creation.

The strawberry model is another model that demonstrates the resampling effectiveness. Additionally, it also demonstrates the effectiveness of the inheriting information during resampling of the surface. Apart from the points' velocity and color, the texture coordinates are also well distributed in the deformed surface, as is shown in 7.7.d where the strawberry texture is uniformly distributed over the model.

7.1.4 Banana-like Models

The next model I created was a banana-like model. Figure 7.8 shows the steps I followed to create this type of models. First, I applied a **ScaledUnitaryGaussian** operators along six vertical stripes around the initial sphere (Figure 7.8.a). These initial velocities make the sphere deform into a hexagonal prism-like with rounded corners. Next, I applied a pulling velocities to each half of the sphere so that the banana will curve longitudinally. These pulling velocities are directly

proportional to the distance of the surface point to the center of the sphere and scaled to one; that is, the maximum velocity will be one and this will be at the top and bottom of the sphere. At the same time I applied upward and downward constant velocities to the upper and lower halves of the sphere respectively. These velocities are also scaled to one, but with the maximum velocity at the center of the sphere and the minimum at the borders (Figure 7.8.b).

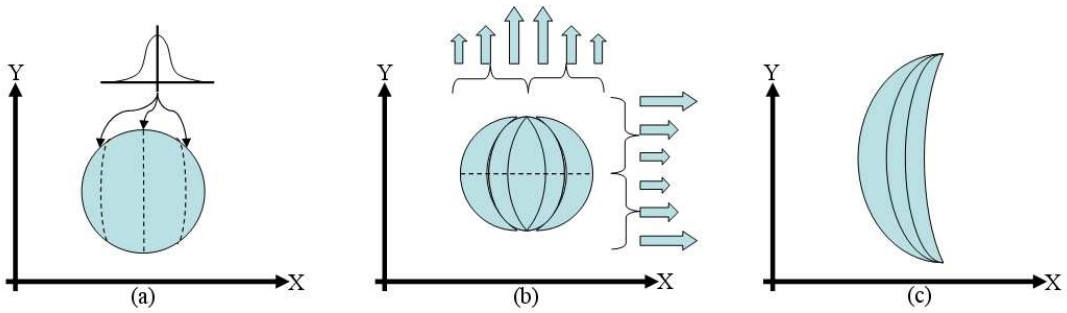

Figure 7.8: Banana model steps.

Figure 7.9 contains snapshots of a banana model at different stages of its creation. The initial sphere, shown in Figure 7.9.a, contains 246 points. Figure 7.9.b shows the model after three seconds of deformation. At this moment, I applied the velocities of the second stage. In Figure 7.9.c., the model has deformed after twenty more seconds. Finally, Figure 7.9.d shows the polygonal mesh obtained from the model (left) and its rendering with a banana-like texture (right).

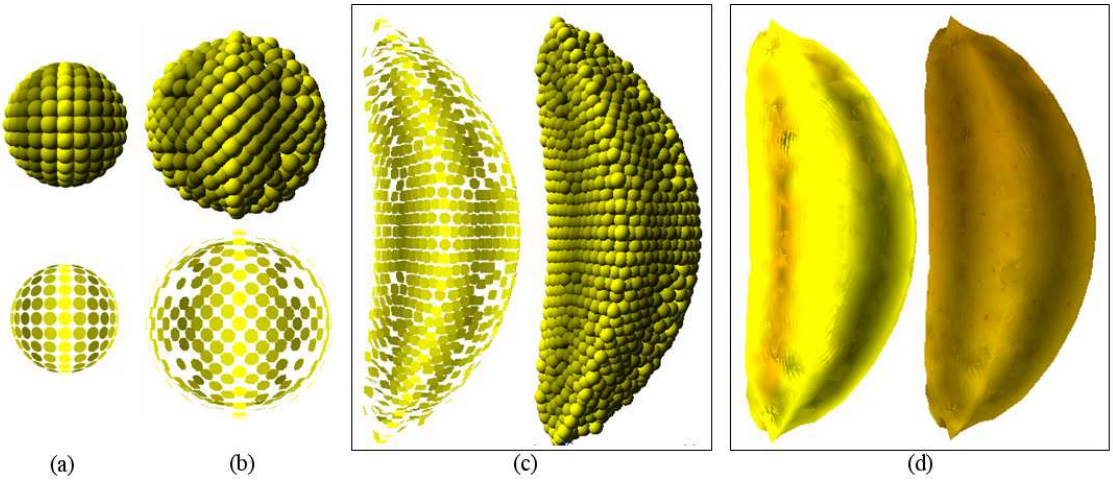

Figure 7.9: Banana model at different stages of its creation.

7.1.5 Apple-like Models

The next model I created was an apple-like model. To create this type of model, I repeated upper deformation of the strawberry model (Figure 7.10.a). At the same time I applied four **ScaledUnitaryGaussian** operators on the lower half of the sphere and one at the very bottom of the sphere, the velocities of this last one has a negative direction. The negative direction makes the surface deform inward into the model volume. These five operators make the model create a crown-like (Figure 7.10.b) detail on the lower half of the sphere. Finally, I apply two **ScaledUnitaryGaussian** operators on the top of the model (Figure 7.10.c) to create a couple of protrusions on the top.

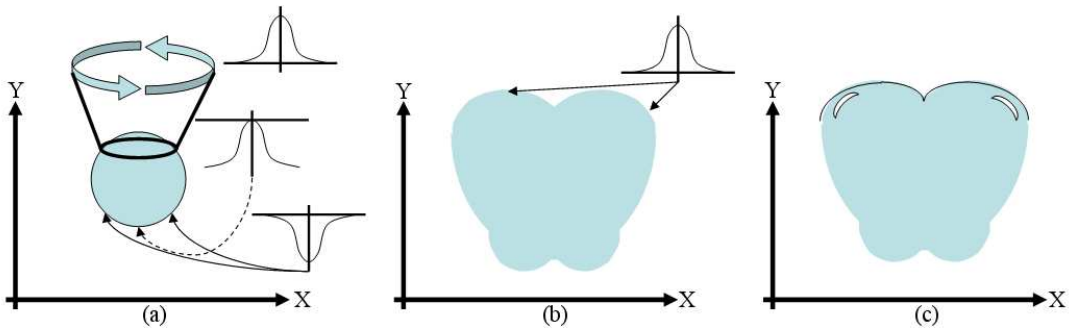

Figure 7.10: Apple model steps.

Figure 7.11 contains snapshots of the apple model at different stages of its creation. The images show the model from three different points of view. The first point of view has the camera pitched approximately 45 degrees downward. The second point of view has no inclination for the camera. The third point of view has the camera pitched approximately 45 degrees upward. Figure 7.11.a shows the initial sphere of 606 points. The points' color intensities are directly proportional to the velocities assigned to the points. Figure 7.11.b shows the model after five seconds of deformation. The model had 787 points at this moment. The velocities for the second stage were applied at this moment. Figure 7.11.c shows the model after five more seconds of deformation. The model had 1026 points then. Finally, Figure 7.11.d shows the polygonal mesh obtained from the model, the polygonal mesh was rendered with an apple texture.

The apple model uses more deformations than the previous ones yet still remains simple to create. A couple of interesting features of the apple model are its bottom and top. The bottom is a combination of stretching (bottom bumps) and compressing (bottom center) deformations, showing that such combinations pose no complication other than their proper design. The top of the apple shows how a model can become asymmetric by simply applying the same deformation with different parameters on different places.

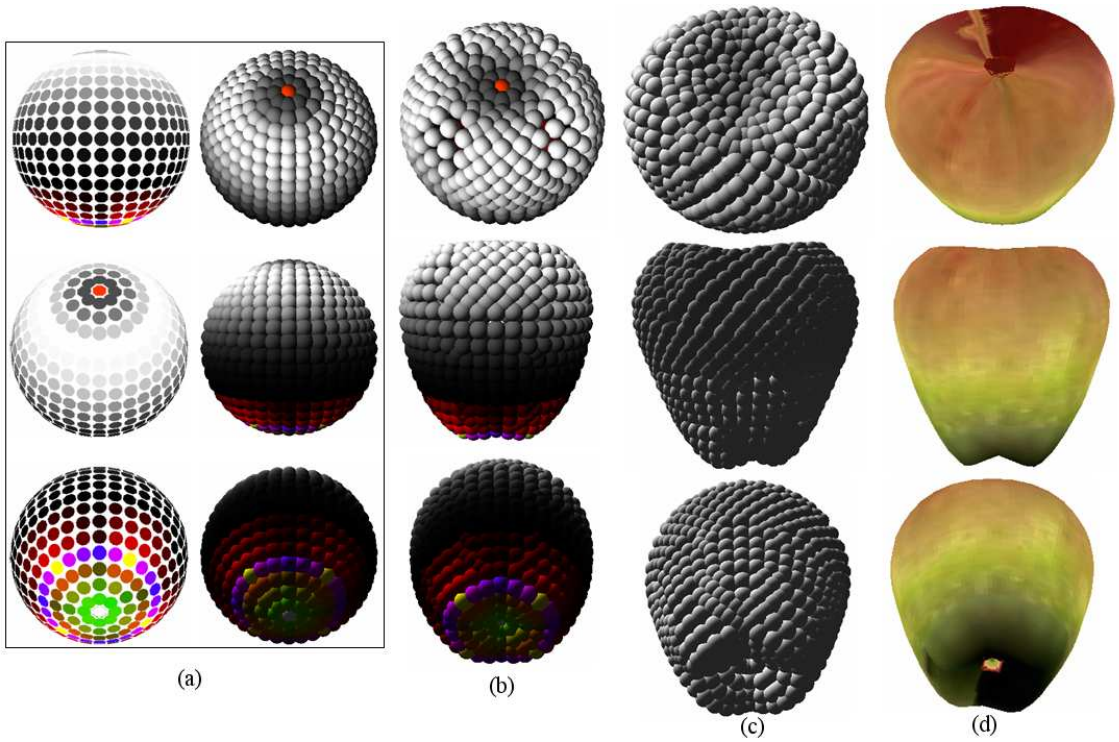

Figure 7.11: Apple model at different stages of its creation.

7.2 Model Parameterization

In this section, I demonstrate how varied models can easily be obtained by manipulating existing deformations. The models in this section were created with the same steps as those in the previous section. However, the deformation parameters and deformation times were slightly changed, thus producing models with different shapes.

The manipulation of deformations consisted of two types: manipulating shape-related deformations and manipulating parameters of deformation operators. The first type involves changing the deformation times as well as the parameters of the shape-related deformations. For example, the time of the first deformation in creating a tomato model can be shortened or prolonged to make the tomato model more or less spherical respectively. Similarly, the time of the deformation that creates the upper bumps of the tomato can be modified so they will be more pronounced. On the other hand, the second type of manipulation involves modifying the parameters of the deformations. For example, the **ScaledUnitaryGaussian** used to create the bumps in a tomato model can be modified to make them more or less thick.

Table 7.1: Parameters for creating model of Figure 7.12

	Figure 7.12							
	(a)		(b)		(c)		(d)	
Model	# points	Time	Parameter	Time	Parameter	Time	# points	Time
Upper	899	0 s.	$\sigma = 0.5$	10 s.	As original	20 s.	2489	21 s.
Lower	899	0 s.	$\sigma = 2.25$	10 s.	As original	20 s.	1652	21 s.

7.2.1 Models Parameterized

Next I show different models obtained by manipulating the initial deformations as well as the parameters of the deformation operators involved in their creation.

The first model variation is for the tomato model. Figure 7.12 shows two different tomato models. Figure 7.12.a shows the initial sphere of 899 points; the points colors intensities are directly proportional to the initial velocities. The upper half of the sphere was deformed with the operator **ScaledUnitaryGaussian**, and the only parameter modified from to the original tomato was the standard deviation σ (see Section 7.1.1). The upper model has a standard deviation σ of 0.5 while the lower model has a standard deviation σ of 2.25.

Figure 7.12.b shows the models after ten seconds of deformation. At this moment, the deformation were paused and the second stage deformations were applied (bump generation). The results of this deformation after ten more seconds are shown in Figure 7.12.c. Finally, Figure 7.12.d shows the polygonal meshes obtained after translating from Volipoc; the meshes use the same tomato-like texture as the original model. Table 7.1 summarizes the aforementioned information.

Modifying the parameters demonstrated that tomatoes with varied shaped could be easily obtained, simply by modifying one parameter. The two models shown in Figure 7.12 have very different shapes. The first had its upper half grow into a big concave region while the bumps produced variations on the concavity's wall. The second model was an almost spherical tomato with bumps barely visible on its upper half.

The modified tomato models show an important feature of my scheme: mass production with variation. The tomato models were created with exactly the same steps as the original one; however, very different models were obtained by simply changing some parameters. This is a very powerful feature of my scheme because it greatly supports the goal of supporting the population of virtual worlds with rich and varied models, yet still discernable to belong to the same class.

The second model variation is for the pear model. For this model I modified two of the parameters in the deformation operator instead of only one. Table 7.2 shows the information related to the creation of the models shown in Figure 7.13.

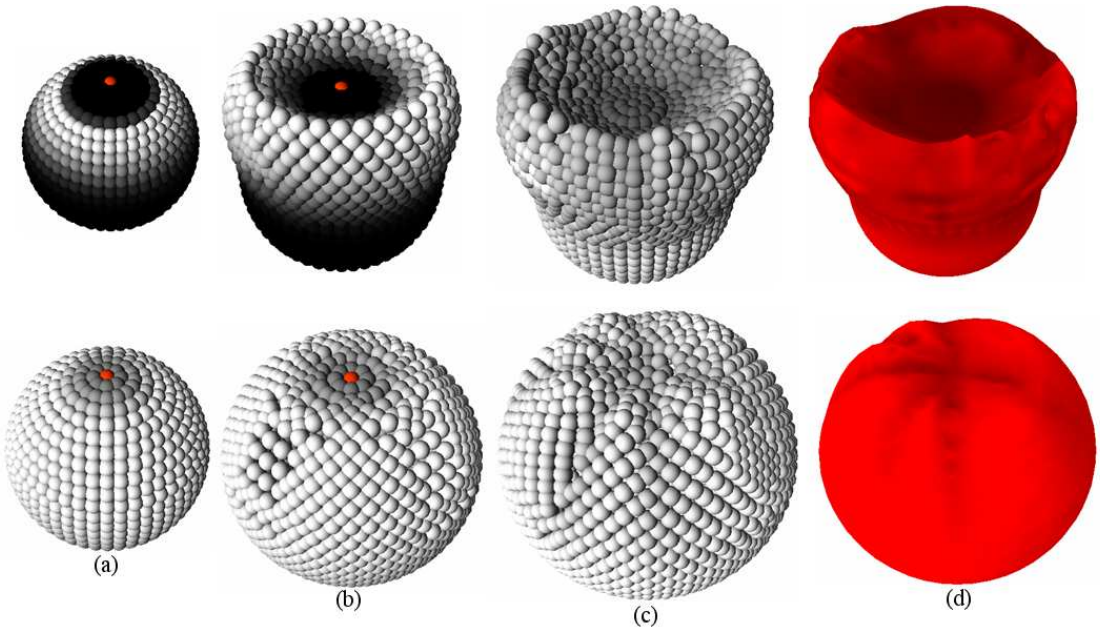

Figure 7.12: A couple of tomato models with generated with different parameters.

Table 7.2: Parameters for creating model of Figure 7.13

	Figure 7.13							
	(a)		(b)		(c)		(d)	
Model	# points	Time	Parameter	Time	Parameter	Time	# points	Time
Upper	899	0 s.	σ=5.0	10 s.	As original	20 s.	2437	21 s.
Lower	899	0 s.	σ=2.0	10 s.	As original	20 s.	2872	21 s.

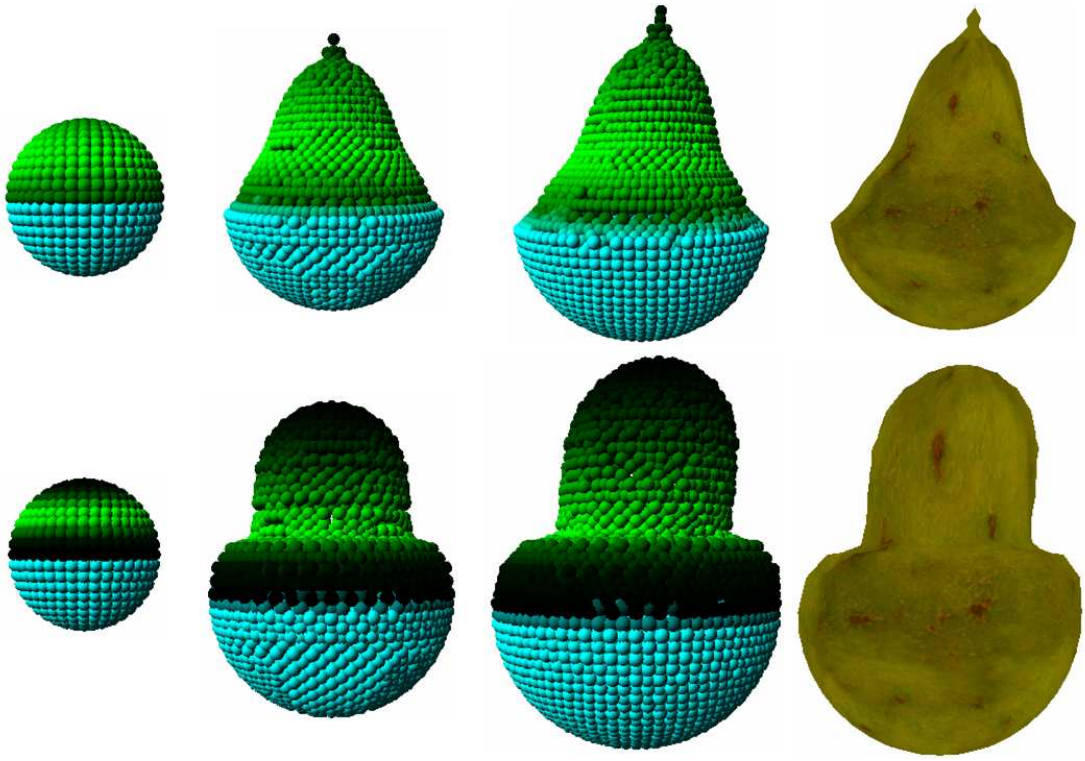

Figure 7.13: A couple of pear models with generated with different parameters.

Table 7.3: Parameters for creating model of Figure 7.14

	Figure 7.14							
	(a)		(b)		(c)		(d)	
Model	# points	Time	Parameter	Time	Parameter	Time	# points	Time
Upper	899	0 s.	$\sigma = 1.0$	10 s.	As original	20 s.	1561	21 s.
Lower	899	0 s.	$\sigma = 4.0$	10 s.	As original	20 s.	1805	21 s.

The pear model demonstrate an important feature of my scheme. Almost the same time is required to generate the varied models of the same class.

The third model variation is for the strawberry model. Table 7.3 shows the information related to the creation of the models shown in Figure 7.14. The manipulation of parameters allowed me to create strawberry models with different tops. The first strawberry had a top not proportional to the rest of the strawberry. The second strawberry had the shape of a compressed egg.

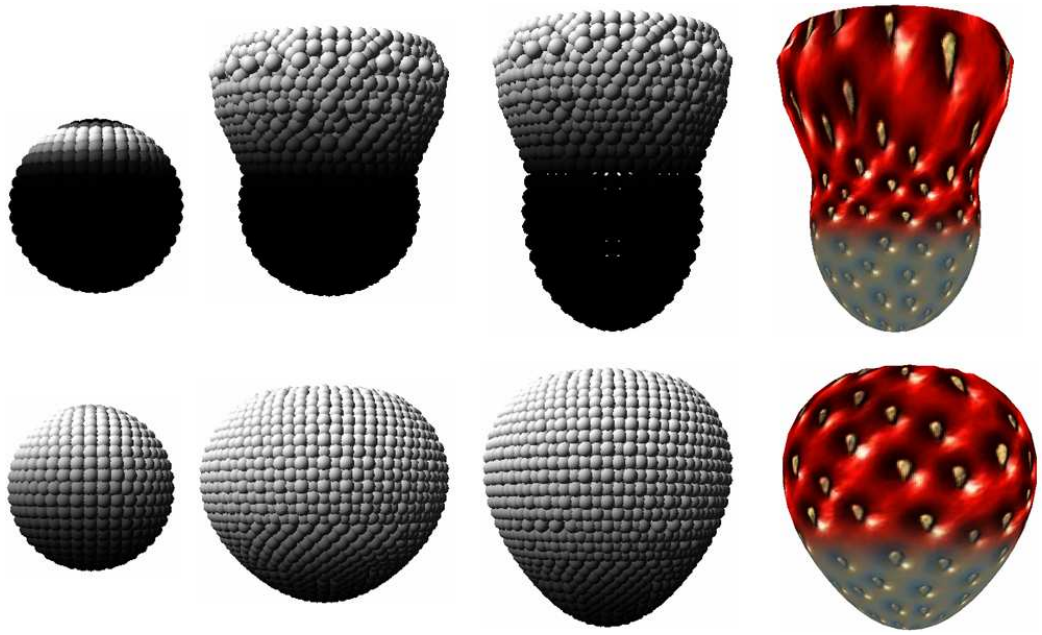

Figure 7.14: A couple of strawberry models with generated with different parameters.

The parameterized strawberries illustrate an important feature: the surface details (texture) grow according to the deformation. For example, the top of the first strawberry shows larger seeds than those of second strawberry; this is an expected behavior because, after all, the top of the first strawberry grew more than that of the second. Even though this is not what happens in nature, it

Table 7.4: Parameters for creating model of Figure 7.15

	Figure 7.15							
	(a)		(b)		(c)		(d)	
Model	# points	Time	Curve Parameter	Time	Parameter	Time	# points	Time
Upper	246	0 s.	H = 1.0, L = 0.5	3 s.	As original	15 s.	1698	21 s.
Lower	256	0 s.	H = 0.0, L = 4.0	3 s.	As original	15 s.	1446	21 s.

is correct for my scheme since my interest is to preserve the surface details despite the deformations, and not to simulate a particular natural phenomenon.

The fourth model variation is for the banana model. Table 7.4 shows the information related to the creation of the models shown in Figure 7.15. The banana models I created showed different angles at which the banana elongated.

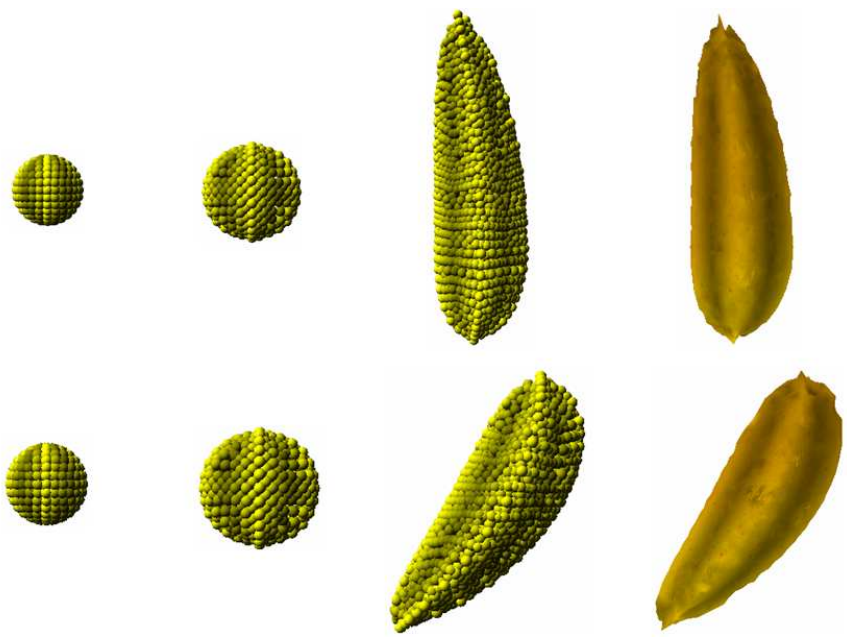

Figure 7.15: A couple of banana models generated with different parameters.

The fourth model variation is for the apple model. Table 7.5 shows the information related to the creation of the models shown in Figure 7.16. The apple models showed different shapes on their top. The first had a top grown faster than the rest of the apple thus generating an apple with a waist. The second had a top grown almost uniformly thus generating a fat apple.

The parameterized models illustrate an interesting feature: modified parameters can have similar effects on models sharing the similar deformations. For example, note how the first and second

Table 7.5: Parameters for creating model of Figure 7.16

	Figure 7.16							
	(a)		(b)		(c)		(d)	
Model	# points	Time	Parameter	Time	Parameter	Time	# points	Time
Upper	246	0 s.	$\sigma = 1.0$	10 s.	As original	20 s.	1295	21 s.
Lower	256	0 s.	$\sigma = 6.0$	10 s.	As original	20 s.	2289	21 s.

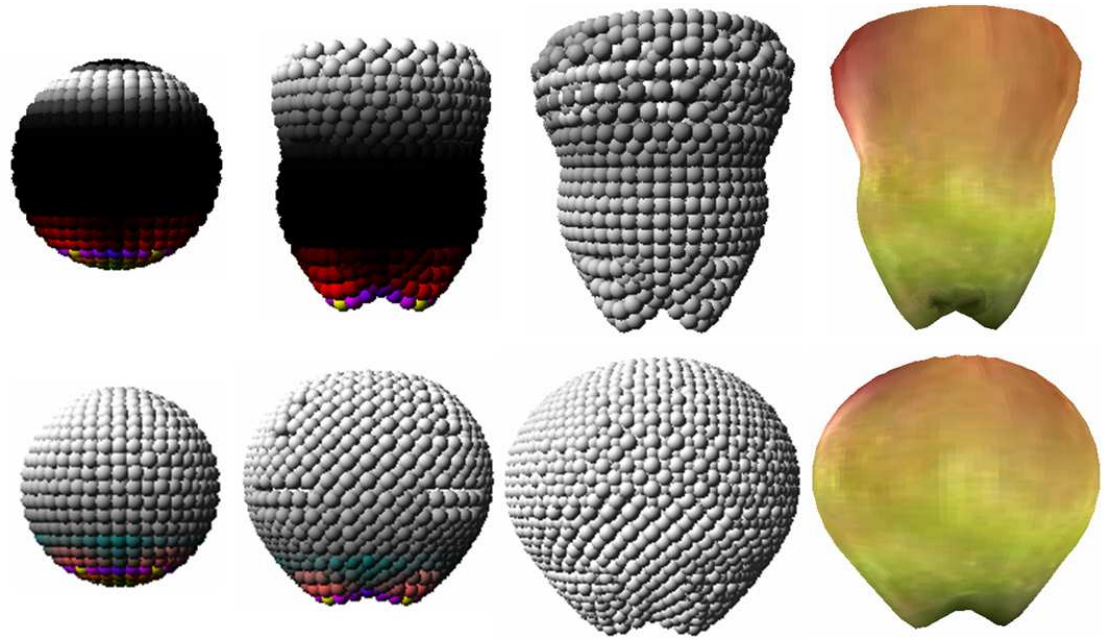

Figure 7.16: A couple of apple models with generated with different parameters.

apple have very similar shapes to those of the first and second parameterized strawberries. This could be an interesting feature from the artistic point of view.

Modifying the parameters to create the new models did not take a lot of time. I spent more time in fine-tuning the parameters to achieve certain aesthetical view rather than growing the models. This was a positive occurrence because it satisfied one of the research goals: quickly generating models so that a user/designer can exploit his/her creativity. The models shown here demonstrate how easily varied models can be obtained with my Automatic Model Creation framework. In the next section I demonstrate how the deformation operators also facilitate the creation of varied models by creating rich details on their surfaces.

7.2.2 Models with Parameterized Operators

Next I show different models obtained by applying deformation operators. The operators were applied directly on the surface of pre-created models. These models are the same as those in Section 7.1. For each model, different operators with different parameters were used to obtain varied models.

The first model I created was a tomato with a couple of thorns. This model was created using two Thorn operators. Figure 7.17 shows the model I created. I applied the Thorn operator on the sides of the tomato. The parameters I used were: Radius = 6, Thinness = 2, and Scale = 1. The trend vector of the thorns were: (-1,0,0) and (+1,0,0), for the left and right thorns respectively. Figure 7.17 shows the thorned tomato I created. The left image is a snapshot of the tomato while rendered with spheres (see Section 6.2.3). The right image is the polygonal mesh translated from Volipoc.

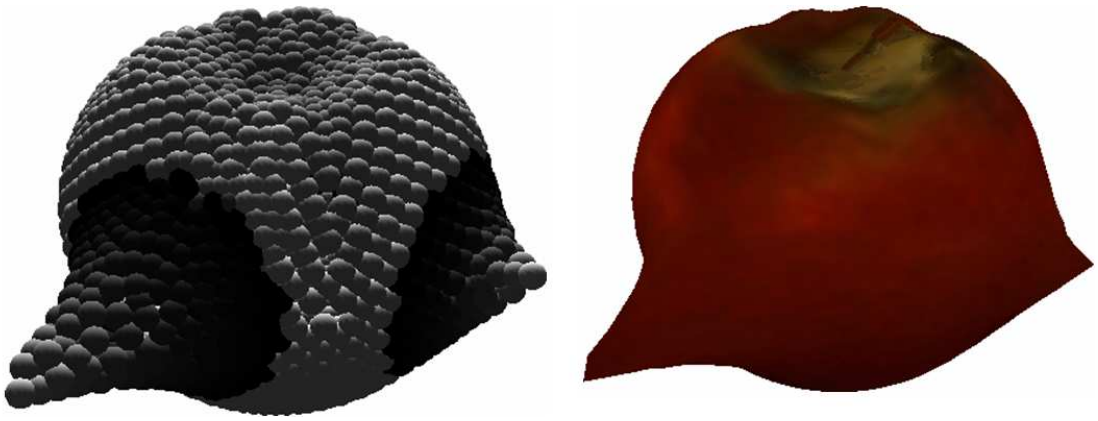

Figure 7.17: Tomato with thorns.

The thorned tomato is the first model that illustrates how deformation operators can be used to expand a model from its original conception. The thorns of the tomato have given it a very

specific characterization of its surface, yet it is still identifiable as belonging to the tomato class. Note that the effort to create such model has been minimal having the original tomato sequence.

The next model I created was a deformed tomato. I used two Cratering and several Thorn operators to create several protuberances and holes on the tomato's surface. I randomly applied the operators on the surface. The Cratering parameters were: Radius = 4, Radius Variation = 30, Base Segments = 8, Number of Peaks = 16, Peak's Height = 1, Peak's Asymmetry = 2, and Peak's Radius = 1.8. The Thorn parameters were: Radius = 3, Thinness = 2, Scale = 1, and Trend = Point's normal. I left the tomato to deform for 20 seconds.

Figure 7.17 shows snapshots of the deformed tomato. The first row shows the tomato spinning around its center. The tomato is rendered with spheres in the first row. The second row shows the polygonal mesh after the tomato was translated from Volipoc.

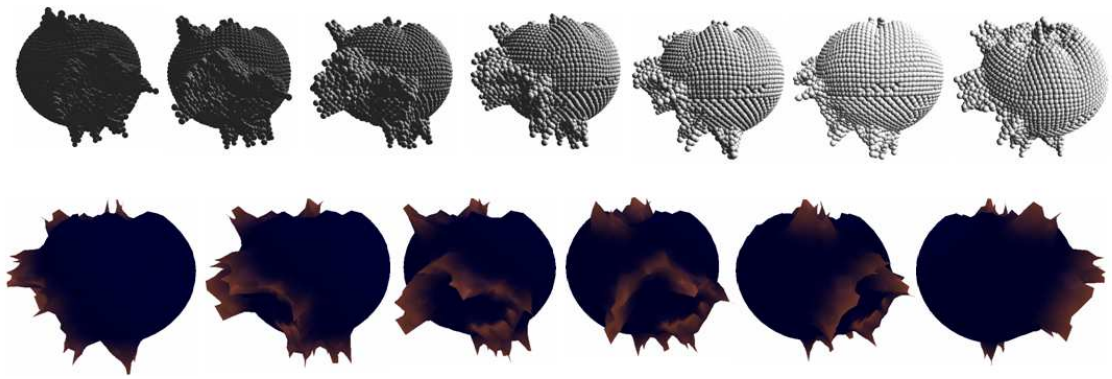

Figure 7.18: Deformed tomato model.

The next operator I used on the tomato model was the Roughening operator. The parameters were: Initial Regions = 10, Multiplier = 2, and Bump's Height = 1. I left the model to deform for 20 seconds. The resulting model is shown in Figure 7.19. The model is sphere-rendered. One can easily appreciate the bumps generated by the Roughening operator.

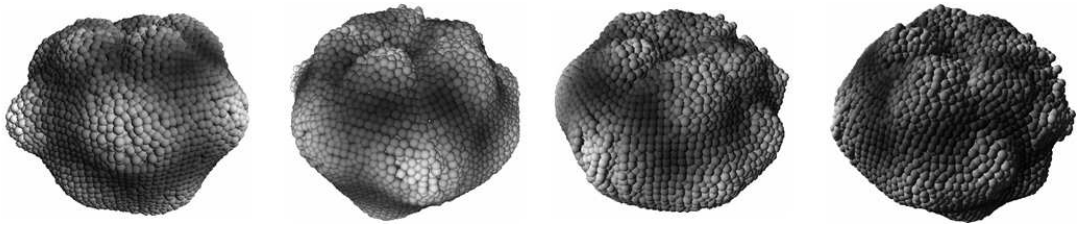

Figure 7.19: Roughened tomato.

The next operator I used on the tomato model was the Cracking operator. I was able to create tomato with missing chunks on its surface. I manipulated the length and width of the crack in

Table 7.6: Parameters for creating model of Figure 7.20

Parameters for model in Figure 7.20		
Cracking Threshold	Operator Radius	Max Crack Width
0.55	20	5
0.35	20	4
0.25	20	3
0.15	20	2

order to created regions that sank into the model. Figure 7.20 a tomato model I created with four Cracking operators. The parameters I used are listed in table 7.6. Figure 7.21 shows the polygonal mesh after the model has been translated from Volipoc. These images show what I consider to be a model that looks like a partially rotten tomato.

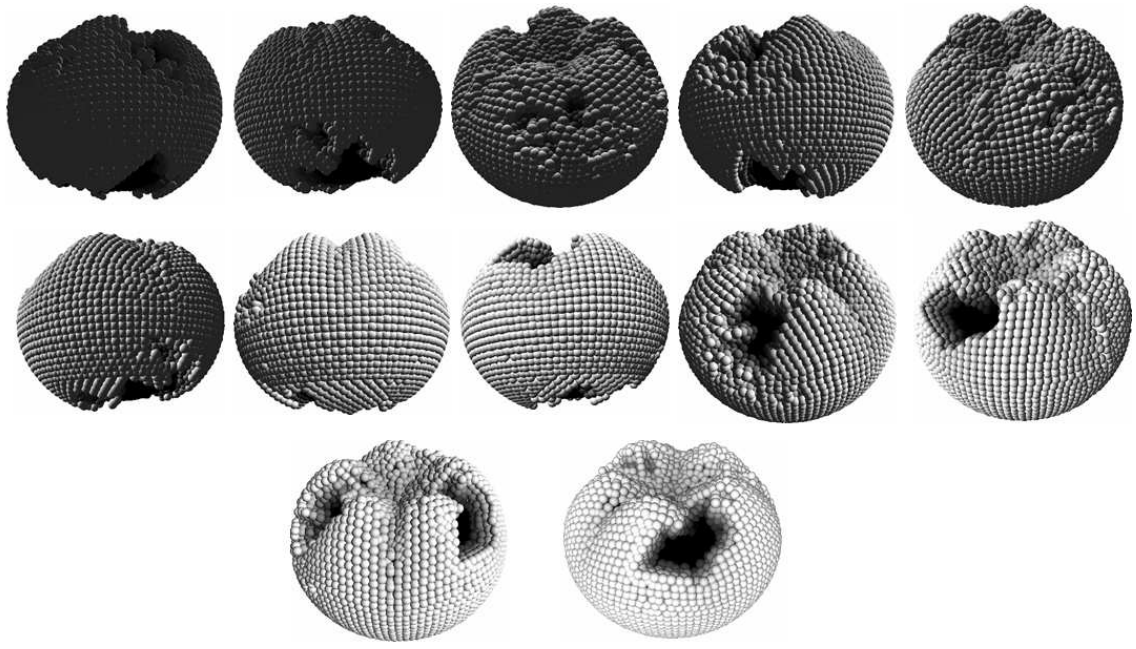

Figure 7.20: Tomato with cracks and sunk regions - sphere rendered.

The tomato models generated with the Thorn, Cracking, and Roughening operators illustrate how simple varied models can be obtained from a known sequence of deformations. Note that the deformation operators can be arbitrarily reused with other types of models without any complication.

The next model I modified with deformation operators was the pear. Figure 7.22 shows snapshots of a pear model I created by applying four Thorn operators. Table 7.7 contains the parameters

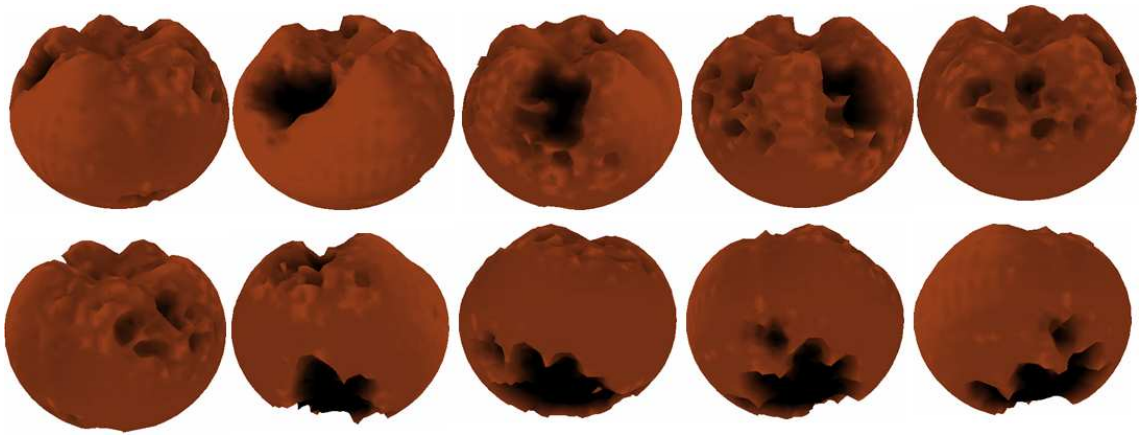

Figure 7.21: Polygonal mesh of the tomato model shown in Figure 7.20.

Table 7.7: Parameters for creating model of Figure 7.22

Parameters for model in Figure 7.22			
Radius	Thinness	Scale	Trend
4	1.5	1	(-1,-0.25,1)
4	1.5	1	(-1,-0.25,1)
4	1.5	1	(-1,-0.25,-1)
4	1.5	1	(-1,-0.75,0)

I used. I applied the thorns around the lower half of the pear. The thorns grew towards a similar direction (controlled with the Trend parameter).

The thorned pear illustrates a big potential of my scheme. Note how the thorns seem to have certain tendency to grow in a specific direction. This opens the door to visualize the effects of tropism on model generation by simply manipulating deformation parameters.

The next operator I used on the pear model was the Roughening operators. Figure 7.23 shows the roughened pear model. The operator parameters were: Initial Regions = 10, Multiplier = 2, and Bump's Height = 1. I let the model deform for fifteen seconds. I used this model as starting point for the next model.

I obtained the next model by applying Cracking to the roughened pear with one crack. Its parameters are: Cracking Threshold = 0.65, Operator Radius = 10, and Max Crack Width = 4.

The next model I created followed the inverse steps of the previous two models. That is, I first applied a Cracking operator and then applied a Roughening operator. I applied these operators to a strawberry model. The resulting model is shown in Figure 7.25.

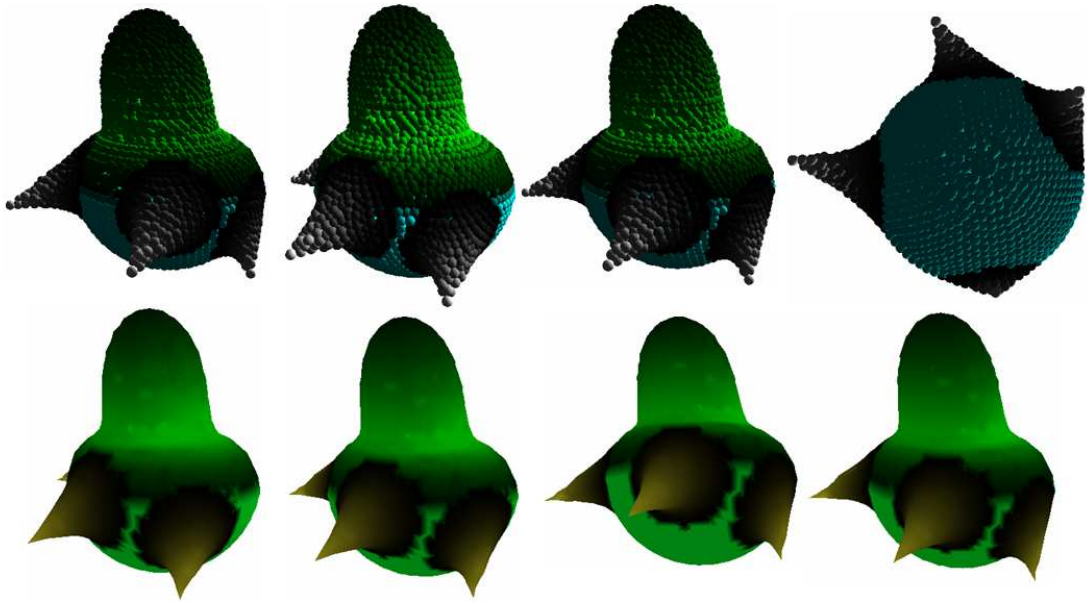

Figure 7.22: Pear with thorns.

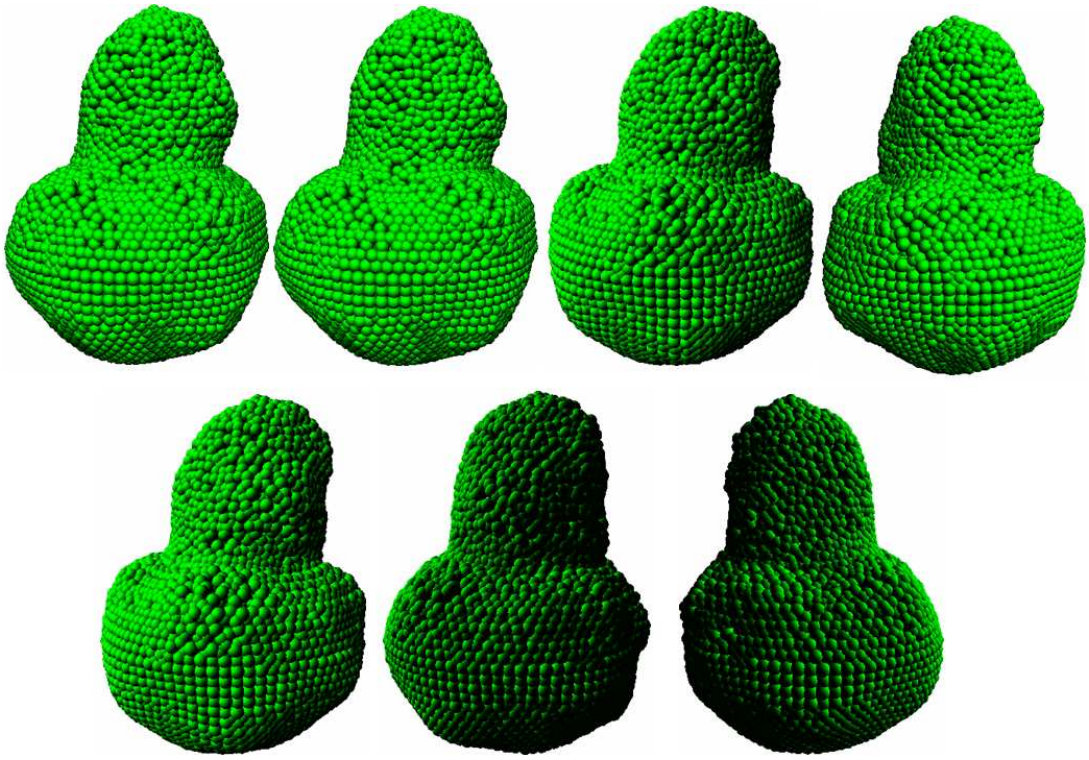

Figure 7.23: Roughened pear.

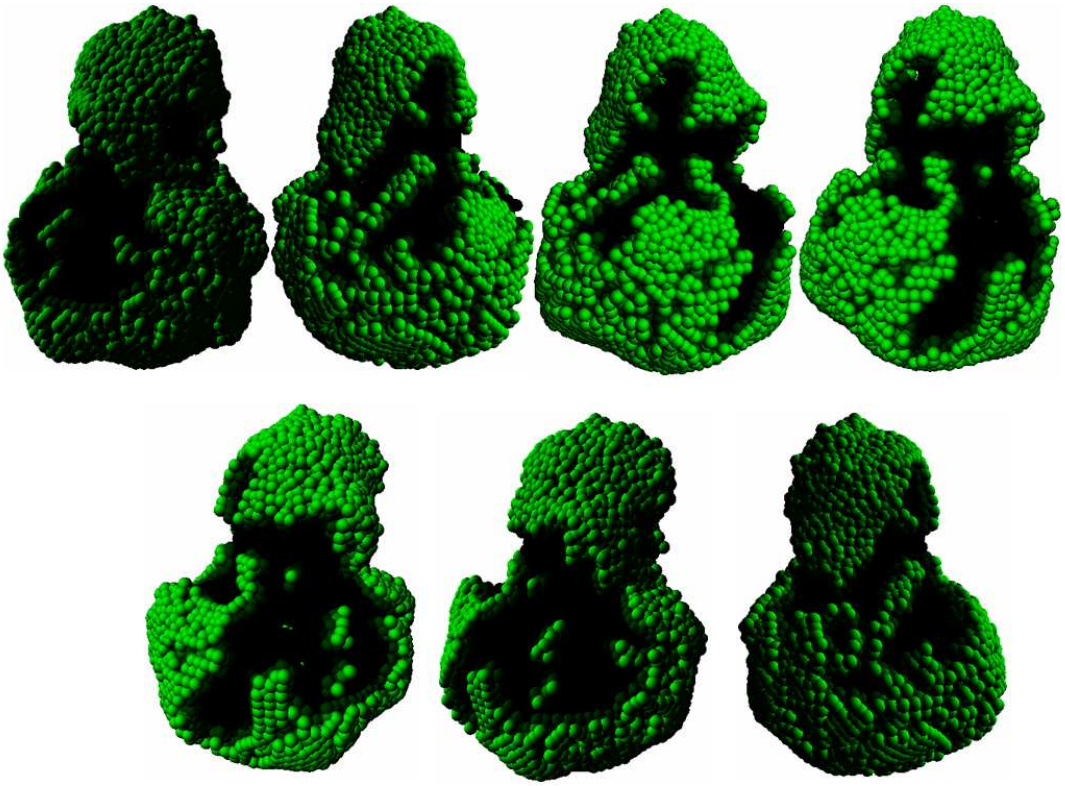

Figure 7.24: Cracks applied to roughened pear.

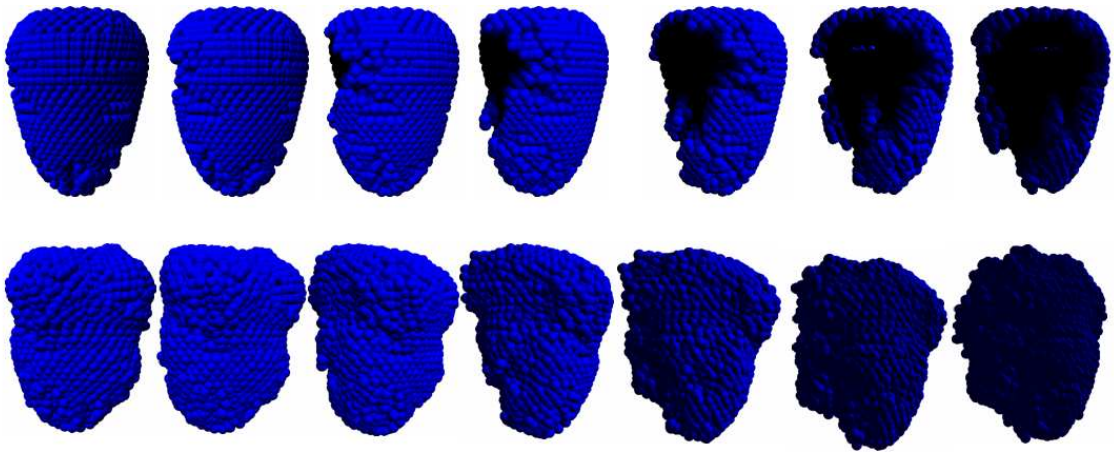

Figure 7.25: Strawberry model created by first applying a Cracking operator and then a Roughening.

As the last instance of this type of model (operators placed by hand), I created a thorned banana. Figure 7.26 shows the model I created. I applied several Thorn operators along the initial model. Figure 7.27 shows the same model rendered with spheres.

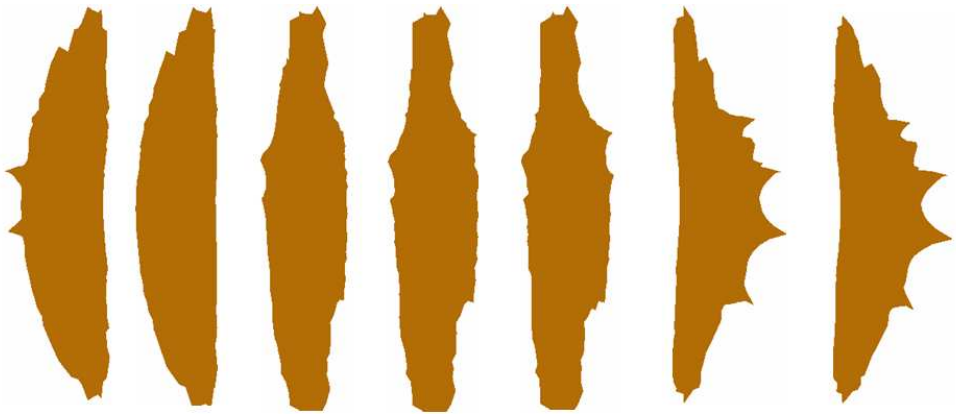

Figure 7.26: Banana with thorns.

In the next section I demonstrate how the deformation operators also facilitate the creation of varied models by creating rich details on their surfaces and by automatizing the operators applications.

7.3 Automatically Generated Models

In this section I show automatically created models: that is, models created with minimal user interaction, using the Surface Evolution Control scheme of my Automatic Model Creation framework.

The models I created were branching structures: models consisting of a stem from which several branches grow. The branches were created by assigning velocities to regions on the side of the stem. When these regions grew to a certain size (initial branch) then the branch is twisted upwards to a certain angle and the branch continued to grow. All is done by applying velocities to regions on the surface.

The steps to create the branching structure are several and may be complicated. The task of creating a branched structure may become repetitive and prone to error if done by hand, and hence is a good candidate for automation. Generating a branch is a very simple operation, repeated several times to create a branching structure. Furthermore, a simple manipulation of branching parameters (frequency, angle, length) can easily generate different types of models that are visually rich [18, 26, 53, 73, 74, 75].

The first step to create the branching structures was to design the Petri Net that would control

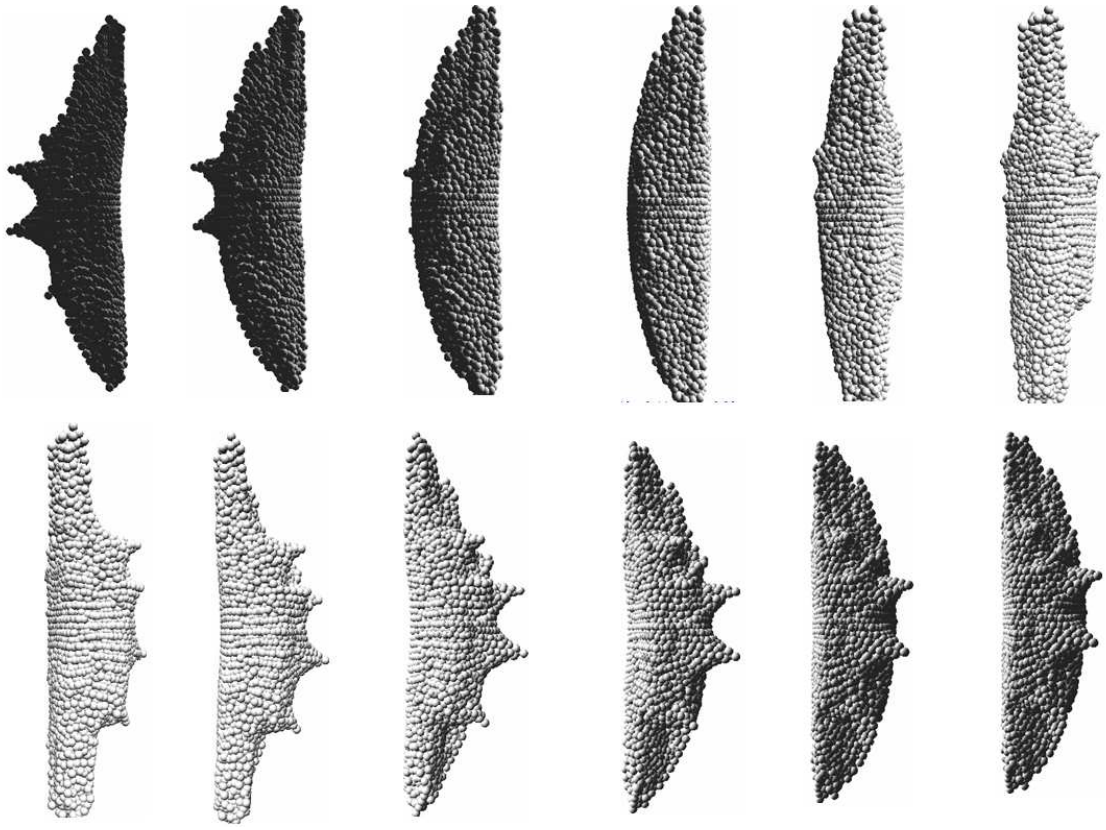

Figure 7.27: Banana with thorns - sphere rendering.

the generation of a single branch. Figure 7.28 shows such Petri net, note that only one token exists, initially placed in the state *Initial seed*. Transition T1 and the "Grow half sphere" state deform the model into a half-sphere, then the half-sphere subset of points advances all together in a specified direction. After $t2$ seconds of growing the first segment of the branch, the half-sphere subset stops and rotates upward around a pivot. Remember that the surface is resampled at each time step, maintaining the proper surface connectivity. Once the rotation has progressed a certain amount, the half-sphere recovers its velocity and continues growing the branch in the new direction for $t4$ seconds.

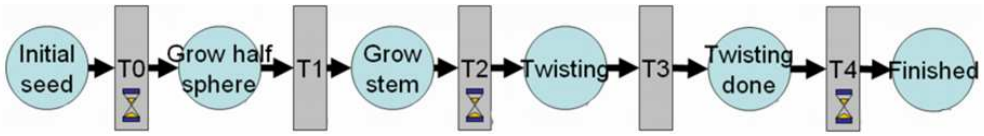

Figure 7.28: Petri Net for Branching Structure Generation.

Figure 7.29 shows a circular surface deforming by the previously described Petri Net. The stages of the branch creation are as follows: (a) Initial state, (b) Upper half grows upward, the blue surface element is chosen as the seed where the Petri Net will be applied, (c) First step in the "Grow half sphere" state, (d) Final step in the same state, (e) First step in "grow stem" state, (f) Final step, (g) Intermediate step in the "twisting" state, and (h) "Finished" state.

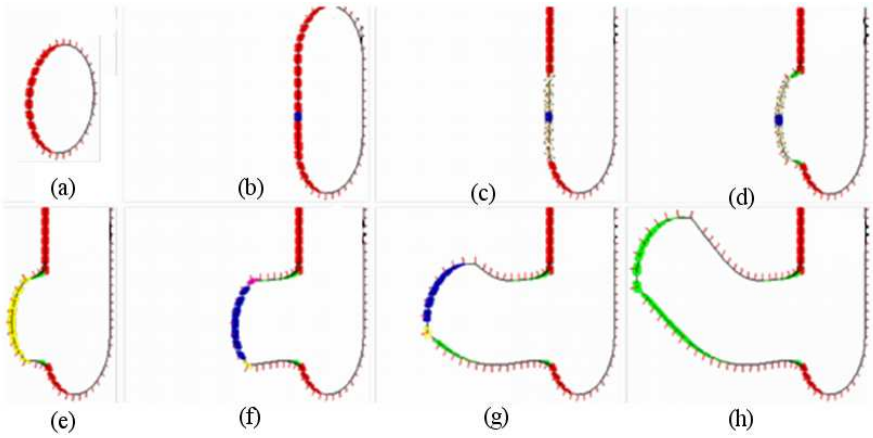

Figure 7.29: Example of a branch generated with Petri Net of Figure 7.28.

The generated branch may seem simple, but it demonstrates a very important aspect of my scheme: automation. Individual assignment and control of the surface points such that they create a branch would be a daunting task. However, it has become very simple when implemented as a Petri net. Note how the state-transition structure of the Petri net mimics the decision-based

process a user would follow. Additionally, property of deformations to be parameterized is naturally extended to the Petri nets, thus obtaining its benefits (replication with variation).

I applied the aforementioned Petri Net to a flat circular surface to demonstrate how it works. Figure 7.30 shows an initial surface. Figure 7.30.a shows the surface rendered with surfels and edges. In Figure 7.30.b the same surface is shown rotated 45 degrees. Figure 7.30.c shows additionally the normals of each point. Figure 7.30.d changes from surfel to sphere rendering. Finally, Figure 7.30.e combines sphere rendering and normal rendering, with the surface in its original position. I used this surface as the initial surface to create the branching structure.

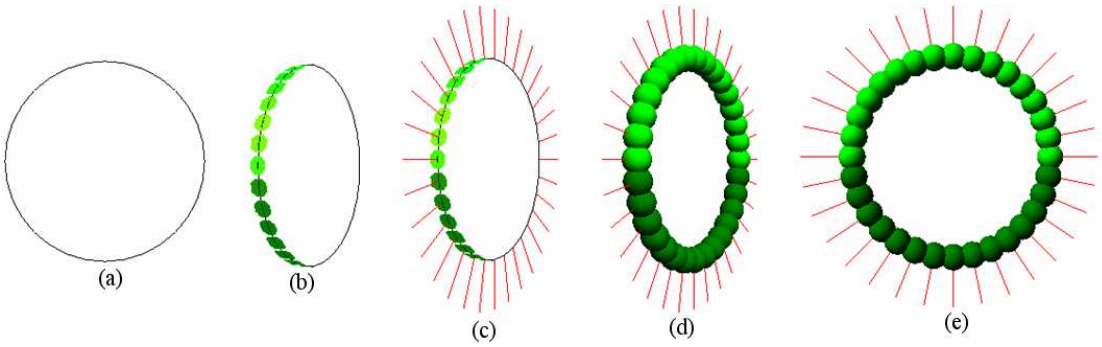

Figure 7.30: Initial surface to create a branching structure.

The branching structure required an initial stem. The stem was created by applying a constant upward velocity to the top half of the initial surface (light green points in Figure 7.30). Once the stem starts to grow, Petri Nets are placed randomly along the edges of the stem. Once a Petri Net has been allocated it starts functioning, thus making a branch grow. The parameters of the Petri Net (transition-triggering time, growing time, and twisting angle) were randomly chosen from given ranges. The previous branch-generation process is reflected in Figure 7.31. Figure 7.32 shows snapshot of the final branched structure rotated vertically.

The creation of branching structures, cactus-like, was a success with a flat initial surface (Figure 7.30). The next step was to create three-dimensional branching structures. To do this was as simple as changing the initial surface from a flat circle to a sphere. Figure 7.33 shows a three-dimensional branching structure obtained from an initial sphere.

These automatically generated models are the epitome of my scheme. They have been produced with minimal user interaction. They have a complex structure that is easily manipulated by a user through a set of simple parameters. Finally, they can be mass produced and include variation for each instance, but still be identifiable as belonging to a particular class.

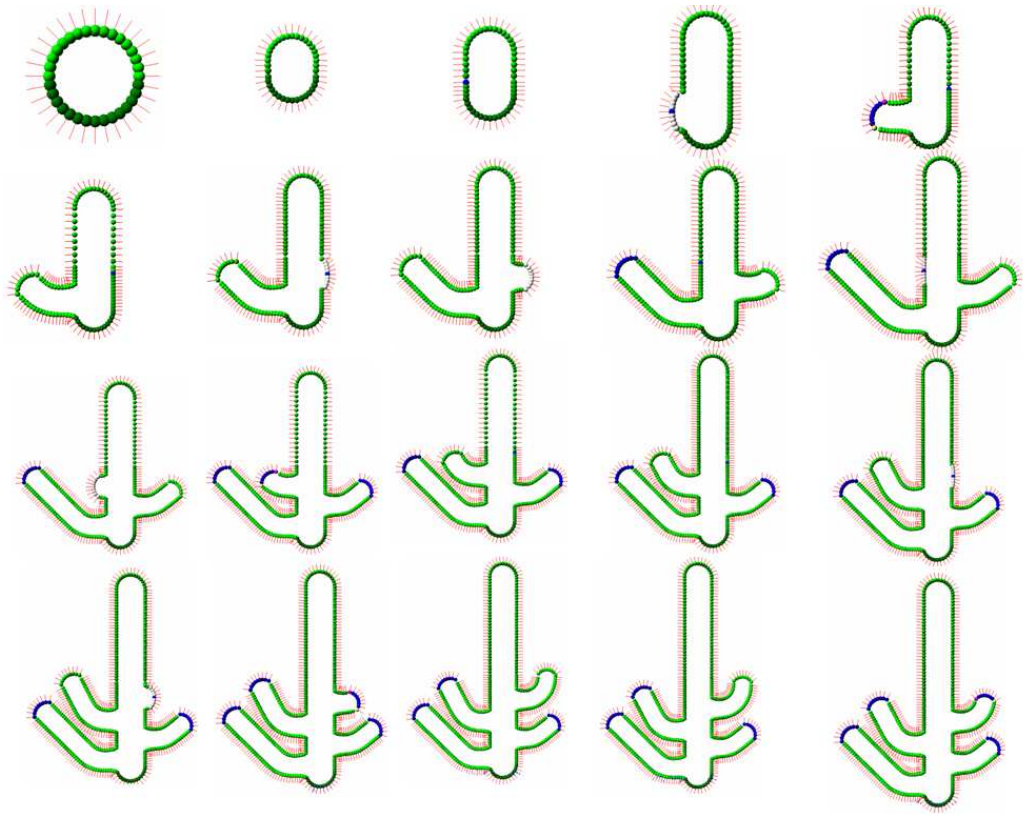

Figure 7.31: Steps of the automatic generation of a branching structure.

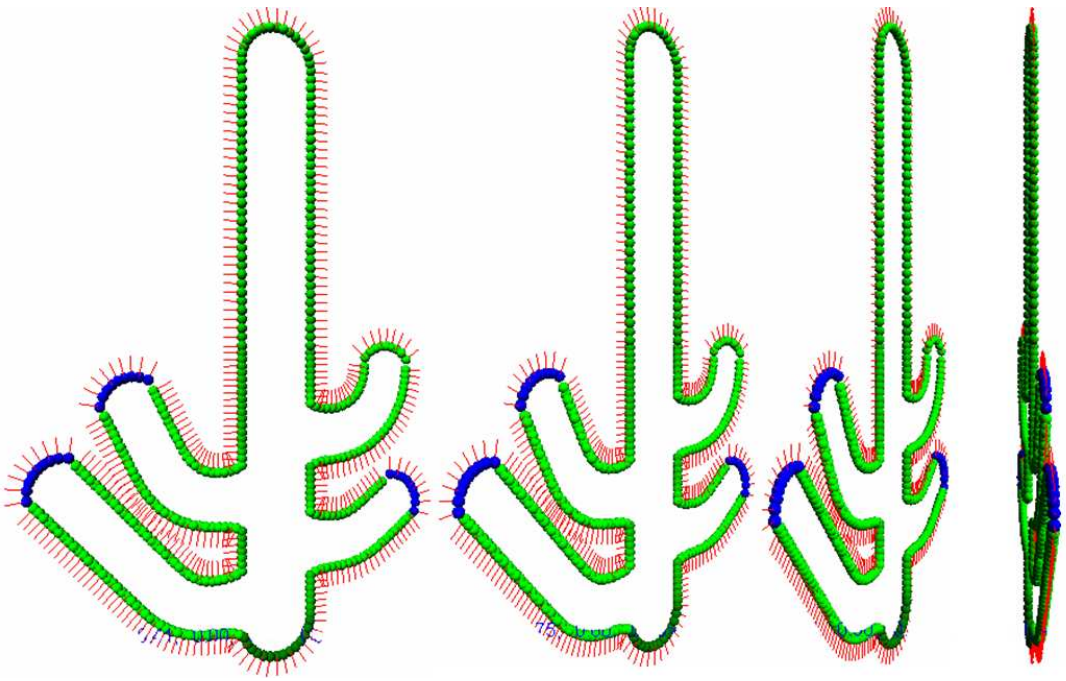

Figure 7.32: Images of final branched structure at different angles.

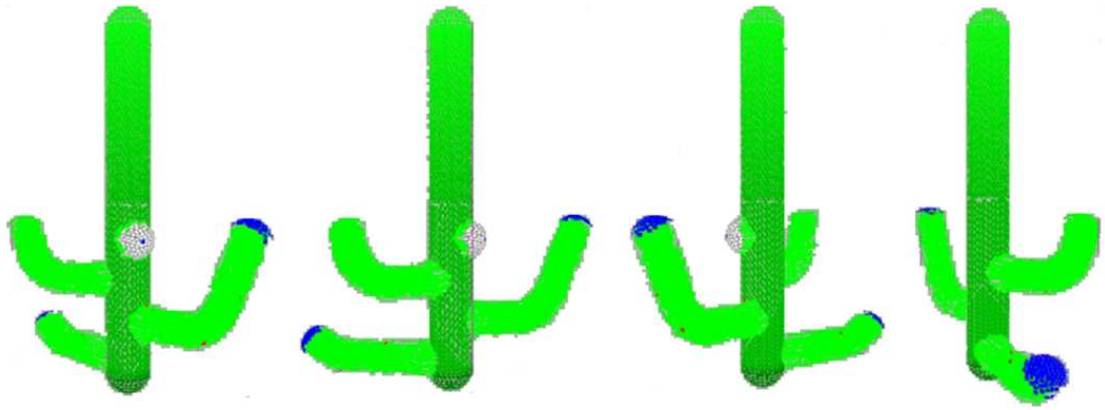

Figure 7.33: Branching structure obtained from an initial sphere.

7.4 Artistic Models

In this section I present what I consider to be artistic models. These were made to illustrate the flexibility of my Automatic Model Creation and to show the variety of models that can be created. The models were created with different combinations of deformation operators and velocities assignment. The models created are: mushrooms, asteroids, sick apples, pyramids, mushroom clouds, and elaborated mushrooms. Some of the models' visual appeal was enhanced by adding detail to their surfaces.

7.4.1 Mushroom-like Models

Next are the steps to create a mushroom-like model. The model starts as a simple sphere. Then a deformation is applied to the upper-half of the sphere. The deformation consists of velocities of the same intensity in the upward direction (see Figure 7.34.a). This deformation splits the sphere in two; the upper half moves upwards from the lower half. The next step is to expand the original upper-half of the sphere to create the cap of the mushroom. The cap is grown with a deformation that assigns a velocity in the same direction as the points normal. The velocity is inversely proportional to the vertical distance between the point and the cap base; that is, the maximum value of the velocity will be at the lowest points of the cap while the minimum will be at the highest points (see Figure 7.34.b).

Figure 7.35 shows snapshots of the mushroom model generated by Vebam. Figure 7.35.a shows the initial sphere of 606 points. After fifteen seconds of deformation the sphere has changed into a cylinder with rounded corners, as shown in Figure 7.35.b. After another fifteen seconds of deformation the sphere has changed into a mushroom of 4023 points.

The previous model can be expanded to create another type of mushroom. In Fig.7.36 I have

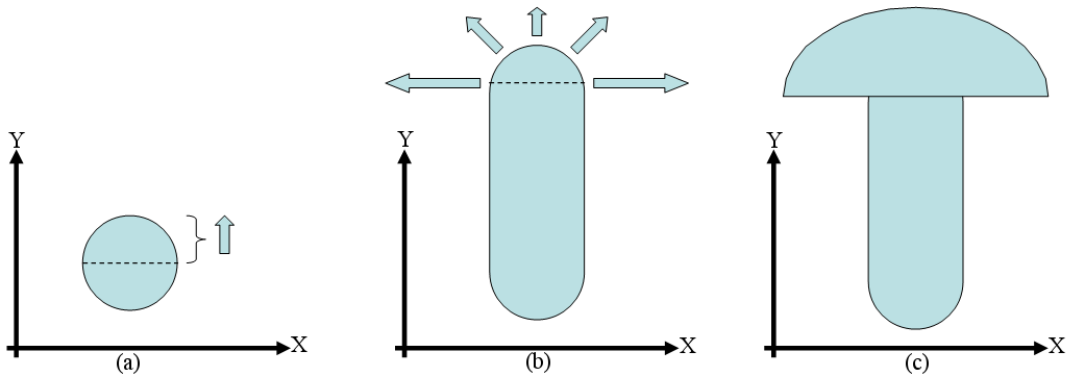

Figure 7.34: Mushroom model steps.

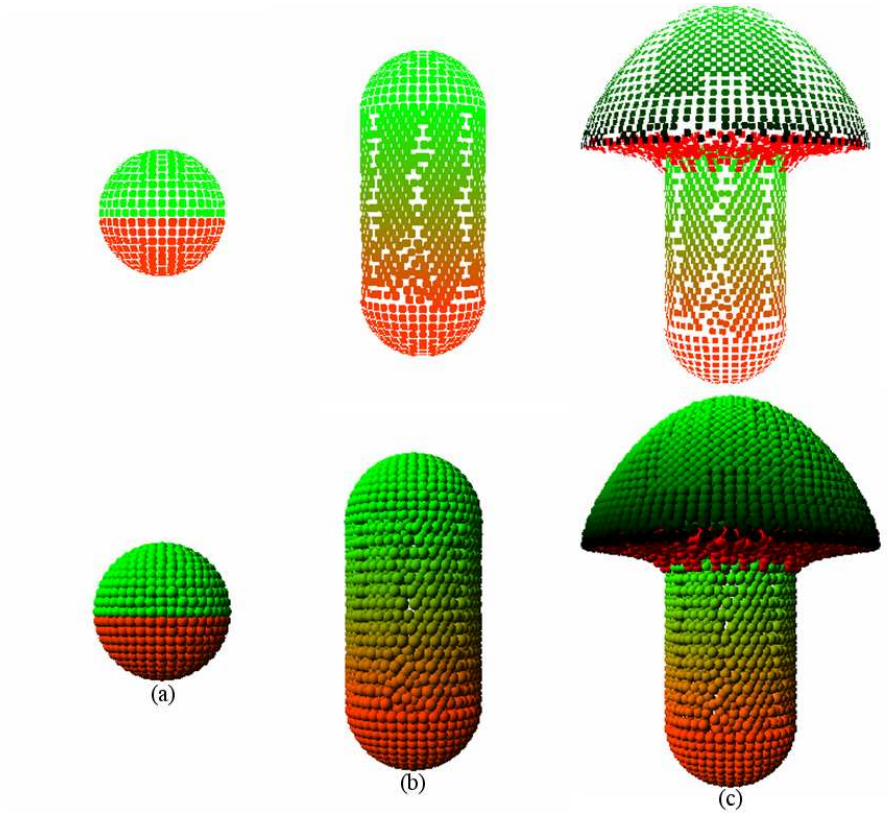

Figure 7.35: Mushroom model steps.

created the basic structure of a mushroom. I start by applying a uniform upward velocity to the upper half of the initial sphere (Fig.7.36.a), which transforms it into a cylinder with rounded edges (Fig.7.36.b). Then I radially apply a horizontal Gaussian velocity to those points on the boundary of the cylinder that are part of the original upper-half sphere; I also apply a Gaussian velocity distribution at the main top (Fig.7.36.c) transforming the original half-sphere into a mushroom-like cap. I then activate a gravity effect proportional to the distance of the point to the cylinder's vertical axis (Fig.7.36.e) allowing us to gradually pull down the cap (Fig.7.36.f,g). Figure 7.36.h shows the resulting polygonal mesh. Notice that no texture is applied in this and the next mushroom models; the models are rendered using colors whose intensities are proportional to the velocities or displacements of the rendered points.

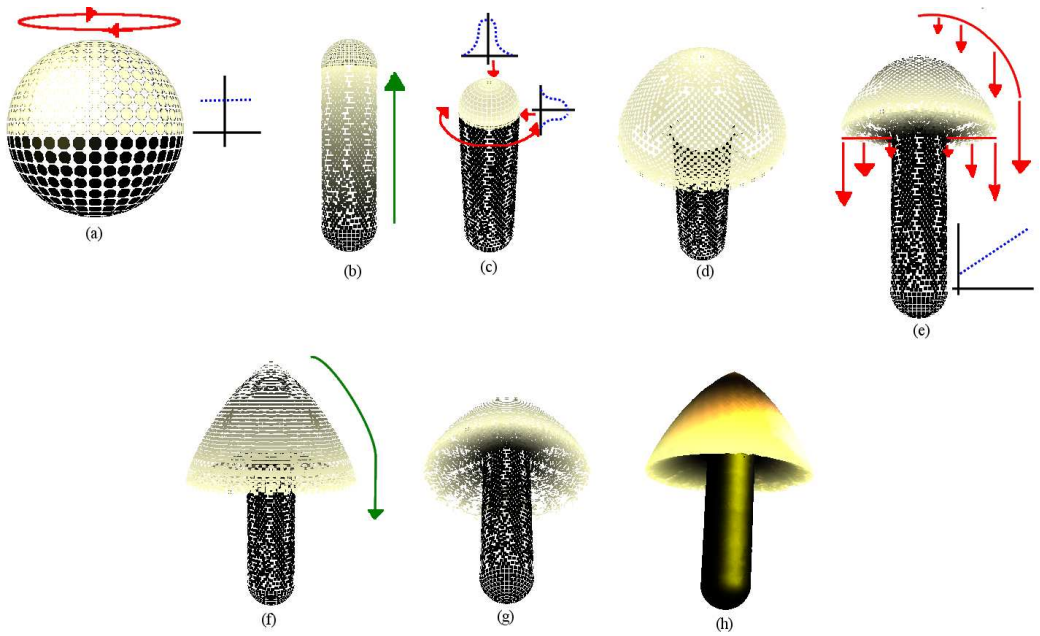

Figure 7.36: Steps to create a mushromm with a concave undercap.

Figures 7.37 and 7.38 show a couple of mushroom-like models generated from initial spheres deformed with simple Velocity Operators, Gaussian and the Voronoi operator. The first mushroom has an asymmetric cap. I achieved this effect by applying Gaussian operators on the edge of the cap. Each operator had different parameters.

The second mushroom had a symmetric cap (Figure 7.38). I achieved this effect by applying the same Gaussian operator at regula intervals. I obtained the cap detail of the second mushroom with the **VoronoiRegionalization** operator. I colored the points between regions as white and the others as black.

Finally, I made a more complex mushroom model, which is shown in Figure 7.39. Initially I

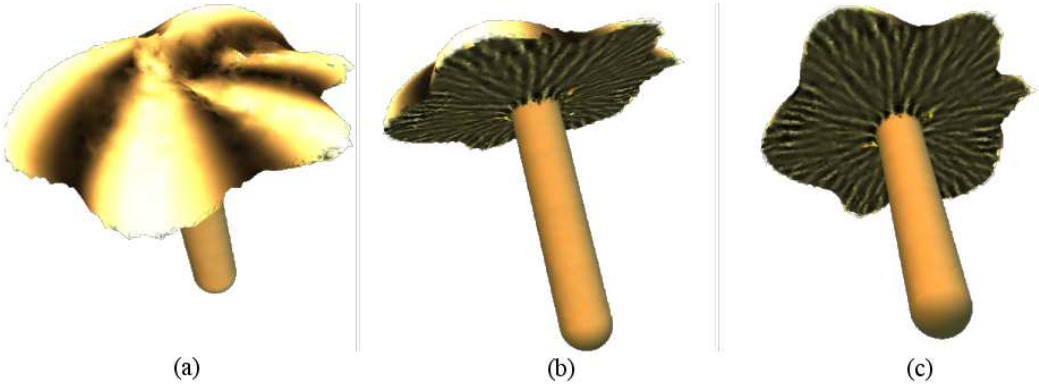

Figure 7.37: Mushroom with asymmetric cap.

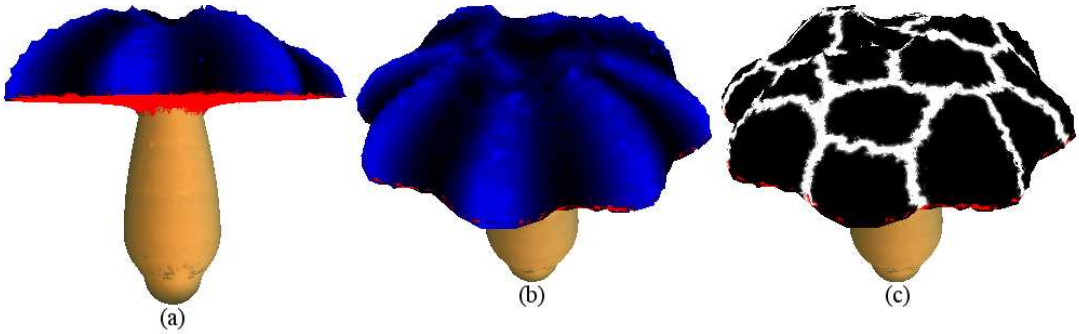

Figure 7.38: Mushroom with symmetric cap.

made a mushroom model with a stem and an egg-shaped cap, and then applied a **VoronoiRegionalization** operator. Then I made the points of the regions not neighboring other regions have a velocity whose direction is opposite to the points' normal. This made them move in toward the cap. The next step was to apply detail to the mushroom stem. Figure 7.39.a shows the result of the **VoronoiRegionalization** operator with initial seeds randomly placed on the stem's base. Figure 7.39.b and Figure 7.39.c show progressively the Voronoi region evolution. In Figure 7.39.d, the point's velocity (the point color intensities are proportional to their velocities) is a function of their distance to the center of their respective Voronoi region; if the velocity is greater than a certain threshold then the applied velocity is zero. Figures 7.39.e to 7.39.h show the obtained polygonal mesh rotated around the vertical axis, and 7.39.i is a closeup of the same polygonal mesh. Note the surface displacements due to the surface evolution.

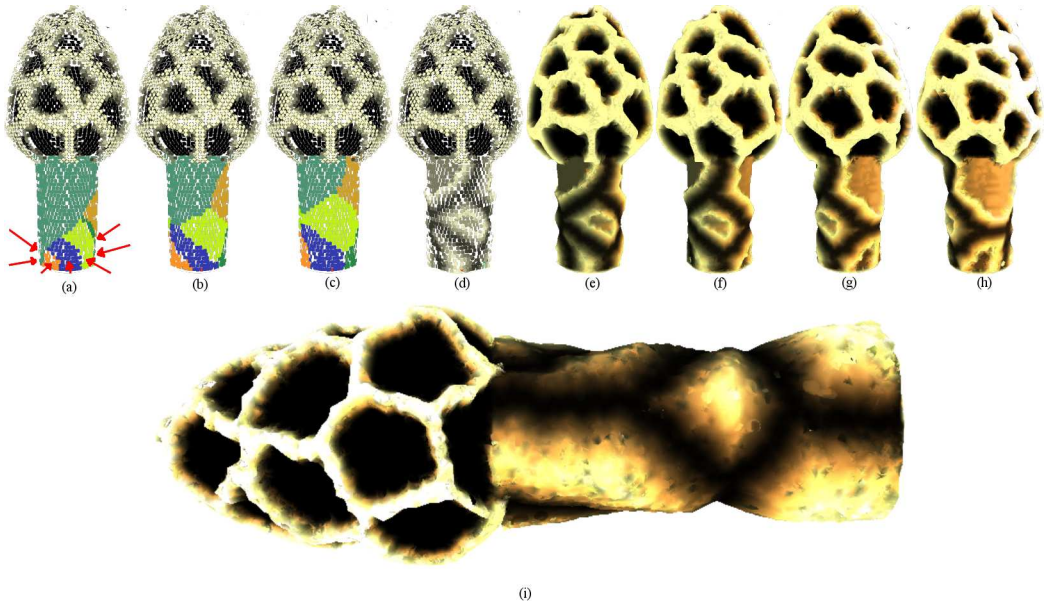

Figure 7.39: Mushroom with egg-shaped cap.

The mushrooms shown in this section are examples of instances of the same type of models, but with different deformation sequence. These type of models will be as similar as a user/designer wants them to be. These demonstrate that the sequence to create a type of model is not unique.

7.4.2 Asteroid-like Models

I created an asteroid-like model. The initial surface was a simple sphere. I made the sphere deform into an ellipsoid by assigning velocities to each half of the sphere. After that I made irregular bumps on the asteroid surface with the Roughening operator. Figure 7.40 shows the initial sphere (left) from which I created an asteroid (right). The initial sphere is made of 13314 points. The

final asteroid model has 31828 points. The total time for its creation was 41 minutes.

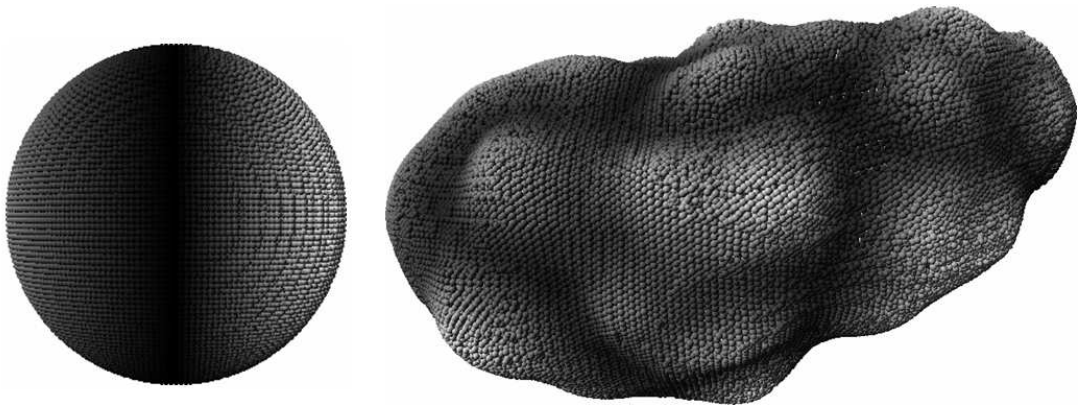

Figure 7.40: Asteroid model.

This model is an example of the scalability of the deformations. Despite having a lot more points than the pear model, the roughening operator has basically obtained the same effect on the model.

7.4.3 Sick Apple-like Models

I created a sick apple model with deformation already used in other models. I used the Cracking and Thorn operators to create the sick apple show in Figure 7.41. I used the Cracking operator to create the huge collapsed black regions on the apple. I used the Thorn operator to create the protuberances on the lower half of the apple.

The Sick Apple-like model shows the scalability of the deformation operators. Note that despite having a huge difference in the number of points, which has the same features as the apple shown earlier.

7.4.4 Pyramid-like Models

I created pyramid-like models also. I used these models to demonstrate that even though my Automatic Model Creation scheme is best suited for biological surfaces that grow (sharing surface features along regions) it can still be used for creating other types of models. Architectural structures are another type of model that can be created with my Automatic Model Creation framework. I chose this type because I consider it greatly different from biological surfaces that grow. There are two main differences: architectural models are not created by expanding or stretching surfaces and they do not necessarily present smooth transitions from region to region.

The process I followed to create architectural models was similar to displacement mapping. However, instead of using a scalar field to denote displacements on the surface, I used a scalar field

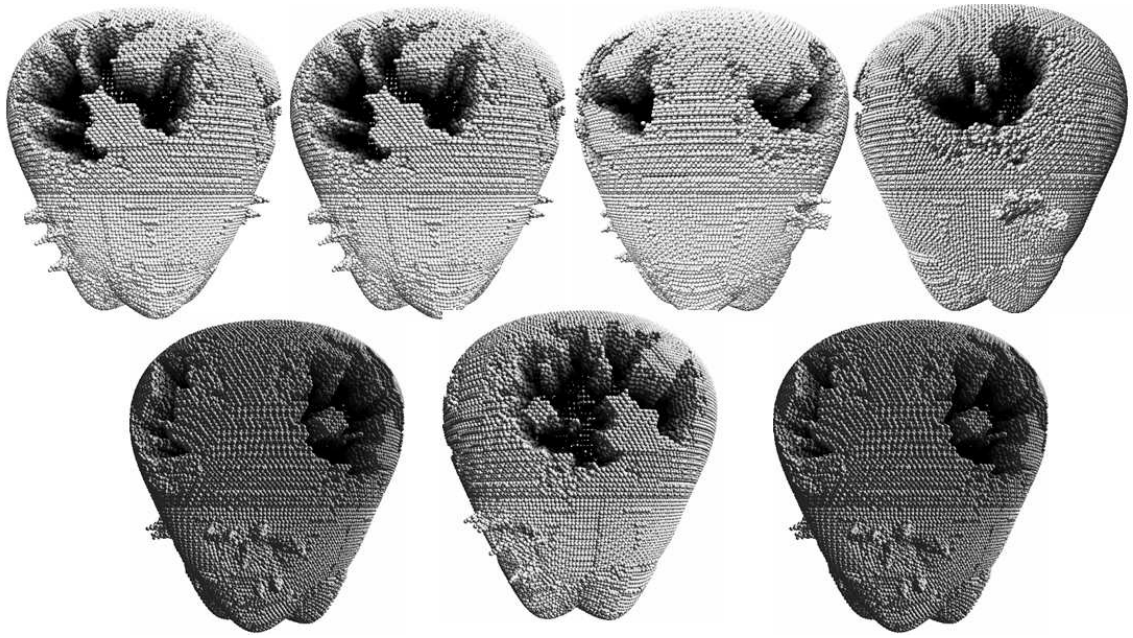

Figure 7.41: Sick apple.

to denote initial velocities on a surface. For example, Figure 7.42 (left) shows an initial flat surface whose points will grow to create a pyramid-like model. The points have been color-coded into regions, each created a distinctive feature of the pyramid. Figure 7.42 (right) shows the resulting model.

Figure 7.43 shows the polygonal mesh obtained from the model in Figure 7.42. Note that it is difficult to obtain perfect right angles between regions with different velocities. This happens because the boundaries between regions are stretched due to the difference on velocities. Therefore at some moment the boundary is resampled, thus creating new points with their own velocities. These points makes the difference between the boundary to get lost as more points are created during resampling. One way to avoid such inconvenience is to increase the number of points used, but this will not eliminate the problem but just reduce its visual impact.

Figure 7.44 shows a pyramid model with 30162 points. This model started as a flat surface with only 10000 points. The time to create it was 30 minutes. The last two images of the Figure (bottom right) are closeups of region boundaries where the loss of sharp angles is less obvious as in models with less points (see Figure 7.42)

The pyramid models test the variety of models that can be generated with my scheme. I expected that my scheme would suit best the generation of organic-like models because its resampling policy facilitates the replication or smoothing of surface details, as occur in organic surface. However, the pyramid models show that models with sharp transitions between surface features can still be created despite the resampling policy's effect (computing the surface normal assuming smooth

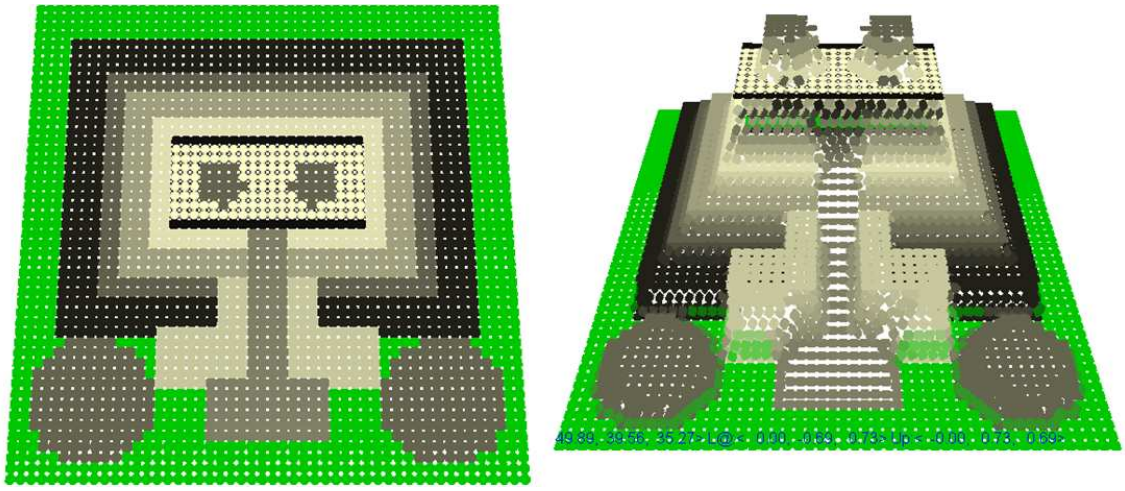

Figure 7.42: Left: Initial surface to create a Pyramid-like structure, Right: Surface after deformation.

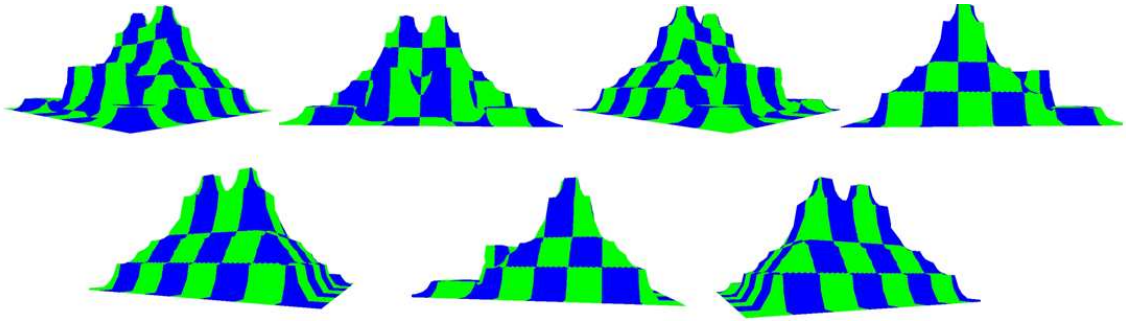

Figure 7.43: Polygonal mesh of the pyramid model in Figure 7.42.

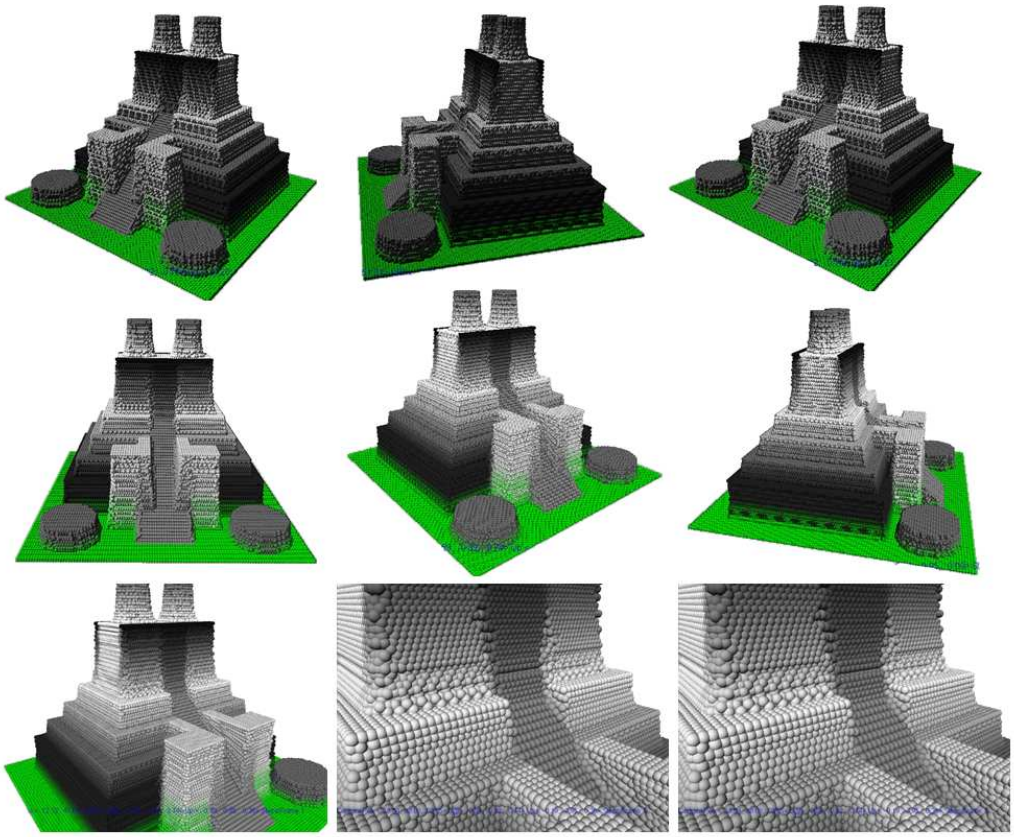

Figure 7.44: Pyramid model with 50000 points.

transitions). Note that this effect is still present, yet it is less severe as the scale of the model increases.

7.4.5 Complex Mushroom-like Models

I also tried combining types of models. In this section I show how I obtained visually rich mushroom-like models. I used deformations originally designed for other models to create the new mushroom-like models.

The first model I created was a mushroom cloud model. This model resembled the cloud of a nuclear explosion. I created this model from the basic model used to create the mushroom of Figure 7.39, shown in Figure 7.45.a. The first new step was to apply a **ScaledUnitaryGaussian** at the lower end of the ovoidal cap (see Figure 7.45.b). Then I used another **ScaledUnitaryGaussian** to expand the remaining cap and applied the Roughening operator to create bumps on the surface (see Figure 7.45.c-d). After some seconds of deformation the model resembles a mushroom cloud, as shown in Figure 7.45.e.

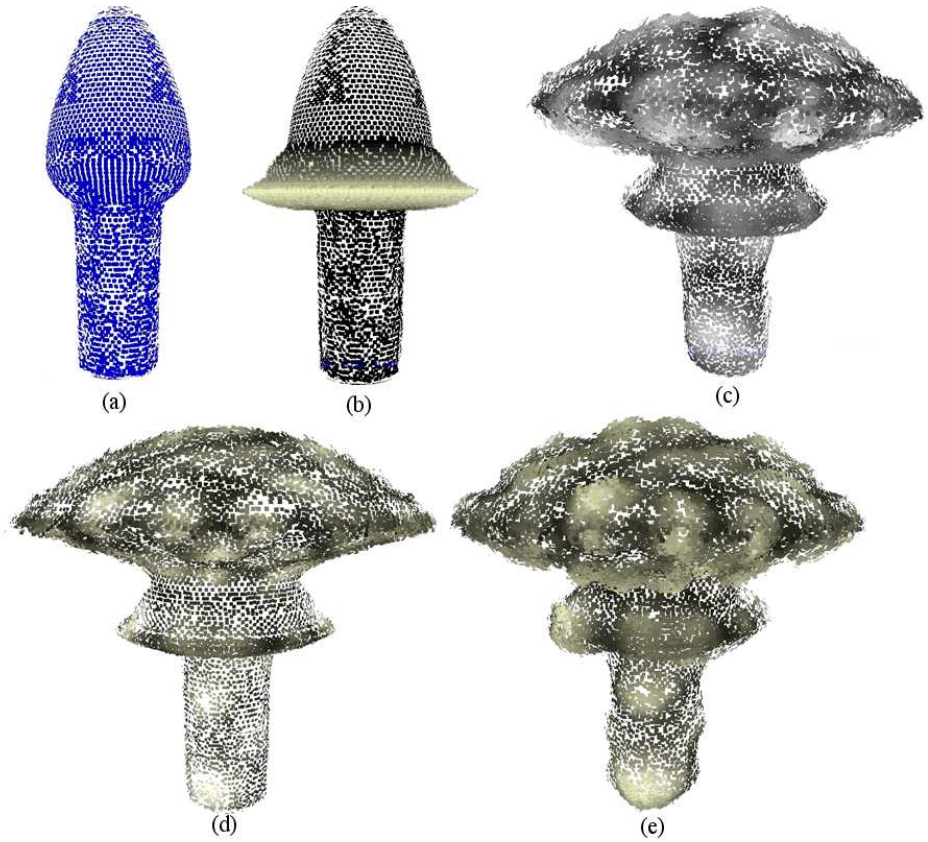

Figure 7.45: Mushroom cloud created with deformations originally designed for other models.

The second model was an elaborated mushroom. For this model I followed the basic steps of creating a mushroom with an irregular cap. Figure 7.46 shows the cylinder (left) obtained from the initial sphere and the irregular cap (right).

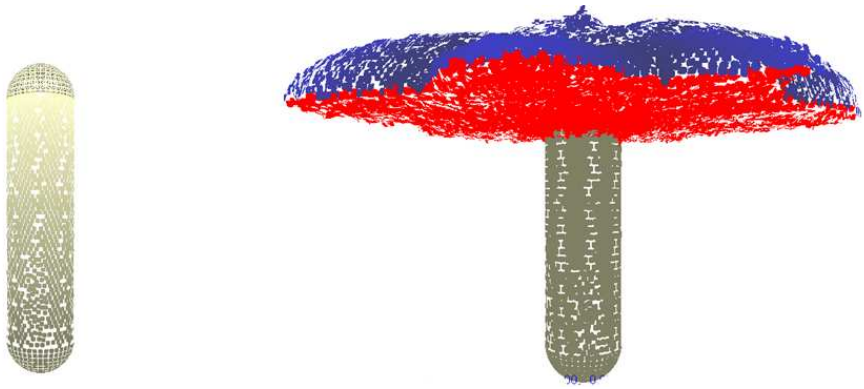

Figure 7.46: For the elaborated mushroom, I followed the basic steps to create a mushroom with irregular cap.

The next step was to use the **VoronoiRegionalization** operator to create regions on the cap (see Figure 7.47 left). Then I assigned a velocity to the borders of the regions and grew them outwards the cap (see 7.47 right), as opposed to the cap of Figure 7.39 where the regions receive a velocity inwards the cap.

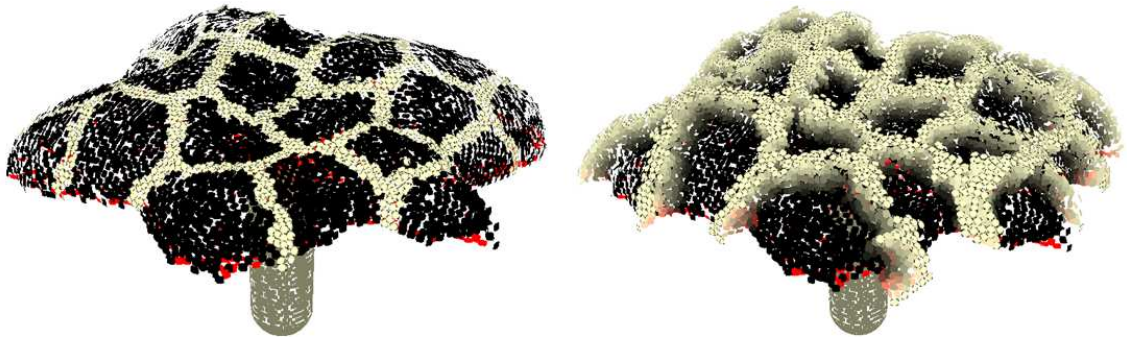

Figure 7.47: The detail of the cap is created with a similar process already used in another mushroom model.

The next step was to provide some detail on the stem. I manually applied eight Thorn operators along the stem. Their parameters were such that the thorns would grow downwards.

Finally, the last deformation was applied to the mushroom base. I simply inflated the lower half of the initial sphere. The inflation was easily obtained by assigning a velocity proportional to vertical distance of the top-most point (of the lower half of the initial sphere) to the lower-most point.

Figure 7.48: Eight thorns were grew by manually applying eight Thorn operators.

Figure 7.49: A inflated root or mushroom base was the final deformation.

Figure 7.50 shows the polygonal mesh obtained of the final model (top left). Also different perspectives of the model are shown, these are rendered with spheres. The final model had 37381 points and was created within two hours.

Figure 7.50: Polygonal mesh (top right) of the final mushroom and different perspectives of the model with sphere-based rendering.

This final model exemplifies how models with complex surface details can easily be created within my scheme. My scheme's components can be used individually or in conjunction to deform a model. The thorns respond to interactive deformation operators applications, while the cap detail responds to automatic application of the deformations. The overall mushroom structure was obtained from the earlier mushroom models, and the scale of the initial surface was not an issue. Note also, that the deformation of a surface has become a simple task for the user since one does not have to worry about maintaining the quality of the surface nor guiding individual deformations.

7.5 Limitations

Volipoc and my model generation framework satisfy the goals initially proposed. However, their design incorporates certain limitations. In this Chapter, I describe the main limitations of Volipoc

and the modeling framework.

1. No volume definition - Volipoc provides an oriented surface which can either be a closed or an open surface; therefore, no volume is associated with the model. The notion of volume is important for applying CSG modeling operators. In order to apply these to Volipoc the initial surface should be either a closed one to generate a volume from it or provide a mechanism to close the open surface.

2. Maximum velocity is 1 - Due to the resampling mechanism and the space partitioning structure used to accelerate resampling, the maximum velocity allowed for any given point on Volipoc is limited to one in order to avoid topology changes. The speed limit may slow the model generation process when velocities vary too much along the surface.

3. No hierarchy is involved - Volipoc is an explicit representation of the model as a whole. No surface section depends on any other section. This difficult the application of animation and modeling techniques that involve the exploits inter-component dependency as in articulated or branched models.

4. Custom collision detection - The lack of having an inside/outside test of the model volume resulted in detecting collisions via local space queries. This makes the surface to arbitrarily stop the deformation and generate some visual artifacts.

5. No support for holes - Volipoc and its resampling policy prevents the appearance of holes on the models. This may be a drawback for designers who want to create a model with holes on its surface.

7.6 Summary

The models I have created with Vebam demonstrate the utility of my Automatic Model Creation framework. Detailed models were created either automatically or semiautomatically. Furthermore, variation of models and models with varied features were easily created by simply manipulating elemental parameters.

The models are rich in surface detail. My framework allowed a user/designer to deform surfaces without worrying about the surface density. My resampling rules (inheritance of points characteristics) allowed to share similar characteristics between points without requiring an individual assignment. Also, the big scale (number of points) handled by Vebam allowed to create distinctive regions/details on models despite the resampling rules.

The use of velocities to implement deformations was helpful for the user/designer follow the underlying process of the model creation. Also the use of velocities allowed a user to interrupt the model creation at different stages and modify the model.

The resampling mechanism worked as expected. Inheriting points' features to spawned points during resampling proved to be an ideal way to obtain surface regions with features without abrupt transitions. Especially after the surface stretches.

My normal approximation made a good approximation of the surface normals. The polygonal meshes obtained from my models showed an acceptable shading. Thus the normals were good enough to shade the models.

The deformation operators were very useful to create details on surfaces. The operators were easy to implement and use, and proved helpful in creating repetitive details.

The automation was demonstrated with the Surface Evolution Control component of the Automatic Model Creation framework. The automation component demonstrated that a variety of models can be obtained by simply manipulating basic parameters. The use of Petri Nets to automatically assign and stop deformations was useful. I consider that Petri Nets facilitated the design of processes to automatically create models due to its separation of states and transitions.

My surface conversion algorithm showed that polygonal meshes can be obtained by only using existing information of the surface, no new information needs to be interpolated. The polygonal meshes also used the normals computed during resampling, which provided good results when shading models.

Models with sharp edges and sharp transitions between surface features were challenging to create, since the resampling scheme created smooth transitions along stretching surfaces. However, it was till possible to create models with sharp edges were still possible to be created.

The deformation time to create a model is not directly proportional to the number of points (see Section 4.2). A user can select in Vebam the number of simulation steps that are made per frame rendered. In this way, big models (such as the asteroid) can be obtained in less time than by making one step per frame.

Automatic generation of models is possible with my Surface Evolution Control. Petri Nets were easy to design and use to automatize complex deformations to create models. I found that the differentiation between states and transitions facilitated the design of the model creation process.

The models I generated demonstrated that my Automatic Model Creation fulfilled the goals of my research. First, I was able to generate three-dimensional models with a minimal user interaction. Additionally, the model generation process can be stopped and rolled back to try different modeling actions. All these was possible by achieving the particular goals of: a surface representation suitable for stretching and compressing deformations expressed in terms of velocities, surface resampling during deformations, deformation specification in terms of operators, and deformation control through a finite state engine. Additionally, my research goals were complemented by designing an algorithm to translate from Volipoc to a polygonal mesh.

Chapter 8

Future Work

My research has several ramifications and opens several options for future research. Examples of such future work are: compare the performance of my surface conversion algorithm, create a grammatical operation for my framework, and incorporate texture synthesis. In the following, I expand on each of these points and suggest some additional ideas.

Compare my surface conversion algorithm to other surface reconstruction algorithms. Even though my algorithm is more a filtering process its results can be compared with those of other surface reconstruction algorithms, in particular, to those of pure-points surfaces. In order to do this I would have to complete the information missing on other surface representations (e.g. links between points in pure-points surfaces). This information could be generated with remeshing operations as those involved in the resampling stage of my framework. Metrics I would look at in making the algorithms comparison would be the number of triangles generated and their quality. Another metric would be to compare the computational complexity of my algorithm with that of the other ones.

Create a grammatical operation for my framework. A grammar description of the operators and the surfaces would introduce an alternative for modeling with my framework. Such alternative could be useful for describing different steps. These alternatives would be tested and reported as to which works better under which circumstances.

Incorporate texture synthesis into my Automatic Model Creation framework. The current state of my Automatic Model Creation framework fulfills the goals of automatic model creation, yet I would like to include texture (as bitmaps) synthesis. Even though textures may not be needed on point-based surfaces they can still provide a lot of visual richness to models. The main problem to solve would be to properly update a texture-coordinates library and define the proper rules for texture synthesis during shrinking and expanding deformations.

Create a dynamic space partitioning system for my framework. Such system should optimize the memory utilization and partition space as needed for the model and avoid reducing the performance. Note, this would not be a simple octree since it increases the number of comparisons required to resample the surface.

Test the effects of shrinking and stretching deformations as techniques to control the surface

samples. This could work as decimation or as smoothing. The resampling behavior can be used to remove high-frequency surface details from models. If such details are the results of noise then the models need to be cleaned.

In the same direction as the previous idea, a study of how much relevant information (valid surface details) is lost during the decimation of models. This study would suggest the possibility of using the resampling as a technique for compressing models.

Optimize Volipoc so that fewer points are used where surfaces have less detail. This would require that edges between points can be allowed between points that are not in adjacent cells of the space partitioning structure. Additionally, this would require to review the resampling process so that resampling can occur between non-adjacent cells.

Use the Petri Net theory to forecast modeling bottlenecks. The Petri Nets theory includes results that forecast unreachable states on the nets. When this is applied to the Petri Nets of my Automatic Model Creation framework one can forecast stages on the model creation that will never be reached. This would be a very useful information for a user/designer.

I would attempt to get a formal mathematical proof of my conversion algorithm. I can provide a soft proof for my conversion algorithm; however, I would like to provide a hard proof of it. I would try to prove that my algorithm creates a water tight polygonal mesh for any possible surface represented with Volipoc.

Consider surface normal correction. Even though my normal computation during resampling offers a good approximation of a surface normal, I would like to include normal recomputation as a postprocessing step.

Investigate the inclusion of tropism into my framework. Tropism could add an additional abstraction layer for model creation. Such layer would allow to define tendencies of deformations, which could be of different styles. Another way to see this layer would be as having a Petri Net managing other Petri Nets.

Finally, in a more technical level I would like to add support into Vebam for plugins development. Currently, the addition of new modeling elements (such as deformation operators) require the compilation of Vebam. I consider that support for plugin development would enhance the acceptance of Vebam by the modeling community.

References

[1] B. Adams and P. Dutré. Interactive boolean operations on surfel-bounded solids. *ACM Transactions on Graphics (TOG)*, 22(3):651, 2003.

[2] J. Allan, B. Wyvill, and I. Witten. A methodology for polygon mesh modelling. *Proc. CG International 89*, 1(1):451, 1989.

[3] P. Bézier. Emploi des machines á commande numérique. *Numerical Control - Mathematics and Applications*, 1(1), 1972.

[4] P. Bézier. Mathematical and practical possibilities of unisurf. *Computer Aided Geometric Design*, 1(1), 1974.

[5] P. Bhat, S. F. Ingram, and G. Turk. Geometric texture synthesis by example. *Proceedings of the Geometry Processing 2004 (Eurographics/ACM SIGGRAPH Symposium)*, 1(1):43, 2004.

[6] H. Biermann, I. Martin, F. Bernardi, and D. Zorin. Cut-and-paste editing of multiresolution surfaces. *Proceedings of the 29th Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'02*, 21(3):312, 2002.

[7] J.F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235, 1982.

[8] R.M. Blomgren and D.J. Kasik. Early investigation, formulation and use of NURBS at boeing. *COLUMN: Computer graphics pioneers*, 1(1):27, 2002.

[9] J. Bloomenthal. *Introduction to Implicit Surfaces*, volume 1. Morgan Kaufmann Publishers, 1997.

[10] J. Bloomenthal and B. Wyvill. Interactive techniques for implicit modeling. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):109, 1990.

[11] K. Brakke. *The Surface Evolver*, volume 1. 1992.

[12] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Department of Computer Science, University of Utah, Salt Lake City, Utah, 1978.

[13] G. Chaikin. An algorithm for high speed curve generation. *Computer Graphics and Image Processing*, 3(4):346, 1974.

[14] B. Cutler, J. Dorsey, L. McMillan, M. Muller, and R. Jagnow. A procedural approach to authoring solid models. *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'02*, 21(3):302, 2002.

[15] Peachey D. Solid texturing of complex surfaces. *Proceedings of the 12th Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'85*, 1(1):279, 1985.

[16] J.S. DeBonet. Multiresolution sampling procedure for analysis and synthesis of texture images. *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'97*, 1(1):361, 1997.

[17] H. Delingette. General object reconstruction based on simplex meshes. *International Journal of Computer Vision*, 2(32):111, 1999.

[18] O. Deussen, P. Hanrahan, B. Lintermann, R. Mech, M. Pharr, and P. Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. *Proceedings of the 25th Annual Conference on Computer Graphics and INteractive Techniques, SIGGRAPH'98*, 1(1):275, 1998.

[19] N. Dyn, D. Levin, and J.A. Gregory. A bufferfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9(2):160, 1990.

[20] D.S. Ebert, F.K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers.

[21] A. Efros and T. Leung. Texture synthesis by non-parametric sampling. *IEEE International Conference on Computer Vision, ICCV'99*, 1(1):1033, 1999.

[22] A.A. Efros and W.T. Freeman. Image quilting for texture synthesis and transfer. *Proceedings of the 28rd Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'01*, 1(1):341, 2001.

[23] K.W. Fleischer, D.H. Laidlaw, B.L. Currin, and A.H. Barr. Cellular texture generation. *Proceedings fo the 22nd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'95*, 1(1):239, 1995.

[24] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, and R.L. Phillips. *Introduction to computer graphics*. Addison Wesley, 1997.

[25] A.R. Forrest. The twisted cubic curve: a computer-aided geometric design approach. *Computer Aided Design*, 12(4):165, 1980.

[26] D.R. Fowler, P. Prusinkiewicz, and J. Battjes. A collision-based model of spiral phyllotaxis. *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'92*, 26(2):361, 1992.

[27] S.F. Frisken, R.N. Perry, A.P. Rockwood, and T.R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, 1(1):249, 2000.

[28] T. Funkhouser, M. Kazhdan, P. Shilane, P. Min, W. Kiefer, A. Tal, S. Rusinkiewicz, and D. Dobkin. Modeling by example. *ACM Transaction on Graphics (TOG)*, 23(3):652, 2004.

[29] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'97*, 1(1):209, 1997.

[30] J.P. Grossman and W.J. Dally. Point sample rendering. *Proceedings of the 9th Eurographics Workshop on Rendering*, 1(1):181, 1998.

[31] A. Guéziec. "meshsweeper": Dynamic point-to-polygonal-mesh distance and applications. *IEEE Transactions on Visualization and Computer Graphics*, 7(1):47, 2001.

[32] I. Guskov, L. Vidimce, W. Sweldens, and Schroeder. Normal meshes. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'00*, 1(1):95, 2000.

[33] S. Hahmann and G.-P. Bonneau. Polynomial surfaces interpolating arbitrary triangulations. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):99, 2003.

[34] A. Hausner. Simulating decorative mosaics. *Proceedings of the 28rd Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'01*, 1(1):573, 2001.

[35] D. J. Heeger and J. R. Bergen. Pyramid-based texture analysis/synthesis. *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'95*, 1(1):229, 1995.

[36] H. Hoppe. Progressive meshes. *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'96*, 1(1):99, 1996.

[37] T. Ijiri, S. Owada, M. Okabe, and T. Igarashi. Floral diagrams and inflorescences: interactive flower modeling using botanical structural constraints. *ACM Transaction on Graphics (TOG)*, 24(3):720, 2005.

[38] T. Ju, S. Schaefer, and J. Warren. Mean value coordinates for closed triangular meshes. *Proceedings of ACM SIGGRAPH 2005*, 24(3):561, 2005.

[39] R. Keiser, B. Adams, D. Gasser, P. Bazzi, P. Dutré, and M. Gross. A unified lagrangian approach to solid-fluid animation. *Eurographics Symposium on Point-Based Graphics*, 1(1):125, 2005.

[40] S. Kim and C. Song. Rendering of unorganized points with octagonal splats. In V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3992 of *LNCS*, pages 326–333. Springer, 2006.

[41] A. Lagae, O. Dumont, and P. Dutré. Geometry synthesis. Technical Report CW 381, Katholieke Universiteit Leuven, Departement of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee, Belgium, March 2004.

[42] J. Lawrence and T. Funkhouser. A painting interface for interactive surface deformations. *11th Pacific Conference on Computer Graphics and Applications, PG'03*, 66(6):418, 2004.

[43] M. Levoy and T. Whitted. The use of points as a display primitive. *Technical Report 85-022, Computer Science Department, University of North Carloina at Chapel Hill*, 1(1), January 1985.

[44] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics*, 20(3):127, 2001.

[45] A. Lindenmayer. Mathematical models for cellular interactions in development, parts I and II. *Journal of Theoretical Biology*, 1(18):280, 1975.

[46] Y. Lipman, O. Sorkine, D. Levin, and D. Cohen-Or. Linear rotation-invariant coordinates for meshes. *Proceedings of ACM SIGGRAPH 2005*, 24(3):479, 2005.

[47] I. Llamas, B.M. Kim, J. Gargus, J. Rossignac, and C.D. Shaw. Twister: a space-warp operator for the two-handed editing of 3d shapes. *ACM Transaction on Graphics (TOG)*, 22(3):663, 2003.

[48] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129, 1982.

[49] S. Lobregt and M. A. Viergever. A discrete dynamic contour model. *IEEE Transactions on Medical Imaging*, 14(1):12, 1995.

[50] C. Machover. The business of computer graphics. *IEEE Computer Graphics and Applications*, 20(1):44–45, January/February 2000.

[51] C. Mandal, H. Qin, and B. C. Vemuri. Dynamic modeling of butterfly subdivision surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):265, 2000.

[52] C. Mandal, B. C. Vemuri, and H. Qin. A new dynamic fem-based subdivision surface model for shape recovery and tracking in medical images. *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI'98)*, 1(1):753, 1998.

[53] R. Mech and P. Prusinkiewicz. Visual models of plants interacting with their environment. *Proceedings of the 23rth Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'96*, 1(1):397, 1996.

[54] J. V. Miller, D. E. Breen, W. E. Lorensen, R. M. O'Bara, and M. J. Wozny. Geometrically deformed models: A method for extracting closed geometrics models from volume data. *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'91*, 25(1):217, 1991.

[55] D. Mould. Image-guided fracture. *Proceedings of the 2005 conference on Graphics Interface*, 1(1):219, 2005.

[56] K. Museth, D.E. Breen, R.T. Whitaker, and A.H. Barr. Level set surface editing operators. *Proceedings of the 29th Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'02*, 21(3):330, 2002.

[57] A. Nealen and M. Alexa. Hybrid texture synthesis. *Proceedings of the Eurographics Symposim on Rendering 2003*, 1(1):97, 2003.

[58] A. Nealen, O. Sorkine, M. Alexa, and D. Cohen-Or. A sketch-based interface for detail-preserving mesh editing. *ACM Transaction on Graphics (TOG)*, 24(3):1142, 2005.

[59] F. Neyret and M.P. Cani. Pattern-based texturing revisited. *Proceeding of the 26th Annual conference on Computer graphics and interactive techniques, SIGGRAPH'99*, 1(1):235, 1999.

[60] Y. Ohtake and A. G. Belyaev. Dual/primal mesh optimization for polygonized implicit surfaces. *Proceeding of the Seventh ACM Symposium on Solid Modeling and Applications*, 1(1):171, 2002.

[61] P. E. Oppenheimer. Real time design and animation of fractal plants and trees. *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'86*, 1(1):55, 1986.

[62] S. Osher and J.A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on hamilton-jacobi formulations. *Journal of Computational Physics*, 1(79):12, 1988.

[63] Y.I.H. Parish and P. Muller. Procedural modeling of cities. *Proceedings of the 28th Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'01*, 1(1):301, 2001.

[64] A. Pasko, V. Savchenko, and A. Sourin. Synthetic carving using implicit surface primitives. *Computer Aided Design*, 33(5):379, 2001.

[65] M. Pauly, R. Keiser, L. P. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. *ACM Transaction on Graphics (TOG)*, 22(3):641, 2003.

[66] K. Perlin. An image synthesizer. *Proceedings of the 12th Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'85*, 1(1):287, 1985.

[67] K. Perlin. Improving noise. *Proceedings of the 29th Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'02*, 1(1):681, 2002.

[68] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'00*, 1(1):335, 2000.

[69] L. Piegl and W. Tiller. *The NURBS book*, volume 1. Springer-Verlag, 2 edition, 1997.

[70] J. Portilla and E.P. Simoncelli. A parametric texture model based on joint statistics of complex wavelet coefficients. *International Journal of Computer Vision*, 40(1):49, 2000.

[71] S.D. Porumbescu, D.B. Budge, D.L. Feng, and K.I. Joy. Shell maps. *ACM Transactions on Graphics (TOG)*, 24(3):626, 2005.

[72] J.L. Power, A.J. Bernheim-Brush, P. Prusinkiewicz, and D. Salesin. Interactive arrangement of botanical L-systems models. *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, 1(1):175, 1999.

[73] P. Prusinkiewicz, M.S. Hammel, and E. Mjolsness. Animation of plant development. *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'93*, 1(1):351, 1993.

[74] P. Prusinkiewicz, M. James, and R. Měch. Synthetic topiary. *Proceeding of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'94*, 1(1):351, 1994.

[75] P. Prusinkiewicz, A. Lindenmayer, and J. Hanan. Development model of herbaceous plants for computer imagery purposes. *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'88*, 1(1):141, 1988.

[76] H. Qin, C. Mandal, and B. C. Vemuri. Dynamic Catmull-Clark subdivision surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):215, 1998.

[77] R. Rangel-Kuoppa and D. Mould. Model creation by velocity controlled surface deformation. In *International Conference on Computational Science (2)*, pages 318–325, 2006.

[78] W. Reisig. *Petri nets: an introduction*, volume 1. Springer-Verlag, 1985.

[79] M. Renton, J. Hanan, and P. Kaitaniemi. The inside story: Including physiology in structural plant models. *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Austalasia and South East Asia*, 1(1):95, 2003.

[80] A. Ricci. A constructive geometry for computer graphics. *The Computer Journal*, 16(2):157, 1973.

[81] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'00*, 1(1):343, 2000.

[82] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen. Decimation of triangle meshes. *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'92*, 26(2):65, 1992.

[83] J. Sethian. *Level Set Methods*. United States of America: Cambridge University Press, 1996.

[84] E. Shaffer and M. Garland. Efficient adaptive simplification of massive meshes. *Proceedings of IEEE Visualization 2001*, 1(1):127, 2001.

[85] R. Shekhar, E. Fayyad, R. Yagel, and J.F. Cornhill. Octree-based decimation of marching cubes surfaces. *Proceedings of the 7th Conference on Visualization '96*, 1(1):335, 1996.

[86] P. Shirley. *Fundamentals of Computer Graphics*, volume 1. A.K. Peter Ltd., 2 edition, 2005.

[87] A.R. Smith. Plants, fractals, and formal languages. *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'84*, 18(3):1, 1984.

[88] J. Smith, J. Hodgins, I. Oppenheim, and A. Witkin. Creating models of truss structures with optimization. *Proceedings of the 29th Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'02*, 21(3):295, 2002.

[89] J. Snel, H. Venema, and C. Grimbergen. Deformable triangular surfaces using fast 1D radial lagrangian dynamics segmentation of 3D MR and CT images of the wrist. *IEEE Transactions on Medical Imaging*, 21(8):888, 2002.

[90] R.W. Sumner, M. Zwicker, C. Gotsman, and J. Popović. Mesh-based inverse kinematics. *ACM Transaction on Graphics (TOG)*, 24(3):488, 2005.

[91] R. Szeliski and D. Tonnesen. Surface modeling with oriented particle systems. *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'92*, 2(26):185, 1992.

[92] D. Terzopoulos and H. Qin. Dynamic NURBS with geometric constraints for interactive sculpting. *Special issue on interactive sculpting*, 1(1):103, 1994.

[93] G. Turk. Generating textures on arbitrary surfaces using reaction-diffusion. *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'91*, 1(1):289, 1991.

[94] X. Viennot, G. Eyrolles, N. Janey, and D. Arqués. Combinatorial analysis of ramified patterns and computer imagery of trees. *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'89*, 1(1):82, 1989.

[95] G. Voronoi. Recherches sur les paralllodres primitives. *Math*, 1(134):198, 1908.

[96] M. Walter, A. Fournier, and M. Reimers. Clonal mosaic model for the synthesis of mammalian coat patterns. *Proceedings of Graphics Interface'98*, 1(1):82, 1998.

[97] C.L. Weeks and J.C. Comfort. The growth process of tropical trees: A simulation with graphic output. *Proceedings of the 15th Conference on Winter Simulation - Volume 2*, 2(1):649, 1983.

[98] L. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. *Proceeding of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'00*, 1(1):479, 2000.

[99] N. Williams, D. Luebke, J. D. Cohen, M. Kelley, and B. Schubert. Perceptually guided simplification of lit, textured meshes. *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, 1(1):113, 2003.

[100] A. Witkin and M. Kass. Reaction-diffusion textures. *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'91*, 25(4):299, 1991.

[101] S. Worley. A cellular texture basis function. *Proceedings of the 23rd Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'96*, 1(1):291, 1996.

[102] B. Wyvill, E. Galin, and A. Guy. Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system. *Computer Aided Forum*, 18(2):149, 1999.

[103] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227, 1986.

[104] G. Wyvill and K. Novins. Filtered noise and the fourth dimension. *International Conference on Computer Grpahics and Interactive Techniques (ACM SIGGRAPH 99)*, 1(1):242, 1999.

[105] P. Yim, G. Vasbinder, V. Ho, and P. Choyke. Isosurfaces as deformable models for magnetic resonance angiography. *IEEE Transactions on Medical Imaging*, 22(7):857, 2003.

[106] Y. Yu, K. Zhou, D. Xu, X. Shi, H. Bao, B. Guo, and H.Y. Shum. Mesh editing with poisson-based gradient field manipulation. *ACM Transaction on Graphics (TOG)*, 23(3):644, 2004.

[107] K. Zhou, J. Huang, J. Snyder, X. Liu, H. Bao, B. Guo, and H.-Y. Shum. Large mesh deformation using the volumetric graph laplacian. *Proceedings of ACM SIGGRAPH 2005*, 24(3):496, 2005.

[108] J. Zobel. *Writing for Computer Science*, volume 1. Springer, 2004.

[109] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. Pointshop 3D: An interactive system for point-based surface editing. *Proceedings of the 29th Annual Conference on computer Graphics and Interactive Techniques, SIGGRAPH'02*, 21(3):322, 2002.

Appendix A

Outermost Triangles Computation

In this Appendix, I present the algorithm to compute the outermost triangles of a set of triangles sharing an edge in Volipoc. This algorithm is part of the surface conversion process presented in Chapter 6. The input of the algorithm is an edge of Volipoc and all the triangles that share the edge. The output of the algorithm is the labeling of some of the triangles as Outermost Triangles. I explain the algorithm in terms of the following 4 definitions:

1. Oriented triangle - A triangle with Volipoc's oriented points as vertices (Figure A.1.a). The triangle normal is computed as the projection of the average of the points orientation (Figure A.1.b) on the normal of the plane defined by the triangle's vertices.

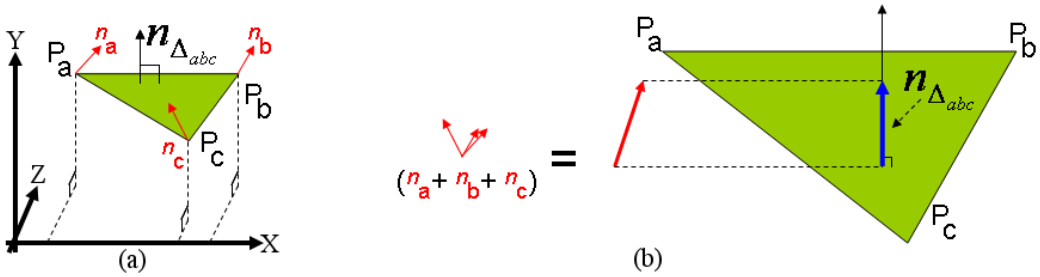

Figure A.1: (a) An oriented triangle whose vertices are oriented points, and (b) The triangle orientation is computed with the projection of the average of the vertices orientations on the normal of the plane formed by the triangle; the orientation may change sign depending on which side of the plane the average points towards.

The identification of outermost triangles uses the angle measurement between oriented triangles. The triangle normal is the projection of the average of the vertices' orientations on the normal of the plane defined by the triangle's vertices; that is, if the average of the vertices' orientations reside in a certain side of the plane then the normal of the triangle resides in the same side. For example, Figure A.1.a shows triangle $\triangle_{abc}$ whose vertices are the oriented points P_a, P_b, and P_c (each point has an orientation defined by vectors n_a, n_b, and n_c respectively). The normal of triangle $\triangle_{abc}$ is computed by:

$$n_{\triangle_{abc}} = \begin{cases} n_{avg} \cdot n_{plane} & \phi_{normals} \in [0, \frac{\pi}{2}) \\ -(n_{avg} \cdot n_{plane}) & \phi_{normals} \in [\frac{\pi}{2}, \pi] \end{cases} \quad \text{(A.1)}$$

where,

$$n_{avg} = \frac{n_a + n_b + n_c}{3} \quad \text{(A.2)}$$

$$n_{plane} = (p_a - p_b) \times (p_a - p_c) \tag{A.3}$$

$$\phi_{normals} = \arccos\left(\left(\frac{(p_a - p_b) \times (p_a - p_c)}{\|(p_a - p_b) \times (p_a - p_c)\|}\right) \cdot \left(\frac{\frac{n_a + n_b + n_c}{3}}{\|\frac{n_a + n_b + n_c}{3}\|}\right)\right) \tag{A.4}$$

2. Oriented triangle plane - The plane defined by the three vertices of an oriented triangle (Figure A.2.d).

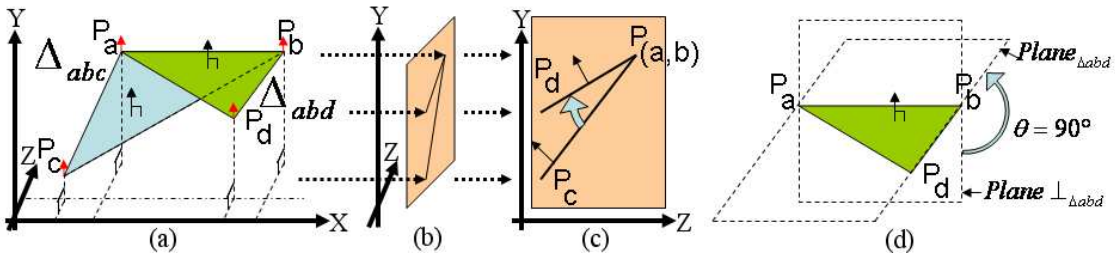

Figure A.2: (a) Two oriented oriented triangles: $\triangle_{abc}$ and $\triangle_{abd}$. (b) Projection of triangles $\triangle_{abc}$ and $\triangle_{abd}$ on a plane perpendicular to vector $P_a - P_b$. (c) Vector $P_a - P_b$ is projected as point $P_{(a,b)}$ on the plane perpendicular to vector $P_a - P_b$. (d) $Plane \perp_{\triangle abd}$ is perpendicular to $Plane_{\triangle abd}$.

From Figure A.2.d, $Plane_{\triangle abd}$ is computed as:

$$Plane_{\triangle abd} = \left\{p \in \Re^3 | n_{\triangle_{abd}} \cdot (p - p_a) = 0\right\} \tag{A.5}$$

and the plane perpendicular to the plane defined by triangle $\triangle_{abd}$ and that has vector $P_a - P_b$ on it as:

$$Plane \perp_{\triangle abd} = \left\{p \in \Re^3 | \left((p_a - p_b) \times n_{\triangle_{abd}}\right) \cdot (p - p_a) = 0\right\} \tag{A.6}$$

3. Oriented triangle behind another - I say that an oriented triangle A is "behind" an oriented triangle B if any of the following conditions is satisfied:

   - The magnitude of the distance between non-shared vertex in B and the plane defined by triangle A is less than zero, and the distance between non-shared vertex in B and the perpendicular plane of A is greater or equal than zero, and the dot product between triangles' normals is greater or equal than zero. See Figure A.3.a.
   - The magnitude of the distance between non-shared vertex in B and the plane defined by triangle A is less than zero, and the distance between non-shared vertex in B and the perpendicular plane of A is less than zero, and the dot product between triangles' normals is less than zero. See Figure A.3.b.

- The magnitude of the distance between non-shared vertex in B and the plane defined by triangle A is greater or equal than zero, and the distance between non-shared vertex in B and the perpendicular plane of A is less than zero, and the dot product between triangles' normals is less than zero. See Figure A.3.c.
- The magnitude of the distance between non-shared vertex in B and the plane defined by triangle A is greater or equal than zero, and the distance between non-shared vertex in B and the perpendicular plane of A is greater or equal than zero, and the dot product between triangles' normals is greater or equal than zero. See Figure A.3.d.

The previous conditions guarantee the identification of one oriented triangle been behind another (see Figure A.3.a-d). The previous conditions also consider the cases where the oriented triangles are both part of the outermost shape; that is, no triangle is behind the other (see Figure A.3.e-h).

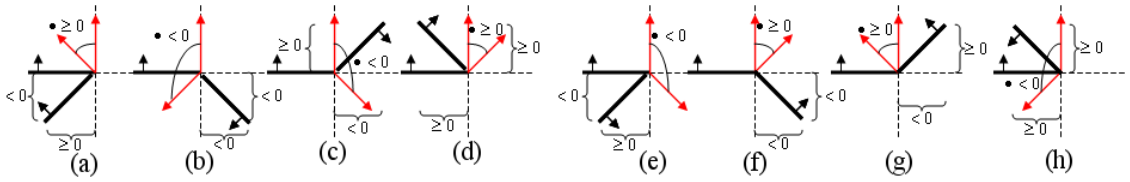

Figure A.3: (a-d) Cases of one oriented triangle behind another. (e-h) Cases of oriented triangles not occluding each other.

4. Angle between oriented triangles - The angle between two oriented triangles is computed by projecting the triangles on the plane perpendicular to the axis defined by the shared edge (see Figure A.2.b). The measurement of the angle between the two projected triangles is the angle between the oriented triangles; for example, from Figure A.2.c the angle between triangles $\triangle_{abc}$ and $\triangle_{abd}$ is the angle between the projected vectors $(P_d, P_{(a,b)})$ and $(P_c, P_{(a,b)})$.

   The computation of the angle between adjacent oriented triangles makes it to be in the range of $[0, 2\pi)$ radians and not in $[0, \pi)$ as is the angle between two planes. Not all pairs of adjacent oriented triangles have an outermost triangle. The concept of outermost triangles applies only when one triangle is to be considered "behind" another one (see Figure A.3.a-d). A triangle from a pair of adjacent oriented triangles is not behind the other one if the normals of the oriented triangles point to the same direction after rotating one of the triangles around the shared edge (Figure A.3.e-h).

The previous definitions are used in the steps of the algorithm. The input of the algorithm is an edge of Volipoc and all the points that are linked to both the edge's points; note that these constitute all the oriented triangles that share the edge. The output of the algorithm is the labeling

of zero, one, or two triangles as Outermost Triangles. The labeling is later used by the Volipoc-to-polygonal conversion process of Chapter 6. An oriented triangle may have been previously labeled as Outermost Triangle during the Volipoc-to-polygonal conversion process. The final piece of this Appendix is the steps of the algorithm in prosecode:

1. (No oriented triangles) If the number of oriented triangles is zero then return from algorithm.
2. (One oriented triangle) If the number of oriented triangles is one then label it as Outermost Triangle and return from algorithm.
3. (Label outermost triangle or triangles)
   (a) (Triangle previously labeled) If a triangle was previously labeled as Outermost Triangle.
      i. (Do not label and return) If all triangles are behind the triangle labeled as Outermost Triangle then do not label any new triangle and return from algorithm.
      ii. (Label and return) Label the triangle that is not behind the previously labeled triangle and that has the smallest angle with the previously labeled triangle.
   (b) (No triangle previously labeled) If there is no triangle previously labeled as Outermost Triangle.
      i. (Label pair of outermost triangles) Label the pair of triangles that are not behind each other and that have the smallest angle between them.

Appendix B

Terminology

The current appendix explains the different terms used around this document, and that may create confusion on people not related to the topic.

3D models 3D models or models are the geometry describing the shape of an object in a computer graphics scene.

Adjacent Triangles Triangles that share one edge.

BSP-tree Binary Space Partitioning tree: A tree-structure that subdivides space in hierarchical binary partitions.

Computer Graphics Model In Computer Graphics, the term "model" denotes the mathematical specification of shape and appearance properties so that it can be stored on computer ([86]).

Constructive Solid Geometry Technique for modeling solids in terms of boolean operators whose operands are simple solids.

Content In Computer Graphics, describes the geometries and textures (bitmaps) used in computer graphics to render a scene.

Decimation Process to reduce the number of samples of a surface.

Deformation Operators Operators that apply deformations to a surface by associating velocities with selected surface elements. There are two types of Deformation Operators: Selection and Velocity.

Deformation scope Scale at which a deformation affects a surface. I identify three levels: surface, local, and global.

Equiangulation Process to make less dissimilar the internal angles of two adjacent triangles, the shared edge is switched for the traversal edge if this makes the internal angles more similar.

Explicit surface representation Surface representation that lists each element that defines a surface, without any implication.

Flat shading Shading technique in which a polygon is rendered with a constant color.

Flooding algorithm Algorithm that distributes information to every part of a connected network.

Frustum culling Process by which objects not residing within the viewing frustum are discarded instead of being rendered.

Geometry The set of vertices and edges that constitute the polygonal mesh of a surface rendered in a computer graphics scene.

Global-level deformation scope Global scope indicates the deformation of the whole set of

surface elements.

Gourad shading Shading technique that uses the estimated surface normal of each vertex to interpolate color values along a polygon with the Phong reflection model.

HCI Acronym for **H**uman **C**omputer **I**nteration.

Local-level deformation scope Local scope indicates the deformation of a subset of surface elements.

Mesh congruency A mesh property indicating whether it has overlapping or intersecting polygons (non-congruent mesh) or not (congruent mesh).

Model A representation of either a concrete or abstract entity in terms of some of its features.

Modeling In Computer Graphics, the area that covers the techniques involved in the mathematical specification of shape and appearance properties.

Neighboring points Points in Volipoc that are linked by a volatile edge.

Normalization Process that changes something to conform a standard, e.g., the normalization of a vector changes it into a unit length in the same direction as the original.

Octree A space partitioning tree data structure in which each node is subdivided into eight octants.

Oriented Triangles Triangles whose vertices have a normal associated to them.

Parallel rewriting grammars Type of formal grammar in which the production rules are executed in parallel over all the characters of a string.

Parameterization The description of a phenomenon in terms of parameters.

Perspective A technique to project three-dimensional information on a two-dimensional surface.

Phong shading Shading technique that computes a color for each individual pixel using normal and lighting parameters using the Phong reflection model.

Polygonal mesh Generic term to define a tesselated organization of polygons joint that share edges between them. In Computer Graphics, the most common polygon used are triangles; therefore, the term "polygonal mesh" is often used to describe a tesselated arrangement of triangles.

Prosecode A formalism for presenting algorithms suggested by Zobel [108] to be a better option than pseudocode. Prosecode guidelines are: number each step, never break a loop over several steps, use subnumbering for the part of a step, and include explanatory text.

Rendering Primitive The simplest of default geometrical figures or shapes.

Selection Operators Type of Deformation Operator focused in selecting a subset of surface elements.

Shading Depiction of depth by varying levels of darkness.

Shape In Computer Graphics, shape encompass a surface's features of interest.

Smoothing Process to remove noise from a surface.

Subdivision Process to represent a surface with a coarser representation.

Surface-level deformation scope Surface scope indicates the deformation of single element of a surface.

Surface density Number of surface elements per volume unit.

Surface Representation Specification of a surface shape.

Surfel "**surf**ace **el**ement". An oriented disc in space.

Surfel rendering Rendering technique in which surfels are used to render a surface.

Texel **Tex**ture **El**ement.

Texture In Computer Graphics has two meanings: Small scale surface details and the bitmap used to simulate such details.

Texture Coordinates Coordinates used to map a texture (as a bitmap) on a triangle. Texture coordinates are in texture space in the [0,1] range.

Thin triangle A triangle with edges that are larger than the norm.

Time Handler Clock generator component of my Automatic Model Creation scheme.

Topology Field that studies the properties of geometric forms that remain invariant under certain transformations, as bending or stretching.

Triangle normal Normal computed with the cross product of two of the triangle's edges.

Tropism Orientation of an organism due to an external stimulus.

Vebam Name of my software application that demonstrates my Automatic Model Creation framework.

Velocity Operators Type of Deformation Operator focused in computing and assigning velocities to a set of selected surface elements.

Volatile edges Edges from Volipoc that may be removed or inserted in the surface an automatic resampling process.

Volipoc Name of my surface representation; Volipoc stands for "**Vo**latile-**Li**nked **Po**int **C**loud".

Win32 The Windows operating system API (application programming interfaces).

z-Buffer Buffer that holds depth information of polygons being rendered. It is used for visibility testing.

Zero level set The set of all parameters of a real-valued function such that the function value is zero.

Appendix C

Introduction to Vebam

Vebam is the software I developed to demonstrate the concepts of my Automatic Model Creation framework. Figure C.1 shows a snapshot of the Vebam's interface. It is basically a rendering interface. Most of the work is done through the menus; however, a user can interact with the models via mouse and keyboard. The upper part contains menus with different command to create models, while the lower part indicate the values of the camera position, "look at" vector, the "up" vector, and the number of simulation steps per rendered frame.

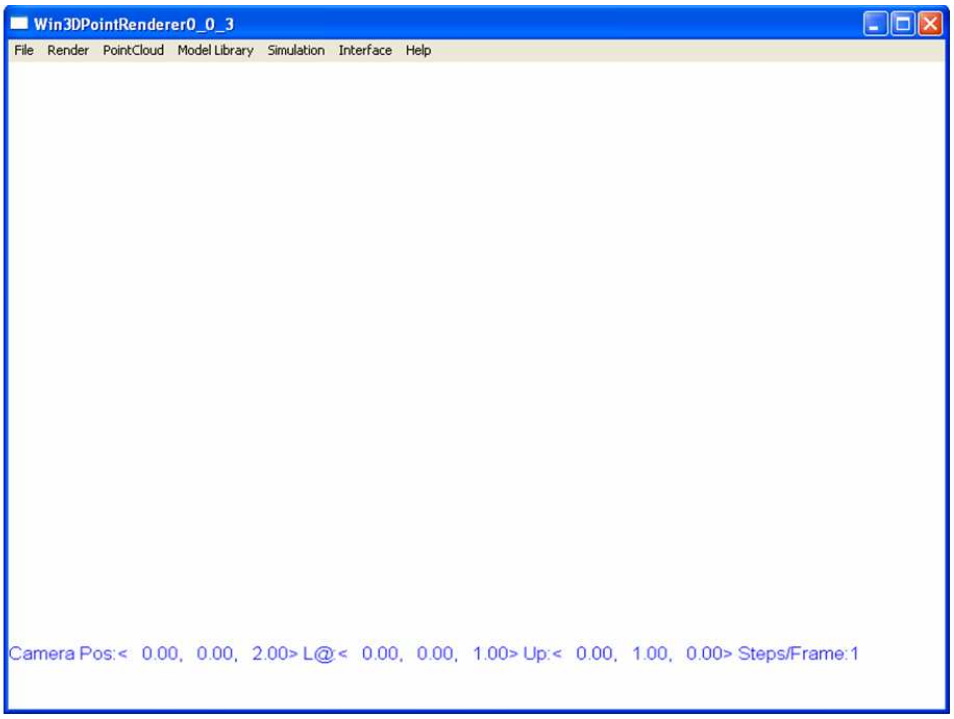

Figure C.1: Vebam interface.

The menus are described next:

- File
  - Save Point Cloud - Saves a model in a binary file.
  - Load Point Cloud - Loads a model from a binary file.
  - Save Point Cloud in Text - Saves a model in a text file (text file elements are tagged).
  - Load Point Cloud in Text - Loads a model from a text file.
  - Exit - Ends the execution of Vebam.

- Render
  - Normal - Renders all normals.
  - Bounding Cube - Renders a cube circumscribing the model.
  - Links - Renders links that have at least one point facing the camera.
  - Absolutely All Links - Renders all links.
  - 3D Grid - Renders the grid cells as long as they have at least one point inside.
  - Polygonal Mesh - Executes the surface conversion algorithm, when it finished the resulting polygonal mesh is rendered.
  - Shading Mode
    - Flat - When rendering a polygonal mesh, triggers the flat shading mode.
    - Gourad - When rendering a polygonal mesh, triggers the Gourad shading mode.
    - Phong - When rendering a polygonal mesh, triggers the Phong shading mode.
    - Texture - When rendering a polygonal mesh, enables the rendering with textures.
    - Texture1 - When rendering a polygonal mesh, enables the rendering of preset texture 1.
    - Texture2 - When rendering a polygonal mesh, enables the rendering of preset texture 2.
    - Texture3 - When rendering a polygonal mesh, enables the rendering of preset texture 3.
    - Load Texture From File - Loads a texture from a file.
    - Texture Loaded - When rendering a polygonal mesh, enables the rendering of a texture loaded from a file.
    - Tex = Tex + Diffuse - When rendering a polygonal mesh, the shading is computed by adding texture and diffuse color components.
  - Gamma Polygonal Mesh
    - Render Edges - Render the edges of a polygonal mesh.
    - Render Triangles - Renders the triangles of a polygonal mesh.
  - Sphere Rendering
    - Sphere-based Pure Emissive - Enables sphere-based rendering, each sphere has only an emissive color component.
    - Sphere-based Soft - Enables sphere-based rendering.

- PointCloud
  - Erase all velocities
  - Point Operators (Note: Operators that are applied to individual points use the mouse to select the point by clicking on it. If the operator has several parameters then a dialogue box appears to input the parameters, see Figure C.2)
    * Gaussian Velocities - Enables the Gaussian operator.
    * Thorns - Enables the Thorn operator.
    * Cratering - Enables the Cratering operator.
    * Roughening - Enables the Roughening operator.
    * Cracking - Enables the Cracking operator.
  - Miscellaneous
    * Paint all points in green - Paints all points of the model green.
    * Paint all points in blue - Paints all points of the model blue.
    * Paint all points in white - Paints all points of the model white.
- Model Library
  - Pear
    * Initial Sphere - Creates the initial sphere (with velocities) for the Pear model.
    * Apply Curving Effect - Applies the velocities to curve the upper half of the Pear model.
  - Tomato
    * Initial Sphere - Creates the initial sphere (with velocities) for the Tomato model.
    * Add Regular Asymmetry - Applies the velocities for the second stage of the Tomato model creation.
  - Strawberry
    * Initial Sphere - Creates the initial sphere (with velocities) for the Strawberry model.
    * Elongation - Applies the velocities for the second stage of the Strawberry model creation.
  - Apple
    * Initial Sphere - Creates the initial sphere (with velocities) for the Apple model.
    * Add Regular Asymmetry - Applies the velocities for the second stage of the Apple model creation.

- Banana
    - Set Initial Sphere - Creates the initial sphere (with velocities) for the Banana model.
    - Apply Elongating Effect - Applies the velocities for the second stage of the Banana model creation.
- Cactus
    - Initial Sphere - Creates the initial sphere (with velocities) for the Cactus model.
    - Activate Petri Net - Enables the Petri Net to create the cactus model.
- Mushrooms (Note: The following menus correspond to the different stages to create the mushroom models.)
    - Set Initial Sphere
    - Set Cap Growth Velocities
    - Create Gills
    - Model 2
        - Set Initial Sphere
        - Apply Inflating Stem Velocities
        - Set Cap Growth Velocities
        - Apply Cap Detail Velocities
    - Model 3
        - Set Initial Sphere
        - Set Cap Growing Velocities
        - Bend Cap Downwards
        - Create Circular Cap Patterns
    - Model 4
        - Set Initial Sphere
        - Set Root Velocities
        - Set Stem
        - Apply Voronoi Regions for Cap
        - Apply Imploding Velocities to Voronoi Regions
        - Apply Voronoi Regions to Stem

- Simulation
    - Step By Step - Enables the deformation of a model one step at a time.
    - TimedSaves - Enables the automatic saving (with naming) of the model at different time intervals.

- Interface
  - Keyboard Enabled - Enables/disables the keyboard input.
  - Camera Look At Center - Enables/disables the camera always looking at the geometric center of the model.
- Help
  - About - Displays information related to Vebam.

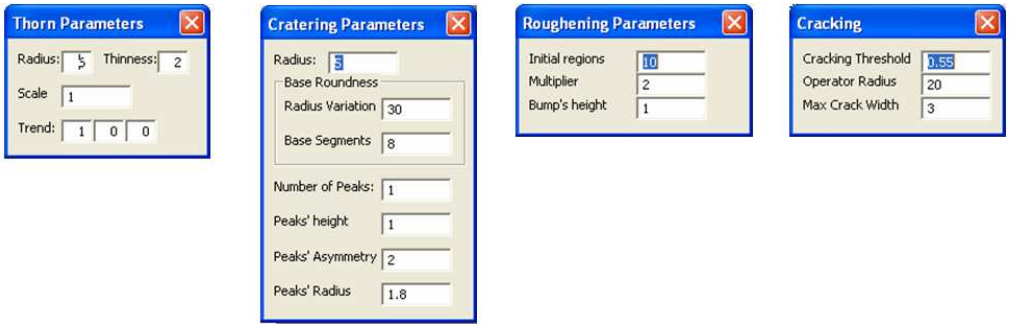

Figure C.2: Vebam operator dialogues.