TEXTURE MAPPING WITH RAY TRACING

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BİLKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

by
Uğur Akdemir
October, 1993

TEXTURE MAPPING WITH RAY TRACING

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

by

Uğur Akdemir

October, 1993

Uğur Akdemir

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Prof. Bülent Özgüç (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Varol Akman

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. Özgür Ulusoy

Approved for the Institute of Engineering and Science:

[illegible]

Prof. Mehmet Baray
Director of the Institute

ABSTRACT

TEXTURE MAPPING WITH RAY TRACING

Uğur Akdemir
M.S. in Computer Engineering and Information Science
Advisor: Prof. Bülent Özgüç
October, 1993

By using texture generation and global illumination techniques, it is possible to produce realistic computer images. Currently, ray tracing is one of the most popular global illumination techniques due to its simplicity, elegancy, and easy implementation. In this thesis, texture mapping techniques are used with ray tracing to generate high quality visual effects. The implementation of the mapping process is presented and an approach for combining prefiltering techniques with ray tracing is introduced. General sweep surfaces produced by Topologybook are used for modeling.

Keywords: texture mapping, ray tracing, area sampling, filtering, summed area tables, sweep surfaces, Topologybook

ÖZET

IŞIN İZLEME YÖNTEMİYLE DOKU KAPLAMA

Uğur Akdemir
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Danışman: Prof. Dr. Bülent Özgüç
Ekim, 1993

Doku kaplama ve küresel ışıklandırma yöntemleri kullanılarak gerçeğe uygun bilgisayar görüntüleri oluşturmak olasıdır. Günümüzde, ışın izlemesi, basitliği, sonuçlarının güzelliği ve kolay uygulanabilirliğinden dolayı en yaygın küresel ışıklandırma yöntemlerinden birisidir. Bu tezde, yüksek kalitede görsel etkiler yaratabilmek için doku kaplama yöntemleri ışın izlemesi ile birlikte kullanılmıştır. Doku kaplama yöntemlerinin uygulanması ve ön-süzme yöntemlerinin ışın izleme yöntemiyle birlikte kullanılmasını sağlayan bir yaklaşım sunulmuştur. Modelleme için Topologybook tarafından üretilmiş süpürülmüş yüzeyler kullanılmıştır.

Anahtar Sözcükler: doku kaplama, ışın izlenmesi, alan örnekleme, süzme, toplanmış alan tabloları, süpürülmüş yüzeyler, Topologybook

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my supervisor Prof. Bülent Özgüç for his guidance, suggestions, and encouragement throughout the development of this thesis.

I would like to thank to Mr. Veysi İşler for his suggestions. I would also like to thank to Mr. Ahmet Arslan for his explanations and helps.

Finally, I would like to thank to Assoc. Prof. Varol Akman and Asst. Prof. Özgür Ulusoy for their comments on the thesis.

Contents

List of Figures

Chapter 1

Introduction

One of the most important aims of computer graphics is to produce high quality realistic images. Realistic image synthesis must achieve two goals:

- advanced modeling
- realistic rendering

To achieve advanced modeling, we used a tool called T♭ ('Topology♭ook'), developed at Bilkent University [2, 3, 4, 5, 8], for modeling sweep surfaces. In T♭ objects are created by sweeping a 2D contour curve that can vary around a 3D trajectory curve. Contour and trajectory can be Bézier or free curves. A number of interesting objects can be created by using T♭. In our implementation, we subdivided the sweep surfaces into triangles.

Modeling every minute detail of an object to give realistic effects is a very difficult and inefficient process. In order to overcome this difficulty, texture mapping is introduced. Texture mapping makes images more complex and interesting at a relatively low cost. Texture becomes an inherent part of an object and thus increases the realism. Texture mapping allows us to add real life details to computer generated images.

In order to achieve realistic rendering of 3D scenes, global illumination techniques must be used. Early rendering techniques such as Gouraud and

Phong shading model local illumination and thus are not adequate for realistic image synthesis. Currently, ray tracing and radiosity algorithms are used for global illumination. Ray tracing is both a visibility and shading algorithm, but radiosity is just a shading algorithm. Ray tracing is the most popular technique in image synthesis. It is easy to implement a ray tracer that handles specular reflection, refraction, shadows, and hidden surface removal, all at once.

In this thesis, we combined ray tracing with texture mapping. We implemented two kinds of texture mapping on sweep surfaces: 2D and 3D (procedural).

Procedural texture mapping has the advantage that it is easily applicable to any kind of surface. Procedural texturing functions do not depend on the surface geometry and the surface coordinate system. Therefore, procedural mapping gives good results for complex general sweep objects. We have embedded a few procedural mapping routines in our system in order to increase the realism of the scenes combined with 2D mapping.

In 2D texture mapping, a texture or an image is fitted onto a 3D surface. 2D texture mapping is the process of mapping colors from a digital image onto a surface. Hence, it is possible to map photo-realistic images on 3D objects by 2D texture mapping. Unlike procedural texturing, it causes severe aliasing problems unless filtering is used.

A way of representing texture for easy antialiasing is prefiltering. A continuous domain must be sampled for filtering applications. Unfortunately, classic (naive) ray tracing is a point sampling technique that creates difficulties for filtering. In this thesis, we propose a method to apply a prefiltering technique, namely repeated integration filtering, in ray tracing by using the same intersection calculations as in classic ray tracing.

Besides the aliasing artifacts, one severe disadvantage of ray tracing is long execution times. To accelerate the algorithm, we have implemented a spatial subdivision technique.

With all the extensions we have developed for the ray tracing system in this thesis, it is possible for a user to model almost any kind of object and ray trace them with improved realism by using 2D and 3D textures in a reasonable amount of time.

An outline of the thesis is as follows:

In Chapter 2, procedural texture mapping is introduced. In Chapter 3, 2D texture mapping along with our implementation is given. In Chapter 4, area sampling in ray tracing is discussed and an algorithm is proposed. In Chapter 5, an acceleration technique, namely 3DDDA algorithm, is introduced. Finally, a conclusion with possible future extensions is offered.

Chapter 2

Procedural Texture Mapping

Procedural texture mapping has been developed as a complementary alternative to 2D texture mapping. As we will see in Chapter 3, by 2D texture mapping it is possible to map a 2D digitized picture onto a surface. Although photo-realistic images can be mapped onto computer generated objects by 2D texture mapping, this has the following disadvantages:

- 2D texture mapping needs a huge database and causes severe aliasing effects unless it is filtered.
- It is hard to give the appearance that complex objects are carved out of some material.
- Textural discontinuities are produced at the surfaces.

Procedural texture mapping does not have the above disadvantages and because of this, it is widely used in computer graphics to generate highly realistic visual effects at a low cost. The main advantage of procedural texture mapping over 2D texture mapping is that any object can receive the texture in a natural way and no discontinuities occur. Also procedural texture mapping does not cause severe aliasing artifacts.

As the name "procedural texture mapping" implies, a procedure or a function is used to define a texture. One trivial way of giving the objects an appearance that they are carved out of some material is to assign color values to each voxel that the bounding volume of an object occupies. Storing a color value for each voxel requires a large amount of memory and this procedure is almost never used. Instead, new functions and tables are defined for a 3D field. An object in 3D space is placed into this field and the intersection is found. This method has been reported simultaneously by Perlin [37] and Peachey [36], and the name "solid texture" was given. Solid texturing functions do not depend on surface geometry and surface coordinate system.

Formally, 3D texture mapping can be defined as follows [23]:

Let τ : $\mathbf{TS} \rightarrow \mathbf{CS}$ be a function mapping each (u, v, w) in a texture space $\mathbf{TS} \subseteq \mathbf{R}^3$ to a color space $\mathbf{CS}$, where τ defines a texture volume.

The name "3D texture mapping" comes from the mapping domain. It is also possible to apply "1D texture mapping". By modulating the diffuse coefficient as a function of angle between the light vector and surface normal, thin film interference can be simulated [44]. This is an example of 1D texture mapping where only one dimension, the angle, is modulated.

2.1 The Noise Function

Perlin suggested a 3D *noise* function to be used for procedural texture mapping applications [37]. A 3D field is divided into 1 unit-cube cells. By using this 3D *noise* function, a large variety of interesting texture effects can be given. Perlin gives some interesting applications of the *noise* function to be used for procedural texture mapping.

The *noise* function takes a 3D vector as input and returns a scalar value. It has the following properties:

- Statistical invariance under rotation
- A narrow bandpass limit in frequency
- Statistical invariance under translation

A 3D grid cell and a point lying inside this cell is shown in Figure 2.1. The

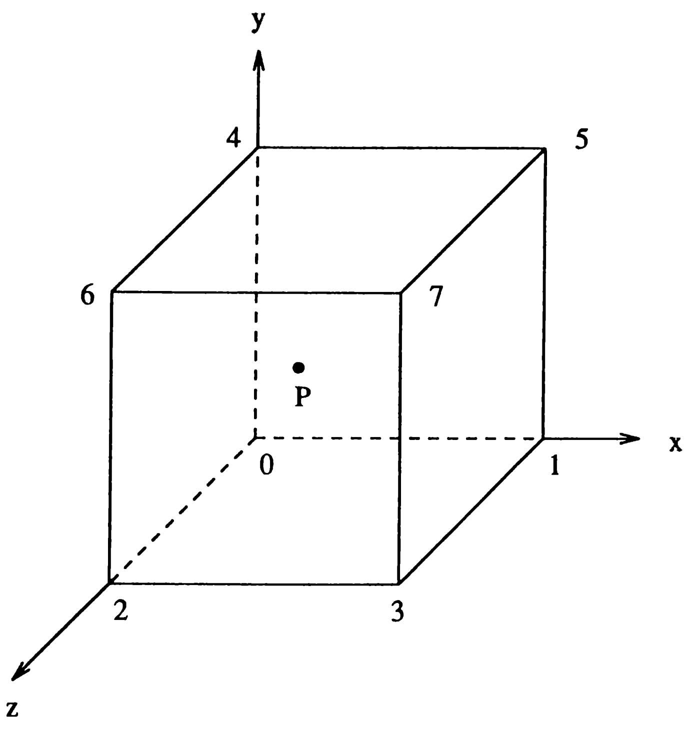

Figure 2.1: Interpolation of noise

value of the point inside this cell, vP, is found by interpolating the random values assigned to the eight vertices of the cell:

```
ox = P_x - v0_x
oy = P_y - v0_y
oz = P_z - v0_z

n00 = v0 + ox*(v1 - v0)
n01 = v2 + ox*(v3 - v2)
n10 = v4 + ox*(v5 - v4)
n11 = v6 + ox*(v7 - v6)

n0 = n00 + oy*(n10 - n00)
n1 = n01 + oy*(n11 - n01)

vP = n0 + oz*(n1 - n0)
```

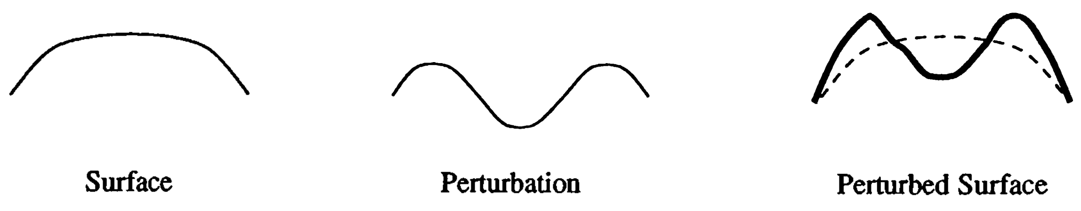

Figure 2.2: Perturbation of a surface

2.2 Surface Perturbation

Unlike most objects in nature, the objects created by geometric modeling techniques are artificially smooth. Adding roughness by using geometric modeling techniques is an expensive operation. In order to give the effect of roughness to objects in a cheap and easy way, a trick called "bump mapping" was introduced by Blinn [14]. Blinn observed that the shade of a surface at a point is directly proportional to the *cosine* of the angle between the normal vector of the surface at that point and the incident light vector. Therefore the effect of roughness can be given by perturbing the normal vector of a surface. Rendering is done by using the perturbed normals.

Bump mapping is different from the other mapping methods in the sense that it does not change the color component of a point on a surface, but modulates the geometry of the surface before shading.

Since the visibility calculations are done with respect to the unperturbed surface and the shading calculations are done with respect to the perturbed surface, silhouette edges of the perturbed surface appear to be smooth. With displacement maps, a 2D height field is used to perturb a surface. Displacement mapping is used when the details of the bumps at the silhouette edges are needed to be seen.

Let $\mathbf{N_s}$ be the normal vector of the surface, $\mathbf{N_p}$ the normal vector of the perturbation function at a point and k a constant. Basically, bump mapping can be achieved as follows:

$$\mathbf{N_s'} = \mathbf{N_s} + k \cdot \mathbf{N_p} \tag{2.1}$$

$$\mathbf{N_s} = \frac{\mathbf{N_s'}}{|\mathbf{N_s'}|} \tag{2.2}$$

A 3D vector valued *dnoise* function can be constructed from the scalar valued *noise* function. *dnoise* is defined by the instantaneous rate of change of the *noise* function along the x, y, and z directions. *dnoise* is used for normal vector perturbation. The C code for *dnoise* is given in Figure 2.3.

By small modulations on the *dnoise* function, it is possible to produce interesting effects. Abram and Whitted [1] suggest giving the effect of the exterior walls of a building by clamping a low frequency *noise* function and then adding high frequency *noise* to roughen the height field.

In our implementation, we created a library for procedural texture mapping applications using the *noise* and *dnoise* functions.

```
void
initialize_dnoise()
{
   int          x, y, z;
   type_3d      vector;

   for  (x = 0; x < MAXNOISE; x++)
      for  (y = 0; y < MAXNOISE; y++)
         for  (z = 0; z < MAXNOISE; z++){
            dnoise_x[x][y][z]=noise[(x+1)%(MAXNOISE+1)][y][z] -
                              noise[x][y][z];
            dnoise_y[x][y][z]=noise[x][(y+1)%(MAXNOISE+1)][z] -
                              noise[x][y][z];
            dnoise_z[x][y][z]=noise[x][y][(z+1)%(MAXNOISE+1)] -
                              noise[x][y][z];
         }
}

type_3d
dnoise(x, y, z)
double          x, y, z;
{
   type_3d      vector;

   vector.x = interpolate_noise(dnoise_x, x, y, z);
   vector.y = interpolate_noise(dnoise_y, x, y, z);
   vector.z = interpolate_noise(dnoise_z, x, y, z);

   return (normalize(vector));
}
```

Figure 2.3: *dnoise* function and its initialization

Chapter 3

2D Texture Mapping

2D texture mapping was developed by Catmull [16] as a cheap way of wrapping a "texture" around an object. The texture that is mapped becomes an inherent part of an object, thus increasing the realism and the complexity of the scenes. Although texture mapping with 3D texture domains is trivial as mentioned in Chapter 2, the effects that can be gained is more restricted than 2D texture mapping. In 2D texture mapping, the source of texture domain can be any 2D image created by scanners, "paint" programs, mathematical functions, etc. The image file is stored as arrays of RGB intensities.

Texture mapping consists of two parts:

- geometric mapping
- rendering

In order to generate high quality realistic scenes, a suitable geometric mapping function and an advanced rendering technique must be chosen. Currently, the most realistic scenes are produced with ray tracing. Hence, as a rendering tool we chose ray tracing.

Different techniques can be used to map a texture onto a surface. Parameterization is one of the easiest ways for 2D texture mapping. If both the texture and the surface are parameterized, then the mapping is trivial.

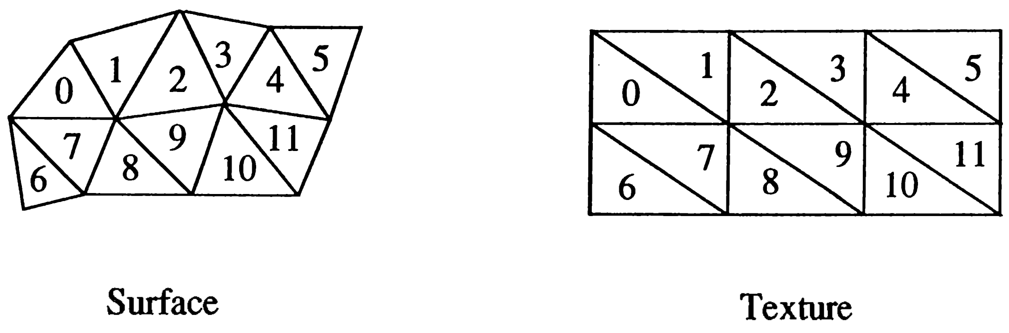

Figure 3.1: 2D texture mapping

A different geometric mapping technique is conformal texture mapping proposed by Fiume *et al.* [22]. Conformal texture mapping allows one to construct a continuous, bijective map from a polygonal texture space to an arbitrary convex polygon.

When a texture is mapped onto an object, it is compressed or stretched to make it take the shape of the object. "Wallpapering" techniques do not work for complex surfaces because discontinuities occur when the surfaces are flattened. Too much interaction is needed for flattening the surfaces. An interactive technique for piecewise flattening of 3D surfaces, leading to non-distorted texture mapping is given in [11]. This technique can only be used for surfaces that are given explicitly by their parametric equations.

Bier *et al.* [12] proposed a 2-part texture mapping. In the first part, texture is embedded in a 3D intermediate surface. In the second part, it is projected onto the target surface in a way that depends only on the geometry of the target object.

Formally, 2D texture mapping can be defined as follows [23]:

Let $\tau : \mathbf{TS} \rightarrow \mathbf{CS}$ be an arbitrary function mapping each (u, v) in the texture space $\mathbf{TS} \subseteq \mathbf{R}^2$ to a color space $\mathbf{CS}$. τ is the texture that is mapped onto the texture.

In our 2D texture mapping implementation, a texture is fitted into a predefined area of a 3D object. The 2D texture mapping implemented is based on triangle-to-triangle mapping. The surface constructed by triangles and the subdivision of the texture into triangles are shown in Figure 3.1. In ray tracing, when a ray hits a triangle that will be 2D texture mapped, the Cartesian

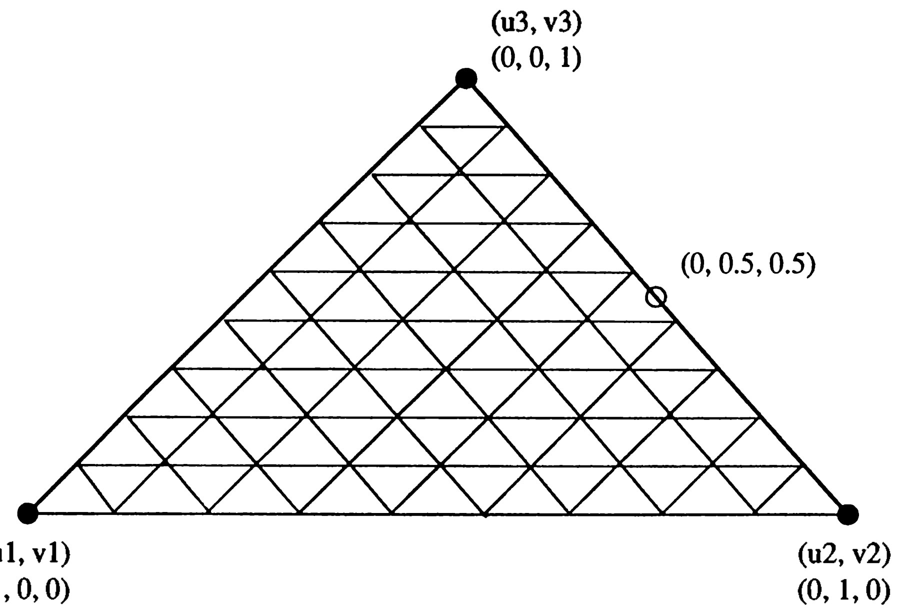

Figure 3.2: Barycentric coordinates in 2D

coordinates of the corresponding point in the texture domain are calculated.

3.1 Geometric Mapping

3.1.1 Barycentric Coordinates

Barycentric coordinates in 2D can be used for mapping an arbitrary triangle to another arbitrary triangle [17]. Barycentric coordinates in 2D for a triangle are shown in Figure 3.2. A point is represented as (u, v) in Cartesian coordinates and as (L_1, L_2, L_3) in Barycentric coordinates. The mapping from Barycentric coordinates to Cartesian coordinates is as follows:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} L_1 \\ L_2 \\ L_3 \end{bmatrix} \tag{3.1}$$

This mapping can be inverted as:

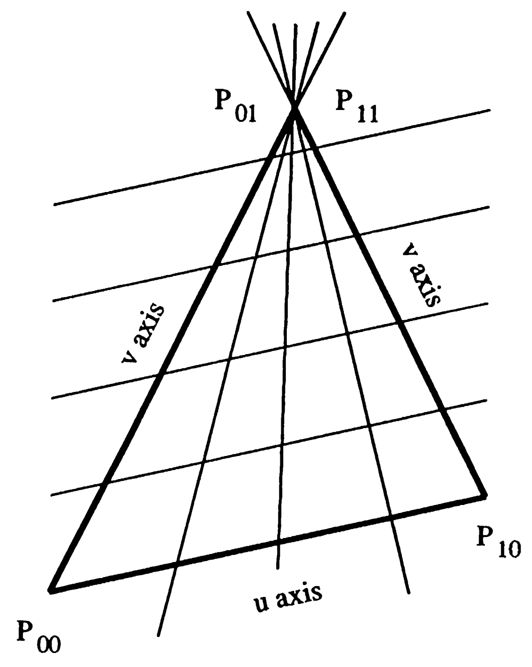

Figure 3.3: Inverse mapping for a triangle

$$\begin{bmatrix} L_1 \\ L_2 \\ L_3 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{3.2}$$

When a ray hits a triangle, the Cartesian coordinates are found at the intersection point.

3.1.2 Extension from Convex Quadrilateral Inverse Mapping to Triangle Inverse Mapping

A convex quadrilateral inverse mapping is given in [27]. The quadrilateral is represented by (u, v) coordinate pairs and the values of (u, v) are in the range $(0..1, 0..1)$. The derivation of this mapping is rather complicated and can be found partially in [27]. The triangle-to-triangle inverse mapping is achieved by doubling the last vertex of the quadrilateral in order to make the algorithm work with four points. The triangle with $u - v$ axes and the doubled vertex is shown in Figure 3.3.

This mapping does not preserve the center of gravity of the triangles. Also at the doubled vertex, all values of one parameter converge. Hence, the precision of the mapping process decreases.

According to the calculations given in [27], 34 floating point multiplications and two square root operations are required for each (u, v) pair.

3.1.3 Rational Linear Interpolation for Polygon Texture Mapping

Heckbert and Moreton [32] propose a method for the interpolation of texture coordinates and shading parameters for polygons viewed in perspective. To remove the rubber sheet effect caused by early linear interpolations, they use a rational linear interpolation across a polygon. This method performs a clipping on a polygon against the plane equations of the six sides of the viewing frustum and is suggested for scan conversion algorithms.

3.1.4 Parameterization of Triangles

Heckbert [31] gives a 2D parameterization for triangles defined in 3D. The equation for parameterization is as follows:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{3.3}$$

The nine variables, A through I, are calculated from the Cartesian coordinates of the inverse mapped triangle by solving three sets of equations with three unknowns. Also in order to directly parameterize a point on a triangle, Equation 3.3 can be inverted as follows:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.4}$$

3.2 Implementation

Mapping by using Barycentric coordinates [17] and the triangle-to-triangle mapping method mentioned in [27] do not preserve the center of gravity. The method mentioned in [27] is an extension of quadrilateral-to-quadrilateral mapping to triangle-to-triangle mapping by combining the two neighboring vertices of the quadrilaterals. The rational linear interpolation for polygon texture mapping suggested in [32] is not suitable for ray tracing. Application of this method with ray tracing will be very costly because of clipping operations. Parameterization of triangles is the best method. It performs a better interpolation and needs less mathematical operations for each (u, v) pair, but its preprocessing cost is high.

All of the methods mentioned above perform the mapping from an arbitrary triangle to an arbitrary triangle. For our purposes, it is enough to do the mapping from an arbitrary triangle to a right triangle. Therefore, the method we propose works from an arbitrary triangle to a right triangle and the mapping preserves the center of gravity of the triangles.

In Figure 3.4, T_1 is a triangle on the surface and T_2 is the corresponding triangle on the texture. The hit point on the surface is H_1 and the corresponding point is H_2. A line passing through H_1 and C_1, and a line passing through H_1 and A_1 are intersected with the two edges of T_1. Our aim is to find the coordinates of H_2 such that:

$$r_{BA} = \frac{|B_1 I_{B_1A_1}|}{|B_1A_1|} = \frac{|B_2 J_{B_2A_2}|}{|B_2A_2|} \quad and \quad r_{BC} = \frac{|B_1 I_{B_1C_1}|}{|B_1C_1|} = \frac{|B_2 J_{B_2C_2}|}{|B_2C_2|} \tag{3.5}$$

If r_{BA} and r_{BC} in Equation 3.5 are equal to 0.5, then the hit point is on the intersection point of the medians of the triangle. The intersection point

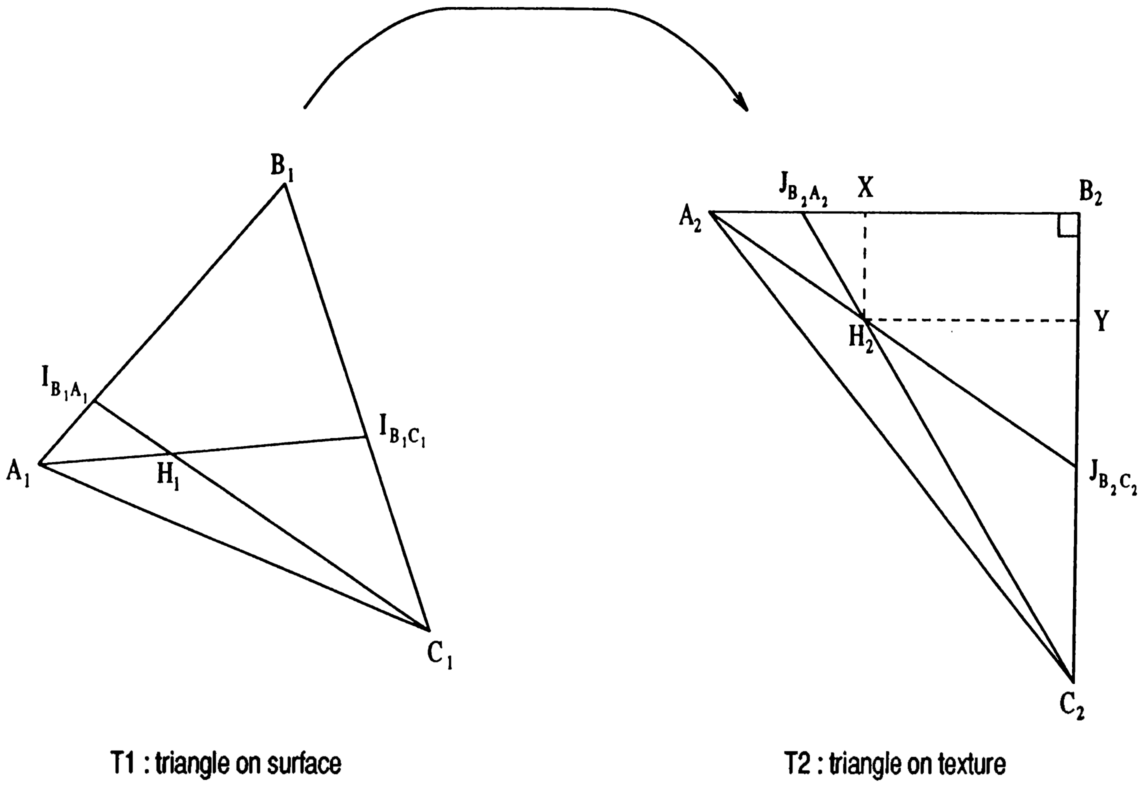

Figure 3.4: Mapping from an arbitrary triangle to a right triangle

of medians in a triangle defines the center of gravity of the triangle. Hence our mapping preserves the center of gravity of the triangles. The texture is divided similar to the surface that it will be mapped. So, the lengths of the edges of T_2, $|B_2A_2|$, $|B_2C_2|$, and $|A_2C_2|$, are calculated beforehand and stored for each texture map. $|B_2J_{B_2A_2}|$ and $|B_2J_{B_2C_2}|$ are calculated after finding the above ratios for T_1. Next step is to find $|B_2X|$ and $|B_2Y|$. From $J_{B_2A_2}\overset{\triangle}{X}H_2 \sim J_{B_2A_2}\overset{\triangle}{B_2}C_2$:

$$\frac{|B_2J_{B_2A_2}| - |B_2X|}{|XH_2|} = \frac{|B_2J_{B_2A_2}|}{|B_2C_2|} \tag{3.6}$$

From $J_{B_2C_2}\overset{\triangle}{B_2}A_2 \sim J_{B_2C_2}\overset{\triangle}{Y}H_2$:

$$\frac{|B_2J_{B_2C_2}| - |B_2Y|}{|YH_2|} = \frac{|B_2J_{B_2C_2}|}{|B_2A_2|} \tag{3.7}$$

After the necessary calculations with Equations 3.6 and 3.7, we get:

$$|B_2Y| = |XH_2| = \frac{|B_2C_2| \cdot |B_2J_{B_2C_2}| \cdot (|B_2J_{B_2A_2}| - |B_2A_2|)}{|B_2J_{B_2A_2}| \cdot |B_2J_{B_2C_2}| - |B_2C_2| \cdot |B_2A_2|} \quad (3.8)$$

$$|B_2X| = |YH_2| = \frac{|B_2A_2| \cdot (|B_2J_{B_2C_2}| - |B_2Y|)}{|B_2J_{B_2C_2}|} \quad (3.9)$$

The two unknowns in Equations 3.8 and 3.9, $|B_2J_{B_2A_2}|$ and $|B_2J_{B_2C_2}|$, can be calculated as follows:

$$|B_2J_{B_2C_2}| = r_{BC} \cdot |B_2C_2| \quad (3.10)$$

$$|B_2J_{B_2A_2}| = r_{BA} \cdot |B_2A_2| \quad (3.11)$$

In order to find $|B_2J_{B_2A_2}|$ and $|B_2J_{B_2C_2}|$, we have to calculate r_{BA} and r_{BC}. The following equation can be written in terms of x, y, and z coordinates:

$$\begin{bmatrix} x_{I_{B_1C_1}} \\ y_{I_{B_1C_1}} \\ z_{I_{B_1C_1}} \end{bmatrix} = (1 - r_{BC}) \cdot \begin{bmatrix} x_{B_1} - x_{C_1} \\ y_{B_1} - y_{C_1} \\ z_{B_1} - z_{C_1} \end{bmatrix} + \begin{bmatrix} x_{C_1} \\ y_{C_1} \\ z_{C_1} \end{bmatrix} \quad (3.12)$$

$\overrightarrow{A_1H_1}$ and $\overrightarrow{H_1I_{B_1C_1}}$ are in the same direction. If T_1 is on neither the x nor the y plane, then the following equation can be written:

$$\frac{x_{H_1} - x_{A_1}}{y_{H_1} - y_{A_1}} = \frac{(1 - r_{BC}) \cdot (x_{B_1} - x_{C_1}) + x_{C_1} - x_{H_1}}{(1 - r_{BC}) \cdot (y_{B_1} - y_{C_1}) + y_{C_1} - y_{H_1}} \quad (3.13)$$

If H_1 and A_1 coincide, H_1 is directly mapped to A_2. If T_1 is on the x plane, x's should be replaced with z's and if T_1 is on the y plane, y's should be replaced with z's in Equation 3.13. An easy way to test whether a triangle is on the $x - y$ plane is to compare the z coordinates of the vertices of the triangle. If z coordinates are the same, then the triangle is on the $x - y$ plane. Similarly, if x coordinates are the same, then the triangle is on the $y - z$ plane and if y coordinates are the same, then the triangle is on the $x - z$ plane. After arranging the terms in Equation 3.13, we get:

$$r_{BC} = 1 - \frac{(x_{H_1} - x_{A_1}) \cdot (y_{C_1} - y_{H_1}) - (y_{H_1} - y_{A_1}) \cdot (x_{C_1} - x_{H_1})}{(y_{H_1} - y_{A_1}) \cdot (x_{B_1} - x_{C_1}) - (x_{H_1} - x_{A_1}) \cdot (y_{B_1} - y_{C_1})} \quad (3.14)$$

r_{BA} can be found in a similar fashion. If H_1 is on $|A_1C_1|$, i.e., both r_{BC} and r_{BA} are 1, then the corresponding point H_2 on the hypotenuse of T_2 is found by calculating r_{AC}, similar to the above calculations. $|A_2C_2|$ is precalculated and stored to speed up the calculations. If H_1 is on $|A_1C_1|$, then:

$$|B_2Y| = r_{AC} \cdot |B_2C_2| \quad (3.15)$$

$$|B_2X| = (1 - r_{AC}) \cdot |B_2A_2| \quad (3.16)$$

Because of the floating point intersection calculations, it is very rare that both r_{BC} and r_{BA} will be 1, i.e., H_1 will be just on $|A_1C_1|$. Therefore, if we ignore the extra calculations for hit points on $|A_1C_1|$ and a few comparison operations, we need only 19 floating point multiplicative operations to find the coordinates on the texture space.

3.3 Antialiasing

In order to do antialiasing in texture mapping, a continuous domain must be sampled. A space variant filter is needed for antialiasing texture mapped surfaces. A single pixel will represent a large area of a texture mapped object if the object is far away from the view point. If the texture is mapped by point sampling, this pixel will have the color of a single point. In point sampling rendering methods, area coherence is not taken into account. If the mapping is done by using a filtering technique, a weighted average of an area will be assigned to the sampled point. Aliasing in texture mapping is also seen at the shrunken images. Therefore, filtering is essential in 2D texture mapping.

Deformed objects and the objects seen in perspective need space variant filters, that is, filters whose shape and area change. Usually, the area in texture domain corresponding to a pixel area is a quadrilateral, cf. Figure 3.5 [13].

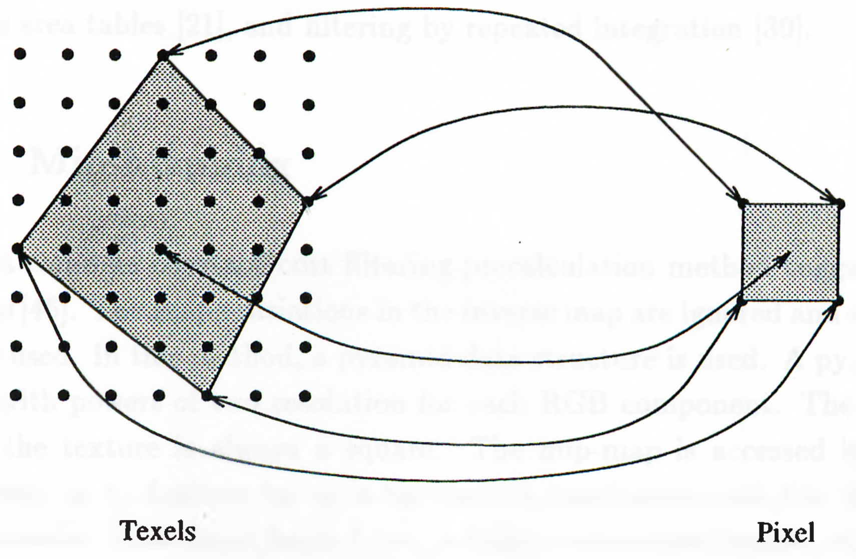

Figure 3.5: Region of texture corresponding to a pixel

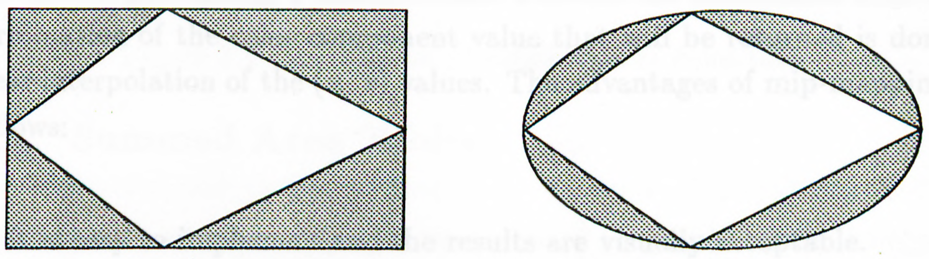

Figure 3.6: A bounding rectangle and a bounding ellipse for a quadrilateral

In our implementation, we put this quadrilateral into a bounding box and filter with respect to this rectangle. If the ratio of the area of the bounding rectangle to the area of the quadrilateral is large, there will be blurring. Bounding ellipses can be used instead of bounding rectangles for better approximations [26]. The approximations by a rectangle and an ellipse are shown in Figure 3.6. An adaptive precision technique is suggested by Glassner [25].

It will be very costly to calculate the weighted average of a texture domain area when rendering techniques that access the environment randomly, such as ray tracing, are used. Direct convolution techniques become very expensive. If prefiltering techniques are used, the cost of filtering remains constant although the area to be filtered grows. Therefore, in our implementation, we used a prefiltering technique, viz. repeated integration filtering [30], which is a generalization of summed area tables [21]. In the following subsections, we will briefly mention three important prefiltering techniques: mip-mapping [46], summed area tables [21], and filtering by repeated integration [30].

3.3.1 Mip-Mapping

This is a common constant cost filtering precalculation method suggested by Williams [46]. The shape variations in the inverse map are ignored and a square shape is used. In this method, a pyramid data structure is used. A pyramid is formed with powers of two resolution for each RGB component. The inverse map of the texture is always a square. The mip-map is accessed by three parameters, u, v, d where (u, v) is the spatial coordinates and d is the level of compression. Selecting a large d, i.e., a highly compressed image, will cause blurring and selecting a small d will cause aliasing. Therefore, determination of d is crucial and usually a blend is made between the two nearest mip-maps. Determination of the color component value that will be returned is done by bilinear interpolation of the (u, v) values. The advantages of mip-mapping are as follows:

- It is easy to implement and the results are visually acceptable.
- It does not require much storage memory. The memory cost is 4/3 times that required for an unfiltered image.
- Besides box filters, Gaussian filters can also be used for constructing the image pyramid.

The disadvantages of mip-mapping are as follows:

- Filter shape is always a square.
- There is no correct way of calculating d.
- Too many interpolation calculations are necessary.

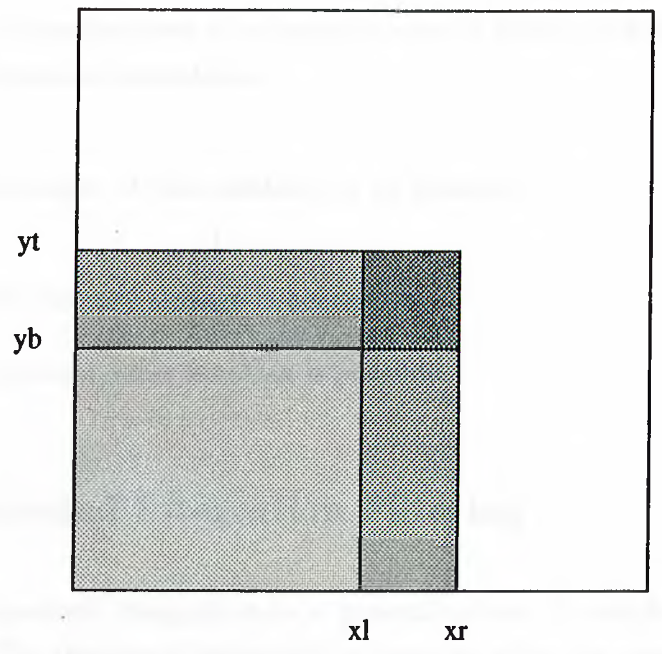

Figure 3.7: Calculation of summed area from table

3.3.2 Summed Area Tables

A generalization of mip-mapping is the summed area tables method suggested by Crow [21]. This method provides a better approximation to the proper texture intensity than mip-mapping, by allowing rectangular regions of texture to be used. A single table containing large numbers is used. From this table, a continuous range of texture densities can be drawn. Each texture intensity is replaced by a value representing the sum of the intensities of all pixels contained in the rectangle defined by the pixel of interest and the lower left corner of the texture image. In Figure 3.7, it can be seen that the sum of intensities over an arbitrary rectangle can be found with a sum and two difference operations:

$$T[xr, yt] - T[xr, yb] - T[xl, yt] + T[xl, yb]$$

where T[x, y] is the value of the summed area table at location (x, y). Dividing this sum by the area of the rectangle gives the average intensity over the rectangle. The advantages of this method are as follows:

- The table is indexed by a rectangular area, so a closer approximation to the inverse pixel map is maintained.

- Sum of intensities over a rectangular area is found with a small number of mathematical operations.

The disadvantages of this method are as follows:

- Too much memory storage is needed.
- Only a constant filter function is possible.

3.3.3 Repeated Integration Filtering

Filtering by repeated integration is a generalization of summed area tables method [30]. The texture is integrated n times to take the advantage of high quality filtering. As n increases, kernel shape of the filter approaches to a Gaussian filter.

Theoretical details of this method can be found in [30]. In Figure 3.8, a C code for integration of the texture array is given. Some grid coefficients for low order filters are shown in Figure 3.9. For $n = 1$, note that the integration and the grid coefficients are the same as in summed area tables method.

The main advantage of this method is the control of filter shape that provides high quality filtering. Unfortunately, the storage cost increases linearly as the quality of the filters increases. For a $2^r \times 2^r$ monochrome image with b bits per pixel, integration n times yields $2nr + b$ bits per integrated array element. For example, 44 bits are needed for triangle filtering ($n = 2$) of an 8-bit 512×512 image.

We observed that box filtering ($n = 1$) blurs the textures. We implemeted triangle filtering by using integrated arrays. This gives satisfactory visual results.

```
void
integrate_array(n)
int          n;
{
  int        i, x, y;

  for (i = 1; i <= n; i++){
      for (y = 0; y < size_y; y++)
          for (x= 1; x < size_x; x++)
              T[x][y] += T[x-1][y];

      for (x = 0; x < size_x; x++)
          for (y=1; y < size_y; y++)
              T[x][y] += T[x][y-1];
  }
}
```

Figure 3.8: Repeated integration of texture array

```
n=0      n=1          n=2            n=3

 1      -1  1       1 -2  1      -1  3 -3  1
         1 -1      -2  4 -2       3 -9  9 -3
                    1 -2  1      -3  9 -9  3
                                  1 -3  3 -1
```

Figure 3.9: Grid coefficients for low order filters

Chapter 4

Area Sampling in Ray Tracing

Although ray tracing is one of the most popular and powerful techniques in image synthesis, it has two major disadvantages:

- Generation time of a scene takes too much time.
- Classic ray tracing is a point sampling method and causes aliasing.

A substantial amount of research is being done to speed up ray tracing algorithms. In our implementation, we took advantage of spatial coherency to speed up the execution. The details of this implementation are discussed in Chapter 5.

Classic ray tracing causes severe aliasing especially when used with texture mapping. In classic ray tracing, a picture is generated by tracing rays backwardly from the eye into the scene, recursively exploring specularly reflected and transmitted directions, and tracing rays toward point light sources to simulate shading [45]. This classic form of ray tracing is a point sampling method.

One easy antialiasing technique used in ray tracing is *supersampling* [45, 20]. A group of rays is used to generate an image, and then the final color at each individual pixel is found by averaging the colors of all the rays within that pixel. *Adaptive supersampling* is a technique developed for reducing the execution time of supersampling [45]. More rays are sent through a pixel only if there is a need for antialiasing. *Adaptive supersampling* assumes that if some fixed number of rays are about the same color, then the pixel is sampled well. Another antialiasing technique is *stochastic ray tracing* [18, 19]. Rather than sending out rays on a regular grid, rays are randomly distributed across the space. *Stochastic ray tracing* helps to get motion blur, depth of field, and penumbra. *Statistical supersampling* is another antialiasing technique to reduce the number of rays per pixel [35, 39]. In this case, sending out rays for a pixel is stopped if enough sampling is achieved.

Unfortunately, the antialiasing methods mentioned above do not help us to filter textures. In the above methods, an average color value is found for the final pixel intensity, but in order to filter textures a weighted average of a continuous space is needed. Also the textures that are reflected or refracted have to be filtered. The only way to filter textures with ray tracing is to do area sampling.

Whitted saw the drawback of point sampling in ray tracing and suggested a version of area sampling [45]. Instead of using linear eye rays, a pyramid defined by the eye and the four corners of a pixel could be used. All the intersection, reflection and refraction calculations are done with these pyramids. This method is very costly and is currently not used.

4.1 Beam Tracing

Beam tracing is introduced by Heckbert and Hanrahan [29]. Polygonal environments are traced with pyramidal beams. The trace is started with a single beam defined by the viewing pyramid and is intersected with the polygons in the environment. New polyhedral pyramids are spawned for reflection and refraction and the trace continues as in classic ray tracing. The cross-section of the new beams are defined by the clipped area of the polygon. Beam tracing takes advantage of coherence, but is limited to polygonal environments and needs complex clipping calculations.

Beam tracing creates shadows in a different manner from the classic ray tracing. Caustic polygons are formed with light beams and these are included to the data structure for each polygon. If a polygon is visible, its caustic polygons are also taken into account for shading.

4.2 Cone Tracing

Cone tracing is introduced by Amanatides [6]. Instead of a ray, a cone is defined including the information on the spread angle and the virtual origin. The spread angle is defined as the angle between the center line of the cone and the cone boundary as measured at the apex of the cone. This angle is chosen such that when the ray is sent from the eye, the radius of the cone at the distance of the virtual screen is the width of the pixel. The virtual origin is the distance from the apex of the cone to the origin.

Intersection calculations are done between the object and the cone. Besides the intersection information, fractional blockage information of the objects are also stored. The virtual origin and the spread angle of the cones are determined by using the surface curvature information.

One impressive advantage of cone tracing is that point light sources can be extended to spheres to simulate fuzzy shadows. A shadow cone is generated whose base is the cross-section of the light source. By calculating how much of the light source is blocked by the intervening object, fuzzy shadows can be generated.

Amanatides suggests that by estimating the size and the shape of the intersection, filters can be generated to average the texture map.

4.3 Pencil Tracing

Shinya *et al.* [41] proposed a technique called pencil tracing for area sampling in ray tracing. A pencil, consisting of a central axial ray and surrounded by a set of nearby paraxial rays, is formed. Paraxial rays are represented by a 4D vector. Two dimensions of this vector express the paraxial ray's intersection with a plane perpendicular to the axial ray and the other two dimensions express the paraxial ray's direction. A well known theory in optical design and electro-magnetic analysis, paraxial approximation theory, is used to trace the rays in the environment [41].

According to the paraxial approximation theory, surfaces are assumed to be smooth. Therefore paraxial rays do not diverge at the edges. Conventional ray tracing is applied when a pencil intersects an edge of an object.

4.4 Area Sampling Buffer

Sung [43] suggested the Area Sampling Buffer (ASB) that allows the use of z-buffer hardware for performing area sampling in the ray direction for a ray tracing style renderer. For each final image pixel, an $N \times M$ frame buffer of m-bits depth is allocated. Each pixel in the frame buffer is filled with a unique id which corresponds to the object that is visible through that pixel. Then, only one ray/object intersection is sufficient to do the tracing. Image generation time is mostly dependent on the number of scan conversions. This implementation suffers from the limitations of z-buffer for transparent objects. Furthermore, the objects in the environment are required to be represented by polygons.

4.5 Implementation

Except the Area Sampling Buffer method, all of the area sampling methods mentioned above replace linear rays with a geometric object. The intersection calculations are done with respect to these geometries. Complex calculations are needed for clipping and intersection. To reduce the complexity of these calculations, the objects in the environment are restricted with simple geometric shapes. Area Sampling Buffer method is different from the other methods because it takes advantage of z-buffer hardware.

In our implementation, all of the intersection calculations are the same as in classic ray tracing. A ray is shot from the view point and is traced in the environment. The ray tracer is written in C language [34] by using *lex* and

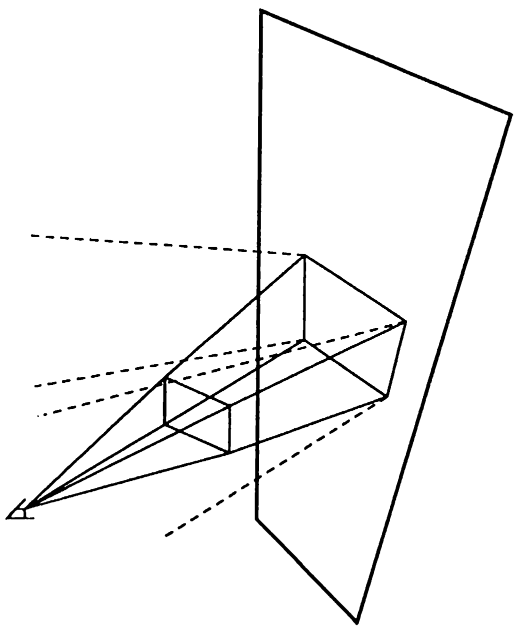

Figure 4.1: A reflection pyramid

yacc [38] and is an extension to the ray tracer developed by İşler [33].

Conceptually, our implementation is the same as sending a pyramid and tracing it for 2D texture mapped areas. A reflection pyramid is shown in Figure 4.1. The reflection pyramid can be very wide if the surface is bumpy.

To achieve area sampling by using a classic ray tracer, we used three buffers each holding the necessary hit information for a screen row. Minimum three rows have to be processed to sample the base area of a reflection or refraction pyramid. Since we trace the rays forming the pyramids, our intersection calculations remain the same as in a classic ray tracer. Shading part is different from classic ray tracing. Shading is delayed until we get three rows of buffers.

Texture mapping is implemented on a triangle-to-triangle basis. Therefore, the sampled areas in the scene are the objects that are represented by triangles. Other geometric shapes such as spheres, boxes, superquadrics [10], etc. can also be present in the environment, but they are not area sampled.

We implemented space variant filtering by using repeated integration filtering [30]. Using a box filter blurs the image excessively. Triangle filtering eliminates blurring and gives reasonable results, but is slower than box filtering and needs much more space than box filtering.

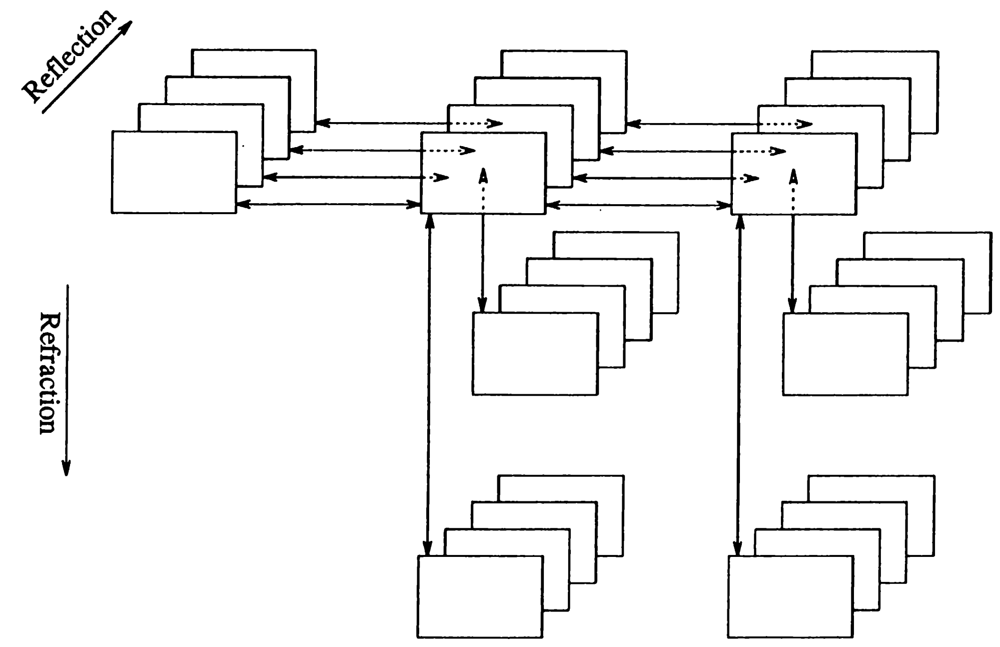

Figure 4.2: A linked list representation of a row buffer

4.5.1 Algorithm

When three row buffers have been filled with the hit information obtained from the ray tracer, we begin calculating the color value for the middle row. The color value for a pixel is calculated by beginning from the node at the maximum reflection depth level until the primary ray level. The color values are transmitted to a lower reflection level.

The linked list representation for a typical row buffer for three pixels is shown in Figure 4.2. An intersection tree is constructed for each pixel. Each node of an intersection tree is represented by a rectangle and contains the id of the object hit and the intersection point. Refraction levels grow vertically and reflection levels grow horizontally.

Our algorithm consists of three parts:

1. Construction of Intersection Trees
2. Shading
3. Shifting Rows of Intersection Trees

```
/* node is a pointer to the top of the intersection tree for a pixel.  */
Store hit information at node[reflect_l];

if ((ndx_reflection > 0) &&
    (ndx_reflection < maxlevel)) then
   ++refract_l;
   Call intersection routine recursively to find the new intersection point;
   --reflect_l;

if (ndx_refraction > 0) then
   Refract the ray at the intersection point of the semi-transparent object;
   If there is an internal reflection when refracted ray passes to a new
   new medium, then call intersection routine recursively to find the new
   intersection point;

   Get the new intersection point at the new medium;

   Create a new node and add it to the down list of
   node[reflect_l];
   ++refract_l;
   Call intersection routine recursively to find the new intersection point;
   --refract_l;
   node = node[reflect_l]->up;
```

Figure 4.3: Construction of an intersection tree

Construction of Intersection Trees

In order to easily access the reflection levels, we store an array of pointers to each intersection tree for a pixel. This provides easy access at reflection levels.

Each node of an intersection tree contains the minimum necessary information, i.e., object id, texture map parameters, and coordinates of the intersection point.

At least three rows of intersection trees must be processed and stored in order to perform area sampling. The algorithm for the construction of an intersection tree for a pixel is given in Figure 4.3.

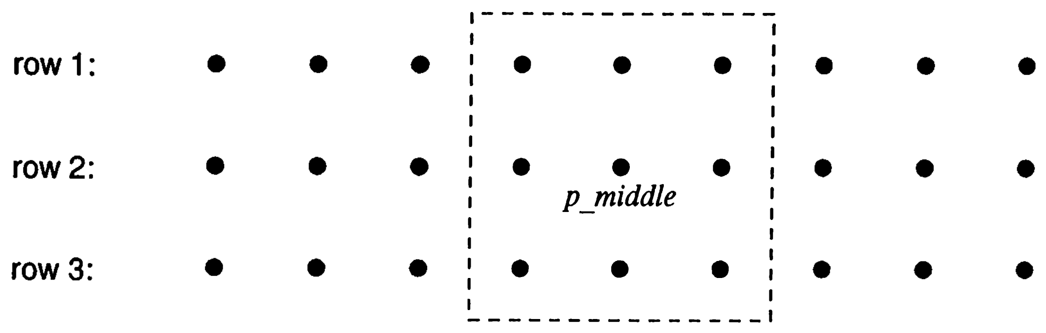

Figure 4.4: Area sampling

Shading

Shading is done after three row buffers have been filled with intersection trees.

In Figure 4.4, the color value for filtering operation, i.e., the diffuse color, for node *p_middle* is calculated by filtering the area surrounded by the box, if the nodes are on the same reflection and refraction depths and they hit onto the same area that will be textured. If one of these conditions does not hold, we get the diffuse color value of the object at that point without filtering. To this diffuse color value we add the transmitted color value coming from a higher reflection level and a refraction level if it exists. If the node is at the maximum depth level, the transmitted color value is zero. If the reflected ray goes out of space, then the transmitted color value for reflection is the background color. Local color is determined by the illumination of the light sources. If the object for a node is a refracting surface, we do the same operations for the refraction list, and contribute the color value from refraction to that point. We transmit the local color value to a lower reflection level. This operation is done until finding the actual pixel color.

The algorithm is given in Figure 4.5.

Shifting Rows of Intersection Trees

After we are finished with row 2, we shift row 2 to the place of row 1 and row 3 to the place of row 2. Then row 3 is filled as previously described.

If subpixel sampling is done by sending nine rays instead of one for each

```
/* node is a pointer to the top of the intersection tree for a pixel.
Branching path at each refraction level is stored in parents array.
Transmitted color from a higher reflection level is stored in colors
array for each refraction level.  */
for each pixel in a row do
    reflect_l = maxlevel;
    refract_l = 0;
    transmit_color = background_color;
    flag = 1;
    while (flag) do
          get local color for node[reflect_l];
          transmit_color = local_color +
                           ndx_reflectivity * transmit_color;
          colors[refract_l] = transmit_color;
          if ((node[reflect_l]->down != NULL) &&
             (node[reflect_l]->processed == FALSE)) then
            /* Traverse down the tree.  */
            parents[refract_l] = reflect_l;
            node = node[reflect_l]->down;
            reflect_l = maxlevel;
            ++refract_l;
            transmit_color = background_color;
          else if (((node[reflect_l]->down == NULL) ||
                  (node[reflect_l]->processed == TRUE)) &&
                 (node[reflect_l]->up != NULL)) then
                while ((refract_l > 0) &&
                       (parents[refract_l-1] == reflect_l)) do
                      /* Go up to a lower refraction level.  */
                      --refract_l;
                      node = node[reflect_l]->up;
                      colors[refract_l] += ndx_reflectivity *
                                           colors[refract_l+1];
                      node[reflect_l]->processed == TRUE;
                --reflect_l;
          else
              --reflect_l;

          if ((refract_l <= 0) && (reflect_l < 0))  then
            flag = 0;
            output colors[refract_l];
```

Figure 4.5: Traversal in the intersection tree for a pixel

pixel, then row 1 and row 2 must be cleared and row 3 must be shifted to the place of row 1.

4.5.2 Shadows

Shadows can easily be incorporated into a ray tracer. Light vectors are formed between each intersection point and the light sources. If there is an occlusion on the path of a light vector, then the point is in shadow. If the point is completely occluded by opaque objects from the light sources, then local illumination at that point is defined by the ambient color at that point. If the occluding object is *semi-transparent*, then the illumination by the light source is multiplied by the transparency index of the object. Hence, the shadows of *semi-transparent* objects become lighter than the shadows of opaque objects.

In fact, the light vector should be refracted when it intersects a *semi-transparent* object. However, it is not possible to refract light vectors with backwards ray tracing because the light vector is calculated as a straight line at the beginning and it is very difficult to include refraction effects. Nevertheless, this implementation is easily implemented and gives satisfactory results. Also we do not include the effects of occluding reflective objects in order not to complicate the calculations.

4.5.3 Refraction

Currently, ray tracing is the only global illumination method that handles refraction. Semi-transparent objects can be incorporated into a ray tracer elegantly, without complicating the implementation. We use the laws of optics to trace a ray in a refractive medium. As can be seen in Figure 4.6, an incident ray entering to a dense medium refracts in the direction of **T**. Here, we will give the derivation of the refraction formula [29] that we used in our implementation.

Referring to Figure 4.6, **N** and **I** are assumed to be unit and **T** is also guaranteed to be unit. Let $c_1 = cos\theta_1 = -\mathbf{I} \cdot \mathbf{N}$. The reflection vector is given as:

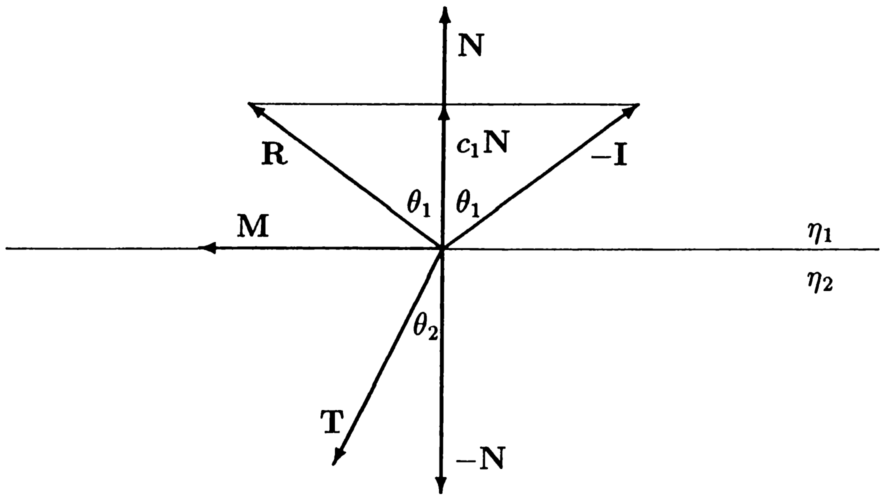

Figure 4.6: Refracting ray

$$\mathbf{R} = \mathbf{I} + 2c_1\mathbf{N} \tag{4.1}$$

The refracted ray is:

$$\mathbf{T} = sin\theta_2\mathbf{M} - cos\theta_2\mathbf{N} \tag{4.2}$$

where **M** is a unit surface tangent vector as shown in Figure 4.6.

$$\mathbf{M}sin\theta_1 = \mathbf{I} + c_1\mathbf{N} \tag{4.3}$$

$$\mathbf{M} = \frac{\mathbf{I} + c_1\mathbf{N}}{sin\theta_1} \tag{4.4}$$

So, we get:

$$\mathbf{T} = \frac{sin\theta_2}{sin\theta_1}(\mathbf{I} + c_1\mathbf{N}) - cos\theta_2\mathbf{N} \tag{4.5}$$

According to Snell's law, $\frac{sin\theta_2}{sin\theta_1} = \frac{\eta_1}{\eta_2} = \eta$. So, **T** can be written as:

$$\mathbf{T} = \eta\mathbf{I} + (\eta c_1 - c_2)\mathbf{N} \tag{4.6}$$

where c_2 can be expressed as:

$$c_2 = cos\theta_2 = \sqrt{(1 - sin^2\theta_2)} = \sqrt{(1 - \eta^2 sin^2\theta_1)} = \sqrt{(1 - \eta^2(1 - c_1^2))} \quad (4.7)$$

Total internal reflection occurs when light tries to pass from a dense medium to a less-dense medium at an angle greater than *critical angle.* In our case, total internal reflection occurs when

$$\eta^2(1 - c_1^2) > 1. \quad (4.8)$$

4.5.4 Uniform Supersampling

One common way of alleviating the aliasing artifacts is *supersampling.* In *supersampling,* a high resolution image is used to generate a low resolution antialiased image. A weighted average of pixels in the high resolution image that contribute to the color value of an individual pixel in the low resolution image is calculated to find the color value for each individual pixel in the low resolution image. In ray tracing, this is equivalent to sending more than one ray for each pixel and then at the primary ray level, calculating the weighted averages of color values of those rays. (*Supersampling* is sometimes called *postfiltering.*) As we mentioned earlier, it does not help us for antialiasing 2D texture maps because it only increases the sampling frequency and reduces the aliasing. We have implemented *uniform supersampling* to reduce the aliasing artifacts seen as staircases at the edges.

A $2n \times 2n$ image can be dwarfed into an $n \times n$ image by using *uniform supersampling* easily, if we assume that nine rays per pixel are used. Three rows in the high resolution image are used to determine a row in the low resolution image. The color values of nine pixels in the high resolution image that determine the color value of a pixel in the low resolution image are weighted averaged. The weights are assigned according to the area that a pixel in a high resolution image contributes to a pixel in a low resolution image, i.e, 1/4 for the pixel in the middle, 1/16 for each of the four pixels at the corners, and 1/8 for each of the remaining four pixels.

Producing a higher resolution image takes about four times longer than the final image, but higher quality images are produced. Staircases appearing at the edges disappear and shadow edges become softer.

4.5.5 Modeling

Using advanced rendering techniques and increasing surface detail by texture mapping are essential in order to generate realistic images, but they are not sufficient. In nature, we see a large number of complex objects and most of them cannot be represented by simple mathematical formulations. Therefore, advanced modeling tools must be used to produce complex objects. As a modeling tool we used a program called T♭ ('Topology♭ook') [2, 3, 4, 5, 8]. T♭ is especially designed to help topologists to illustrate their ideas more effectively. In T♭, the modeling of 3D shapes is based on sweeping. A 2D contour curve is swept around a 3D trajectory curve. Contour curves can also be varying, and several contour curves with different shapes can be assigned interactively to the trajectory curve. T♭ has other features such as depth modulation, blending, twisting, deformation, etc.

T♭ generates an output file containing the coordinates of the discrete points that are obtained after the sweep operation. Since modeling is done interactively, there is no compact mathematical formulation of these discrete points. Each point on the object can be represented parametrically by a contour and trajectory number pair. The four vertices of the quadrilaterals formed by the parametric representation are not guaranteed to lie on the same plane. Therefore, we represented the surfaces by triangles. Triangulization is done by using the contour and trajectory number pair. Triangulization by using the mesh model effects the shape of the surface, but if a large number of contour and trajectory curves are used to model a surface, then this effect is negligible.

After the triangulization, a scene is created. Besides the objects represented by triangles, there may be other geometric objects (e.g., spheres, superquadrics, boxes, rectangles, etc.) in the scene. An input file is prepared in *lex* and *yacc* [38] and by using the texture mapping library, mapping is applied only to the objects represented by triangles. Each triangle has a triangle id and texture information id besides the surface property id and the coordinates of the vertices. Finally, the input file is given to the ray tracer, cf. Figure 4.7.

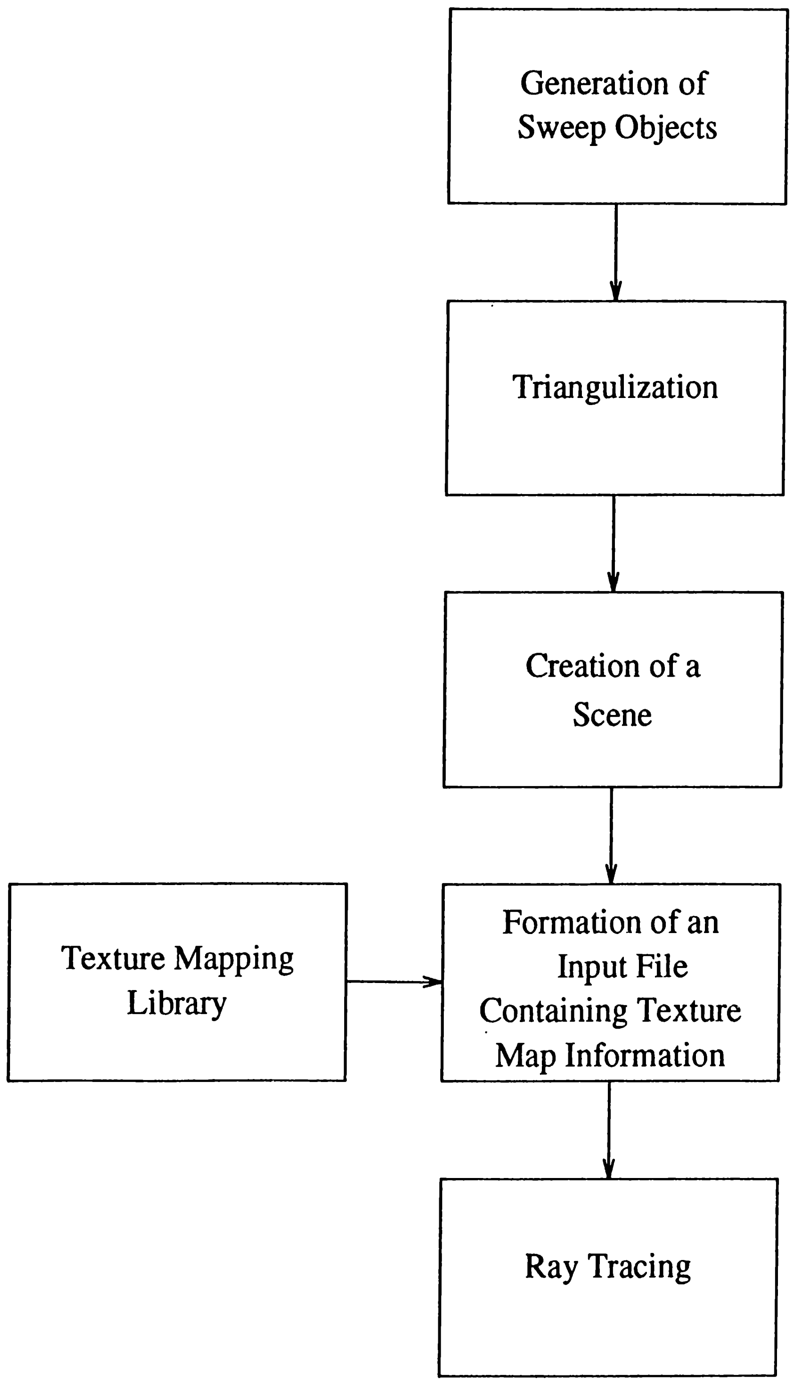
Generation of Sweep Objects
Triangulization
Creation of a Scene
Texture Mapping Library
Formation of an Input File Containing Texture Map Information
Ray Tracing

Figure 4.7: Image generation steps

Chapter 5

Acceleration of Ray Tracing

Although ray tracing is one of the best methods for generating realistic computer images, it is very slow. Most of the time in ray tracing is spent on intersection calculations. Scan-line methods take into account the object and/or image coherence. In ray tracing, each ray is traced independently and hence no information is available about object coherence.

5.1 General Approaches for Acceleration

Most of the research in ray tracing is concentrated on speeding up the intersection calculations. There are broadly two types of approaches that can reduce the number of intersection calculations:

- Hierarchical bounding volumes
- Space Partitioning

A survey of ray tracing acceleration techniques can be found in [9]. According to Whitted [45], 95% of the total execution time of ray tracing may be spent on intersection calculations for complex environments.

There are mainly three strategies to accelerate the process of ray tracing:

1. Faster Intersections

2. Fewer Rays

3. Generalized Rays

Faster intersections can be achieved by implementing faster and fewer ray/object intersections. This is the method we chose for our implementation. We put the objects into bounding volumes for achieving faster ray/object intersections and subdivided the bounding volumes into 3D grids.

By adaptive tree-depth control, the number of rays used can be reduced. A threshold for the maximum contribution to a pixel color can be set and the recursion is stopped when this threshold is reached [28]. If supersampling is used, then the number of rays used can be reduced by statistical optimizations.

Generalized rays have been mentioned in Chapter 4. Tracing with beam, cone, and pencil rays take advantage of spatial coherency and achieve the idea of tracing many rays simultaneously, but they have severe application limitations.

Before intersecting a ray with the object inside a bounding volume, the bounding volume is intersected. If the ray hits the bounding volume, we test intersections with the object inside. Also a tree of bounding volumes where each node enveloping the bounding volume of its children, can be constructed.

Partitioning the space into voxels and traversing in these voxels is another approach to perform fast intersections. The sizes of the voxels can be uniform or nonuniform as in octree structures. Two important features of uniform spatial subdivision are as follows:

1. Subdivision does not depend on the complexity of the environment.

2. Traversal of a ray in the voxels is done fast.

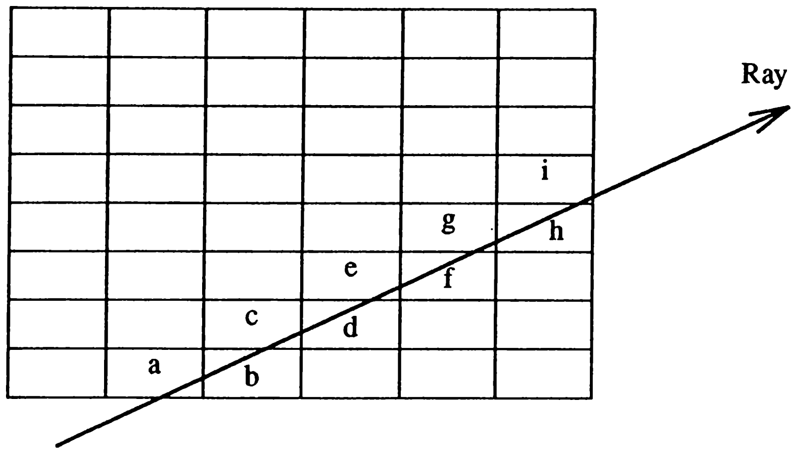

Figure 5.1: Grid traversal

5.2 3DDDA Algorithm

In our implementation, we put the objects inside axis-aligned bounding boxes and subdivided the boxes into uniform sized voxels. The traversal inside the voxels is done by using the 3DDDA algorithm [7]. The objects inside the bounding boxes are represented by using a large number of triangles to form an object. The subdivision of the bounding boxes can be made such that each voxel contains a maximum of about one or two triangles. Therefore, it is better to do a uniform subdivision and apply the 3DDDA algorithm.

3DDDA steps through each voxel which is pierced by the given ray. Rasterization algorithms, such as 2DDDA, identify pixels which are only close to a line. In Figure 5.1, a 2D grid and a ray traversing the grid is shown, [7]. The traversal algorithm visits all the grids a, b, c, d, e, f, g, h, and i, in that order.

The 3DDDA algorithm, which is an extension for traversing all the voxels that a ray pierces, is given in Figure 5.2 [7].

The variables *xstep*, *ystep*, and *zstep* are determined from the signs of parametric components of the ray equation. *xstep*, *ystep*, and *zstep* are either 1 or -1.

tmaxX is initialized to the value of t at which the ray crosses the x boundary of the voxels. Similarly, *tmaxX* and *tmaxZ* are also initialized. The minimum of these three values indicate how much we can travel along the ray and still

```
if (tmaxX < tmaxY){
  if (tmaxX < tmaxZ){
    x += xstep;
    tmaxX += tdeltaX;
  }
  else{
     z += zstep;
     tmaxZ += tdeltaZ;
  }
else{
   if (tmaxY < tmaxZ){
     y += ystep;
     tmaxY += tdeltaY;
   }
   else{
      z += zstep;
      tmaxZ += tdeltaZ;
   }
}

if ((x < 0) || (x >= box->divx) ||
    (y < 0) || (y >= box->divy) ||
    (z < 0) || (z >= box->divz))
  return (NULL); /* Ray exits the box.  */
else
  return (objects[x][y][z]);
```

Figure 5.2: 3DDDA algorithm

remain in the current voxel.

tdeltaX, *tdeltaY*, and *tdeltaZ* indicate the length of movement along the ray in t units to cross x, y, and z voxel boundaries, respectively.

Each voxel has an object list containing the objects in it. This object list is constructed beforehand. Since the traversal algorithm does not contain a floating point multiplicative operation, it is better to do the subdivision such that each voxel has one object in its list at maximum.

As the voxels are traversed, each of the objects in the current voxel are tested for ray/object intersection. If a ray/object intersection is found in a voxel, traversal of the ray continues on from that voxel.

Chapter 6

Conclusion

In this thesis, we proposed a method for combining texture mapping with ray tracing in order to generate highly realistic scenes. We developed our scenes by using sweep surfaces. The methods we proposed can be applied to any object represented by triangles.

We proposed a geometric mapping from an arbitrary triangle to a right triangle. This method preserves the center of gravity of the triangles. It has a low preprocessing cost and does not need much storage.

Area sampling in ray tracing is essential for filtering textures. We also proposed an algorithm for area sampling in ray tracing. The advantage of this algorithm is that the intersection calculations remain the same as in classic ray tracing. The method we proposed delays shading until enough row buffers are filled with necessary hit information. After area sampling, 2D textures are filtered by using a prefiltering technique, repeated integration filtering. The aim of the prefiltering techniques is to provide a constant cost execution time. These techniques are very fast when compared to direct convolution techniques. We observed that if textures with large sizes are used, too much memory is needed and the program makes too much swapping (thus increasing the execution time).

To remove the aliasing artifacts seen at the edges as staircases, we used a postfiltering technique, i.e., uniform supersampling.

To achieve better visual results, subpixel sampling can also be implemented. Extra rays can be shot for each pixel and fractional blockage information of the objects can be calculated as done in beam and cone tracing. This gives better antialiasing with an extra cost. Better antialiasing can also be achieved by stochastic sampling.

In order to accelerate the algorithm, we used a spatial subdivision technique. The bounding boxes of the objects are uniformly subdivided into 3D grids and the traversal in the grids is done by using the 3DDDA algorithm which performs integer arithmetic. One common way of speeding up the execution of programs is to use parallel algorithms. All the parallel algorithms developed for classic ray tracers can be applied to our system, but in practice this may not be always possible. Precalculation tables consume too much memory. A 24-bit, 512×512 resolution texture needs about 6 megabytes of memory for triangle filtering. If five such textures are used in a scene, then at least 30 megabytes of memory are needed just for keeping the precalculation tables. Therefore, if parallelization is going to be implemented, most of the efforts must be spent in managing the virtual memory. The *noise* function used for procedural texture mapping applications is asynchronously parallelizable. Therefore, procedural texture mapping library can be easily used with parallel algorithms.

Area coherence can be used for speeding up primary level ray object intersection tests.

As an improvement to this study, a user friendly interface combining modeling and rendering phases can be developed.

Appendix A

Sample Images

In this appendix, some images generated by texture mapping with ray tracing are presented. Some filtered and unfiltered versions of the same scenes are also given to show the difference between them. All of the images have been produced on Sun Sparc Workstations[1]. Approximate generation time for each image is also given. All the final images have a resolution of 512×512. Supersampled images have been generated from 1024×1024 resolution images.

[1]Sun Sparc Workstation is a registered trademark of Sun Microsystems, Inc.

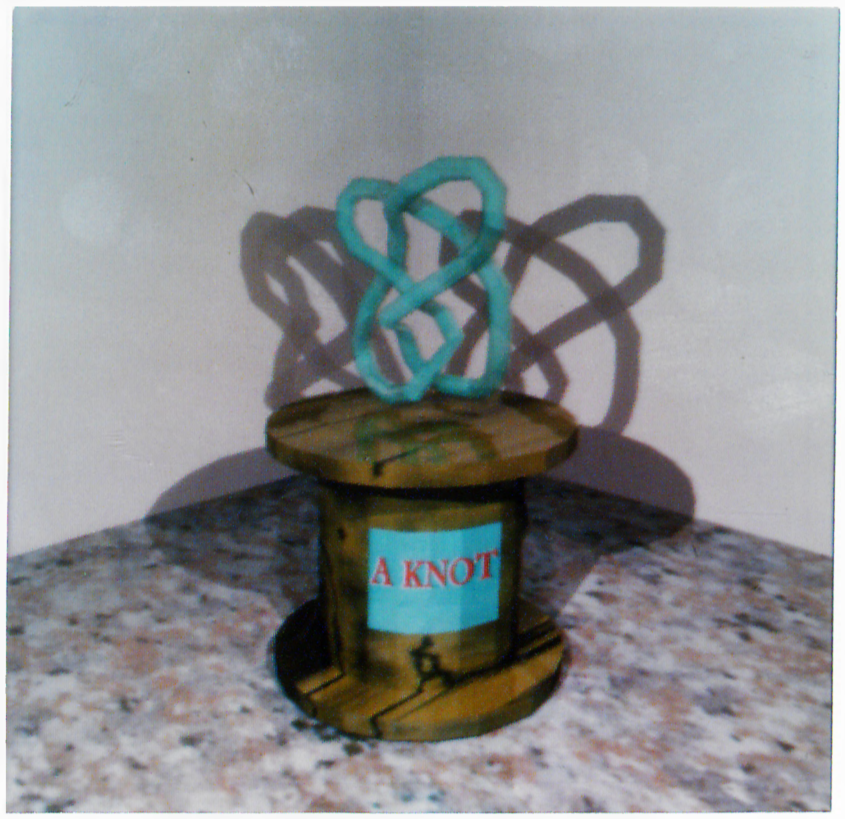

Figure A.1: A semi-transparent depth modulated sweep object on a marble column

In Figure A.1, the marble texture is produced by procedural texture mapping. A 2D granite texture is mapped onto the floor. The scene consists of approximately 6000 triangles. It took about three hours to produce the prefiltered and supersampled image in Figure A.1.

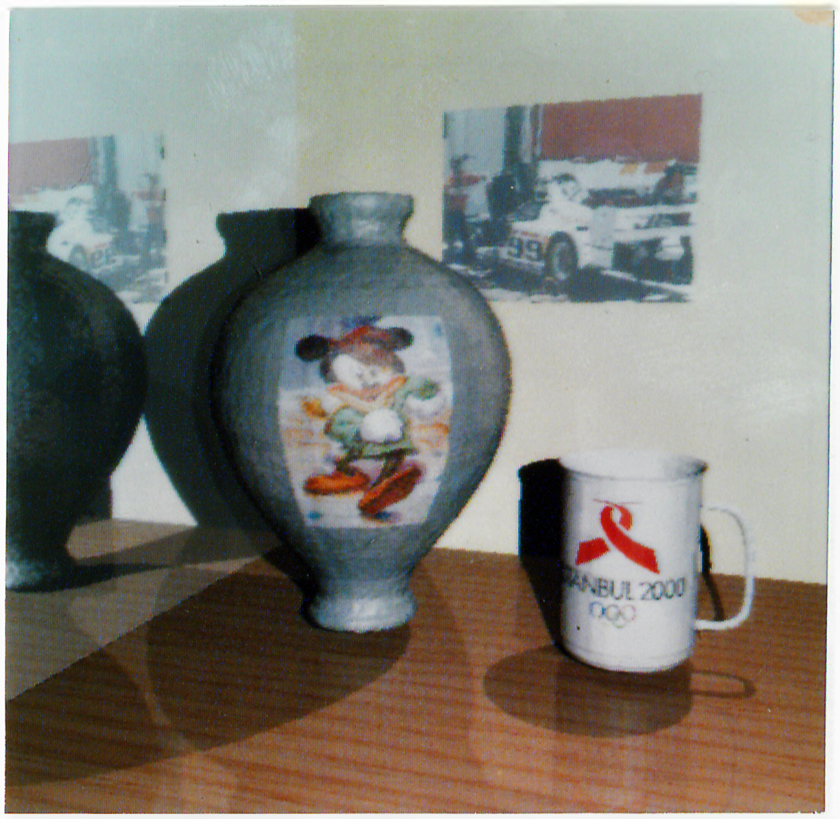

Figure A.2: A point sampled image

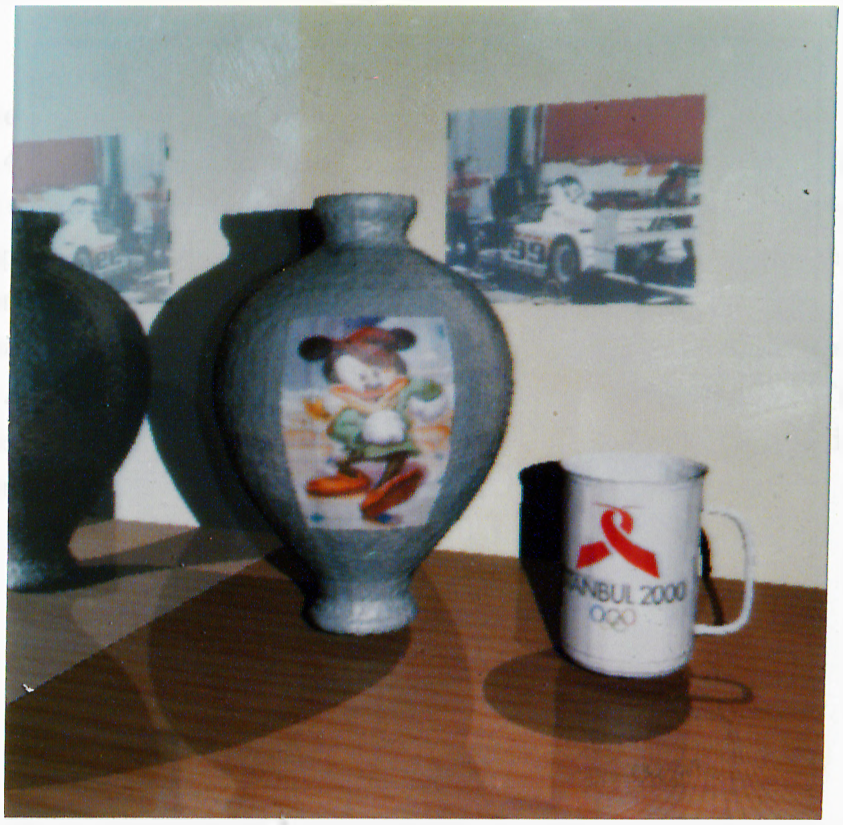

Figure A.3: A prefiltered image

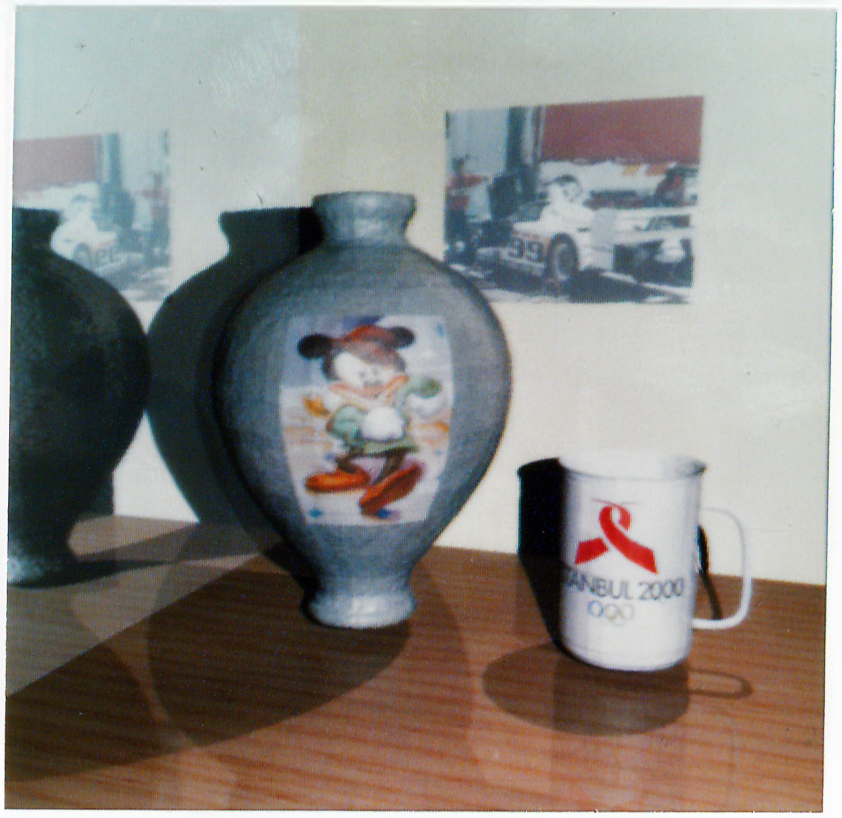

Figure A.4: A prefiltered and supersampled image

Three different versions of a scene are presented in Figures A.2, A.3, and A.4. The vase and the cup are rotational sweep objects. The handle part of the cup is created by sweeping a circle shaped contour curve around a 'U' shaped trajectory curve. This handle is then combined with the cup. The vase consists of 7200 triangles. The cup consists of 8400 triangles. The vase has also a bumpy texture. It took about 15 minutes to produce the image in Figure A.2 and 20 minutes to produce the prefiltered version in Figure A.3. The prefiltered and supersampled image in Figure A.4 is produced in about 1.5 hours. Note how the staircases disappear in Figure A.4.

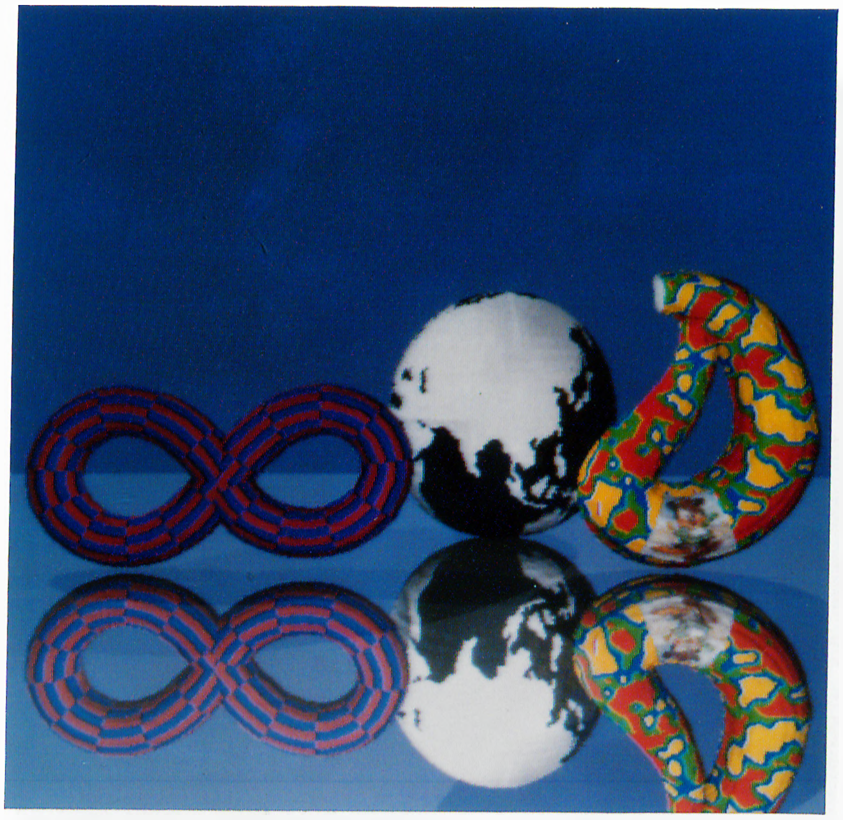

Figure A.5: A point sampled image

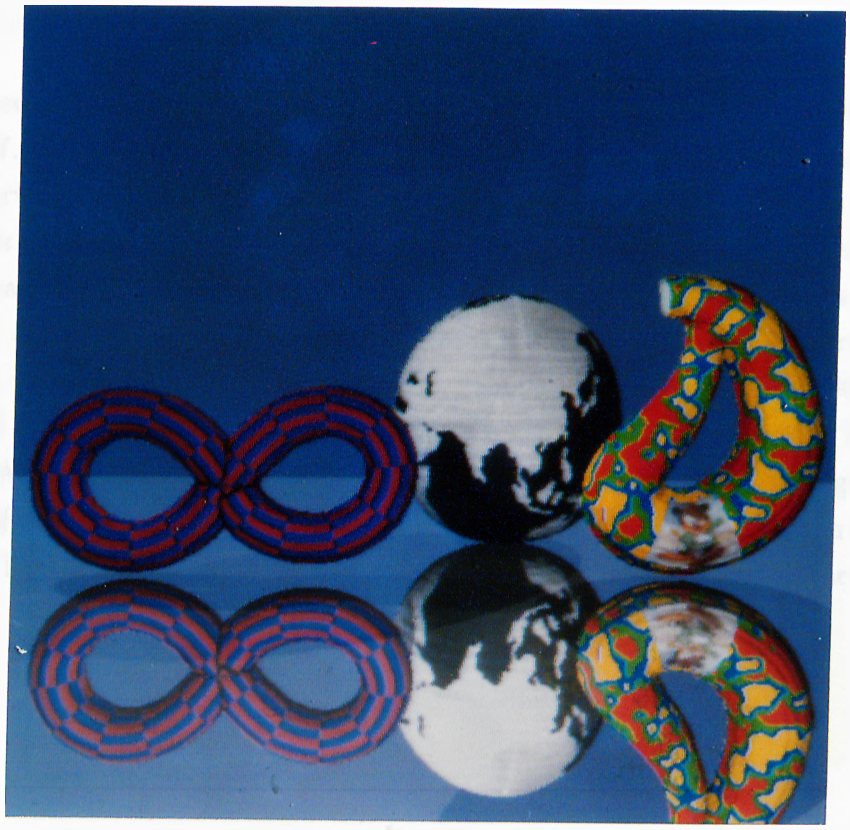

Figure A.6: A prefiltered image

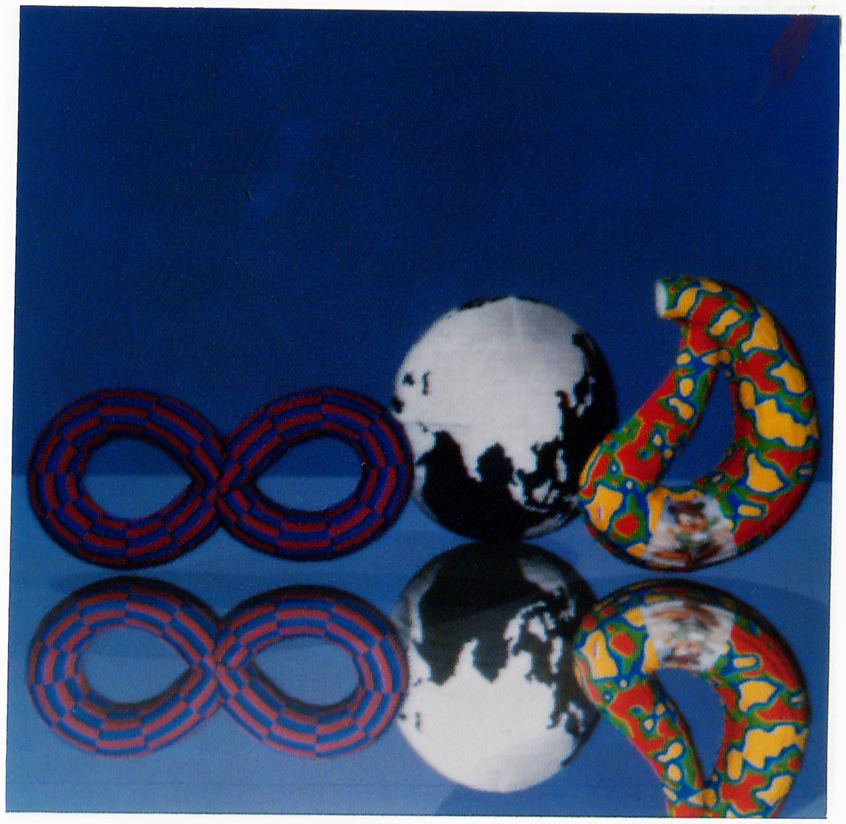

Figure A.7: A prefiltered and supersampled image

Three different versions of a scene are presented in Figures A.5, A.6, and A.7. The '8' shaped object is created by sweeping a circle shaped contour curve around an '8' shaped trajectory curve. The world chart mapped object is created by rotating a half circle around a rotational axis. Klein bottle is an example for a profiled sweep object. It is created by sweeping a varying contour curve around a '6' shaped trajectory curve. It has also a procedural texture. Each of objects in Figures A.5, A.6, and A.7 consists of 9800 triangles and has a reflectivity. It took about 25 minutes to produce the image in Figure A.5 and 30 minutes to produce the prefiltered version in Figure A.6. The prefiltered and supersampled image in Figure A.7 is produced in about 2 hours. The difference in Figure A.7 can be seen at the edges of the objects easily.

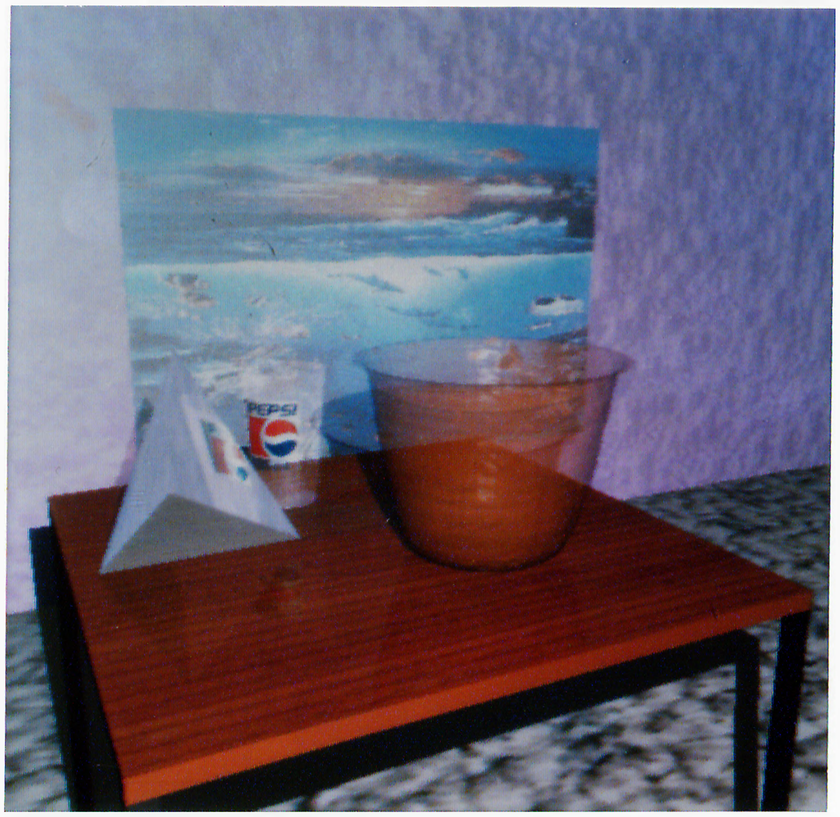

Figure A.8: A point sampled image including semi-transparent objects

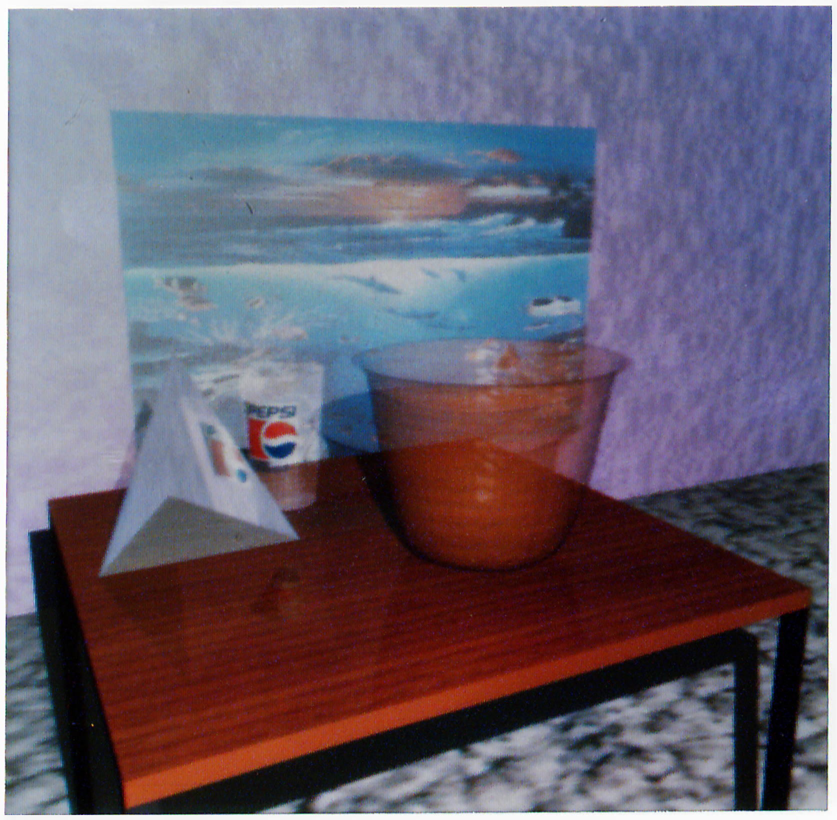

Figure A.9: A prefiltered and supersampled image including semi-transparent objects

Figure A.8 is produced by point sampling and Figure A.9 is prefiltered and supersampled. The difference can be seen at the 'sea' texture. Also note how the texture is filtered at the back of the water glass. The staircases disappear in Figure A.9. The object seen at the left on the table is made from glass. It is put there in order to show internal reflection. Procedural texture on the wall is generated by adding a high frequency noise to a clamped low frequency noise function [1]. The scene seen in Figures A.8 and A.9 consists of about 14500 triangles. It took about 30 minutes produce the image seen in Figure A.8 and 3.5 hours to produce the prefiltered and supersampled version seen in Figure A.9. Refraction and transparency increase the execution time considerably.

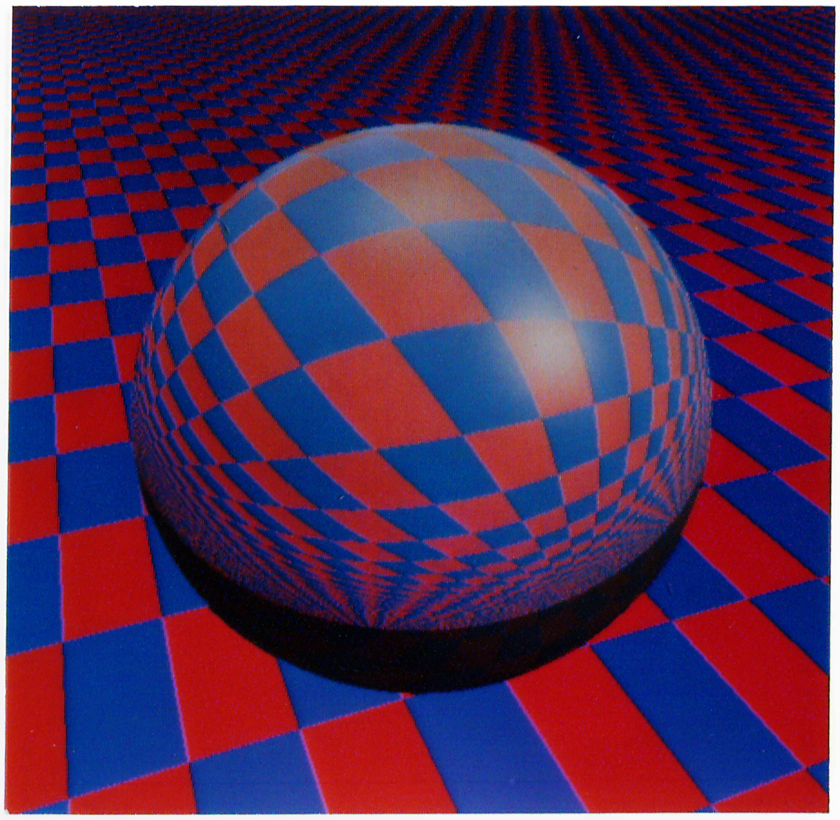

Figure A.10: A glass sphere above a checker board pattern showing aliasing

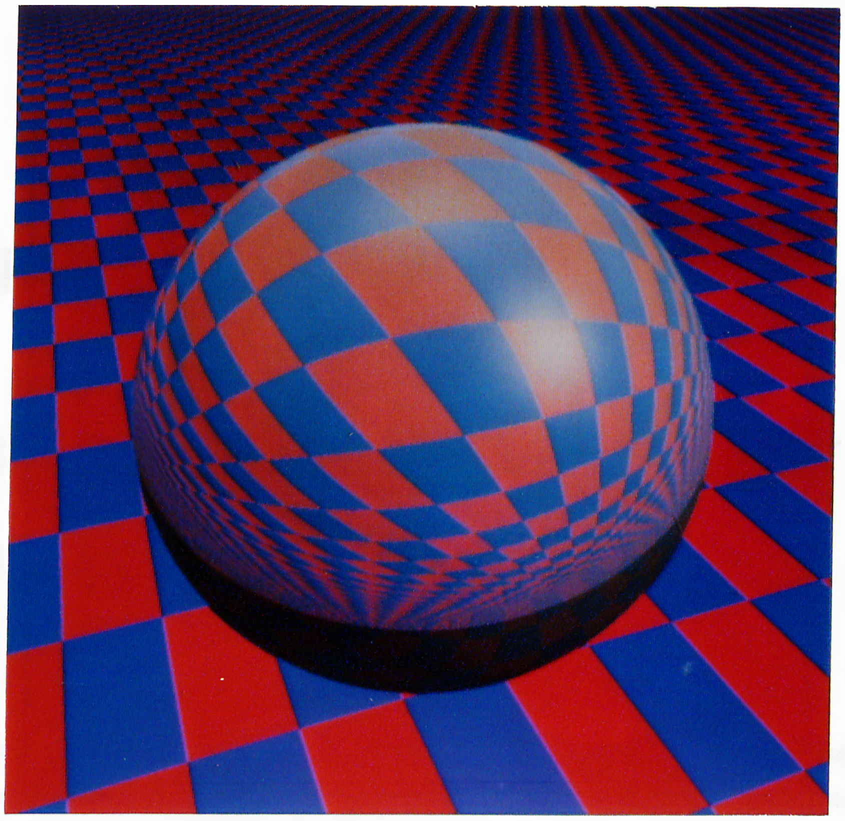

Figure A.11: A prefiltered and supersampled image showing a glass sphere above a checker board

The glass sphere in Figures A.10 and A.11 is not a sweep object. Note how severe aliasing artifacts can occur if filtering is not used. It took about 10 minutes to produce Figure A.10 and 50 minutes to produce the prefiltered and supersampled version seen in Figure A.11.

Appendix B

Technical Details of the Ray Tracing Package

The executable form of T♭ can be found in the directory *~akdemir/ths/Tb* and the source code of the ray tracer can be found in the directory *~akdemir/ths/rt* on the Sun computer system at Bilkent University.

Objects produced by T♭ can be subdivided into triangles by using the program *triangulate* in the directory *~akdemir/ths/maketri.*

The user interface system of T♭ is based on the Sunview[1] windowing system.

The source files of the ray tracer consist of a total of about 5000 lines. The compilation of the source files of the ray tracer is accomplished by using the `make` command. The compilation takes 2.5 minutes. The size of the compiled code is about 300 kilobytes. The program can be executed as follows:

```
ray_trace < input_file
```

An example input file can be found in the directory *~akdemir/ths/rt/Inp.* The ray tracer produces a gamma corrected output file in 24-bit RGB format.

[1]Sunview is a registered trademark of Sun Microsystems, Incorporated.

Bibliography

[1] Abram, G. D., and Whitted, T., "Building Block Shaders," *Computer Graphics (Proc. of SIGGRAPH '90)*, 24(4), pp. 283-288 (1990).

[2] Akman, V., and Arslan, A., "Sweeping with All Graphical Ingredients in a Topological Picturebook," *Computers & Graphics*, 16(3), pp. 273-281 (1992).

[3] Akman, V., and Arslan, A., "An Electronic Topological Picturebook," paper presented in *NATO ASI on Cognitive and Linguistic Aspects of Geographic Space*, Las Navas del Marques, Spain (1990).

[4] Akman, V., Arslan, A., and Franklin, W. R., "Implementing a Topological Picturebook," *Proc. of Thirteenth IMACS World Congress on Computation and Applied Mathematics*, Dublin (1991).

[5] Akman, V., Arslan, A., and Franklin, W. R., "Excursions to a Topological Picturebook," *Proc. of First International Conference on Computational Graphics and Visualization Techniques*, Sesimbra, Portugal (1991).

[6] Amanatides, J., "Ray Tracing with Cones," *Computer Graphics (Proc. of SIGGRAPH '84)*, 18(3), pp. 129-135 (1984).

[7] Amanatides, J., and Woo, A., "A Fast Voxel Traversal Algorithm for Ray Tracing," *Proc. of Eurographics '87*, pp. 3-10 (1987).

[8] Arslan, A., "An Electronic Topological Picturebook," Ph.D. Thesis, Department of Computer Engineering and Information Science, Bilkent University (1992).

[9] Arvo, J., and Kirk, D., "A Survey of Ray Tracing Acceleration Techniques," in Glassner A. (ed.), *An Introduction to Ray Tracing*, Academic Press, pp. 201-262 (1991).

[10] Barr, A. H., "Superquadrics and Angle-Preserving Transformations," *IEEE Computer Graphics & Applications*, 1(1), pp. 11-23 (1981).

[11] Bennis, C., Vězien, J. M., and Iglésias, G., "Piecewise Surface Flattening for Non-Distorted Texture Mapping," *Computer Graphics (Proc. of SIGGRAPH '91)*, 25(4), pp. 237-246 (1991).

[12] Bier, E. A., and Sloan, K. R., "Two-Part Texture Mapping," *IEEE Computer Graphics & Applications*, 6(9), pp. 40-53 (1986).

[13] Blinn, J. F., and Newell, M. E., "Texture and Reflection in Computer Generated Images," *Communications of the ACM*, 19(10), pp. 542-547 (1976).

[14] Blinn, J. F., "Simulation of Wrinkled Surfaces," *Computer Graphics (Proc. of SIGGRAPH '78)*, 12(3), pp. 286-292 (1978).

[15] Carey, R. J., and Greenberg, D. P., "Textures for Realistic Image Synthesis," *Computers & Graphics*, 9(2), pp. 125-138 (1985).

[16] Catmull, E. A., "Computer Display of Curved Surfaces," in Freeman H. (ed.), *Tutorial and Selected Readings in Interactive Computer Graphics*, New York (IEEE), pp. 309-315 (1980).

[17] Celniker, G., and Gossand, D., "Deformable Curve and Surface Finite-Elements for Free-Form Shape Design," *Computer Graphics (Proc. of SIGGRAPH '91)*, 25(4), pp. 257-266 (1991).

[18] Cook, R. L., Porter, T., and Carpenter, L., "Distributed Ray Tracing," *Computer Graphics (Proc. of SIGGRAPH '84)*, 18(3), pp. 137-145 (1984).

[19] Cook, R. L., "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics*, 5(1), pp. 51-72 (1986).

[20] Crow, F. C., "A Comparison of Antialiasing Techniques," *IEEE Computer Graphics & Applications*, 1(1), pp. 40-48 (1981).

[21] Crow, F. C., "Summed Area Tables For Texture Mapping," *Computer Graphics (Proc. of SIGGRAPH '84)*, 18(3), pp. 207-212 (1984).

[22] Fiume, E. L., Fournier, A., and Canale, V., "Conformal Texture Mapping," *Proc. of Eurographics '87*, pp. 53-64 (1987).

[23] Fiume, E. L., *The Mathematical Structure of Raster Graphics*, Academic Press (1989).

[24] Fujimoto, A., Tanaka, T., and Iwata, K., " ARTS: Accelerated Ray Tracing System," *IEEE Computer Graphics & Applications*, 6(4), pp. 16-26 (1986).

[25] Glassner, A., "Adaptive Precision in Texture Mapping," *Computer Graphics (Proc. of SIGGRAPH '86)*, 20(4), pp. 297-306 (1986).

[26] Greene, N., and Heckbert P. S., "Creating Raster Omnimax Images Using the Elliptically Weighted Average Filter," *IEEE Computer Graphics and Applications*, 6(6), pp. 21-27 (1986).

[27] Haines, E., "Essential Ray Tracing Algorithms," in Glassner A. (ed.), *An Introduction to Ray Tracing*, Academic Press, pp. 33-77 (1991).

[28] Hall, R. A., and Greenberg, D. P., "A Testbed for Realistic Image Synthesis," *IEEE Computer Graphics & Applications*, 3(8), pp. 10-20 (1983).

[29] Heckbert, P. S., and Hanrahan, P., "Beam Tracing Polygonal Objects," *Computer Graphics (Proc. of SIGGRAPH '84)*, 18(3), pp. 119-128 (1984).

[30] Heckbert, P. S., "Filtering by Repeated Integration," *Computer Graphics (Proc. of SIGGRAPH '86)*, 20(4), pp. 315-321 (1986).

[31] Heckbert, P. S., "Survey of Texture Mapping," *IEEE Computer Graphics & Applications*, 6(11), pp. 56-67 (1986).

[32] Heckbert, P. S., and Moreton, H. P., "Interpolation for Polygon Texture Mapping and Shading," in Rogers D. F., and Earnshaw R. A. (eds.), *State of the Art in Computer Graphics (Visualization & Modelling)*, Springer Verlag (1991).

[33] İşler, V., and Özgüç, B., "Fast Ray Tracing 3D Models," *Computers & Graphics*, 15(2), pp. 205-216 (1991).

[34] Kelley, A., and Pohl, I., *A Book on C*, Benjamin/Cummings (1984).

[35] Lee, M. E., Rodner, R. A., and Uselton, S. P., "Statistically Optimized Sampling for Distributed Ray Tracing," *Computer Graphics (Proc. of SIGGRAPH '85)*, 19(3), pp. 61-67 (1985).

[36] Peachey, D. R., "Solid Texturing of Complex Surfaces," *Computer Graphics (Proc. of SIGGRAPH '85)*, 19(3), pp. 279-286 (1985).

[37] Perlin, K., "An Image Synthesizer," *Computer Graphics (Proc. of SIGGRAPH '85)*, 19(3), pp. 287-296 (1985).

[38] Programming Utilities for the Sun Workstation, *Lex and Yacc*, pp. 118-182, Sun Microsystems, Inc., (1986).

[39] Purgathofer, W., "A Statistical Method for Adaptive Stochastic Sampling," *Proc. of Eurographics '86*, pp. 145-152 (1986).

[40] Rubin, S., and Whitted, T., "A Three Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics (Proc. of SIGGRAPH '80)*, 14(3), pp. 110-116 (1980).

[41] Shinya, M., and Tokiichiro, T., "Principles and Applications of Pencil Tracing," *Computer Graphics (Proc. of SIGGRAPH '87)*, 21(4), pp. 45-54 (1987).

[42] Speer, L. R., DeRose, T. D., and Barsky, B. A., "A Theoretical and Empirical Analysis of Coherent Ray Tracing," *Computer-Generated Images (Proc. of Graphics Interface '85)*, pp. 11-25 (1986).

[43] Sung, K., "Area Sampling Buffer: Tracing Rays with Z-Buffer Hardware," *Proc. of Eurographics '92*, 11(3), pp. 299-310 (1992).

[44] Watt, A., *Fundamentals of Three Dimensional Computer Graphics*, Addison-Wesley (1989).

[45] Whitted, T., "An Improved Illumination Model for Shaded Display," *Comm. ACM*, 23(6), pp. 343-349 (1980).

[46] Williams, L., "Pyramidal Parametrics," *Computer Graphics (Proc. of SIGGRAPH '83)*, 17(3), pp. 1-11 (1983).