BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-95-M9

"A Parallel Adaptive Point-Sampling Algorithm"

by

Mark Marcus

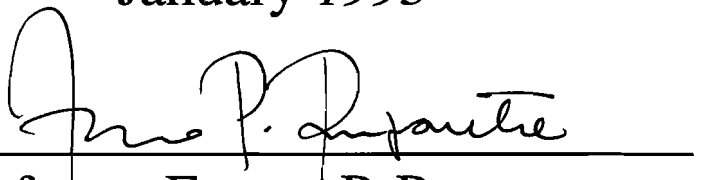
A Parallel Adaptive Point-Sampling Algorithm

Mark Marcus

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University

January 1995



Professor Franco P. Preparata
Advisor

A Parallel Adaptive Point-Sampling Algorithm

Mark Marcus

January 26, 1995

Chapter 1

Introduction

There is an increasing commercial need for computers to quickly generate photorealistic images. Photorealistic computer graphics is the branch of computer graphics concerned with the production of computer synthesized images that look, to the observer, as much like real photographs as possible. Today, computer generated photorealistic images are routinely used in the motion picture industry, in films such as Jurassic Park, Lion King, and Terminator II, in television commercials, and in medical imaging for the visualization of CAT and MRI scans.

The reduction of time required to produce photorealistic images would provide two commercial benefits: (1) lowering the cost of existing uses by providing quicker turn around time and (2) opening up new application areas. For example, a CAD tool could be built to aid architects by allowing them to walk-through a building being designed. Current virtual-reality systems, that would allow this, require the generation of a new image every 1/20th of a second. Alas, today's technology can't produce a photorealistic image in 1/20th of a second.

The production of photorealistic images is computationally intensive. It is not uncommon to take 30 minutes to generate a single photorealistic image. And, some photorealistic images take hours to produce. This process is computationally intensive due to the complexity of visual clues that is required to be included in an image in order for it to look photorealistic. Among those visual clues that make an image look photorealistic and that can be observed in real photographs are: hidden surface elimination, shadows, reflections, refractions, surface details, depth of field, and motion blur.

Surface details are especially important to photorealism. When an observer looks at a surface, such as a brick wall or an orange, the observer not only sees the colors of the surface but also the detail shadows that are caused by pits, bumps, and imperfections. These details and shadows give a sensation of realism to a computer generated image. It also take substantial time to generate this level of detail for an entire scene of objects.

Photorealistic graphics provides the ability to depict an imagined scene. Photorealistic graphic systems are designed specifically to produce images of imagined scenes that are similar to those found in the real world. In addition, the techniques of photorealistic graphics can be applied to scientific visualization that depict non-real world scenes. Examples of scientific visualization are (1) the visualization of viruses for the drug industry and (2) meteorological diagrams of wind flow in a hurricane. Scientific visualization is itself a field of endeavor.

Typically in computer systems utilizing 3D graphics, of which photorealism is a part, images are produced as a result of the simulation of a camera, known as a synthetic camera, taking a photograph of a 3D specified scene. The kind of scene, as well as how the computer internally encodes the scene, depend on the application area. For example, the encoding of a scene will undoubtedly be different for an automobile design tool, a drug design tool, or a motion picture animation system. In the automobile design tool, the scene may be stored in a graph. The nodes of the graph representing automobile parts and the edges of the graph representing how the parts are connected. In the drug design tool, the drug may be stored by a description of its chemical element composition and how these elements are connected.

To visualize the scene, the application must first calculate the geometry of the scene. For example, in the automobile design tool, the tool must read the graph representing the car, as mentioned above, and compute the exact physical dimensions of an assembled car. In computer graphics terminology, creating the exact physical dimensions of a scene is known as **modeling**; and, an application that performs modeling is known as a **modeler**. Complementarily, in computer graphic terms, a **renderer** takes the output of a modeler and creates an image. The modeler must output the physical dimensions of a scene in terms of the physical dimensions of simple objects that the renderer is capable of displaying. In other words, the modeler decomposes the scene into these simple objects (primitive objects). The renderer, for example, doesn't receive an automobile to render but, perhaps, a set of

curved surfaces.

There are two types of simple objects by which modelers and renderers communicate the geometry of the scene, namely, surfaces and solids.

Examples of simple solid objects that are used in solid modeling are cubes, cylinders, and cones. Solid modelers communicate not only the simple solid objects composing the scene but also what boolean set operators are used to combine the simple solid objects. Boolean set operators used are: intersection, union, and difference. For example, the scene might, in part, be specified by an overlapping cylinder and cube and, in addition, a boolean operator of intersection may further specify what parts of these simple solids end up in the final scene.

For modeler/renderer systems that use solid modeling, many different types of representations are popularly used. Some of these are, as listed in [FvDFH90], primitive instancing, sweeps, b-reps, spatial partitioning (octrees and BSP trees), and CSG. See [FvDFH90] for further details on these.

Surfaces are either planar polygons or **patches**. The main focus of this thesis will be on modeler/renderer systems that communicate a scene by the union of a set of surfaces, in particular, patches. Intuitively, one can think of a patch as an elastic rectangle that can be stretched, twisted, and contorted into any of a variety of curved surfaces. The modeler can placed these patches anywhere in 3D cartesian space. Typically, many small patches are connected, sharing edges, to approximate the surface of a physical object, such as an automobile. These patches are generated by the modeler. Mathematically, patches are characterized by parametric polynomials parameterized in two variables. In computer graphics terminology, these two variables are often named **u** and **v**. There exists a continuous mapping between all point in a u-v rectangle, in which both u and v span [0,1], and all the points on the 3D surface of a patch. This mapping is specified by a given set of polynomials in two variables. The names of popular patches are quadrics, bezier, hermite, and non-uniform rational B-splines (i.e., NURBS). Details of these parametric polynomials can be found in [FvDFH90]. The details of these parametric polynomial-based surfaces will not be as important, in this thesis, as the fact that both variables span [0,1].

In addition to the pure geometry of the scene, the modeler usually communicates descriptions of the color or type of surface each patch is. These surface descriptions are highly variable among different modeler/renderer systems. Some simple modeler/renderer system may simply indicate the

color and shininess of the surface while, at the other end of the spectrum, the photorealistic modeler/renderer system, considered in this thesis, can be quite detailed. For example, the surfaces can be characterized by so-called programmable routines that have the power to simulate real surfaces, such as, wood-grain, concrete, cloth, texture-mapped pictures, etc. These surface descriptions can not only mimic the color of real surfaces, such as concrete, but also can describe the texture found in real surfaces. The surface descriptions, in effect, perturb the geometry of the patch based surfaces. This adds to the photorealism by depicting the subtle shadows produced by these textural perturbations.

Modelers also communicate to the renderer varying amounts of bookkeeping information, for example, camera position and settings, kinds and positions of lights, and output format.

In the class of modeler/renders we are exploring in this thesis, the renderer will receive the geometry of the scene in terms of a set of patches, along with, the descriptions of the surface features of each patch. From this information and some additional bookkeeping information, such as, lighting and camera information, the renderer must produce an image.

An image produced by the renderer is a result of a projection of a 3D scene, represented by patches, onto a modeler-specified plane. This plane is referred to as the **image plane**. A modeler-specified rectangle on this plane will represent the output image; the specified rectangle is homologous to a frame of film in a camera, and is referred to as the **frame rectangle**. The location of the frame rectangle and the kind of projection used, i.e. orthogonal or perspective, is specified by the modeler. The image projected onto the frame rectangle is then mapped onto a corresponding 2D pixel array whose values represent the colors of dots (i.e, pixels) of the image. Typically, each pixel represents its color value as a triple, (red, green, blue). Each of these primary colors is assigned an integer value in the range [0,255]. The exact nature of this mapping is controlled by a modeler-specified rasterization/quantization process. The rasterization process breaks the frame rectangle into little squares whose color values will be mapped to the 2D pixels array. The quantization process takes floating point values of color components and converts them into integer values between [0,255].

Each pixel, then, represents the color, as projected onto the frame rectangle, of portions of one or more patches. In the case of a pixel representing the color of more than one patch, the pixel value represents a blending of the two

surface colors. A specific technique to determine the projected surface colors is known as an **illumination model**. There are many viable illumination models described in the literature. The illumination models range from the very simple, corresponding to low quality graphics, to those that are more computationally intensive and that are appropriate to photorealistic rendering. In practice, and in the literature, techniques of different illumination models are combined to create hybrid illumination models.

There are two broad categories of illumination models, namely: (1) **global illumination models** and (2) **local illumination models**.

Global illumination models tend to be the most computationally expensive illumination techniques. And, as their name implies, global illumination models often times take the entire scene of objects and light sources into consideration in order to determine the color of a point on the surface of a single object in the scene. This allows for the correct portrayal of **interobject reflection** and **indirect lighting**. An example of interobject reflection is the reflection of other kitchen objects seen on the surface of a shiny pot. An example of indirect lighting is the light found in a closet, when, from the closet, one cannot directly view a window or other light source. This indirect closet light comes from light bouncing off of the walls and other objects in the room. Light in the sky, after sunset, is another example of indirect lighting.

The two most popular global illumination models are **ray tracing** and **radiosity**. Local illumination models tend to consider only light sources that directly illuminate an object of the scene.

Ray tracing is very good at displaying specular, i.e. mirror-like, reflections of light. In fact, ray tracing can render mirror surfaces. In the physical world, photons first emanate from light sources, then bounce off of objects and then hit the retina of the eye. In order to avoid trying to simulate the very large number of photons leaving a light source, ray tracing, performs a simulation of this phenomenon in reverse order. The ray tracing simulation proceeds first by considering a pixel on the image (homologous to a receptor on the retina), then the simulated photons are traced back to see what object, if any, these photons could have bounced off of and then these photons are traced back again to see if some of these photons could have come from a light source and/or a reflection from another object. If some photons are determined to have come from the reflection off of another object, then the determination of the color and intensity of photons reflected from the second object is calculated by a recursive call of the ray tracing algorithm. This

recursive ray tracing call traces back photons from the point of view of the first object looking towards the second object. The second objects color could be influenced by yet a third object. The recursion continues until it is determined that the amount of photons traveling along a path is insignificant. The refraction phenomenon, photons traveling through partially transparent objects, is also calculated in ray tracing. The calculations for refraction proceed in a manner similar to the reflection calculations. This whole process is invoked to assign a single color to the pixel.

Ray tracing is an illumination technique that can be used to produce photorealistic images. As mentioned above, a ray tracing renderer produces specular interobject reflection; and in addition, it also produces shadows. It tends to be a both computationally-expensive and memory-intensive technique to use. It is computationally expensive because it is called recursively for both reflection and refraction in order determine the color of a single pixel. It is memory-intensive because the entire scene of objects could be involved in the assignment of a color value to a single pixel. Therefore, the entire scene must be stored in memory—for reasonable performance. There are many techniques to improve the performance of ray tracing, but, they still have the fundamental problem of potentially having to process an entire scene of objects in order to calculate the color of a single pixel.

Besides the high computational costs of ray tracing, another drawback of ray tracing is that the technique is not ideally suited to display diffuse reflection. Whereas specular reflection is the phenomenon where light bounces off of an object close to a specified direction, diffuse reflection is the phenomenon where light bounces off of an object in all directions. Real surfaces tend to exhibit some combination of both specular and diffuse reflection.

Radiosity is another global illumination technique. It is better suited than ray tracing for displaying diffuse reflection. And, it is better at illuminating objects that are exposed to area light sources. Fluorescent light panels found in office buildings are examples of area light sources. Radiosity also produces more natural looking shadows than does ray tracing. These shadows can exhibit soft edges as opposed to the hard edges seen in an image produced by ray tracing. Radiosity is based on considering that each small patch of a scene as either a light source or light sink. And, it then uses thermal engineering techniques to determine the steady state of light energy being radiated and absorbed by each small patch. Determining the steady state of all the small patches is extremely computationally and memory intensive.

As mentioned above, illumination models are ofttimes combined. Combinations of ray tracing and radiosity techniques have been developed. These combined ray tracing and radiosity techniques are, perhaps, the finest photorealistic rendering techniques known today. However, they are prohibitively computationally and memory expensive techniques. It can literally take hours upon hours to render a single image.

As a matter of completeness, it should be mentioned that the following local illumination techniques are often combined with global illumination techniques.

There are many local illumination modeling techniques. Uniformly, they tend to be much less computationally and memory intensive than global illumination techniques. Local illumination modeling techniques determine the color of points on a surface by techniques, that, by definition, do NOT involve any of the other objects in the scene. In contrast to global illumination models, local illumination modeling techniques do NOT have to have the entire scene of objects in memory at once. Since it has been estimated that complicated scenes may have on the order of 10,000 to 1,000,000 simple objects [Ren89], the computational costs of handling the interaction of such a large set of simple objects simply overwhelms most global illumination rendering systems.

In the context of local illumination modeling, the term **local** refers to the type of information that can be used to determine the projected surface colors of a single object. The locality of a single object includes all the properties of the object itself, as well as, its environment, such as, the light sources in the scene and the synthetic camera location. The notion of locality only excludes other objects of the scene. The projected surface colors of each object of the scene can, therefore, be determined independently of any other object in the scene.

But beyond the locality restriction, mentioned above, there are no other imposed restrictions on local illumination modeling techniques. In fact, using a combination of local illumination modeling techniques a renderer can produce very impressive photorealist images. These images are produced at a fraction of the cost of an equivalent image produced using global illumination techniques. Because of the reduced cost, local illumination modeling is oftened used for commercial photorealistic rendering. However, photorealistic rendering using local illumination can still take significant processing time, in the order of hours. A technique to reduce the processing time for

local illumination rendering is the focus of this thesis.

In order to produce photorealistic images, local illumination modeling/renderering systems usually orchestrate a combination of local illumination techniques. **Programmable shading function** renderers are designed to facilitate such an orchestration. Renderers based on the **RenderMan Interface** are examples of this type of renderer. The RenderMan Interface is the photorealist graphics interface standard developed by the Pixar Corporation. (Use of the Renderman Interface can be, to my understanding, freely licensed from Pixar Corporation). The RenderMan Interface graphics standard is a protocol, between modeler and renderer, which incorporates programmable shading functions. Hereafter, a RenderMan Interface based renderer will simply be referred to as RenderMan.

In this thesis, we will demonstrate our techniques for reducing the processing time of local illumination rendering, by embedding the design and implementation of our techniques in an implementation of RenderMan.

Although the Renderman specification allows for an optional capability of global illumination techniques, such as, ray tracing and radiosity, its primary current use is as a local illumination renderer. And, currently, traditional local illumination based RenderMan systems can take 1/2 hour or more to render a frame.

RenderMan is designed to allow for the programming of a very wide class of local illuminations such as those described in [JGMH88] or [FvDFH90]. In RenderMan, the local illumination model used, in a particular instance, is wholly determined by a set of modeler-provided programmable routines known as **shaders**. Each object of a scene has its own **set of shaders** assigned to it, and this set of shaders is used by RenderMan to determine the color of a point on the surface of an object as it is projected onto the image plane. The color of this point is referred to as the **projected color**. The term **shader** is used to indicate that the primary purpose of this set of programmable routines is to determine the shade, i.e. color, of points on the surface of an object.

The set of shaders, mentioned above, is composed of six different kinds of members, namely, **surface shader**, **displacement shader**, **atmosphere shader**, **light source shader**, **interior shader**, and **exterior shader**. Each object of a scene has one and only one of each kind of shader assigned to it. However, there is one exception to this rule: there can exist more than one light source shader assigned to an object. Each of these shaders

is programmed using a C-like language. The RenderMan system, however, in effect, treats each shader as a black box. There is a RenderMan imposed ordering on the data flow and flow of control among these shaders. This imposed ordering is designed to facilitate a shader performing the task implied by its name. So, for example, typically, the C-like code in a light source shader will be used to model lights and the C-like code in an atmosphere shader will be used to model the effect of light being transmitted through the atmosphere. A shader can be a **null** shader, providing no functionality.

The primary shader responsible for determining the projected color of a point on a surface is the surface shader. All the other shaders can be considered, to some extent, as auxiliary routines to the surface shader. During a RenderMan execution, RenderMan determines which points on which surfaces it wants the projected colors for. Then, for each of these points, RenderMan, one at a time, asks, i.e. calls, the assigned surface shader for a determination of the projected color of this desired surface point. Each invocation of the surface shader is provided with a number of appropriate pieces of information that it may base its output color on. The surface shader does not have to consult these pieces of information in order to make its color decision; however, it is indeed very likely that a programmable surface shader will base its color result on some of the information provided to it by RenderMan.

The information provided by RenderMan each time it invokes a surface shader includes:

1. surface color,
2. surface opacity,
3. 3D surface position,
4. derivative of surface position along u,
5. derivative of surface position along v,
6. surface shading normal vector,
7. surface geometric normal vector,
8. surface texture coordinates, and

9. 3D position of the synthetic camera.

See [Ren89] or [Ups90] for details on this information.

It should be emphasized that how the surface shader uses this information in determining the projected color of a surface point is specific to a particular surface shader. Surface shaders have been programmed to simulate such surfaces as wood, marble, bricks, and rusted copper.

All shaders that get invoked by RenderMan share the same general mode of operation. Each shader invocation is provided, by RenderMan, with appropriate information, often geometric in nature, about a particular surface point and its environment (excluding information about other objects, of course), and is then required to produce some result. The RenderMan system, itself, controls the flow of information passed among the individual shaders.

As an example of how RenderMan passes information among the set of shaders, let us consider how information is passed between the displacement shader and the surface shader. As mentioned earlier, on each invocation of the surface shader, RenderMan provides the surface shader with information. This information has been listed above. RenderMan allows the displacement shader to preprocess this information in order to model the texture of a surface. The displacement shader can modify the values, if appropriate, of the 3D surface position, item (3) listed above, and the surface shading normal vector, item (6) listed above; these modified values will be subsequently provided, by RenderMan, to the surface shader. These modifications could, for example, perturb the geometry of the surface in a way that models the bumpy surface texture of concrete.

On the invocation of a surface shader by the RenderMan system, light source information is not automatically provided to the surface shader; it must be requested by the surface shader itself. In general, in RenderMan, extra flexibility is gained by providing the surface shader with the ability to request additional information from the RenderMan system. For example, a surface shader, may request, from the RenderMan system, information about what types of lights, in terms of intensity, color and direction, strike the point on the surface of an object being shaded. RenderMan allows the modeler to specify programmable light sources, known as light source shaders. In order to fulfill the surface shader's request for light information, RenderMan invokes each light source shader that could possibly shed light on the point

being shaded. Each light source shader is provided with information in order to determine its results. This information is similar in nature to information provided to a surface shader. Each light source shader's result, in terms of intensity, color and direction, is then passed onto the surface shader. Light source shaders have been programmed to simulate such lights as spotlights, point lights, and distant lights such as the sun.

In addition, the surface shader could request, from the RenderMan system, other types of information. RenderMan would obtain this information by invoking other programmable functions, i.e. shaders. These shaders go by the names of **internal volume shader** and **external volume shader**. All these shaders are provided with information by the RenderMan system and produce an answer that can be used by the surface shader. After the color of the surface point is calculated by a surface shader, RenderMan invokes another shader, named the atmosphere shader, to account for atmospheric effects, such as fog, that may influence the color of the surface point as observed by a synthetic camera. The total set of programmable routines, i.e. shaders, viewed as a whole, embody the local illumination model used.

The exact nature of how these shaders are programmed is not important for this thesis. The shaders are, in fact, programmed in a C-like language, see [Ren89] or [Ups90] for details. The idea to emphasize here is that a RenderMan system must select some set of surface points in the scene to shade. And, it must invoke a set of programmable routines, i.e. shaders, to calculate each color of these points. There is not much performance optimization a RenderMan system can apply to the shaders. After all, they are C-like programs which RenderMan invokes but does not know the internal structure of. RenderMan has to invoke them unmodified. On a uni-processor computer, differences in RenderMan system performances, in terms of speed, are primarily due to the differences in how cleverly the renderer can pick what points on what objects it wants to shade.

The RenderMan interface between modeler and renderer calls out for the specification of a shading rate in terms of shaded points per pixel. This amounts to specifying the density of shading points that have to be processed in order to determine the color of a pixel of the image. Objects very far away from the synthetic camera may appear small on the image and will not have to have many shaded points in order to satisfy the shading rate requirement. On the other hand, the same object, if closer to the synthetic camera, may appear larger on the image, therefore, requiring more shading points to satisfy

the shading requirement.

The focus of our thesis is on an algorithm that will judiciously pick the surface points to be shaded, and in addition, will shade, in parallel, these surface points. Thus, by parallel shading, we will gain a significant overall rendering time performance improvement over traditional uni-processor approaches. Initially, the algorithm selects a few surface points of an object to shade and then determines whether or not it has satisfied the shading requirement, if not, even more points on the surface are selected and shaded in parallel. This process continues until the surface shading rate requirement is satisfied.

As mentioned previously, the total set of user-programmable routines, i.e. shaders, viewed as a whole, embody a local illumination model. RenderMan shaders were designed to allow the programming of a wide class of local illumination modeling techniques. With this in mind, in the remainder of this chapter, we will review a few of the more commonly used local illumination modeling techniques that can, of course, be programmed using RenderMan shaders.

Portraying realistic surface details is a major contributor to the sense of realism in a rendered image. Surface details include such real life features as bumps, scratchs, marks, grains, textures. These surface details can be provided by **texture mapping** and **bump mapping**. Intuitively, **texture mapping** amounts to taking an existing image and wrapping it around an object. The image being texture-mapped is often a picture of surface material, such as, wood grain, concrete, brick. But it could be of any image, for example, a picture of a person could be used as a texture map. **Bump mapping** is a similar process to texture mapping; however, instead of the image, in terms of color, being wrapped around the object, the bump map contains height information and the geometric surface of the object being wrapped gets correspondingly perturbed, analogously to an embossing process.

Texture mapping and bump mapping can together produce very realistic surfaces, such as an orange peel surface or brick wall surface. The process of texture mapping and bump mapping can be generalized. Instead of consulting a texture map or bump map for information on how to modify the surface color and geometry, a programmable function can be used. This is the historical context in which shader functions were created. Shading functions have been programmed to simulate such diverse surfaces as wood grain, marble, and the texture of cloth.

Many of the global illumination effects, such as mirror-like specular reflection, can be approximated by using just a local illumination modeler. Since local illumination does NOT allow other objects to affect the shading process, mirror-like speculation must be performed, using local illumination, as a multi-pass operation, i.e., executing the renderer multiple times.

To simulate the global illumination effect of mirror-like specular reflection by utilizing only local illumination techniques, the following multi-pass operation could be used. Place a cube around the mirror-like object. Place the synthetic camera at the center of the cube. Now direct, in turn, the synthetic camera to point to each of the six faces of the cube. At each face render an image with each face used as the frame rectangle and the center of the cube used as the location of the synthetic camera. These six images will then be used as texture maps that surround the mirror-like object.

For example, if the scene contained a shiny pot in a kitchen, one would render six frames taken at the location of the shiny pot looking out in all six directions. These frames would be used as texture map information to be wrapped around the shiny pot.

This simulates interobject reflection. However, this method has a couple of problems: (1) it requires special handling by the modeler to obtain interobject reflections and (2) depending on the scene, the shiny pots surface could have exhibited self reflection which won't be shown by this method. But the reason this technique is used commercially is because the results, although not a 100% accurate, look pretty good and can be obtained at a fraction of the cost of global illumination techniques. And typically, there are only a relatively few really shiny objects in a natural scene.

Shadows from light sources, normally considered a global illumination effect, can also be approximated by a local illumination modeler using a two-pass technique. This time a frame is rendered at the light source looking out in the direction that the light source is pointing. The frame, instead of recording color, records the location of the surface of the objects in the scene that the light source directly shines on. This information is used when rendering objects of the scene in determining which part of the object may be in shadow. Again, the same argument of using this technique over a global illumination technique applies.

See [JGMH88] and [FvDFH90] for detailed discussions on several popular illumination models.

Example

The intention of this very simple example is to give the reader a flavor of the RenderMan interface protocol for communicating between a modeler and a renderer. The following protocol example illustrates the specification of single patch. Normally a modeler will specify thousands of patches. The patch, in this case, is a section of a surface of a sphere. This patch is texture-mapped with a picture of the book cover of [PS85].

In this example, the protocol is:

```
(0)  Format 500 400 -1
(1)  Display "example.tiff" "tiff" "rgb"
(2)  Projection "perspective"
(3)  Rotate   -10 1.0 0  0
(4)  WorldBegin
(5)     LightSource "distantlight" 1
(6)     Surface "texture_mapper" "mapname" "book_cover"
(7)     Translate 0 -2  40
(8)     Rotate    80.0   0.0 1.0 0.0
(9)     Rotate   -80.0 1.0 0.0 0.0
(10)    Sphere    30.0 -12.0 12.0 60.0
(11) WorldEnd
```

RenderMan assumes many defaults. In this example, the default for the location of the image plane is used. This default is the plane perpendicular to the z-axis and intersecting z=1. Lines (0)-(3) take care of bookkeeping information and camera location. Line (0) defines the resolution, in terms of pixels, of the output image. Line (1) specifies the output format of the output image. Line (2) indicates that perspective projection is to be used. And, line (3) changes the orientation of the synthetic camera.

The WorldBegin command, line (4), heralds the beginning of the descriptions of the objects in the scene. RenderMan is a graphics-state based renderer. That is, RenderMan accumulates a state as it processes the protocol. Line (5) adds to the graphics-state a light source shader named **distantlight**. Line (6) adds a surface shader named **texture_mapper** to the graphics state. Lines(7)-(9) specify where the upcoming patch will be located. Line (10) specifies the patch itself. The patch, in effect, gets stamped with

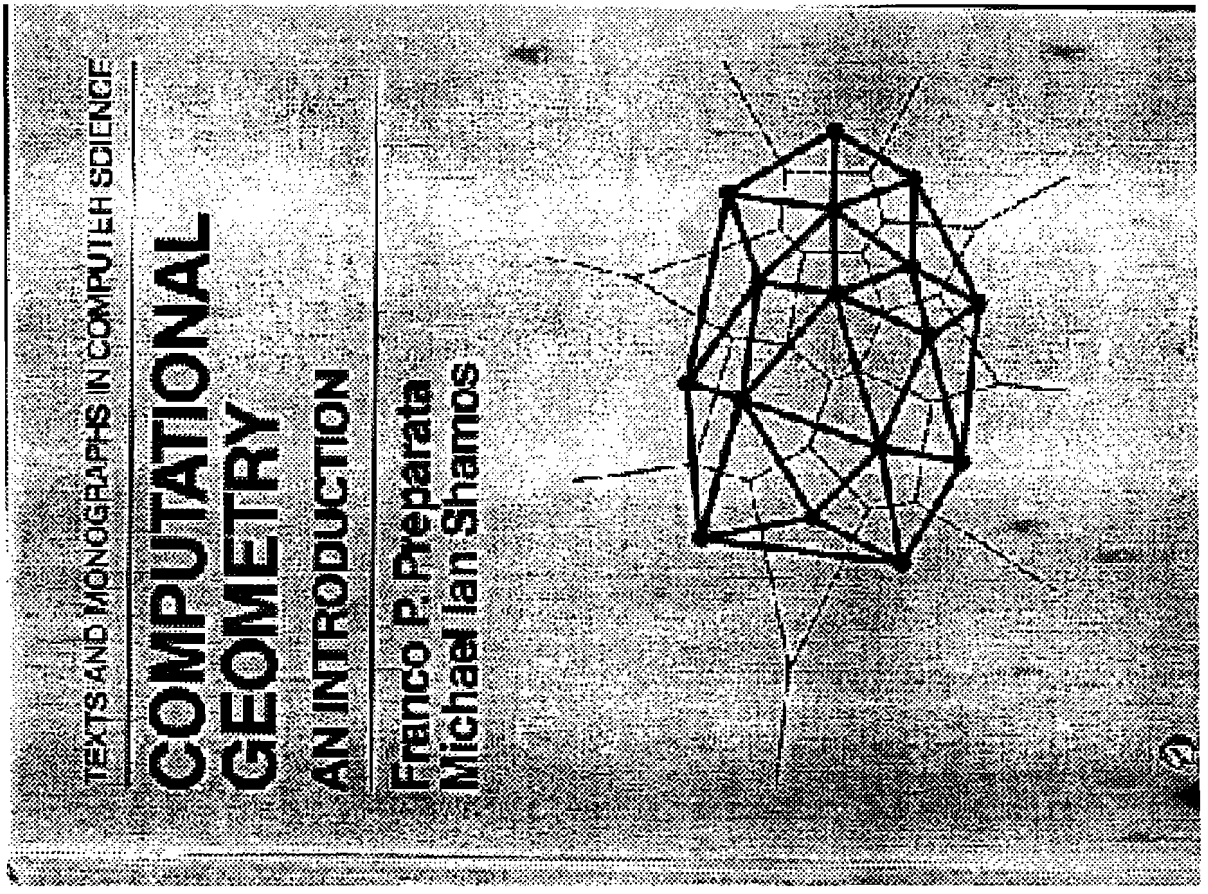


Figure 1.1: Example texture map of a book cover

the graphics-state that exists when RenderMan processes the patch specification. So, in this case, the sphere patch, gets stamped with the surface shader named texture_mapper. In determining the colors of the surface of this sphere patch, RenderMan will use the surface shader named texture_mapper for each point it shades. And, if the texture_mapper surface shader requests information, from the RenderMan system, about light, RenderMan will, in turn, obtain this information from the light source shader named distantlight.

Line (11) indicates that the description of the scene of objects has ended, and RenderMan is now required to produce an image.

In this example, the surface shader is aptly named **texture_mapper**. This surface shader uses the texture map shown in Figure 1.1 to wrap around

the patch. This surface shader also makes use of the **distantlight** light source shader. The result is shown in Figure 1.2.

Figure 1.2: Example of a texture-mapped patch

Chapter 2

The Parallel Adaptive Point-Sampling Algorithm

2.1 Motivation

This thesis reports on the design and implementation of a set of related parallel algorithms used to implement a fast version of RenderMan.[1] The algorithms are designed to execute on the C*/CM-200, software/hardware, combination. The C* software and CM-200 SIMD hardware are both manufactured by Thinking Machines Corporation. C* is a variant of the C programming language. C* contains constructs that allow the programming of parallel algorithms. However, the algorithms could be easily altered to run and take advantage of a MIMD architecture.

The CM-200 hardware used in this project possesses a small amount of memory. The RenderMan standard encourages the ability to process very large files. This led to a design that renders each primitive geometric object specification, i.e. planar polygon or patch, immediately after it is observed in the protocol stream provided by the modeler. The RenderMan standard does indeed allow a RenderMan process to delay the rendering of a primitive geometric object; however, this design option would require too much memory. The rendering of each primitive geometric object is done utilizing the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm described in the following

[1] RenderMan is Pixar Corporation's photorealistic interface standard as mentioned in the **Introduction** section of this thesis.

section. It is designed to run quickly, and if a primitive geometric object does not project onto the frame rectangle, it will be quickly dismissed and further processing of the protocol stream will resume.

Our design allows for either a painter's or a z-buffer algorithm. This is required by the RenderMan standard. A painter's algorithm depicts a scene of primitive geometric objects as though during the processing of the protocol stream, each primitive geometric object rendered will have its 2D projected representation painted over the entire scene of the, up to this point, processed set of primitive geometric objects. In order to produce an output image that properly depicts the occlusion of one object by another, the modeler must order the sequence of object specifications, sent in the protocol stream, by the object's distance from the synthetic camera; the farthest object is ordered to be first.

In contrast, the z-buffer algorithm depicts each primitive geometric object's 2D projected representation, in the output image, by considering the depth of its corresponding 3D surface points, as observed by the synthetic camera. That is, if part of the surface of one of the primitive geometric objects is totally or partially occluded by another, as observed by the synthetic camera, this occlusion will be properly depicted in the output image.

The RenderMan standard requires a number of features. For example, the RenderMan standard requires the density of sampling (i.e., the number of points shaded per pixel) be specifiable in the protocol stream. The RenderMan standard also requires the inclusion of image processing capabilities. And in addition, although not implemented, our design allows for depth of field and motion blur calculations.

These are the considerations that went into the design and implementation of the algorithms described in the following section.

2.2 Algorithm

RenderMan processes the following primitive geometric types: planar polygons and patches, including quadric patches. These primitive geometric types are the only geometric types supported by RenderMan. All higher level objects, such as cars, chairs, faces, etc., must be composed of these basic primitive geometric objects. RenderMan supports both surface modeling and CSG (i.e., Constructive Solid Geometry). This projects only focuses on

surface modeling, in particular, the current implementation for this project processes only quadric patches.

In RenderMan, each primitive geometric object, in essence, is required to have a u-v rectangle associated with it, **u** and **v** being the labels of the axis of the rectangle: u, the horizontal, and v, the vertical. In each **associated u-v rectangle**, both u and v span [0,1]. Each point of a 2D associated u-v rectangle maps to a single point on the corresponding 3D patch; although the reverse is not necessarily true.

The RenderMan system selects, for the purpose of shading, surface points of a primitive geometric object. In our RenderMan system, we use the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm to determine the selection of these surface points. Our PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm begins by selecting points on the u-v rectangle associated with a primitive geometric object. These u-v points are then mapped to their corresponding surface points, in 3D cartesian space, and are then shaded by their assigned surface shader.

The u-v rectangle is an essential information structure available to the user of RenderMan. As mentioned earlier, a surface shader, assigned to each primitive geometric object, is used to determine the projected surface colors of a primitive geometric object. When a surface shader is invoked, the RenderMan system makes available to the surface shader the u-v point that corresponds to the 3D surface point being shaded. The surface shader can use this u-v coordinate in any of a number of ways. It can use this u-v coordinate to access values from a texture map or bump map. Or, it could use it as the parameter of a function that determines the color of surface. This function could be, for example, a **wood** function, whose domain are the u-v coordinates and whose functional image appears to be wood grain.

When implemented on the C*/CM-200 system, most of the informational data structures described in the following PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm are defined in terms of C* parallel arrays. Each element of a C* parallel array has its own processor associated with it. During the following description of the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, it seems natural to anthropomorphize these elements or, more appropriately, view them as **agents**. For example, the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm uses an information structure which contains pixel values and another information structure that contains values related to surface points. During the description of the algorithm, we might use a

phrase such as *each surface point sends its color information to its corresponding pixel.* By this we mean, the processors holding the color values of a surface point send their color values to the processors holding the color values of the pixels.

PARALLEL-ADAPTIVE-POINT-SAMPLING

The PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm is described as a recursive algorithm; however, it is implemented iteratively.

Our RenderMan system invokes a top-level call of the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm for each primitive geometric object in the scene. This algorithm depicts, i.e. renders, one object of the scene, as viewed from the synthetic camera. It will create and store this depiction in a 2D pixel array named the **object-image** array. It will then **composite** this newly created image stored in the object-image array with an image stored in the **scene-image** 2D pixel array. Compositing is a method of combining two images and will be described later. The scene-image array is structurally similar to the object-image array but, as its name suggests, it represents the scene as a whole. That is, after our RenderMan system has processed all the primitive geometric objects, it will contain a composited image of all the visible primitive geometric objects. Before any primitive geometric object is subjected to the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, the scene-image array is initialized.

In the process of rendering each object, the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, initially, selects a few surface points to render and then subjects these points, in parallel, to the surface shader[2]. Until the **sampling rate**, as specified by the modeler, is satisfied, this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm is recursively called in order to process more surface points, in parallel. The term sampling rate, as used in RenderMan, is defined as the number of surface points processed per pixel.

[2]In some cases these points are subjected to an "interpolation" procedure rather than a surface shader. See step 5, page 40.

A more complete definition of sampling rate can be found in step 4, page 36.

Inputs

- An array of **u-v subrectangles** related to the geometric primitive object being rendered (to be characterized below).

- A set of parametric polynomials in two variables that specifies, for the primitive geometric object being rendered, the mapping from the u-v space to the 3D cartesian space.

The term **u-v subrectangle** is used to emphasize that each u-v rectangles used as input to this algorithm is located within the given geometric primitive object's associated rectangle. u-v subrectangles are used to designate points on the surface of a geometric primitive object. Each u-v subrectangle designates four u-v points; one point for each corner of the u-v subrectangle. These four u-v points are mapped, by this algorithm, to their four corresponding 3D cartesian surface points; these 3D points are considered to be the sampled points. Sampled points are subjected to projected surface color determination, e.g., the surface shader. See step 5, page 40 for details.

When our RenderMan system invokes the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, it inputs a single u-v subrectangle: the primitive object's associated u-v rectangle. If the four points of this associated u-v rectangle do not satisfy the sampling rate requirement, the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm breaks up the rectangle into four new u-v subrectangles. These four u-v subrectangles then comprise the array of u-v subrectangles used as input to a recursive call of this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm. If the sampling rate is still not satisfied, each of the four u-v subrectangles could be broken up resulting in a total of 16 new u-v subrectangles. This process continues until the sampling rate is satisfied for all input u-v subrectangles. See step 4, page 36, of this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm for details on how a u-v subrectangle satisfies the sampling rate.

Step 0: Initialization

Initialize the object-image array. This initialization is applied once and only once per geometric primitive object. If this algorithm were implemented in a recursive manner, this initialization step would only be executed at the top-level call to this recursive PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm.

Step 1: If no u-v subrectangles are input, then the object-image array and scene-image array are composited

This step determines whether or not the entire PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm terminates at this step. The termination condition is satisfied if no u-v subrectangles are input to this particular recursive call of the algorithm.

The termination condition indicates that the sampling rate for the object being processed has been satisfied, and that a fully sampled image of the primitive geometric object resides in the object-array. If this is the case, in this step, the object-image array will then be composited with the scene-image array and, then, the invocation of the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm will terminate.

On the other hand, if the input array of u-v subrectangles has one or more elements, then this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm does not terminates and proceeds directly to step 2, page 30.

If the object-array has not been written into, then the primitive geometric object being rendered is not visible to the synthetic camera and this recursive PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm terminates without performing compositing.

Compositing is a technique to combine two images. Each pixel of each image contains three values, namely, the color, α **(pixel coverage)**, and **z-depth** to be defined below. In this step, the scene-image array already contains these values. While for the object-image array, these values must be calculated from information produced in step 6, page 42.

The meaning of the term **color** is the standard one. Each color is represented by the triple (Red,Green,Blue).

The α value represents the fraction of a pixel's area that contains color. For example, a pixel may represent the color of a surface that, when projected, only covers 1/4 of area of this pixel. See Figure 2.1. If this were the

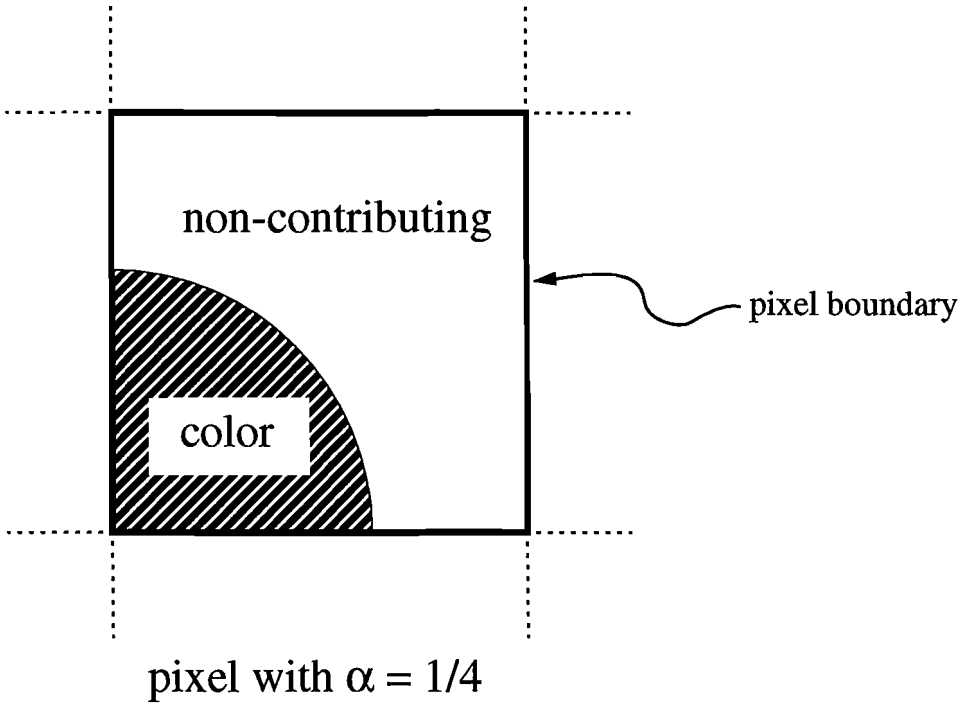


Figure 2.1: Example of α

only surface affecting the color of this pixel, it would contain an α value of 1/4. The remaining 3/4 area of the pixel would be considered **non-contributing**, that is, this part of the pixel does not contain color information. It should be noted that once an α value is determined, the information concerning where, on the surface area of the pixel, the color is located, is lost. Therefore, conceptually, the color is considered to be randomly distributed around the area of the pixel.

The **z-depth** value represents the distance from the synthetic camera of the 3D surface that is projected onto the pixel. In this implementation, the z-depth value is stored as a scaled value that lies between the values of [0,1]. This concept of a scaled z-depth value is related to **3D pixel space** described in step 2, page 30.

The modeler can specify whether the surface of a geometric primitive object, i.e., a patch, is rendered to show only one side of the surface, the outside, or both sides, the inside surface, as well as, the outside. Determining

the object-array values of color, α (pixel coverage), and z-depth is different for each case. Each case is considered separately below.

Case 1. WHERE ONLY THE OUTSIDE SURFACE IS DESIGNATED TO BE SHOWN. The algorithm proceeds as follows.

During step 6, page 42, of this algorithm, for each pixel of the object-image array, a running **weighted average of color and z-depth** is recorded. These current weighted average values of color and z-depth are, in this step, stored in the object-image as final values. See step 6 for details on the term running **weighted average of color and z-depth** .

The modeler-specified shading and sampling rate, as defined in step 4, page 36, dictates the total number of shaded points (samples) taken by the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm in the case of a fully covered pixel. A fully covered pixel is a pixel in which the projected surface of a primitive geometric object totally covers the surface area of the pixel. The number of samples taken for a pixel in the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm is proportionally related to the amount of pixel area covered by the projection of the outside surface of the object onto the pixel.

Thus, in this step, the α (pixel coverage) value for each pixel is determined by dividing the actual number of points (samples) made for the pixel by the number of points (samples) of a fully covered pixel. An α value of less than one would result if the pixel represents an edge of the primitive object being rendered. So, if 2 samples were taken for a pixel that would have 16 samples in a fully covered pixel, then the α value would be 1/8th. All pixels are processed in parallel in determining α values.

Case 2. THE OUTSIDE AND INSIDE SURFACES ARE SHOWN. The algorithm proceeds as follows.

The outside surface of a primitive geometric object may, from the viewpoint of the synthetic camera, occlude the inside surface. For example, Figure 2.2 depicts a pixel on which a section of an open-ended cylinder is projected. In this example, the outside surface is red and the inside surface is brown. The red outside surface occludes the brown inside surface.

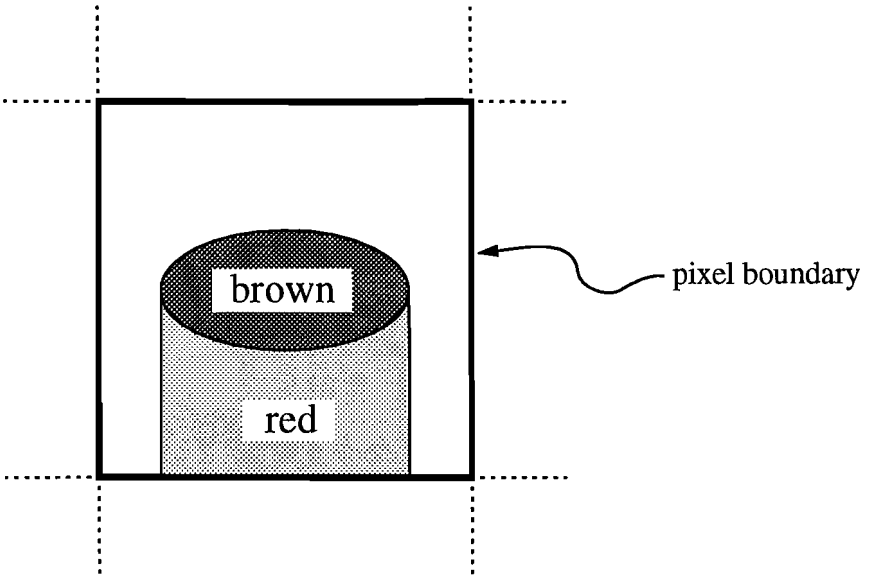


Figure 2.2: Example of a projection, onto a pixel, of an outside and inside surface of an open-ended cylinder

So, for example, during the sampling process of the pixel shown in Figure 2.2, some of the brown inside-surface points sampled are occluded by the red outside surface and should not be included in the final determination of the color or α value of this pixel. In order to approximate, in the output image, this occlusion phenomenon, separate bookkeeping is kept, during step 6, page 42, for the information related to the sampled points of the outside surface and inside surface.

Running weighted averages of the color and z-depth values of the sampled points for both the outside and inside surfaces are recorded in step 6. In addition, the average location of the sampled points for both the outside and inside surfaces, in terms of their projected pixel space location, is also recorded during step 6. In order to approximate the occlusion phenomenon from this information record in step 6, the following procedure involving two constructed squares is used.

Figure 2.3 shows two squares constructed from the two sets of sampled points taken for the pixel shown in Figure 2.2. The red square represents the outside sampled surface points and the brown square represents the inside sampled surface points. The centers of the squares are the average location,

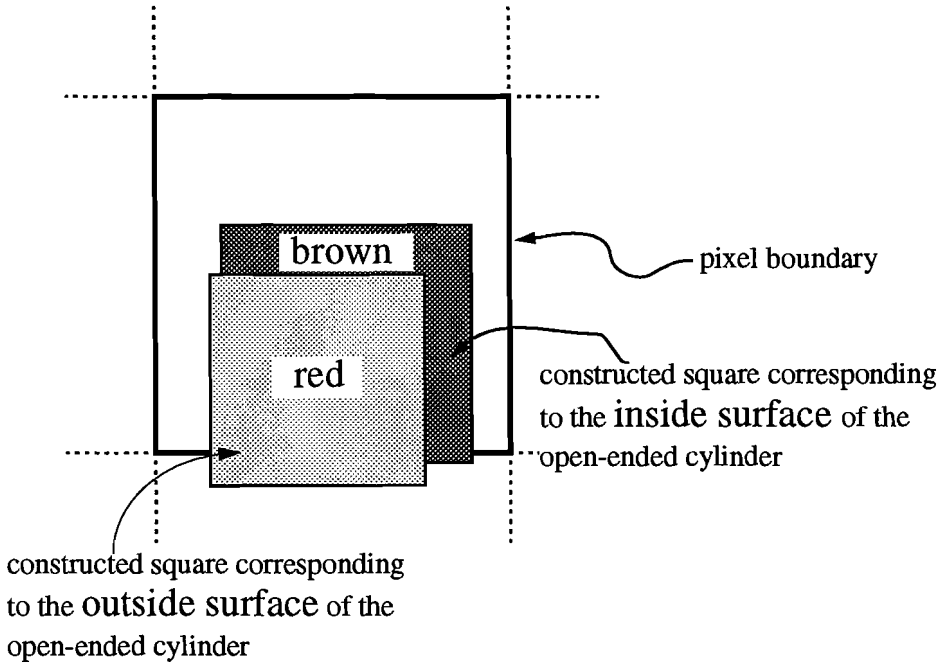


Figure 2.3: Example pixel with an illustration of the constructed squares corresponding to the outside and inside surfaces of the open-ended cylinder shown in Figure 2.2

as mentioned in the previous paragraph, for each set of sampled points. The size of each square corresponds to the number of samples taken by this algorithm, for each type of surface, relative to the total number of samples taken by this algorithm for a fully covered pixel. In our example, the sampling rate dictates that the RenderMan system processes 20 samples per pixel for a fully covered pixel. In our example, we do not have a fully covered pixel. And, in this example, 7 samples are taken for the red outside surface and 10 samples are taken for the brown inside surface. Thus the area of red square is 7/20th the area of the pixel. And the area of the brown square is 10/20th the area of a pixel.

The constructed square on top, with respect to the viewer, is the square with the smallest average z-depth value for all its sampled points.

It is acceptable that the constructed squares go outside the boundary of the pixel; these constructed squares are only for calculation purposes. The

α value for each pixel is calculated by dividing the visible shaded area, as shown in Figure 2.3, by the area of a single pixel.

In our example, as shown in Figure 2.3, all of the area of red square is visible (7/20th the area of a pixel) and 40% of the area of the brown square is visible (4/20th the area of a pixel). Adding the visible area of the two squares results in the total fraction of the area of the pixel containing color, in this example, (4/20 + 7/20) or 0.55. This is the pixel's α value[3]. The pixel's color and z-depth values are determined by the average of the outside and inside surface square's color and z-depth values weighted by the amount of visible surface area of each square. All pixels are processed in parallel.

As mentioned above, after color, α (pixel coverage), and z-depth values of the object-image have been determined, this step composites the object-image and the scene-image array.

There are many ways to combine two images using color, α (pixel coverage), and z-depth information. RenderMan supports **painter's** and **z-buffer** compositing. These techniques are fully motivated and described in [PD84]. All techniques, described in [PD84], composite two images by combining corresponding pixels from the two images, independent of other pixels in the images. Thus, during compositing, this algorithm combines each corresponding pair of pixels in parallel. Although this thesis does not motivate the derivation of the compositing formulas, we will, in the following *Notes on compositing* section, specify the compositing formulas used in this step.

Notes on compositing

The previous step involves compositing the object-image array with the scene-image array. This note explains, in detail, how compositing is performed. There are two types of modeler specified compositing styles: painter's and z-buffer. The compositing algorithms used in the previous step are derived from [PD84]. This section only contains formulas. Consult [PD84] for details.

As mentioned earlier, these two images are stored in arrays that have identical structures, each element of an array representing a pixel. The size of the arrays corresponds to the number of horizontal and vertical pixels requested, by the modeler, for the final output image.

[3] α is defined on page 23

Each pixel, whether stored in the object-image or scene-image, contains color, α (pixel coverage), and z-depth information. During the compositing process, each pixel location is processed independently. Each pair of corresponding object-image and scene-image pixels are composited in parallel. This parallel compositing process results in a new scene-image.

If the painter's compositing style is requested by the modeler, the new scene-image is calculated by compositing the object-image over the scene-image. Otherwise, the z-depth value is used to determine which pixel, considering corresponding pixels from the scene-image and object-image, is composited over the other. After this determination, the composition algorithm proceeds in an identical manner, regardless of whether the painter's or z-buffer composition style is requested. We will use **A** to designate the pixel on top, closer to the viewer, and **B** to designate the pixel on the bottom.

As is typical in compositing systems, color, α, and z-depth values, (R,G,B,α,z-depth), are stored as (αR,αG,αB,α,z-depth).

In the following formulas, **F** stands for the fraction of the color area of a pixel that ends up in the composited pixel. Remember, the color area refers to the amount of pixel area which contains projected surface color, as illustrated in Figure 2.1.

From the [PD84] paper:

$$\begin{aligned} F_A &= 1.0 \\ F_B &= 1.0 - \alpha_A \end{aligned}$$

Composition of color and α value is as follows: The color of the new scene-image pixel is

$$F_A color_A + F_B color_B$$

The new α value of the new scene-image pixel is

$$F_A \alpha_A + F_B \alpha_B$$

The calculation for the composited z-depth value is slightly different in form from the composited α or color value just described. The above formu-

las, for composited α and color values, take into consideration that the composited value may have non-contributing area, that is, the pixels being combined might not have their entire pixel area covered with color—embodied in the notion of an α value. However, the z-value is based solely on the weighted average of color contribution area, if any, each pixel contributes to the composited pixel.

There is no exact z-value for a composited pixel. Our pragmatic decision, for this implementation, was to use a weighted average of the z-depth values of both pixels based on the amount of color, by pixel surface area, contributed by each pixel.

W_A and W_B designate, respectively, the weight given to A and B's z-depth value.

If the divisor, used in the following two equations, is zero, then no z-depth value is assigned.

$$W_A = \frac{F_A \alpha_A}{F_A \alpha_A + F_B \alpha_B}$$
$$W_B = \frac{F_B \alpha_B}{F_A \alpha_A + F_B \alpha_B}$$

The new z-depth value of the new scene-image pixel is:

$$W_A z_A + W_B z_B$$

Step 2: Select additional points on each u-v subrectangle and then map them to their corresponding points in 3D pixel space

In this step, all the u-v subrectangles are processed in parallel.

An image plane, on which the projected surface colors are represented and whose integer-valued coordinates demarcate rectangular areas which are mapped to pixel values, is referred to, in this thesis, as **2D pixel space**. Similar to the pixel plane, the **3D pixel space**, has, in addition, a z-component.

3D cartesian surface points that are displayed in the output image lie, when mapped (projected) to the 3D pixel space, in the **view volume** of the 3D pixel space. The view volume consists of an x-component spanning [0, number of horizontal pixels in the output image], a y-component spanning [0,

number of vertical pixels in the output image], and a z-component spanning [0,1]. In the view volume, a z-component having a positive value closer to zero would be a point closer to the viewer.

For each image rendered, each sampled 3D cartesian point which influences the color of a pixel maps to a 3D-pixel-space point which lies within the view volume. Those sampled 3D cartesian points that do NOT influence the color of any pixel map to 3D-pixel-space points that lie outside of the view volume. For each image rendered, the mapping between 3D cartesian space and 3D pixel space is determined by the modeler's specifications,.e.g., the synthetic camera location and the output-image size.

The view volume is the raison d'etre for the concept of 3D pixel space. The width and height of the view volume corresponds, by design, to the width and height, in terms of the number of pixels, of the output image. Applying the floor function to a sampled point's x and y coordinate in 3D-pixel-space results in the x and y designation of the pixel that is influenced by the sampled point. The z-depth of a sampled point is its distance away from the synthetic camera. In 3D-pixel-space, the values of the z-components of all sample points that affect the output image lie between [0,1]. Thus, the z-component of a 3D-pixel-space point represents a scaled z-depth in which cartesian-space points that are closer to the synthetic camera map to points closer to 0 in 3D pixel space, and conversely those cartesian space points that lie farther away from the synthetic camera map to points that are closer to 1 in 3D pixel space. This scaled z-depth is used by step 1, page 23 for compositing.

Thus, there are three types of spaces involved in this step, namely,

- the u-v space,
- the 3D cartesian space, and
- the 3D pixel space.

In this step, selected (sampled) u-v space points get mapped to 3D surface points. These 3D surface points then get mapped, or projected, into the 3D pixel space. The protocol between the modeler and the renderer specifies the mapping among these three spaces. For example, Figure 2.4 illustrates a mapping of a point in u-v space to its corresponding 3D cartesian space point on the surface of a cone. And, Figure 2.5 illustrates the mapping of this

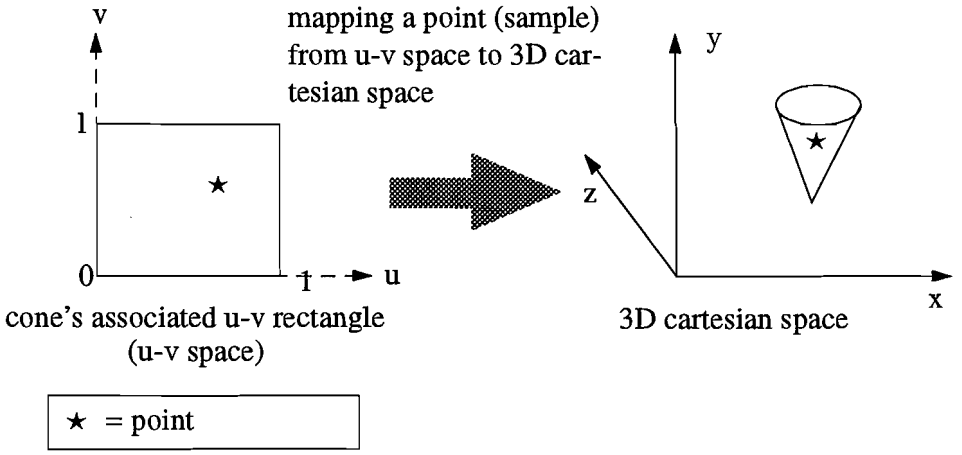


Figure 2.4: Example of mapping a point (sample) from a geometric primitive's associated u-v subrectangle to a point in 3D cartesian space

point on the surface of the cone, in 3D cartesian space, to its corresponding point, in 3D pixel space, which, in this example, happens to be located in the view volume.

The mapping between u-v space and 3D cartesian space is determined by a set of parametric polynomials in two variables that specifies the surface, i.e., the patch, of the primitive geometric object being rendered. In the example illustrated in Figure 2.4, the primitive geometric object is a cone. The set of parametric polynomials is part of the input to this algorithm, see page 22.

The mapping between 3D cartesian space and the 3D pixel space is specified in the protocol between the modeler and renderer. This specification is in terms of the type of projection requested, orthogonal or perspective, and other details such as those concerning the position and orientation of the synthetic camera.

In our RenderMan system, as is typical, the mapping of a 3D-cartesian-space point to 3D-pixel-space point is performed by a single matrix multiply between a vector, representing the point[4], and a **4x4 matrix** representing the mapping. A 4x4 matrix is a matrix whose elements can embody the

[4]The point is represented in homogeneous coordinates—a four valued vector. Homogeneous coordinates are commonly used in computer graphics. See [FvDFH90] and [PS85]. Homogeneous coordinates allow a single matrix multiply to perform the mapping between coordinate 3D cartesian space and 3D pixel space.

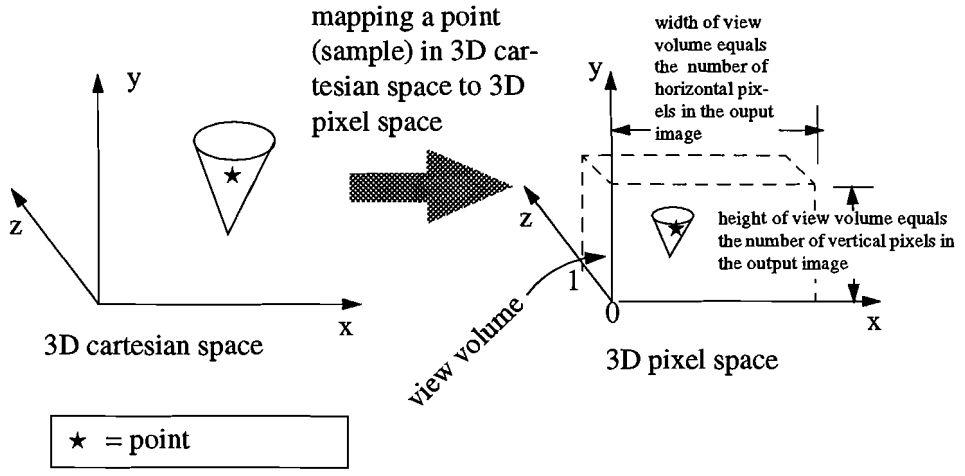


Figure 2.5: Example of mapping from a point in 3D cartesian space to a point in 3D pixel space

mapping from 3D cartesian space to 3D pixel space in its 16 elements. The use of matrices in performing mappings between spaces is standard practice in the computer graphics field. The details of how this is done can be found in [FvDFH90].

This step calculates (selects), in parallel, the u-v points, indicated in Figure 2.6, for each u-v subrectangle. These u-v subrectangles are those that are input into this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm. These newly selected u-v points are midpoints in u-v space and are, namely, points "e", "f", "g", "h", and "i". These points are then mapped to 3D-pixel-space points and their locations in both u-v space and 3D pixel space is stored.

This PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm requires, at this point in the algorithm, that the u-v space locations and corresponding 3D-pixel-space locations of the input points "a", "b", "c", and "d", shown in Figure 2.6, be already calculated and stored.

Step 3: Prune u-v subrectangles that, when mapped into the 3D-pixel space, do not wholly reside in the view volume

In this step, all the u-v subrectangles are processed in parallel.

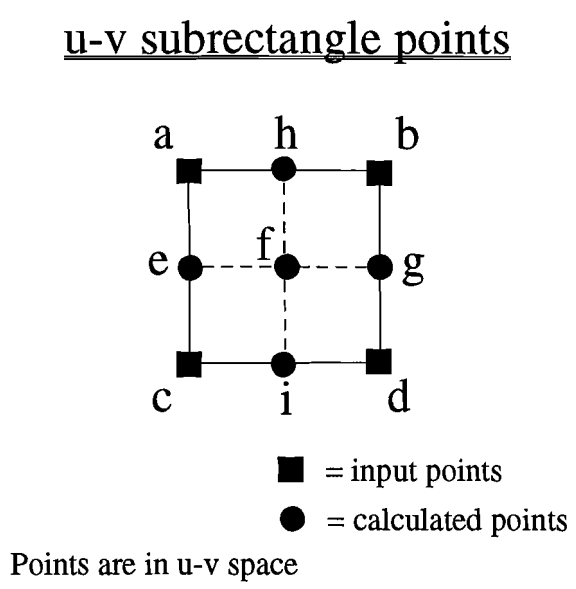


Figure 2.6: Labeled u-v subrectangle points

When a u-v subrectangle is designated as **pruned**, it is not further processed by this algorithm. A u-v subrectangle is designated as pruned on two conditions. One condition is the fulfillment of the shading and sampling rates, and is discussed in step 4, page 36. And, the other condition is when the u-v subrectangle's corresponding 3D patch is not at all viewable by the synthetic camera, and this is further discussed in this step.

Surface points viewable to the synthetic camera are those surface points that map into the 3D pixel space's view volume. The view volume is shown in Figure 2.4. In order for a u-v subrectangle to be designated as pruned by this step, no part of it's corresponding 3D patch, in 3D pixel space, must reside in the view volume.

A conservative technique is used to determine whether or not a u-v subrectangle should be designated as pruned. In other words, while it is acceptable to miss a pruning opportunity, it is not acceptable to prune a u-v subrectangle that shouldn't be pruned. A missed pruning opportunity is acceptable because it does not affect the output image and only potentially increases the processing time of the algorithm. In fact, this pruning step is entirely an optimization step.

Determining whether or not a patch is at all located within the view vol-

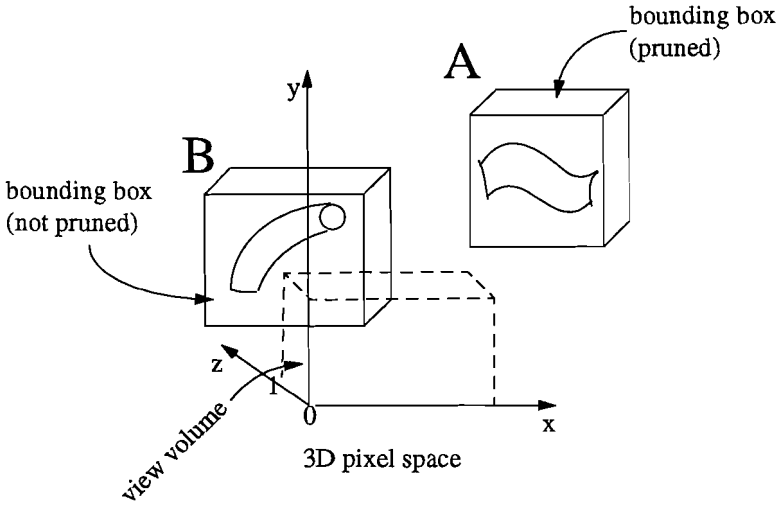


Figure 2.7: Example bounding boxes

ume is computationally difficult. So instead, a pruning technique commonly used is the **bounding-box** technique. Figure 2.7 illustrates two patches with their corresponding bounding boxes. Detecting whether or not a bounding box intersects the view volume is a relatively small computational problem to solve, see [FvDFH90]. A bounding box is any box that completely encases the patch being considered, but for pruning determination, the smallest possible bound box is desirable. The calculations for determining the smallest bounding box varies for each type of patch, consult [FvDFH90] for details.

In Figure 2.7, two patches and their corresponding bounding boxes are shown. The u-v subrectangle corresponding to patch "A" is designated as pruned because its bounding box does not intersect the view volume. As a result of using a conservative pruning technique, patch "B's" u-v rectangle is designated as not pruned even though patch "B" itself does not at all reside in the view volume. This bounding-box technique relies totally on whether or not a patch's corresponding bounding box intersects the view volume. Patch "B" is thus an example of a missed pruning opportunity that sometime occurs when using a conservative pruning technique.

There exists more computationally expensive pruning techniques than the bounding-box technique that will always accurately determine whether or not any part of a patch resides in the view volume. These are not used in this implementation. There is a trade-off between more intelligent pruning

techniques, which always detects situations that can be pruned but takes significant processing time, and simple pruning, which takes less processing time but may require more iterations of the overall PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm. A simple pruning technique may result in a missed opportunity to designate a u-v subrectangle as pruned. This will, in turn, result in the creation of additional (smaller) u-v subrectangles that will be processed in the next iteration of this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm. See step 7, page 45. This, however, may not result in any additional processing time if another iteration of this algorithm would occur any way—remember, all u-v subrectangles are processed in parallel.

Step 4: Prune u-v subrectangles based on fulfilling the modeler specified sampling rates

This step only applies to non-pruned u-v subrectangles. And, all of these non-pruned u-v subrectangles are processed in parallel.

Renderers based on the Renderman Interface standard allow the modeler to specify the **shading rate**. Informally, the shading rate specifies the density of points to be sampled on the surface of a primitive geometric object during the rendering process.

In this RenderMan implementation, the sampled points are the four corner points of each and every u-v subrectangle generated by this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm. The four corner points of a u-v subrectangle are indicated as points "a", "b", "c", and "d" in Figure 2.6. These u-v-sampled points are mapped to their corresponding 3D cartesian space points on the surface of the primitive geometric object being rendered. Sampled points are rendered, i.e., their projected surface color is determined. An example of mapping a point from u-v space to 3D cartesian space to 3D pixel space is shown in Figure 1.2. This mapping is described in step 2, page 30.

The shading rate dictates the maximum distance allowed between sampled points. The maximum distance is specified in terms of **2D pixel space**. The mapping of a point from 3D pixel space to 2D pixel space is accomplished by simply ignoring the z-component of the 3D-pixel-space point.

There is another important related RenderMan concept: the modeler

definable **sampling rate**[5]. The sampling rate is similar in concept to the shading rate and is applicable only if the sampling rate specifies more resolution than is afforded by the shading rate.

After the shading rate requirement has been fulfilled, the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm continues to select (sample) points, i.e., breaking up u-v subrectangles into smaller u-v subrectangles, in order to fulfill the more demanding, in terms of resolution, sampling rate requirement. See step 7, page 45. However, the points selected to fulfill the sampling rate are not subjected to the surface shader for projected surface color determination, instead, they are subjected to an interpolation calculation of already shaded surface points.

For a sequential machine, the use of the sample rate to extend the resolution provided by the shading rate is a time saving feature; because, typically, the interpolation calculation is much less computationally intensive than a surface shader calculation. But this time savings doesn't necessarily happen in our parallel, SIMD, implementation. Since sets of points are rendered in parallel, if part of the set requires shading rate (surface shader) rendering and the remaining part of the set requires sampling rate (interpolation) rendering both subsets of points have to be rendered, one after the other, in the SIMD model. Only if there are no shading rate points left to be rendered will there be a savings in time. Since there is a semantic difference between shading rate rendering and sampling rate rendering, both are supported in this parallel implementation.

Before entering this step, the projected surface colors of points "a", "b", "c", and "d" of Figure 2.6 have been determined and stored in the object-image array.[6] See step 5, page 40 and step 6, page 42. And, all of the points labeled in Figure 2.6 have been mapped to their corresponding 3D pixel-space points.

If a u-v subrectangle maps into 2D pixel space in such a way as to satisfy the requirements of both the shading and sampling rate then no further

[5]Specified in RenderMan by the RiPixelSamples routine

[6]As mentioned on page 22, when our RenderMan system invokes this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, it inputs a single u-v subrectangle: the primitive object's **associated** u-v rectangle. It is required that before the RenderMan system invokes the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, the projected surface colors of points "a", "b", "c", and "d" of Figure 2.6 of the associated u-v rectangle are determined and stored in the object-image array.

processing of this u-v subrectangle is required. Such a u-v subrectangle is designated as pruned.

On the other hand, if the density of sampled points selected from a u-v subrectangle does not fulfill the sampling rate requirement, then the u-v subrectangle will be broken up into smaller u-v subrectangles, and these smaller u-v subrectangles will be then subjected to a recursive call of this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm. See step 7, page 45.

Determining whether or not a u-v subrectangle satisfies either the shading or sampling rate requirement is non-trivial and untractable by a sampling method—if done with 100% accuracy.

A u-v subrectangle represents a portion of the surface of a primitive geometric object. The corner points of a u-v subrectangle are the sampled points that are rendered. A u-v subrectangle that fulfills the shading or sampling rate requirement implies that no two points in the u-v subrectangle map to two points in 2D pixel space that are farther apart than the allowable maximum separation distance dictated by the shading or sampling rate.

It may be thought that determining whether or not the shading or sampling rate is satisfied is a matter of determining whether or not all of the following line segments, whose endpoints are shown in Figure 2.6, $\overline{ab}$, $\overline{bd}$, $\overline{cd}$, $\overline{ac}$, $\overline{ad}$, and $\overline{bc}$ are shorter, when mapped into 2D pixel space, than the maximum allowed separation distance between sampled points dictated by the shading or sampling rate.

But, this isn't the case. Intuitively, one can imagine the u-v subrectangle as a rectangular elastic which is wrapped around the surface of its corresponding primitive geometric object in 3D space. The surface of a primitive geometric object, i.e., a patch, can contain very large bends and twists. With this in mind, it is easy to envision a case where points "a", "b", "c", and "d" of Figure 2.6, when mapped into either 3D cartesian space or 3D pixel space, are close to each other but where point "f" may be far away. Figure 2.8, shows an example of a primitive geometric object whose surface is a section of a torus and whose corresponding u-v points "a", "b", "c", and "d" are close to each other and whose point "f" is far away.

Therefore, it is incorrect to consider only points "a", "b", "c", and "d" when determining whether or not a u-v subrectangle is small enough to fulfill either the shading or sampling rate requirement. In fact, the imagined elastic u-v subrectangle, could be bent in very peculiar ways. And, no amount of sampling could make sure that there isn't a spike bend in the elastic, some-

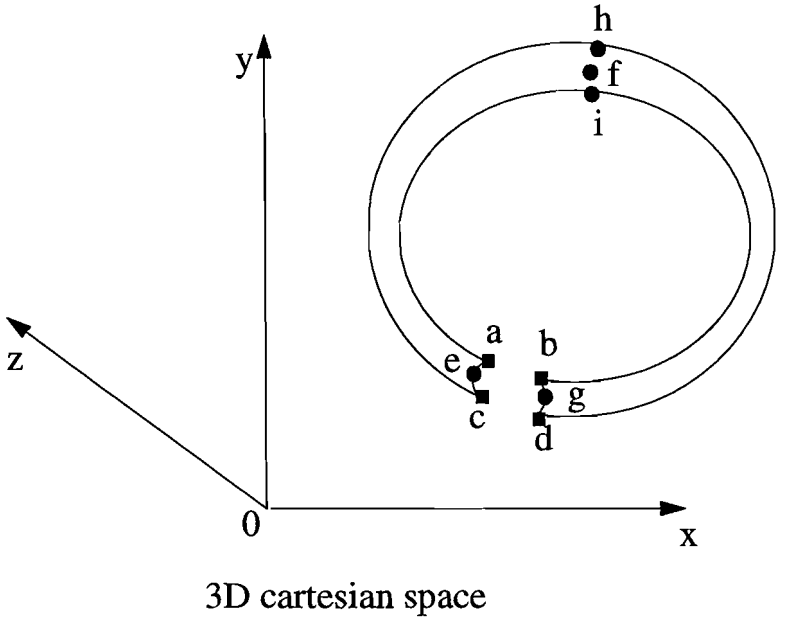


Figure 2.8: Torus patch with corresponding u-v subrectangle points indicated

where. However, pragmatically speaking, the primitive geometric objects generated by many modelers are well behaved, that is, not having these dramatic spike bends. The following engineering solution is used to determine whether or not the shading or sampling rate requirements are fulfilled.

The 2D pixel space points corresponding to all the u-v space points labeled in Figure 2.6 will be used in determining whether or not a u-v subrectangle fulfills the sampling rate requirement. If each of the eight 2D pixel space distances, shown below, is less then the maximum separation distance between sampled points specified by the sampling rate, then the u-v subrectangle will be designated as pruned, that is, it will not be further processed by this algorithm.

$$|\overline{ah}| + |\overline{hb}| \tag{2.1}$$
$$|\overline{ae}| + |\overline{ec}| \tag{2.2}$$
$$|\overline{af}| + |\overline{fd}| \tag{2.3}$$
$$|\overline{dg}| + |\overline{gb}| \tag{2.4}$$

$$|\overline{bf}| + |\overline{fc}| \tag{2.5}$$
$$|\overline{di}| + |\overline{ic}| \tag{2.6}$$
$$|\overline{ef}| + |\overline{fg}| \tag{2.7}$$
$$|\overline{hf}| + |\overline{fi}| \tag{2.8}$$

Otherwise, for each non-pruned u-v subrectangle, a similar test for the less-demanding shading rate is performed and the result of whether or not the shading rate requirement is fulfilled is recorded.

Step 5: Determine the projected surface colors of points "e", "f", "g", "h", and "i"

This step only applies to non-pruned u-v subrectangles. And, all of these non-pruned u-v subrectangles are processed in parallel.

This step determines the projected surface color of each of the following points "e", "f", "g", "h", and "i"; these points are shown in Figure 2.6.

The u-v subrectangles processed during this step are divided into two categories; those whose points will be subjected to interpolation (sampling rate) rendering and those whose points will be subjected to surface shader (shading rate) rendering. Which category a u-v subrectangle is in is determined by step 4, page 36.

In this SIMD implementation, all the interpolation (sampling rate) rendered u-v subrectangles will be processed in parallel; and then, all the surface shader (shading rate) rendered u-v subrectangles will be processed in parallel. If we were using a MIMD machine, both interpolation (sampling rate) and surface shader (shading rate) rendering could overlap in time.

For a point rendered by the interpolation process (sampling rate), the point's color is based on a linear interpolation[7] of the color of other already calculated points.

The table below indicates which points, as shown in Figure 2.6, the interpreted points are based on.

[7]There is one exception, point "f" is based on a bilinear interpolation.

interpreted point	*based on*
e	a,c
g	b,d
h	a,b
i	c,d
f	a,b,c,d

For a point rendered by a surface shader (shading rate), the point's color is calculated by a modeler defined surface shader. As mentioned previously, a surface shader can utilize other types of shaders, such as light source shaders, while it is rendering a point.[8] In this implementation, the modeler's choice of shaders to correspond to a primitive geometric object is restricted to the following.

Surface Shaders	*Light Source Shaders*
Constant surface	Ambient light source
Texture-map surface (non-standard)	Distant light source

Descriptions of the above mentioned standard shaders, namely, constant surface, ambient light source, and distant light source can be found in [Ren89] or [Ups90]. The **texture-map** surface shader is specific to our implementation. It takes as a parameter the name of a texture map, and it utilizes the named texture map and the specified light sources in its color determination of a point.

In this implementation, these shaders are hard-coded. The optional RenderMan C-like SHADER language capability is not implemented in this project. The optional SHADER language capability is not required by the RenderMan Interface specification. See [Ren89]. The SHADER language provides a modeler with full flexibility in specifying the surface shading process. Although this implementation doesn't support the SHADER language, it is designed to allow for a SHADER language extension to this project.

After this step, the projected surface color of each of each point indicated in Figure 2.6 is calculated and stored.

[8]Rendering based on shaders is explained in the "Introduction" section of this thesis.

Step 6: Send color and 3D pixel-space-coordinate values to appropriate pixels, in the object-image array, utilizing filtering

This step only applies to non-pruned u-v subrectangles. All non-pruned u-v subrectangles are processed in parallel.

The object-image array is a 2D array. The dimensions of this 2D array corresponds to the output image size, in pixels, requested by the modeler. The object-image is where the result of rendering a single geometric primitive object is stored. Each top-level invocation of this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm renders a single geometric primitive object and the result is stored in the object-image array.

Each pixel of the object-image is represented by an element of the object-image array. Each element of the object-image array is assigned its own processor. In the C*/CM-200 system used in our implementation, each C* array element is automatically assigned its own processor.

At this point in the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, processors (implemented as elements of a C* array) have stored, in their local memories, values corresponding to each of the u-v subrectangle points "e", "f", "g", "h", and "i". These corresponding values are the projected surface color, see step 5, page 40 and the 3D pixel-space coordinate, see step 2, page 30. Each processor send its above-mentioned values to an appropriate element (processor) of the object-image array. The appropriate element is determined by rounding down the x and y components of the 3D pixel-space coordinate corresponding to a u-v subrectangle point. These rounded-down values, which are integers, comprise the address of an element (processor) of the object-image array. However, it makes sense for only only those processors that contain u-v subrectangle points that map into the view volume, as defined on page 30, to send their values to the object-image array.

Each receiving object-image array element (pixel) keeps track of a **weighted average of color and z-depth values** sent to it during the execution of a top-level call of this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm. As mentioned previously, the z-depth value is the z-component of the 3D pixel-space coordinate. The weighted average calculation is made in accordance with a modeler specified filter function. A filter function is used by a renderer to prevent the appearance of jaggies, i.e. staircasing, artifacts in the digital output image that could otherwised be produced during

the rendering process. In addition, when rendering a sequence of images for animation, temporal artifacts can also be prevented by the use of filtering. See [FvDFH90] for details on filtering. Typical filter functions weigh, with more importance, the values of points that, in 2D pixel space, lie closer to the center of a pixel than the values of points lying farther away. This implementation allows the modeler to specify any of the standard RenderMan filter functions, namely: triangle, box, catmull-rom, sinc, and gaussian.

If the modeler specified filter width is wider than one pixel, then the u-v points must send their values to not only the pixel of the object-image array to which it is mapped, but also, to neighboring pixels, as well. Typically, the height and width of a filter is between one and three pixels in length. Therefore, if the filter width and height are the same, which is typical, then the number of pixel processors that are sent values from each of the points "e", "f", "g", "h", and "i" is

$$\begin{array}{ll} (\lceil \text{filter width} \rceil)^2 & \text{for odd } \lceil \text{filter width} \rceil \\ (\lceil \text{filter width} \rceil + 1)^2 & \text{for even } \lceil \text{filter width} \rceil \end{array}$$

where "filter width" is the number of pixels covered by the width (or height) of the filter. For example, if the filter width is 1.3 pixels; then $\lceil \text{filter width} \rceil$ is 2.0, the even case; and therefore 9 pixel processors are sent values from each active u-v subrectangle point.

On the C*/CM-200 SIMD system, used in our implementation, each u-v point's processor can send information to only one designated pixel (processor) at a time; therefore multiple sends are necessary. Multiple designated (non-broadcast) sends require multiple iterations of C* statements on the CM-200.

It has been previously mentioned that only those u-v subrectangle points that map (project) into the view volume send values (color and 3D pixel-space coordinate) to the pixels stored in the object-image array. This is not strictly true. If the filter width is wider than one pixel, a u-v subrectangle point that maps to a 3D pixel-space point outside of the bounds of the view volume but near an edge of the view volume might need to send information to one or more pixels of the object-image array. Therefore, even when a u-v point doesn't send values to its own pixel, because it is out of bounds, it may be involved in sending information to neighbor pixels in order to support a

more than one-pixel wide filter.

In addition, some of the points of the u-v subrectangles, "e", "f", "g", "h", and "i", do not send their point-related values to the object-image array. Step 7, page 45, determines which u-v subrectangle points do not send their values to the object-image array. Such **don't-send** points are designated as such because near-by u-v subrectangles, generated by this algorithm, contain identical u-v points. Sending the values of identical u-v points, more than once, to the object-image array would distort the color of a given pixel in favor of those more than once represented u-v points.

RenderMan allows the user to specify rendering on either one or both sides of a primitive geometric object. If rendering on one side is requested, RenderMan only renders the outside of the geometric primitive object. In this case, this step only sends u-v points whose outside surface is facing the synthetic camera; this is determined by the direction of the u-v point's **3D-pixel-space surface-normal vector** (to be discussed below). A running tally of the total number of points sent to each pixel element (processor) is also recorded, in this step, for use by step 1, page 23. Step 1 uses this information to determine the final α value of a pixel of the object-image array.

In our implementation, the 3D-pixel-space surface-normal vector of a point on the surface of a primitive geometric object (patch) is calculated as follows. As mentioned on page 22, the surface of a primitive geometric object is specified by a set of parametric polynomials in two variables. This set of parametric polynomials in two variables specifies the mapping from u-v space to 3D space. In our implementation, we use three parametric polynomials whose two variables are "u" and "v", designating a u-v-space point, and whose three polynomial expressions evaluate to "x", "y", and "z", designating a 3D-cartesian-space point. Two vectors tangent to the surface, at the given point, are then calculated. They are calculated by taking two partial derivatives (i.e., two partial derivatives for each of the three polynomials), one with respect to "u" and one other with respect to "v" and then evaluated at the u-v-space point being processed. The cross-product of these two tangent vectors result in the surface-normal vector in 3D cartesian space. By mapping two points of a vector in 3D cartesian space to their corresponding points in 3D pixel space, a 3D-pixel-space surface-normal vector is determined. This mapping can be performed by the homogeneous 4x4 matrix described on page 32.

Whether or not a surface point is front facing or back facing, with respect to the synthetic camera, can be determined by taking the dot product of the 3D-pixel-space vector (0,0,-1), a vector directly pointing towards the synthetic camera, and the 3D-pixel-space surface-normal vector just calculated. The dot product will be positive if the 3D-pixel-space surface-normal vector is facing the synthetic camera, thereby indicating a front-facing surface; otherwise, the dot product will be negative conversely indicating a back-facing surface.

If both sides are requested to be rendered, then, in order to facilitate the calculation of a pixel's α value, a running tally of the number of back-facing and front-facing u-v points sent to each pixel is recorded, as well as, the average 2D-pixel-space location and z-depth value of each of these two sets of points (back-facing and front-facing).

The final α and z-depth value for each pixel is calculated in Step 1, page 23.

Step 7: Create four new u-v subrectangles, from each non-pruned u-v subrectangle, for further processing by a recursive call to this Parallel-Adaptive-Point-Sampling algorithm

This step only applies to non-pruned u-v subrectangles. All non-pruned u-v subrectangles are processed in parallel.

Each non-pruned u-v subrectangle will be the source of 4 new (smaller) u-v subrectangles. See Figure 2.9. A list of all these newly generated u-v subrectangles is used as an argument to a recursive call, made during this step, of this PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm.

As indicated in Figure 2.9, some of the u-v points on the new u-v subrectangles are designated, by this step, as "don't send". The projected surface color and 3D-pixel-space information corresponding to these u-v points should not be sent, in Step 6, page 42, to the object-image array because a sibling u-v subrectangle contains the same u-v point.

End of PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm.

Implementation and design notes on the Parallel-Adaptive-Point-Sampling algorithm

- Since this project's CM-200 possesses a small number of processors and a limited-memory capacity, a modeler's request for an output image containing a large number of pixels is transformed, by our renderer, into several requests for smaller sub-images. These smaller sub-images are rendered independently and then assembled into one large output image.

- In this SIMD-based PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, even though the rendering process for each primitive geometric object utilizes parallel processing, every primitive geometric object is rendered one at a time.

  However, if a MIMD machine were available, this SIMD-based PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm could be modified to render several primitive geometric objects in parallel. In a large enough MIMD machine, all the geometric primitives of the scene could be rendered in parallel. On such a large MIMD machine, a parallel compositing operation taking O(log <number of geometric primitive objects>) time could be performed. Thus, this large MIMD machine could support an extremely fast RenderMan renderer. Photorealistic virtual reality rendering could be realized.

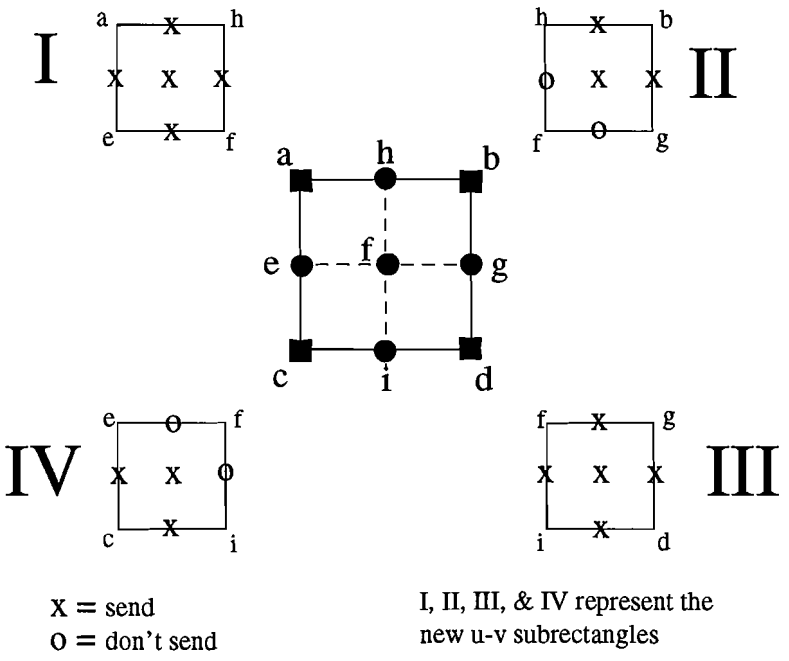


Figure 2.9: New u-v subrectangles

Chapter 3

Implementation Results and Future Work

This project's RenderMan system is implemented on Thinking Machine's C*/CM-200 system, a SIMD computer. This RenderMan implementation is designed to be easily ported to other parallel computer systems—SIMD or MIMD. The source files comprising this system are divided into two categories: (1) scalar sources; ANSI C, lex, and yacc and (2) parallel sources; C*. The majority of source files in this implementation are scalar.

C* is a variant of the ANSI C programming language. C* contains all of the features of the ANSI C programming language and in addition contains constructs for parallel operations on arrays. Since portability of the source files is a major design criterion of this project, only those C* features that are available on most other parallel computer systems are used in this project.

A sample image rendered using this project's RenderMan system, which includes the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, is shown in Figure 3.3. It contains two texture-mapped patches. The texture maps are shown in Figure 3.1 and Figure 3.2. Each patch depicts a section of the surface of a torus. The RIB code used to generate the image is contained in the Appendix.

The following table lists the number of u-v subrectangles generated for each patch during each iteration of the major loop (or recursion) of the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm.

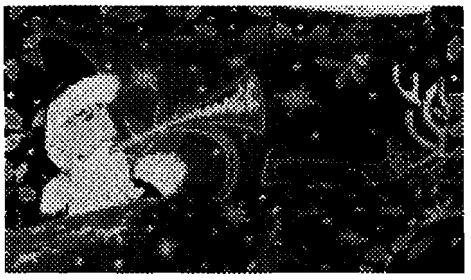


Figure 3.1: Texture map of santa motif (color image)

Figure 3.2: Texture map of flower motif (color image)

	number of u-v subrectangles generated	
loop number	santa-motif patch	flower-motif patch
1	1	1
2	4	4
3	64	64
4	256	256
5	1024	1024
6	4096	4096
7	16384	16384
8	0	13280
9	—	0

There were two RenderMan systems built during this project. One which uses a pre-selected evenly-distributed set of 36,000 u-v points as the basis for sampling, and the other one which uses the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm described in this thesis. Of course the former system does not correctly observe the modeler specified shading rate—a big shortcoming. The former system performed about 5-10 times faster than the com-

Figure 3.3: Image Produced by the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm (color image)

mercially available (scalar) RenderMan system; and the latter system, using the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm performed about 5 times slower.

These performance numbers indicate that the first of the two RenderMan systems built during this project—the one using 36,000 pre-slected u-v points—efficiently uses this project's SIMD computer. And, the performance of the PARALLEL-ADAPTIVE-POINT-SAMPLING-based renderer can be easily explained, as will be done (see below), and easily overcome.

This project's SIMD computer is a C*/CM-200 system containing 4K of SIMD processors. The processors themselves aren't particularly fast. To measure the relative speeds of the floating point operations of this project's scalar machine, namely a Sun Microsystem Inc. Sparc station with that of an individual processor of the C*/CM-200 system, timings were taken for the task of performing 2 million floating point operations on each of the two processors. On the scalar machine the floating point operations were performed on a scalar double-precision variable and on the C*/CM-200 the floating point operations were performed on a 4K array of double-precision elements. The scalar machine took 2.03 seconds to execute this task while the C*/CM-200 system took 172.580 seconds to execute this task. This, then, measures the C*/CM-200 processors' floating point operation speed to be

approximately 85 times slower than this project's scalar processor's floating point operation speed.

Much of the computation done in a renderer is floating point operations. Thus, although there are 4K of processors in this project's SIMD computer, each of the 4K processors is 85 times slower than our scalar processor, so the upper bounds on the time improvement of our SIMD-implemented renderer is (4096/85), i.e. 48.18, times over the scalar version of the renderer. The first renderer, using pre-selected points and obtaining a 5-10 time speed improvement over the scalar computer, approaches this performance number. And, in addition, this renderer has the burden of data movement between processors and a host machine that a scalar version of a renderer does not have.

The goal of the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm is to provide the renderer with the ability to fulfill the modeler specified shading rate requirement and at the same time minimize any time-performance impact this additional feature might incur. At a first glance, the performance of the PARALLEL-ADAPTIVE-POINT-SAMPLING-based renderer, 5 times slower than the scalar version, looks as though the overhead of this algorithm is indeed too great for the functionality it provides. However, as will be explained below, the performance hit can be almost entirely ascribed to the code being written in a portable style.

The approximately 50 times performance hit observed in the renderer after inserting the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm into it has little to do with the performance of the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm per se but rather reflects the implementation choices made for portability reasons. These issues, listed below, will be discussed in more detail in the paragraphs following this list.

- the number of initially selected u-v subrectangles (at least a 5 time speed up)
- oversampling
- the use of structs (at least a 5 time speed up)

In general, high performance on the C*/CM-200 is obtained by full parallel utilization of all 4K processors. As shown by the number of u-v subrectangles generated per each loop of the sample image rendered, the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm underutilizes the 4K of available

processors during the first five loops. Thus, for a C*/CM-200 implementation, this suggests modifying the algorithm to initially pick a much larger data set of sampled points, i.e., 4K samples.

Another characteristic of this algorithm is its tendency to oversample. While not a RenderMan semantic problem, it decreases the performance. A u-v subrectangle that is just a little too large to satisfy the sampling rate results in the creation of 4 new u-v subrectangles, and each new u-v subrectangle contains 5 new sample points. In practice, this results in taking 20-25 times the number of samples required. This suggests that a more judicious method of selecting new sample points is desirable.

The current implementation uses the C-language **struct** construct to store together all of the information related to the points, indicated in Figure 2.6, of a u-v subrectangle. A struct is C's version of a record. In this implementation, when processing the point information found in the u-v subrectangles' structs, all u-v subrectangles are, indeed, processed in parallel; however, a separate iteration is used for each type of u-v subrectangle point, indicated in Figure 2.6. For example, all of the "a" points of all the u-v subrectangles are processed in parallel in one iteration, followed by, an iteration to process all of the "b" points, etc. This is a result of the C* language restriction on accessing fields of a struct, i.e., only one field can be designated at a time. An alternate, less descriptive, data structure could be used to allow the parallel processing of all of the points of all of the u-v subrectangles.

Instrumenting the PARALLEL-ADAPTIVE-POINT-SAMPLING algorithm, on the C*/CM-200, reveals that 32.9% of the execution time is spent in the low level C*/CM-200 routine, **__CMI_wait_until_read_data_available**, and an additional 24.8% of the execution time is spent in the low level C*/CM-200 routine, **__CMI_wait_for_room_in_ififo**. These routines are all part of the C*/CM-200 runtime package; they are not directly called from the source files. The names of these routines suggests that much of the execution time used by the C*/CM-200 for this algorithm is spent on communication between processors. Therefore, fine tuning of the data-set memory layout, in order to minimize communication distances on the underlying C*/CM-200 hypercube would seem to be in order—truly a machine-specific optimization.

This project's C*/CM-200 system is considered to be a small-sized machine, in terms of memory and number of processors. The current implementation runs out of CM-200 memory, on our C*/CM-200 system, when an

attempt to process a large-sized, in terms of 2D pixel space, patch is made. The implementation needs a larger machine for better performance.

Future work for this RenderMan-based renderer includes porting the code to a new suitable parallel-machine platform, fine tuning this system for high-performance, and implementing the full set of features of RenderMan.

The usefulness of the current implementation is twofold: (1) as a test-bed for parallel-machine builders and (2) as an evolutionary step towards building high-performance photorealistic renderers. First, as a test-bed, the RenderMan system provides an important application which embodies a high degree of inherent parallelism. Secondly, the need for a step towards high-performance photorealistic rendering can be seen in the current time-demanding uses of photorealistic rendering found in the motion-picture, medical imaging, architecture, and scientific visualization industries.

Appendix A

RIB code for Figure 3.3

```
version 3.03

# RIB code for the image of two texture-mapped patches
# shown in the "Implementation Results and Future Work" chapter.

Display "donut.tiff" "tiff" "rgb"
Format 180 180 1
Projection "perspective"

PixelFilter "box" 1.0 1.0
ShadingRate  1.0
Sides 1

WorldBegin
   Translate 0.0 0.0 650

   TransformBegin
     Surface "texture_mapper"  "mapname" "santa"
     Translate -200.0 0 0
     Torus 400.0 #majorradius
           155.0 #minorradius
          -150.0 #phimin
           20.0  #phimax
           70    #thetamax
```

```
   TransformEnd

   TransformBegin
     Surface "texture_mapper"  "mapname" "flowers"
     Translate 200 0 0
     Rotate 180 0 0 1
     Torus 400.0  #majorradius
           155.0  #minorradius
          -150.0  #phimin
            20.0  #phimax
            75.0  #thetamax
    TransformEnd
WorldEnd
```

Bibliography

[FT90] Ross L. Finney and George B. Thomas, Jr. *CALCULUS.* Addison-Wesley Publishing Company, 1990.

[FvDFH90] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice.* The Systems Programming Series. Addison-Wesley Publishing Company, second edition, 1990.

[JGMH88] Kenneth I. Joy, Charles W. Grant, Nelson L. Max, and Lansing Hatfield. *Tutorial: Computer Graphics: Image Synthesis.* Computer Society Press: IEEE, 1988.

[PD84] Thomas Porter and Tom Duff. Compositing digital images. In *Proceedings of SIGGRAPH '84*, pages 253–259. Association for Computing Machinery, Inc., 1984.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* Texts and Monographs in Computer Science. Springer-Verlag, 1985.

[Ren89] The Renderman Interface: Version 3.1. Pixar, 3240 Kerner Blvd., San Rafael, CA 94901, September 1989.

[Ups90] Steve Upstill. *The RenderMan Companion.* Addison-Wesley Publishing Company, 1990.