Fall 2014

Generating Surfaces of Variable Eccentricity Within a Ray Tracer

Joshua A. Smith

GENERATING SURFACES OF VARIABLE ECCENTRICITY WITHIN A RAY TRACER

by

JOSHUA A. SMITH

(Under the Direction of Yingkang Hu)

ABSTRACT

Polynomial surfaces used in ray tracing have recently been improved upon allowing for three dimensional applications. Among these are surfaces that have a varying eccentricity. This paper will discuss a method for finding real roots of polynomials [allowing us to create these surfaces]. First, we will give the reader a basic comprehension of the workings of a ray tracer, a general understanding of three dimensional polynomial surfaces, how this newly implemented root finder functions, and how these concepts enable us to create surfaces of variable eccentricity. Then, examples will be provided to demonstrate the capabilities of the program.

Key Words: Ray Tracing, Bernstein Polynomials, Polynomial Surfaces, Algebraic Surfaces, Elliptic surfaces with varying eccentricity, polynomial root finders

2009 *Mathematics Subject Classification*: Applied Mathematics

GENERATING SURFACES OF VARIABLE ECCENTRICITY WITHIN A RAY TRACER

by

JOSHUA A. SMITH

B.S. in Physics

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial Fulfillment

of the Requirement for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

2014

GENERATING SURFACES OF VARIABLE ECCENTRICITY WITHIN A RAY TRACER

by

JOSHUA A. SMITH

Major Professor: Yingkang Hu

Committee: Scott Kersey

Xiezhang Li

Jiehua Zhu

Electronic Version Approved:

December 12, 2014

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer graphics design has come a long way since its start. When smooth surfaced objecets are involved, their boarders may be approximated by mathematical functions. One of the more common techniques practiced today is the projection-styled algorithm which generates scenes with the objects in the scene being 'projected' with all of the information regarding them done locally. However, there is another method, known as tge image-space algorithm, that draws scenes pixel by pixel using information from the scene to determine how light affects the color of any selected pixel. It creates each pixel of the image the viewing plane sees by tracing a virtual ray to an object and seeing how the different elements alter the pixel in question, thus the term 'ray-tracing.' [12] Ray tracing is a very accurate model for light's effect on the object's color, but challenges arise when dealing with complex mathematical surfaces. Here, we restrict ourselves to dealing with algebraic surfaces defined with multivariate polynomials. Previously, in the ray tracer we use methods implemented for drawing surfaces of revolution defined by polynomials. After introducing the inner workings of a ray tracer, we will describe the mathematical definition of our surfaces, how the Bernstein basis root finder works, and the general principel(s) behind the computer code that draws these objects. We will also use the program to provide examples of surfaces of variable eccentricity.

CHAPTER 2

BASIC INTRODUCTION TO RAY TRACING

Ray-tracing involves defining all of the traits within a *world*, then using various information from said world to render a textitscene pixel by pixel. But, we need to understand the basic inner workings of a ray-tracer. Primarily, we need to create objects, their materials, some light source(s), and define a window that the camera sees and tracer draws. Then, we need to *ray cast* from the camera to the window so we obtain an image.

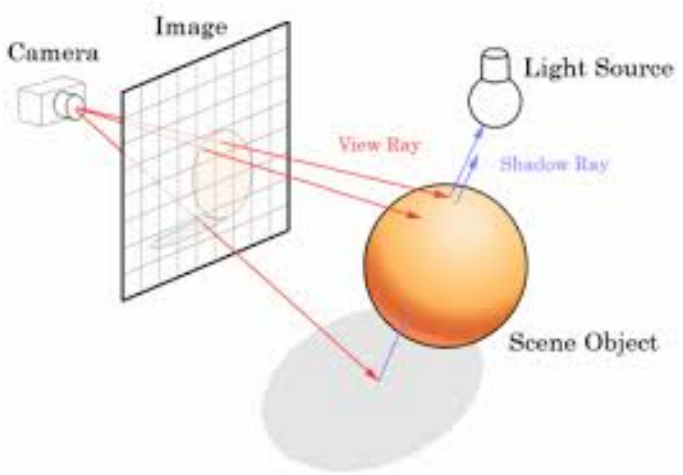

Figure 2.1: The camera, sending rays through pixels along a view plane and gathering information based on the object it contacts and the light source located within the world. Image taken from Wikipedia.

When ray casting, you shoot rays toward each pixel and the objects determine the closest point of contact, if the ray hits at all. Then you use the information gathered from the hit location (i.e. material, light, etc.) to generate the color and brightness of any pixel in question. Should the ray hit nothing you set the color of the background, or color based on a pre-defined image. Then, you repeat this process for every pixel in the window.

When the tracer sends out a ray, it draws the first point the tracer hits, since it is on the closest side of the nearest object from the camera along the ray's path, it should cover and obscure all other hit points, if any. In this case, the tracer identifies the hit point by discenring the smallest, positive value that t, a parameter representing the hit point of the ray, assumes. We only consider positive values of t as $t = 0$ is a location on the camera, and any negative t represents a point behind our field of view. Neither is seen by the camera.

2.1 Building a World

At this point we will explore all of the elements that go into making a world and rendering a scene and describe how these come into play while forming a world.

2.1.1 Objects

In order to generate an object we need a shape, a material, and color. For the shape, we can use basic geometric definitions. For example, to generate a sphere we would need a center point (p_0) and a radius (r). Using this information we can make a code that states: if a ray passes through a point located a distance r from p_0, that ray hits the sphere. Then we gather any other infromation aquired from the process to determine the pixel color. For a cube, we can define two opposite corner points and see if a ray passes through any x, y, or z coordinates between them. However, the t value only determines where the object is. Normal vectors, colors and materials are used for shading the object and making it appear 3D.

In order to vary the pixel color, we can define some material that creates the illusion of texture. This material tells the rays how light reflects off the object. The material determines the reflective nature at any given point. Some basic materials are *matte* and *phong*. For the *matte* material we have a diffuse shading that creates

a 'dull' reflection. For *phong*, we have a specular aspect which generates a 'shiny' finish.

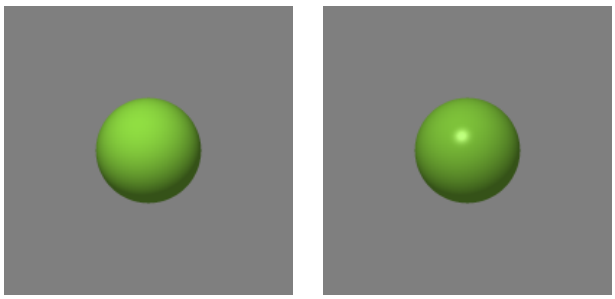

Figure 2.2: Spheres with Matte finish (a) and Phong finish(b).

More complex materials are created by mapping an image onto a surface. Some even alter the normal vectors registered by the ray; these are called bump, or normal, mappings which can be generated in various other applications.

The whole purpose of shading is to determine the color and brightness of pixels. For this we have a C++ class known as RGBColor, which uses a 3D point to define a color where x represents red, y represents green and z represents blue. The numbers go from 0 to 1 and determine the intensity of the color. By using multiple components you can mix the colors at different intervals to make other colors such as purple (1,0,1) and orange (1,1,0). The set of all colors that can be represented by a RGB "point" is called *color gamut*. Using the normal of an object, its position relative to the window and light intensity, you can alter the color by using simple numeric values to accurately describe the color observed at a pixel.

2.1.2 Light Sources

If a single color is defined for an object, the ray returns a solid color regardless of the three dimensional shape of the object. Using some light source and the normal vector at the point of contact, we can calculate the color intensity of the surface to generate

a realistic shading effect based on material. For this we need to create a light of some nature. The color intensity is calculated by the *bidirectional reflectance distribution function* (BRDF) which uses the normal of an object to calculate how light reflects off its surface. Using this function and material, it is possible to create various surface types, including a mirror like reflective surface. There are three general types of light sources used: ambient, directional and point light.

For ambient light, the program generates an overall light intensity that permiates throughout the world. This is used to brighten areas that aren't lit by any other source which causes a minimum color intensity on all objects.

Figure 2.3: Ambient light at different intensities from lowest to highest.

Directional lights hit the entire scene from a given direction. The difference from ambient lighting is that a directional light casts shadows onto the ground and other objects, generating a 'sunlight' effect. For a single source (such as a lamp or light bulb) we have point lights, which radiate light from a point outward.

By using all three light sources we can create accurate shadows and build atmosphere. However, as the color is also determined by intensity, and the maximum intensity of any of the three primary colors is 1, the amount of light can actually be a problem because it causes an *out-of-gamut* effect. At such a point 'clamping' is needed to set the value for any of the three prime colors back to 1 before the tracer draws the scene, which results in an incorrect shading. So, light intensity is a crit-

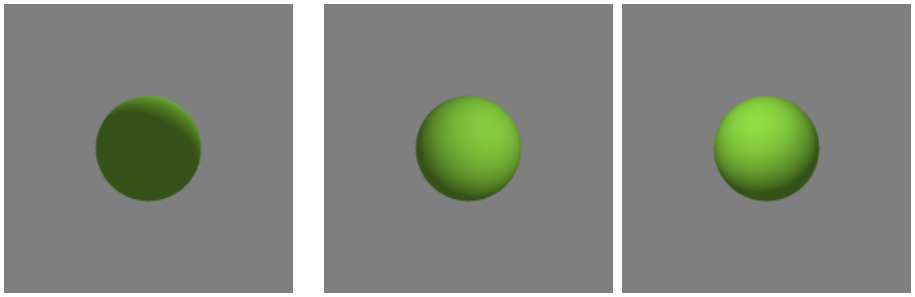

Figure 2.4: Directional light from the back (a) and front (b) as well as a Point Light example (c).

ical concern for people geared towards building a scene. Note that it is possible to generate an object that emits light, as well as defining the color of said light source.

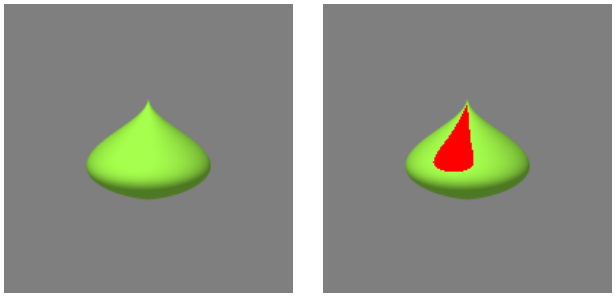

Figure 2.5: Demonstrating out of gamut effect (a), while clamping out the area, painted in red for clarity (b).

2.1.3 Windows

One can build a world with defined objects and light sources but cannot render the scene without a window to view it. Using the ViewPane class we can create a window, covered in pixels, whose dimensions we define. There are a few aspects to this: the view, its dimensions in terms of position, the number of pixels in the rows and columns and the size of those pixels. The window dimensions affect the size of our image, more pixels means a larger window and thus more of the world is rendered within the scene.

Pixel size effects the 'zoom' of the view plane, with a smaller pixel size showing less of the world and a larger pixel size showing more. But, a smaller pixel size is more accurate at drawing objects than a larger one.

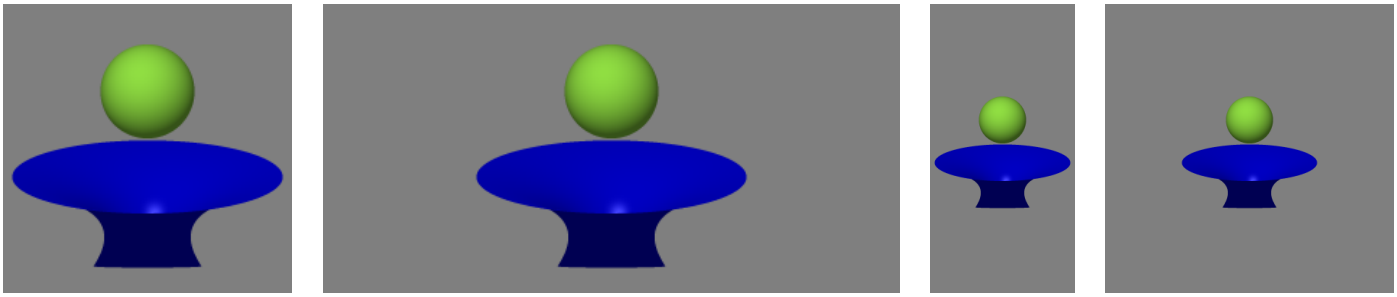

Figure 2.6: Demonstration of changing the different dimensions of the view plane. This does not affect the image, only what is seen.

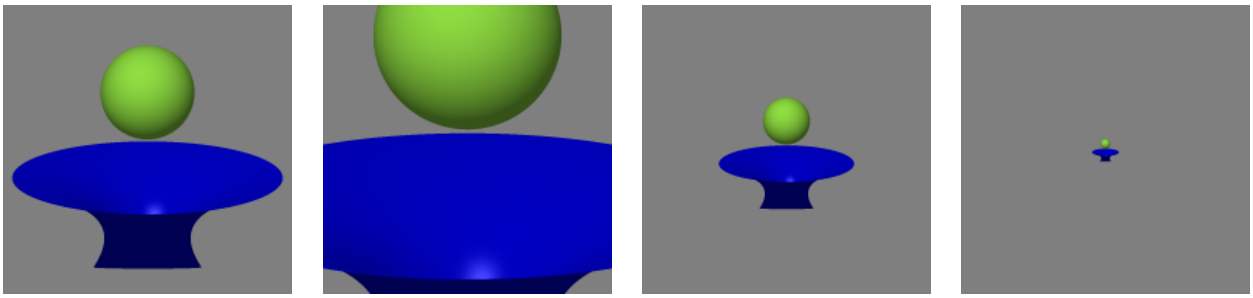

Figure 2.7: Showing Change in pixel size. (a) Control: size: 0.01, (b) Size: 0.005, (c) Size: 0.02, (d) Size: 0.1

One aspect of the window is a phenomenon known as *aliasing*, and its counter measures *anti-aliasing*. The computer picks a color based on what a ray sees at that pixel. However, the ray, being shot from the center of the pixel, may register a color that does not accurately describe what is contained by the whole pixel. More complex images can cause a worse aliasing effect, generating a totally incorrect pattern. There are three general ways to account for this; primarily by changing the window dimensions and pixel size, or by increasing the sampling size for each pixel and/or

how the tracer samples each pixel. When we increase the size of the window and decrease the pixel size proportionatelly to create a window with the same image, we also increase the resolution making the image appear less pixillated.

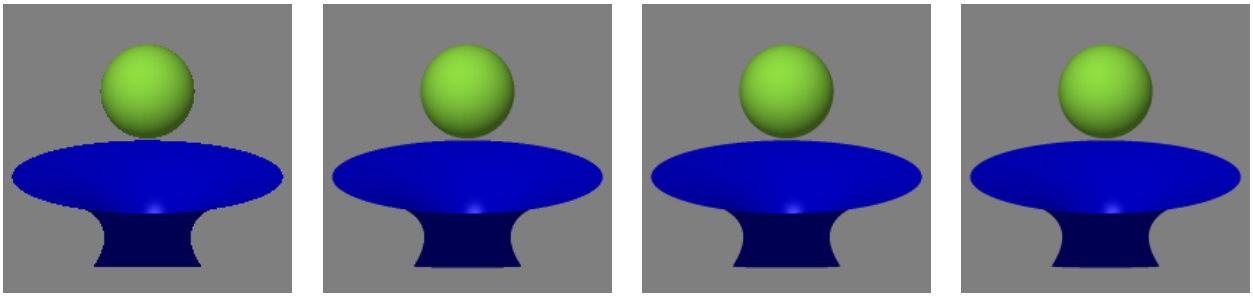

Figure 2.8: Demonstrating the difference caused by increasing the number of samples taken per pixel. (a) Samples: 1, (b) Samples:4, (c) Samples: 9, (d) Samples: 16 Due to the current resolution capabilities, it is difficult to distinguish between higher sampling sizes.

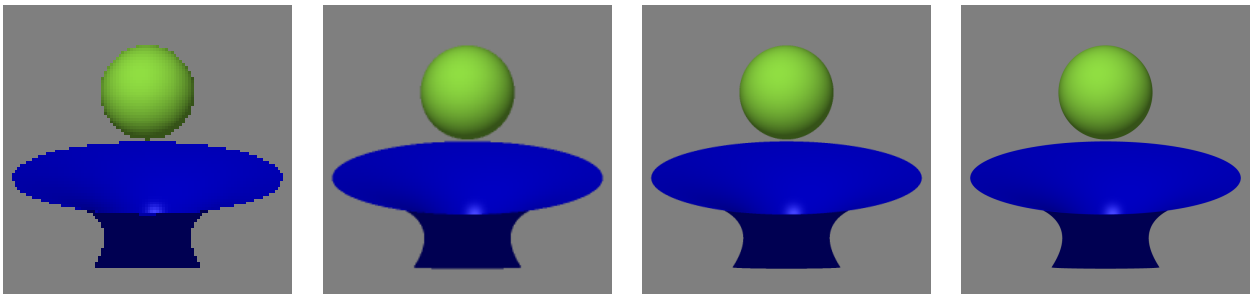

Figure 2.9: Showing resolution change by altering all parameters. (a) Dimensions:100X100, Pixel Size: 0.02 Samples: 1, (b) Dimensions:200X200, Pixel Size: 0.01 Samples: 4, (c) Dimensions:400X400, Pixel Size: 0.005 Samples: 9, (d) Dimensions:800X800, Pixel Size: 0.0025 Samples: 16. There is little difference between (c) and (d) as the resolution of the screen does not allow a clearer image.

As for the sampling, by increasing the number of rays shot through each pixel and taking an average color over the entire pixel, we can get better, more natural looking,

colorings. But, the distribution of samples is equally important. Generally, there are various ways to sample by either a uniform distribution, or one of many ways to randomize where each ray passes through the pixel, known as *jittered* sampling. Each method is described in Chapter 5 of [12] and has its own advatages and disadvantages.

After we have a viewing window, the image seen through the window depends on where we are viewing from. This is known as perspective viewing. By creating an 'eye' (camera) position we determine the veiwing angle. This also enables us to keep a scene and simply change where we view it without the hassle of changing all of the object positions and rotations.

2.2 Rays

Without an understanding of the various types of rays, we cannot create accurate light and shading effects. There are four types of rays, *primary*, *secondary*, *light*, and *shadow*, which interact with objects and generate a scene. There are also ways to render more complex objects faster using bounding surfaces. Using the numerical information, we can also use matrices to scale, translate and rotate objects.

2.2.1 Types of Rays

Of the four types of rays, the simplest, and most important, are the primary rays. These are cast from the eye, through the view plane and register when they hit an object. The goal of the primary ray is to find out the closest hit point to the view plane. Secondary rays come in two varieties and, unlike primary rays, originate from the surfaces of objects. Their purpose is to determine how light is either reflected off, or transmitted through a material, giving them their names: *reflected rays* and *transmitted rays*. Reflected rays are used with mirrors, as well as any surface that reflects some amount of light. This topic is reffered to as *caustics*. For example, if you

were to place a green sign near a white wall in bright daylight, the sign would bounce green light off of the sign and onto the wall, giving it a green coloring. Transmitted rays use Snell's Law and a given index of refraction to determine the angle of refraction through a refractive surface, such as glass or water.

Light rays are cast from a light source and simulate the effects of illumination on an object. Shadow rays, like secondary rays, are cast from the surface of objects and determine if there is anything between the surface and the light sources. Using the information that the shadow rays acquire the tracer will create shading for the surface.

2.2.2 Ray/Object Interaction

When the program tries to register the hit location of the object, due to memory errors, the exact hit point may be slightly before or slightly after the object. When this occurs, the tracer may think the hit point is inside the object, where no light reaches. This effect makes speckles of black cover the object's surface, as if the image has 'static'. To account for this we define a small number, ϵ, that creates a virtual 'thickness' prevents this anomoly by allowing for a small error.

For complex shapes the runtime on the rendering can be problematic. To generate these objects faster we use a *bounding object*. Bounding objects are defined to be larger than the object they cover, and less expensive to trace, with bounding boxes, spheres and cylinders being most common. In fact bounding objects are referred to as bounding surfaces. By using bounding surfaces, the tracer will bypass any objects whose bounding surfaces are not hit by the ray.

2.3 Transformations

We can also change the shape of an object with linear transformations. Transformations work as one would suspect, by using matrices to alter the 3D coordinates of the surface of the object. The types of transformations used are scaling, rotation, reflection, translation and shearing.

2.3.1 Affine Transformations

Scaling, translation, rotation, reflection and shearing are all *affine transformations*, and can be represented as a series of linear transformations. Because of this representation, we can use square matrices to perform these transformations to an object. We can use 3×3 matrices to scale, rotate, reflect and shear an object, however translation is problematic as a 3×3 matrix does not have enough degrees of freedom. Thus, we need to use a 4×4 matrix that will have components which account for all affine transformations, if we represent each point or vector $(x, y, z)^T$ as $(x, y, z, 1)^T$.

To scale a component we need to multiply it by some constant. To account for all components we use the following matrix (u, v, and w represent constants.):

$$S(u, v, w) = \begin{bmatrix} u & 0 & 0 & 0 \\ 0 & v & 0 & 0 \\ 0 & 0 & w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This generates the scaling effect ($[ux, vy, wz, 1]^T$) for our object. For translations, we must add three constants to x, y, and z. The matrix that does this is the following, which adds d_x, d_y, and d_z to their respective components.

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix will add the values associated with d_x, d_y, and d_z to each component in turn, causing a translation. For reflection along any axis we simply need to multiply the desired component by -1.

$$R_x = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For rotation we use the following matrices:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shearing is the effect of 'sliding' an object so that it is skewed along an edge. There are six possible ways to skew an object in 3D with three axies to choose from and two afterwards to skew with respect to. The matrix generated becomes:

$$R_y = \begin{bmatrix} 1 & h_{yx} & h_{zx} & 0 \\ h_{xy} & 1 & h_{zy} & 0 \\ h_{xz} & h_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Because matrix multiplication is not commutative, except under special circumstances, one should be wary of the order they transform an object.

How do we ray-intersect a transformed object? It is just as complicated to implement as it sounds. Fortunately, we can bypass this by intersecting the original object with the inversely transformed ray. To be more concise, let T denote the matrix for the total effect of all affine transformations, defined as the product of all matrices representing each individual transformation. Then T^{-1} represents the inverse transformation. The inversely transformed ray is obtained by multiplying the origin point and the direction vector of the ray by T^{-1} from the left. Denote the intersection point of the original ray and transformed object as p', with its normal vector n'. Similarly, we can describe the intersection point of the inversely transformed

ray and original object by p, as well as its normal vector n. Then it is not difficult to prove that $p' = Tp$, and that the two points will have the same value of t. The relation between the normal vectors is $n' = T^{-T}n$, where T^{-T} is the transpose of T^{-1}.

CHAPTER 3

THREE DEMINSIONAL POLYNOMIAL SURFACES

Now, we will take the time to discuss how polynomials can be used to generate the surface of an object and how it relates to the development of the SAVE and sqrtSAVE classes.

3.1 Polynomial Surfaces

As previously explained, when drawing within a ray-tracer one needs to define the surface based on some input. In order to ray-trace any *implicit* polynomial surface

$$f(x, y, z) = 0, \tag{3.1}$$

by any given ray

$$x = a_0t + b_0, y = a_1t + b_1, z = a_2t + b_2, \tag{3.2}$$

we need to substitute (3.2) into (3.1) and obtain a ray-surface intersection equation of a single variable t:

$$F(t) := f(a_0t + b_0, a_1t + b_1, a_2t + b_2) = 0. \tag{3.3}$$

In order to implement this in a ray-tracer, we need to find the smallest positive solution of (3.3) for t. There are already root finders implemented within our ray tracer. Note a similarity between this and computed tomography (CT) where ray tracing needs only the closest (real) root while CT modeling needs all real roots [6].

For polynomails of degree $n \leq 4$, there exist analytic root formulas (such as the quadratic formula) to deal with them, but according to the Abel-Ruffini theorem, when $n \geq 5$ no such general algebraic formula exists. This reason kept many people away from higher degree surfaces for quite some time. There is another major

complication involved. The first is simplifying (3.3) into the standard basis:

$$F(t) = \sum_{i=0}^{n} c_i x^i. \tag{3.4}$$

Fortunetely, the Polynomial class generates (3.4) upon defining the polynomial.

3.2 Rotated Polynomials

Some three dimensional objects are polynomials rotated along a single axis. To design a class for generating them, first an abstract class PolySurf is created to handle the basic information. Then, a child class, RotPoly, is used in order to create polynomials of the form

$$x^2 + y^2 = p(z)^2, \quad \alpha \leq z \leq \beta, \tag{3.5}$$

which generates an object with a horizontal cross secton that is a circle of radius $|p(z)|$. However, (3.5) is a symmetric object about the z-axis and can be improved upon to allow more objects to be generated. In order to change the center, PolySurf has two more polynomial members, p_x and p_y, that can be used. The equation becomes

$$(x - p_x(z))^2 + (y - p_y(z))^2 = p(z)^2, \quad \alpha \leq z \leq \beta, \tag{3.6}$$

where

$$x = p_x(u), \quad y = p_y(u), \quad z = u$$

is the 3D parametric curve the object rotates around, with its degree being

$$2 * max(deg(p), deg(p_x), deg(p_y), 1).$$

One thing to keep in mind is the overall shape of the object as the maximum number of extrema in a curve is one less than the degree of its polynomial. We find that generating more 'turning points' is problematic as $p(z)$'s max degree is doubled from the outset, thus slowing down the computation. However, if we use

$$(x - p_x(z))^2 + (y - p_y(z))^2 = p(z), p(z) \geq 0, \alpha \leq z \leq \beta, \tag{3.7}$$

we can define a child of Rotpoly, named RotSqrtPoly whose degree of intersection equation is evaluaed as

$$max(deg(p), 2 * (deg(p_x), deg(p_y), 1)).$$

With this in hand we can now generate a cross sectional radius of $\sqrt{p(z)}$. But there is still one limit: all of the horizontal cross sections are either circles or ellipses with the same eccentricity.

3.3 Surfaces Allowing Varying Eccentricity

Within an ellipse,

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1,$$

we can define $r_1 = min(a, b)$ and $r_2 = max(a, b)$, the eccentricity will be

$$e = \frac{\sqrt{{r_2}^2 - {r_1}^2}}{r_2} = \sqrt{1 - \frac{{r_1}^2}{{r_2}^2}}.$$

In order to vary it, we must convert a and b into polynomials of z and use another inherited class SAVE to solve the following equation:

$$\frac{(x - p_x(z))^2}{a(z)^2} + \frac{(y - p_y(z))^2}{b(z)^2} = p(z)^2, a(z) > 0, b(z) > 0, \alpha \leq z \leq \beta, \tag{3.8}$$

which can be rewritten as the polynomial

$$((x - p_x(z))b(z))^2 + ((y - p_y(z))a(z))^2 = a^2(z)b^2(z)p^2(z), p(z) \geq 0, \tag{3.9}$$

with a maximum degree of

$$2 * max(deg(p) + deg(a) + deg(b), max(deg(p_x), 1) + deg(b), max(deg(p_y), 1) + deg(a)).$$

This is a rotated polynomial whose eccentricity varies based on $\frac{b(z)}{a(z)}$. Like the rotated polynomials, the degree of p, which determines the number of turning points of the

outline of the solid, is doubled here. Like before, we will create the sqrtSAVE class by removing the square on $p(z)$.

$$\frac{(x-p_x(z))^2}{a(z)^2}+\frac{(y-p_y(z))^2}{b(z)^2}=p(z), p(z)\geq 0, a(z)>0, b(z)>0, \alpha\leq z\leq\beta. \quad (3.10)$$

This equation can be rewritten as the polynomial

$$((x-p_x(z))b(z))^2+((y-p_y(z))a(z))^2=a(z)^2b(z)^2p(z), \quad (3.11)$$

whose maximum degree is

$$max(2(deg(a)+deg(b))+deg(p), 2(max(deg(p_x),1)+deg(b)), 2(max(deg(p_y),1)+deg(a))).$$

The lowest degree that we need to make a significant surface of varying eccentricity using both eccentricity parameters is four, where the equation giving the lowest degree is:

$$\frac{x^2}{(c_1+c_2x)^2}+y^2=c_3$$

with no central variance, $deg(p)=deg(a)=deg(b)=0$.

CHAPTER 4

BERNSTEIN ROOT FINDING

Previously, a global root finder, the Jenkins-Traub algorithm, was used in our ray tracer. Even though seeking out all roots of a polynomial would be adequite, finding all real and complex roots weighs down the program's processing power. To account for this we use a Bernstein basis root finder that only searches for real roots locally.

4.1 Properties of the Bernstein Basis

Bernstein Polynomials of degree n are defined by

$$B_i^n(x) := \binom{n}{i} x^i (1-x)^{n-i}, i = 0, 1, ..., n, 0 \leq x \leq 1. \tag{4.1}$$

This forms a basis for the space of polynomials of degree $\leq n$ on $[0, 1]$. Because of this, any polynomial $p(x)$ can be written as a Bézier Curve:

$$p(x) = \sum_{i=0}^{n} a_i B_i^n(x). \tag{4.2}$$

The coefficients $a_0, a_1, ..., a_n$ can be used to discern useful infromation about the curve. [10] As such, we will begin examining various properties of the Bernstein Basis and Bézier Curves.

4.1.1 Properties of B_i^n

These are some useful properties of the Bernstein Basis:

positivity: $B_i^n(x) \geq 0$ for $x \in [0, 1]$

partition of unity: $\sum_{i=0}^{n} a_i B_i^n(x) \equiv 1$

recursion: $B_i^n(x) = xB_{i-1}^{n-1}(x) + (1-x)B_i^{n-1}(x)$

derivative: $\frac{d}{dx}B_i^n(x) = n(B_{i-1}^{n-1}(x) - B_i^{n-1}(x))$

curve derivative: $\frac{d}{dx}p(x) = \sum_{i=0}^{n-1} \frac{a_{i+1}-a_i}{1/n} B_i^{n-1}(x)$

In order to fully utilize the Bernstein Basis, we must be able to change the interval $[0, 1]$. To perform this change we must map x onto an arbitrary interval $[\alpha, \beta]$. This is done by the change of variable

$$u = \frac{x - \alpha}{\beta - \alpha} \tag{4.3}$$

which maps $x \in [\alpha, \beta]$ onto $u \in [0, 1]$. [4] This changes (4.2) into

$$p_{[\alpha,\beta]}(x) = \sum_{i=0}^{n} a_i B_i^n(u) = \sum_{i=0}^{n} a_i \binom{n}{i} \frac{(\beta - x)^{n-i}(x - \alpha)^i}{(\beta - \alpha)^n}. \tag{4.4}$$

4.1.2 Variation Diminishing

Another useful property of the Bernstein basis is the ability to generate control points and a control polygon from the coefficients.

Control Points: $P_i = (\alpha + \frac{i}{n}(\beta - \alpha), a_i)$.

Control Polygon: $\overline{P_0 P_1 ... P_n}$

Control Points are used with the *convex hull property* of Bézier curves. This states that the curve lies within the *convex hull* of its contol points. This occurs because of the positivity and partition of unity properties of the Bernstein Basis. This leads to the *variation diminishing property*; a Bernstein Basis version of DeCartes rule of signs says no line intersects the curve more times than it intersects the control polygon. If we were to consider the line to be the x-axis we can conclude that if the x-axis intersects a Bézier curve n_1 times and the control polygon n_2 times then

$$n_1 = n_2 - 2n_3 \tag{4.5}$$

where n_3 is a non-negative integer. In particular, if the control points change signs exactly once, then the polynomial has exactly one root in $[\alpha, \beta]$.

4.2 Real Root Finding

In order to find the roots of a Bernstein polynomial, a technique known as *root isolation* is employed. Commonly referred to as *root localization*, there are various methods and theorems involved, with most localizations being done based on the coefficients of the polynomials. (There are also methods of bounding the roots.) Bernstein basis root finders subdivide using the recursive property on the polynomial and isolate the real root, as well as approximate it. The methods are discussed and various algorithms are defined in [11]. In general, real root finding in the Bernstein basis is a special case for finding the intersection points between two Bézier curves, with one curve being a line. For more information regarding finding intersection points between two Bézier curves see Sederberg and Nishita's work [9].

4.2.1 Convex Hull Marching and Root Deflation

The current Bernstein root finding algorithm our tracer uses is based on two concepts, *convex hull marching* and *root deflation*. The algorithm takes a step-by-step 'march' toward the root from the left, then it deflates the polynomial. The algorithm, known as the *Bernstein convex hull approximating step algorithm*, finds each real root within $[0, 1]$ ($[\alpha, \beta]$ if the conversion mapping is in place) by computing a sequence of subdivided Bézier curves. [10] It looks for the leftmost point where the convex hull intersects the x-axis, since the root cannot exist between α and this intersection, denoted by x_1, (because all roots are inside the hull and the interval $[\alpha, x_1]$ is outside the hull) the algorithm sets x_1 as the new α and generates a new convex hull of the same number of control points using the subdivision algorithm and then searches the new interval of $[x_1, \beta]$ for the next intercept point x_2. It repeats the process until $x_i = x_{i+1}$ within floating point error. This value then becomes the first root. If more

roots are needed, $p(x)$ is deflated and the whole process starts over again. In deflation the program finds a polynomial of degree $n-1$, $p^1_{[x_i,1]}(x)$ which has the same roots as the previous curve, with exception to the root that was previously found. The deflation of the Bézier curve is done by the following formula:

$$p_{[x_i,1]}(x) = (x - x_i)p^1_{[x_i,1]}(x)$$

with the coefficients of $p^1_{[x_i,1]}(x)$ being

$$a^1_j = \frac{n}{j+1}a_{j+1}, j = 0, 1, ..., n-1.$$

The advantage of this algorithm is that it always converges to the left-most root if p has at least one root in $[\alpha, \beta]$. However, there is the possiblility of skipping roots due to roundoff errors, at which point a sign comparison between $p_{[x_j,1]}(x_i)$ and $p_{[x_{i+1},1]}(x_{i+1})$ is needed after each iteration to prevent the algorithm from going past the root [10]. If they do not have the same sign, take x_i as the root.

4.3 Other Methods of Finding Roots

There are various other methods for isolating and finding the real roots of Bernstein basis polynomials. These include recursive subdivision, Newton method variants, Hull approximations, and higher degree envelopes. These are listed expressly as further reading subjects for those interested in Bernstein root finding, and are not elaborated on for that purpose.

4.3.1 Recursive Subdivision

These techniques use a midpoint to bisect the interval within a Bézier curve and use the recursive property to subdivide the interval until a root is found or the algorithm can discern that no root exists within. These are all variations of work done by

Lane and Risenfeld [8], and all use the recursive property. Some others use linear interpolation, or the convex hull to determine the roots.

4.3.2 Newton Based Methods

There are two methods that use the Newton method to calculate the roots. The first, done by Grandine [5], uses spline functions and the derivative of the curve along with Newton's iterative method to converge on the root. The other is from a research report by Marchepoil and Chenin [2] and combines recursive subdivision and Newton's method. Be warned, the entire report is in French and no translation exists as of yet, and, unless you are well versed in the language, may take some time to understand.

4.3.3 Hull Approximation

This type of root finding includes the approximation technique discussed earlier. However, there is another method that uses a parabolic representation of the convex hull property and can be used to quickly find high degree roots. This particular variation of the convex hull property was developed by Rajan, Klinkner, and Farouki [3] and exibits cubic convergence.

4.3.4 Quadratic and Cubic Envelopes

These methods are fairly modern and utilize an envelope formed by quadratic or cubic Bézier cuves and converge on the root from both sides by shrinking the interval the envelop creates as it crosses the x-axis. These methods are useful for higher degree curves and find the roots significantly faster than their counterparts. The quadratic variation was developed first by Bartoň and Jűttler [1], and adapted later into cubic

form by Liu, Zhang, Lin, and Wang [7].

CHAPTER 5

GENERATING SURFACES OF VARIABLE ECCENTRICITY

We will begin to explain the implementation of the SAVE and sqrtSAVE classes, enabling us to draw surfaces of variable eccentricity. SAVE itself is an acronym for *surface allowing for variable eccentricity* as it can also draw any rotated polynomial, as well as adding a center variance effect, by setting $a(z)$ and $b(z)$ to the default value of 1.

5.1 Required Definitions

In order to generate the SAVE class we first need to define some variables and operators that will allow us to solve (3.11). We let the sqrtSAVE class inherit the RotSqrtPoly class, which already has $p(z)$, p_x, p_y, as well as their derivatives, and utilizes a Bernstein basis root finder.

5.1.1 Variable Assignments

To create (3.11) we need to add $a(z)$, $b(z)$ and their derivatives to RotSqrtPoly as new class members. We need to declare them as polynomials, take their derivatives, and have a means to set and utilize them when building the object. However we also need to input their coefficients within the SAVE class. Two operations, set_a and set_b, input coefficients and generate the polynomials a, b, and take their derivative. The remaining information is obtained from the constructors of RotSqrtPoly (p, p_x and p_y).

5.1.2 Constructors

When making a surface with eccentricity we need to construct it using a default and copy constructors along with a few operators that will be defined and implemented later in the default constructor of sqrtSAVE. We call the default constructor for the RotSqrtPoly class and set a and b to a constant function 1.0. For the copy constructor we call the corresponding constructor of RotSqrtPoly and modify it using a, b, and their derivatives in order to describe the object. We also set up some classes to be used later in the implementation; the full sqrtSAVE file can be viewed in the appendix.

5.2 Implementation

We can now account for eccentricity. We build an operator function that inputs the information for a rotated polynomial, and sets both a and b.

5.2.1 Setup

The setup code is neccessary to build an object within the world of the ray tracer. First, we find the degree of the ray-surface intersection equation (3.11) by the formula below it, then resize the memory of the polynomial to account for it. Next, we perform the setup operation to generate a bounding cylinder for our surface. We make a small 'pace' in z, a base x-value and y-value $(S_x(z), S_y(z))$, then examine our polynomial at each pace as follows:

$$S_x(z) = |p_x(z)| + |a(z)|\sqrt{p(z)}$$

$$S_y(z) = |p_y(z)| + |b(z)|\sqrt{p(z)}$$

The cylinder generated by the maximum radius $R = max_z\sqrt{S_x^2(z) + S_y^2(z)}$ will be used to bound our object.

5.2.2 Parametrized Equation for t

We will breifly describe how to create the make_eqn_t function,which is (3.11) in the form of (3.3). We need to parameterize p, a, b, p_x and p_y in terms of $z(t)$ and create (3.11) within the program. For this, we substitute (3.2) for x, y, and z in (3.11). For more details see the function make_eqn_t in the appendix.

5.2.3 Normal Calculation

To produce shading, we need to compute the normal of (3.11). For efficiency, we store a new polynomial,

$$q(z) = a(z)^2 b(z)^2 p(z),$$

in the memory, which simplifies the code. (Note: within the program we don't need an extra memory definition for q just a replacement: $p = p * a * b$.) The normal is then calculated by

$$\nabla F(x,y,z) = \frac{\partial F}{\partial x}\hat{x} + \frac{\partial F}{\partial y}\hat{y} + \frac{\partial F}{\partial z}\hat{z} = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right), \tag{5.1}$$

which we apply to (3.11). This generates the following equations:

$\frac{\partial F}{\partial x} = 2(x - p_x(z))b^2(z) = c_x * b(z)$, where $c_x = 2b(z)(x - p_x(z))$

$\frac{\partial F}{\partial y} = 2(y - p_y(z))a^2(z) = c_y * a(z)$, where $c_y = 2a(z)(y - p_y(z))$

$\frac{\partial F}{\partial z} = c_x(0.5c_x * \frac{db(z)}{dz}/b(z) - \frac{dp_x(z)}{dz}b(z)) + c_y(0.5c_y * \frac{da(z)}{dz}/a(z) - \frac{dp_y(z)}{dz}a(z)) - \frac{dq(z)}{dz}$

5.2.4 Using sqrtSAVE

To call upon the sqrtSAVE, you need to generate an instance of the object, then define the five vectors for p, p_x, p_y, a, and b by using the set operations. Finally you define its desired material, color, scale, translation, and rotation before using the add_object command to place your surface within the world.

CHAPTER 6

ANALYSIS OF THE SAVE AND SQRTSAVE PROGRAMS

In theory the program should be able to generate a surface of varying eccentricity by simply utilizing $a(z)$ and $b(z)$ with any rotated polynomial. However, there are a few restrictions that should be noted before drawing objects.

6.1 Polynomial Requirements

When generating objects, one must consider the roots of $p(z)$. This determines where the object crosses the axis of rotation. For example, a sphere requires roots where the sphere intersects the z axis. For a radius of one, we must ensure $p(z) = 0$ when $z = 1$ or -1. For this particular sphere the function is as follows:

Figure 6.1: Unit sphere generated by the sqrtSAVE program.

$$p(z) = -(z+1)(z-1) = -z^2 + 1$$

When written as code we have $p = [-1, 0, 1]$. If the desired object is an open one, $p(z)$ must not have a root within the clipped interval. By considering the roots

of $p(z)$, their multiplicity, and the interval being traced, we can generate a closed section anywhere on the object.

For central variance, p_x and p_y, the rules are less strict. However, for $a(z)$ and $b(z)$ there is a very strict requirement, the eccentricity functions MUST NOT have any root within the interval (see (3.8)). Otherwise, the only limitations are those on the imagination.

6.2 Objects

Now it is time to examine a few objects and how they can be varied. We will examine some simple objects, their functions, and what occurs when we vary the axis of rotation. Then, an eccentricity variance will be applied to demonstrate its effect.

6.2.1 Examples of Rotated Polynomials With No Variance

Now some basic objects will be introduced and defined. First, we start with a few closed objects.

$$p(z) = -z^4 + 1 = -(z^2 + 1)(z^2 - 1) \tag{6.1}$$

$$p(z) = z^4 - 2z^2 + 1 = (-z^2 + 1)^2 \tag{6.2}$$

$$p(z) = z^3 - z^2 - z + 1 = (z - 1)(z + 1)^2 \tag{6.3}$$

These polynomials have roots at both 1 and -1 and no roots in between. Also used are the following open objects:

$$p(z) = -\frac{1}{4}(z^3 + 4z^2 + z - 6) \tag{6.4}$$

$$p(z) = -\frac{1}{10}(z^4 + z^3 + 17z^2 - 2z - 24) \tag{6.5}$$

$$p(z) = z^3 + z^2 + z + 1.5 \tag{6.6}$$

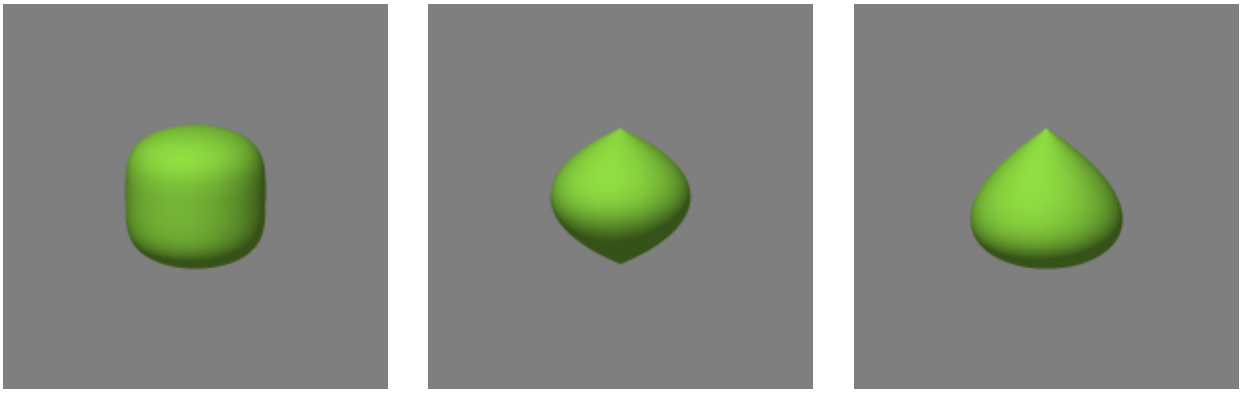

Figure 6.2: Closed objects with no variance. From left to right: (a) (6.1), (b) (6.2), (c) (6.3)

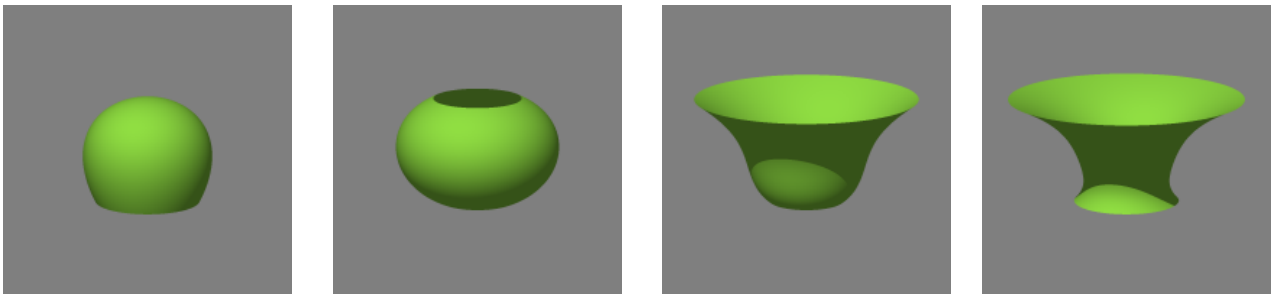

Figure 6.3: Objects with no variance from left to right: (a) (6.4), (b) (6.5), (c) (6.6), (d) (6.7)

$$p(z) = z^4 + z^3 + z^2 + z + 1 \tag{6.7}$$

Upon examining the horizontal cross sections of these objects, one should note they are all circles of various sizes.

Next, we demonstrate the effect of central variance and how it alters the object.

6.2.2 Central Variance

By using the parameters p_x and p_y we create a variation in the axis of rotation. Let's compare (6.2) to its counterpart with linear variance. If we vary p_x alone we can generate the same effect as varying p_y alone by simply rotating the object.

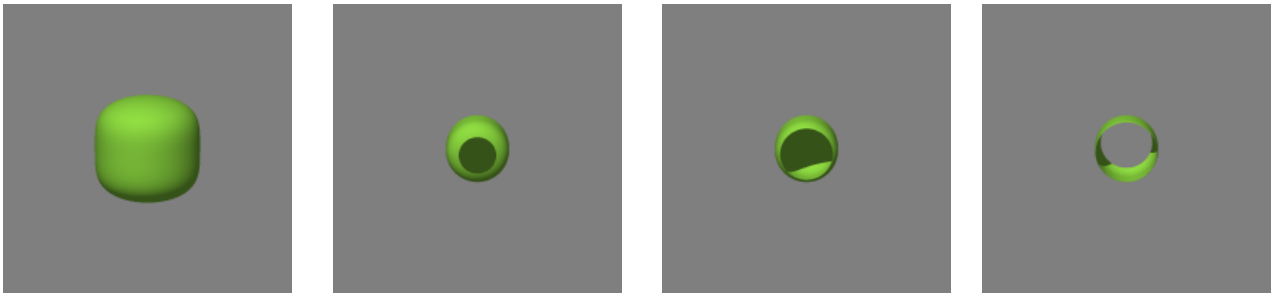

Figure 6.4: Object (6.1) (a) in base form, then clipped to demonstrate the cross sections at (b) [-1.0, 0.9], (c) [-1.0, 0.75] and (d) [-0.75, 0.5]

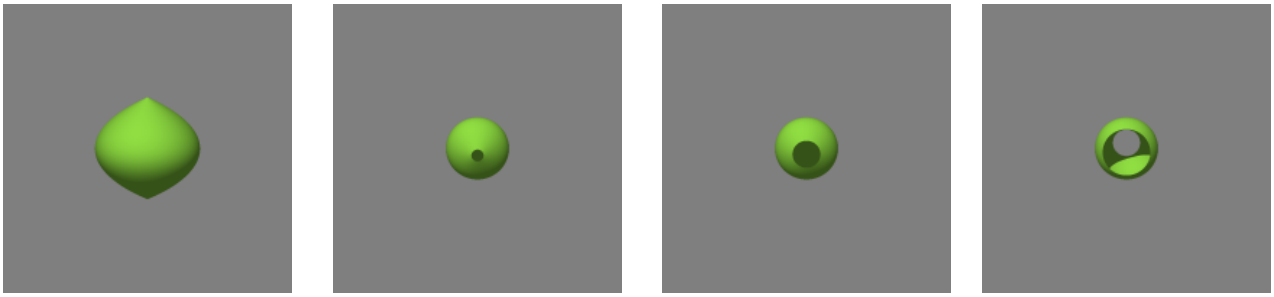

Figure 6.5: Object (6.2) (a) clipped to demonstrate cross section at (b) [-1.0, 0.9], (c) [-1.0, 0.75] and (d) [-0.75, 0.5]

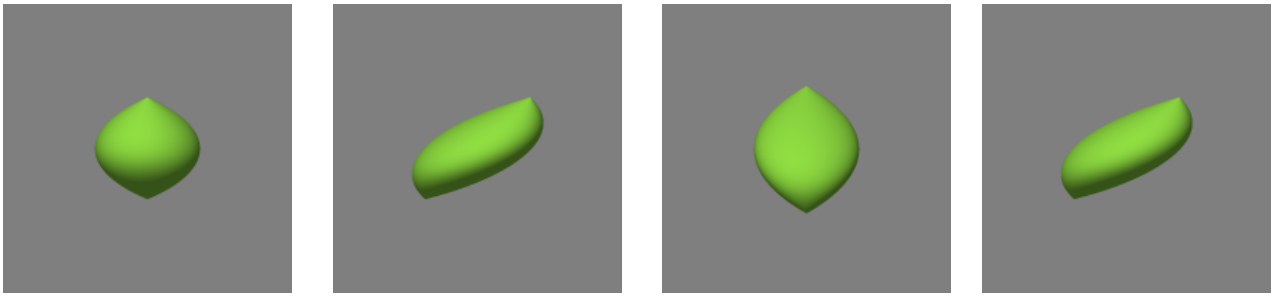

Figure 6.6: Object (6.2) (a) in base form, then (b) varied by $p_x = z$, (c) by $p_y = z$ and (d) rotating the previous image by -90 degrees.

There are some noteworthy traits of this variance. Let's use (6.1) and (6.3) to examine what happens when we vary the center in multple ways.

But this does not change the horizontal cross sections, they are still always circles.

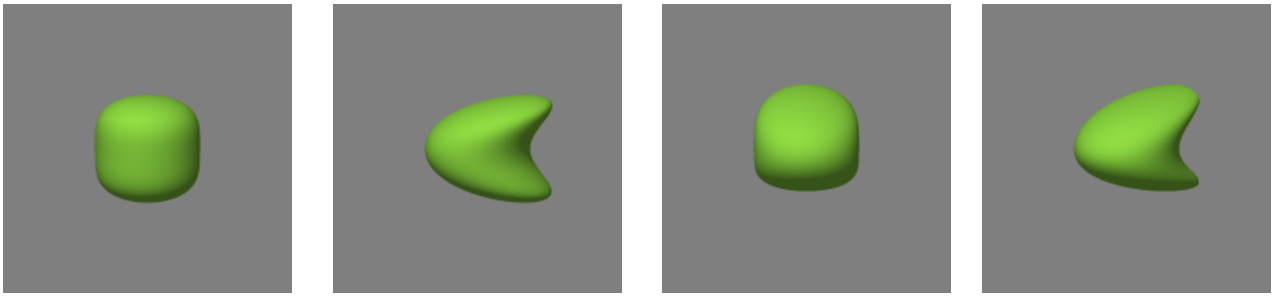

Figure 6.7: Object (6.1) (a) in base form, (b) varied by $p_x(z) = z^2$, (c) varied by $p_y(z) = z^2$, and (d) by $p_x(z) = p_y(z) = z^2$.

We could scale $p(z)$ in the x or y direction by a constant but the horizontal cross sections would all be ellipses with the same $\frac{b}{a}$ ratio.

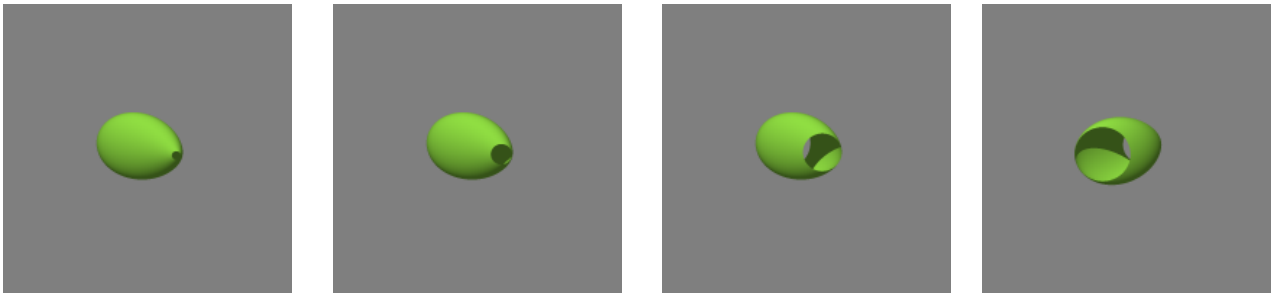

Figure 6.8: Object (6.3) with linear central variance, $p_x = z$, viewed from the top clipped at (a) [-1.0, 0.9], (b) [-1.0, 0.75] and (c) [-0.75, 0.5] with the bottom view (d).

In order to vary the eccentricity we need to use the new parameters $a(z)$ and $b(z)$.

6.2.3 Varying Eccentricity

To demonstrate the effects of a and b we will use the (6.6) and (6.7). To start we focus on the (6.6) and vary $a(z)$ linearly by $0.5z + 1$.

Now we vary $b(z)$ by the quadratic function $0.5z^2 + 0.5z + 1$.

The eccentricity is noticably different at the top and bottom of the object. To

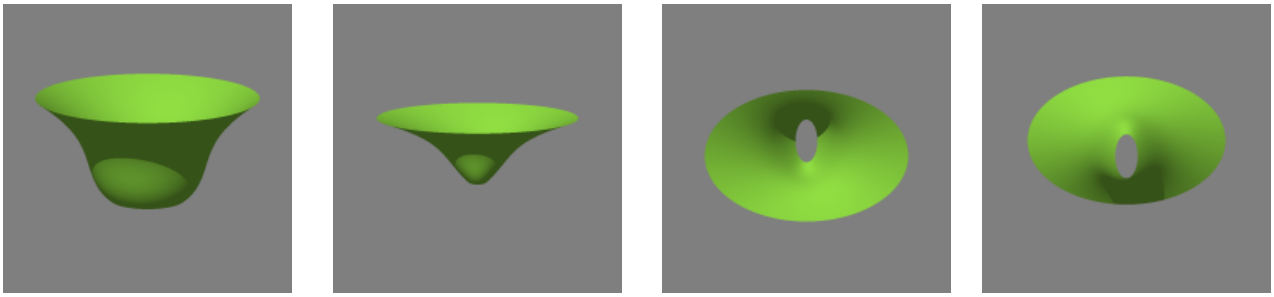

Figure 6.9: (a) Object (6.6), then adding a linear $a(z)$ as viewed from different angles. (b) Front, (c) Top, (d) Bottom

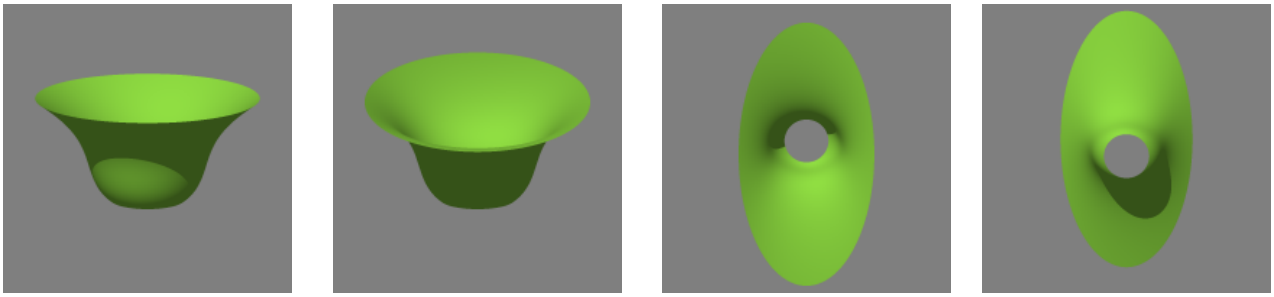

Figure 6.10: (a) Object (6.6), then adding quadratic $b(z)$ as viewed from different angles. (b) Front, (c) Top, (d) Bottom

achieve a more drastic effect, we will use (6.7) and choose two simple functions: $a(z) = 0.5z + 1$, an increasing function, and $b(z) = -0.5z + 1$, a decreasing function.

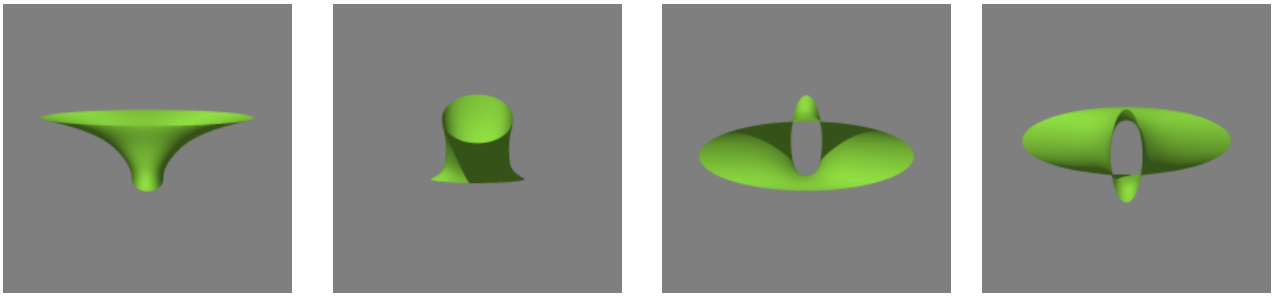

Figure 6.11: Object (6.7) varied by $a(z) = 0.5z+1$ and $b(z) = -0.5z+1$ as seen from different angles. (a) Front, (b) Side, (c) Top, (d) Bottom

We will now use (6.4) to show the effects of central and eccentrical variance on

an object by taking the cross section at various points along $[-1, 1]$. We will use the following parameters:

$$p_x = z$$

$$p_y = -z^2 + 1.0$$

$$a(z) = 0.5z + 1.0$$

$$b(z) = 0.25(z-2)^2 = 0.25z^2 - 1.0z + 1.0$$

First we examine the object using $a(z)$ and $b(z)$ alone.

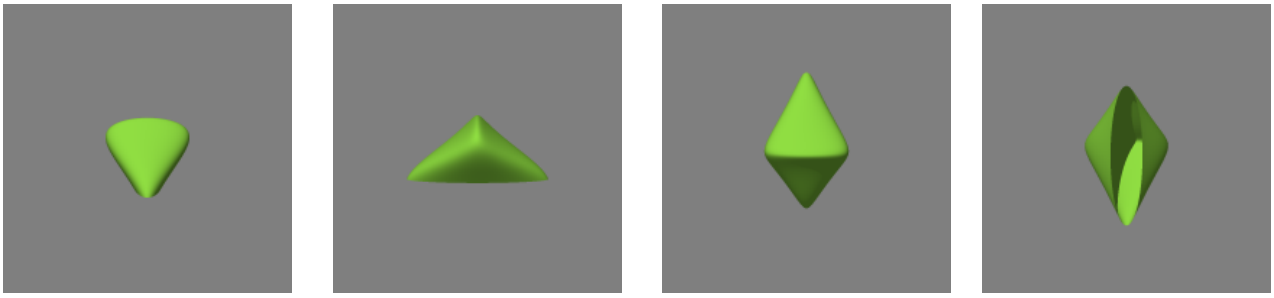

Figure 6.12: Object (6.4) varying by $a(z) = 0.5z + 1$ and $b(z) = -0.25z^2 - z + 1$ as viewed from different angles. (a) Front (b) Side (c) Top (d) Bottom

Now, we add the central variance parameters to generate the following object.

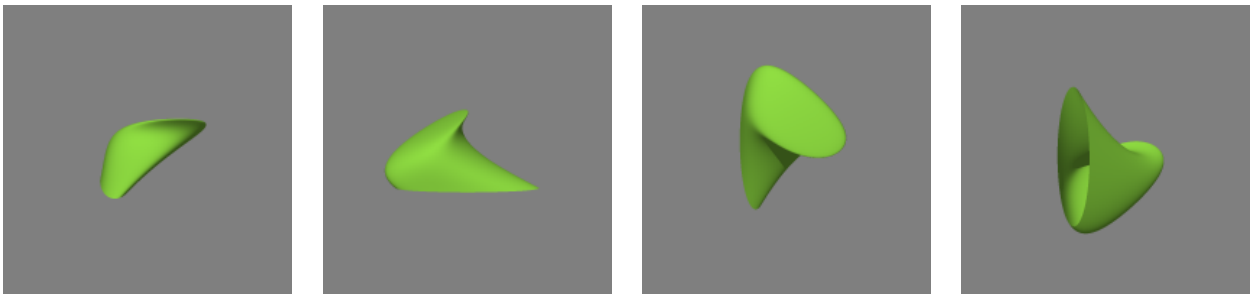

Figure 6.13: Object (6.4) with added central variance $p_x = z$ and $p_y = -z^2 + 1.0$, as seen from various angles. (a) Front, (b) Side, (c) Top, (d) Bottom

We examine the top and bottom of the object with a small clipping applied and compare it to the top and bottom views to get a general understanding of its shape.

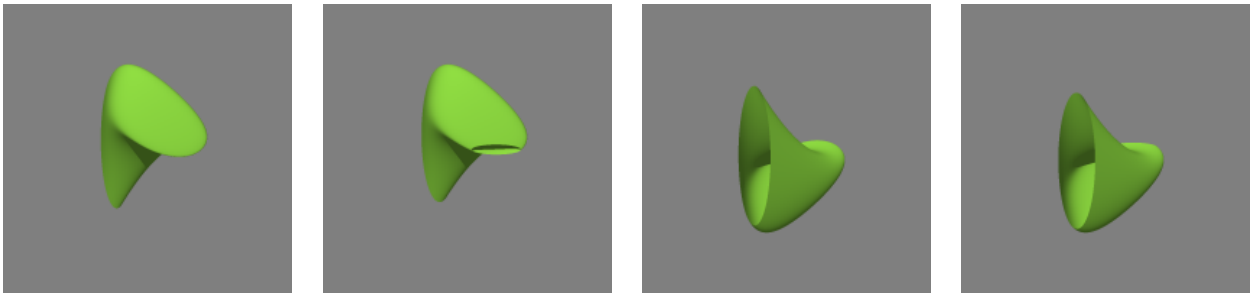

Figure 6.14: (6.4), (a) seen from the top and clipped at (b) 0.9 as well as (c) the bottom with a clipping at (d) -0.9.

To fully demonstrate the effects of $a(z)$ and $b(z)$ we gradually cut away from the top and bottom.

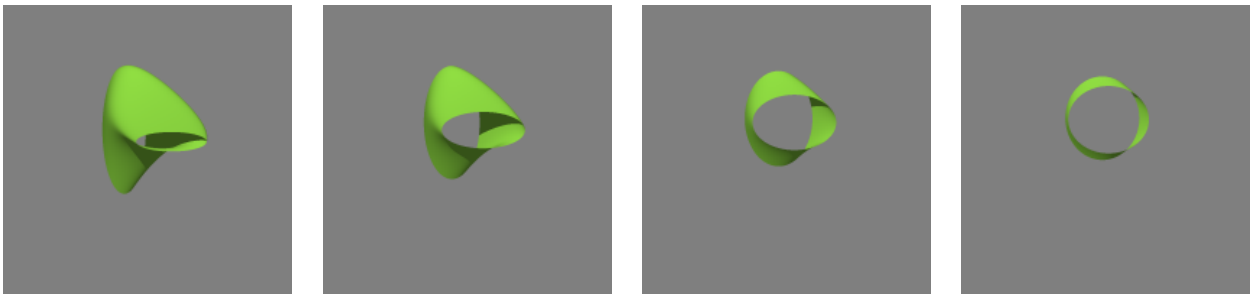

Figure 6.15: (6.4) as seen from the top; clipped at (a) 0.75, (b) 0.5, (c) 0.25, (d) 0.1

6.3 Bernstein Conversion

The ray tracer also a Bernstein to power basis conversion, enabling us to account for Bernstein coefficients. This allows the user to choose between Power and Bernstein Basis. We will compare (6.5) in power basis to its Bernstein basis counterpart whose coefficients are $[0.5, 2.35, 2.8667, 2.65, 0.7]$.

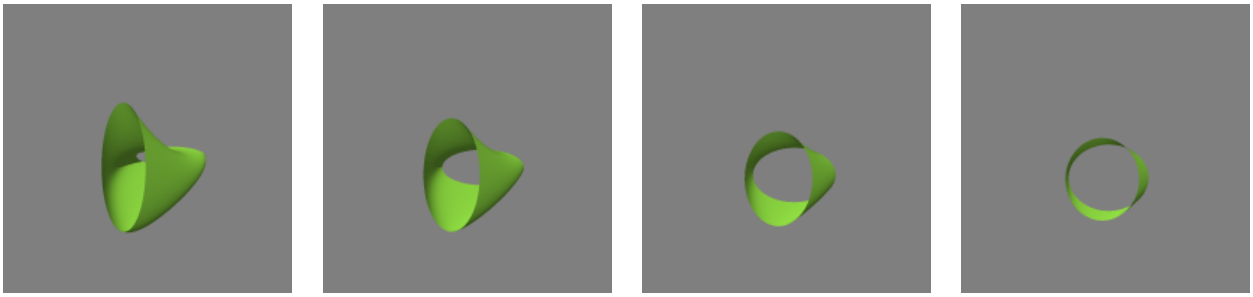

Figure 6.16: (6.4) as seen from the bottom; clipped at (a) -0.75,$(b) - 0.5$, (c) -0.25, (d) -0.1

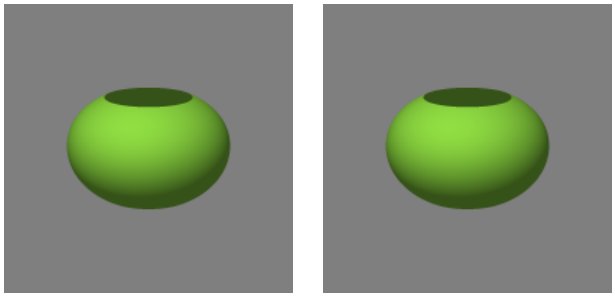

Figure 6.17: Comparison of (6.5) with (a) p_x to (b) its Bernstein coefficient equivalent.

The conversion can also change central variance and eccentrical variance to account for Bernstein Basis coefficients. Let's demonstrate this using $p_x(z) = z$, whose coefficients in Bernstein Basis are $[-1, 1]$.

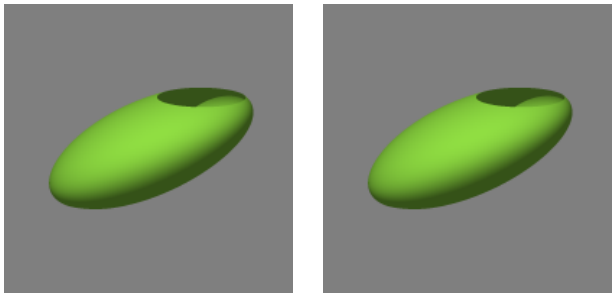

Figure 6.18: Comparison of (6.5) (a) to its Bernstein coefficient equivalent. (b)

To compare eccentricity we will convert $a(z) = 0.5z + 1$ and $b(z) = -0.5z + 1$ to their Bernstein equivalents $[0.5, 1.5]$ and $[1.5, 0.5]$ respectively.

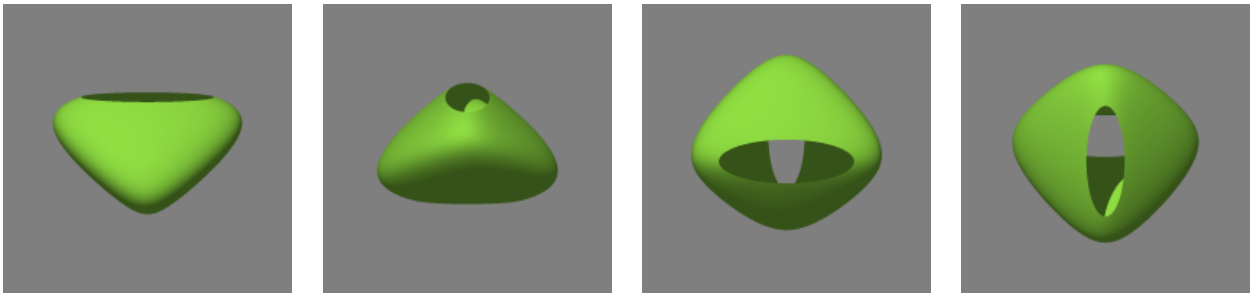

Figure 6.19: (6.5) with $a(z)$ and $b(z)$ as seen from the (a) front, (b) side, (c) top, and (d) bottom

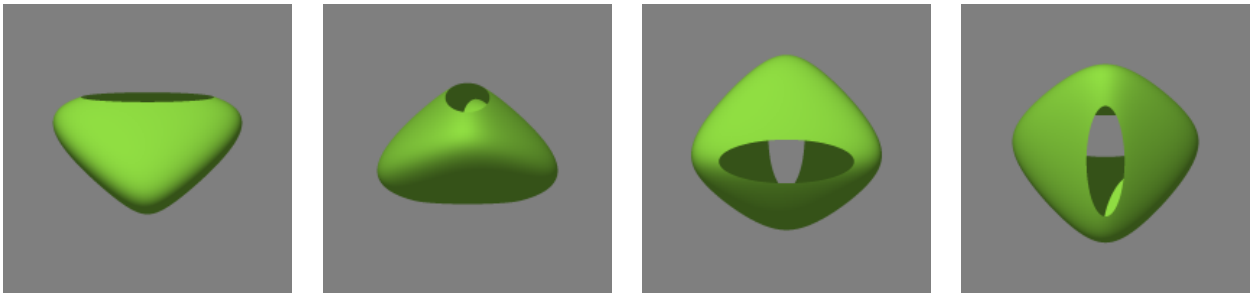

Figure 6.20: (6.5) using Bernstein coefficients for $a(z)$ and $b(z)$ as seen from the (a) front, (b) side, (c) top, and (d) bottom.

As expected the conversion works and we achieve identical images for power and Bernstein basis.

CHAPTER 7

CONCLUSION

We have examined the basic workings of a ray tracer, the mathematics behind generating various polynomial surfaces, the Bernstein basis root finding algorithm, and how to create the SAVE and sqrtSAVE program. We examined the different objects that are generated by the program and applied conversions for Bernstein basis coefficients. Overall, the program does what is expected. A variation in eccentricity has been displayed and thus opens up a new group of objects that can be now generated within a ray tracer.

REFERENCES

[1] M. Bartoň, B. Jűttler. "Computing Roots of Polynomials by Quadratic Clipping" (2007), *Computer Aided Geometric Design*. 24:3:125-141.

[2] P. Chenin, D. Marchepoil. Algorithmes de recherche de zéros dúne fonction de Bézier. (1990) Laboratoire de Modélisation et Calcul RR 834-I & M-, Institut National Polytechnique de Grenoble, France. Université Joseph Fourier Grenoble 1, Centre National de la Recherche Scientifique, Ecole Normale Supérieure de Lyon.

[3] R. T. Farouki, S. R. Klinkner, V. T. Rajan "Root isolation and root approximation for polynomials in Bernstein form" (1988) IBM Research Report RC14224, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, N.Y. 10598.

[4] R.T. Farouki, V.T. Rajan. "On the Numerical Condition of Polynomials in Bernstein Form" (1987), *Computer Aided Geometric Design*. 4:191-216.

[5] T. A. Grandine. "Computinig Zeros of Spline Functions" (1989), *Computer Aided Geometric Design*. 6: 129-136.

[6] Y. Hu, J. Zhu. "High-Degree Polynomial Models for CT Simulation" (2012), *Journal of X-Ray Science and Technology*. 20:4: 447-457.

[7] B. Lin, L. Liu, G. Wang, L. Zhang. "Fast Approach for Computing Roots of Polynomials Using Cubic Clipping" (2009), *Computer Aided Geometric Design*. 26:5:547-559.

[8] J. M. Lane, R. F. Riesenfeld. "Bounds on a polynomial" (1981), *BIT*. 21:112-117.

[9] T. Nishita, T. W. Sederberg. "Curve Intersection Using Bézier Clipping" (1990), *Computer Aided Geometric Design*. 22:9:538-549.

[10] T. W. Sederberg. "Computer Aided Geometric Design" (2012), *All Faculty Publications*. Paper 1. *http://scholarsarchive.byu.edu/facpub/1*

[11] M. R. Spencer. "Polynomial Real Root Finding in Bernstein Basis" (1994), Dissertation Preprint, *http://students.cs.byu.edu/ tom/papers/mel.pdf*

[12] K. G. Suffern. (2007) *Ray Tracing From the Ground Up*. Wellesley, MA: A.K. Peters Ltd.

Appendix A

APPENDIX

The appendix will cover the entire code involved with the sqrtSAVE program as well as Bernstein root finding and basis conversion.

A.1 sqrtSAVE

The header file is as follows:

sqrtSAVE.h

```
//Surface Allowing for Variable Eccentricity
#ifndef __sqrtSAVECC__
#define __sqrtSAVECC__
#include "RotSqrtPoly_Bern.h"

class sqrtSAVE:public RotSqrtPoly{

public:

//Default Constructor default a and b to constant 1
sqrtSAVE() : RotSqrtPoly(){
IsEccBernstein = false;
double coeff[] = {1.0};
a = Polynomial(coeff, 1);
b = a;}

//Copy Constructor
```

```
sqrtSAVE(const sqrtSAVE& obj):
RotSqrtPoly(obj), a(obj.a), ader(obj.ader), b(obj.b), bder(obj.bder){}

virtual sqrtSAVE* clone(void) const
{ return (new sqrtSAVE(*this)); }
//assignment Operator
virtual sqrtSAVE& operator= (const sqrtSAVE& rhs);

//changes a and b from power to Bernstein basis
void BernsteinEcc()
{IsEccBernstein = true;}

//set a and b
virtual void set_a(double* u, const int n_coeff)
{
//checks to see if a is Bernstein and converts it
if(IsEccBernstein){
Bernstein t = Bernstein(u, n_coeff);
t.set_A_B(-1,1);
a = Polynomial(t.power_basis());}
else {
a = Polynomial(u,n_coeff);}
ader = a.derivative();
}
virtual void set_b(double* u, const int n_coeff)
{
```

```
//Checks for b and changes it to Bernstein if neccessary
if(IsEccBernstein){
Bernstein t = Bernstein(u, n_coeff);
t.set_A_B(-1,1);
b = Polynomial(t.power_basis());}
else {b = Polynomial(u, n_coeff);}
//calculates derivative for normal
bder = b.derivative();
}
//make eqn
virtual void make_eqn_t(const Ray& ray) const;
//setup class
virtual void setup(void);
//calculates normal
virtual Normal find_normal(double x, double y, double z) const;

bool IsEccBernstein;
Polynomial a,b, ader, bder;
//define a and b as polynomials everything else in base classes



};
#endif
```

Next we have the .cpp file.

sqrtSAVE.cpp

```
#include "sqrtSAVE.h"
```

```
#include "BoundingCyl.h"

/* Cunstructors (Default Copy)
   Operator
*/

//setup class
void sqrtSAVE::setup(void)
{
//generates right side of equation and takes derivative
p = p*square(a)*square(b);
pder = p.derivative();
//checks and sets the degrees of the polynomials involved
int deg = p.degree(), degx=px.degree(), degy=py.degree();
degx=max(1, degx); degy=max(1, degy);
degx+=b.degree(); degy+=a.degree();
deg = max(max(degx, degy), deg);
maxn = deg+1;
//sets the degree to the max degree of our function
p.resize(maxn);
//set maxn and bounding cylinder
PolySurf::setup();
//set up bounding cylinder
double z, pace = 1/256.0;
double Sx, maxSx, Sy, maxSy = -1.0;
//checks for max values for bounding surface
```

```
for(z=-1.0; z<=1.0; z+=pace)
{
Sx = abs(px.value(z)) + abs(p.value(z)/b.value(z));
if(maxSx<Sx) maxSx=Sx;
}
for(z=-1.0; z<=1.0; z+=pace)
{
Sy = abs(py.value(z)) + abs(p.value(z)/a.value(z));
if(maxSy<Sy) maxSy=Sy;
}
//multiplies to increase maximums
maxSx *= 1.2;
maxSy *= 1.2;
//make bounding cylinder
Casing_I* cs1_ptr = new Casing_I;
BoundingCyl *bcyl_ptr = new BoundingCyl;
bcyl_ptr->scale(maxSx, maxSy);
cs1_ptr->bs_ptr = bcyl_ptr;
bound_ptr = cs1_ptr;

}

//generates the operator from RotPoly and adds a and b

sqrtSAVE& sqrtSAVE::operator = (const sqrtSAVE& rhs)
{
```

```
if (this == &rhs) return (*this);
RotSqrtPoly::operator= (rhs);
a=rhs.a, b=rhs.b;
return (*this);
}



//parameterizes the equation in terms of t
void sqrtSAVE::make_eqn_t(const Ray& ray) const
{
double c[]={ray.d.x, ray.o.x+ t0*ray.d.x},
d[]={ray.d.y, ray.o.y+ t0*ray.d.y},
e[]={ray.d.z, ray.o.z+ t0*ray.d.z};
Polynomial xmPx_t(c,2,maxn), ymPy_t(d,2,maxn), z_t(e,2,2);
Polynomial pofz_t(p.of(z_t)); // p(z(t))
Polynomial aofz_t(a.of(z_t));
Polynomial bofz_t(b.of(z_t));
xmPx_t -= px.of(z_t);//bofz; // x(t) - px(z(t))
xmPx_t = xmPx_t*bofz_t;
ymPy_t -= py.of(z_t);//aofz
ymPy_t = ymPy_t*aofz_t;
// y(t) - py(z(t))
eqn_t  = square(xmPx_t); // squares px and py
eqn_t += square(ymPy_t);
eqn_t -= pofz_t;
```

```
}

//finds the normal
 Normal sqrtSAVE::find_normal(double x, double y, double z) const{
 Normal normal;

 //saves values used multiple times when calculating the normal
 double cx = (x - px.value(z));
 double cy = (y - py.value(z));
 double az = a.value(z); double bz = b.value(z);
 //calculates x and y values for the normal
 normal.x = 2*cx*bz*bz;
 normal.y = 2*cy*az*az;
 //calculates the z value for the normal
 normal.z = 2*cx*bz*(cx*bder.value(z) - pxder.value(z)*bz);
 normal.z+= 2*cy*az*(cy*ader.value(z) - pyder.value(z)*az);
 normal.z-= pder.value(z);
 //normalized in PolySurf
 return normal;



}
```

A.2 SAVE

A header file, SAVE.h is used to square the value of $p(z)$.

```
#ifndef __SAVECC__
```

```
#define __SAVECC__

#include "sqrtSAVE.h"

class SAVE: public sqrtSAVE {

public:

// Default constructor
SAVE() : sqrtSAVE() {}

// Copy constructor
SAVE(const SAVE& obj) : sqrtSAVE(obj) {}

// Virtual copy constructor
virtual SAVE* clone(void) const
{ return (new SAVE(*this)); }

// This will allow input and square p(z)
virtual void set_p(double* a, const int n_coeff)
{ p = square(Polynomial(a, n_coeff));
pder = p.derivative();
}

};
```

```
#endif __SAVE__
```