1 Synthetic Texturing

1.1 Introduction

This dissertation describes improved methods for computer generation of many of the patterns found on animal fur, scales and skin and shows a method for placing these patterns on complex surfaces. Examples of such patterns include the clusters of spots on leopards and jaguars called rosettes, the large, plate-like spots on giraffes and the stripes-within-stripes found on the lionfish. This dissertation explores the patterns that can be created by simulating a chemical mechanism called *reaction-diffusion*. This is a process in which several chemicals diffuse over a surface and react with one another to produce stable patterns of chemical concentration. Reaction-diffusion is a model of pattern formation that developmental biologists have proposed to explain some of the cell patterns that are laid down during embryo development. This dissertation demonstrates that simulation of a reaction-diffusion system can be used to create *synthetic texture*, that is, patterns on the surfaces of computer models. As an example of this, the top of Figure 1.1 shows a horse model with a white surface and the bottom shows this same model with zebra stripes created by reaction-diffusion.

My thesis is:

> *Noticeably improved biological textures on complex surfaces can be generated by first tessellating a surface into a mesh of fairly uniform regions and then simulating a reaction-diffusion system on that mesh to create a final texture.*

The field of computer graphics has been quite successful at creating models and images of man-made objects. Most manufactured objects have surfaces that are fairly smooth and that have very little color variation, and such objects are captured well by current graphics techniques. More of a challenge to computer graphics is simulating natural scenes that contain surfaces with rich patterns of color and surface detail. Nature abounds with complex patterns: the ruggedness of a mountain range, the cracking pattern of tree bark, the blades of grass in a field, the veins of a leaf, the pattern of bristles on a fly. This dissertation is directed towards capturing some of the richness and complexity of patterns found in nature. The patterns presented here take us a step closer to creating convincing natural scenes by simulating a biological model of pattern formation (reaction-diffusion) to create an array of textures that resemble patterns found in nature.

The advantages of using simulations of reaction-diffusion to create patterns are both in the natural look of the patterns and also in the natural way in which these textures can be fit over

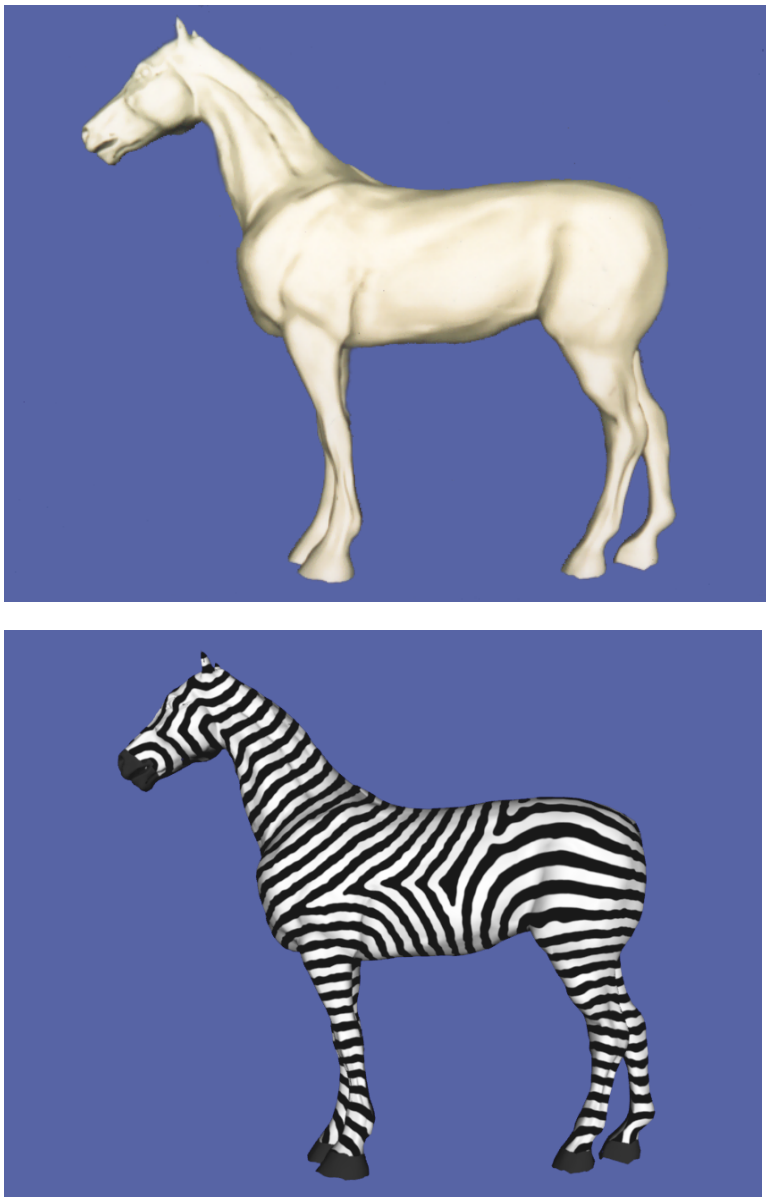

Figure 1.1: Stripe texture created using reaction-diffusion. Top is an untextured horse and the bottom shows zebra stripes on the same model.

the surface of a particular model. This gives substantially improved images over previous procedural texturing methods for creating images of animals that have color variation across their fur or skin.

What advantages might there be of using a model that is biologically inspired to create realistic images? Often the computer generated images that look the most real are those that take into account as much knowledge as is possible about the real world. For instance, the highlights on shiny surfaces originally were created using ad-hoc formulas for light reflection

[Phong 75]. Now more convincing highlights are made by taking into account the microscopic features of a surface [Blinn 77] [He et al 91]. Computer graphics researchers learn about optics and the properties of materials to better simulate how light illuminates surfaces. As another example, consider the convincing images of plants that have been created by taking into account the measured sizes and growth patterns of the leaves and branches of specific plants [de Reffye 88]. Here again, more knowledge of the real world results in more convincing images. Similarly, to create better looking synthetic patterns, we can try to understand how patterns of color form in nature. In particular, biologists have shown how reaction-diffusion systems can produce a wide number of the patterns that are found on animals; these patterns can then be used to texture the surfaces of objects in computer generated scenes.

There are three major issues in texturing that guide much of the computer graphics work on textures: user control, image realism and image quality. *User control* is concerned with what influence users can exert over the final image. Although we would like not to burden the user with too many tasks and choices while creating a textured object, they should be able to guide the look and placement of the texture. The controls provided to them should be simple and intuitive, just as the accelerator and brakes in a car are easy to understand and to use. Another important issue in texturing is *realism*. The patterns of the textures should look as close as possible to textures in the real world. These textures should not be stretched or distorted and should not show unnatural seams. Related to the issue of realism is *image quality*. The most important aspect of texture quality in a final image is how well the texture is filtered. That is, how well is the texture represented on the screen, especially when the texture is magnified or compressed. Texture realism, quality and user control are issues that will come up often in this dissertation.

1.2 Overview of Dissertation

This dissertation demonstrates that reaction-diffusion can be integrated with each of the stages needed to create an image that contains textured objects. The dissertation is organized as follows:

Synthetic Texturing (Chapter 1): This chapter gives a definition of texture and outlines the three steps needed to create a synthetic texture. Much of the literature on textures in computer graphics is surveyed here. An idealized system is outlined for creating textures using reaction-diffusion. This is a system that may be built in the future using the techniques presented in this dissertation. Chapters 3, 4 and 5 present new results that take significant steps towards realizing such a texturing system.

Developmental Biology and Reaction-Diffusion (Chapter 2): This chapter gives an overview of various models of animal development. Much of this chapter is devoted to two views of the cell: cell actions and the different forms of information available to a cell. This gives two perspectives that are useful for understanding the formation of patterns in a developing embryo, and several pattern formation models are discussed. In particular, this chapter introduces the reaction-diffusion model of pattern formation.

Reaction-Diffusion Patterns (Chapter 3): This chapter is concerned with creation of specific patterns using reaction-diffusion. It begins by providing a more detailed view of the literature on reaction-diffusion in developmental biology than is given in the previous chapter. It then demonstrates that a cascade of more than one reaction-diffusion system can be used to create a variety of patterns found in nature. Using multiple reaction-diffusion systems to create patterns is a new result presented in this dissertation.

Mesh Generation for Reaction-Diffusion (Chapter 4): This chapter presents a method for creating a reaction-diffusion texture that is tailored to fit a given surface. This is a central contribution of this dissertation. The chapter shows that a fairly uniform mesh of cells can be created over an arbitrary polygonal model. Such a mesh can be used to simulate reaction-diffusion systems for texture creation. To build such a mesh, first a set of points is evenly distributed over a model by causing the points to repel one another. Then the final simulation mesh is constructed by building the Voronoi regions surrounding each of these points. A reaction-diffusion simulation can then be carried out over this mesh to produce patterns of chemical concentration. The final patterns of concentration fit the geometry of the given model. Also, this method allows users to control the way in which feature size or other parameters are to vary over the surface of a model.

Rendering Reaction-Diffusion Textures (Chapter 5): This chapter describes two new ways of rendering reaction-diffusion textures. Textures are created by interpreting as colors the patterns of chemical concentration over an irregular mesh whose creation was described in the previous chapter. The high-quality (but relatively slow) method averages the colors of nearby mesh points to give the color at a particular pixel. This averaging of mesh points is based on a cubic weighting function that falls off smoothly with distance. A faster (but lower-quality) way to render such textures is to re-tile a model based on the simulation mesh and color the triangles from the re-tiling to create the pattern. Creating a hierarchy of meshes can speed up both these methods of rendering.

1.2.1 Guide to Related Work

Because this dissertation touches on a number of issues in graphics, there are many related articles in the computer graphics literature. For ease of understanding, the description of a particular article is placed at the beginning of the chapter where the article's results are the most relevant. Chapter 1 describes previous work on texture synthesis and texture mapping. Chapter 2 gives pointers into the literature of developmental biology that relates to pattern formation. This is standard material in developmental biology that is organized in a manner that is most relevant to pattern creation for computer graphics. Chapter 3 gives an overview of the work done on reaction-diffusion patterns in the biology literature. Chapter 4 describes other work done in computer graphics on mapping of reaction-diffusion textures [Witkin and Kass 91]. Chapter 5 covers the literature on texture filtering.

1.3 A Definition of Texture with Examples

Before discussing the details of synthetic texturing it is useful to have a working definition of *texture* in the context of creating images by computer:

Texture *is any change in surface appearance across an object that can be effectively separated from the gross geometry of that object.*

The most straightforward way that a surface's appearance can change is to have its color vary at different positions on the surface. A picture on the wall of a living room is a good example of a texture that varies the color of a surface. The picture can be separated from the description of the room's geometry by storing the picture in a two-dimensional array of colors.

Computer-generated textures were first used in early flight simulators built by General Electric for NASA [Schachter 83]. The motivation for using textures in these simulators was to give cues for judging distances during flight training. The first of these simulators, completed in 1963, displayed no more than a patterned ground plane and sky in perspective. These were two-color textures created by combining four levels of square mosaic patterns using modulo 2 addition. The more detailed pattern components faded with distance, but no attempt was made to smooth the edges between the square components of the texture. This resulted in distracting jagged edges in the texture. Textures provide important visual cues in flight training, and nearly all modern flight simulators incorporate textures in scene generation.

Texture mapping was introduced to graphics in its modern form by the work of Ed Catmull [Catmull 74]. Catmull's work centered on the display of objects that are described by cubic surface patches. His algorithm for creating images used a *depth-buffer*, which is an array of distance values for each pixel in the final image. The display algorithm divides patches into smaller and smaller pieces until they are the size of a pixel, and then places the color of a small patch into the final image if its distance is less than the currently stored depth in the depth-

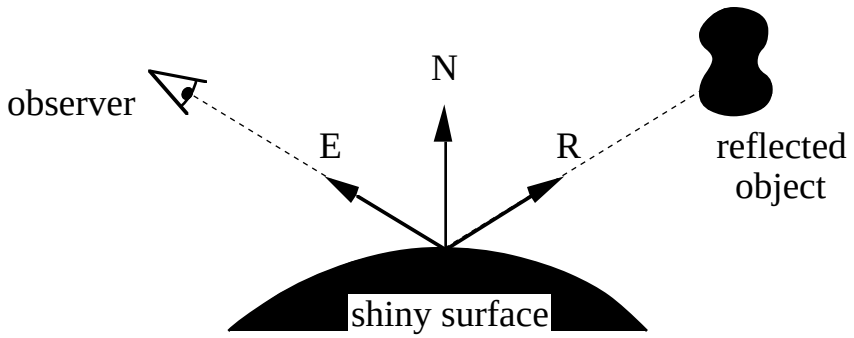

Figure 1.2: Light interacting with perfectly shiny surface.

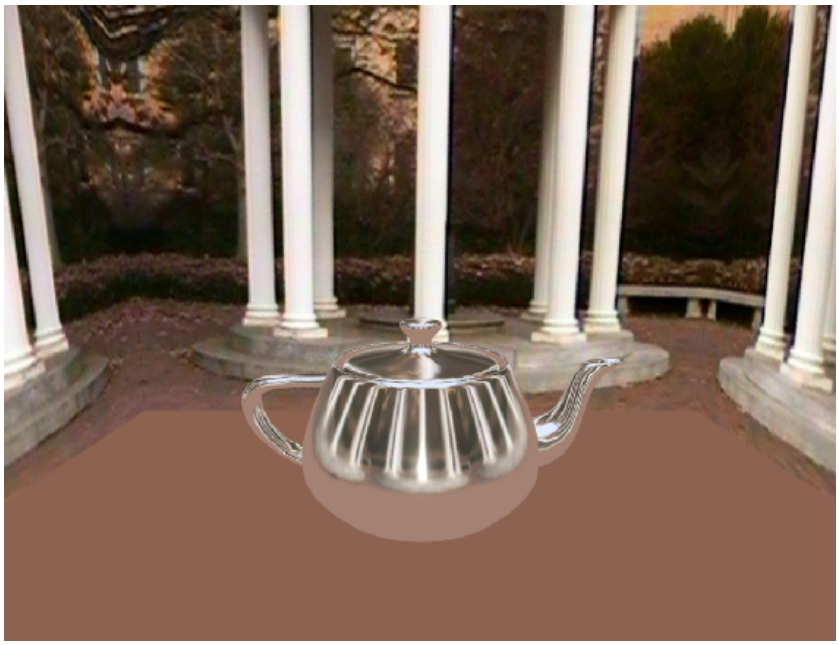

Figure 1.3: Example of environment mapping. The Utah Teapot reflecting the UNC Old Well

buffer. The patches are described as cubic functions that map a square in 2D parameter space into 3D positions. Textures are placed on a patch by creating a correspondence between a rectangular stored array of colors and this 2D parameter space. The parameter limits for each patch are kept during the subdivision process, and the texture color is averaged over these parameter limits to give the color at a pixel.

Textures can also be used to change surface properties other than color. For instance, textures can be used to approximate the way in which light reflects off a perfectly shiny surface. With mirrored surfaces, light bounces off the surface much like a pool ball bounces off the side of a billiard table. This is shown in Figure 1.2, where we can find what light travels in the direction *E* towards the observer's eye by tracing the path of a ray away from the surface in the reflected direction *R*. The color of the surface that this ray would strike is the color that an observer would see at this location on the mirrored object. Performing this ray tracing is computationally expensive when compared with the simple intensity calculation used for the diffuse surface [Whitted 80]. A simple and fast way to approximate this effect is given in [Blinn and Newell 76]. We can make the simplifying assumption that the objects surrounding the reflecting surface are far from the surface. We can then compute an *environment map* from the viewpoint of the object. The environment map is a collection of six rendered views of the world surrounding the object, one view for each face of a cube surrounding the mirrored object. Now when the mirrored object is rendered, the environment map is consulted to find the reflected color instead of performing a costly ray trace through the world. Figure 1.3 shows the use of environment mapping to simulate the effect of reflection on a shiny teapot. Here again, surface appearance is separated from geometry, in this case by storing the distant environment in a texture.

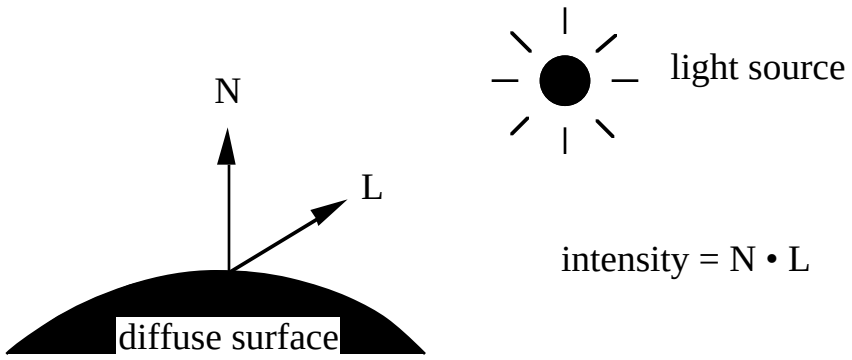

Figure 1.4: Diagram of how light interacts at a location on a diffuse surface.

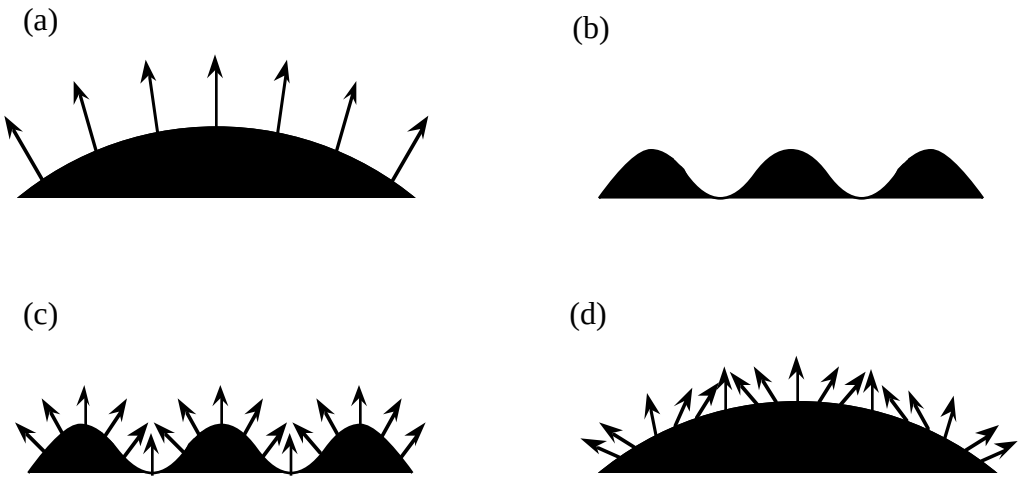

Figure 1.5: Using bump mapping to modify surface normals.

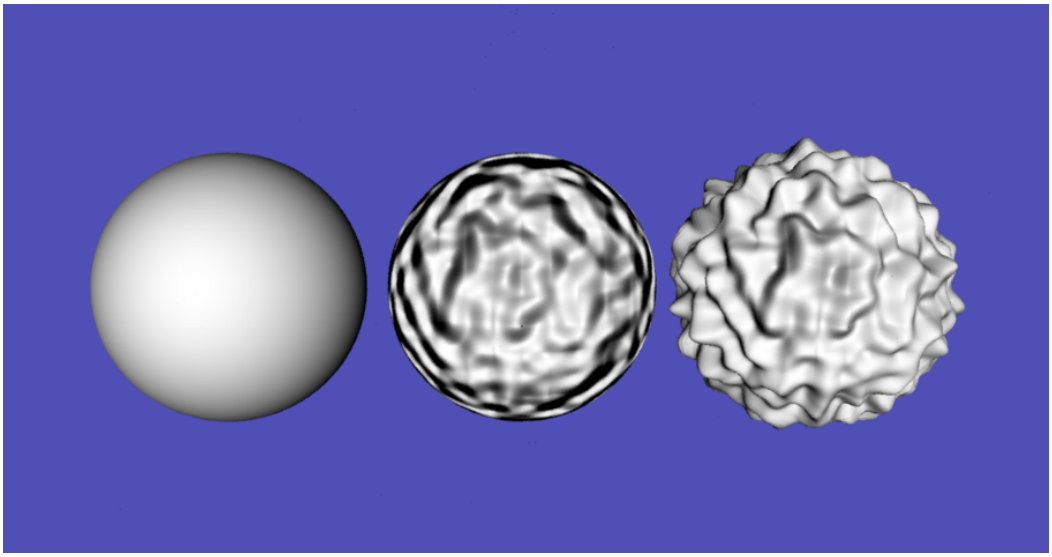

Figure 1.6: Example of bump mapping (center) and displacement mapping (right).

Another property that a texture can modify is the normal vector to the surface [Blinn 78]. To understand what this will do, we first need to examine a simple lighting model. Let us assume that we have a surface that is a completely diffuse reflector, that is, assume that light striking the surface will be scattered equally in all directions, regardless of the direction of the light source. Figure 1.4 shows a small piece of a surface, where the unit vector N points in the normal direction (directly outward from the surface) and the unit vector L points towards the light. For a totally diffuse surface, the intensity I of light leaving this surface in any direction is proportional to the dot product $N \bullet L$. This means that more light impinges upon and leaves a diffuse surface when the direction of the light is nearly pointing head-on to the surface. If the light is nearly grazing a diffuse surface then the surface will catch and reflect very little of the light.

We are now ready to see what change in appearance will occur when the surface normals are changed based on a texture. The idea of *bump mapping*, described in [Blinn 78], is that a two-dimensional array of scalar values can be used to represent small height changes to a surface. Blinn found that a surface can be made to look bumpy if the surface normals are perturbed based on these height values, and that the actual position of the surface doesn't need to be altered to give convincing bumps. Figure 1.5 shows an example of this. Part (a) of the figure represents a one-dimensional slice of a flat surface. Part (b) shows a height field that will modify the left surface, and part (c) shows the new height that results from putting together parts (a) and (b). Part (d) shows the original surface shape with the modified surface normals. Rendering the surface using the modified surface normals produces patterns of light and dark across the surface that are convincingly like what actual bumps would look like, yet the surface actually remains smooth. Figure 1.6 shows an un-textured sphere on the left and a bump mapped sphere in the center. It is the change in the normal direction N as it is used in the lighting model that gives the change in intensity over the center surface. Notice that the

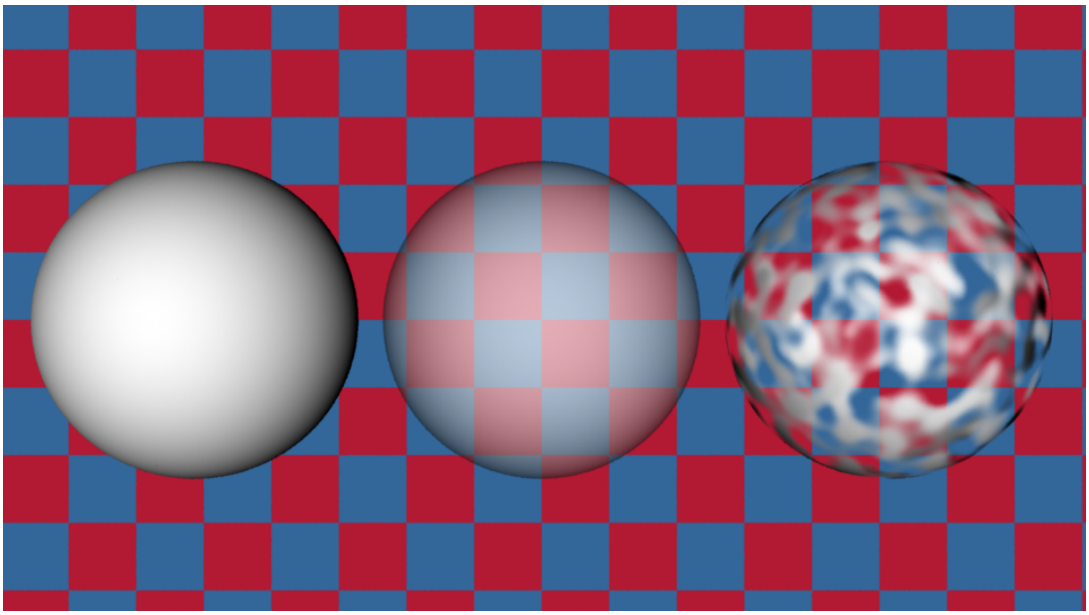

Figure 1.7: Transparent texturing.

silhouette of the center surface is still smooth because the geometry has not been changed, only the surface normals are altered during the lighting calculations.

Another surface characteristic that can be modified by textures is transparency. Figure 1.7 shows a translucent sphere, an opaque sphere and a sphere whose transparency has been modulated by a simple texture. Gardner has used transparency-altering textures to produce realistic images of clouds [Gardner 85].

If textures can be used to change the color, reflection, normal direction, and transparency of a surface, then can textures also be used to modify the actual position of a surface? The answer is yes, and the technique is called *displacement mapping* [Cook 84]. The right portion of Figure 1.6 shows an a sphere whose surface position has been altered by a displacement map. Notice that not only does the intensity vary over the surface as it does with bump mapping, but the silhouettes are also bumpy. This is because the position of the surface has been changed by a displacement map prior to hidden surface elimination. Displacement mapping separates the fine geometric detail of an object from its gross shape. Notice that displacement mapping satisfies the definition of texture given above.

1.4 The Three Steps to Texturing (Previous Work)

The process of texturing a surface can be broken down into three stages: 1) texture acquisition, 2) texture mapping, and 3) texture rendering. As an example of these three stages, consider the cubic surface patch in the left portion of Figure 1.8. The image in the middle of this figure was captured using a digital camera. Next, the mapping stage consists of finding a function that maps points in the texture's space onto points of the surface being textured. In this case, the texture's space is the two-dimensional rectangle that contains color values of the image. This space can be indexed by a two-dimensional vector (u,v). The cubic

Figure 1.8: Texturing a cubic surface patch.

surface is a patch that is defined by mapping parametric values in the unit square $[0,1] \times [0,1]$ into three-space. Positions on the patch correspond to two-dimensional coordinates (s,t) in this unit square. The texture can be mapped onto the surface patch simply by identifying the texture coordinates (u,v) with the parametric coordinates (s,t). Finally, the actual rendering of the texture must bring together the texture definition, the texture mapping function and the reflective properties of the surface to create a final textured image, shown at right in the figure.

Below we will discuss each of these stages of texturing. Notice that decisions about one of the stages will often influence the methods chosen for the other stages in the texturing process.

1.4.1 Texture Acquisition

How a texture is acquired for a given object depends on the kind of texture being used. There are currently two broad classes of textures recognized in computer graphics: *image-based textures* and *procedural textures.* Image-based textures are stored arrays of color values that represent a particular picture or pattern. Using a digital paint program is a fairly common way of defining image-based textures. Another way to capture an image is to use an image scanner or a digital camera. The photographic image of Figure 1.8 is an example of this. Another method of creating an image-based texture is to render a synthetic image and use the resulting picture as a texture. All of these methods create two-dimensional arrays of gray-scale or color values that represent a picture or a pattern.

Quite a different approach for defining a texture is to give a mathematical description of a pattern. Such a texture is known as a *procedural texture*, and creating such textures is often called *texture synthesis*. Several methods have been proposed that use composition of various functions to generate textures. Gardner introduced the idea of summing a small number of sine waves of different periods, phases and amplitudes to create a texture [Gardner 84]. Pure sine waves generate fairly bland textures, so Gardner uses the values of the low period waves to perturb the shape of the higher period waves. This method gives textures that are evocative of patterns found in nature such as those of clouds and trees. Perlin uses band-limited noise as the basic function from which to construct textures [Perlin 85]. He has shown that a wide variety of textures (stucco, wrinkles, marble, fire) can be created by manipulating such a noise function in various ways. [Lewis 89] describes several methods for generating noise functions to be used for texture synthesis.

Blinn and Newell briefly described how a texture can be created by specifying a two-dimensional frequency spectrum and taking its inverse Fourier Transform to make a final pattern [Blinn and Newell 76]. Lewis expanded on this idea to make a method of creating textures that is halfway between digital painting and procedural texture creation [Lewis 84]. He demonstrated how a user can paint an “image” in the frequency domain and then take the Inverse Fourier Transform to create the final texture. He shows how textures such as canvas and wood grain can be created by this method.

None of the methods for texture synthesis described above attempt to model the actual

physical processes that produce patterns in the real world. Some of the textures generated by function composition produce images that look quite real, but some physical phenomena are likely to prove too difficult to mimic without modeling the underlying processes that create the texture. One example of this is the wood solid texture demonstrated by Peachy that mimics the way trees create concentric rings of dark and light wood [Peachy 85]. Another example of modelling natural processes are the way different wave functions can be summed to produce the effect of water waves [Schachter 80] [Max 81]. Still another example of using physical simulation for texture creation is the dynamic cloud patterns of Jupiter in the movie 2010 [Yaeger and Upson 86]. Another example of how physical simulation can be used to generate textures is the texture synthesis method using reaction-diffusion presented in this dissertation. Creating patterns by physical simulation of natural processes is a rather involved method of defining textures procedurally.

An important issue that arises in making procedural textures is how to guide the synthesis to give a desired pattern. For instance, we might want to use Gardner's sum-of-sines to make a texture of cirrus clouds. How many sine functions should we use and how large should we make the sinewave amplitudes? Typically, we would find ourselves trying out different values for parameters of the texture function until we get a pattern we like. We would like to get a "feel" for which parameters changed particular aspects of a procedural texture function. This kind of parameter exploration can be made more pleasant if the textures can be created and displayed rapidly. For instance, some procedural textures can be generated at a rate of several frames per second on the Pixel-Planes graphics engine [Rhoades et al 92]. Rhoades and his co-authors describe a system in which a user can change texture parameters and cause dynamical updating of textured objects by turning a joystick knob or moving a slider bar. The issue of user parameter control must be addressed for any procedural method of texture creation, and we will return to this issue when we look at the parameters for reaction-diffusion textures.

1.4.2 Texture Mapping

Once a texture has been created, a method is needed to map it onto the surface to be textured. Often an image texture is represented as a two-dimensional array of color values, and one can think of such a texture as resting in a rectangle $[0,m]\times[0,n]$ in the plane. Mapping these values onto a complex surface is not easy, and several methods have been proposed to accomplish this. A common approach is to define a mapping from the unit square to the natural coordinate system of the target object's surface. For example, latitude and longitude can be used to define a mapping onto a sphere, and parametric coordinates may be used when mapping a texture onto a cubic patch [Catmull 74].

In some cases an object might be covered by multiple patches, and in these instances care must be taken to make the patterns match at the common seams of adjacent patches. For example, suppose a model of a cow's body was composed of five large patches, one patch for each of the four legs and a fifth patch for the rest of the body. Further suppose there are separate dappling textures for each of these patches. A leg patch must have a pattern that matches itself where opposite edges join together from the patch wrapping around the leg.

Furthermore, the patterns of each leg must match the pattern of the body patch where the leg meets the cow's body. A successful example of matching textures across patches has been shown for the bark texture of a maple tree [Bloomenthal 85].

An important issue in texture mapping is finding a way to give the user control over how a texture is placed on a surface. One method of mapping that offers the user some amount of control is to create a projection of the texture onto the surface of the object. An example of this approach is to allow the user to orient the texture rectangle in 3-space and perform a projection from this rectangle onto the surface [Peachey 85]. Related to this is a two-step texture mapping method given by [Bier and Sloan 86]. The first step maps the texture onto a simple intermediate surface in 3-space such as a box or cylinder. The second step projects the texture from this surface onto the target object.

A different method of texture mapping is to make use of the polygonal nature of many graphical models. This approach was taken by [Samek 86], where the surface of a polygonal object is unfolded onto the plane one or more times and the average of the unfolded positions of each vertex is used to determine texture placement. A user can adjust the mapping by specifying where to begin the unfolding of the polygonal object.

Each of the above methods have been used with success for some models and textures. There are pitfalls to these methods, however. Each of the methods can cause a texture to be distorted because there is often no natural map from a rectangle to an object's surface. This is a fundamental problem that comes from defining the texture pattern over a geometry that is different than that of the object to be textured. One solution to this problem can be found in a sub-class of procedural textures known as solid textures.

A *solid texture* is a color function defined over a portion of 3-space, and such a texture is easily mapped onto the surfaces of objects [Peachey 85] [Perlin 85]. A point (x,y,z) on the surface of an object is colored by the value of the solid texture function at this point in space. This method is well suited for simulating objects that are formed from a solid piece of material such as a block of wood or a slab of marble. Solid texturing is a successful technique because the texture function matches the geometry of the material being simulated, namely the geometry of 3-space.

1.4.3 Texture Rendering

The purpose of texturing is to create images of synthetic objects that are visually enhanced by the surface patterns given by the textures. *Rendering* a textured object is the process of bringing together the information about the texture, the mapping of the texture and the object's geometry to create a final image. Much of the work in texturing has concentrated on minimizing the visual artifacts that can arise in texture rendering.

The most noticeable texturing artifacts result from a phenomenon known as *aliasing*. Aliasing occurs when high-frequency patterns appear as if they are lower-frequency patterns. This can be seen in the checkerboard at the left of Figure 1.9. The moiré patterns near the

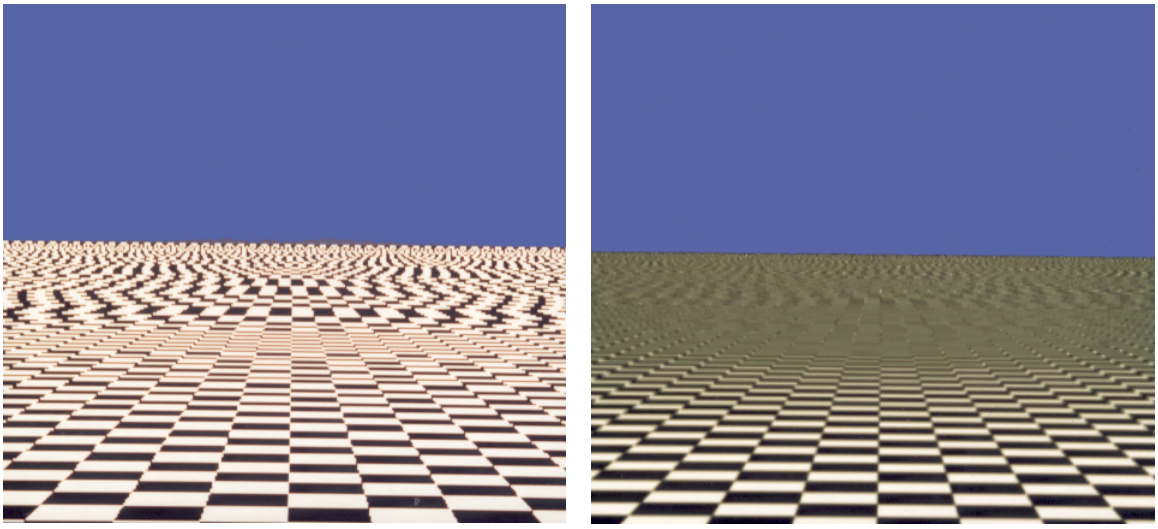

Figure 1.9: Aliased checkerboard (left) and anti-aliased version (right).

horizon illustrate that the high frequencies from many squares bunched close together can look as though there is a lower-frequency pattern with fewer features in a given region. The jagged edges between the squares is another example of aliasing. One way to minimize aliasing artifacts is to *filter* the texture, averaging the color values in an area of the texture to determine the color at a given pixel. There is a large body of literature on texture filtering, and good starting points in this literature are [Heckbert 89] and [Mitchell and Netravali 88].

One way to think about the issue of texture filtering is to consider what needs to be done to faithfully render a texture in extreme scene instances. For example, consider what should happen when a textured object is far away in a given scene so that it covers very few pixels in the image. In this case, many texture elements should contribute to the color of a single pixel. Averaging the color values from many texture elements means filtering the texture. A filter that averages together many color values from a texture is known as a *decimation filter*. The opposite extreme is when an object is very near the viewer in a scene, so that a single texture element may cover many pixels in the final image. The surface pattern will look blocky if care is not taken, revealing the discrete nature of the stored texture. Averaging between several nearby color values can make the underlying grid of the texture less apparent. A filter that reconstructs a color value from very few adjacent texture elements is called a *magnification filter*.

The extremes of magnification and decimation do not give the whole story of texture filtering. There are many issues to consider when deciding how texture values should be averaged together to contribute to a final image. This topic will be considered in more detail in Chapter 5 as a preliminary to how reaction-diffusion textures can be rendered. All the issues described above are relevant to rendering of patterns created by reaction-diffusion.

1.5 An Idealized System

Now that we have examined the topics of texture synthesis, mapping, and rendering, we can consider how all these methods may be brought together in practice. This section describes a session that a user might have with a hypothetical system for texture creation. This is an idealized system for texturing objects using reaction-diffusion. The example will serve to highlight the major issues involved in texturing a model with a reaction-diffusion pattern. Let us assume that a user has a polygonal model of a large cat and wants to fit a leopard spot pattern onto this model. Traditionally, the user would create a spot pattern on a rectangular surface (probably using a digital paint program) and then painstakingly work to map this image texture onto the surface of the cat. Using this traditional approach, the user would have trouble avoiding visible seams when wrapping the texture around the legs and would also encounter problems when trying to make the pattern uninterrupted at the junction between the legs and main body. As an alternative to this method, let us see how we might use reaction-diffusion to place a spot pattern on the cat model.

The first step is to find a reaction-diffusion system or a cascade of such systems that produces the desired pattern of spot clusters. The user picks a pattern that most closely resembles the desired pattern from a catalog of cascade patterns. Perhaps the spots look too regular, so the user decides to increase the randomness in the underlying chemical substrate. A simulation of this on a small grid gives the look the user wants, and now he or she begins to make decisions about how the pattern's scale should vary across the model. The leopard should have smaller spots on the legs and head than on the main portion of the body. The user specifies this by selecting key points on the head and legs of an un-textured image of the cat and indicating that at these locations the spots should be roughly three centimeters in diameter. Likewise, key positions on the cat's body are selected to have six centimeter spots. These parameters are automatically interpolated across the model's surface, and the values are bundled together with the parameters of the cascade process and sent to a program that generates the final texture on the surface of the cat. The program creates a mesh over the surface of the model and then simulates the given reaction-diffusion process on the mesh to create the spot pattern.

There are two ways that the resulting pattern can be used to create a final image. The first method is to create a re-tiled version of the polygonal model based on the simulation mesh. The polygons of this re-tiled model are then colored based on the chemical concentration given by the reaction-diffusion simulation. The user can view and manipulate this polygonal model on a graphics workstation. For a higher quality image, the user can invoke a renderer that has been enhanced to display textures directly from a texture mesh. Such a renderer uses a cubic weighted average of mesh values to avoid visual artifacts resulting from the discrete nature of the underlying mesh. If the cat is positioned far from the viewer, the renderer may generate the cat's image from a more coarse representation of the model's geometry, and it may also use a version of the texture where the higher frequency components have been eliminated. Using such simplified models speeds the rendering process without noticeably affecting the image quality.

There are four issues that the above example illustrates:

- creating complex patterns with reaction-diffusion
- specifying how parameters vary across a surface
- generating a texture that fits the geometry of a model
- rendering images efficiently and without artifacts

The remainder of this dissertation tells how each of these issues in texturing with reaction-diffusion patterns can be addressed.

2 Pattern Formation in Developmental Biology

The goal of this chapter is to provide an overview of pattern formation models in developmental biology. This will provide context for understanding the specific developmental model of reaction-diffusion and also present other models of pattern formation. Some of these other models may also prove useful to computer graphics. The chapter begins by giving an overview of the important issues in embryo development. Further details should be sought in an introductory text on developmental biology such as [Gilbert 88]. The remainder of the chapter describes models of pattern formation, describing them according to the information cells are thought to receive and actions that cells can perform.

A major goal of developmental biology is to explain how the seemingly unstructured object that is the fertilized egg can, over time, change into a complex organism such as a fruit fly or a human being. The first few stages of development of an embryo are very similar for most multi-cellular animals, from simple worms to amphibians, birds, and mammals. During early stages of development, the cells of the embryo undergo numerous divisions to create many cells. At the same time, these cells are arranging themselves spatially to lay out the basic body plan of the animal. The fates of these cells are also being decided during this process. A cell of the early embryo has the possibility of becoming almost any cell type. Later in the developmental process, however, a given cell has fewer possible fates with respect to its ultimate cell type.

The central issue of animal development is the arrangement of all the cells into their proper positions in the embryo according to cell type. There are roughly 200 different cell types in humans [Alberts et al 89], and during development all of these cells must be positioned correctly in the embryo. For instance, pigment cells should find their way to the skin, and the cells of the central nervous system must come to be in the spinal column or brain. Each of the cells ends up in its proper place within the animal, and this occurs either through cell movement or by cells becoming specialized based on their position. Developmental biology seeks to understand the mechanisms that produce this geometric arrangement of cells according to cell type. We will need an understanding of some of the components of a cell and their functions before we return to this issue.

2.1 Introduction to the Cell

Biologists commonly view the embryo as a collection of cells. One reason for this view is that a cell is the smallest portion of an embryo that can act somewhat independently of other

parts of the developing organism. In mammals, for example, the macrophage cells act in such an independent fashion that they almost seem to be whole, single-celled creatures that freely travel through our circulatory system and the surrounding tissues. A cell's plasma membrane is a clear division between the cell and the environment outside of the cell. It is this membrane that allows a cell to regulate the concentration of proteins and ions present inside its cytoplasm. The chromosomes, the DNA within the nucleus of the cell, store the vast majority of the information needed to guide the actions of the cell, including those actions necessary for development. The synthesis of proteins within the cell is dictated by the DNA. Together, the plasma membrane, the DNA and the many proteins in a cell can be thought of as the skin, the brain and the active and structural parts of a cell that give the cell a separate identity.

The information in DNA includes directions for the general actions and upkeep of the cell and also directions for the cell's embryonic development. The information takes the form of genes that control the production of specific proteins and RNA. Protein are created by the processes of *transcription* and *translation. Transcription* is the creation of an RNA molecule that is a copy of the information present in a gene. When this RNA codes for a protein, it is called a *messenger* RNA or mRNA. mRNA's are an intermediate form of information along the pathway from DNA to protein. An mRNA molecule encodes the information controlling the amino acid sequence from which one or more protein molecules may be produced. The process of reading the mRNA and forming a protein molecule is called *translation.*

Transcription is pivotal in embryo development. Control of transcription is a large part of the control over what proteins are produced by a given cell. The difference between one cell type and another is a difference in the proteins that are being manufactured within the cells. All cells have the same information contained in their DNA, so the differences between cells are a result of which genes are being transcribed within each cell. Thus cells of a given tissue type can be characterized by which genes are turned "on" (which are being transcribed) and which are turned "off."

Throughout development, the cells in the embryo become increasingly specialized. Any single cell from the four-cells stage of a sea urchin embryo can develop to produce an complete sea urchin if it is separated from the other cells. This shows that such an early cell is unspecialized. Later in development, a cell separated from the rest of the embryo is quite incapable of developing into a whole animal and will instead form abnormal structures that resemble parts of an animal. Such a cell is said to be *determined*, which means that the cell has travelled some distance down the path of increased specialization. Determination of a cell is closely related to what genes are being transcribed in the given cell. To give another example, a cell of the neural crest might become either a pigment cell or a neuron depending on later events, but it cannot become a blood cell. It is presumed that these cells have particular genes that are turned on or off, and that this limits the fate of these cells. Thus we can view development as the control of gene transcription in order to create the geometry of the embryo.

2.2 Animal Development

We are now in a better position to understand the central question of animal development: How are cells geometrically arranged according to cell type? This is a question about the way genes control the geometry of an embryo. How can the genes (and the proteins that the genes encode) dictate the final geometry of an animal? It is known that there are groups of genes that are somehow responsible for placing the legs of a fly on its thorax instead of on its head. What signals are present in an embryo to guide the placement of the thumb of the left hand, in contrast to its final position in the right hand? In the regeneration of a salamander arm, the nearer portion grows back first and the hand and fingers are the last parts to re-form. What mechanism keeps track of the parts that have already been regenerated and what part should be created next?

There are two basic mechanisms for placing a cell correctly in the embryo according to its tissue type. One way is for the cell to *move* to the correct position in the embryo. This can mean that a cell actually crawls through the embryo to the correct position, a process known as *migration.* This can also mean that the embryo folds, stretches or otherwise changes its shape to bring the cells to their proper location. The second way a cell can come to be positioned correctly is for the cell to somehow "know" where it is and for it to change into the proper cell type based on this information. This means that a cell may differentiate based on geometric information.

The process of development is a complex mixture of the two mechanisms of cell placement described above. Cells do not purely rely on movement to arrive at their proper position, nor do all cells remain stationary and change their cell type based on position. Development is a complex web of events that can involve both basic mechanisms. For instance, a group of cells may differentiate into different cell types based on their position along the length of embryo, then some of these cells may migrate to a new location, and finally these cells might further differentiate based on their new position. In the sections that follow, it will be helpful to keep in mind these two broad mechanisms of development. Cell movement and differentiation based on information about position *both* contribute to the embryo's development.

The remainder of this chapter is divided into three sections. The first of these sections describes in more detail some of the actions a cell may take during the course of development. This will give us a better understanding of how cells participate in shaping an embryo. The next section describes the forms of information available to cells in a developing embryo. A cell may use information to guide its migration or to decide how to differentiate further. The final section describes how the information available to a cell and the actions that a cell may perform can be brought together to lay down patterns in an embryo.

2.3 Cell Actions

There are two complementary aspects that are essential to a cell's participation in development: the *actions* a cell can perform to influence development and the *information* on which

the cell bases its actions. Many of the actions a cell can perform can directly influence the form of an embryo. Examples of such actions include changing cell shape or size, migrating, dividing, and dying. Some actions, however, have a rather indirect influence on development; examples include the internal change in the state of a cell (determination) and the secretion of chemicals that will later influence development. Both direct and indirect mechanisms of change are discussed in more detail below.

2.3.1 Cell Division

Cell division is the most conspicuous mechanism of change in the very early embryo. The first task of the fertilized egg is to make a rapid transition from an egg to an embryo with many cells. It is only when the embryo is composed of a number of cells that it can begin to change its shape and that different portions of it can become specialized. Most of the cells in the early embryo divide rapidly, providing many cells with which to form patterns. An exception to this, however, is that the cells of the early embryo of vertebrates do not divide rapidly.

The *rate* of cell division is not the only contribution that division has on development, however. Also important is the *direction* in which a cell divides. The early cell divisions in an embryo are carefully oriented to aid in setting down the overall body plan. In the sea urchin, for example, cells that arise from what is known as the animal pole have a different fate than cells of the vegetal pole (see Figure 2.1). The first and second divisions of the sea urchin embryo split the egg into four cells that each have an animal and vegetal portion. Each

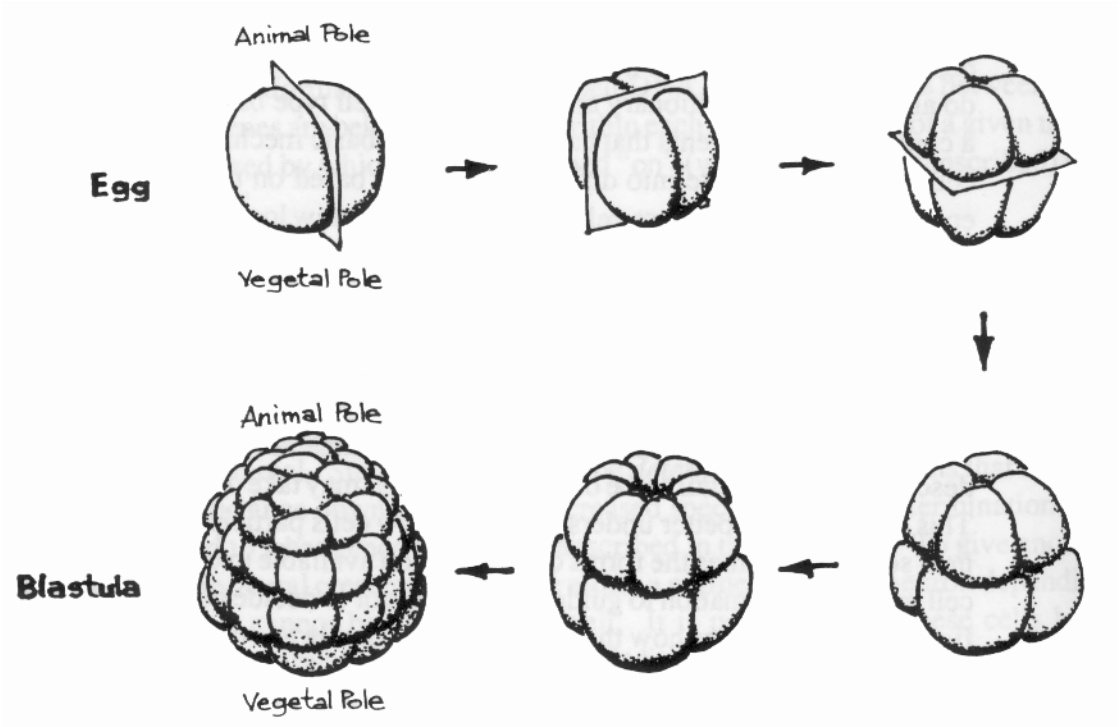

Figure 2.1: Cell division transforms the egg into a blastula.

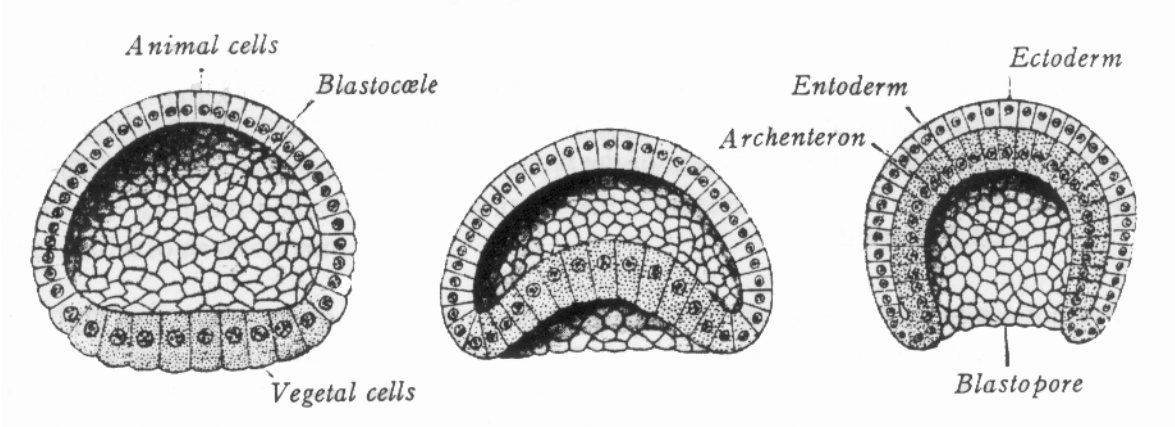

Figure 2.2: Gastrulation.

of these cells can go on to develop into a complete sea urchin if separated from the others. The third cell division then splits each of these four cells into an animal and a vegetal cell, defining much of the orientation of the embryo. None of the eight cells resulting from this division can create a whole normal embryo if separated from the other cells.

2.3.2 Shape Change

Another mechanism of development in the embryo is shape change of cells. An important example that emphasizes the importance of shape change can be found in early development in sea urchins and frogs. During *gastrulation* in these animals, the vegetal portion of the embryo flattens out and then fold in towards the hollow cavity called the blastocoel (Figure 2.2). The change in shape at the lower portion of the embryo is thought to be caused by the cells actively changing their shape. First, these cells become elongated in the direction perpendicular to the cell's surface. This flattens the embryo at the pole. Then these same cells become constricted on their outward-pointing end, giving them a characteristic pyramid

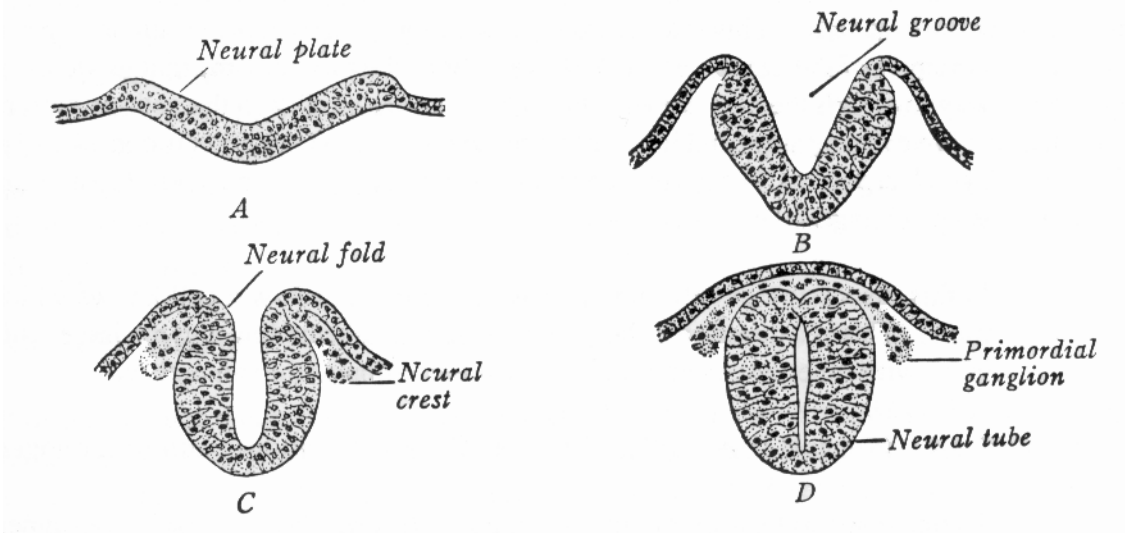

Figure 2.3: Formation of neural tube.

shape. This shape change causes the lower sheet of cells to fold in towards the hollow center of the embryo. These cells form the endothelial layer of cells that are destined to give rise to particular organs and tissues in the adult embryo. Similar changes in cell shape appear to contribute to the formation of the neural plate and the neural tube (Figure 2.3).

2.3.3 Migration

Cell migration is movement of a group of cells from one portion of an embryo (away from their cell neighbors) to another part of the developing animal. Migrating cells typically take on quite a different role at the destination site than the cells that were already present at the destination. This is a more curious developmental mechanism than cell division and cell shape change. Why do cells need to migrate, instead of having cells at the destination site take on the role of the migrators? The answer to this is probably not simple, but it is thought that much of the answer can be ascribed to historical, or evolutionary, reasons.

Whatever the reason for cell migration, it is a fact of development. One example of cell migration in sea urchin embryos is the movement of cells from the far vegetal pole into the blastocoel during gastrulation. These cells, called micromeres, separate themselves from their neighbors, crawl through the outer shell of the embryo, and individually take up new positions attached to the inner surfaces of the cells that compose the hollow blastocoel. The individual cell motions are distinct from the flattening and folding of the lower region that also is part of gastrulation. The cells arising from micromeres secrete the skeleton of the sea urchin.

Cell migration plays an important role in other aspects of development besides gastrulation. Another important migration occurs when many of the cells from the neural crest move to other portions of the embryo to become pigment cells and various cells in ganglia, glands, and teeth. Still another example of cell migration can be found in animals with a central nervous system. In these higher organisms, migration of the axons of neurons plays a central role in wiring the brain. This is a different kind of migration, where the cell body remains fixed and only part of the neuron actually moves. Both the mechanisms that guide the axons and the way in which the axons move through tissue are similar to the actions performed when an entire cell migrates, and it is for this reason that axon migration is considered a special form of cell migration. However, axon migration could also be considered an extreme form of cell shape change.

In this discussion of cell migration we have ignored how a cell knows which direction to travel. This is a question of how a cell acquires information, and the issue will be addressed later, with other issues about forms of information within an embryo.

2.3.4 Materials from Cells: The Cell Matrix, Hormones and Morphogens

There are a number of ways that a cell can help steer the course of development by creation and release of chemical products. An important example is the way in which cells create the support structures within an organism. The spaces between cells are composed of many

proteins and carbohydrates that comprise the *extracellular matrix*. The components of the extracellular matrix include *collagen*, a fibrous material that forms a strong framework for tissue, and *proteoglycans* that allow tissue to resist compression. Also part of the matrix are molecules that form potential adhesive sites, places where a cell can attach permanently, or temporarily as "hand-holds" during migration. All of these materials are created by cells and are essential to development.

There are other sorts of molecules that a cell can release that contribute to development. *Hormones* are chemicals that are released into the circulatory system that can trigger responses by cells at many points in an animal. For instance, ecdysone is an insect hormone that causes the transition from one larval stage to another or causes the final transition to insect adulthood. It causes the shedding of the insect's old exoskeleton to reveal the new, still soft exoskeleton of the next insect form. Hormones are chemical signals that regulate changes across the entire organism.

Another kind of chemical signal is called a *morphogen*, and, as the name implies, morphogens are signals for change of form. Although there is some debate within developmental biology about the scope of the term, we will interpret the word broadly. We will say a chemical is a morphogen when it freely diffuses through groups of cells or tissue and causes cells to follow different developmental pathways depending on its local concentration at each cell. Morphogens play a large role in two models of how patterns in an embryo are formed, and we will return to these models in section 2.5 on pattern formation.

2.3.5 Cell Death

A well-documented and widespread mechanism in development is the case of cells dying to aid in shaping the embryo. Cell death plays a major role in the shaping of hands and feet in birds and mammals. These extremities are shaped like paddles early in limb formation, without any separation of the digits. Then, cells in four regions on the hand-to-be die off, leaving four gaps between the portions of the hand that become the fingers. These cells are fated to die in order to help shape the hand or foot. Cell death is also important in the development of the nervous system. In vertebrate embryos, several motor neurons may initially form synapses with the same muscle cell. Then, through some form of competition, many of the neurons lose their connection with the muscle cell. The neurons that are left without a connection to a muscle cell then die. Here again, cell death is a normal part of the development process.

2.3.6 Determination

All of the cellular mechanisms of development described above are ways that a cell may interact with its embryonic environment. There is another cell mechanism that plays a vital role in development but that goes on entirely within a cell. *Determination* is an internal change of state in a cell that locks the cell into a particular course of development. The cells of the neural crest illustrate determination. At a certain point in development, these cells migrate to other locations in the embryo to become pigment cells, cells in the sensory ganglia,

or one of several other kinds of cells. Cell transplant studies have shown that a cell is not yet fated to become a particular cell type before it begins migration from the neural crest. Before migration the cell is said to be *undetermined*. Once the cell has migrated away from the neural crest and arrived at one of the possible destinations, that cell soon is locked into a particular pathway of change that fits the role it will play in the destination tissue. At that point the cell is said to be *determined*. It should be noted that there is often no one determination event for a particular cell, but instead a cell becomes more and more specialized throughout the course of development. Thus calling a cell determined or undetermined really should be viewed in the context of how specialized other cells are at the same point in development.

For cells migrating from the neural crest, the determination of the cell is triggered by cues from the environment at its destination. The act of determination, however, is an event that is internal to the cell. In some cases (probably most) the actual determination event is a change in the way proteins are bound to DNA often reinforced by slight modifications to the DNA itself (such as methylation). A change might either block or turn on the expression of a particular gene, and the protein coded by that gene may trigger any number of other events within the cell. Biologists speak of a cascade of gene expression, where one gene's activity causes many other genes to become active in a particular sequence. The details of such cascades of expression are only now becoming evident in a few specific cases. One such case is the cascade of expression that lays down the segmentation in the developing fruit fly.

2.4 Information that Guides a Cell

As mentioned earlier, the actions a cell may perform is only half the story about a cell's participation in development. Equally important are the various forms of information that guide a cell's actions. These forms of information include: chemical messages, haptotactic gradients (changes in stickiness), mechanical forces, electric fields and the internal state of a cell. Some of the forms of information described below (such as chemical messages) naturally correspond to particular actions of a cell (such as production of a chemical). Viewing these products again, as information instead of actions, gives us a useful additional perspective.

The discussion below is organized according to the *form* taken by information, that is, according to the information carrier. It will also be useful to consider the information *contents*. There are several different kinds of information important to a cell in a developing embryo. These include: guidance for migrating cells, information about position within the embryo for cells that remain stationary and information about the timing of events within the embryo. This view of information in an embryo will be especially useful in Chapter 3, where our goal is to simulate pattern formation using a computer model.

2.4.1 Chemical Messages

Morphogens and hormones are two kinds of chemical messages that guide development. Hormones are very simple chemical triggers that are meant to be received by many cells at once through an organism in order to synchronize an event, such as the casting off of an insect

exoskeleton, or to control the rate of a given process. A cell that detects the hormone is not concerned with where the message came from or the quantity of the chemical. The hormone is just a messenger that says it is time to perform a particular action. In contrast, morphogens are chemical messages that convey information about geometry. The morphogen typically conveys this geometric information by its variation in concentration (chemical gradient) across different portions of an embryo. A cell that detects a morphogen will travel down one of several developmental pathways depending on the morphogen concentration that it detects.

Probably the most clear examples of morphogens (in the above sense) are some of the proteins that diffuse through the early fruit fly embryo. Seven pairs of segments are formed during the course of development of a fruit fly larva. More than a dozen genes have been identified that participate in determining this pattern of segmentation of the larval fly. There are at least four distinct stages to this process of segment formation, and different genes that participate in defining the segments are active at different times during this process. The proteins produced by some of these genes have been shown to be present in graded concentrations along the head/tail axis of the embryo. In some cases it has been shown that the concentrations of such proteins cause different cells to follow different developmental pathways. Experiments have shown that if the concentrations of certain chemicals are changed then this disturbs later development of the embryo. See [Alberts et al 89] for an overview of this subject.

Hormones and morphogens usually deliver their messages to stationary cells. There is another form of chemical message that is used to guide cells that are in motion. *Chemotaxis* is the attraction of a cell to a particular chemical, called a *chemoattractant*. The job of a chemoattractant is geometric, like a morphogen, but in this case the task is to guide a migrating cell to a particular position. Biologists have identified potential instances of chemotaxis in developing embryos, but identifying the chemoattractants has proved to be difficult.

2.4.2 Haptotactic Gradients

Chemical messages are not the only form of information that may be used to guide a migrating cell. *Haptotaxis* is the attraction of a cell to regions of increased stickiness. The word "haptic" means "relating to touch," and in this case it means the sense of touch of a cell. A migrating cell makes and breaks contact with the extracellular matrix. The idea behind haptotaxis is that a migrating cell may preferentially move in the direction of higher adhesiveness because a stickier contact is likely to pull the cell in that direction. Haptotaxis has been demonstrated in the laboratory [Harris 73] and there is evidence that haptotaxis guides the migration of the pronephric duct cells in the salamander embryo [Poole and Steinberg 82].

2.4.3 Electric Fields

Another possible source of information in an embryo is electric fields. Electric fields have been detected in many living organisms, including early chick embryos [Jaffe and Stern 79]

and in the regenerating limbs of some amphibians [Borgens 82]. Experiments are inconclusive about whether these fields play a role in development. Measurement of an electric field at a particular location gives information about both field strength and direction, so there is certainly the potential for such fields to be a source of geometric information.

2.4.4 Structural Information and Mechanical Forces

As we have discussed, migrating cells may be directed by chemical or adhesive gradients. Another guiding factor for migrating cells is the structure of the extracellular matrix in which they are moving. Some portions of the extracellular matrix contain fibers that are stretched preferentially in one direction. It has been shown in laboratory conditions that migrating cells are influenced in their motion by the way the structures in the matrix are aligned [Weiss 34]. When cell motion or shape is influenced in this manner it is called *contact guidance*. Because contact guidance has been demonstrated in the laboratory, it is tempting to guess that this is also a factor for cells migrating in the developing embryo.

Everything within an embryo is made either directly or indirectly by cells. This means that the preferential stretching of fibers within the extracellular matrix must result from the activities of cells (excepting forces from outside the embryo). Indeed, fibers taken from the extracellular matrix have been shown to be preferentially stretched in laboratory experiments. In particular, fibroblast cells in a petri dish will gather together into clumps when placed on a collagen substrate [Stopak and Harris 82]. The cells alter the collagen so that fiber bundles are stretched between these clusters of cells. Moreover, film of this organizing activity shows that cells not already in such a cluster seem to move along these stretched fibers as if the fibers are providing contact guidance. As with any study of cells outside the living organism, we should be cautious when drawing conclusions about what goes on in the developing animal. Nevertheless, these studies are suggestive, and we will return to mechanical forces when we talk about broad mechanisms for pattern formation.

2.4.5 Contact Inhibition

There is another kind of response a migrating cell may have to certain forms of contact. Often when a migrating cell comes to touch another cell it will back away from the point of contact. This is known as *contact inhibition.* It is easy to see how contact inhibition can help to disperse a clump of cells or to more evenly distribute a given type of cell through a region of the embryo. There is evidence to show that contact inhibition plays a role in the initial dispersion of cells from the neural crest [Rosavio et al 83].

2.4.6 Internal State of Cell

In the earlier section on determination we discussed how a cell may change its internal state to set the course of its development. Thus we can think of a cell's internal state as another source of information for the cell. A large part of a cell's state is determined by which genes are actively being transcribed from the cell's DNA and translated into proteins. The

transcription of DNA is partly controlled by the binding of *transcription factors* to particular locations along the DNA.

There are other sources of information internal to cells. Some of the actions that a cell performs are guided by timing mechanisms within the cell. An important example of this is the timing of cell divisions. This is a complex topic, and it is only recently that much of the timing mechanism for cell division has become understood [Murray and Kirschner 89].

Still another source of information is the protein and mRNA already present in the egg during the time of fertilization. The genes coding for such molecules are termed *maternal-effect* genes because the gene products are transmitted solely from the female zygote; the male zygote does not contribute to the proteins and mRNA of the fertilized egg. An example of a maternal-effect gene is demonstrated by a maternal-effect mutation called *snake* in the fruit fly. The wild-type mRNA product of this gene (no mutation) is present in the fertilized fruit fly egg. The protein coded by this gene has been shown to be important in determining the dorsal/ventral orientation of the embryo (backside versus belly). An embryo with the *snake* mutation on both chromosomes lacks this protein coded by the maternal-effect gene, and such embryos are badly deformed [Anderson and Nusslein-Volhard 84]. This illustrates that part of the geometry of the embryo can be dictated by information purely internal to the early cells of the organism.

2.5 Pattern Formation

One of the largest differences between higher animals and plants is in their strictness of body plan. Although each plant species has a distinct style or motif in the way its leaves, branches, flowers and so on are laid out, there is considerable variation in geometric detail between individuals from the same species. We can recognize a weeping willow by a collection of characteristics, like the typical bend of its branches, but two willows never have the same number of branches in an identical configuration. This variation of geometric detail in plants stands in strong contrast to the fixed body plan of animals. A given normal individual animal always has the same geometric form as other members of the same species. It is this constancy of geometry that allows us to say that a spider has eight legs and that the human femur is the same shape across all individuals. The process of realizing the blueprints for a developing animal is called *pattern formation*. To be more precise, pattern formation is the process by which differentiated tissue is laid out within an embryo according to a fixed geometric plan.

Pattern formation is a higher-level view of development than the cellular viewpoint that we adopted earlier. In examining how patterns form, we are interested in observing the fate of large collections of cells. Nevertheless, we can take a reductionist attitude towards pattern formation and ask how the actions of many individual cells ultimately determines the geometry within the animal. The sections that follow examine four mechanisms for pattern formation: the gradient model, reaction-diffusion, chemotaxis, and mechanical change. Each of these mechanisms can be examined from the standpoint of information available to the cells and the actions that these cells perform.

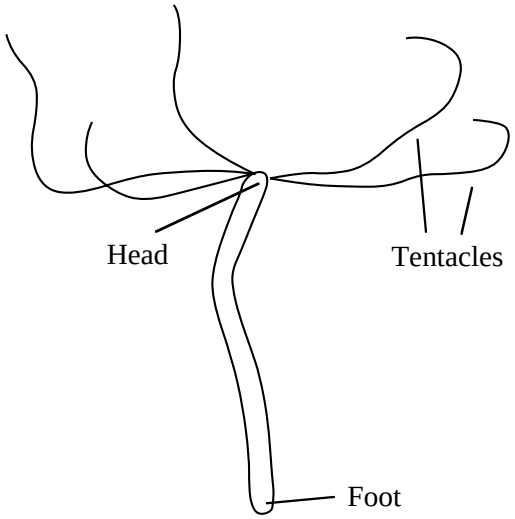

Figure 2.4: *Hydra*.

2.5.1 Gradient Model

In an earlier section we discussed cell differentiation based on the quantity of a chemical called a morphogen. Because one of the characteristics of a morphogen is that it can freely diffuse through an embryo, the best candidates for morphogens are small molecules or ions. The central issue for pattern formation based on a morphogen is the creation of a fixed, stable pattern of concentration throughout the embryo. There is more than one way in which the concentration of a morphogen can be caused to vary across different positions in the embryo. One way is for the chemical to be produced at one location in the embryo (the *source*) and for that chemical to be broken down at another location (the *sink*). Diffusion causes the morphogen concentration to vary from regions of high concentration (near the source) to regions of lower concentration (near the sink). This gives a stable variation of concentration of the morphogen if a balance is struck between the creation and breakdown of the morphogen. This basic model is known as the *gradient model* for pattern formation. This model has recently received a good deal of attention, largely due to the work of Lewis Wolpert [Wolpert 71], although the notion of chemical gradients in development has been around for quite some time [Child 41].

There is indication that a gradient mechanism is responsible for maintaining the body plan of the *Hydra*. The *Hydra* is a small, freshwater animal found in ponds and streams, and it is related to jellyfish and anemones. A *Hydra* has a collection of stinging tentacles surrounding its mouth at one end called its "head" and it attaches itself to plants or rocks at its "foot" (see Figure 2.4). The *Hydra* is rather unusual because its cells are constantly moving from one part of the organism to another, yet its basic body plan remains intact. There is evidence to suggest that there are several chemicals in the *Hydra* that are responsible for maintaining the position of the head and foot. One chemical, dubbed the *head activator*, has a high concentration in the head region, and this concentration tapers off towards the foot. The head activator appears to be a signal for maintaining the structures in the *Hydra's* head. The head

region is the source of the head activator. There is another chemical, called the *head inhibitor*, that is also produced by the head region. The head inhibitor seems to diffuse more rapidly than the head activator, and this inhibitor is thought to prevent other head regions from being formed elsewhere on the *Hydra*. In addition, there are two chemicals that seem to be responsible for maintaining the foot in much the same fashion. These four candidate morphogens pose a somewhat more complicated picture than the simple one-chemical gradient model of pattern formation. Still, there is much overlap here with the basic gradient model. The key similarity is the simple gradient from one part of the organism to the other that signals to the cells the role they should play in the body. (See [Gilbert 88] for a more complete description of morphogens in *Hydra* and for references.)

Another candidate for a morphogenic gradient is found in the study of limb-bud formation in chick embryos. The limb-bud is a bump on the chick embryo that later becomes a wing. A small molecule known as retinoic acid (a form of vitamin A) is present in graded concentrations across the limb-bud. Retinoic acid has recently has been discovered to affect development in some animals and it is a very strong producer of birth defects of the limbs and heart. New discoveries about its role in limb development suggest that it is a morphogen and that it directs the anterior/posterior (head/tail) orientation of limbs. It remains to be seen if the mechanism for maintaining retinoic acid's concentration across the limb bud is in agreement with the source/sink gradient model of pattern formation.

2.5.2 Reaction-Diffusion

The gradient model of pattern formation uses a source and a sink to create a stable chemical concentration across part of an embryo. Are there other ways a diffusible chemical can hold a fixed pattern of concentration in a developing organism? The mathematician Alan Turing answered this question in the affirmative in a landmark theoretical paper on morphogenesis [Turing 52]. Turing showed that two chemicals that diffuse at different rates and that react with one another can form stable patterns of concentration. Such a chemical system is called a *reaction-diffusion* system. In his paper, Turing used as an example a ring of cells such as is found at the mouth of a *Hydra*. He showed that a reaction-diffusion system acting over such a ring of cells could form a stable pattern of peaks and valleys of chemical concentration. This is much like the standing waves on a string that is vibrating at a fixed frequency. Figure 2.5 shows a graph of the concentration of one chemical from a simulation of such a system. More recently, other biologists have shown how reaction-diffusion systems that act in two dimensions can produce patterns of spots or stripes. See Sections 3.1 and 3.2 for a discussion of this work.

Both the "reaction" and the "diffusion" components of a reaction-diffusion system are needed to produce patterns such as spots and stripes. "Reaction" refers to how the concentrations of the two chemicals affect the creation or degradation of one another in a systematic manner. Turing assumed that these reactions operate identically in all regions of the embryo. Thus, if the concentration of the chemicals are initially uniform throughout the embryo then the amounts of the two chemicals will rise and fall in exactly the same way through the entire embryo, giving no spatial variation in concentration. Likewise, if there is

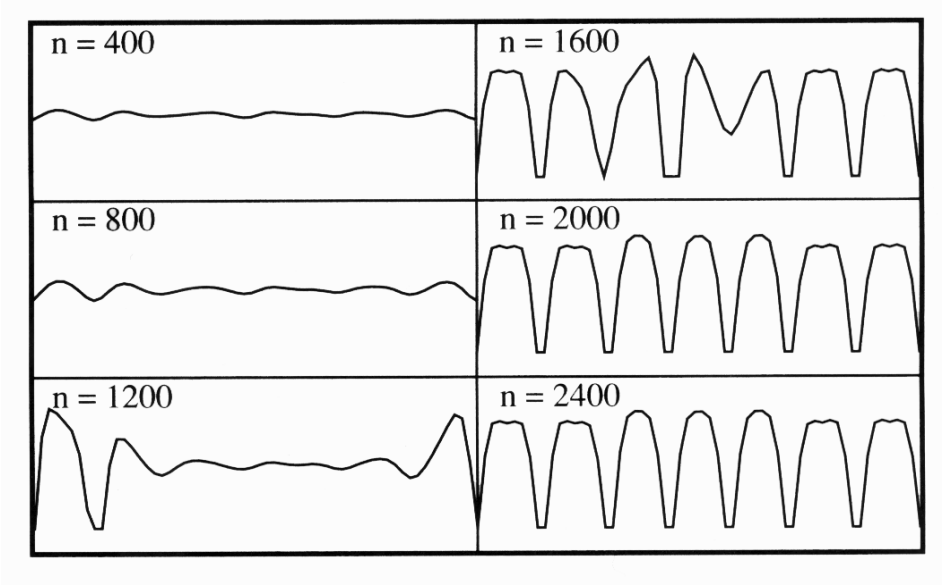

Figure 2.5: Pattern of chemical concentration from reaction-diffusion system.

no variation in concentration, then diffusion does not change the concentration at any point in the embryo. What Turing demonstrated, however, is that in many cases even a small variation in the initial concentration of one or both chemicals is enough to cause the two chemicals to reach a stable state where their concentrations vary in a regular manner across the embryo. This model of pattern formation is the basis of the textures created in this dissertation. We will examine reaction-diffusion systems in greater detail in Chapter 3.

2.5.3 Patterns Created by Chemotaxis

There is still another way in which chemical agents can play a role in pattern formation. Murray and Myerscough have presented a model of snake skin patterns based on chemotaxis [Murray and Myerscough 91]. They propose that the patterns on snakes result from different densities of cells over the skin and that the amount or kind of pigment produced is a factor of this cell density. Their model supposes the existence both of a chemical agent that attracts migrating cells and of a relationship between production of this chemical and the density of cells in a given region of skin. This system can be described as a pair of partial differential equations that share many characteristics with reaction-diffusion equations. Numerical simulations of this system with different parameters over several different geometries result in a wide variety of patterns. Many of these patterns match skin patterns of various snake species.

2.5.4 Mechanical Formation of Patterns

We discussed earlier how mechanical forces can act to clump together cells in a petri dish. In experiments the fibers underneath such groups of cells are stretched parallel to the lines running between neighboring groups of cells [Harris et al 84]. This clumping and the

stretching between the groups is reminiscent of some patterns found in developing organisms. This has led to the suggestion that mechanical forces can act as a mechanism of pattern formation [Oster, Murray and Harris 83]. This paper suggests that this mechanism may be responsible for the formation of feather primordia on chick embryos. These primordia are arranged in a diamond lattice on the surface of the chick, and there are lines of tension running between neighboring buds.

2.6 Summary

Development is the process of arranging cells in an embryo according to cell type. We have seen how this can be viewed from the perspective of cell actions and the various forms of information available to a cell. Patterns of tissues in an embryo are formed by many cells acting together in the embryo. Biologists have proposed several models of pattern formation, including the gradient model, reaction-diffusion, chemotaxis, and mechanical pattern formation. These mechanisms of pattern formation are actively being explored by developmental biologists. It is my hope that the field of computer graphics can benefit from further exploration of developmental mechanisms. At this point, however, we will leave behind other issues in developmental biology and concentrate only on reaction-diffusion.

3 Reaction-Diffusion Patterns

This chapter presents some of the patterns that can be formed by simulating reaction-diffusion systems. We begin by examining the components of a simple reaction-diffusion system and present a mathematical model of such systems. Next, we give an overview of the literature in developmental biology on pattern formation by reaction-diffusion. Once we have covered these basics, we will then demonstrate that more complex patterns can be formed using *cascades* of reaction-diffusion systems. This is new work that widens the range of patterns that can be created using reaction-diffusion. We will then look at an interactive program called *Cascade* that was made for exploring cascade systems. The methods presented in this chapter provide a user with a wide range of control over the textures that can be created using reaction-diffusion.

3.1 The Basics of Reaction-Diffusion Systems

To begin our discussion of reaction-diffusion systems we will describe a hypothetical arrangement of cells and give a description of the interactions between some of the chemicals in these cells. The assumptions are those made by Alan Turing in his paper that introduced reaction-diffusion as a possible pattern formation mechanism [Turing 52]. From these biological components we will form a mathematical description of what are considered the important aspects of a reaction-diffusion system.

Let us begin by considering a one-dimensional arrangement of cells in an embryo. Figure 3.1 gives a diagram of this hypothetical row of cells. We will give attention to two chemicals that are present in the cells, and we will refer to them as chemicals *a* and *b*. Assume that there are processes within these cells that can cause more of these chemicals to be manufactured

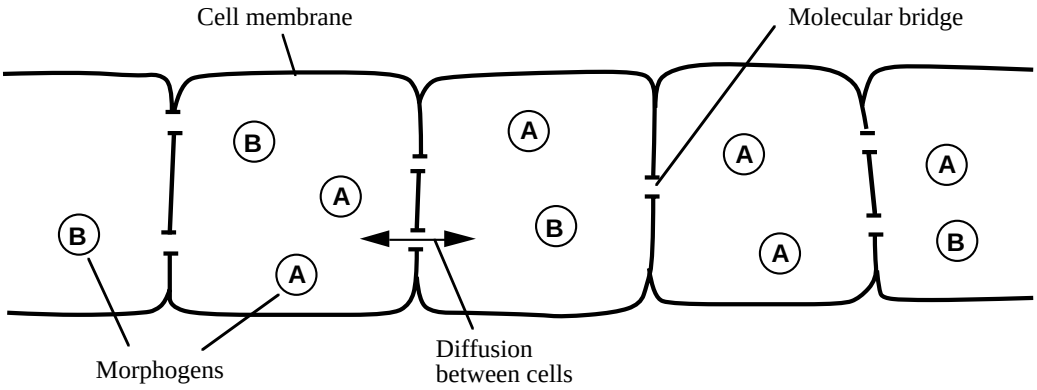

Figure 3.1: A row of cells. Molecular bridges allow morphogens to diffuse between neighboring cells.

and other processes that can cause these chemicals to be broken down (degraded). Further assume that the rates of these processes are dictated directly by the amounts of chemicals *a* and *b* already in a cell. That is, the amount of chemical *a* in a cell changes based on the quantity of the chemicals *a* and *b* are already in the cell. Likewise, the concentration of chemical b will increase, decrease or remain fixed depending on the concentrations of chemicals *a* and *b*. This is the *reaction* portion of the system.

The molecules within a cell are usually confined to the cell by the cellular membrane. In our model, however, we will assume that there are pathways or bridges between adjacent cells through which some molecules can pass. As an aside, such bridges between certain animal cells do in fact exist. *Gap junctions* are bridges between cells that are found more widely distributed in developing embryos than in adults, and they allow molecules of up to 1000 Daltons to pass between cells [Alberts et al 89]. Although Figure 3.1 shows only a few bridges, adjacent cells are often joined by many such molecular bridges. In our model we will assume that chemicals *a* and *b* can diffuse between neighboring cells in the row through these bridges. The process of diffusion will tend to even out the concentration of the chemicals over the row of cells. If a particular cell has a higher concentration of chemical *b* than its neighbors, then that cell's concentration of *b* will decrease over time by diffusion to its neighbors, if all else remains the same. That is, a chemical will diffuse away from peaks of concentration. Likewise, if the concentration of *b* is at minimum at a particular place along the row of cells, then more of *b* will diffuse from adjacent cells to this cell to raise the concentration of *b* at that cell. Diffusion acting alone over time tends to smooth out the concentration of chemicals across a given domain. It wears down the peaks and fills in the valleys.

Reaction and diffusion are the driving forces of change in the concentrations of chemicals *a* and *b* in the row of cells. Such a system may have one of several patterns of change in chemical concentrations over time depending on the details of the processes of reaction and diffusion. These possible trends of chemical concentration include: oscillating chemical quantities, unbounded increase of chemicals, vanishing of the chemicals, and a steady state of chemical concentrations. Turing singled out a special case of the last of these, achieving a steady state of concentrations, as the most interesting behavior for pattern formation. He found that under certain conditions a stable pattern of standing waves of concentration will arise. It happens that there are two necessary conditions for such a pattern to appear. The first of these is that the two chemicals must diffuse at unequal rates. That is, one of the chemicals must spread to neighboring cells more rapidly than the other chemical. The second condition necessary to make these patterns is that there must be some random variation in the initial concentration of chemicals over the cells. This is a reasonable assumption to make because there are bound to be small variations from one cell to another in a real organism. A specific reaction-diffusion system that follows these assumptions is given in Section 3.1.2.

3.1.1 A Mathematical Description of Reaction Diffusion

Now we will abstract from this model of cells and chemicals a mathematical description of the system. The result of this abstraction will be a set of partial differential equations. In our

mathematical model, the concentrations of these two chemicals can be represented as real numbers. The units of these quantities are not important, but the units might be something like parts per million or moles. (See [Lengyel and Epstein 91] for a description of the chemical reactants and their amounts in an actual reaction-diffusion system.) The positions along the row of cells can be either discrete (one value per cell) or they can be represented as positions along a continuous domain. For now, let us represent positions along the row as a continuous value. Let us now use the symbols a and b to represent the concentration of the corresponding chemicals. We can capture the changes in concentration of these two chemicals over time by the following partial differential equations:

$$\frac{\partial a}{\partial t} = F(a,b) + D_a \nabla^2 a$$

$$\frac{\partial b}{\partial t} = G(a,b) + D_b \nabla^2 b$$

The first of these equations describes the change over time in the amount of chemical a at a particular position in the line of cells. The reaction term $F(a,b)$ says that part of this change is some function of the local concentrations of a and b. The term $D_a \nabla^2 a$ says that the change in the concentration of a also depends on the diffusion of a from nearby positions. The constant D_a describes the speed of diffusion of chemical a. The Laplacian $\nabla^2 a$ is a measure of the concentration of a at one location with respect to the concentration of a nearby. If nearby places have a higher concentration of a, then $\nabla^2 a$ will be positive and a will diffuse towards this position. If nearby places have lower concentrations, then $\nabla^2 a$ will be negative and a will diffuse away from this position. The meaning of the second equation is analogous to the first, but with a different reaction function G and a different diffusion rate D_b. The

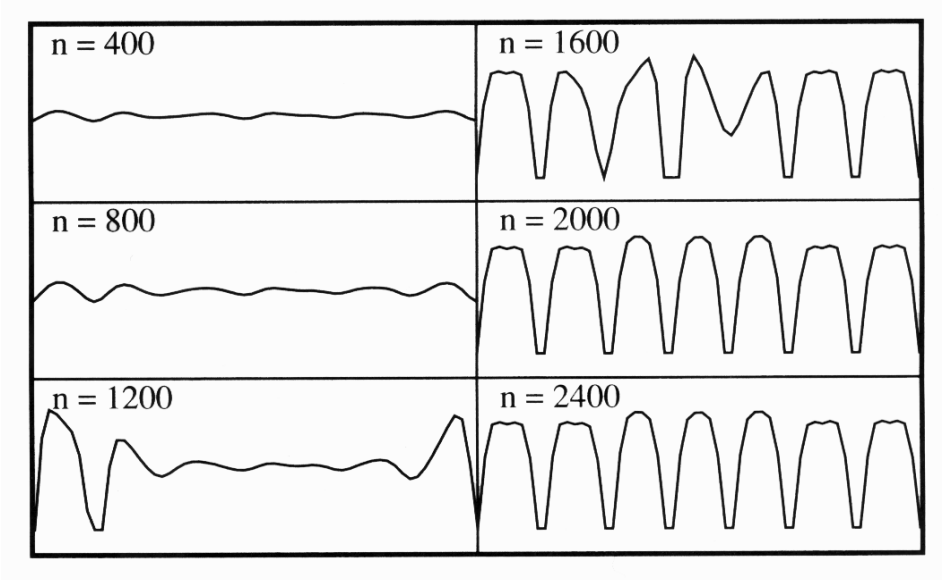

Figure 3.2: One-dimensional example of reaction-diffusion. Chemical concentration is shown in intervals of 400 time steps.

reaction terms in these equations say what is going on *within* a cell, whereas the diffusion terms describes the flow of chemicals *between* cells.

3.1.2 Simulation of Reaction-Diffusion

The partial differential equations above give a concise description of the reaction-diffusion process. It is natural to ask whether these equations can be solved in closed form. The answer is that closed-form solutions are difficult or impossible to find except when the reaction functions *F* and *G* are very simple. For this reason, reaction-diffusion equations are usually discretized and solved numerically instead of analytically. Here is the discrete form of one of the one-dimensional reaction-diffusion systems that Turing gave in his 1952 paper:

$$\Delta a_i = s\,(16 - b_i) + D_a\,(a_{i+1} + a_{i-1} - 2a_i)$$
$$\Delta b_i = s\,(a_i\, b_i - b_i - \beta_i) + D_b\,(b_{i+1} + b_{i-1} - 2b_i)$$

where

a_i = concentration of the first morphogen at the *i* th cell
b_i = concentration of the second morphogen at the *i* th cell
β_i = random substrate at *i* th cell
D_a = diffusion rate for *a*
D_b = diffusion rate for *b*
s = speed of reaction

For the equations above, each a_i is one "cell" in a row of cells and its neighbors are a_{i-1} and a_{i+1}. The diffusion term is approximated by the expression $D_a\,(a_{i+1} + a_{i-1} - 2a_i)$. The values for β_i are the sources of slight irregularities in chemical concentration across the row of cells. Figure 3.2 shows the results of these equations by graphing the change in the concentration of chemical *b* across a row of 60 cells over time. Initially the values of a_i and b_i were set to 4 for all cells along the row. The values of β_i were clustered around 12, with the values varying randomly by ±0.05. The diffusion constants were set to $D_a = 0.25$ and $D_b = 0.0625$, which means that *a* diffuses more rapidly than *b*. The reaction speed constant *s* has the value 0.03125. The values of *a* and *b* were constrained to be always greater than or equal to zero. Notice that after about 2000 iterations the concentration of *b* has settled down into a pattern of peaks and valleys. The simulation results look different in detail from this when a different random seed is used for β_i, but such simulations all have the same characteristic peaks and valleys with roughly the same scale of these features.

In the above reaction-diffusion system, β_i acts as a random substrate to keep the values of *a* and *b* from remaining constant across all cells. The initial values for a_i, b_i and β_i were set so that if there was no randomness introduced in the equations then the value of the reaction portions of the equations would always evaluate to zero. In other words, without the variation from β_i, the chemical concentrations of both *a* and *b* would remain flat and unchanging over time. As we will see below, the amount of randomness introduced by β_i affects just how regular or irregular the final pattern will be.

The above reaction-diffusion system can also be simulated over a two-dimensional domain. In this case the discrete equations become:

$$\Delta a_{i,j} = s\,(16 - a_{i,j}\,b_{i,j}) + D_a\,(a_{i+1,j} + a_{i-1,j} + a_{i,j+1} + a_{i,j-1} - 4a_{i,j})$$
$$\Delta b_{i,j} = s\,(a_{i,j}\,b_{i,j} - b_{i,j} - \beta_{i,j}) + D_b\,(b_{i+1,j} + b_{i-1,j} + b_{i,j+1} + b_{i,j-1} - 4b_{i,j})$$

These equations are simulated on a square grid of cells. Here the diffusion terms at a particular cell are based on its value and the values of the four cells that surround the cell. Each of the neighboring values for a chemical are given the same weight in computing the diffusion term because the length of the shared edge between any two cells is always the same on a square grid. This will not be the case when we perform a similar computation on an irregular mesh in Chapter 4, where different neighbors will be weighted differently when calculating $\nabla^2 a$ and $\nabla^2 b$.

Figure 3.3 (upper left) shows the result of a simulation of these equations on a 64×64 grid of cells. The values of the parameters used to make this pattern are the same as the values of the one-dimensional example (Figure 3.2) except that $s = 0.05$ and $\beta = 12 \pm 0.1$. Notice that the valleys of concentration in *b* take the form of spots in two dimensions. It is the nature of this system to have high concentrations for *a* in these spot regions where *b* is low. Sometimes chemical *a* is called an *inhibitor* because high values for *a* in a spot region prevent other spots from forming nearby. In two-chemical reaction-diffusion systems the inhibitor is always the chemical that diffuses more rapidly. The inhibitor communicates information about the distances to extremes of concentration (e.g. distances to the centers of spots) by this faster diffusion.

We can create spots of different sizes by changing the value of the constant *s* for this system. Small values for *s* ($s = 0.05$ in Figure 3.3, upper left) cause the reaction part of the system to proceed more slowly relative to the diffusion, and this creates larger spots. Larger values for *s* produce smaller spots ($s = 0.2$ in Figure 3.3, upper right). The spot patterns at the top of Figure 3.3 were generated with $\beta_{i,j} = 12 \pm 0.1$. If the random variation of $\beta_{i,j}$ is increased to 12 ± 3, the spots become more irregular in shape (Figure 3.4, upper left). The patterns that can be generated by this reaction-diffusion system were extensively studied in [Bard and Lauder 74] and [Bard 81].

The literature on reaction-diffusion systems is filled with different reaction functions that create either spot or stripe patterns. It is interesting to observe the many forms that a reaction function may take. These functions may be linear, may use product terms (as does Turing's), or may use quotients [Murray 81]. There is much work yet to be done to characterize what reaction functions will produce stable patterns in reaction-diffusion systems.

The method of introducing randomness into the above simulation may appear somewhat arbitrary. Albert Harris has pointed out that there are many ways to introduce randomness to a reaction-diffusion system and he has observed that many of these ways still produce the patterns that are characteristic of the particular system [Harris 92]. We have verified this for

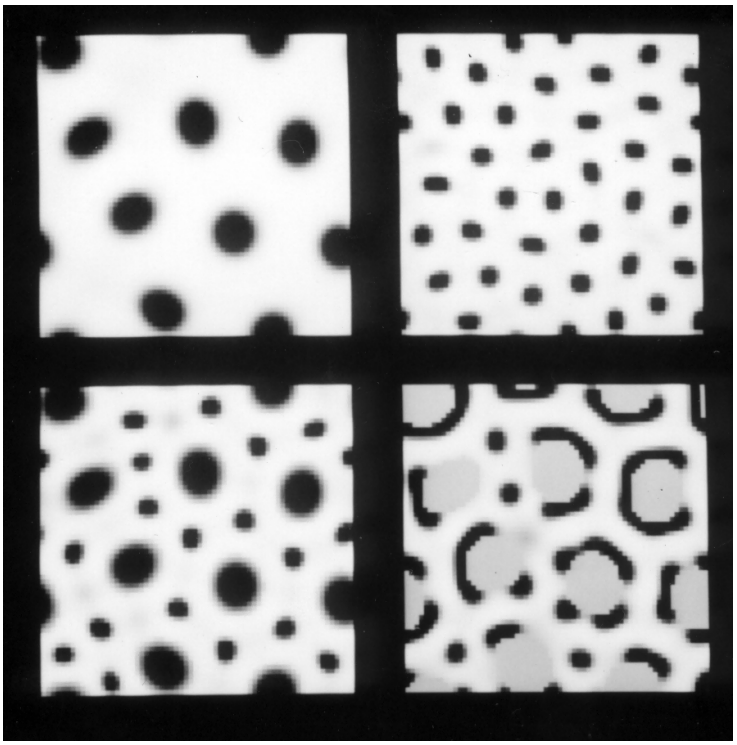

Figure 3.3: Reaction-diffusion on a square grid. Large spots, small spots, cheetah and leopard patterns.

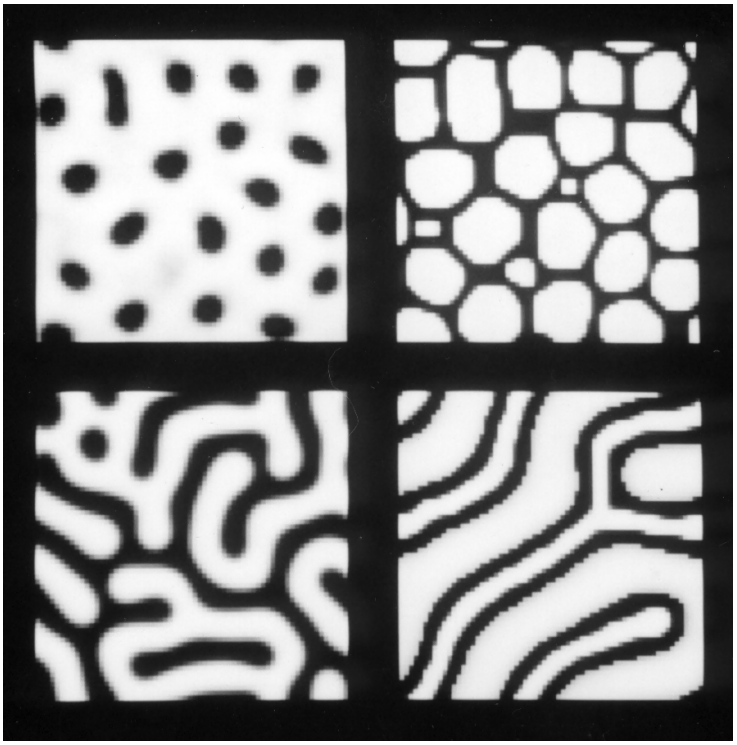

Figure 3.4: Irregular spots, reticulation, random stripes and mixed large-and-small stripes.

Turing's spot-formation system using the *Cascade* program (described below). For example, spots will form even if β is held constant at the value 12 (without random variation) and both the values for a and the diffusion rate D_a have slight random variations. The literature in reaction-diffusion does not appear to address this issue of how randomness is introduced into a system, and this is a fruitful topic for future research.

3.2 Reaction-Diffusion in Biology (Previous Work)

This section examines the literature on reaction-diffusion in developmental biology.

As was mentioned previously, reaction-diffusion was introduced as a model of morphogenesis by Alan Turing [Turing 52]. The paper introduced the idea that chemical substances called *morphogens* could react with one another and diffuse through an embryo to create stable patterns. Turing's paper is thorough in its examination of different aspects of this basic idea. It gives considerable detail about the breaking down and formation of chemicals due to reactions between different molecules in a reaction-diffusion system. The paper examines several simple reaction-diffusion systems. It describes a system with linear reaction terms and solves both the discrete and continuous forms of these equations. The solutions to both equations are a set of standing waves of chemical concentration (sinusoidal functions). The paper examines reaction-diffusion systems on a ring of cells and on a sphere. Turing suggests that the breakdown of symmetry of a reaction-diffusion system on a sphere may be responsible for triggering gastrulation in the early embryo. The paper also shows a dappling pattern made by reaction-diffusion that is reminiscent of the spots on cows.

Jonathan Bard and Ian Lauder thoroughly examined one of Turing's reaction-diffusion equations that acts over a row of cells [Bard and Lauder 74]. This is the same two-morphogen system that was described earlier in this chapter. Through computer simulation, they explored the kinds of patterns that this particular system will produce under a variety of initial conditions. Their conclusion was that the patterns that are generated by this system are not sufficiently regular to explain most patterns in development. They suggest, however, that reaction-diffusion could explain less regular patterns in development such as leaf organization or the distribution of hair follicles.

In 1981 both Jonathan Bard and James Murray published independent papers suggesting that reaction-diffusion mechanisms could explain the patterns on coats of mammals. Bard's work concentrated on showing that a variety of spot and stripe patterns could result from such systems [Bard 81]. He showed that a reaction-diffusion system can produce the small white spots on a deer or the large dark spots on a giraffe. He demonstrated that a wave of activation that sweeps over a surface of cells might account for creating striped patterns. These stripes can be either parallel or perpendicular to the wave of activation depending on the assumptions of the model. Murray's work showed that a single reaction-diffusion system might produce rather different patterns of light and dark on a mammalian coat depending on the size of the animal [Murray 81]. For example, the pattern created on a small animal may be dark regions at the head and tail of the animal separated by a white region in the center. The same mechanism may produce several irregularly shaped dark regions on a larger animal. On an

even larger animal, the mechanism might produce a large number of small dark spots. This same paper also demonstrated that a reaction-diffusion may account for some patterns found on butterfly wings.

Hans Meinhardt dedicated much of a book on pattern formation to exploring the patterns that reaction-diffusion systems can create [Meinhardt 82]. The book gives equations and FORTRAN code that produce spots and stripes in two dimensions. The lower left of Figure 3.4 shows a random stripe pattern created by a five-morphogen reaction-diffusion system that is described in this book. Appendix B of this dissertation gives the five simultaneous differential equations for stripe-formation. Also presented in Meinhardt's book is a reaction-diffusion system that creates patterns much like the veins on a leaf. Much of Meinhardt's book is concerned with simple chemical gradients that can be set up and remain stable under a number of different assumptions about the embryo's geometry and under different initial conditions.

David Young demonstrated that irregular striped patterns can be created by a reaction-diffusion model [Young 84]. These kinds of patterns strongly resemble the ocular dominance columns found in the mammalian visual system [Hubel and Wiesel 79]. Young's stripe-formation model is based on the work of N. V. Swindale, who simulated such patterns by a model of local activation and inhibition between synapses [Swindale 80].

Hans Meinhardt and Martin Klinger presented a model showing that reaction-diffusion systems may explain the patterns of pigment found on mollusc shells [Meinhardt and Klinger 87]. The model assumes that pigment is laid down on the growing edge of the shell. The shell's pattern reflects the history of chemical concentrations along the one-dimensional domain that is this growing edge. Different reaction functions and initial conditions result in different patterns. Many of the patterns generated by this model bear a striking resemblance to patterns found on actual molluscs. Deborah Fowler and her co-workers generated striking images of synthetic shells by bringing this work into the computer graphics domain [Fowler et al 92].

Perhaps a dozen or more researchers have attempted to model the segmentation of fruit fly (*Drosophila*) embryos using reaction-diffusion. Early work along these lines met with the problem of irregular stripe formation, where the reaction-diffusion systems proposed would not consistently form the seven segments that arise in *Drosophila* embryos [Kauffman et al. 78]. More recent models have met with success. Thurston Lacalli found that a particular model that uses four-morphogens will produce a stable pattern of stripes [Lacalli 90]. Axel Hunding, Stuart Kauffman and Brian Goodwin have showed that a hierarchy of reaction-diffusion systems can lay down a stable pattern of seven segments [Hunding et al. 90]. Their model is what we will call a *cascade* of reaction-diffusion systems, a topic we will return to in Section 3.3. Their paper also provides a good summary of other work on *Drosophila* segmentation and reaction-diffusion.

Reaction-diffusion is one developmental model within a broader class that have been termed local activation and lateral inhibition (LALI) models. George Oster has pointed out the

similarities between several models of pattern formation, including models based on chemical, neural and mechanical means of lateral inhibition [Oster 88]. Simulations of all of these models give a similar range of patterns, that of spots or stripes of varying sizes. For reaction-diffusion systems, the *local activation* says that a high concentration of one chemical (call it *x*) will cause even more of chemical *x* to be created. The *lateral inhibition* refers to the way production of another, more rapidly diffusing chemical (call it *y*) can be triggered by production of *x* and how in turn *y* can inhibit the production of *x*. The local activation causes peaks of concentration of *x*, and the lateral inhibition prevents this from being a runaway process. The lateral inhibition also keeps two peaks of *x* from forming too close to one another.

3.3 Cascade Systems

In earlier sections we saw that reaction-diffusion can be used to make simple spot and stripe patterns. This section presents a method of creating more complex patterns using more than one reaction-diffusion system. This expands the range of patterns that can be created for texture synthesis using reaction-diffusion, and it is one of the main contributions of this dissertation. Complex patterns can be created using reaction-diffusion by causing one chemical system to set down an initial pattern and then allowing this pattern to be refined by simulating a second system. We will refer to such a sequence of reaction-diffusion systems as a *cascade*. As mentioned earlier, one model of embryogenesis of the fruit fly is based on a series of reaction-diffusion systems. These systems may lay down increasingly refined stripes to give a final pattern that matches the segmentation pattern of the fly larva [Hunding 90]. Bard has suggested that such a cascade process might be responsible for some of the less regular coat patterns of some mammals [Bard 81]. The last paragraph in his article contains the following remarks:

> The jaguar has on its basically light coat, dark spots surrounded either by a dark ring or a circle of spots. Such a pattern could perhaps be derived from either a cascade process or a very complex interpretation system. It is however hard to see how these or any other mechanisms could generate the pattern of the thirteen-lined ground squirrel: this animal, basically brown, has seven A-P [anterior-posterior] stripes with six intervening rows of white spots...

Bard gives no details about the way two reaction-diffusion systems might interact. The patterns shown in this section are new results that are inspired by Bard's suggestion of a cascade process.

The upper portion of Figure 3.3 demonstrates that the spot size of a pattern can be altered by changing the size parameter *s* of Turing's reaction-diffusion system from 0.05 to 0.2. The lower left portion of Figure 3.3 demonstrates that these two systems can be combined to create the large-and-small spot pattern found on cheetahs. We can make this pattern by running the large spot simulation, "freezing" part of this pattern, and then running the small spot simulation in the unfrozen portion of the computation mesh. Specifically, once the large spots are made (using $s = 0.05$) we set a boolean flag *frozen* to TRUE for each cell that has

a concentration for chemical *b* between 0 and 4. These marked cells are precisely those that form the dark spots in the upper left of Figure 3.3. Then we run the spot forming mechanism again using $s = 0.2$ to form the smaller spots. During this second phase all of the cells marked as frozen retain their old values for the concentrations of *a* and *b*. These marked cells must still participate in the calculation of the values of the Laplacian for *a* and *b* for neighboring cells. This allows the inhibitory nature of chemical *a* to prevent the smaller spots from forming too near the larger spots. This final image is more natural than the image we would get if we simply superimposed the top two images of Figure 3.3. For the patterns made in this dissertation, the choice of when to freeze a pattern was made interactively by a user of the simulation program (described below).

We can create the leopard spot pattern of Figure 3.3 (lower right) in much the same way as we created the cheetah spots. We lay down the overall plan for this pattern by creating the large spots as in the upper left of Figure 3.3 ($s = 0.05$). Now, in addition to marking as frozen those cells that form the large spots, we also change the values of chemicals *a* and *b* to be 4 at these marked cells. When we run the second system to form smaller spots ($s = 0.2$) the small spots tend to form in the areas adjacent to the large spots. The smaller spots can form near the large spots because the inhibitor *a* is not high at the marked cells. This texture can also be seen on the horse model in Figure 5.12.

In a manner analogous to the large-and-small spots of Figure 3.3 (lower left) we can create a pattern with small stripes running between larger stripes. The stripe pattern of Figure 3.4 (lower right) is such a pattern and is modeled after the stripes found on fish such as the lionfish. We can make the large stripes that set the overall structure of the pattern by running Meinhardt's stripe-formation system with diffusion rates of $D_g = 0.1$ and $D_s = 0.06$ (see Appendix B for equations). Then we mark those cells in the white stripe regions as frozen and run a second stripe-forming system with $D_g = 0.008$ and $D_s = 0.06$. The slower diffusion of chemicals g_1 and g_2 (a smaller value for D_g) causes thinner stripes to form between the larger stripes.

We can use both the spot and stripe formation systems together to form the web-like pattern called reticulation that is found on giraffes. Figure 3.4 (upper right) shows the result of first creating slightly irregular spots as in Figure 3.4 (upper left) and then using the stripe-formation system to make stripes between the spots. Once again we mark as frozen those cells that compose the spots. We also set the concentrations of the five chemicals at the frozen cells to the values found in the white regions of the patterns made by the stripe-formation system. This causes black stripes to form between the marked cells when the stripe-formation system is run as the second step in the cascade process.

3.4 An Interactive Program for Designing Complex Patterns

The patterns presented in the section 3.3 indicate that there is a wide range of possible textures that can be made using cascades of reaction-diffusion systems. This section presents an interactive program called *Cascade* that is used for exploring the space of cascade patterns. Perhaps the most difficult aspect of creating new cascade patterns is that there are many ways

that one reaction-diffusion system can pass information to a second system. It may be desirable to have the first system change any of the following parameters of the second system: the diffusion rates, the initial chemical concentrations, the degree of randomness, the reaction rates, or additional parameters in the reaction functions. *Cascade* lets a user specify changes to these in a simple fashion using a small language designed for this purpose. *Cascade* uses a MasPar MP-1 parallel computer to quickly generate new patterns, and these patterns are presented in gray-scale on the user's screen. As an indication of the Maspar's simulation speed, it requires less than a second to compute 1,000 iterations of Turing's system on a 64×64 grid. The same simulation requires about 80 seconds on a DECstation 5000. The remainder of this section describes *Cascade* in more detail and presents patterns that have been created using the program.

3.4.1 Creating a New Pattern

Figure 3.5 shows the screen from a session with *Cascade*. In this particular session the user is creating a two-stage cascade texture to make cheetah spots. The left portion of the screen is dedicated to the first reaction-diffusion system and the right of the screen is for the second system in the cascade. Both are Turing's spot-formation system in this figure. Each half of the screen is further divided into a graphical portion at the top, where the chemical concentrations are shown, and a text portion below where parameters to the system are specified.

The conceptual model underlying *Cascade* is simple and consists of two parts. One part is that a user specifies the initial values of a collection of parameters across a grid. Each parameter is given a value at each position in the grid, so any parameter may vary spatially over the grid. This means that chemical concentration, reaction rates, diffusion rates, and all other parameters can differ depending on location. The second part of the conceptual model is that a user can invoke a routine that simulates a particular reaction-diffusion system over the grid. The simulation is carried out based on the initial values of the parameters that are specified by the user. A simulation routine changes the values of some of the parameters over time, namely those that represent the concentrations of chemicals in the reaction-diffusion system. Let us examine the way these two components are controlled through the user interface of *Cascade*.

A typical session with *Cascade* beings with the user choosing the basic systems that will be used in the cascade process. Currently there are three simulation routines to choose from: Turing's spot-formation system, Meinhardt's spot-formation system, and Meinhardt's stripe-formation system. Reasonable default parameter values are provided for each system, and these values appear in the text portion of the window. In Figure 3.5, for example, the initial value for chemical a at the left is given as 4, and the random substrate β is specified as 12 ± 0.1. The user is free to change any of these values and run a simulation at any time based on the specified parameters. Several words on the screen are surrounded by ovals, and these are called *buttons* (refer to Figure 3.5). A user can trigger an event by *clicking* on one of these buttons using a mouse. A simulation is begun by first clicking on the "Initialize" button and then clicking one or more times on the "Simulate" button. The "Initialize" button

Figure 3.5: *Cascade* user interface.

tells Cascade to evaluate the parameter specifications over the square simulation grid. Clicking "Simulate" runs the particular reaction-diffusion simulation for a number of time steps specified by the *iteration* parameter. The graphical representations of the chemicals are updated after each simulation run.

Most often a user will change parameters and run simulations on the first system until the pattern looks as if it will be a good basis for the desired final pattern. For example, in Figure 3.5, the user has created a large spot pattern that needs smaller spots added to it. The user can change the second system so that the initial values of one or more of its parameters are based on values from the first system. In Figure 3.5 this is accomplished by specifying a value of "b < 4" for the *freeze* parameter. This says that a cell in the simulation grid is to be frozen if the concentration of chemical *b* is less than 4 in the first system. To make the new spots smaller, the value for *ka* (another name for *s*) in the second system is different than its value in the first system. We will now take a closer look at parameter specification in *Cascade*.

3.4.2 Patterns Created by *Cascade*

Figure 3.6 shows twelve patterns that were created using *Cascade.* These patterns are not meant to be an exhaustive catalog of those that can be created. They are the result of a couple of days of exploration using *Cascade*, and they are presented as an indication of the range of patterns that can be created using cascaded systems.

The mechanisms for generating these patterns are outlined below. Complete expressions for each of these systems are given in Appendix C.

Autocatalytic Spots

These are spots created by Hans Meinhardt's autocatalytic spot-formation system. Notice that they are more irregularly spaced than the spots from Turing's system.

Vary Spots

This shows a gradation of spot sizes over a region. This was made by varying the speed constant *s* from Meinhardt's spot system.

Eroded Stripes

This pattern was created by cascading two stripe-formation systems. The first system created thick stripes, and the second system carved cavities in these thick stripes.

Thin Lines

The basic form of this pattern was laid down by a stripe-forming system. This pattern was then altered by Turing's spot-forming system, where the substrate parameter β was altered by the first system.

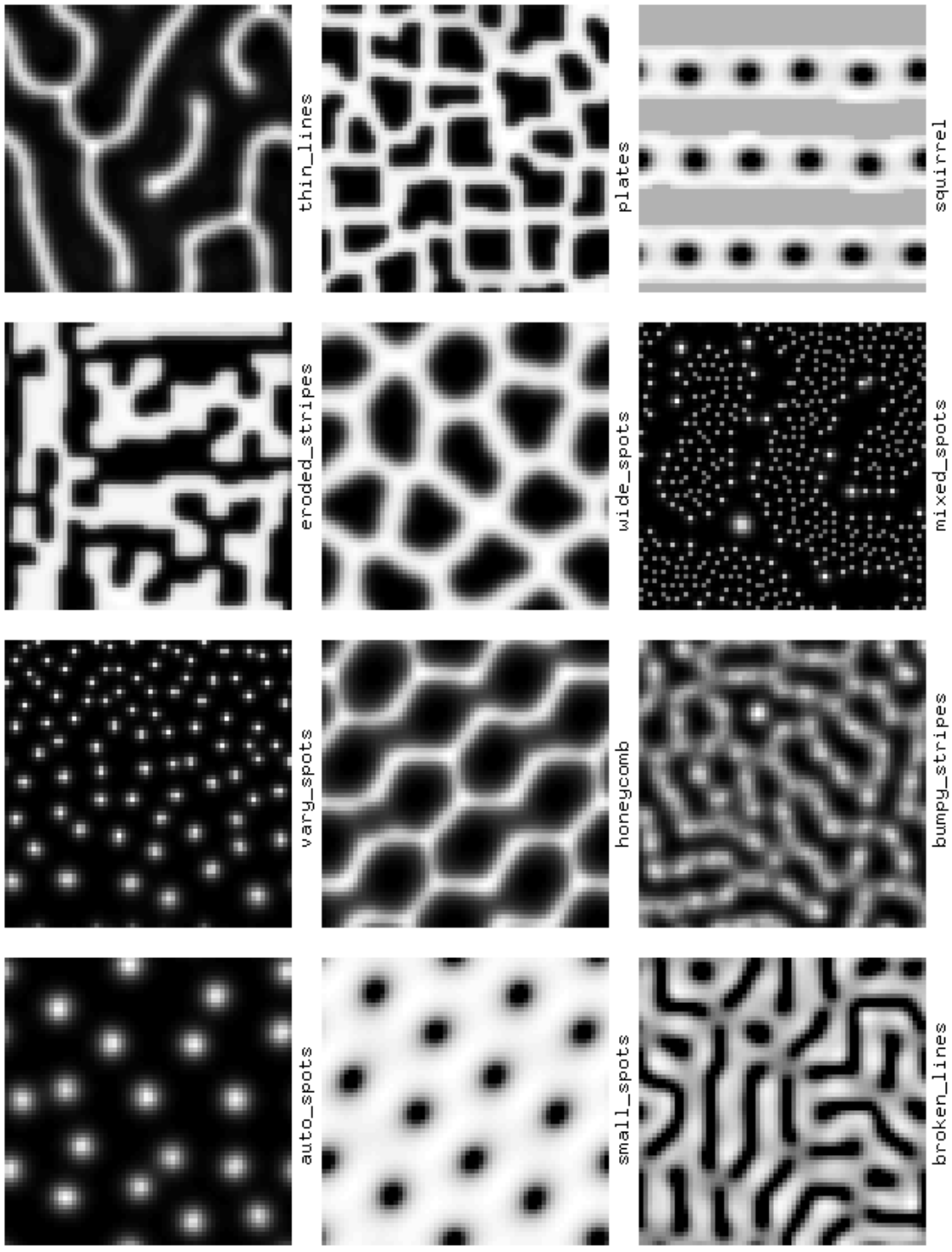

Figure 3.6: A variety of patterns created with *Cascade.*

Small Spots

This was created by cascading two Turing spot-formation systems. The second system's substrate β was derived from the chemical *a* of the first system.

Honeycomb

These two patterns are more extreme cases of the above cascade (Small Spots). Here, the initial value of β was higher than in the previous system.

Wide Spots

These spots were created by cascading large spots from Turing's system together with a stripe-formation system. The chemical *b* in the stripe system was altered based on chemical *b* of the spot system.

Plates

This is a similar system to the previous one (Wide Spots). The stripe-forming system was set to produce more thin stripes than the earlier pattern.

Broken Lines

This is a cascade of two Turing spot-formation systems. The second system's diffusion rate for chemical *a* was controlled by the concentration of chemical *b* from the first system.

Bumpy Stripes

This pattern was created from one instance of Meinhardt's spot-formation system. Here, the parameter *p*2 has been raised to cause the system to produce stripes instead of spots. This demonstrates that *Cascade* can be useful for exploring isolated reaction-diffusion systems.

Mixed Spots

This pattern was made by cascading thick stripes from Meinhardt's stripe-formation system together with spots from the autocatalysis spot system. The spot sizes were controlled by setting the speed parameter *s* from the result of the earlier stripe-formation system.

Squirrel

This pattern shows that the mixed stripe-and-spot pattern of the thirteen-lined ground squirrel can be created using a cascaded system. The stripes were set down using Meinhardt's stripe-formation system. The stripes were oriented vertically by having a higher degree of randomness at the right border. Then portions of the stripes were frozen and spots were created between the stripes using Turing's system.

4 Simulating Reaction-Diffusion on Polygonal Surfaces

This chapter describes a new method for placing patterns that are generated by reaction-diffusion onto polygonal surfaces. In order to understand what is presented in this chapter, let us consider the way a texture placement task would be accomplished using more traditional methods. Let us assume we wish to place a pattern of spots on a polygonal model of a frog. The most obvious way to do this begins by simulating a spot-forming reaction-diffusion system on a rectangular patch that is divided into a grid of squares. The resulting patterns of chemical concentration could then be considered as colors of an image texture, and that texture could be mapped onto the frog surface by traditional means. The rectangular patch could be wrapped over the surface of the frog by assigning patch coordinates to vertices of the frog model. This method has the advantage of using well-understood methods of texture mapping and rendering. Unfortunately, it also retains the problems of stretched textures and noticeable seams when the texture is placed on any but the most simple models.

Fortunately, we can take advantage of the physical nature of reaction-diffusion to place the resulting patterns on a given surface. Specifically, we can simulate any reaction-diffusion system directly on a mesh that is fit to a given surface. Presumably this is similar to what happens in naturally occurring reaction-diffusion systems. That is, a pattern-formation mechanism that makes spots on a real frog during development acts over the entire geometry of the skin or pre-skin cells. The frog's spots are not unusually stretched and show no seams because such a pattern is formed on the frog's surface. Likewise, if we simulate a spot-making reaction-diffusion system directly on the surface of a polygonal model of a frog, there will be no stretching or seams in the pattern. The spot texture is tailor-made for the given surface. This method of fitting a texture to a given surface is a central contribution of this dissertation.

The majority of this chapter is dedicated to describing an automated method of creating a mesh over which a reaction-diffusion system can be simulated. This is work that was presented in [Turk 91]. Quite a different approach to simulating reaction-diffusion on surfaces was presented in [Witkin and Kass 91]. The next section gives an overview of the technique of Witkin and Kass. The remaining sections turn to the central mesh creation method.

4.1 Simulation on Joined Patches (Previous Work)

One approach to fitting a reaction-diffusion system to a given surface was described by Andrew Witkin and Michael Kass [Witkin and Kass 91]. It should be noted that this method has not yet been implemented for complex surfaces. Their method begins by dividing the surface to be textured into rectangular patches. This step currently must be done by hand since there are as yet no automated methods of taking a polygonal model and fitting rectangular patches over the model. The next step is to gather together information about the way the patches stretch over the model and where they join one another. This information is then used during simulation of a reaction-diffusion system. The simulation is performed on several rectangular grids of square cells, one grid for each of the rectangular patches. Let us examine the way in which simulation on grids can be modified to avoid the twin pitfalls of texture distortion and seams between patches.

For isotropic diffusion systems, where the chemicals diffuse at equal rates in all directions, simulation on a grid of square cells is straightforward. Several examples of isotropic diffusion systems were described in Chapter 3. The basic method replaces the differential equations with discrete versions of the equation that approximates the Laplacian operator with sums and differences of adjacent grid cells. For instance, consider Turing's spot-formation system as simulated on a grid:

$$\Delta a_{i,j} = s\,(16 - a_{i,j}\,b_{i,j}) + D_a\,(a_{i+1,j} + a_{i-1,j} + a_{i,j+1} + a_{i,j-1} - 4a_{i,j})$$
$$\Delta b_{i,j} = s\,(a_{i,j}\,b_{i,j} - b_{i,j} - \beta_{i,j}) + D_b\,(b_{i+1,j} + b_{i-1,j} + b_{i,j+1} + b_{i,j-1} - 4b_{i,j})$$

The last terms of the first equation show that the diffusion term at cell $a_{i,j}$ is approximated by the sum of the concentrations of a at the four neighboring cells minus four times the concentration at cell $a_{i,j}$. This is the isotropic case. Such grid simulations can be modified to take into account anisotropic diffusion. For instance, the diffusion term for $a_{i,j}$ could be changed to the following:

$$D_a\,(2a_{i+1,j} + 2a_{i-1,j} + a_{i,j+1} + a_{i,j-1} - 6a_{i,j})$$

This equation says that chemical a diffuses twice as rapidly in one direction as in the other direction. This has the effect of stretching the pattern in the direction of faster diffusion. Figure 4.1 shows an isotropic pattern on the left and a stretched pattern on the right. This ability to create patterns that appear to be stretched was suggested by Witkin and Kass as a way to generate textures that are pre-distorted to fit a given surface. For instance, suppose one of the rectangular patches that covers a model is stretched in the vertical direction near the top of the patch than at the bottom. This could be taken into account during the simulation by changing the diffusion term so that the pattern would be bunched up near the top. Then when the final pattern is stretched onto the surface, the top of the patch would be stretched to un-distort the pattern. The proper pre-distortion amount can be computed from the Jacobian of the parametric equation for the surface patches. This is described in full in [Witkin and Kass 91].

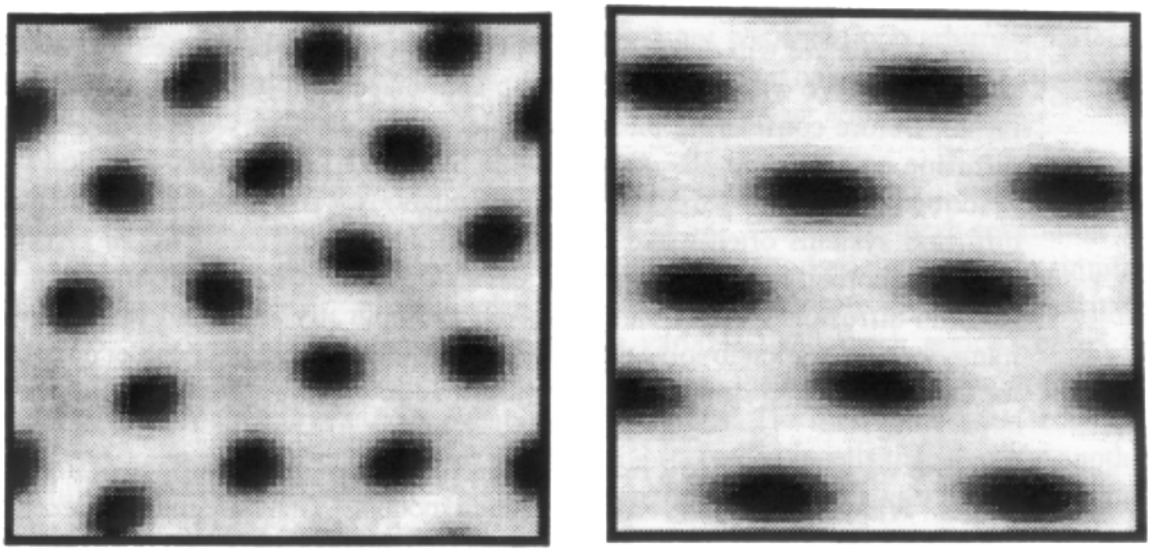

Figure 4.1: Isotropic and anisotropic patterns of spots.

Creating pre-distorted patterns avoids textures that look stretched, but what can be done about avoiding seams between patches? This issue is addressed by using appropriate boundary conditions for the patches, that is, by sharing information between patches during simulation of the physical system. Chemical quantities can be passed from one patch to an adjacent patch by incorporating cells from more than one patch into the sums of cell concentration that approximate the Laplacian terms. The most simple case is that of the periodic patch, where the top and bottom edges of a patch are identified with one another and likewise the left and right edges are joined to one another. These conditions are appropriate for a torus that is covered by a single four-boundary patch. The diffusion terms in the simulation equation are modified to take this into account simply by changing the meaning of "neighboring cell" for cells on the edge of the rectangle. Each of the cells on the left edge of a grid has the three "typical" neighbors and, in addition, has a fourth neighbor that is a cell from the far right of the grid along the same row of cells. All the simulations on square grids shown in Chapter 3 use such periodic boundary conditions. Patches that cover more complicated objects can also have their boundary conditions modified to join the patches together during the simulation that creates reaction-diffusion patterns. This is the approach suggested (but not yet implemented) by Witkin and Kass to avoid seams between patches. Each of the reaction-diffusion patterns shown in their paper were simulated on one surface patch.

The strength of the above method for simulating reaction-diffusion on surfaces is that many grid-based simulation techniques can be brought to bear. For instance, Witkin and Kass demonstrate that a rapidly computable approximation to Gaussian convolution [Burt 81] can be used to speed up reaction-diffusion simulations. The drawback of the patch-based method is that the user is required to divide the model into surface patches. The remainder of the chapter is devoted to a method of simulating reaction-diffusion directly on the surface of a model that requires no such work on the part of the user.

4.2 Requirements for Simulation Meshes

In this section we will discuss the desired characteristics of meshes for reaction-diffusion simulations. We will consider the acceptable shapes and sizes for the cells of a simulation mesh. Before considering these issues, let us ask whether we need to simulate reaction-diffusion on a mesh at all. As we saw in Section 3.1, a reaction-diffusion system can be described by a set of partial differential equations. The non-linearity of many reaction-diffusion systems often make closed-form solutions difficult. Solving such systems on complex geometry such as a giraffe model makes analytic solutions harder still. Because few reaction-diffusion systems can be solved symbolically in closed form, our system uses numerical integration to solve the differential equations. Let us consider possible sources for meshes on which to simulate a reaction-diffusion system.

Suppose we wish to place a reaction-diffusion pattern on the surface of a given polygonal model. What are the characteristics of such models? There are many sources of polygonal models in computer graphics. Models generated by special-effects houses are often digitized by hand from a scale model. Models created by computer-aided design might be made by converting a model from constructive solid geometry to a polygonal boundary representation. Some models are generated procedurally, such as fractals used to create mountain ranges and trees. Still other models are captured by laser scanning of an object. It is natural to ask whether we can use the original polygonal mesh as the mesh on which to simulate a reaction-diffusion system. Unfortunately, the above methods of model creation give few guarantees about the shapes of the polygons, the density of vertices across the surface or the range of sizes of the polygons. If a model has polygons that are larger than the size of the spots we want to place on the surface then we cannot capture the detail of the spots by using the original polygonal mesh for simulation. Long mesh cells are not good for simulations because the numerical approximations to the derivatives at such cells are poor. We do not want to simulate a system on a mesh that is too dense, otherwise we will spend more time to compute a texture than is necessary. For these reasons it is unwise to use the original polygons as the mesh to be used for creating textures. Let us examine the properties we want for our simulation meshes.

The simulations of Chapter 3 all took place in a grid of squares. The regularity of such a grid makes simulation programs particularly easy to write. All the squares are the same size and each of them has four nearest neighbors, given the appropriate boundary conditions. Is it important that the squares are the same size? Our goal in simulating reaction-diffusion systems is to produce patterns such as those described in Chapter 3. None of these patterns have features that vary radically in size. For instance, the spots in the upper right square of Figure 3.3 are roughly the same size throughout the square. The same observation can be made of striped patterns. We will refer to the average width of the spots or stripes in a pattern as the *natural feature size* of the pattern. We need grid cells that are small enough to capture the features of a given reaction-diffusion system, but these cells do not need to be very much smaller than the natural feature size of the system. Features will be lost if the cells aren't small enough, but extra work will go to waste if the cells are too small. It makes sense, then, to have grid cells that are all roughly the same size. Another requirement is for the shapes of the cells

to be fairly regular so that the chemicals will diffuse isotropically across the surface. Ideally we would like a mesh to be composed of cells that are all exactly the same shape, such as regular squares or hexagons. Unfortunately, this is not possible upon arbitrary surfaces. Instead, our goal will be to automatically generate a mesh that has cells that are all roughly the same size and shape.

Mesh generation is a common problem in finite-element analysis, and a wide variety of methods have been proposed to create meshes [Ho-Le 88]. Automatic mesh generation is a difficult problem in general but the requirements of texture synthesis will serve to simplify the problem. We don't need to invoke the elaborate methods from the finite-element literature. These complex methods can create variable-sized mesh regions based on geometric features or physical properties of the surface. Instead, we only require that the model be divided up into relatively evenly-spaced regions. The mesh generation technique described below automatically divides a polyhedral surface into cells that abut one another and fully tile the polygonal model. The only input necessary from the user is a specification of the number of cells to be in the final mesh.

There are three steps to generating a mesh for reaction-diffusion simulation. The first step is to distribute *n* points randomly over the surface of the given polygonal model. Step two causes all these points to repel one another, thus spreading themselves evenly over the surface of the model. The final step is to build a cell (called a Voronoi region) around each of these points. The following two sections (4.3 and 4.4) describes these steps in detail.

4.3 Even Distribution of Points over a Polygonal Surface

This section describes a new method of evenly spreading points over the surface of a polygonal model. These points will eventually become the center of cells in a simulation mesh. First we will describe a method of placing points randomly over a given model so that no part of the model is more likely to receive points than any other portion of the surface. Then these points will be spread more evenly over the model's surface by having each point repel all other nearby points.

4.3.1 Random Points Across a Polyhedron

Distributing points randomly over a polyhedral model is non-trivial. Care must be taken so that the probability of having a point deposited within a fixed-size region is the same for all such regions on the surface. When placing a point randomly on a model, we cannot simply choose randomly with equal weight from among all the model's polygons. This would result in a higher density of points on the smaller polygons. Instead, when randomly placing a point, we need to make an area-weighted choice from among the polygons. (We will show later a method of placing a point randomly on this polygon.) Suppose we have a model composed of *n* polygons: $P_1, P_2, \ldots P_n$. Let each polygon P_i in the model have an area A_i. We want the probability of placing a point on polygon P_i to be proportional to this polygon's area relative to the surface area A_{total} of the model. This probability is A_i / A_{total}. This can be accomplished first by creating a list of sums $S_1, S_2, \ldots S_n$, where S_i is the sum of all polygon areas from A_1 up

to and including A_i. To choose a random polygon, we will first pick a random value r chosen uniformly in range of zero to A_{total} (the total surface area of the model). Now we can look through the list of partial sums S_i to find the smallest i such that S_i is great than r, that is, find i such that $S_{i-1} \le r < S_i$. This search can be performed quickly using binary search over the values S_1, S_2,... S_n. Since r is chosen from zero to A_{total} and because $S_i - S_{i-1} = A_i$, this search will yield the value i with probability A_i / A_{total}, as we desired. This indicates that the point should be deposited on the polygon P_i. We now require a method of placing a point randomly on this polygon. The approach described below for this is taken from [Turk 90].

In what follows we will assume that we are picking a random point on a triangle. Any polyhedral model can be triangulated to satisfy this requirement. Call the vertices of the triangle A, B, and C. Let s and t be two uniformly distributed random values chosen from the interval [0,1]. Then the pseudo-code below picks a random point Q in the triangle:

```
if s + t > 1 then          { reflect if necessary }
    s = 1 - s
    t = 1 - t

a = 1 - s - t              { compute barycentric coordinates within the triangle }
b = s
c = t

Q = aA + bB + cC
```

Without the "if" statement, the point Q will be a random point in the parallelogram with vertices A, B, C, and $(B + C - A)$ (see Figure 4.2). A point that lands in the triangle B, C, $(B + C - A)$ is moved into the triangle A, B, C by reflecting it about the center of the parallelogram.

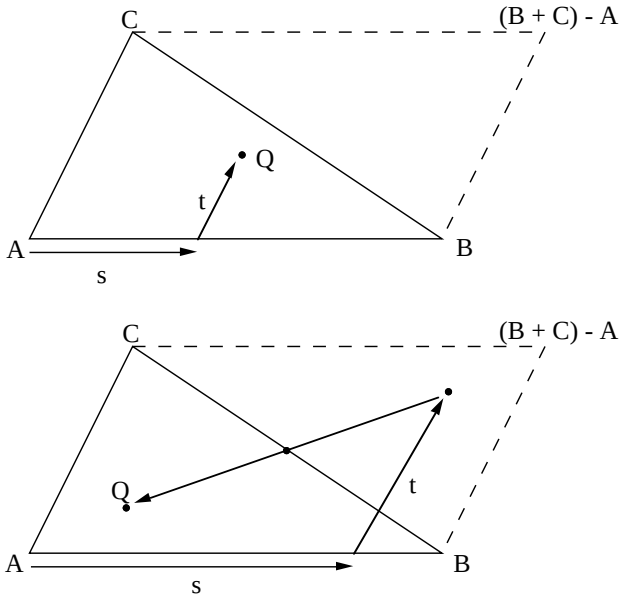

Figure 4.2: Picking random point in triangle. Top: $s + t \le 1$. Bottom: $s + t > 1$.

4.3.2 Point Relaxation

Once the proper number of points have been randomly placed across the surface, we need to move the points around until they are somewhat regularly spaced. This is accomplished using relaxation. Intuitively, the method has each point push around other points on the surface by repelling neighboring points. The method requires choosing a repulsive force and a repulsive radius for the points. It also requires a method for moving a point that is being pushed across the surface, especially if the point is pushed off its original polygon. Here is pseudo-code giving an outline of the relaxation process:

```
loop k times
    for each point P on surface
        determine nearby points to P
        map these nearby points onto plane containing the polygon of P
        compute and store the repulsive forces that the mapped points exert on P
    for each point P on surface
        compute new position of P based on repulsive forces
```

Each iteration moves the points into a more even distribution across the polyhedron. Figure 4.3 shows an initially random distribution of 200 points in a square and the positions of the same points with $k = 50$ iterations of the relaxation procedure.

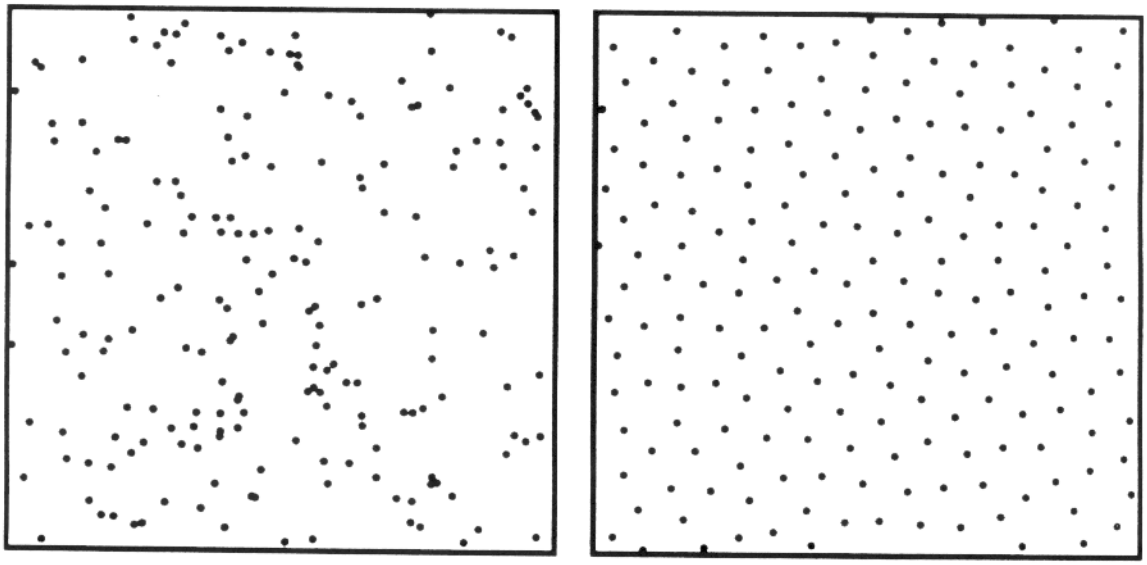

Figure 4.3: Random points in plane (left) and the same points after relaxation (right).

The repulsive radius of the points should be chosen based on the average density of the points across the whole surface. The meshes used in this paper were created using a radius of repulsion given by:

n = number of points on surface
a = area of surface
$r = 2\sqrt{a/n}$

The above value for r gives a fixed average number of neighboring points to any point, independent of the number of points on the surface and independent of surface geometry. This is important because uniform spatial subdivision can then be used to find neighboring points quickly. It has been our informal experience that the exact nature of the equation for the force between points does not significantly affect the quality of the final point distribution. For instance, we could have used a force that varies inversely with the square of the distance between points, but this would have meant that points arbitrarily far apart would affect one another. This would have been an added computational burden with no advantage in the final configuration of points.

For repulsion by nearby points, we require a distance function across the surface. For points that lie on the same polygon the Euclidean distance function can be used. For points that lie on different polygons we need to do something reasonable. A logical choice is to pick the shortest distance between the two points over all possible versions of the polyhedral model where each model has been unfolded and flattened onto a plane. Unfortunately, determining the unfolding that minimizes this distance for any two points is not easy. Since there is no limit to the number of polygons that may be between the two points, finding this distance could take an arbitrary amount of computation. As a compromise, we choose only to use this flattened distance when the two points are on adjacent polygons, and we will use an approximation if the points are further removed than this. We use this approximation for the sake of speed. The approximate distance between points that is described below can be computed in constant time for any pair of points. There are clearly more elegant approaches to finding distances over a surface that have a higher computational cost. Fortunately, we do not affect the final point distribution much by using an approximation because points that are far from one another do not affect each other's final position much.

The left portion of Figure 4.4 illustrates computing the flattened distance between two points that are on adjacent polygons. We can pre-compute and store a transformation matrix M for adjacent polygons A and B that specifies a rotation about the shared edge that will bring polygon B into the plane of polygon A. With this transformation, the distance between point P on polygon A and point Q on polygon B is determined by applying M to Q and then finding the distance between this new point and P.

For points P and Q on widely separated polygons A and B, we first find in which direction Q lies with respect to the polygon A. Then we can apply to Q the rotation given by the transformation matrix of the edge associated with this direction. Usually this will not bring Q into the plane of A, so this point is then projected onto the plane of A and we can use the

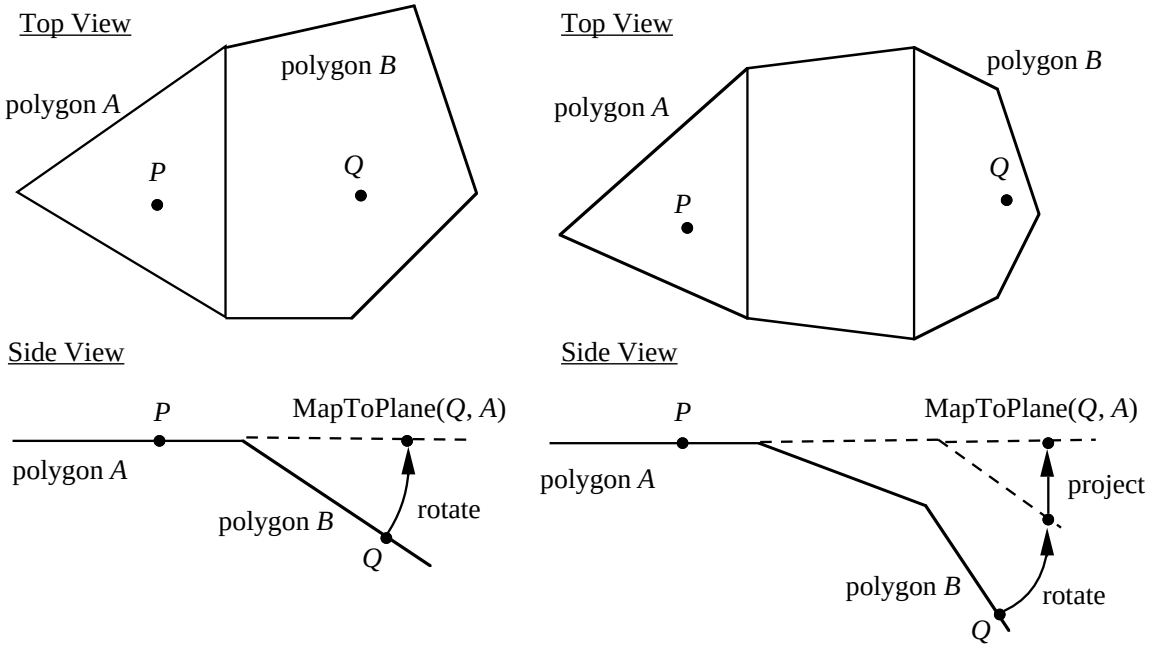

Figure 4.4: Mapping nearby point Q onto plane of point P. Left shows when Q is on adjacent polygon. Right shows more remote point Q.

distance between this new point and *P* as our final distance. This is shown in the right portion of Figure 4.4. This gives a distance function across the surface, and in addition it gives a method of making every point near a given point *P* seem as though it lies on the plane of the polygon of *P*. The procedure of moving a point *Q* onto polygon *A* is called MapToPlane(*Q*,*A*).

With a distance function in hand, making the points repel each other becomes straightforward. For each point *P* on the surface we need to determine a vector *S* that is the sum of all repelling forces from nearby points. Here is the determination of *S* based on the repulsive radius *r*:

```
S = 0
for each point Q near point P
    map Q onto plane of P's polygon; call the new point R
    V = normalized vector from R to P
    d = distance from R to P
    if d < r then
        S = S + (r - d) V
```

Once this is done for each point on the surface, the points need to be moved to their new positions. The new position for the point *P* on polygon *A* will be $P' = P + kS$, where k is some small scaling value. The meshes used here were made with $k = 0.15$. In many cases the new point *P*' will lie on *A*. If *P*' is not on *A* then it will often not even lie on the surface of the polyhedron. In this case, we determine which edge of *A* that *P*' was "pushed" across and also find which polygon, call it *B*, that shares this edge with *A*. The point *P*' can be rotated about

the common edge between *A* and *B* so that it lies in the plane of *B*. This new point may not lie on the polygon *B*, but we can repeat the procedure to move the point onto the plane of a polygon adjacent to *B*. Each step of this process brings the point nearer to a polygon, and eventually this process will terminate.

Most polygons of a model should have another polygon sharing each edge, but some polygons may have no neighbor across one or more edges. Surfaces that have edges such as these are called manifolds with boundaries. A cube with one face removed is an example of such a surface, and this surface has a four-sided boundary. If a point is "pushed" across such a boundary then the point should be moved back onto the nearest position still on the polygon. This is simply a matter of finding where the line between *P* and *P*' intersects the boundary of the polygon that *P* is on. For a surface with boundaries, this will cause some of the points to be distributed along the boundaries.

The result of the point repulsion process is a set of points that are fairly evenly distributed over the surface of the given polygonal model. These points can now serve as centers for cells of a simulation mesh. Once this mesh is built, any reaction-diffusion system can be simulated on the surface of the model. Let us turn to creating the cells surrounding the mesh points.

4.4 Generating Voronoi Cells on the Model

The final step in creating a mesh is to take the points that were distributed evenly by the relaxation process and determine the cells centered at each of these points. Each of these cells is like a container of small quantities of the chemicals that are participating in the reaction-diffusion process. These cells will abut one another, and the chemicals will diffuse between pairs of cells through their shared wall. Figure 4.5 gives an abstract picture of this model. The size of the shared wall between two cells determines the amount of diffusion between the cells. This quantity will be referred to as the *diffusion coefficient* between the two cells. We need a way to form regions around the points to determine adjacency of cells and to give the diffusion coefficients between adjacent cells. In keeping with many finite-element mesh-generation techniques, we choose to use the Voronoi regions of the points to form the regions surrounding the points. Other choices of mesh generation procedures are possible, and we will discuss this issue below.

A description of Voronoi regions can be found in books on computational geometry, e.g., [Preparata and Shamos 84]. Given a collection of points *S* in a plane, the *Voronoi region* of a particular point *P* is that region of the plane where *P* is the closest point of all the points in *S*. For points on a plane, the Voronoi regions will always be bounded by lines or portions of lines that are positioned halfway between pairs of points. Figure 4.6 shows the Voronoi regions in the plane for the points from Figure 4.3. When simulating a diffusing system on such a set of cells we will use the lengths of the edges separating pairs of cells to determine the quantity of a given chemical that can move between the two cells. It is important to understand that constructing the Voronoi region around a particular point tells us which points are its neighbors. We need to know which points are neighbors in order to compute the diffusion terms at a point.

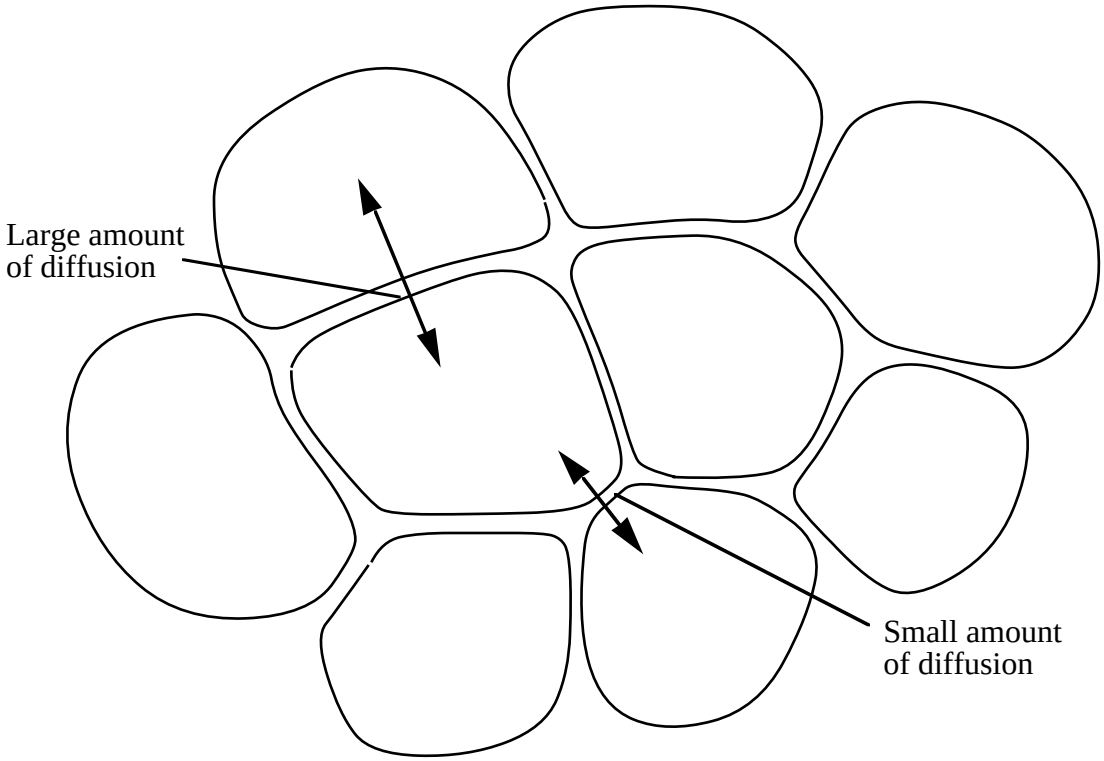

Figure 4.5: Amount of diffusion between adjacent cells depends on size of shared boundary.

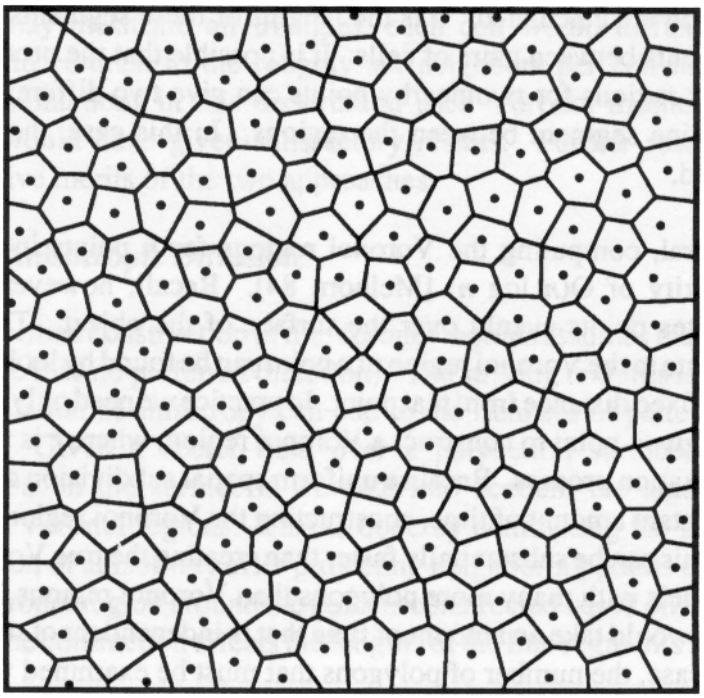

Figure 4.6: Voronoi regions of the points shown in Figure 4.3.

The concept of a Voronoi region gracefully generalizes to three dimensions. A Voronoi region of a point *P* from a set of points *S* in 3D is a region of 3-space that is separated from other regions by planes or portions of planes that are midway between *P* and other points in *S*. These three-dimensional cells are always convex. This leads directly to one possible generalization of Voronoi regions for a polyhedral surface. We can take the Voronoi region of a point *P* on the surface as being the intersection between the 3D Voronoi region and the polyhedral surface. Unfortunately, this sometimes leads to regions that are composed of several disjoint pieces on different portions of the surface. Moreover, these Voronoi regions do not adequately capture the notion of adjacency of points on a surface because the distance function does not measure distance across the actual surface.

A more reasonable generalization of Voronoi regions to a surface makes use of a measure of distance over the surface. We can use the notion of shortest path (geodesic) over a polyhedral surface to give a better definition of Voronoi region. Using the length of the shortest path over the surface as our distance measure, the Voronoi regions that this induces are never disjoint. They also capture the idea of adjacency better than the previous definition of Voronoi regions for a surface. An exact solution to finding these Voronoi regions on an arbitrary polyhedron is given in [Mount 85]. Unfortunately, the algorithm to determine exact Voronoi regions is rather involved. Instead of using Mount's method, the meshes in this dissertation were created using planar approximation of the exact Voronoi regions.

The simplified method of creating Voronoi regions on a polygonal surface makes use of the unfolding process used in the point repulsion process. Using the same procedure as before, all points near a given point *P* are mapped onto the plane of the polygon *A* containing *P*. Then the planar Voronoi region of *P* is constructed and the lengths of the line segments that form the region are calculated. It is the lengths of these segments that are used as the diffusion coefficients between pairs of cells. It is possible that the process of unfolding and creating Voronoi regions for two nearby points can give two different values for the length of the shared line segment between the regions. In this case, the values of the two lengths is averaged.

In general, computing the Voronoi regions for n points in a plane has a computational complexity of $O(n \log n)$ [Melhorn 84]. Recall, however, that the relaxation process distributes points evenly over the surface of the object. This means that all points that contribute to the Voronoi region of a point can be found by looking only at those points within a small fixed distance from that point. In practice we need only to consider those points within $2r$ of a given point to construct a Voronoi region, where r is the radius of repulsion used in the relaxation process. Because uniform spatial subdivision can be used to find these points in a constant amount of time, constructing the Voronoi regions is of $O(n)$ complexity in this case. This can be substantially faster than creating the true Voronoi regions over the surface. For models with many more polygons than Voronoi regions, computing the *exact* Voronoi regions would take an amount of time that is independent of the number of Voronoi regions. In this case, the number of polygons that must be examined would dictate the computation time required to build the Voronoi regions.

It is important to consider whether the planar approximation to the exact Voronoi regions is acceptable for mesh building. There are two characteristics of the planar approximation method that make it an attractive alternative to Mount's exact technique. The first favorable aspect is the O(n) time complexity of the approximation method, in contrast to the longer time to compute the exact regions. The second favorable aspect is that the approximation method is a more simple method to implement. Ease of implementation has the positive side-effect of being easier to debug and make robust. The mesh generation method described above has been successfully applied to creating simulation meshes for a wide variety of polygonal models. These models include a hand-digitized model of a horse, a mathematically interesting minimal surface, an isosurface created using the marching cubes algorithm [Lorensen and Cline 87], and a model of a giraffe that was created freehand using a model-building program. The reaction-diffusion textures created using these meshes showed no apparent artifacts from the simulation mesh. The success of this mesh-building technique over a large number of models indicates that using an approximation of the Voronoi regions was a reasonable compromise between speed and accuracy. Furthermore, building the planar approximation to the Voronoi regions is independent of all other aspects of reaction-diffusion simulation and rendering. If the approximation was somehow found to be inadequate, Mount's method could be substituted and then all other aspects of the techniques presented here could be used with these exact Voronoi regions.

As an alternative to the Voronoi mesh, we could choose that the relaxed points be used as the *vertices* of the simulation cells instead of as the cell *centers*. This approach would yield a mesh that is the dual of the Voronoi diagram, called the Delaunay triangulation. If the Voronoi mesh had C cells and V vertices, then the corresponding Delaunay mesh would have V cells and C vertices, and the two meshes would have the same number of edges. Since the cells of the Delaunay mesh are all triangles, each cell would diffuse to exactly three neighboring cells, in contrast to the roughly six neighbors per cell in a Voronoi mesh. Although all the simulations in this dissertation used Voronoi meshes, using Delaunay meshes probably would have given satisfactory results. Further research is needed to determine the relative merits of the two approaches.

4.4.1 Meshes for Anisotropic Diffusion

The previously described construction of the Voronoi regions assumes that the diffusion over a surface is isotropic (has no preferred direction). The striking textures in [Witkin and Kass 91] show that simulation of anisotropy can add to the richness of patterns generated with reaction-diffusion. Given a vector field over a polyhedral surface, we can simulate anisotropic diffusion on the surface if we take into account the anisotropy during the construction of the Voronoi regions. This is done by contracting the positions of nearby points in the direction of anisotropy after projecting neighboring points onto a given point's plane. Then the Voronoi region around the point is constructed based on these new positions of nearby points. The contraction affects the lengths of the line segments separating the cells

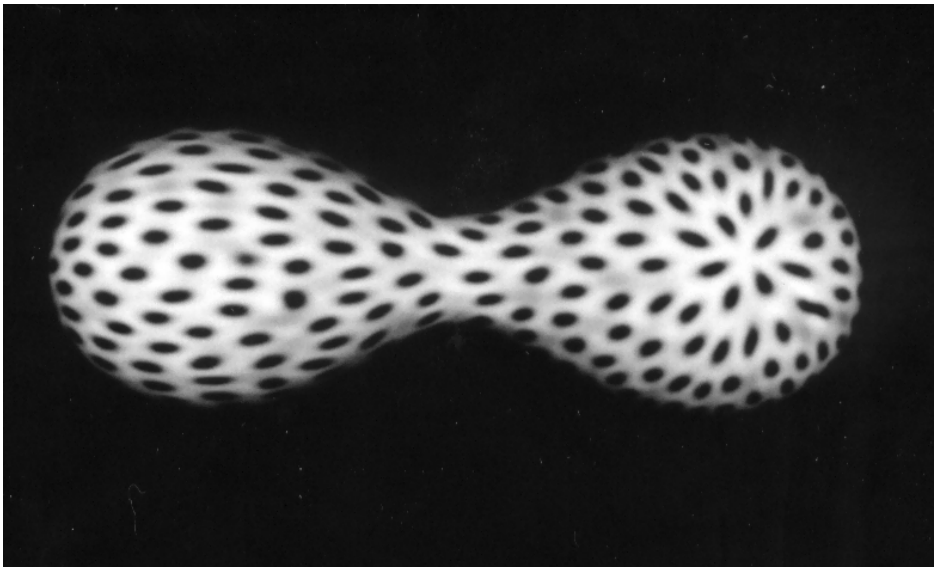

Figure 4.7: Spot pattern from anisotropic diffusion on test object.

and thus affects the diffusion coefficients between cells. The contraction will also affect which cells are neighbors. Figure 4.7 shows that anisotropic diffusion creates spots that are stretched when Turing's system is simulated on the surface of a model.

4.5 Simulation on a Mesh

The previous section gave a method for creating a simulation mesh for any polygonal model. We can now create any of the reaction-diffusion patterns from Chapter 3 on a given model by simulation on this mesh. The square cells of a regular grid are now replaced by the Voronoi regions that comprise the cells of the mesh. Simulation proceeds exactly as on a grid of

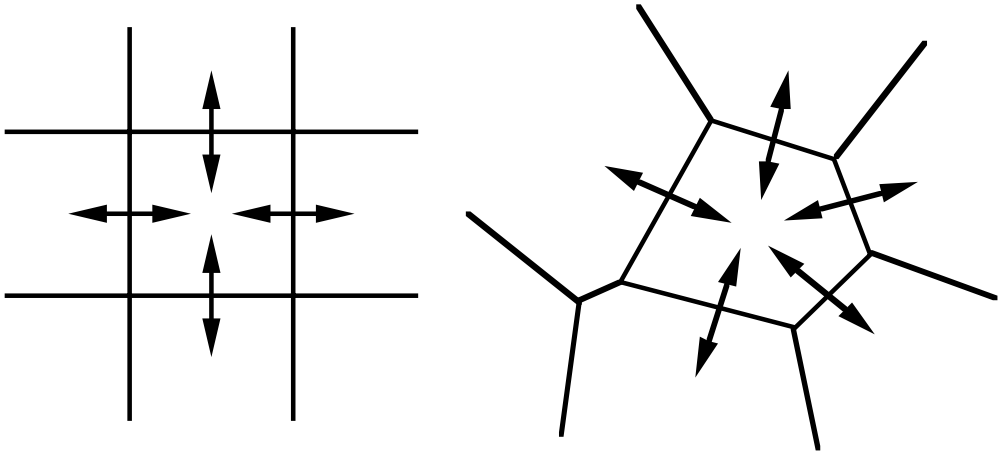

Figure 4.8: Diffusion between cells on square grid (left)
and between Voronoi regions (right).

squares except that calculation of the Laplacian terms now takes into account that the cell shapes are irregular. Figure 4.8 shows both the square grid case and the instance of a mesh of Voronoi regions. Consider determining the rate of diffusion of a chemical *a* at a particular cell. The diffusion term $\nabla^2 a$ on a square grid has the form:

```
da[i,j] =  (a[i-1,j] + a[i+1,j] + a[i,j-1] + a[i,j+1]) - 4a[i,j]
```

The diffusion quantity da[i,j] is computed from the value of the concentration a[i,j] at the cell and the concentrations at the surrounding cells. The computation is much the same on a mesh of Voronoi regions. The diffusion coefficients can be calculated by approximating the derivatives of concentration by a path integral around the boundary of the cell [Thompson et al 85, pages 156–157]. $\nabla^2 a$ is computed at a particular cell by multiplying each diffusion coefficient of the cell by the value of *a* at the corresponding neighboring cell, summing these values for all neighboring cells, and subtracting the value of *a* at the given cell. Let the concentrations of chemical *a* be stored in a one-dimensional array called a. Let coeff[i,j] be the diffusion coefficient between cells *i* and *j*. Then the Laplacian term da[i] for a particular cell *i* is computed as follows:

```
da[i] = - a[i]
for each neighbor j of cell i
        da[i] = da[i] + coeff[i,j] * a[j]
```

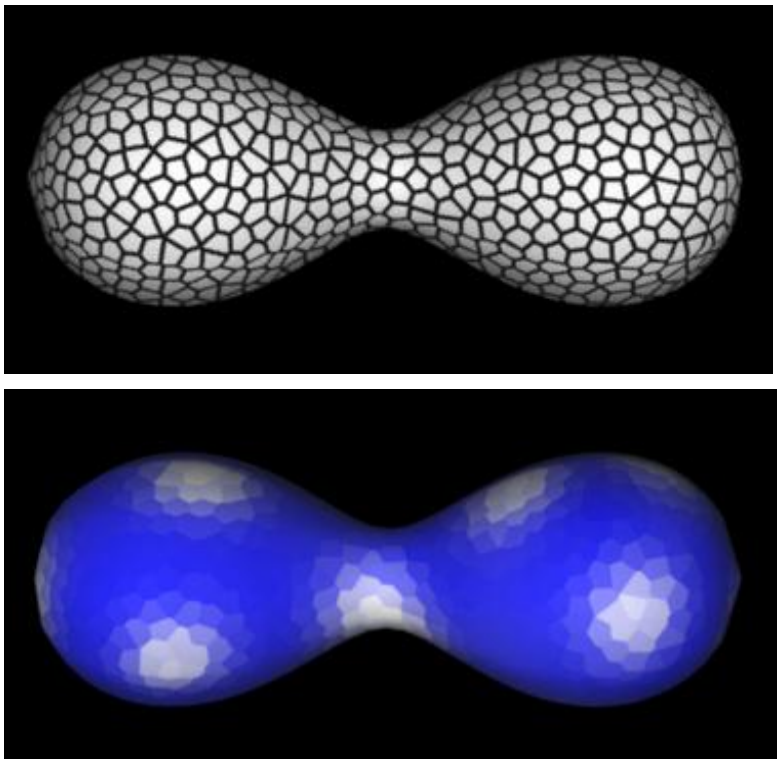

Figure 4.9: Reaction-diffusion on test object. Top shows Voronoi regions, bottom shows chemical concentration as color.

We have found that the diffusion coefficient between a pair of adjacent cells can be approximated by the lengths of the boundary separating the two cells. For a particular cell, these lengths are normalized by the circumference of the cell, so that the coefficients sum to one. The values given by this approximation are usually within 15% of the values given by the method described in [Thompson et al 85]. Most of the simulations in this dissertation used this boundary length approximation for the diffusion coefficients.

When a reaction-diffusion simulation over such a mesh is complete, the result is a concentration at each cell in the mesh for each participating chemical. Figure 4.9 shows the results of simulating Turing's spot-formation system on a mesh of Voronoi regions over a test surface. The color of the cells are based on the concentration of chemical *b*, where low values of *b* are shown blue and high values are white. The underlying simulation mesh is quite evident in this image. Chapter 5 shows that such patterns can be placed on surfaces without displaying artifacts of the mesh.

4.6 Pattern Control Across a Surface

User control is an important issue in synthetic texturing. We saw in section 4.5 that an arbitrary reaction-diffusion pattern can be placed on any given polygonal model. This gives a user the ability to choose from a wide variety of patterns such as those presented in Chapter 3. In this section we will examine several techniques for giving a user further control of reaction-diffusion textures. The methods presented here give a user control over the manner in which features vary over the surface of a particular model. The first technique allows a user to specify the placement of stripes on a model. The second technique demonstrates that diffusion can smoothly spread parameter values over a particular surface. This can be used to vary such characteristics as the size or regularity of spots and stripes over a model. The final technique allows parameters to be specified based on an approximation to the curvature of the underlying surface.

4.6.1 Stripe Initiation

The reaction-diffusion system that produced the random stripes in the lower left of Figure 3.4 can also be used to create more regular stripe patterns. The random stripes are a result of the slight random perturbations in the "substrate" for the chemical system. If these random perturbations are completely removed, then no stripes whatsoever will form. If, however, the substrate and chemical concentrations are homogeneous at all but a few special cells, then stripes will radiate from these few designated mesh cells. Such a stripe initiator cell can be made by marking the cell as frozen and raising the initial concentration of one chemical at that cell. Such a cell acts as a source of the particular chemical. The stripes shown in Figure 4.10 were created in this manner. These stripes were created by creating several stripe initiator cells on the head and one such cell on each of the hooves. Figure 4.11 shows the stripes radiating from these positions part-way through the simulation. These cells were marked as unchanging and the initial value of chemical g_1 was set to be slightly higher than at other cells (see Appendix B for the simulation equations).

Figure 4.10: Zebra stripes on horse model, created by simulating reaction-diffusion on surface.

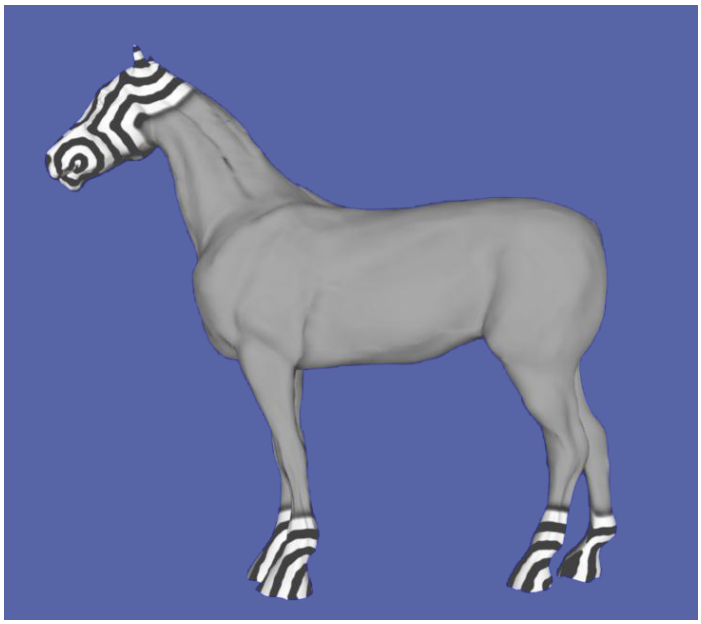

Figure 4.11: Stripes are initiated at the head and hooves.

Figure 4.12: Result of flood-fill over horse model from key positions.

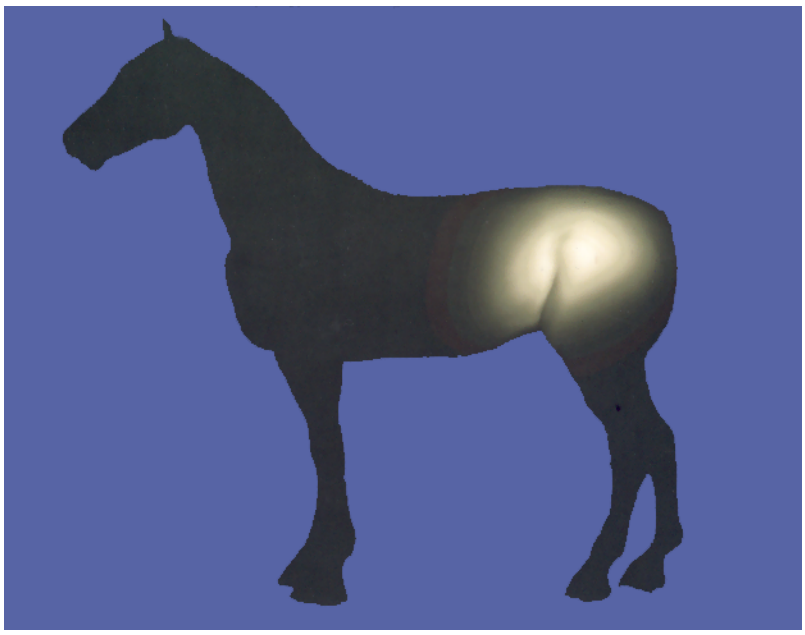

Figure 4.13: Values from Figure 4.12 after being smoothed by diffusion.

4.6.2 Parameter Specification Using Diffusion

Section 4.5 demonstrated that we can place any single reaction-diffusion pattern onto an arbitrary surface. Sometimes a user might want to have a pattern vary based on position on a surface. For example, we might recognize that the stripes on a zebra are thicker on the zebra's haunches than elsewhere on its body. This variation could be achieved by slowing the diffusion rate on and near the haunches in contrast to the rate of diffusion over the rest of the model. There are many ways this variation in the diffusion rate might be specified. We will discuss one such method that uses diffusion over a simulation mesh to smooth our parameter values over a surface. This method was used to create the variation in stripe thickness for the model in Figure 4.10.

The first step to specifying variation in a particular parameter is to specify the value of the parameter at particular key positions. These key positions will be particular cells in the simulation mesh. For the example of the zebra, twelve key positions on the haunches were designated to have low diffusion rates (call it r_1). Twelve more key positions on other parts of the body were marked as having high diffusion rates (r_2). Then these parameter values were spread outward cell-by-cell from these special cells to other parts of the mesh. This stage can be thought of as a flood-fill of the cells in the mesh. A given cell on the mesh receives either the value r_1 or r_2 depending on which key position is closest. At the end of this process, all the cells of the mesh have one or the other parameter value. Figure 4.12 shows these values, representing a high value as black and low value as white. The next stage in this process is to smooth out these parameters to remove the sudden jump in values at the border between black and white.

Diffusion of the parameter values over the simulation mesh can smooth out this discontinuity of values. (This should not be confused with reaction-diffusion. There is no reaction process here.) As was discussed in section 3.1, diffusion acting alone has the tendency to smooth out any spatial variation of a substance. Figure 4.13 shows the results of allowing the parameter values of Figure 4.12 to diffuse for 400 time steps. Notice that the sharp jumps in value have been removed. The numerical values illustrated in Figure 4.13 could be used to vary any parameter of a reaction-diffusion system. Two of the parameters it could vary are the degree of randomness or the rate of diffusion. The stripes of Figure 4.11 were created by using the parameter variation shown in Figure 4.13 to specify the rate of diffusion over the model. This created the variation in stripe width near the hind quarters.

4.6.3 Using Curvature to Specify Parameters

Another way to specify parameter variation over a model is to allow attributes of the surface to contribute to the parameter values. This section demonstrates that a surface's curvature can be used to set parameter values for reaction-diffusion. Suppose we notice that a giraffe's spots are smaller at places of higher curvature on its body. We would like to use the amount of curvature to guide the rate of diffusion during pattern simulation on a giraffe model. Figure 4.14 shows an estimate of curvature on a giraffe model, and Figure 4.15 is the same model with a texture whose spots vary according to this curvature information.

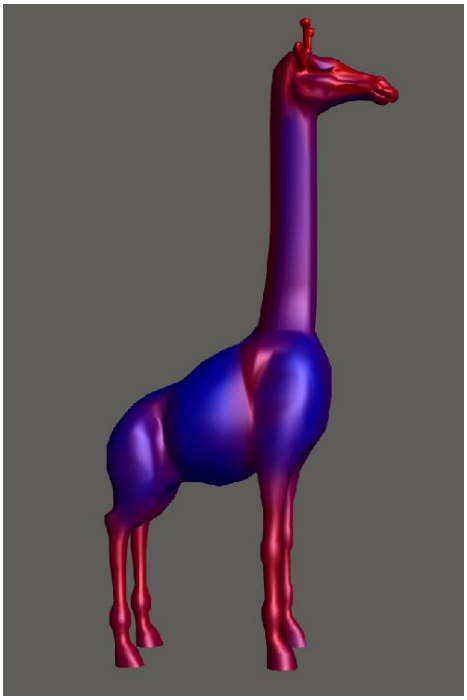

Figure 4.14: Approximation of curvature over giraffe model. Areas of high curvature are red and areas of low curvature are blue.

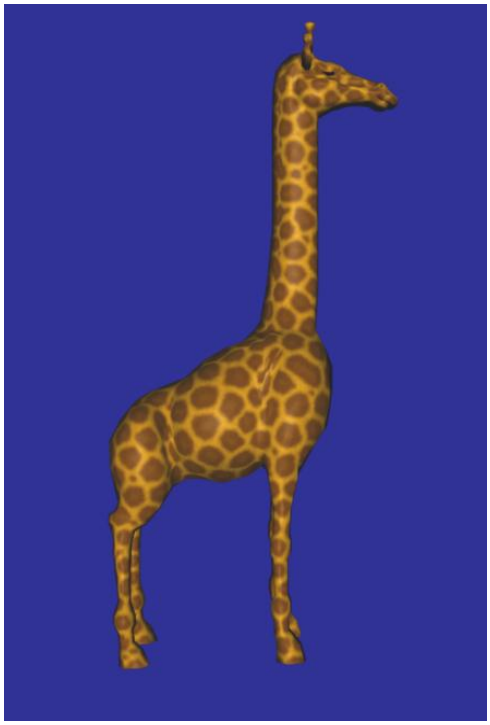

Figure 4.15: Size of spots on giraffe model are guided by the curvature estimate shown in Figure 4.14.

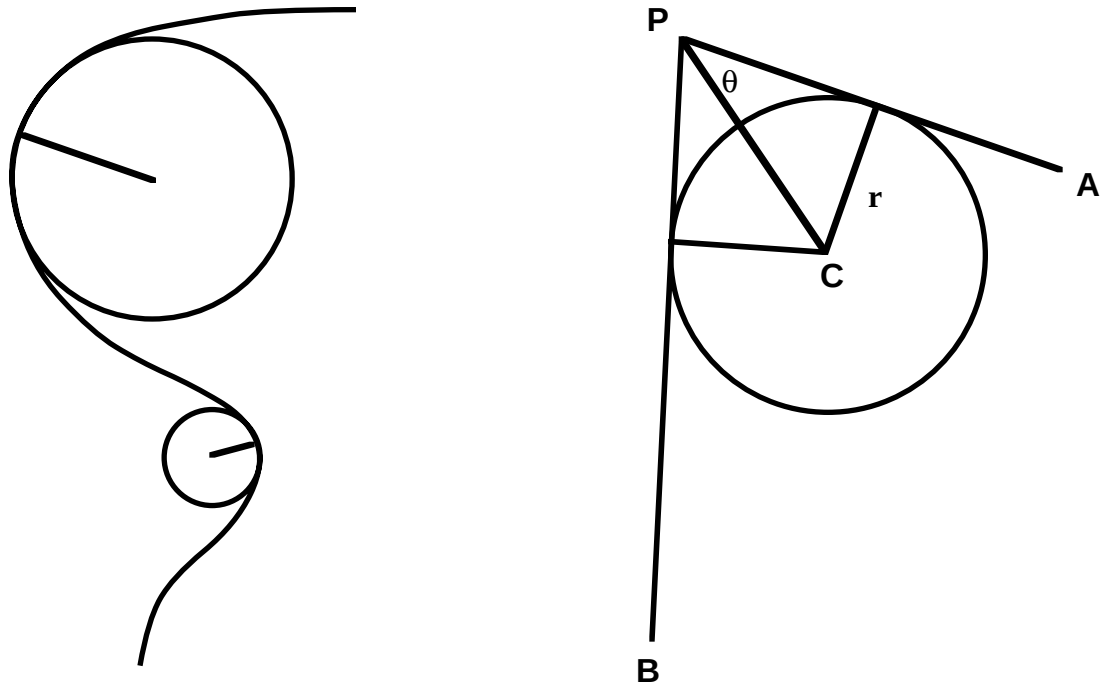

Figure 4.16: Curvature along a path in the plane (left) and curvature approximation used at vertices (right).

The curvature estimate used for Figure 4.14 was presented in [Turk 92], and we will describe that estimate below. Ideally, we would like to have an exact measure of curvature from the object that the polygonal model is meant to represent. Often, however, this information is not available, either because the object being represented is not available (e.g. volume data was not retained) or because there never was an exact description of the object (e.g. a cat model was created freehand by a human modeler). For these reasons it is useful to have a way to approximate surface curvature from the polygonal data alone. More precisely, we want to know the maximum principal curvature at any given point on the model. See any text on differential geometry for a mathematical description of principal curvature, such as [O'Neill 66]. Intuitively, this is asking for the radius of the largest sphere that can be placed on the more curved side of the surface at a given point without being held away from the surface by the manner in which the surface curves. The left portion of Figure 4.16 shows the radius of curvature at two points along a curve in the plane.

The right portion of Figure 4.16 illustrates the curvature approximation used here. This figure shows the two-dimensional version of the curvature estimate near a point *P*. Here a circle has been drawn that is tangent to the edge *PA* at its mid-point and that is also tangent to the longer edge *PB*. The radius of this circle is $r = \tan(\theta) \, |P - A| / 2$. In this figure, the line segment *PC* bisects the angle *APB*. This figure will act as a starting point for approximating the curvature of a polygonal surface in 3-space at a vertex *P*.

In the three-dimensional case, the line segment PC is replaced by an approximation to the surface normal N at the vertex P. Then, each edge in the polygon mesh that joins the vertex P to another vertex Q_i is examined, and an estimate of the radius of curvature from each of the n edges $PQ_1, PQ_2, \ldots PQ_n$ can be computed. Let V be the normalized version of the vector $Q_i - P$, that is, a unit vector parallel to the edge PQ_i. Then an estimate for θ_i is $\arccos(N \bullet V)$, and the radius estimate for the edge PQ_i is $r_i = \tan(\theta_i) \mid P - Q_i \mid / 2$. The final estimate r of minimum radius of curvature at the vertex P is the minimum of all the r_i. This estimate of curvature is a little noisy for some models, so we can smooth the estimate by averaging a vertex's radius r with that of all of its neighbors, and we can take this to be the minimum radius of curvature at the vertex. Figure 4.14 shows the results of this estimate, where the surface is colored red in areas of high curvature (small radius) and is colored blue in regions that are more nearly flat.

Curvature information can be used to specify any parameters of a reaction-diffusion simulation. These include reaction rates, speed of diffusion, random substrates, or initial chemical concentrations. The spot sizes of Figure 4.15 were set by varying the diffusion rates of a cascade system. It is likely that other surface properties will also prove useful in specifying reaction-diffusion parameters. Some of these properties are: direction of maximum curvature, surface normal direction, mean curvature, and Gaussian curvature. Exploring the uses of these surface properties is a logical direction for future research.

4.6.4 Future Work in Pattern Control

Future work on pattern control over a surface should concentrate on user interface issues. For example, a user should be able to specify the size of the stripes or spots directly instead of providing numerical values for parameters. Specifying feature sizes would be much more intuitive with the aid of a graphical pointing device such as a mouse. The user might point at one portion of a giraffe model and drag the pointer over the model to indicate the average width of a spot at that position.

5 Rendering Reaction-Diffusion Textures

Rendering a textured object is the process of bringing together the information about the texture, the mapping of the texture, and the object's geometry to create a final image. This chapter is concerned with how to render scenes that contain objects that are textured using reaction-diffusion patterns. The chemical concentrations from a reaction-diffusion simulation are converted into colors to give a final texture. This is analogous to a biological model in which the concentration of a morphogen triggers the production of a pigment in an animal's skin.

We will begin by examining the major issues of rendering image-based textures, namely projection and texture filtering. This overview will give us a point of reference for examining two new methods of rendering reaction-diffusion textures. We require new methods to render the patterns resulting from simulation on a mesh because our mesh cells are irregularly distributed over a model. Previous texture rendering methods used regularly spaced grids of color information. The first of the new rendering methods re-tiles the textured model with small polygons that are colored based on the texture. This allows rapid display of textured models on graphics workstations. The second method renders from the original surface's geometry and uses a weighted average of texture values to interpolate smoothly the reaction-diffusion texture pattern. This is a more compute-intensive rendering approach, but the final images are usually of higher quality than those from the re-tiling method. The same issues that are important for image-based textures will allow us to evaluate the quality of these two techniques for rendering reaction-diffusion textures. Both the methods presented to render reaction-diffusion textures are new contributions of this dissertation.

5.1 Rendering Image-Based Textures (Previous Work)

In this section we will examine the rendering process and look at several visually distracting artifacts that can appear in textured images. Much of the material presented here is taken from [Heckbert 89]. First, we will look at the fundamentals of creating a textured image by using an image-based texture to vary the color across an object's surface.

Photorealistic rendering has concerned itself almost exclusively with creating a rectangular array of color values that are output to such devices as color cathode-ray tubes, liquid crystal displays and film scanners. The task of rendering is to determine what the color components (most often red, green, blue) should be for each pixel of the image. When a textured object is rendered, one issue is bringing the color information in the texture to the correct place in the final image. For instance, what process brings the blue dome from the center of Figure 1.8 to the appropriate position in the right portion of the figure? We need to find the

appropriate correspondence between positions in the texture and positions in the final image. This information is found in two places: first, in the mapping function that the user describes to wrap the texture around the object; and second, in the projection from the three-dimensional scene description onto the two-dimensional screen.

5.1.1 Transformation Between Texture Space and Image Space

Let us define a few functions that will allow us to discuss the correspondence between positions in an image-based texture and screen positions. Let the function g: $\mathbf{R}^2 \rightarrow \mathbf{R}^3$ describe the mapping of a point in texture space into a three-dimensional position on the object's surface in the scene. For a particular image of the scene, we also have a mapping from points in the three-dimensional scene to their projected positions on the screen, and we will call this function h: $\mathbf{R}^3 \rightarrow \mathbf{R}^2$. The composite function $h \bullet g : \mathbf{R}^2 \rightarrow \mathbf{R}^2$ describes mapping a position in the texture to the final image. Figure 5.1 illustrates the functions g and h. The composite mapping function $h \bullet g$ gives one way to render textured objects: For each texture element at texture position (s,t), apply the function $h \bullet g$ to get a position $(x,y) = (h \bullet g)\ (s,t)$ in the final image. The color values texture[s,t] of the texture are then used to determine the final color at the screen location screen[x,y] in the final image. This is *texture order* mapping, and can be written as:

```
for s
  for t
    (x,y) = (h • g) (s,t)
    screen[x,y] = texture[s,t]
```

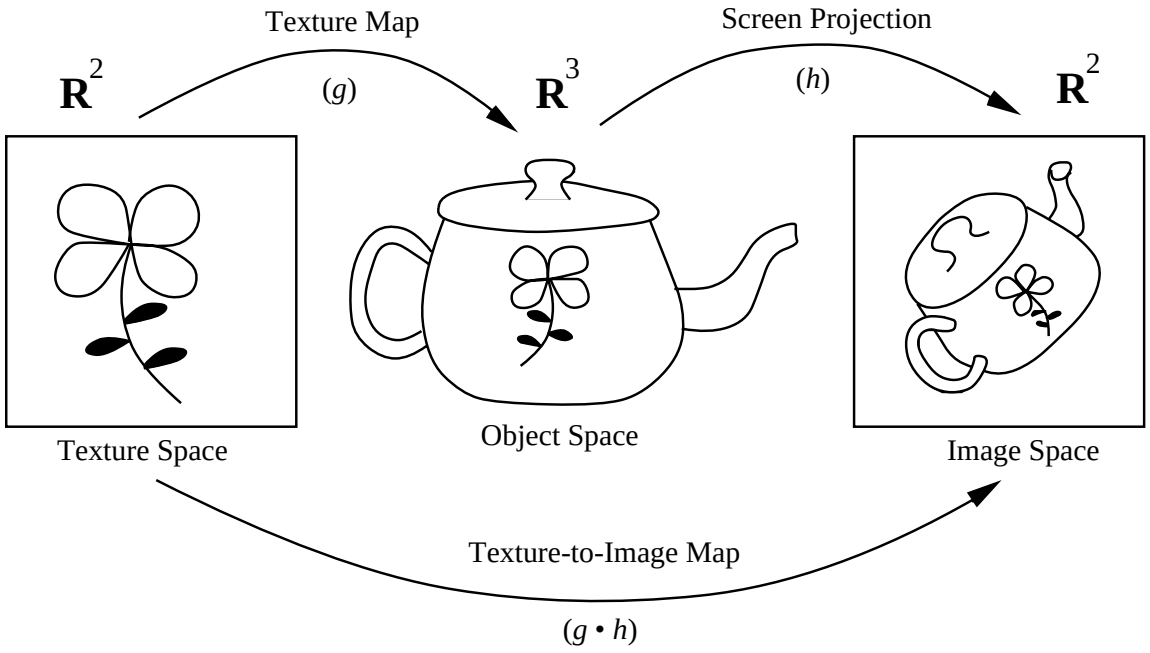

Figure 5.1: Mapping from texture to image space.

To a first approximation, this is the forward mapping method of rendering textured objects. For the sake of simplicity, this description ignores the issues of lighting models, hidden surface removal and sampling.

The forward texture mapping approach has two potential difficulties. The first problem is that there may be holes in the textured surface because no texture elements mapped to some of the pixels in the final image. This problem can be overcome by careful attention to how much the mapping function $h \bullet g$ stretches the texture. This information about stretching is contained in the Jacobian of the mapping function. The second problem is correctly filtering the texture. This is a complex issue that we will return to in section 5.1.2. A solution to both these issues was presented in [Levoy and Whitted 85], where the authors mapped texture elements to fuzzy points whose contributions were summed in the final image. The contribution of a fuzzy point varied in cross-section according to a Gaussian distribution. A similar approach was taken in [Westover 90] to render volume data. Westover used the regularity of the 3-D grid of volume data to sort the Gaussian footprints, which makes combining their contributions simple.

Due to the difficulties mentioned above in the forward mapping approach, the more common approach to texture mapping is to reverse the above process. Let the function f represent the composite mapping function: $f = h \bullet g$. Then we can use the inverse function f^{-1} to determine which position (s,t) in the texture space corresponds to a given position (x,y) in the final image: $f^{-1}(x,y) = (s,t)$. Texture rendering proceeds as follows: Render the objects in the scene using some surface algorithm such as ray tracing or polygon scan conversion. When a screen pixel at position (x,y) is determined to be covered by a portion of an object, the reverse mapping $f^{-1}(x,y)$ gives a location (s,t) in the texture space where the color values in texture$[s,t]$ can be looked up directly. This method is called *screen order* mapping, and can be written as follows:

```
for x
  for y
    (s,t) = f^-1(x,y)
    screen[x,y] = texture[s,t]
```

This method of texture rendering is the more commonly used approach of the two for determining the color of an object. In a real renderer, the color value texture$[s,t]$ would be passed along with additional surface properties to a lighting model, and the result of this computation would be stored at screen$[x,y]$.

5.1.2 Aliasing and Filtering

Both screen and image order mapping depend on determining a correspondence between a point in the texture space and a point in the final image. It is rarely correct to use just a single entry from a texture color table to determine the color of a pixel in the final image. If, for instance, an object is very far away from the viewer, the image of that object may cover only a handful of pixels in the final image. This means that any detail from a texture on the object

is compressed into a very small number of pixels, so that many texture elements should contribute to just one pixel's color. The above sketches of forward and inverse texturing do not address this problem because they do not account for sampling. The naive method of inverse texturing looks up a single texture element based on the color value from the inverse mapping of screen position (x,y). This is known as *point sampling*. The left half of Figure 1.9 shows an image of a checkerboard using this method of point sampling. Notice the distracting moiré patterns near the horizon of the image and the jagged edges between each of the black and white squares. Both of these image artifacts are a result of *aliasing*. Aliasing is the result of trying to re-create a function from an inadequate number of samples of the function. More precisely, aliasing is an artifact that results from high-frequency information of a signal erroneously appearing as low-frequency information. In the case of the checkerboard, the moiré patterns are a result of not properly averaging the texture values near the horizon. The jagged edges between the squares are a result of not collecting enough texture information near the edges between squares. To avoid these kinds of artifacts, we need to filter the textures to remove these high-frequency components.

The problem of aliasing is well-understood, and the issues involved in sampling a signal such as a texture are described in the literature on signal processing. The main approach to avoid aliasing is to remove the high-frequency components by *filtering* the signal (e.g. the texture) before it is sampled (e.g. represented as pixels). A *band-limited* filter takes a given signal and removes the high frequencies that cannot be adequately represented in the final, sampled representation of the signal. Hopefully, this filter will retain the low-frequency information that can be represented. Refer to books on image processing such as [Castleman 79] or [Oppenheim and Schafer 75] for more information about aliasing, filtering, and sampling as they relate to images.

To gain an understanding of texturing artifacts and how to avoid them, we need a conceptual model of the entire texture rendering process. Rendering creates discrete values on a regular grid (the final image) from a set of discrete values in another regular grid (the image texture). For the sake of simplicity, let us assume that both the texture and the final image are black and white so that we can view them as real-valued functions. As a further simplification, we will consider both texture and image to be one-dimensional samples instead of two-dimensional grids of values. Figure 5.2 (from [Heckbert 89]), parts (a) and (e) show representations of the texture and the final image. All the functions in this figure are real-valued functions, and these two functions in particular are zero everywhere except at regularly spaced sample points. The arrows in the figure show the four processes involved in the rendering of textures. The first process is *reconstruction*, where a continuous function is created from the original, sampled function (in our case, the discrete grid of the texture). The second step is to *warp* this continuous version of the texture, using the mapping $h \bullet g$. This is the process of squashing, stretching or otherwise transforming the texture to place it on the final image. The third step is to *pre-filter* the warped image, and we discuss this step in detail below. Here we will use the term *pre-filter* to mean a filter that is applied before resampling. The final step is to *sample* the pre-filtered function, and the result of this step gives us the final image. Figure 5.3 gives a two-dimensional illustration of these four processes.

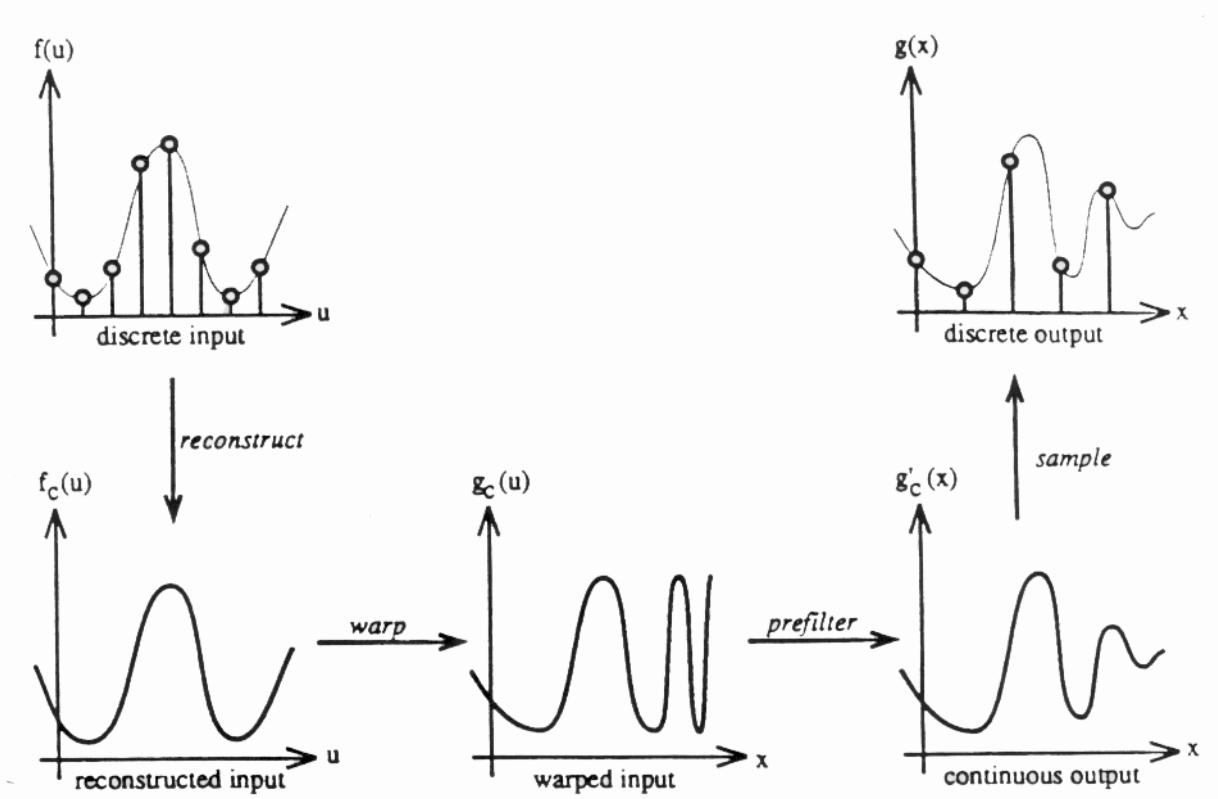

Figure 5.2: Conceptual model of the processes involved in rendering a texture.

Two of the four processes shown in Figure 5.2 are already dictated by the given scene that is to be rendered. The warping step is completely determined by the way the texture is mapped onto an object and from the position of the object in the scene. We represented this warping by the function $f = h \bullet g$ above. Sampling, the last stage of the model, is fully determined by the size of the final image. We are free to make decisions about both the reconstruction and the pre-filtering processes. Let us take a closer look at reconstruction. This process is the recognition that the discrete color values that make up an image-based texture really are meant to represent some continuous color function. For our application, reconstruction is the process of re-creating, as best we can, that continuous function. One simple reconstruction method is to use linear interpolation between adjacent values in a texture. Say, for instance, we wish to know the color of a texture at a position one-third of the way between a completely black texture element (value 0) and an adjacent texture element that is pure white (value 1). Linear interpolation of these color values gives a value of 1/3 at the place in question. In general, if we want a color value that is the fraction t of the way between texture values texture[i] and texture[i+1], then the linearly interpolated value is:

value = texture[i] + t * (texture[i+1] - texture[i])

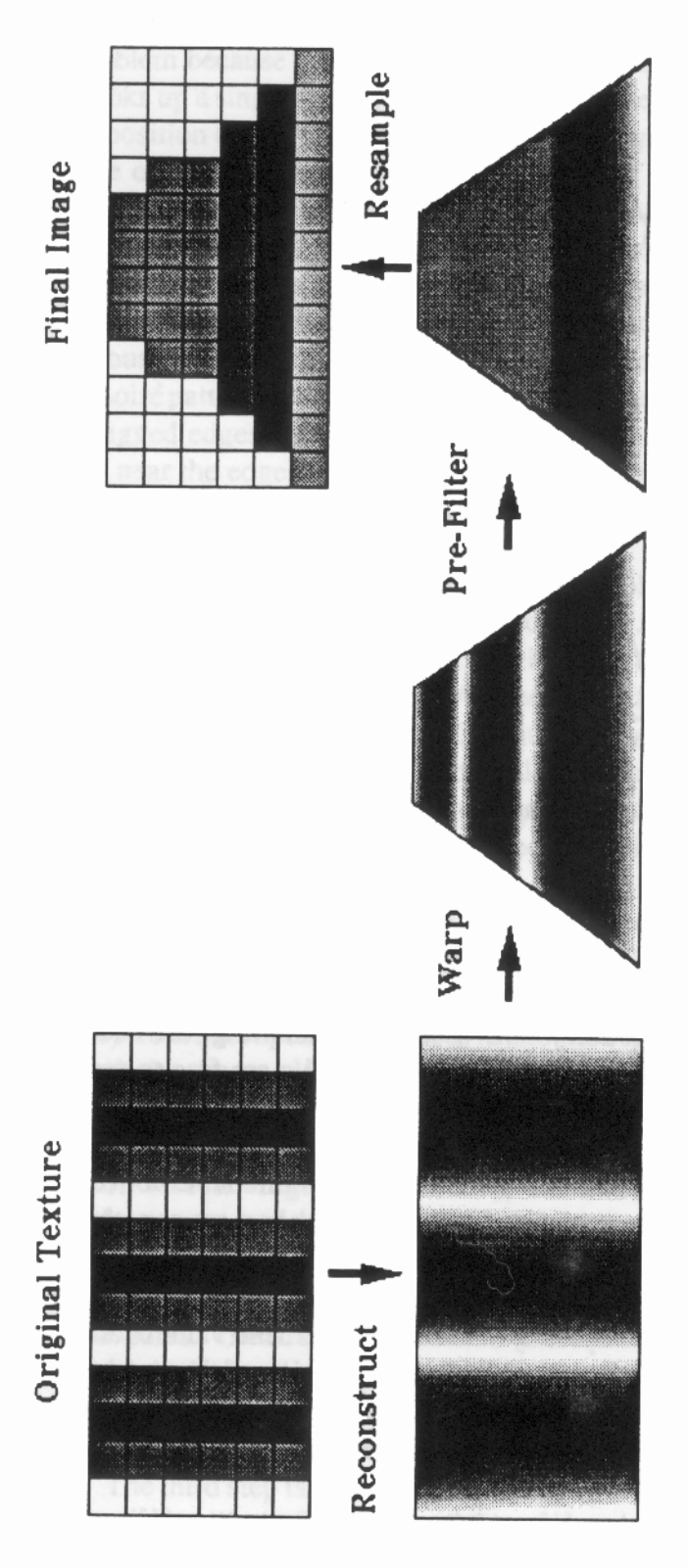

Figure 5.3: Two-dimensional illustration of the rendering process.

Linear interpolation is the same as convolution of a discrete function with the triangle filter:

$$\text{triangle}(t) = \max(0, 1 - |t|)$$

Convolving a discrete function with an appropriate filter is in fact the most common view of reconstruction. Linear, quadratic and cubic reconstruction filters are all common in computer graphics. Much work has been done exploring the qualities of different reconstruction filters. Reconstruction filters are covered in many books on signal processing such as [Oppenheim and Schafer 75]. See [Mitchell and Netravali 88] for a good overview of this topic as it relates to computer graphics.

Let us now turn to the pre-filtering stage of texture rendering. The purpose of this stage is to band-limit the warped texture function before it is discretely sampled to create the final image. Ideally, this pre-filter should remove any frequencies in the function that are above the Nyquist limit given by the sampling rate of the final image (see, for example, [Castleman 79]). This will remove the high-frequency information that could, due to aliasing, appear as low-frequency information. Here again we are convolving a function with a filter. Sampling theory tells us that the best shape for this filter is the sinc function ($\text{sinc}(x) = (1/x)\sin(x)$) because only this filter cuts off high frequencies completely and leaves the lower frequencies untouched. Unfortunately, the sinc function gives some difficulties in practice. One problem is that it is infinite in extent. This means that *every* function value from part (d) of Figure 5.2 contributes to any given discrete value in part (e), which means a high computational expense. Secondly, the sinc function can produce *ringing* (visible ripples near edges) which is the result of the many closely spaced lobes of the sinc function. For these reasons, other filters are used in practice. Two popular families of filters are the Gaussian filters and cubic filters.

Although the processes of reconstruction and pre-filtering can be separated conceptually, often they are implemented together in one part of a program. When a texturing algorithm has the two processes tightly intertwined, they are referred to as a single process, the *texture filter*. Under certain conditions, one component of the texture filter is more prominent than the other. Depending on the position of a textured object in a scene, either the reconstruction or pre-filtering process may dominate the look of a texture. If an object is very close to the viewer then the reconstruction filter becomes important because a very small number of texture elements may cover a large portion of the screen. In such cases, the interpolation method used between texture elements dominates the look of the texture. When this happens, the texture filter is sometimes called a *magnification filter*. The opposite extreme occurs when a textured object covers very few pixels of the final image. Then the texture filter's job is to average many of the texture elements together to give one color value. Here, the pre-filtering is foremost in importance, and such a filter is called a *decimation filter*.

5.1.3 Two Texture Sampling Methods: Point Sampling and Mip maps

Let us examine two concrete examples of filters for image-based textures. These two filters are similar to the two methods of rendering reaction-diffusion textures that will be described

later. The first texture "filter" we will look at is point sampling of the texture. This is the naive method of texturing that was outlined in the description of screen order mapping of textures. The texture value is simply the color of the closest texture element to the inverse function value $f^{-1}(x,y)$. This is actually the absence of any filtering, although it can be viewed as convolution with a box filter. Conceptually, the reconstruction filter is a box filter that stretches halfway between two texture elements. This means that *no averaging* between neighboring texture elements is performed. This filter will give blocky looking textures when a textured object is close to the viewer. The pre-filter is an impulse function centered at the origin. This means that the filter uses just one texture element for the color, no matter how small the object appears in the final image. The effect of this will be most noticeable if such an object is moving, in which case the texture will sparkle and shimmer distractingly as different portions of the texture move under the sampled positions. Clearly, point sampling makes a poor texture filter. The re-tiling method of displaying reaction-diffusion textures (described later in this chapter) has much in common with the point sampling method of image-based texture rendering.

A substantially better texturing technique called a *mip map* was introduced by Lance Williams [Williams 83]. Map maps should be thought of as a family of filters, and a specific filter is given by the particular mip map implementation. Mip maps are now in common use, probably because they are easy to implement and because they have a low, fixed computational cost per textured pixel. To motivate this fixed cost, consider what happens when a pixel in a final image spans a large portion of a texture. The pixel's color should be an average of the colors from a large region of the texture, so a single lookup into the original texture will be inadequate. Depending on the size of this region, we may have to make several hundred texture lookups to find the appropriate average. If we create several pre-filtered versions of the texture, however, just a few lookups into these pre-filtered versions will suffice to give a reasonable color average over much of the texture. Each of these pre-filtered versions of the texture is an appropriately band-limited (blurred) version of the original texture. The collection of pre-filtered textures is what the image processing literature calls an *image pyramid*. The high-quality method of displaying reaction-diffusion textures (described later in this chapter) is similar to mip maps.

A mip map is just such a technique that uses multiple pre-filtered versions of a texture. Figure 5.4 is a diagram of a mip map. Consider a 256×256 element black and white texture. The first level of the mip map is the original array of 256×256 texture values, shown at the base of the pyramid at the right of Figure 5.4. The second level is a 128×128 version of the texture in which each element is the average of nearby elements (typically four) from the first level of the mip map. The additional levels are smaller by a factor of 2 on each side from the previous level, and each of their elements are averages of texture elements from the previous level. The final level of the mip map is a single value that is the average value over the entire texture.

Finding a texture value using a mip map begins by determining the amount of compression or stretching of the texture at a given pixel. A typical measure for this compression factor is the size that a one-pixel circle on the final image would be if it was inverse mapped back

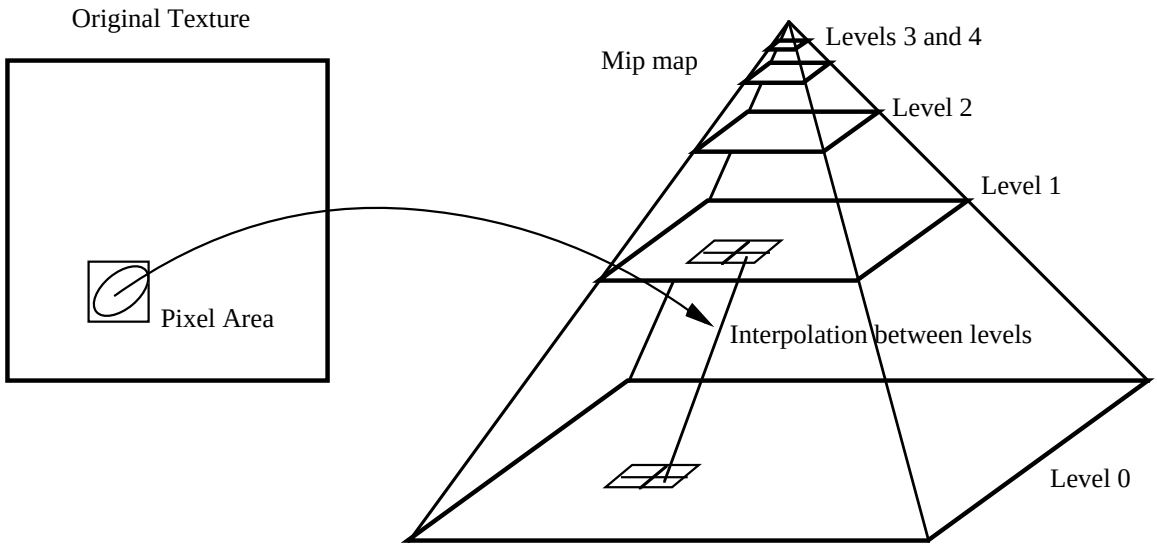

Figure 5.4: Pixel's area mappped into texture space (left) and the appropriate filtering computed from the mip map (right).

into texture space. This will usually give an elliptically shaped region. The equation given in [Williams 83] for the compression factor is:

$$d = \max\left(\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\right)$$

This value *d* determines the two levels of the mip map that will be used to compute the filtered texture value. It is a measure of the size of the smallest bounding square around the elliptical pixel-coverage region shown and the left of Figure 5.4. The larger the bounding square, the higher the levels we will use in the mip map. Most of the time, the value of *d* will not specify one particular level, but instead will indicate that the resulting color should be a blending of values taken from two adjacent levels of the mip map. The pixel's color is given by averaging the values found from these two pre-filtered versions of the texture. The mip map approach to filtering allows any number of ways to interpolate between the levels, and it also gives us a choice about how a particular value within one level is computed. Linear interpolation is the most commonly used choice in both cases, and this is what we will describe here. Given a position (*s*,*t*) in texture space, the texture value within one level of the mip map can be computed using bi-linear interpolation between the four nearest texture elements within a level. This is done for both the levels indicated by *d*, and then the fractional portion of *d* is used to linearly interpolate between the values at the two levels to arrive at the final filtered texture value. This value is the result of seven linear interpolations between a total of eight values in the mip map. It is this fixed cost that makes the mip map approach computationally attractive.

It is instructive to contrast point sampling with the mip map filter described above. The filter give by a particular implementation of a mip map is a concatenation of several transfer functions, and this makes an exact analysis of a mip map filter difficult. To my knowledge, no analysis of any filter that has been implemented using a mip map has appeared in the graphics literature. The filter used in the above mip map is partially determined by the bi-linear interpolation within a level. The filter shape is that of a four-sided pyramid and is sometimes called a *Bartlett filter*. When the mip map turns into a magnification filter, the resulting textures look a good deal smoother than a point sampled texture because the linear interpolation makes the individual texture elements less noticeable. The second determining factor in a mip map's filter is the way one level is computed from the next lower level in the pyramid. In the case of the implementation described above, point sampling is equivalent to convolving with a box filter that is aligned with the axes of the texture space. When a textured object is small, the mip map texture gives a rough average of all the texture elements present within a pixel. For surfaces that are more nearly edge-on, the pre-filter of the mip map will cause more texture elements than necessary to be averaged together, resulting in over-blurring in one direction. This is a good deal less distracting than the sparkling that results from point sampling. Overall, this mip map implementation does a fair job of filtering a texture at a very reasonable computational cost. An analysis of mip map implementations of filters is a topic for further research.

Point sampling and mip maps are by no means the only ways to filter textures. Other texture filtering methods include summed-area tables [Crow 84] [Glassner 86], repeated integration [Heckbert 88], space-variant kernels [Fournier and Fiume 88], and the Elliptical Weighted Average filter [Greene and Heckbert 86]. We examined point sampling because the method of rendering described in Section 5.3 has much in common with point sampling. We examined mip maps in detail because they are similar to the high-quality method of rendering reaction-diffusion textures that are described in Section 5.4. They are similar in the way that they interpolate between two low-pass filtered versions of a given texture and because they perform interpolation between sample points within a given filter level.

5.2 Interactive Texture Rendering

We would like to view reaction-diffusion textures interactively on graphics workstations. In order to achieve interactive display rates we may need to make compromises in the quality of the final image. In this section we will examine methods that have been used to display textures rapidly using graphics hardware. Then in section 5.3 we will describe a fast technique for displaying a reaction-diffusion texture that has been made for a specific model.

The most common method of generating textures in real-time is to use special-purpose hardware that is designed to render image-based textures. The first texturing hardware was built for flight simulators, and to this day most of the graphics machines that have rapid texturing capabilities are targeted for the vehicle simulation market. The details of texture rendering on present-day hardware varies from machine to machine, but the overall approach to real-time texturing is quite consistent across vendors. Texture lookup and filtering is

usually performed as a final step just after primitive rasterization (e.g. polygon scan-conversion) and directly before sending the pixel's color information to the framebuffer. The basic approach is to have one or (more often) several processors take a surface ID and texture coordinate information from the rasterization stage and perform a texture lookup. These texture processors typically have large amounts of memory to store several textures. Textures are usually stored at several levels of detail (as is done for mip maps) although the exact form of these levels varies among architectures.

An unconventional alternative to image-based texture hardware was demonstrated by the Pixel-Planes group [Rhoades et al 92]. Pixel-Planes 5 is a graphics engine that uses 128×128 arrays of one-bit processors to rasterize polygons [Fuchs et al 89]. Although these small processors only have 208 bits of memory each and operate in SIMD fashion, they provide enough processing power to re-compute some procedural textures anew each frame. This method of texturing is only acceptable when the desired textures can be represented as simple functions that do not require large amounts of data to evaluate. The technique cannot provide a texture from an arbitrary image. Examples of textures that have been displayed interactively using this approach include wood, ceiling and floor tiles, sheet music, fire, and bricks. Scenes that contain a dozen such procedural textures can be displayed at about ten frames per second on Pixel-Planes 5.

When using a fast polygon display engine with no special-purpose texture hardware, textures can be displayed as collections of many small polygons. This approach often strains the polygon budget on such devices, however. If the texture is a 128×128 array of color values, for example, then $128^2 = 16{,}384$ squares can be used to render the texture. Polygon display engines often provide linear interpolation of colors across polygons. Color interpolation across the texture element squares can be used to improve the look of the texture when the textured object is near the viewer and the squares cover larger areas of the screen. Often the graphics hardware provides linear interpolation of colors between polygon vertices. When the texture squares become much smaller than pixel size, however, this texture method is equivalent to point sampling, and the texture will sparkle objectionably.

The next section will describe a technique for rendering reaction-diffusion textures on graphics workstations with no special-purpose hardware.

5.3 Using Surface Re-Tiling for Rapid Rendering

This section describes the rapid display of reaction-diffusion texture on a graphics workstation. The motivation for this is to give rapid feedback to the user about the overall look of a reaction-diffusion texture. Using the technique presented in this section, a user can interactively change the position and orientation of an object and see these new views immediately. For example, the horse model shown in Figure 5.6 was displayed at a rate of more than 15 frames a second on Pixel-Planes 5, a high-end graphics engine [Fuchs et al 89].

Recall from Chapter 4 that we can simulate any reaction-diffusion system on a mesh that is fit to a particular surface. The results of this simulation are chemical concentrations at the

centers of the mesh cells. The basic method of texture rendering described in this section is to re-tile a given surface based on a simulation mesh. This re-tiling process results in a new model that is composed of polygons that are colored according to a pattern created by reaction-diffusion. This is similar to displaying an image-based texture using a large collection of color-interpolated squares or rectangles. Re-tiling based on a reaction-diffusion pattern has the same image quality problems that are found when rendering image textures using collections of rectangles. However, this method has a speed advantage over the higher-quality rendering technique that will be described later in this chapter.

The approach of this re-tiling process is to throw away all the old polygons of the model and replace them with new polygons that are placed according to the cells in the simulation mesh. Then the chemical concentrations at each cell will dictate the colors of the vertices of the surrounding polygons. A graphics workstation can then interpolate these colors across the polygons and display the model at interactive rates. It is important to recognize that the polygons of the original model cannot be used for this purpose because the simulation mesh can have many cells for any single polygon of the model. Details of the texture would be lost if the frequency contents of the pattern are higher than the density of the polygon vertices. We want to create a new collection of polygons. One possible candidate for these polygons are the Voronoi regions that comprise the cells of the mesh. Recall, however, that these regions are never saved explicitly as geometric information. Only the lengths of the sides of these regions are kept for calculating the diffusion terms of the cells. In fact, it would be quite difficult to use the Voronoi regions to re-tile the polygonal model. The reason for this at least in part is that many of the Voronoi regions are not flat but instead should lie on several of the original polygons. It would be difficult to create a polygon for each Voronoi region in a manner that could even allow all the new polygons to meet seamlessly at the polygon edges. Since many of these regions may have six edges or more it would be difficult to make them planar.

A natural alternative to using the Voronoi regions as polygons is to have the *centers* of the cells become the vertices of the new polygons. This turns out to be much easier to accomplish than using the Voronoi regions, and this is the approach used to create the re-tiled models in this dissertation. One way to accomplish this would be to use the dual graph of the Voronoi tessellation, known as the Delaunay triangulation. This is formed by replacing region centers with vertices and vice versa. This results in a triangular mesh. Unfortunately, we tried this method for several models and found that there were often several Voronoi regions in a model that were particularly troublesome when creating the dual model, resulting in holes or overlapping polygons. The source of these problems is in the nature of the planar approximation used for the Voronoi regions, in the cases where nearby regions do not agree on whether they are adjacent to one another. This problem results from using the planar approximation of the Voronoi regions and it could be corrected by using the exact Voronoi regions. In the work that follows, however, the Delaunay triangulation was dropped in favor of a more robust method of re-tiling that did not pose these kinds of difficulties.

The solution to the problem of robust re-tiling was to create an intermediate form between the original and the re-tiled model that allowed the topology to always remain faithful to that

of the original surface. This re-tiling technique was introduced in [Turk 92]. This method replaces all the old polygons of the model with a collection of triangles, each of whose vertices are mesh points that were positioned by relaxation. The method begins by incorporating the mesh points into the polygonal model to create a new model that has all the original vertices of the model as well as new vertices from the mesh points. Then the old vertices are removed from the model one-by-one, leaving only the new vertices that are also mesh points. The triangles in this new model receive their vertex colors from the chemical concentrations of the reaction-diffusion simulation. Figure 5.5 shows a close-up of an original model of a horse (top) and a re-tiled version of the same model (bottom). The user has control over the number of polygons in the re-tiled model, although in Figure 5.5 the re-tiled model has more polygons than the original model. Figure 5.6 shows a textured version of the re-tiled model using color-interpolated triangles.

5.3.1 Re-Tiling using Constrained Triangulation

The first stage in re-tiling, incorporating the mesh points as vertices of the model, relies on a problem in computational geometry called *constrained triangulation.* Figure 5.7 shows a constrained triangulation problem. The left half of this figure shows a collection of points in the plane and a few edges between some of the points. The problem of constrained triangulation is to connect all the points into a mesh consisting only of triangles while incorporating the already given edges as part of the mesh. The right half of Figure 5.7 shows a triangulation that meets the given constraint edges given at the left.

There are many ways to solve a constrained triangulation problem [Preparata and Shamos 85]. The solution method employed for the models in this dissertation is a commonly used technique called *greedy triangulation.* Consider a set of n points and some constraint edges. This algorithm starts by initializing the final list of edges with all the constraint edges for the given problem. It then considers all other pairs of points of the problem and places these on a candidate list of edges that is ordered based on the distance between the points. The algorithm proceeds by removing the shortest edge on the candidate list and then adding this edge to the final edge list if this edge does not intersect any of the edges already on the final list. The "greedy" aspect of the algorithm is that it always tries to find the shortest edge it can to add to the solution. The algorithm terminates when the candidate edge list is empty. The final step determines what the resulting triangles are from the edges in the final edge list. The greedy triangulation algorithm has the advantages that it is simple to implement and that it gives reasonably shaped triangles. Its drawback is that it has computational complexity $O(n^2)$.

Incorporating the mesh points into the polygonal model is a simple process that relies on constrained triangulation. All of the polygons in the original model are processed one at a time. Each of the vertices of a particular polygon are taken together with the mesh points that lie on that polygon, and this collection of points is considered as a triangulation problem in the plane of the given polygon. The edges of the original polygon serve as constraint edges in this problem. The left half of Figure 5.8 shows a five-sided polygon taken from a

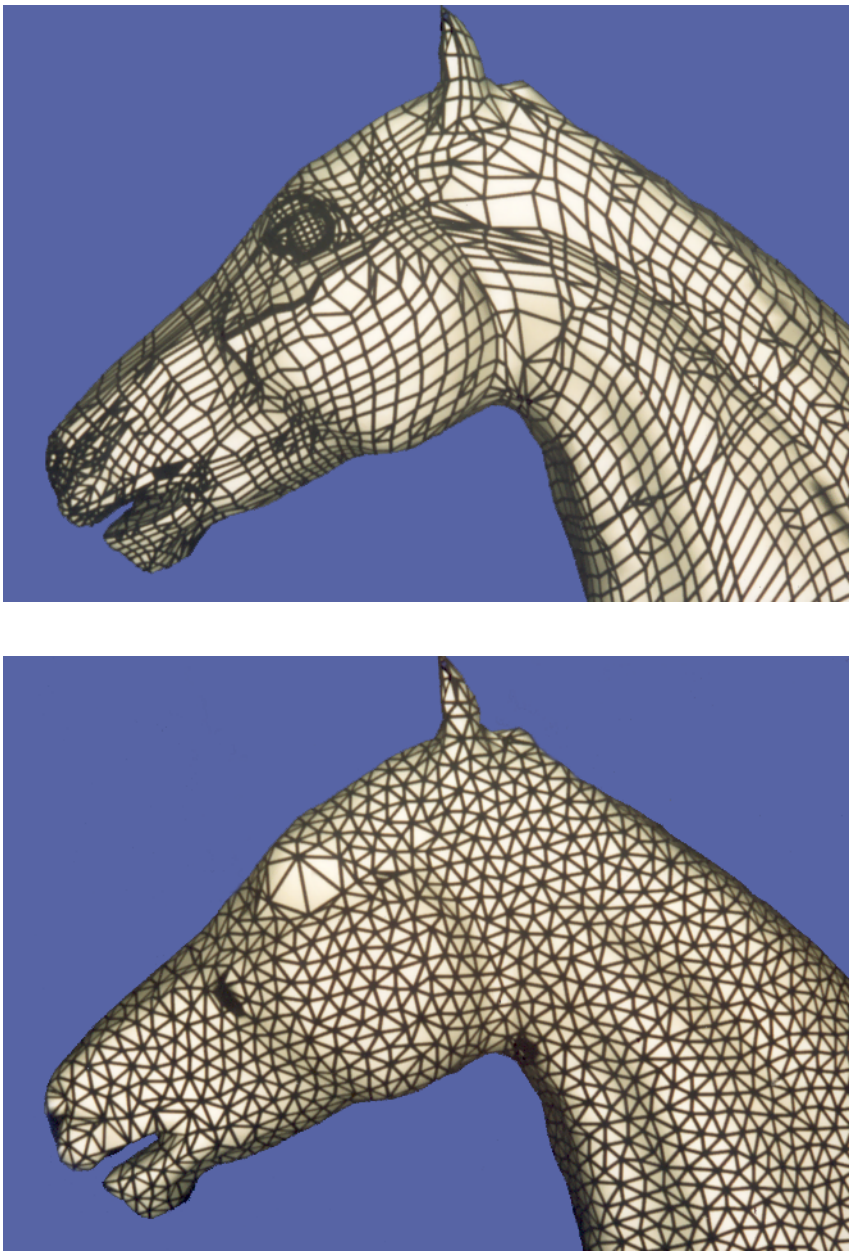

Figure 5.5: Original polygons of horse model (top, 13352 polygons) and a re-tiled version of the same model (bottom, 32137 polygons).

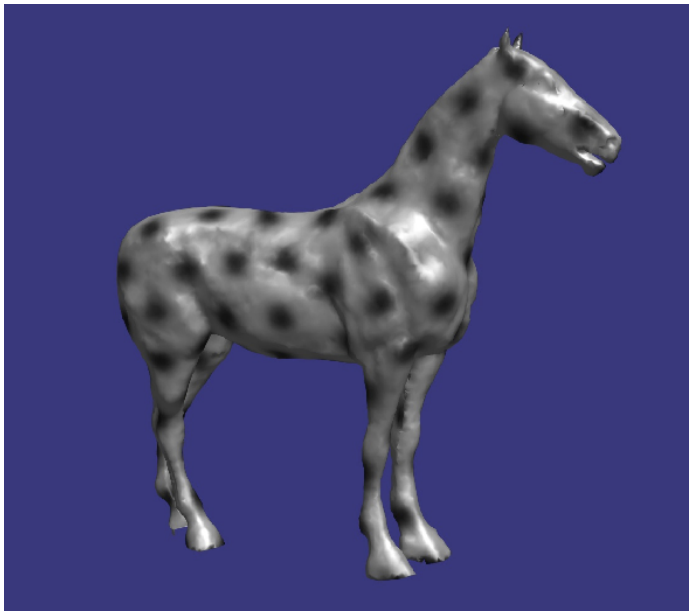

Figure 5.6: Re-tiled horse model, where vertices are colored based on a reaction-diffusion pattern.

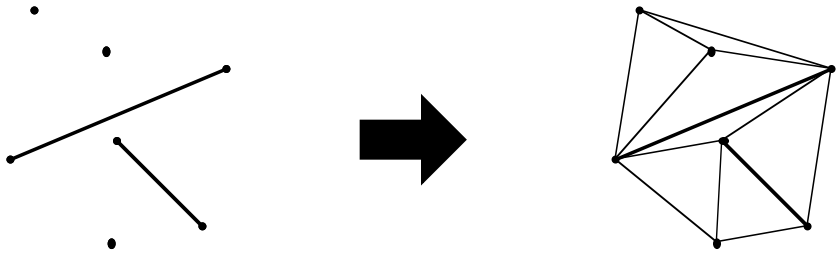

Figure 5.7: Constrained triangulation.

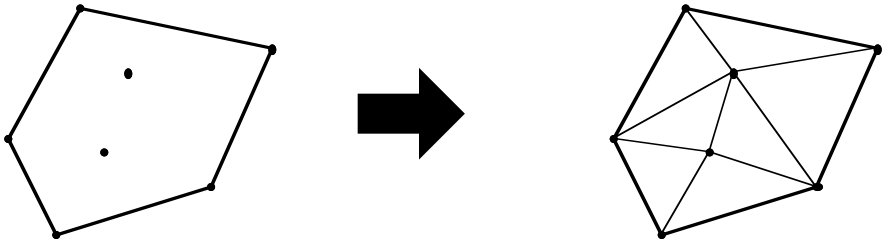

Figure 5.8: Mutual tessellation of a face to incorporate mesh points.

hypothetical polygonal model and the two mesh points that lie on this polygon. This problem is given to the greedy algorithm, and it returns a list of triangles as the solution. The right half of Figure 5.8 shows the constrained triangulation solution for the example on the left. Now, the original polygon is removed entirely from the model and is replaced with the new collection of triangles. It is because we use the edges that bound the original polygon as constraints during triangulation that we know the boundary of the new triangles will be the same as the boundary of the polygon we remove. When this process is carried out for all polygons in the model, the result is a model (entirely composed of triangles) that incorporates all the points from the simulation mesh. This intermediate model in the re-tiling process is called the *mutual tessellation* of the model.

The next step in the re-tiling process is to remove all the old vertices from the original model, leaving only those vertices that were part of the simulation mesh. This can be accomplished by once again invoking constrained triangulation. The idea is to replace the triangles that immediately surround an old vertex *V* with a set of new triangles that do not include *V* and that have the same collective boundary as the original triangles. To remove a particular vertex *V*, we first find a plane that is an approximation of the tangent to the surface at *V*. Any reasonable tangent approximation will do, and for the re-tilings found here we have used the average of the surface normals of the polygons that share the vertex *V*. The next step is to consider all the vertices of the model that form the triangles immediately surrounding *V*. The left portion of Figure 5.9 shows a vertex *V* we wish to remove from a mutually tessellated model and the triangles surrounding *V*. We now project all the vertices of these triangles *except V* onto the tangent plane, and this gives us a new set of points for a constrained triangulation problem. The additional edge constraints are those edges that did not have *V* as an endpoint and that were part of the original set of triangles. These edges form the boundary of the triangles surrounding *V*. The middle of Figure 5.9 shows such a constrained triangulation problem from the model and the vertex *V* shown at the left. Constrained triangulation is now invoked to give a new set of triangles (show at the right in Figure 5.9) with the same boundary as the old set. The old triangles are then removed and replaced by this new set of triangles. This process is repeated for each old vertex in the model until the

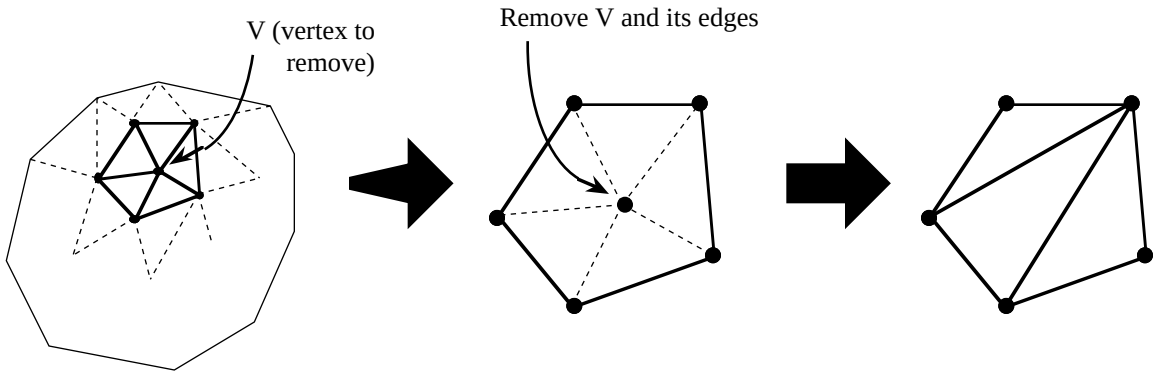

Figure 5.9: Removing an old vertex from a mutual tessellation.

only vertices that are left are those new vertices from the simulation mesh. The model has now been re-tiled.

With this re-tiled model, it is now simple to assign the colors of the vertices based on the concentrations of chemicals given by a reaction-diffusion simulation. The pattern on the horse in Figure 5.6 was created using Turing's spot-formation system.

The re-tiled models shown in this dissertation all have had the original vertices removed from the model. A natural variant of the above re-tiling method would be to incorporate all the mesh points into the model but then retain all the original vertices. There are good and bad aspects to retaining the original vertices of the model. We have no values of chemical concentration at the original vertices of the model. If we retained the original vertices we would require some form of interpolation of chemical concentration at these vertices. On the other hand, when we throw out the original vertices we are throwing out some of the geometry of the model, and this is undesirable. In some instances this may cause sharp corners to be lost. Models that keep the original vertices will have more polygons and will take longer to display, but they will be faithful to the original model's geometry. The ability to choose between retaining and removing the original vertices would be a simple one to provide, and this would allow a user to choose between geometric fidelity and display speed.

5.3.2 Macro-Displacement Mapping

Some patterns found in nature are not simply patterns of color but are also variations in the roughness or bumpiness of a surface. For instance, the warts and bumps on a frog can be

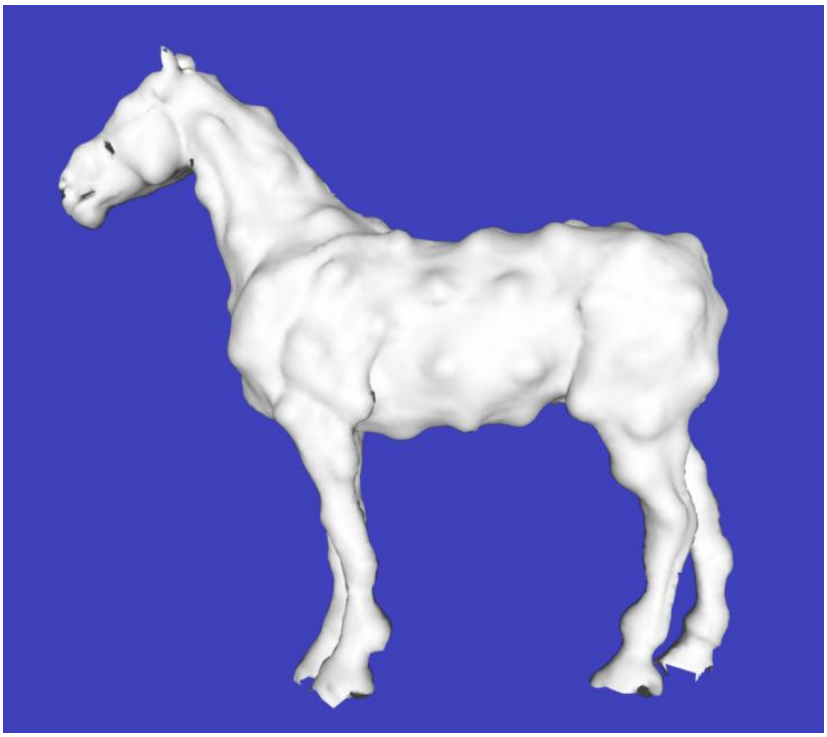

Figure 5.10: Macro-displacement bumpy horse.

considered a pattern. We can create such patterns of geometry based on reaction-diffusion simulations. Using the newly tiled model of an animal from the simulation mesh, we can move the positions of the vertices of the model based on the pattern from a reaction-diffusion simulation. Figure 5.10 shows a bumpy horse created in this manner. The vertices of this model were pushed in a direction tangent to the surface, and the amount of motion of each vertex was proportional to the concentration of chemical *b* from the Turing spot-formation system. We will call this technique *macro-displacement mapping.* This is much the same as the original displacement mapping technique described in [Cook 84].

In true displacement mapping, the positions of micropolygons are moved according to a texture. *Micropolygons* are sub-pixel sized polygons that are created on-the-fly during the rendering process. In contrast, macro-displacement mapping moves the polygons of a model *once* to create a new model. This may be an important disadvantage. With macro-displacement mapping, the individual polygons that are displaced will become noticeable if the object is near enough to the viewer. This is not the case with true displacement mapping because surfaces are split into micropolygons in a manner that is sensitive to the size of the model on the screen. Macro-displacement mapping was used in this dissertation for two reasons. First, no renderer was available that could be easily modified to do true displacement mapping. This is a straightforward task left for future work. Second, the re-tiled models that result from macro-displacement mapping can be rapidly displayed on fast graphics hardware. This is an advantage over true displacement mapping. For instance, the model shown in Figure 5.10 was displayed at more than ten frames per second on Pixel-Planes 5.

5.4 High-Quality Rendering of Reaction-Diffusion Textures

We now turn to a higher quality method of texturing an object using the pattern from a reaction-diffusion simulation. The result of a simulation on a mesh is a set of irregularly distributed measurements of chemical concentration over the surface of a given polygonal model. The texturing method described in this section is based on a method to compute values of chemical concentration at *any* position on the surface from these irregularly spaced values. This is an interpolation problem over an *irregular* mesh of points. Texture interpolation is usually done over *regular* meshes, and the method of irregular mesh interpolation described in this dissertation is a new contribution to computer graphics. The interpolation method described below produces images that minimize visible artifacts due to the simulation mesh. As before, chemical concentration is mapped to color values to create the final texture of the surface. This section begins by describing the sparse interpolation method that generates function values at arbitrary positions on a polygonal model.

5.4.1 Sparse Interpolation on Polygonal Models

The problem at hand is to take discrete values of a function and to construct a continuous function from these values. The numerical analysis literature is filled with ways of interpolating between regularly spaced function values along a line or on a grid [Press 88]. Within the computer graphics field, the most commonly used method of interpolation is low-order polynomial fitting to regular grids of data. Linear, quadratic and cubic interpolation

methods are all commonly used for reconstruction in texturing, image warping and volume rendering. It is far less common in graphics to reconstruct a continuous function from irregularly spaced samples. One place where this problem does arise is in ray tracing. Sometimes more rays are cast at pixels in places where the scene is more complex. These additional rays may be evenly spaced, but more often their positions are randomly perturbed to combat aliasing [Cook 86]. This is a very similar problem to that of interpolation of mesh values over a polygonal model. The solution usually adopted for ray tracing is the *weighted-average filter*. This is the method we will use for mesh interpolation. This and other methods are evaluated in [Mitchell 87].

Let us examine the weighted-average filter in one dimension. We begin with a list of n positions $P_1, P_2, ..., P_n$ and the function values $F_1, F_2, ..., F_n$ at each of these positions. The idea behind the interpolation method is to allow nearby values to have a strong effect on the value at a given position Q, and for that effect to fall off with distance from a sample position. We will call this fall-off effect w: $\mathbf{R} \rightarrow \mathbf{R}$, our weighting function from the reals into the reals. This weighting function should have a maximum value at zero, fall off monotonically in the positive and negative directions, and be symmetric around the origin. Figure 5.11 is a one-dimensional illustration showing that the value at a position Q is found by weighting the values of two nearby sample points. The equation for the weighted-average filter is:

$$v(Q) = \frac{\sum_{i=1}^{n} F_i w(|P_i - Q| / s)}{\sum_{i=1}^{n} w(|P_i - Q| / s)}$$

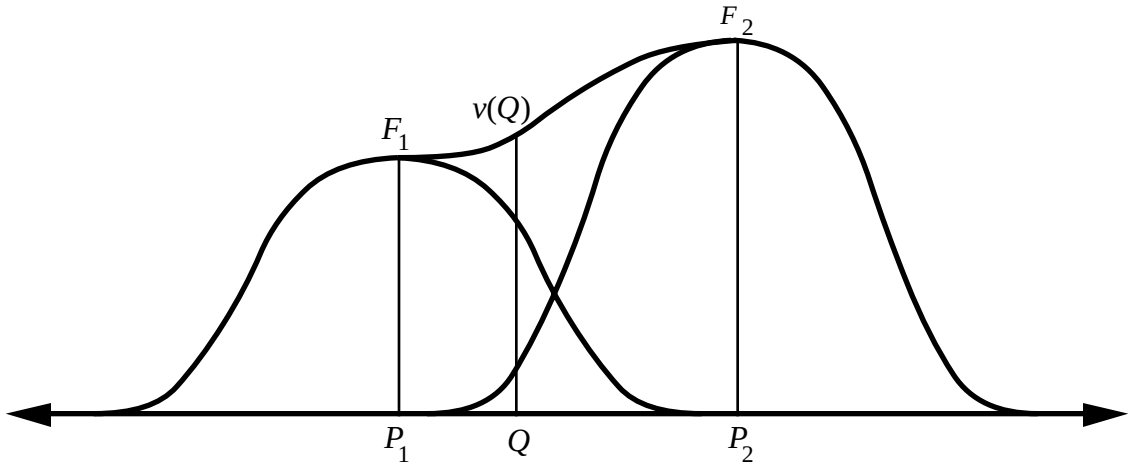

Figure 5.11: Weighted-average at Q of two nearby sample points.

Each nearby function value is weighted according to its distance to the position Q. The scalar quantity s stretches or contracts the range of action of the weighting function w. This method of sparse interpolation works well if the discrete function values are not spaced too irregularly. Fortunately, our mesh points are fairly regularly spaced due to the relaxation step during mesh generation. The weighted-average filter generalizes quite naturally to our problem.

Adapting the above equation to color values in three dimensions is straightforward. Allow the function values $F_1, F_2, ..., F_n$ to be color values $F_i = (\text{red}_i, \text{green}_i, \text{blue}_i)$ and let the positions $P_1, P_2, ..., P_n$ be positions in three-space $P_i = (x_i, y_i, z_i)$. Then the interpolated color $v(Q)$ of a position $Q = (x, y, z)$ can be evaluated directly from the above equation of the weighted-average filter. All that is left to do is pick a suitable weighting function w. The two aspects guiding our choice of w are smoothness of the function and speed of evaluation. A good compromise between these two is given by the following function:

$$w(d) = 2d^3 - 3d^2 + 1 \quad \text{if } 0 \le d \le 1$$
$$w(d) = 0 \quad \text{if } d > 1$$

This function falls smoothly from the value 1 down to 0 in the domain [0,1], and its first derivative is zero at both 0 and 1. This function is the only cubic polynomial with these properties, and it can be found in several places in the graphics literature, e.g. [Perlin and Hoffert 89]. The images in this dissertation have been made using a value of $s = 2r$, where r is the radius of repulsion from the relaxation step of mesh generation. An important property

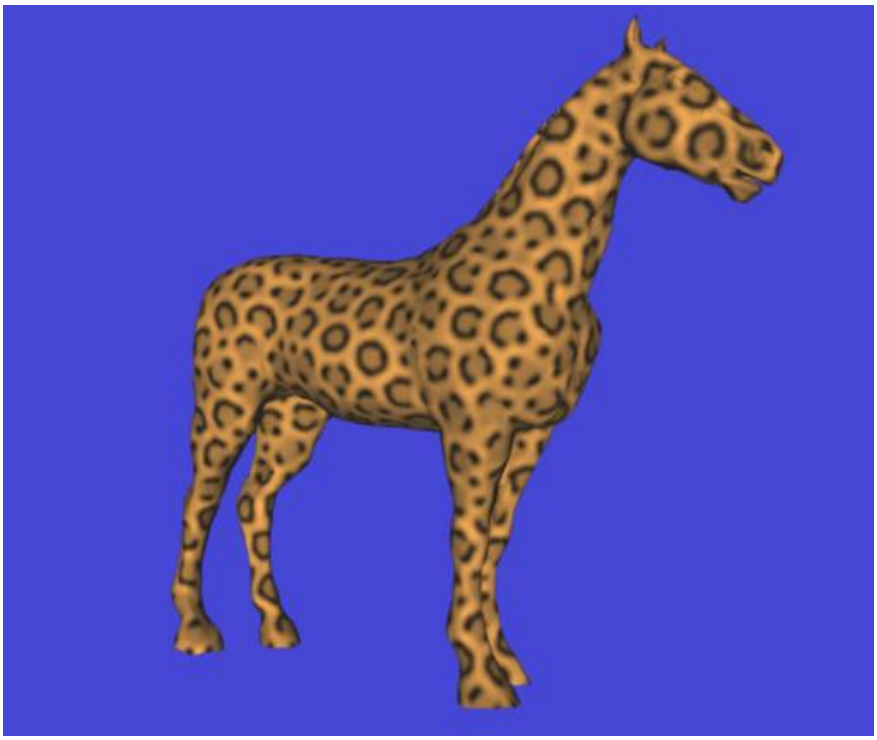

Figure 5.12: Leopard-horse, rendered using the weighted-average of nearby mesh values.

of the weighting function is that it is identically zero for distances greater than 1. This means that mesh points that are further away than this distance do not contribute at all to the value at a given position on the surface. We can use uniform spatial subdivision to rapidly determine all the mesh points that fall within the radius of effect of the position. This reduces the interpolation calculation at a particular location from an $O(n)$ algorithm (examine all mesh points in model) to $O(1)$ since the number of nearby mesh points is a small finite value.

The leopard-horse in Figure 5.12 shows the result of this method of color interpolation. The giraffe of Figure 4.15 shows another example of this texture rendering method. The patterns on these models were created by two of the cascade systems described in Chapter 3.

The interpolation method described above allows a textured object to be as close to the viewer as desired without displaying artifacts of the simulation mesh. The method as described does not, however, address the problem of an object that covers very few pixels. We will examine this issue now.

5.4.2 Diffusion of Colors for Pre-Filtering

As we look at texturing using mesh values, it will be helpful to refer to Figure 5.2, which shows the four processes of texture filtering: reconstruction, warping, pre-filtering, and resampling. As with image-based textures, the warping and resampling processes are dictated by the scene composition and the screen size. The weighted-average filter described above takes care of the reconstruction process. Now we will consider the issue of pre-filtering using mesh values. The solution presented here is taken from the notion of an image pyramid such as a mip map [Williams 83], and this idea is extended to our irregular texture meshes.

We cannot display information that has a higher frequency than our display device, so we can use low-pass filtering to avoid aliasing. The idea of a mip map is to pre-compute different versions of a texture at different amounts of low-pass filtering. We can extend the concept from mip maps to reaction-diffusion textures by creating low-pass filtered (blurred) versions of the texture meshes. The texture meshes are nothing more than color values F_i at positions P_i. We will call level 0 the original set of color values, that is, an un-blurred version of the texture. Level 1 will be a slightly blurred version of this original texture, and levels 2 and higher will carry increasingly blurred color values. What we need now is a method of low-pass filtering those color values in a controlled fashion. It is our good fortune that *diffusion* can provide us with a solution.

Let us look at diffusion on a two-dimensional grid in the plane. When the values of a gray-scale image are allowed to diffuse across the image, the result is exactly the same as if the image had been convolved with a Gaussian filter. Larger amounts of blurring (wider Gaussian filters) are obtained by diffusing for longer periods of time. The relationship between diffusion for a time t and convolution with a Gaussian kernel of standard deviation s is $t = s^2$ [Koenderink 84]. This gives us a method of producing our filtered levels 1, 2, and higher. Texture level k+1 is created by allowing the colors of texture level k to diffuse for the appropriate amount of time. Figure 5.13 shows three low-pass filtered versions of a spot

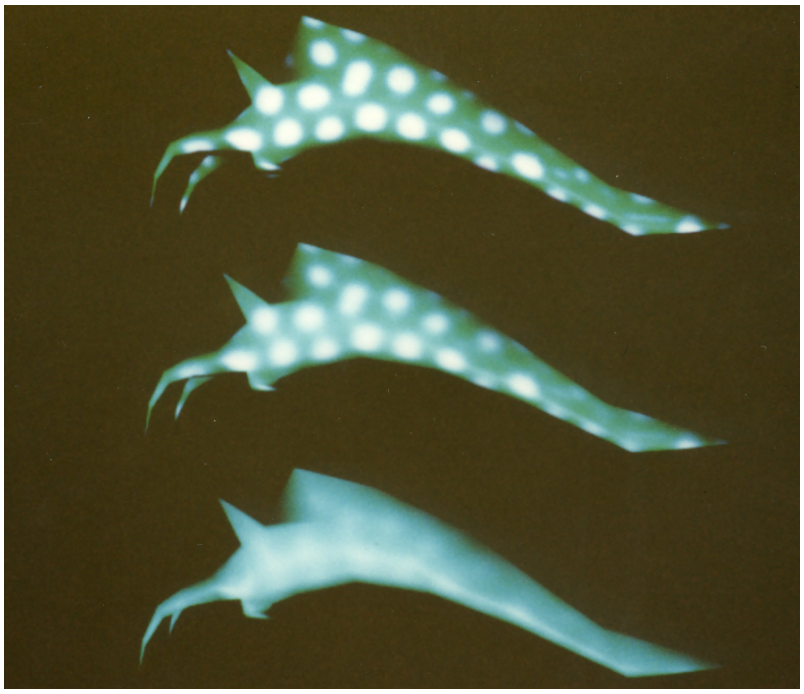

Figure 5.13: Increasingly blurred versions of a texture on a sea slug.

texture on a sea slug. Each level's Gaussian kernel has a standard deviation that is twice that of the next lower level.

Now that we have several band-limited versions of the texture's color, we can select between the appropriate levels based on the size of the textured object on the screen. The texture color at a point will be a weighted average between the colors from two band-limited texture levels. We can choose between blur levels and calculate the weighting between the levels at a pixel from an approximation to the amount of textured surface that is covered by the pixel. This estimate of surface area can be computed from the distance to the surface and the angle the surface normal makes with the direction to the eye. The natural unit of length for this area is r, the repulsion radius for mesh building. Let a be the area of the surface covered by a given pixel, and let this area be measured in square units of r. Then the proper blur level L at a pixel is:

$$L = \log_2(\sqrt{a})$$

We have produced short animated sequences using this anti-aliasing technique, and these show no apparent aliasing of the textures. Figure 5.14 shows some stills from such an animation. The texture of the figure is a blue-and-red version of the zebra texture shown in Figure 4.10. Notice that as the horse model becomes smaller the colors blend together towards purple. Compare this with the textures produced without using blur levels (Figure 5.15).

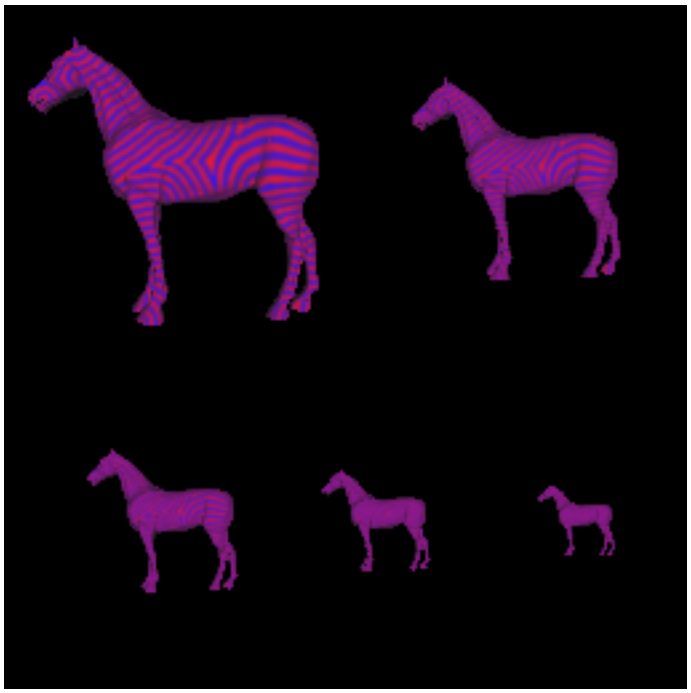

Figure 5.14: Frames from animation that was anti-aliased using multiple band-limited versions of a stripe texture.

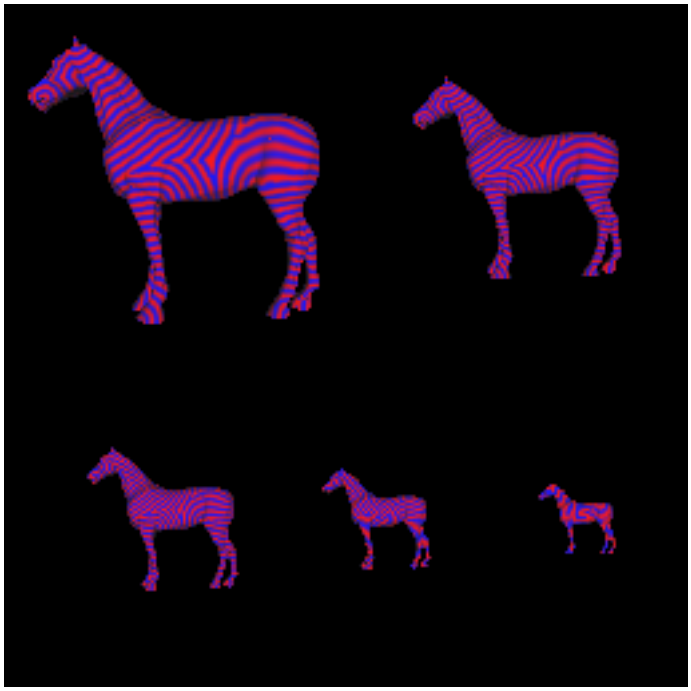

Figure 5.15: Frames from animation with no anti-aliasing.

5.4.3 Filter Quality

Let us evaluate the texture filter that is described above. We will look at the visual quality, the computational cost, and the memory cost of the filter. Examining these issues will suggest possible directions for future research in rendering of reaction-diffusion textures.

As we found earlier, the visual quality of the filter depends on both the reconstruction from discrete samples and the pre-filter. The reconstruction filter used is a radial cubic weighted-average filter. Unfortunately, the irregularity of the positions of the sample points makes it rather difficult to evaluate rigorously the quality of this filter. We can evaluate some of its properties by examining the shape of the filter in the spatial domain. The upper portion of Figure 5.16 shows the shape of the weighting function. The function is continuous and has continuous first and second derivatives. This means that the reconstructed function will be continuous and will show no discontinuities in its first or second derivative. This is in contrast to the mip map implementation described earlier which uses a reconstructed function that has a discontinuous first derivative. We can get more of an idea of the filter's properties by examining it in the frequency domain. The lower portion of Figure 5.16 shows the weighting function's frequency spectrum. Notice that the filter's spectrum does preserve most of the low-frequency components of a signal while attenuating the high-frequency components. This appears to be a reasonable approximation to the ideal resampling filter, the sinc function, which has a spectrum that preserves all the low-frequency signal but removes all high-frequency components.

The pre-filter of our texture filter is based on multiple blurred levels of the texture. Using diffusion, the blurred levels are equivalent to convolution of the original texture with a Gaussian filter. The top portion of Figure 5.17 shows a Gaussian filter. Gaussian filters have several favorable properties. First, the frequency spectrum of a Gaussian is another Gaussian. The bottom portion of Figure 5.17 shows the spectrum of the Gaussian on the right. Notice that this filter preserves much of the low-frequency information while attenuating the high-frequency components. Unfortunately, the Gaussian filter is not perfect since it attenuates frequency components at the high end of the baseband that we would like to retain. Nevertheless, this still compares favorably with the box filter often used in mip maps. The box filter's spectrum is a sinc function (Figure 5.18), which means that there are still significant high-frequency components present in the filtered function. We used a Gaussian filter because it is an acceptable compromise between quality of filtering and convenience of computation.

The second important aspect of the pre-filter is the *support* of the filter, that is, the shape of the region over which the filter averages samples. The process used to make the blur levels is isotropic diffusion, which means that the texture is blurred evenly in all directions. This means that the support of the filter is circular. This is a reasonable support when the textured surface is nearly facing the viewer in a given scene. When the surface is viewed nearly edge-on, however, a circular support blurs the texture more than necessary. Mip maps also blur textures for nearly edge-on orientations. A better support shape for nearly edge-on

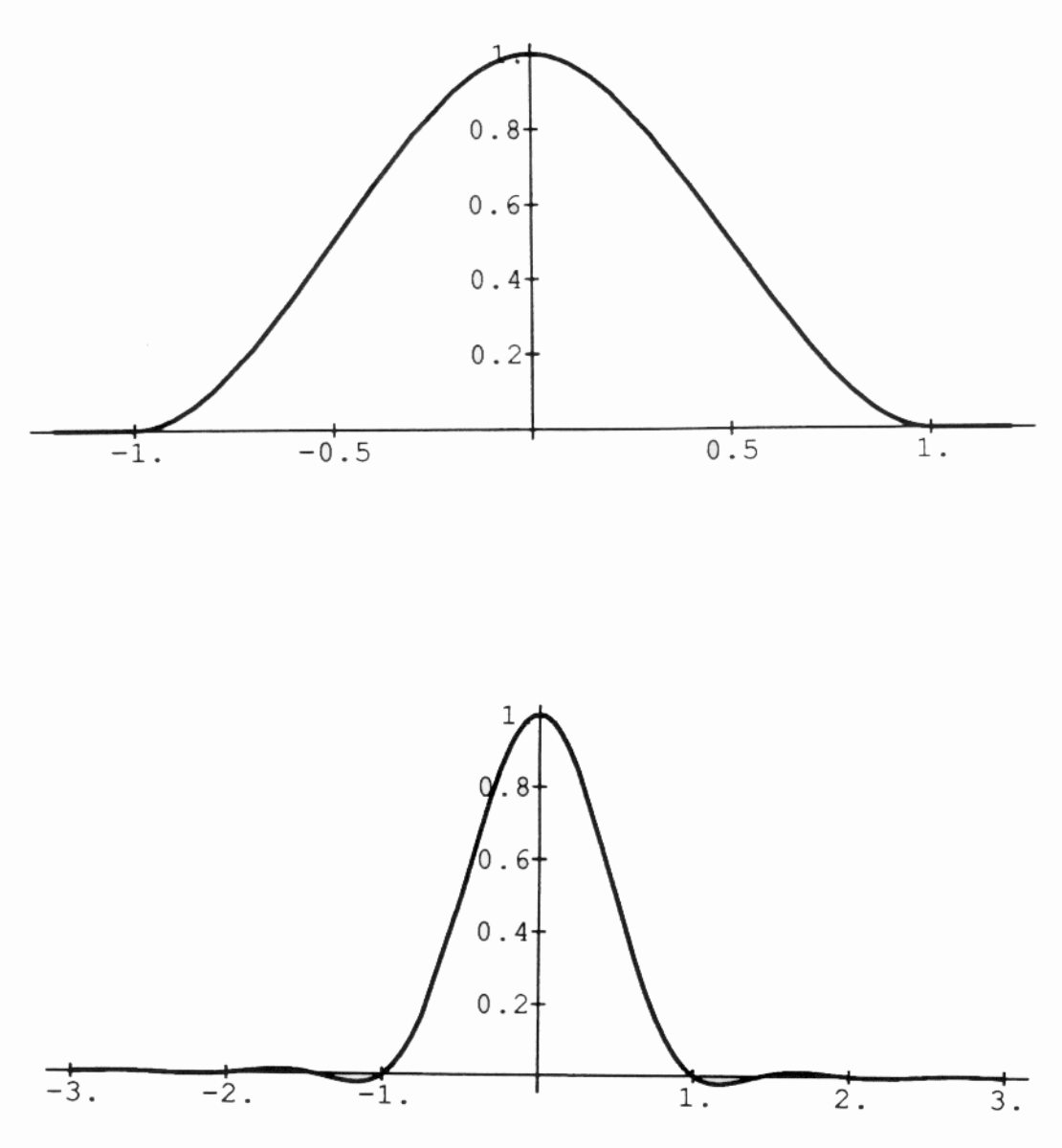

Figure 5.16: Spatial (top) and frequency (bottom) graph of the cubic weighting function *w*.

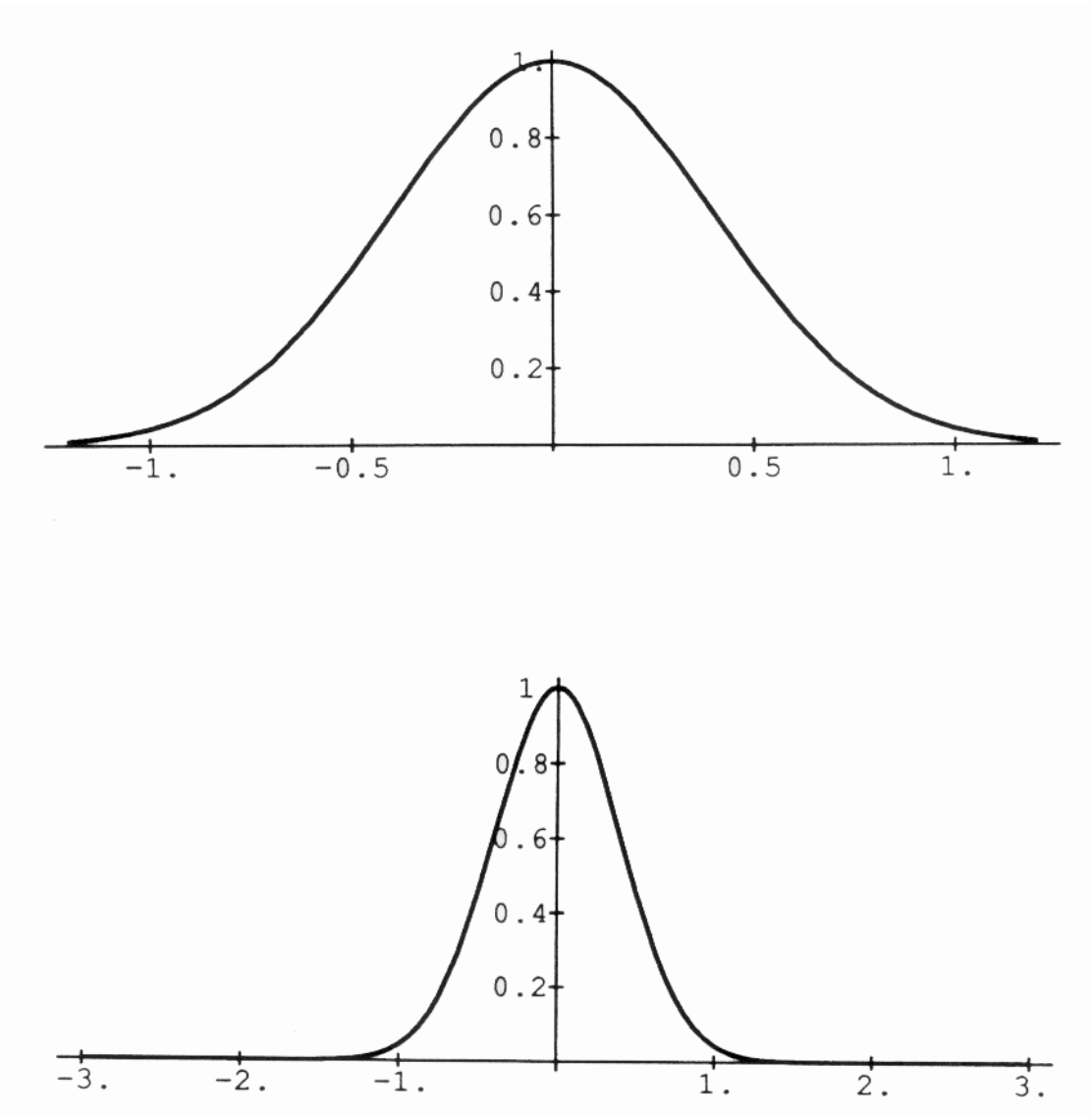

Figure 5.17: Spatial (top) and frequency (bottom) graph of a Gaussian filter.

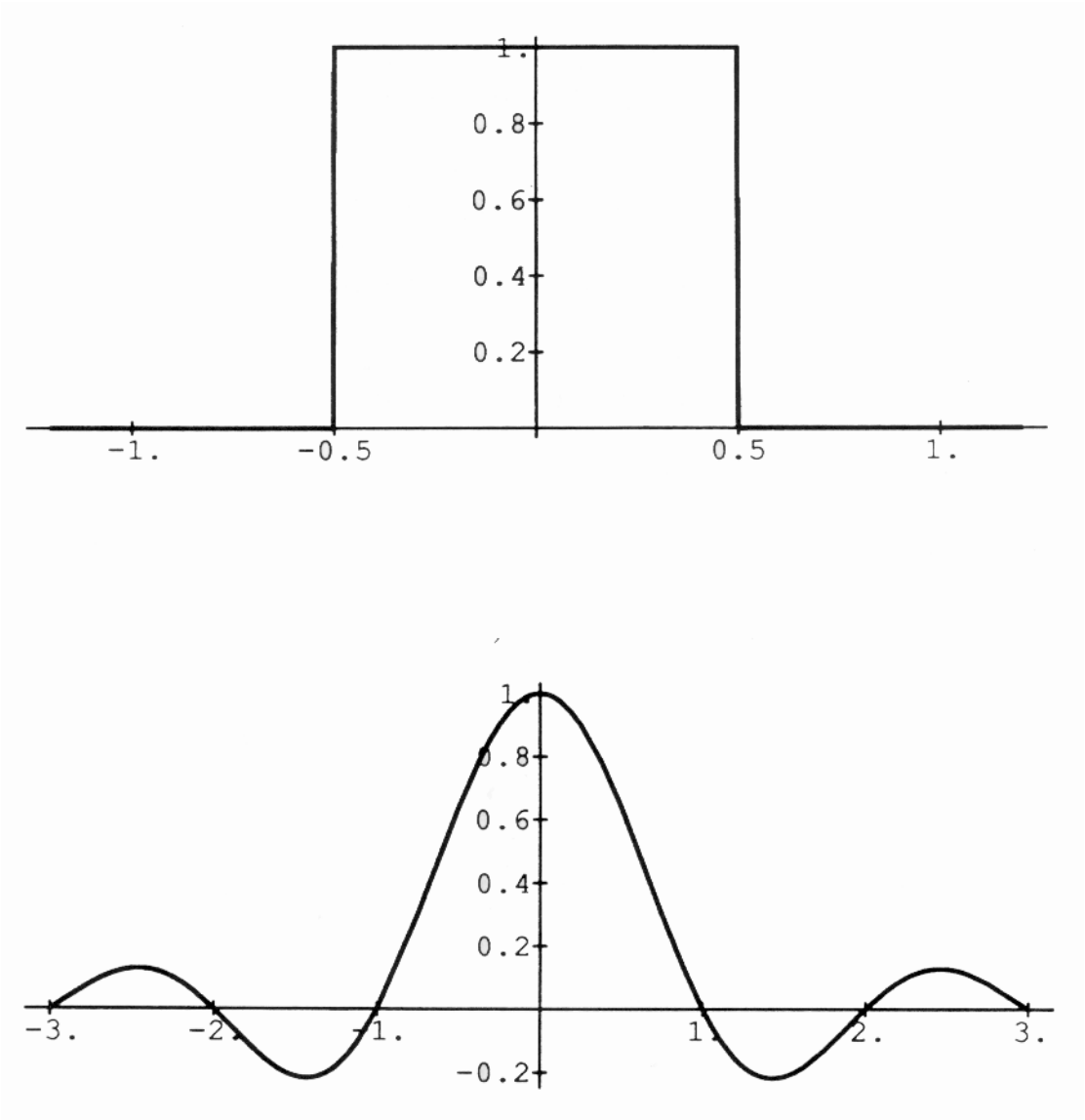

Figure 5.18: Spatial (top) and frequency (bottom) graph of a box filter.

orientations is an ellipse. An elliptical support would average together more texture samples along the direction of the surface that is more steeply angled with respect to the viewer. A high-quality filter with this property called the Elliptical Weighted Average filter (EWA) is described in [Greene and Heckbert 86] and [Heckbert 89]. The EWA filter averages together an arbitrary number of samples within an elliptical region in the texture space, depending on the orientation of the surface being textured. A similar scheme could be implemented for reaction-diffusion texture meshes, although this would have a much higher computational cost. This higher cost would be due to having to filter the texture in many orientations and at several different ellipse eccentricities.

It is fortunate that images of scenes containing reaction-diffusion textures can be produced without a dramatic cost in memory or computation time. As was mentioned earlier, the amount of computation needed for a single evaluation of the texture filter is fixed. The breakdown of costs is as follows:

1) Roughly a dozen color lookups in a spatial data structure for each of two levels.
2) The same number of evaluations of the weighted-average function *w*.
3) Linear interpolation between the two levels.

The cost of (3) is negligible compared to (1) and (2). One lookup in the hash table for the uniform spatial subdivision requires two integer multiplies and a modulo evaluation. Evaluation of *w* requires four multiplies and two add/subtracts with one more multiply and two more adds to keep the running sums for the weighted average. This gives roughly 120 multiplies and 96 add/subtracts per texture evaluation. Several of the multiplies can be eliminated by pre-computing and storing values of the weighted-average function in a table. This amount of computation is the same magnitude as is needed to compute Gardner's sum-of-sines texture [Gardner 84] or a fractal texture based on Perlin's solid noise function [Perlin 85]. As an example of rendering times, the leopard-horse shown in Figure 5.12 took 81 seconds to compute at 512×512 resolution on a DECstation 5000-125. This is a workstation with 32 megabytes of memory and a 25Mhz clock for the CPU/FPU (R3000A/R3010) which has roughly 26 MIPS and 3 MFLOPS (double precision) performance. The same scene without texture required 15 seconds to render. The renderer used was a z-buffer polygon renderer, and the shader in both cases used Phong interpolation of normal vectors and used the Phong approximation for specular highlights [Phong 75].

The memory cost for a reaction-diffusion texture is, of course, dependent on the number of cells in the simulation mesh. The mesh for Figure 5.12 had 64,000 cells in the mesh. Both the positions $P_1, P_2, ..., P_n$ and the colors $F_1, F_2, ..., F_n$ need to be stored for each cell, as well as a pointer for the hash table. When four different blur levels are stored (a typical number of levels), this gives a cost of about 30 bytes per sample. This comes out to roughly two megabytes for the single reaction-diffusion texture of Figure 5.12. For comparison, a 1024×1024 texture element mip map requires one megabyte of storage.

The table below gives an overview of some of the properties of the high-quality filter for reaction-diffusion textures. It gives the same information for mip maps as a point of comparison.

	Mip map	Texture Meshes
Reconstruction	Linear filter	Cubic filter
Pre-filtering	Box filter	Gaussian filter
Computational cost	Constant	Constant (using spatial subdivision)
Additional memory	1/3	Number of blur levels (typically four)

5.4.4 Bump Mapping

Bump mapping (described in section 1.3) is a technique used to make a surface appear rough or wrinkled without explicitly altering the surface geometry [Blinn 78]. The rough appearance is created from a gray-scale texture by adding a perturbation vector to the surface normal and then evaluating the lighting model based on this new surface normal. Perlin showed that the gradient of a scalar-valued function in $\mathbf{R}^3$ can be used as a perturbation vector to produce convincing surface roughness [Perlin 85]. We can use the gradient of the values $v(P)$ of a reaction-diffusion simulation to give a perturbation vector at a point P:

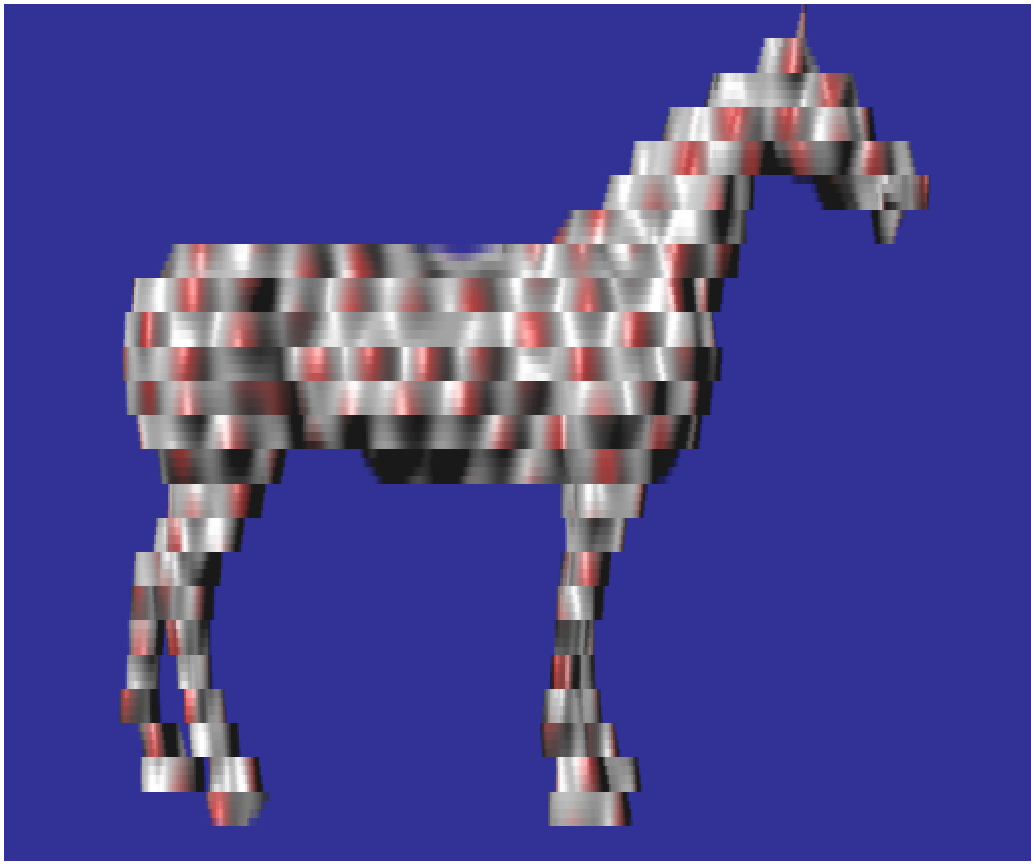

Figure 5.19: Bumpy horse created by perturbing the surface normals based on a reaction-diffusion pattern.

$d = r\,\varepsilon$
$g_x = (v(P) - v(P + [d,0,0])) / d$
$g_y = (v(P) - v(P + [0,d,0])) / d$
$g_z = (v(P) - v(P + [0,0,d])) / d$
perturbation vector = $[k\,g_x, k\,g_y, k\,g_z]$

The above method for computing the gradient of v evaluates the function at P and at three nearby points in each of the x, y, and z directions. The value d is taken to be a small fraction ε of the repulsive radius r to make sure we stay close enough to P that we get an accurate estimate for the gradient. The gradient can also be computed directly from the definition of v by calculating exactly the partial derivatives in x, y, and z. The scalar parameter k is used to scale the bump features, and changing the sign of k causes bumps to become indentations and vice versa. Figure 5.19 shows bumps created in this manner based on the results of a reaction-diffusion system.

5.5 Hierarchy of Meshes

Storing information at various levels of detail for the purpose of faster rendering is a common technique in computer graphics. The mip map is one example of this, where different levels in the mip map give different amounts of detail of the texture. With mip maps, a low-detailed level contains one-quarter the number of sample points as the next higher level of detail. In contrast, the reaction-diffusion texture filter described earlier uses the same number of mesh points at each of the different levels of texture meshes. In this section we will discuss building a set of meshes for texturing that contain different numbers of sample points, just as is done with mip maps. If, for example, we create a reaction-diffusion pattern on a mesh of 64,000 cells, we can also build meshes of 16000, 4000 and 1000 cells that are less-detailed versions of the same texture. Such a hierarchy of meshes provides a more efficient way to render a given object. When the object covers a small portion of the screen then one of the low-detail meshes can be used to render the object. When the object covers much of the screen we can use the fully-detailed texture mesh to render the object. Such a mesh hierarchy is more efficient both in rendering time and in the use of memory. The sections below show that a mesh hierarchy can be used to create several re-tiled models for rapid rendering on polygon display hardware. To create such a mesh hierarchy, we will first need to create a "nested" collection of mesh points. We will also determine the way a pattern can be passed down from the most detailed mesh level to the other levels in the hierarchy.

5.5.1 Building Nested Point Sets

We will say that a set of points S on the surface of an object is *nested* within a larger point set L if all the points in S are also present in the larger set L and if the points in L fill in the spaces between the points in S. This is analogous to hierarchies of points on a regular grid as is shown in Figure 5.20. In this figure, the solid grid points are a low-detail version of a grid, and both the solid and hollow grid points taken together give the higher-detail grid. The hollow points cover the area of the plane between the solid grid points. Figure 5.21 shows a nested point set where the mesh points are irregular in spacing. It is possible to create several

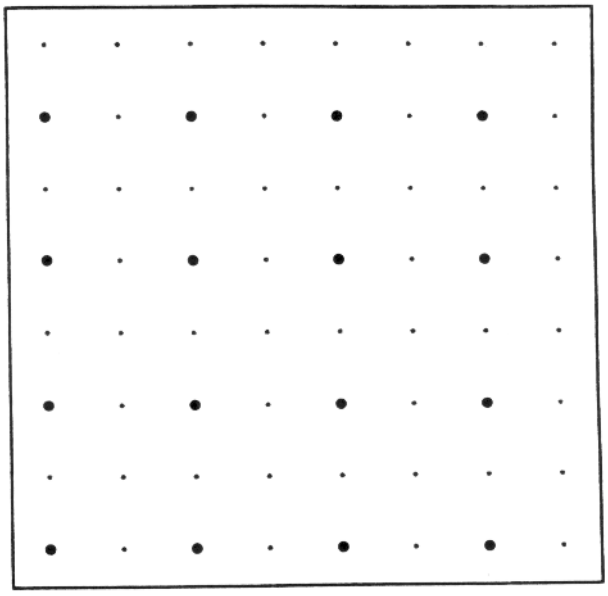

Figure 5.20: A hierarchy of points on a regular grid.

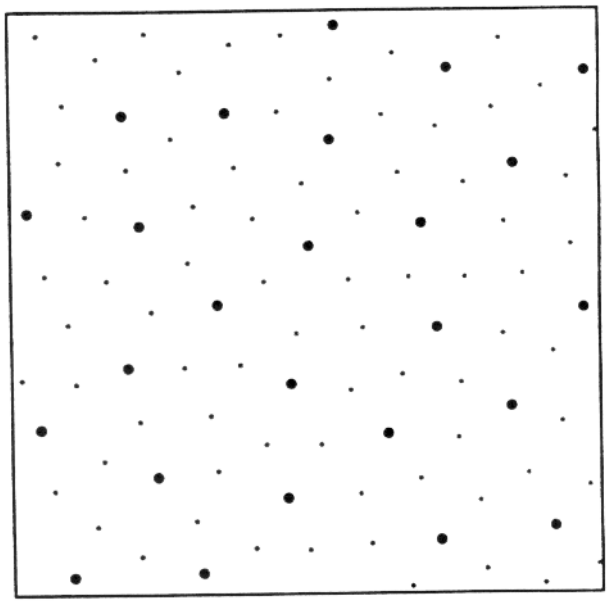

Figure 5.21: An irregular set of nested points in the plane.

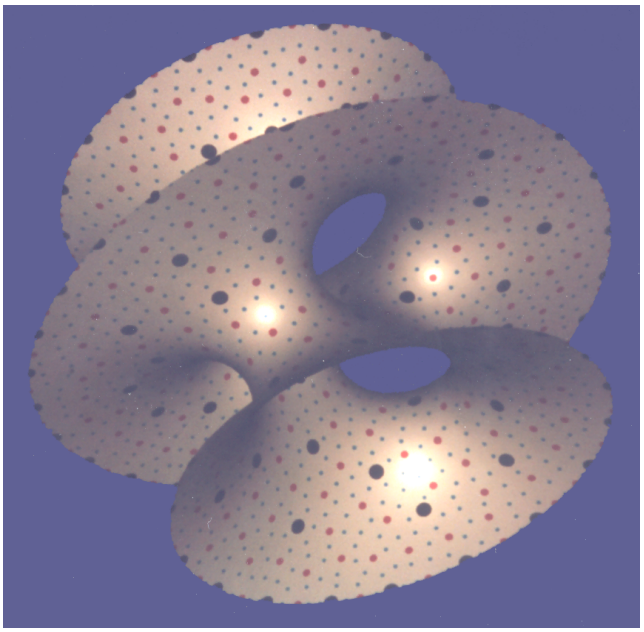

Figure 5.22: Nested points on a minimal surface.

levels of nested points. Let us examine the creation of such a hierarchy of mesh points over a given polygonal model.

Assume we have a polygonal model and we wish to create four meshes over the surface that contain 1000, 4000, 16000, and 64000 points. Further assume that we want all the mesh points in the lower-detailed models to be present in the meshes with more detail. The first step is to position 1,000 points on the original polygonal surface using point-repulsion. The 4,000 vertex mesh can be created by fixing the positions of the first 1,000 points, then placing 3,000 additional points on the object's surface, and finally by allowing these new points to be repelled by one another as well as by the 1,000 fixed points. The next mesh level is made by fixing these 4,000 vertices and adding 12,000 more in the same manner in which we added the previous 3,000. Now we have 1,000 points that have the same position in three meshes and 4,000 points that are present in the same location in two of the meshes. Now 48,000 mesh points can be positioned in the same manner as the previous two sets of mesh points. Figure 5.22 shows the positions of the points from three such levels of detail that were created in the manner just described. For visual clarity, the levels in these meshes contain 200, 800 and 3200 points. The large black spots are the 200 initial points, the red spots are 600 additional points, and the cyan spots are the final 2,400 points. The original object is a portion of a minimal surface of mathematical interest that was modeled using 2,040 vertices. The spots in this figure were rendered by changing the color at a given surface position if it is inside a sphere centered at one of the 3,200 points.

5.5.2 Sharing Patterns Between Levels in Hierarchies

We need to determine a method of sharing patterns between the levels in a hierarchy of meshes. Let us assume that we have created a reaction-diffusion pattern using a 64,000 point

mesh and we want to pass this pattern on to the mesh levels with 16000, 4000 and 1000 points. To do this properly we should filter the pattern to match the amount of frequency information that the different levels can properly represent. That is, we need to band-limit the pattern appropriately for each level. We can do this by using the same Gaussian filtering (by diffusion) that we used to create the different color levels for anti-aliasing. The steps for sharing a pattern between levels are as follows:

1) Create reaction-diffusion pattern on full 64,000 point mesh.
2) Convert chemical concentration to a color pattern on this large mesh.
3) Create different blurred versions of the pattern on the large mesh by diffusion.
4) Obtain the appropriately band-limited colors for the less detailed meshes (16,000 points and fewer) from the correctly blurred patterns.

The idea behind this process is to pass the color information from the mesh points that only appear in the high-detail meshes to the points that are in the lower-detailed meshes. Diffusion over the high-detail mesh is the carrier of this information.

5.5.3 Hierarchies of Polygonal Models

Often for interactive graphics applications it is desirable to have several versions of a model, each containing different amounts of detail. With such levels of detail we can choose from among these versions to trade between fast rendering and high image quality. Earlier in this chapter we saw that surface re-tiling can be used to build a polygonal model that incorporates a pattern created by simulation. Since we can share pattern information in a hierarchy of meshes, it is straightforward to create several polygonal models that capture these patterns at different levels of detail. This gives us a choice of the number of polygons to use to display the given object. A lower-detailed version of the model can be used to speed up rendering when a textured object is far from the eyepoint in a given scene. The method described here to create a hierarchy of polygonal models is based on the work described in [Turk 92].

We can start building a hierarchy of polygonal models by first creating a re-tiled model of the object using the full set of points in our nested point set. This re-tiled model will have as its vertices all of the mesh points of the full simulation mesh, and will capture all the detail of the reaction-diffusion pattern. From the nested point set example above, this re-tiled model would have 64,000 vertices. Then a 16,000 vertex model can be made by removing the 48,000 vertices in the original model that are not present in the more simple level of detail. These vertices can be removed using constrained triangulation, just as was used earlier to create re-tiled models. The vertices of this 16,000 vertex model should be colored according to the blurred pattern that was described in the previous section. Simplified polygonal models with 4,000 and 1,000 vertices can then be made by removing more un-shared vertices from this 16,000 vertex model. Figure 5.23 shows four such polygonal models for a minimal surface that has a striped texture.

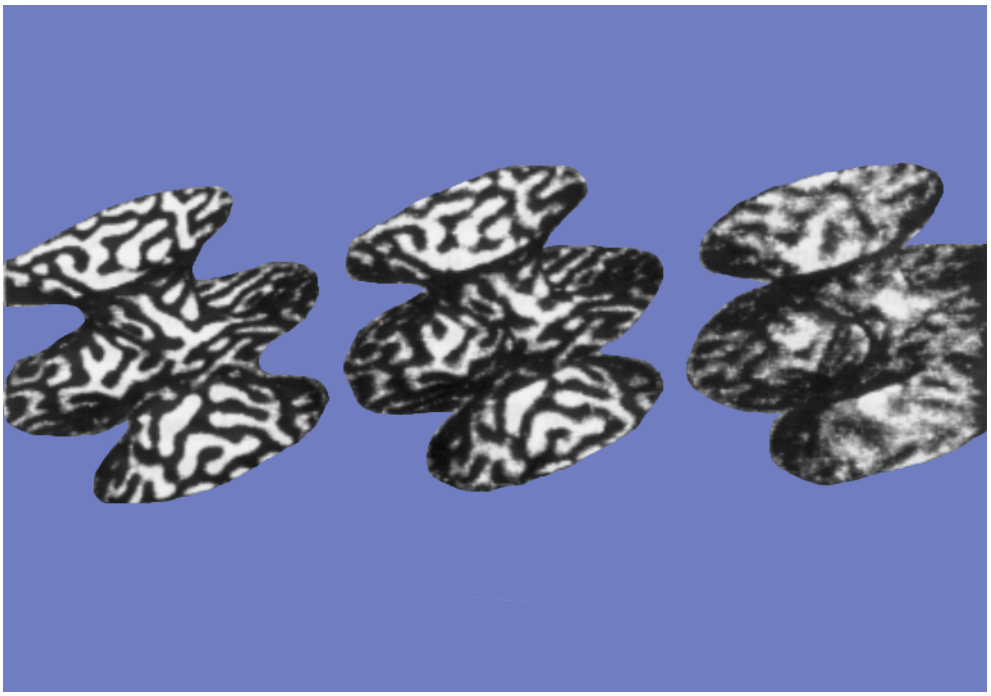

Figure 5.23: Three re-tiled versions of a minimal surface with texturing. From left to right the models contain 16000, 4000, and 1000 vertices.

6 Conclusion

This dissertation has demonstrated that a wide range of textures can be generated by simulation of reaction-diffusion systems. These kinds of textures expand the array of patterns that have been generated by procedural textures. The two sections below list the specific contributions of this dissertation and outline directions for future work in this area.

6.1 Summary of Contributions

This dissertation demonstrates that reaction-diffusion textures can be integrated into the three stages of texturing: acquisition, mapping, and rendering. Here is a list of the contributions:

• Introduced reaction-diffusion patterns to the field of computer graphics.

• Demonstrated that cascades of reaction-diffusion systems can produce a variety of two-dimensional patterns. Several of these are patterns found in nature and have not previously been created by any other model of pattern formation: stripes-within-stripes of the lionfish, the mixed size spots on a cheetah, the rosettes of jaguars and leopards, and the stripe-and-spot pattern on the thirteen-lined ground squirrel.

• Presented a procedure by which a reaction-diffusion system can be simulated over an arbitrary polygonal model. This is done by creating a simulation mesh that is tailor-made for the given model. This mesh is made by generating the Voronoi regions of a set of points that have been evenly distributed over the surface. Simulating a reaction-diffusion system over such a mesh results in textures that match the geometry of a model.

• Demonstrated two methods of rendering reaction-diffusion textures created on a simulation mesh. One of these methods re-tiles the surface into triangles and incorporates a pattern by specifying the colors at the vertices of this mesh. This results in fast rendering of reasonable quality. A higher-quality method of rendering retains the original polygonal model and uses a weighted average of mesh values to color each textured pixel.

• Presented a new method of re-tiling a polygonal model given a new set of vertices. This technique was used to render reaction-diffusion textures, but it has applications wider than that of texturing. The basic technique is useful for the automatic simplification of polygonal models [Turk 92].

6.2 Future Work

A grand challenge of texture synthesis is the ability to create an arbitrary amount of texture to fit a particular model upon being given a texture sample or a texture description. For instance, a user might present a scanned image of tree bark to a hypothetical texturing system and request the creation of more bark for a tree model based on this sample. Such a system is still a distant goal and should provide fuel for future research on synthetic texturing. Any attempt to solve this problem must address the aspects of a texture that are important to the human visual system. A step towards this was taken by Coggins and Jain, who recognized that texture in the perceptual domain is the local average energy across scales [Coggins and Jain 85]. A logical step towards bringing together reaction-diffusion textures and perceptual issues is to examine reaction-diffusion patterns in the frequency domain. Perhaps a bridge could be formed between the addition of texture detail in cascaded systems and the features detected by the human visual system at different scales.

There are still a number of areas related to reaction-diffusion textures that can be explored. One area of research is to increase the variety of patterns that can be generated using reaction-diffusion. New patterns could be created by combining anisotropic systems [Witkin and Kass 91] with the cascade systems described here. It is likely that this would yield a rich set of new textures. Exploring the patterns created by other equations is another open area of research. *Cascade* could be enhanced to allow a user to enter new reaction-diffusion equations that would cause code to be compiled and dynamically linked into the program. This would make creating new reaction-diffusion systems easy and would greatly speed their exploration.

This dissertation focuses on two-dimensional reaction-diffusion patterns. It would be useful to see how three-dimensional reaction-diffusion patterns could benefit computer graphics. For instance, reaction-diffusion could be incorporated into a system such as Perlin and Hoffert's hypertextures [Perlin and Hoffert 89]. Hypertextures are a way to extend the function composition idea for creating solid textures to making three dimensional shapes. These shapes are created by applying various shape-changing functions to volume elements. Adding reaction-diffusion to a hypertexture system could add to the variety of available primitive shapes and could also provide new operators for modifying objects that are defined by other means.

Reaction-diffusion is only one of many models of physical systems that might be used to create synthetic textures. The meshes used to simulate reaction-diffusion systems could be used to simulate some of these other physical systems. These other systems include chemotaxis (for snake patterns), diffusion-limited aggregation (for dendritic patterns) and crack propagation (to create dried mud beds or patterns of tree bark). Nature is rich with patterns that are waiting to be explored, understood and recreated.

Appendix A: *Cascade* Language

Cascade is an interactive program for exploring cascades of reaction-diffusion systems. At the core of this program are the routines for simulating three reaction diffusion systems, two for creating spots and one for making stripes. These simulation routines are provided in Appendix B. Sitting above these simulation routines is a small language for specifying initial values of parameters to the reaction-diffusion systems. This appendix describes the language.

Central to a user's interaction with *Cascade* is an interpreter for a language that specifies the initial values of parameters for a reaction-diffusion system. It is this language that allows a user to try new ideas about cascade systems without writing new code and performing program re-compilation. There are three main sources of values for a parameter: 1) a user-specified constant, 2) another parameter in the same system, 3) another parameter in the *previous* system. It is this ability to set one parameter based on a parameter in a previous system that makes it easy to explore new cascade systems. Suppose, as is shown in Figure 3.5, that we are running Cascade with two systems, each having the parameters *a* and *b*. We can set both parameters *a* and *b* of the second system to the value 4 by entering the following in the parameter section:

a: 4.0
b: 4.0

We could just as well specify that *b* gets its value from the parameter *a* of the same system:

a: 4.0
b: [a]

The square brackets denote copying from a parameter in the same system. If we wanted *b* to acquire its values from *b* of the previous system, we would enter:

a: 4.0
b: b

The above specification for *b* causes information from the first system to be passed on to the second reaction-diffusion system, creating a cascade. Notice that the brackets are omitted if a value is being copied from the previous system. *Cascade* also provides a set of arithmetic operators to form more complex expressions. These operators include: addition, subtraction, multiplication, division, minimum, maximum. Also included are boolean operators: and, or,

not, as well as relational operators: greater than, less than, equals, not equals. There is a function that produces random numbers and there is a conditional expression "if" that selects between two values based on the result of a boolean expression.

It is important to understand that these expressions that specify the initial values of each parameter are computed only once for a given simulation. This is why these expressions can be changed easily and evaluated interpretively without sacrificing interactivity. The reaction-diffusion simulation code, however, is hard-coded into the program, and thus is compiled. A typical reaction-diffusion system requires several thousand simulation steps before the system has produced a final pattern, so speed is critical. The simulation routines for the three basic reaction-diffusion systems that are built into *Cascade* are provided in Appendix B. For this particular implementation of *Cascade*, a MasPar MP-1 parallel computer is used to perform the simulations. The MasPar is particularly well suited to such computations. It consists of a 64×64 array of processing elements (PEs), all executing the same code in single-instruction multiple-data (SIMD) fashion. In *Cascade*, this grid of PEs maps one-to-one with the simulation grid of the reaction-diffusion system. Each PE retains a value for each of the parameters in the system. Each of the PEs can rapidly communicate with its eight nearest neighbors, and this communication makes calculating the diffusion terms simple and fast. The MasPar can run 2000 simulation steps for a 64×64 cell Turing spot-forming system in under 1.5 seconds. This speed is quite adequate for interactive exploration of cascade patterns. The same simulation on a 64×64 grid takes 160 seconds on a DECstation 5000-125.

Appendix B: Reaction-Diffusion Equations

This appendix gives code for the three reaction-diffusion systems that were used to create the textures in this thesis. The systems are:

Alan Turing's spot-formation system [Turing 52]:

$$\frac{\partial a}{\partial t} = s(16 - ab) + D_a \nabla^2 a$$
$$\frac{\partial a}{\partial t} = s(ab - b - \beta) + D_b \nabla^2 b$$

Hans Meinhardt's spot-formation system [Meinhardt 82]:

$$\frac{\partial a}{\partial t} = s(ap_1 + \frac{0.01 a_i a^2}{b} + p_3) + D_a \nabla^2 a$$
$$\frac{\partial b}{\partial t} = s(bp_2 + 0.01 a_i a^2) + D_b \nabla^2 b$$

Hans Meinhardt's stripe-formation system [Meinhardt 82]:

$$\frac{\partial g_1}{\partial t} = \frac{c s_2 g_1^2}{r} - \alpha g_1 + D_g \nabla^2 g_1 + \rho_0$$
$$\frac{\partial g_2}{\partial t} = \frac{c s_1 g_2^2}{r} - \alpha g_2 + D_g \nabla^2 g_2 + \rho_0$$
$$\frac{\partial r}{\partial t} = c s_2 g_1^2 + c s_1 g_2^2 - \beta r$$
$$\frac{\partial s_1}{\partial t} = \gamma (g_1 - s_1) + D_s \nabla^2 s_1 + \rho_1$$
$$\frac{\partial s_2}{\partial t} = \gamma (g_2 - s_2) + D_s \nabla^2 s_2 + \rho_1$$

The procedures listed below are written in a variant of C for the Maspar MP-1 that has extensions for SIMD parallel execution. A variable declared as plural is an indication that each of the 4096 PEs (processing elements) are to allocate room to store a value for a given variable name. Expressions that use plural variables look the same as normal C expressions but they operate in a data-parallel fashion across all processing elements. Portions of expressions such as xnetN[1].a are near-neighbor communications, using the Maspar's xnet communication network. This particular expression gets the value of the variable a from the neighbor that is one PE to the north across the grid of PEs.

B.1 Turing Spots

```
/*****************************************************************************
Turing's spot-forming reaction-diffusion equations.

Entry:
  iters - number of iterations to perform
*****************************************************************************/

visible void turing(iters)
  int iters;
{
  int i;
  register plural float a,b;
  register plural float beta,ka;
  register plural float diff_a,diff_b;
  register plural float delta_a,delta_b;
  register plural float laplace_a,laplace_b;
  register plural float speed;
  register plural int frozen;

  a      = load_float ("a");
  b      = load_float ("b");
  diff_a = load_float ("diff_a");
  diff_b = load_float ("diff_b");
  speed  = load_float ("speed");
  beta   = load_float ("beta");
  ka     = load_float ("ka");
  frozen = load_int ("frozen");

  for (i = 0; i < iters; i++) {

    /* diffuse */

    laplace_a = xnetN[1].a + xnetS[1].a + xnetE[1].a + xnetW[1].a - 4 * a;
    laplace_b = xnetN[1].b + xnetS[1].b + xnetE[1].b + xnetW[1].b - 4 * b;
```

```
    /* react */

    delta_a = ka * (16 - a * b) + diff_a * laplace_a;
    delta_b = ka * (a * b - b - beta) + diff_b * laplace_b;

    /* affect change */

    if (!frozen) {
      a += speed * delta_a;
      b += speed * delta_b;
      if (b < 0)
          b = 0;
    }
  }

  store_float (a, "a");
  store_float (b, "b");
}
```

B.2 Meinhardt Spots

```
/*****************************************************************************
Meinhardt's spot-making system that uses autocatalysis with lateral inhibition.

Entry:
  iters - number of iterations to perform
*****************************************************************************/

visible void auto_simulate(iters)
  int iters;
{
  int i;
  register plural float a,b,ai;
  register plural float diff1,diff2;
  register plural float delta_a,delta_b;
  register plural float laplace_a,laplace_b;
  register plural float speed;
  register plural float ka,kb;
  register plural float temp;
  register plural float s;
  register plural int frozen;
  plural float p1,p2,p3;
```

```
  a      = load_float ("a");
  b      = load_float ("b");
  ai     = load_float ("ai");
  diff1  = load_float ("diff1");
  diff2  = load_float ("diff2");
  speed  = load_float ("speed");
  p1     = load_float ("p1");
  p2     = load_float ("p2");
  p3     = load_float ("p3");
  s      = load_float ("s");
  frozen = load_int ("frozen");

  /* initial setup */

  ka = -s * p1 - 4 * diff1;
  kb = -s * p2 - 4 * diff2;

  for (i = 0; i < iters; i++) {

    /* diffuse */

    laplace_a = xnetN[1].a + xnetS[1].a + xnetE[1].a + xnetW[1].a;
    laplace_b = xnetN[1].b + xnetS[1].b + xnetE[1].b + xnetW[1].b;

    /* react */

    temp = 0.01 * ai * a * a;

    delta_a = a * ka + diff1 * laplace_a + s * (temp / b + p3);
    delta_b = b * kb + diff2 * laplace_b + s * temp;

    /* affect change */

    if (!frozen) {
      a += speed * delta_a;
      b += speed * delta_b;
    }
  }


  store_float (a, "a");
  store_float (b, "b");
}
```

B.3 Meinhardt Stripes

```
/*****************************************************************************
Meinhardt's stripe-making reaction-diffusion system.

Entry:
  iters - number of iterations to perform
*****************************************************************************/

visible void stripe_simulate(iters)
  int iters;
{
  int i;
  register plural float a,b,c,d,e,ai;
  register plural float diff1,diff2;
  register plural float delta_a,delta_b,delta_c,delta_d,delta_e;
  register plural float laplace_a,laplace_b,laplace_d,laplace_e;
  register plural float speed;
  register plural float ka,kc,kd;
  register plural float temp1,temp2;
  register plural int frozen;
  plural float p1,p2,p3;

  a      = load_float ("a");
  b      = load_float ("b");
  c      = load_float ("c");
  d      = load_float ("d");
  e      = load_float ("e");
  ai     = load_float ("ai");
  diff1  = load_float ("diff1");
  diff2  = load_float ("diff2");
  speed  = load_float ("speed");
  p1     = load_float ("p1");
  p2     = load_float ("p2");
  p3     = load_float ("p3");
  frozen = load_int ("frozen");

  /* initial setup */

  ka = -p1 - 4 * diff1;
  kc = -p2;
  kd = -p3 - 4 * diff2;
```

```
  for (i = 0; i < iters; i++) {

    /* diffuse */

    laplace_a = xnetN[1].a + xnetS[1].a + xnetE[1].a + xnetW[1].a;
    laplace_b = xnetN[1].b + xnetS[1].b + xnetE[1].b + xnetW[1].b;
    laplace_d = xnetN[1].d + xnetS[1].d + xnetE[1].d + xnetW[1].d;
    laplace_e = xnetN[1].e + xnetS[1].e + xnetE[1].e + xnetW[1].e;

    /* react */

    temp1 = 0.01 * a * a * e * ai;
    temp2 = 0.01 * b * b * d;

    delta_a = a * ka + diff1 * laplace_a + temp1 / c;
    delta_b = b * ka + diff1 * laplace_b + temp2 / c;
    delta_c = c * kc + temp1 + temp2;
    delta_d = d * kd + diff2 * laplace_d + p3 * a;
    delta_e = e * kd + diff2 * laplace_e + p3 * b;

    /* affect change */

    if (!frozen) {
      a += speed * delta_a;
      b += speed * delta_b;
      c += speed * delta_c;
      d += speed * delta_d;
      e += speed * delta_e;
    }
  }

  store_float (a, "a");
  store_float (b, "b");
  store_float (c, "c");
  store_float (d, "d");
  store_float (e, "e");
}
```

Appendix C: Cascaded Systems

This section provides the exact parameter specifications for the cascade textures presented in Section 3.4.2. Parameters in the left column are for the first reaction-diffusion system, and those in the right column are for the second, cascaded system.

Cheetah

```
system_name: turing_spots
iterations: 13000
interval: 4000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 1/200
```

```
system_name: turing_spots
iterations: 3000
interval: 1000
frozen: b < 4
speed: 1.0
a: a
b: b
diff_a: 1/8
diff_b: 1/32
beta: beta
ka: 1/60
```

Lionfish

```
system_name: stripes
iterations: 20000
interval: 4000
frozen: 0
speed: 1.0
p1: 0.04
p2: 0.06
p3: 0.04
a: [p2] / (2 * [p1])
b: [a]
c: 0.02 * [a] * [a] * [a] / [p2]
d: [a]
e: [a]
ai: random(1,0.02)
diff1: 0.04
diff2: 0.06
```

```
system_name: stripes
iterations: 8000
interval: 1000
frozen: b < 0.7
speed: 1.0
p1: 0.04
p2: 0.06
p3: 0.04
a: if([frozen],a,[p2]/(2*[p1]))
b: if([frozen],b,[a])
c: if([frozen],c,0.02*[a]*[a]*[a]/[p2])
d: if([frozen],d,[a])
e: if([frozen],e,[a])
ai: if([frozen],ai,random(1,0.02))
diff1: 0.009
diff2: 0.06
```

Leopard

```
system_name: turing_spots
iterations: 28000
interval: 4000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 0.05 * 1/16
```

```
system_name: turing_spots
iterations: 7000
interval: 1000
frozen: b < 4
speed: 1.0
a: if([frozen],4,a)
b: if([frozen],4,b)
diff_a: 1/8
diff_b: 1/32
beta: beta
ka: 0.2 * 1/16
```

Autocatalytic Spots

```
system_name: autocatalysis
iterations: 20000
interval: 4000
frozen: 0
speed: 1.0
p1: 0.03
p2: 0.04
p3: 0.0
a: [p2] / [p1]
b: 0.01 * [a] * [a] / [p2]
ai: random(1,0.2)
diff1: 0.01
diff2: 0.2
s: 0.2
```

```
system_name: turing_spots
iterations: 0
interval: 1000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 1/80
```

Vary Spots

```
system_name: autocatalysis
iterations: 6000
interval: 2000
frozen: 0
speed: 1.0
p1: 0.03
p2: 0.04
p3: 0.0
a: [p2] / [p1]
b: 0.01 * [a] * [a] / [p2]
ai: random(1,0.2)
diff1: 0.01
diff2: 0.2
s: x + 0.2
```

```
system_name: turing_spots
iterations: 0
interval: 1000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 1/80
```

Eroded Stripes

```
system_name: stripes
iterations: 16000
interval: 4000
frozen: 0
speed: 1.0
p1: 0.04
p2: 0.06
p3: 0.04
a: [p2] / (2 * [p1])
b: [a]
c: 0.02 * [a] * [a] * [a] / [p2]
d: [a]
e: [a]
ai: random(1,0.02)
diff1: 0.04
diff2: 0.06
```

```
system_name: stripes
iterations: 4000
interval: 1000
frozen: 0
speed: 1.0
p1: 0.04
p2: 0.06
p3: 0.04
a: a + 0 * [p2] / (2 * [p1])
b: b + 0 * [a]
c: c + 0 * 0.02 * [a] * [a] * [a] / [p2]
d: d + 0 * [a]
e: e + 0 * [a]
ai: ai
diff1: 0.01
diff2: 0.06
```

Thin Lines

```
system_name: stripes
iterations: 12000
interval: 4000
frozen: 0
speed: 1.0
p1: 0.04
p2: 0.06
p3: 0.04
a: [p2] / (2 * [p1])
b: [a]
c: 0.02 * [a] * [a] * [a] / [p2]
d: [a]
e: [a]
ai: random(1,0.02)
diff1: 0.03
diff2: 0.06
```

```
system_name: turing_spots
iterations: 3000
interval: 1000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: 12 + (a - .7) * 2
ka: 1/80
```

Small Spots

```
system_name: turing_spots
iterations: 20000
interval: 4000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 1/200
```

```
system_name: turing_spots
iterations: 1000
interval: 1000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: a * 2
ka: 1/80
```

Honeycomb

```
system_name: turing_spots
iterations: 20000
interval: 4000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 1/200
```

```
system_name: turing_spots
iterations: 1000
interval: 1000
frozen: 0
speed: 1.0
 a: 4.0
 b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: b * 2.7
ka: 1/80
```

Wide Spots

```
system_name: turing_spots
iterations: 20000
interval: 4000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 1/250
```

```
system_name: stripes
iterations: 2000
interval: 1000
frozen: 0
speed: 1.0
 p1: 0.04
p2: 0.06
p3: 0.04
a: [p2] / (2 * [p1])
b: [a] + 0.04 * b
c: 0.02 * [a] * [a] * [a] / [p2]
d: [a]
e: [a]
ai: random(1,0.02)
diff1: 0.02
diff2: 0.06
```

Plates

```
system_name: turing_spots
iterations: 20000
interval: 4000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 1/250
```

```
system_name: stripes
iterations: 2000
interval: 1000
frozen: 0
speed: 1.0
p1: 0.04
p2: 0.06
p3: 0.04
a: [p2] / (2 * [p1])
b: [a] + 0.04 * b
c: 0.02 * [a] * [a] * [a] / [p2]
d: [a]
e: [a]
ai: random(1,0.02)
diff1: 0.01
diff2: 0.06
```

Broken Lines

```
system_name: turing_spots
iterations: 16000
interval: 4000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 1/250
```

```
system_name: turing_spots
iterations: 32000
interval: 4000
frozen: 0
speed: 0.5
 a: 4.0
b: 4.0
diff_a: 1/8 + 0.02 * b
diff_b: 1/32
beta: random(12,0.1)
ka: 1/60
```

Bumpy Stripes

```
system_name: autocatalysis
iterations: 28000
interval: 4000
frozen: 0
speed: 1.0
p1: 0.03
p2: 0.1
p3: 0.0
a: [p2] / [p1]
b: 0.01 * [a] * [a] / [p2]
ai: random(1,0.1)
diff1: 0.01
diff2: 0.2
s: 0.5
```

```
system_name: turing_spots
iterations: 0
interval: 1000
frozen: 0
speed: 1.0
a: 4.0
b: 4.0
diff_a: 1/8
diff_b: 1/32
beta: random(12,0.1)
ka: 1/80
```

Mixed Spots

```
system_name: stripes
iterations: 12000
interval: 4000
frozen: 0
speed: 1.0
p1: 0.04
p2: 0.06
p3: 0.04
a: [p2] / (2 * [p1])
b: [a]
c: 0.02 * [a] * [a] * [a] / [p2]
d: [a]
e: [a]
ai: random(1,0.02)
diff1: 0.04
diff2: 0.06
```

```
system_name: autocatalysis
iterations: 2000
interval: 1000
frozen: 0
speed: 1.0
p1: 0.03
p2: 0.04
p3: 0.0
a: [p2]/[p1]
b: 0.01 * [a] * [a] / [p2]
ai: random(1,0.2)
diff1: 0.01
diff2: 0.2
s: 4.0 * (1.45 - b) + 0.3
```

Squirrel

```
system_name: stripes
iterations: 3000
interval: 1000
frozen: 0
speed: 1.0
p1: 0.04
p2: 0.06
p3: 0.04
a: [p2] / (2 * [p1])
b: [a]
c: 0.02 * [a] * [a] * [a] / [p2]
d: [a]
e: [a]
ai: random(1,0.003) + (x > 0.9)
diff1: 0.04
diff2: 0.06
```

```
system_name: turing_spots
iterations: 6000
interval: 1000
frozen: b < 0.5
speed: 0.5
a: if([frozen],4.15,a)
b: if([frozen],4.15,b)
diff_a: .25
diff_b: .0625
beta: random(12,0.1)
ka: 0.01875
```

References

[Alberts et al 89] Alberts, Bruce, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts and James D. Watson, *Molecular Biology of the Cell*, Garland Publishing, Inc., 1989, New York.

[Anderson and Nusslein-Volhard 84] Anderson, K. V. and C. Nusslein-Volhard, "Information for the Dorsal-Ventral Pattern of the *Drosophila* Embryo is Stored in Maternal mRNA," *Nature*, Vol. 331, pp. 223–227 (1984).

[Bard 81] Bard, Jonathan B. L., "A Model for Generating Aspects of Zebra and Other Mammalian Coat Patterns," *Journal of Theoretical Biology*, Vol. 93, No. 2, pp. 363–385 (November 1981).

[Bard and Lauder 74] Bard, Jonathan and Ian Lauder, "How Well Does Turing's Theory of Morphogenesis Work?," *Journal of Theoretical Biology*, Vol. 45, No. 2, pp. 501–531 (June 1974).

[Bier and Sloan 86] Bier, Eric A. and Kenneth R. Sloan, Jr., "Two-Part Texture Mapping," *IEEE Computer Graphics and Applications*, Vol. 6, No. 9, pp. 40–53 (September 1986).

[Blinn and Newell 76] Blinn, James F. and Martin E. Newell, "Texture and Reflection in Computer Generated Images," *Communications of the ACM*, Vol. 19, No. 10, pp. 542–547 (October 1976).

[Blinn 77] Blinn, James F., "Models of Light Reflection for Computer Synthesized Pictures," SIGGRAPH '77, pp. 192–198.

[Blinn 78] Blinn, James F., "Simulation of Wrinkled Surfaces," *Computer Graphics*, Vol. 12, No. 3 (SIGGRAPH '78), pp. 286–292 (August 1978).

[Bloomenthal 85] Bloomenthal, Jules, "Modeling the Mighty Maple," *Computer Graphics*, Vol. 19, No. 3 (SIGGRAPH '85), pp. 305–311 (July 1985).

[Borgens 82] Borgens, R. B., "What is the Role of Naturally Produced Electric Current in Vertebrate Regeneration and Healing?" *International Review of Cytology*, Vol. 76, pp. 245–300 (1982).

[Burt 81] Burt, Peter J., "Fast Filter Transformations for Image Processing," *Computer Graphics and Image Processing*, Vol. 16, No. 1, pp. 20–51 (May 1981).

[Castleman 79] Castleman, Kenneth R., *Digital Image Processing*, Prentice-Hall, 1979, Englewood Cliffs, New Jersey.

[Catmull 74] Catmull, Edwin E., "A Subdivision Algorithm for Computer Display of Curved Surfaces," Ph.D. Thesis, Department of Computer Science, University of Utah (December 1974).

[Child 41] Child, Charles Manning, *Patterns and Problems of Development*, University of Chicago Press, 1941, Chicago, Illinois.

[Coggins and Jain 85] Coggins, James M. and Anil K. Jain, "A Spatial Filtering Approach to Texture Analysis," *Pattern Recognition Letters*, Vol. 3, No. 3, pp. 195–203 (May 1985).

[Cook 84] Cook, Robert L., "Shade Trees," *Computer Graphics*, Vol. 18, No. 3 (SIGGRAPH '84), pp. 223–231 (July 1984).

[Cook 86] Cook, Robert L., "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics*, Vol. 5, No. 1, (January 1986).

[Crow 84] Crow, Franklin C., "Summed-Area Tables for Texture Mapping," *Computer Graphics*, Vol. 18, No. 3 (SIGGRAPH '84), pp. 207–212 (July 1984).

[de Reffye 88] de Reffye, Phillippe, Claude Edelin, Jean Francon, Marc Jaeger and Claude Puech, "Plant Models Faithful to Botanical Structure and Development," *Computer Graphics*, Vol. 22, No. 4 (SIGGRAPH '88), pp. 151–158 (August 1988).

[Fournier and Fiume 88] Fournier, Alan and Eugene Fiume, "Constant-Time Filtering with Space-Variant Kernels," *Computer Graphics*, Vol. 22, No. 4 (SIGGRAPH '88), pp. 229–238 (August 1988).

[Fowler et al 92] Fowler, Deborah R., Hans Meinhardt and Przemyslaw Prusinkiewicz, "Modeling Seashells," *Computer Graphics*, Vol. 26, No. 2 (SIGGRAPH '92), pp. 379–387 (July 1992).

[Fuchs et al 89] Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs and Laura Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics*, Vol. 23, No. 3 (SIGGRAPH '89), pp. 79–88 (July 1989).

[Gardner 84] Gardner, Geoffrey Y., "Simulation of Natural Scenes Using Textured Quadric Surfaces," *Computer Graphics*, Vol. 18, No. 3 (SIGGRAPH '84), pp. 11–20 (July 1984).

[Gardner 85] Gardner, Geoffrey Y., "Visual Simulation of Clouds," *Computer Graphics*, Vol. 19, No. 3 (SIGGRAPH '85), pp. 297–303 (July 1985).

[Gilbert 88] Gilbert, Scott F., *Developmental Biology*, Sinauer Associates, Inc., 1988, Sunderland, Massachusetts.

[Glassner 86] Glassner, Andrew, "Adaptive Precision in Texture Mapping," *Computer Graphics*, Vol. 20, No. 4 (SIGGRAPH '86), pp. 297–306 (August 1986).

[Greene and Heckbert 86] Greene, Ned and Paul S. Heckbert, "Creating Raster Omnimax Images from Multiple Perspective Views Using The Elliptical Weighted Average Filter," *IEEE Computer Graphics and Applications*, Vol. 6, No. 6, pp. 21–27 (June 1986).

[Harris 73] Harris, Albert K., "Behavior of Cultured Cells on Substrate of Various Adhesiveness," *Experimental Cell Research*, Vol. 77, pp. 285–297 (1973).

[Harris et al 84] Harris, Albert K., David Stopak and Patricia Warner, "Generation of Spatially Periodic Patterns by a Mechanical Instability: A Mechanical Alternative to the Turing Model," *Journal of Embryology and Experimental Morphology*, Vol. 80, pp. 1–20 (1984).

[Harris 92] Harris, Albert K., personal communication.

[He et al 91] He, Xiao D., Kenneth E. Torrance, Francois X. Sillion and Donald P. Greenberg, "A Comprehensive Physical Model for Light Reflection," *Computer Graphics*, Vol. 25, No. 4 (SIGGRAPH '91), pp. 175–186 (July 1991).

[Heckbert 86] Heckbert, Paul S., "Filtering by Repeated Integration," *Computer Graphics*, Vol. 20, No. 4 (SIGGRAPH '86), pp. 317–321 (August 1986).

[Heckbert 89] Heckbert, Paul S., "Fundamentals of Texture Mapping and Image Warping," M.S. Thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley (June 1989).

[Ho-Le 88] Ho-Le, K., "Finite Element Mesh Generation Methods: A Review and Classification," *Computer Aided Design*, Vol. 20, No. 1, pp. 27–38 (January/February 1988).

[Hubel and Wiesel 79] Hubel, David H. and Torsten N. Wiesel, "Brain Mechanisms of Vision," *Scientific American*, Vol. 241, No. 3, pp. 150–162 (September 1979).

[Hunding 90] Hunding, Axel, Stuart A. Kauffman, and Brian C. Goodwin, "*Drosophila* Segmentation: Supercomputer Simulation of Prepattern Hierarchy," *Journal of Theoretical Biology*, Vol. 145, pp. 369–384 (1990).

[Jaffe and Stern 79] Jaffe, L. F. and C. D. Stern, "Strong Electrical Currents Leave the Primitive Streak of Check Embryos," *Science*, Vol. 206, pp. 569–571 (1979).

[Kauffman et al 78] Kauffman, Stuart A., Ronald M. Shymko and Kenneth Trabert, "Control of Sequential Compartment Formation in *Drosophila*," *Science*, Vol. 199, No. 4326, pp. 259–270 (January 20, 1978).

[Koenderink 84] Koenderink, Jan J., "The Structure of Images," *Biological Cybernetics*, Vol. 50, No. 5, pp. 363–370 (August 1984).

[Lacalli 90] Lacalli, Thurston C., "Modeling the *Drosophila* Pair-Rule Pattern by Reaction-Diffusion: Gap Input and Pattern Control in a 4-Morphogen System," *Journal of Theoretical Biology*, Vol. 144, pp. 171–194 (1990).

[Lengyel and Epstein 91] Lengyel, István and Irving R. Epstein, "Modeling of Turing Structures in the Chlorite–Iodide–Malonic Acid–Starch Reaction System," *Science*, Vol. 251, No. 4994, pp. 650–652 (February 8, 1991).

[Levoy and Whitted 85] Levoy, Marc and Turner Whitted, "The Use of Points as a Display Primitive," Technical Report TR 85-022, University of North Carolina at Chapel Hill (1985).

[Lewis 84] Lewis, John-Peter, "Texture Synthesis for Digital Painting," *Computer Graphics*, Vol. 18, No. 3 (SIGGRAPH '84), pp. 245–252 (July 1984).

[Lewis 89] Lewis, J. P., "Algorithms for Solid Noise Synthesis," *Computer Graphics*, Vol. 23, No. 3 (SIGGRAPH '89), pp. 263–270 (July 1989).

[Lorensen and Cline 87] Lorensen, William E. and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, Vol. 21, No. 3 (SIGGRAPH '87), pp. 163–169 (July 1987).

[Max 81] Max, Nelson, "Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset," *Computer Graphics*, Vol. 15, No. 3 (SIGGRAPH '81), pp. 317–324 (August 1981).

[Meinhardt 82] Meinhardt, Hans, *Models of Biological Pattern Formation*, Academic Press, London, 1982.

[Meinhardt and Klinger 87] Meinhardt, Hans and Martin Klinger, "A Model for Pattern Formation on the Shells of Molluscs," *Journal of Theoretical Biology*, Vol. 126, No. 1, pp. 63–89 (May 1987).

[Melhorn 84] Melhorn, Kurt, *Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, 1984.

[Mitchell 87] Mitchell, Don P., "Generating Antialiased Images at Low Sampling Densities," *Computer Graphics*, Vol. 21, No. 4 (SIGGRAPH '87), pp. 65–72 (July 1987).

[Mitchell and Netravali 88] Mitchell, Don P. and Arun N. Netravali, "Reconstruction filters in Computer Graphics," *Computer Graphics*, Vol. 22, No. 4 (SIGGRAPH '88), pp. 221–228 (August 1988).

[Mount 85] Mount, David M., "Voronoi Diagrams on the Surface of a Polyhedron," Technical Report 1496, University of Maryland (1985).

[Murray and Kirschner 89] Murray, Andrew W. and Marc W. Kirschner, "Dominoes and Clocks: The Union of Two Views of the Cell Cycle," *Science*, Vol. 246, No. 4930, pp. 614–621 (November 3, 1989).

[Murray 81] Murray, J. D., "On Pattern Formation Mechanisms for Lepidopteran Wing Patterns and Mammalian Coat Markings," *Philosophical Transactions of the Royal Society B*, Vol. 295, pp. 473–496.

[Murray and Myerscough 91] Murray, J. D. and M. R. Myerscough, "Pigmentation Pattern Formation on Snakes," *Journal of Theoretical Biology*, Vol. 149, pp. 339–360 (1991).

[O'Neill 66] O'Neill, Barrett, *Elementary Differential Geometry*, Academic Press, 1966, New York.

[Oppenheim and Schafer 75] Oppenheim, A. V. and R. W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975, Englewood Cliffs, New Jersey.

[Oster 88] Oster, George F., "Lateral Inhibition Models of Developmental Processes," *Mathematical Biosciences*, Vol. 90, No. 1 & 2, pp. 265–286 (1988).

[Oster, Murray and Harris 83] Oster, G. F., J. D. Murray and A. K. Harris, "Mechanical Aspects of Mesenchymal Morphogenesis," *Journal of Embryology and Experimental Morphology*, Vol. 78, pp. 83–125 (1983).

[Peachey 85] Peachey, Darwyn R., "Solid Texturing of Complex Surfaces," *Computer Graphics*, Vol. 19, No. 3 (SIGGRAPH '85), pp. 279–286 (July 1985).

[Perlin 85] Perlin, Ken, "An Image Synthesizer," *Computer Graphics*, Vol. 19, No. 3 (SIGGRAPH '85), pp. 287–296 (July 1985).

[Perlin and Hoffert 89] Perlin, Ken and Eric M. Hoffert, "Hypertexture," *Computer Graphics*, Vol. 23, No. 3 (SIGGRAPH '89), pp. 253–262 (July 1989).

[Phong 75] Bui-Tuong, Phong, "Illumination for Computer Generated Pictures," *Communications of the ACM*, Vol. 18, No. 6, pp. 311–317 (June 1975).

[Poole and Steinberg 82] Poole, T. J. and M. S. Steinberg, "Evidence for the Guidance of Pronephric Duct Migration by a Cranio-caudally Traveling Adhesive Gradient," *Developmental Biology*, Vol. 92, pp. 144–158 (1982).

[Preparata and Shamos 85] Preparata, Franco P. and Michael Ian Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985, New York.

[Press 88] Press, William H., Brian P. Flannery, Saul A. Teukolsky and William T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1988.

[Rhoades et al 92] Rhoades, John, Greg Turk, Andrew Bell, Andrei State, Ulrich Neumann and Amitabh Varshney, "Real-Time Procedural Textures," *1992 Symposium on Interactive 3D Graphics*, Cambridge, Massachusetts, March 29 – April 1, 1992.

[Rosavio et al 83] Rosavio, R. A., A. Delouvee, K. M. Yamada, R. Timpl and J.-P. Thiery, "Neural Crest Cell Migration: Requirements for Exogenous Fibronectin and High Cell Density," *Journal of Cell Biology*, Vol. 96, pp. 462–473 (1983).

[Samek 86] Samek, Marcel, Cheryl Slean and Hank Weghorst, "Texture Mapping and Distortion in Digital Graphics," *The Visual Computer*, Vol. 2, No. 5, pp. 313–320 (September 1986).

[Schachter 80] Schachter, Bruce, "Long Crested Wave Models," *Computer Graphics and Image Processing*, Vol. 12, pp. 187–201 (1980).

[Schachter 83] Schachter, Bruce, *Computer Image Generation*, Wiley, 1983, New York.

[Stopak and Harris 82] Stopak, David and Albert K. Harris, "Connective Tissue Morphogenesis by Fibroblast Traction," *Developmental Biology*, Vol. 90, No. 2, pp. 383–398 (April 1982).

[Swindale 80] Swindale, N. V., "A Model for the Formation of Ocular Dominance Stripes," *Proceedings of the Royal Society of London, Series B*, Vol. 208, pp. 243–264 (1980).

[Thompson et al 85] Thompson, Joe F., Z. U. A. Warsi and C. Wayne Mastin, *Numerical Grid Generation*, North Holland, 1985, New York.

[Turing 52] Turing, Alan, "The Chemical Basis of Morphogenesis," *Philosophical Transactions of the Royal Society B*, Vol. 237, pp. 37–72 (August 14, 1952).

[Turk 90] Turk, Greg, "Generating Random Points in Triangles," in *Graphics Gems*, edited by Andrew Glassner, Academic Press, 1990.

[Turk 91] Turk, Greg, "Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion," *Computer Graphics*, Vol. 25, No. 4 (SIGGRAPH '91) pp. 289–298.

[Turk 92] Turk, Greg, “Re-Tiling Polygonal Surfaces,” *Computer Graphics*, Vol. 26. No. 2 (SIGGRAPH ’92), pp. 55–64 (July 1992).

[Weiss 34] Weiss, P., “In Vitro Experiments on the Factors Determining the course of the Outgrowing Verve Fiber,” *Journal of Experimental Zoology*, Vol. 68, pp. 393–448 (1934).

[Westover 90] Westover, Lee, “Footprint Evaluation for Volume Rendering,” *Computer Graphics*, Vol. 24, No. 4 (SIGGRAPH ’90), pp. 367–376 (August 1990).

[Whitted 80] Whitted, Turner, “An Improved Illumination Model for Shaded Display,” *Communications of the ACM*, Vol. 23, No. 6, pp. 343–349 (June 1980).

[Williams 83] Williams, Lance, “Pyramidal Parametrics,” *Computer Graphics*, Vol. 17, No. 3 (SIGGRAPH ’83), pp. 1–11 (July 1983).

[Witkin and Kass 91] Witkin, Andrew and Michael Kass, “Reaction-Diffusion Textures,” *Computer Graphics*, Vol. 25, No. 4 (SIGGRAPH ’91), pp. 299–308 (July 1991).

[Wolpert 71] Wolpert, Lewis, “Positional Information and Pattern Formation,” *Current Topics in Developmental Biology*, Vol. 6, Edited by A. A. Moscona and Alberto Monroy, Academic Press, 1971.

[Yeager and Upson] Yeager, Larry and Craig Upson, “Combining Physical and Visual Simulation — Creation of the Planet Jupiter for the Film 2010,” *Computer Graphics*, Vol. 20, No. 4 (SIGGRAPH ’86), pp. 85–93 (August 1986).

[Young 84] Young, David A., “A Local Activator-Inhibitor Model of Vertebrate Skin Patterns,” *Mathematical Biosciences*, Vol. 72, pp. 51–58 (1984).