11-16-2004

Perceptual evaluation of point-based object representation

Paul Schwarz
Trinity University

A Perceptual Evaluation of Point-Based Object Representation

Paul Schwarz

A departmental honors thesis submitted to the Department of Computer Science at Trinity University in partial fulfillment of the requirements for Graduation.

April 21, 2004

_____	_____	_____
Thesis Advisor	Committee Member	Department Chair

_____

Associate Vice President for

Academic Affairs

A Perceptual Evaluation of Point-Based Object Representation

Paul Schwarz

Abstract

Researchers have proposed several methods of representing three-dimensional objects in computer memory along with different methods of displaying these objects on two-dimensional computer displays. Using a polygon-based representation that linearly interpolates between a set of sample points is by far the most popular way of representing objects today, mostly because of its ability to display an object on a computer screen at a fast and consistent rate. However, as graphics hardware increases in speed, polygon-based models prove to be increasingly inefficient. Point-based representations have been proposed that consist of densely sampling some surface and using solely the samples to portray the object on a computer display. This research proposes a similar approach to point-based representation while making use of current graphics hardware to compare the benefits and drawbacks of using point-based representations over polygon-based representations in interactive environments. An experiment has been conducted with human subjects to gather perceptual data about each representation method. The results from this experiment are presented and analyzed.

A Perceptual Evaluation of Point-Based Object Representation

Paul Schwarz

Acknowledgements

I would like to thank my thesis advisor, Dr. John Howland, for his constant support and advice throughout this project. I greatly appreciate his guidance.

Dr. Glenn Meyer provided instrumental insight into the experiment design and usability of the experiment, and I thank him for lending me his expertise.

I would also like to thank Dr. Gerald Pitts for his encouragement and motivation.

Thanks goes to the Trinity University Computer Science department. This work would not be possible without the dependable facilities they provide.

I would like to sincerely thank all my friends and family who have inspired me and struggled with me. I certainly could not have finished this work without them.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Object Representation and Its Impacts on Visual Fidelity

Interactive computer graphics deals with an essential tradeoff between the detail of computer-generated images and the speed at which these images can be generated. The more detailed a scene becomes, the more information must be processed into an image, and therefore display speed decreases as scene detail increases. Object detail and rendering speed are two aspects that contribute to the overall visual fidelity of the scene. Visual fidelity is a qualitative metric of how closely the synthesized scene mimics its real-world counterpart. Rendering is the proper name for the procedure that translates object information into images. The purpose of much computer graphics research is to find a way to maximize both of these parameters to yield the highest visual fidelity possible. The manner in which one decides to represent objects in computer memory impacts the method of rendering that one can use, and thus the speed of the rendering [11].

A major problem in computer graphics is finding a way to represent a real-world object in computer memory, to yield the most effective method of converting this representation into an image. There have been three major different types of approaches to object representation: volume-based representations, manifold-based representations,

and point-based representations [2]. In this paper, I will present the point-based representations as offering a higher level of visual fidelity than that of the traditionally used manifold-based representations. I will use results gathered from perceptual experiments to support my hypothesis.

1.2 Volume-Based Object Representations

Volume-based representations divide the three-dimensional object space into a large set of cubes, which make up the basic element of this representation. These elements are referred to as voxels [7]. A voxel contains information that conveys some characteristic of the object (e.g. density, heat, etc.). Often, voxels simply represent whether or not the object exists in that location [5] [7]. A voxel is essentially a sampled point of the space around the object. Because the basic primitives of volume-based representations are voxels, the sampling rate must be high enough to sufficiently convey the details of the object. The primary advantage of volume-based representations are that they convey more than just the surface of the object, like the manifold and point based representations. Making volume-based representations more appropriate for objects that are characterized by their volumetric qualities, such as smoke and liquids [5] [13]. The rendering algorithms associated with voxels however must be very complex to account for the representation of the volumetric data projected onto a two-dimensional plane [7]. This makes volume-based representations slow and unsuitable for interactive applications. The resulting visual fidelity of a scene containing objects with volume-based

representations will be low because the scenes will not be able to render fast enough. In practice, for more efficient rendering, volume-based representations will often be converted into polygon representations discussed later.

1.3 Manifold Based Object Representations

Manifold-based representations are characterized by the fact that they use interpolation methods to create an approximation of the surface between a relativly sparse set of sample points [2]. Different types of interpolation methods account for different types of manifold-based surfaces. The most popular manifold-based surface is the polygon-based surface, which linearly interpolates between sample points. That is, a line is drawn between each adjacent sample point to form polygons. This representation is popular because of its simplicity as compared to many of the other manifold surfaces. Most other manifold-based representations use curves to interpolate between sample points [1] [14]. Splines [10] and Non-Uniform Rational B-Splines [18] fall under this category. The use of curves as a rendering primitive greatly increases the complexity of the rendering algorithm. Therefore, these complex curve-based representations are usually estimated by a polygon-based representation prior to rendering. Though manifold-based representations do not require the storage of a high volume of sample points, they do require that some sort of connectivity or topological information be stored and processed for use in rendering [9].

Polygon-based representations have been the standard in computer graphics [4]. Computer hardware has been optimized to support the efficient rendering of polygons. Among the optimizations include a pipeline of computation that speeds up the projection of the sample points onto a two-dimensional image plane, and the filling in of polygons between these sample points [16]. Most applications that use three-dimensional representations of objects use a polygonal representation. As stated before, this popularity is accounted to their low complexity, and therefore speed of rendering.

The enhanced speed of polygon-based rendering boosts the visual fidelity produced by using polygon-based representations. The difficulties with polygon-based representations lie in that they must sample an object more densely where the surface of the object curves. The rendering speed of a polygon-based object is inversely proportional to how well its curves are represented. Often curves in objects cannot be adequately sampled, resulting in the model appearing blocky. Many techniques have been proposed to resolve the jagged aliasing of polygon representation, such as interpolation of the shading across the surface to give it the appearance of smoothness [11]. Polygons remain very useful for representing very geometric and jagged surfaces, but inefficient at the display of curved more organic objects. To summarize, polygons sacrifice some object detail for rendering speed, and can therefore be considered as a sort of compressed object representation that trades off a lot of the details of the real-world object for enhanced rendering speed.

The increasing speeds of computer hardware have allowed for the appearance of more highly sampled objects in interactive environments. The effect is that more complex objects may be better represented by polygons, but as more polygons are added to an object, the smaller the average screen size of a polygon on any given frame. Polygons are not inherently output-sensitive. Meaning that the amount of computation required to render a polygon does not take into account how large the polygon will appear on the screen. Consider when hardware speeds increase further, and rendering hardware will be able to accommodate so many polygons such that on average they occupy less than a pixel on the screen. Now, even though the polygons appear as points they require the extra overhead associated with rasterizing polygons [16]. Several techniques have proposed a solution to this. Level-Of-Detail algorithms pre-compute different representations for an object at different sampling densities and determine which one is appropriate to display during render-time, thus reducing the amount of time spent on polygons that will only appear very small in the rendered image [19]. Some adaptations on this idea will compute level-of-detail representations during render-time, thus providing a more adaptable method of increasing render speed [8]. Generally, Level-of-Detail algorithms suffer from not being able to smoothly transition from one representation to another, resulting in "popping" artifacts [19].

1.4 Point-Based Object Representations

Point-based representations of real-world objects consist of a very dense set of sample points along the object's surface. They are much like volume-based representations, except that the samples exist only along the surface of the object. No attempt is made to directly represent the volume of the object. These surface sample points, or surfels, are then used as the sole display primitives of the object; therefore the sampling density of the surface must be very high [17]. Point-based representations are to polygon representations what rasterized shapes are to vector shapes. Point-based representations attempt to sample an object at a high enough density so that the object has the illusion of a continuous surface [9]. An advantage of the point-based representation is that points are very simple to render, allowing for several points to be rendered at the same rate of a single polygon [16]. Another advantage to point-based representations are that they are output sensitive—not all of the object primitives must be rendered all the time in order to convey the shape of the object [15]. This gives objects represented by points the opportunity to be faster than polygon-based objects.

Currently the study of points as a surface primitive is fairly novel, but is becoming more commonplace. Researchers typically arrange a dense series of points representing an object into a hierarchical data structure such as an oct-tree [2] [15], where all the leaf nodes of the tree are the original sampled points, and the parent nodes contain points that encompass all the children nodes. During rendering this tree-structure is traversed, and if the projected point size of a point falls below a threshold, the point's

parent node can be rendered in its place and in the place of all of its siblings, reducing the amount of the points that must be displayed, and the amount of the tree that must be traversed [2] [15]. Therefore providing a solution to the output sensitive problem.

Though point-based representations handle the output-sensitive problem they do not currently interpolate between the points. Therefore, if one views an object from a very small distance, the object can appear very blocky or pixelated. Because point-based representations do not interpolate between samples, the points grow in size if one gets closer to an object than the sampling density accommodated for. This is analogous to the effect one gets from upsizing a bitmap image. Pixelation artifacts become apparent. Therefore in order for many objects to enable being seen from up-close, they must be sampled at very high densities, and consequently, contain a large number of point primitives to be handled by the rendering algorithms.

Overall, point-based representations allow for an output-sensitive display of a surface whose complexity is limited by the resolution of the display screen and the resolution at which the object was sampled. A major drawback of point representations is that they require a lot more rendering primitives to be handled by the traditional rendering pipeline. Though the cost of rasterizing a point is much less than a polygon, many more of them must be computed.

1.5 Visual Fidelity Definition and Past Work

Visual fidelity is the quality at which an image or environment, in this case, is perceived [22]. This kind of metric has been used in the past with applications such as image compression [3], movie compression [6], and the simplification of different objects represented in the computer [19]. Watson et al. review different metrics to measure the visual fidelity of models that have been simplified [22]. They use three quantitative metrics: rating, preference and mean naming time. Rating is a test where the subject ranks certain objects. Preference is where the subject selects a particular object from a set of objects. Mean naming time is the measure of how quickly a user can identify and name a presented object. The amount of time it takes to identify a presented object indexes a number of factors that contribute to object recognition, thus it is a particularly good signifier of visual fidelity [23]. In this paper, I will use the concept of visual fidelity to compare scenes that use a polygon-based representation of objects, to scenes that use point-based representations. To measure visual fidelity I set up tests that enable the user to either identify objects, or rank objects, therefore I can rate the fidelity of the object-representation methods based on the results I obtain.

1.6 Past Work in Point-Based Representations

Levoy and Whitted first proposed a method that used points primitives to render any object [9]. They are interested in using points as a meta-primitive, believing that the reduced rendering overhead of the point primitive is advantageous, and that objects

represented using all different kinds of methods should first be converted to a point representation and then rendered. Levoy and Whitted demonstrate this using a polygon-based surface. They convert the surface to points by placing a point at each vertex in the polygon mesh, and interpolating in-between with more points. Then they proposed a complex rendering method that calculates the contribution of each point to the image as a Gaussian curve centered at the point. Though this produces images that have correctly anti-aliased edges, it results in "fuzzy points" that must be properly blended and occluded.

Levoy and Whitted's "fuzzy point" is their solution to a problem in point-based rendering called splatting [15]. Points are mathematically zero-sized locations in three-dimensional space. Splatting deals with how to represent these points as part of a surface in a final image. Levoy and Whitted demonstrate the popular approach of using Gaussian weighted filter functions to determine how much each point contributes to the area around it. This method is slower than other methods, but it yields a quality image [13]. Other methods include those used by the OpenGL programming interface [13] [16]. OpenGL allows for two different kinds of splat types: a square, or a fuzzy square much like the gaussian weighted points used by Levoy and Whitted [16]. Circles and ellipsis have also been used as splat types [13]. Ellipsis are used by drawing an ellipses around the specified point, and using the point's normal vector to determine the ellipsis's orientation and size [13].

Szeliski and Tonneson approach point-based representations from the background of volume-based representations [17]. They started from the consideration of a particle system, where the particles move according to a set of predefined rules. Szeliski and Tonneson use these particles to arrange themselves into objects, to where the particles have attraction and local repulsion forces acting on them to evenly space them across a surface. They then include an orientation with the particles so that the particles include a vector that must be facing away from the surface. They call these oriented particles "surfels," because they can exist only on the surface of an object, and not the entire volume as the volume-based representations use [17].

Rusinkiewicz and Levoy produced a program that effectively renders highly sampled objects at interactive rates using a point-based representation, called QSplat [15]. They start from objects that had been scanned using 3D scanning technology. This technology produced a large collection of sample points from the surface of the scanned object. Rusinkiewicz and Levoy then arrange the sample points into a bounding sphere hierarchy. In this bounding sphere hierarchy the original sample points make up the leaf level of the hierarchy and parent nodes are created as spheres that contain their child nodes, thus solving the output-sensitive problem discussed earlier. They also use the hierarchical data structure to perform culling at coarser levels of detail. In this set-up if a parent node is found that can be completely removed from view, than it's children can also be removed and do not need to be traversed. This results in rendering speeds of around 0.1 seconds for 200 to 300 thousand points.

Botsch et al. uses an oct-tree structure to arrange points [2]. Botsch employs a variety of simplification methods to reduce the intensity of the calculations that must be performed during rendering at each node to boost the speed of point-based rendering. Botsch's pure software renderer performed at 14 million points per second, roughly seven times faster than QSplat [2] [15].

Other notable research in point-based rendering includes research by Wand et al. who uses points to sample a scene represented in polygons to display at a very fast rate [20]. Wand et al. samples the given polygon-based scene based on the location of the viewpoint, so that sample points would only be created if that area of the scene affected the output image. These points are then splatted onto the final image to render scenes with a very high polygon-count in a efficient output-sensitive manner [20].

Chapter 2: Experiment Design

2.1 Goals

I wish to examine the differences between the visual fidelity of scenes rendered using polygon-based representations and scenes rendered using point-based representations. Furthermore, I wish to evaluate the visual fidelity that these object representation schemes produce in interactive scenes. In order to evaluate visual fidelity one must gauge how well the user is able to correctly perceive and interact with each scene. Thus an experiment program must be set up that allows the users to navigate in a scene that uses polygon-based representations and in a scene the uses point-based representations. This program must record data (e.g. as the user's position at each frame) that will lead to the generation of metrics that contribute to visual fidelity. The purpose of this is to weigh the drawbacks and the benefits of each representation method, and therefore I have had to make certain decisions as to how to implement each method.

2.2 Choosing Constants

A significant question is how much, if any, should the polygon models be simplified to portray their ability to roughly estimate a model and allow for efficient rendering? I

approach this issue experimentally, reducing the models as needed to keep the frame rates of the polygon-based scenes competitive with those of the point-based scenes. As will be discussed later, the polygon-based scenes need no simplification, and in many cases prove to be much faster than their point-based counterparts.

Another difficult issue is the fact that so much of current graphics hardware is optimized to handle polygon-based representations [4] [16]. Given this, how much, if any hardware rendering should be performed in the point-based scenes? In the interest of keeping the point-based rendering comparable to the polygon-based rendering, I decide to use as much hardware support during point-based rendering. I use the graphics hardware to perform vertex transformation and rasterization. I believed that for transformations and rasterization, hardware support would provide faster times than a pure software implementation.

A benefit to using the point-based representation comes from the fact that per-vertex shading is often an adequate way to represent textures across a surface. In polygon-based representations the vertices are often too sparse or non-uniform across the surface so that defining the color at the vertices does not typically give enough texture information, thus texture mapping is employed to give the illusion of further detail. Vertex shading works for point-based representations because the vertices are the smallest primitive viewable. Therefore, for the sake of testing the benefits and drawbacks to each representation I use only vertex shading for the point-based representation, and texture mapping for the polygon-based representations.

2.3 Test Specifications

The testing program will administer a series of tests in random order to human subjects. For each test, the program will first run a polygon-based representation of a scene, and then, based on a random number, the program will either run a point-based representation of the same scene, or run the polygon-based scene over again. This puts each subject into either a control category, if they interact with two polygon-based scenes; or a variable category, if the second scene used point-based representations. If a user is faced with the same scene twice in a row, they will likely do better the second time. This dual-scene setup is used to allow for the effect of learning on the users performance. Because I have placed users in control and variable groups I will be able to compare the amount of improvement in the variable and the control group. Thus I will use a paired-difference test to statistically analyze the results. If the control group improves more on the second test than does the variable group than the polygon-based representation was able to present a better visual fidelity. If the variable group improved more, than this supports the idea that point-based representations lead to better visual fidelity.

In order to set up the tests for this experiment one must define the series of metrics that we need to obtain. Watson et al. did a similar experiment using static images of polygon-based models at different levels of simplification [22]. They gathered three different metrics: mean naming time, rating, and preference [22]. Mean naming time is how long it takes for a subject to name a presented object. Rating is the result obtained by an experiment where the experimenter presents the subject with a series of objects,

and the subject must rank the objects based on the perceived quality. Preference is similar to rating. The subject must chose one object out of a series that represents the original object the most.

The time it takes for a subject to name an object is a useful metric in that it references many factors in object recognition [21]. Naming times are effected by factors from the low to high cognitive levels. One must recognize an object and then find the correct word to describe it. Watson and others have noted that naming times vary between man-made artifacts and natural objects [21]. The reason for this has been hypothesized as natural objects tend to have many related objects that are structurally similar, therefore needing more disambiguation. For example if one were to have to name an object that was a dog, they would subconsciously have to find a difference between that and the rest of the four legged mammals. Man-made artifacts, on the other hand, tend to have more varied shapes. Due to this effect, all the objects in the object recognition tests for this paper are natural objects, so that disambiguation will be relatively constant among all tests. Also noted by Watson is that preferences and ratings involve mostly very low levels of cognition. These tests only reference the recognition and visual perception parts of cognition and do not deal with higher level linguistic functions.

To evaluate the performance of point-based representations in relation to polygon-based representations I am mainly interested in the preference and naming time metrics. The users will express their preferred representation scheme by how well they can

perform the specified tasks using either point or polygon representations. I also include naming time metrics because they are an effective and easy to measure signifier of visual fidelity. Naming time is also especially suited for scenes that contain only the object to be identified, which is much like most of the interactive scenes to be produced by point-based research so far.

Specifically, the tests the experiment program will administer are: a far object recognition test, a near object recognition test, a high-speed navigation test, and a low-speed navigation test. The two object recognition tests are proofs of concept. They do not represent interactive scenes, but only the display of single objects at a fixed distance. These tests rate the effectiveness of both algorithms in producing a single object at an interactive rate. They mimic the programs produced by previous research in point-based representations, such as Rusinkiewicz's QSplat program [15], in order to compare the visual fidelity of the results of past research with their polygon-based counterpart.

I have chosen to present objects in these object recognition tests at distance that will bring out the advantages and shortcomings of each method. The nature of the point-based representation concludes that objects when viewed from very close up will appear pixilated or blocky. The near object recognition test is meant to test how much this affects the visual fidelity of the object at this non-optimal range. The far test puts the polygon-based scene at a disadvantage, because polygons are inherently not output-sensitive. A high density model when viewed from far away may result in a very slow

frame rate in that all polygons must still be processed even though they may appear to be less than a pixel in size on the screen.

During the Object Recognition Test the user must identify the object presented. The user is allowed to rotate the object around, but he is not allowed to change his distance from the object. Once the subject has identified the object he will hit the enter key. Then the program will present the user with a dialog box where he will enter the name of the object he just identified. The program will record the user's position at each display cycle of the program, therefore allowing the researcher to measure how long it took for the subject to name the object.

The high-speed navigation test displays a canyon that the user must navigate a ship through. This tests the visual fidelity of the canyon's representations for both methods where fluid movement is important. During this test the program will record the position of the user as well as the time delay between the last frame rendered and the number of points rendered, if applicable. The position and time variable will be used to see how well the user was able to navigate the canyon in either representation. I will use this measure of performance to signify the user's preference. This test requires for the point-based representation to render a large flat surface that the viewer most likely see from very up-close, which is not at all an optimal configuration for the point-based scheme. However, the canyon walls will require a relatively high polygon count to render effectively using the polygon-based representation, because the canyon walls are curvy and cannot be accurately represented using a few polygons.

The low-speed navigation test is much like the high-speed navigation test. The low-speed navigation test presents the user with a rocky terrain to navigate. Within this rocky landscape exists a smooth path the user must stay on. If the user strays from the path a red warning light will flash, and the user will be bumped back onto the path. This test will record the same data as the high-speed navigation test, and performance will be used in the same way to signify preference. This test measures how well the user navigates a scene at a slower pace, but requires a lot more polygons to render the intense detail of the terrain. For the polygon-based scenes the rocky terrain test is much like the canyon walls in the high-speed terrain test in that the rocky terrain requires a very high amount of polygons to represent a jagged rocky area. For the point-based scenes the rendering of the terrain models differ because the canyon model relies on large obvious detail, whereas the important information in the rocky terrain is more subtle and requires the point-representation to render at its highest detail.

This experiment was conducted on 15 undergraduate students, who reported that they use computers in an everyday capacity.

2.4 Evaluating Performance

The position results of the navigation tests require a method to extract some rating of the performance of any particular subject. Important events to consider are when the subject is running into a boundary, either a canyon wall or rocky terrain area, and when the user is going the wrong way down a path.

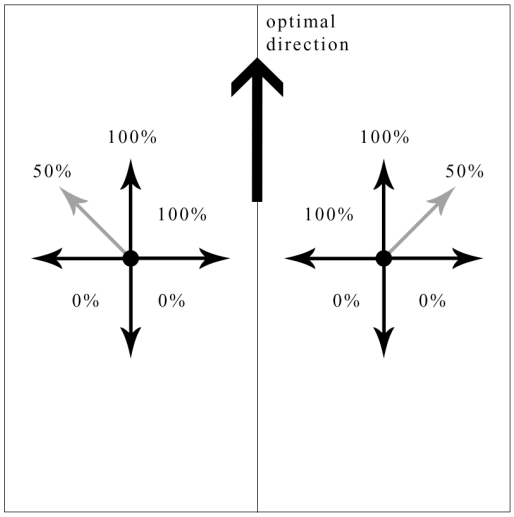

Figure 2.1: Direction grading for the navigation tests

I decided to grade the subject's direction of movement. For each sample collected while the subject was taking the test, I examine those that show a change in position. I then compare the subject's direction vector to an optimal direction for that part of the path. If the subject is moving along the path or towards the center of the path I grade that sample as a 100%. If the subject is moving away from the center of the path, I grade it between 100% and 0%, giving the user a 0% when they are moving perpendicular to the optimal direction. If the angle between the subject's direction and the optimal direction is beyond 90 degrees than the subject is going backwards down the path, and I grade that sample as 0%. If the user is beyond the bounds of the path, in the case that the collision detection has failed to keep the user within the bounds, than I also grade that sample as 0%. The total grade for each subject is the average of the grades for each sample.

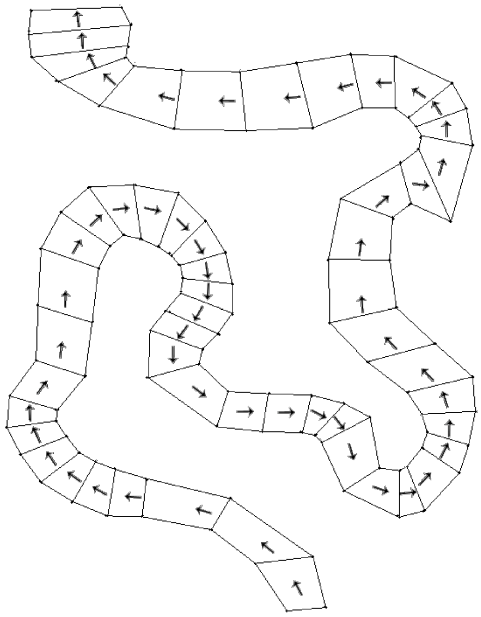

Figure 2.2: Path divisions for grading the Low-Speed Navigation Test

In the low-speed navigation test, I find the optimal direction by splitting the correct path into a set of discrete four-sided polygons. Because the path is essentially two-dimensional in the low-speed navigation test, I only need to specify these sample polygons in two-dimensions. Each polygon has two sides that correspond to the walls of the path, and the other two sides crossing over the path. The optimal direction is then $\vec{v} = (a - b)$ where a is the average of the points on the line segment crossing further down the path, and b is the average of the points on the line segment crossing further up the path.

The high-speed navigation test uses the same method as the low-speed navigation test, but because the path is three-dimensional, I specify several layers of paths. So if the

subject is closer to the ground I compare his direction to a slightly different optimal direction than if he were closer to the ceiling of the canyon.

Because I have only approximated the path with discrete segments, I have introduced some built-in error to the results. However, I am interested only in a paired-difference comparison. Given that this error does not change between the two performances of the same test, than this error will cancel out. For the purposes of this experiment I will assume that the error due to approximating the optimal direction will not change between two executions of the same test.

Chapter 3: Program Design

3.1 Generating a Point-Based Object

Point-based representations of objects are related to manifold-based representations, like bitmapped images are related to vector-based images. Point-based representations are essentially the results of a dense sampling of either a real-world object (such as can be obtained by a 3D-scanner) or a manifold-based object. For the purposes of this paper one can start generating a point-based representation of an object from a polygon-based object that contains enough primitives to accurately represent the detail of the intended real-world object. Many of the polygon-based models I start from were results of 3D-scans of real-world objects.

3.1.1 Sampling the Object-Space

We start the conversion from polygon-based objects by dividing the area surrounding the object into a number of cubes, then we determine which cubes intersect each polygon. To define our starting object, let $\exists S_X \subset \Re^3$, such that S_X is the set of all points along the surface of the object. Let $\exists V_X \subset S_X \subset \Re^3$, such that V_X is the set of all the sample

points defined in our starting object. In the polygon-based case, V_X would represent all the vertices in the object. Now let $\exists C \in \Re^3$ such that C is the mean of all the elements in V_X. Because V_X is finite, $\exists v_i \in V_X$ where v_i has the greatest distance, *D*, from C in a dimension, where $\exists D \in \Re$.

Now that the base model is clearly defined we must divide the area around the model into discrete chunks that represent portions of the continuous space around the object. Let $\exists A_X \subset \Re^3$ such that:

$$\forall a_i \in A_X \Leftrightarrow (C_x + D \geq x_i \geq C_x - D) \wedge (C_y + D \geq y_i \geq C_y - D) \wedge (C_z + D \geq z_i \geq C_z - D)$$

where $a_i = (x_i, y_i, z_i)$ and $x_i, y_i, z_i \in \Re$. Therefore A_X defines a cube around the center point of the object, and A_X encompasses each vertex in the object. Assuming that the input object is a polygon-based object, this means that A_X encompasses the entire object. In other manifold-based surfaces it is possible to have some portion of the surface whose distance from C is greater than D, but such representations are out of the scope of this paper.

Next, we will divide A_X into a set of $n \times n \times n$ subsections, where $n \in \mathrm{N}$ and represents the sampling density. Also, because we eventually want to represent this subdivided space as leaf nodes in an oct-tree, we want $n = 8^k$, where $k \in \mathrm{N}$ represents the depth of the oct-tree. Let $\exists G_X \subset \mathrm{N}^3$,

$$\forall g_i \in G_X \Rightarrow (n \geq x_i) \wedge (n \geq y_i) \wedge (n \geq z_i)$$

where $g_i = (x_i, y_i, z_i)$ and $x_i, y_i, z_i \in \mathrm{N}$ This way we have turned the continuous object space into discrete quantities, and therefore I am ready to obtain a uniform sampling of the surface of the object in object-space.

3.1.2 Simplifying Polygons

Now we will find each subdivison that intersects each polygon in the model, but first we must simplify each polygon into a series of triangles. Going back to the original source model, let F_X be the set of all polygons in the model, $F_X = \{P_1, P_2, \mathrm{K}, P_w\}$ and let $P_i \subseteq V_X$ be the set of vertices in the *i*th polygon of the model, such that $P_i = \{v_1, v_2, \mathrm{K}, v_k\}$ where $v_1, v_2, \mathrm{K}, v_k \in V_X$ and $v_1, v_2, \mathrm{K}, v_k$ are coplanar. Meaning that, for $\forall m_1, m_2, \mathrm{K}, m_k \in \Re$; and $\exists \vec{u}_1, \vec{u}_2, \mathrm{K}, \vec{u}_k \in \Re^3$ such that $\vec{u}_i = v_i - p$, $1 \le i \le k$, and p is any point on the *i*th polygon of the model; then:

$$m_1\vec{u}_1 + m_2\vec{u}_2 + \mathrm{K} + m_k\vec{u}_k = r_1\vec{b}_1 + r_2\vec{b}_2$$

where $r_1, r_2 \in \Re$ are variables and $\exists \vec{b}_1, \vec{b}_2 \in \Re^3$ are basis vectors for the plane that contains the represented polygon. Given this, we can simplify any P_i into a series of triangles $T_1, T_2, \mathrm{K}, T_s \subseteq V_X$, where $T_i = \{t_{i,1}, t_{i,2}, t_{i,3}\}$; $t_{i,1}, t_{i,2}, t_{i,3} \in V_X$, because

$$m_1(t_2 - t_1) + m_2(t_3 - t_1) = r_1\vec{b}_1 + r_2\vec{b}_2$$

obviously holds for $\forall m_1, m_2 \in \Re^3$. So now let $F'_X = \{T_1, T_2, \mathrm{K}, T_h\}$ where $\forall T_i$ is a triangle in $\Re^3$. The benefits to triangles are: from the previous argument one can

guarantee that the vertices of a triangle will always be coplanar; and triangles are always convex.

3.1.3 Finding Polygon Vertices

Now we start to find all the cube-shaped subdivisions $c_1, c_2, \ldots, c_k \in G_X$ that intersects the triangle defined by $T_j \in F'_X$. To do this, we first find each subdivision that contains a vertex of T_j. This is a fairly straightforward process, given $\forall t_i \in T_j$ where $t_i = (x_t, y_t, z_t);\ x_t, y_t, z_t \in \Re^3$, then $\exists c_i \in G_X$ where $c_i = (x_c, y_c, z_c);\ x_c, y_c, z_c \in \mathrm{N}$ such that:

$$x_t - (C_x - D) = n(x_c) + r_x \text{ such that } r_x \in \Re \text{ and } n > r_x \geq 0$$

$$y_t - (C_y - D) = n(y_c) + r_y \text{ such that } r_y \in \Re \text{ and } n > r_y \geq 0$$

$$z_t - (C_z - D) = n(z_c) + r_z \text{ such that } r_z \in \Re \text{ and } n > r_z \geq 0$$

I used the Division Algorithm to divide by *n* to emphasize the conversion from real numbers to natural numbers, and therefore the change from continuous quantities to discrete quantities.

3.1.4 Finding Polygon Edges

After finding the subdivision that contains each vertex of T_j, we can define a subset of subdivisions that bounds only the current triangle, $G_j \subseteq G_X$. We need to find all the subdivisions of $E_j \subset G_j$, such that $e_a \in E_j$ if and only if e_a intersects an edge of the

triangle, T_j. Given $\forall t_{j,a} \in T_j$ and $\forall t_{j,b} \in T_j$ such that $a \neq b$, we can define the function for an edge of T_j to be:

$$e_j(s) = t_{j,a} + s(t_{j,b} - t_{j,a})$$

for $s \in \Re$ and $1 \geq s \geq 0$. The symmetric equations are then:

$$s = \frac{x - x_a}{x_b - x_a} = \frac{y - y_a}{y_b - y_a} = \frac{z - z_a}{z_b - z_a}$$

where $(x, y, z) \in \Re^3$; $t_{j,a} = (x_a, y_a, z_a)$; and $t_{j,b} = (x_b, y_b, z_b)$. We can now test the bounds of each subdivision in G_j. For $\forall c_i \in G_j$, let $B_i \subset \Re$ be the set of bounds of c_i such that $B_i = \left\{x_c - \frac{n}{2}, x_c + \frac{n}{2}, y_c - \frac{n}{2}, y_c + \frac{n}{2}, z_c - \frac{n}{2}, z_c + \frac{n}{2}\right\}$, where $(x_c, y_c, z_c) \in \Re^3$ is the center of c_i in real continuous space. If $\exists d \in B_i$ such that when you apply the symmetric equation you get a valid *s* and point within the bounds of c_i; then the edge E_j intersects c_i, and $c_i \in E_j$. Repeat this process for each edge in T_j to find all $c_i \in E_j$.

3.1.5 Finding the Polygon Interior

Once we have found all the subdivisions that intersect the edges of T_j we can perform a sort of flood-fill to find all the elements of I_j, the subdivisions that intersect with the interior of T_j. We start this process by picking a dimension. Any dimension will work, but for the sake of demonstration we will pick the *x* dimension. Let $H_{x_a} \subset G_j$ where

H_{x_a} is the set of all subdivisions that share the *x*-value, x_a. Now find $E_j \mid H_{x_a}$. If there is only one subdivision in this intersection, then the slice H_{x_a} intersects only a vertex of T_j and contains no subdivisions that intersect the interior of T_j. If there is more that one subdivision in $E_j \mid H_{x_a}$, then pick a second dimension, for demonstration *y*, and find $c_s, c_l \in E_j \mid H_{x_a}$ such that c_s contains the smallest *y*-coordinate, and c_l contains the largest *y*-coordinate. Define a subset, $W_{x_a} \subseteq H_{x_a}$, that is inclusively bounded by the rectangle with c_s and c_l as corners. Now for $\forall c_i \in W_{x_a}$, because we are constraining the input to subdivisions that lie between two edges we can assert that, $c_i \in I_j$ if and only if c_i intersects the plane defined by T_j.

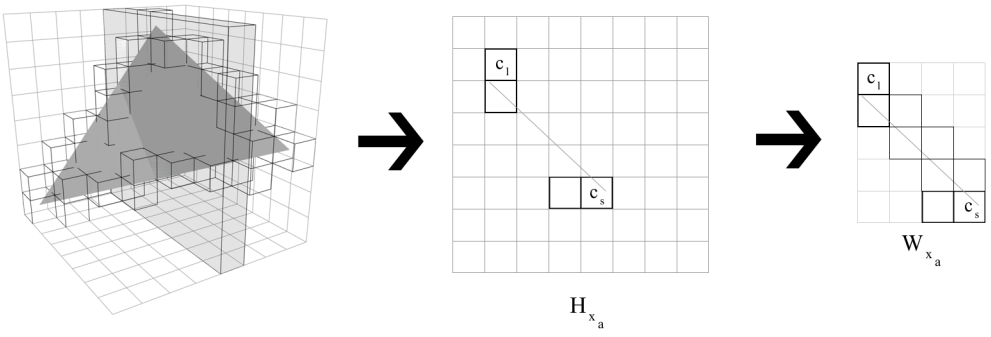

Figure 3.1: Finding interior subdivision intersections of a polygon

This is a fairly straightforward calculation. First we will define the plane, $P_j = \{(x, y, z) \in \Re^3 \mid Ax + By + Cz = D\}$ where $A, B, C, D \in \Re$ are coefficients and

$\left|\langle A,B,C\rangle\right| = 1$. Now we can project the center point of c_i, (c_x, c_y, c_z), onto the plane. We first need to find the distance, d, between (c_x, c_y, c_z) and P_j:

$$d = Ac_x + Bc_y + Cc_z - D$$

Then the projected point $p_i \in P_j$ is:

$$p_i = (c_x, c_y, c_z) - d(A,B,C)$$

If p_i lies within the bounds of c_i, then c_i intersects the plane defined by T_j. Therefore $c_i \in I_j$.

The subset $Z_j = E_j \cup I_j \subseteq G_X$ is the set of all subdivisions that intersect the polygon, T_j, where $\forall T_j \in F'_X$. Then define $Z_X = \{Z_1 \cup Z_2 \cup K \cup Z_k\}$ is the set of all subdivisions that intersect the surface of the original polygon-based model.

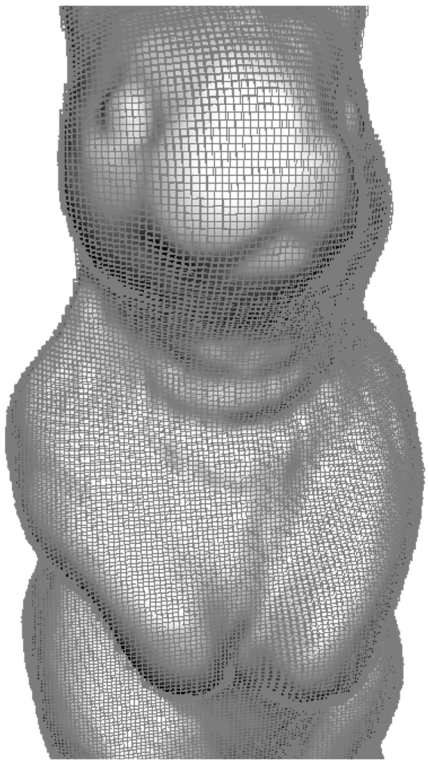

Figure 3.2: A surface with all its subdivisions computed

3.1.6 Calculating the Per-Sample Normal Vectors

Lighting effects are essential in providing the illusion of a continuous surface, and because each sample point is rendered separately, it is advantageous to calculate a normal per sample. This involves the interpolation of the normal vectors defined in the original polygon-based object. For this process we assume that the input object has a normal defined for each vertex. Here, we will use linear interpolation for simplicity, but other more advanced interpolation methods can be incorporated, and because this all pre-process more advanced interpolation methods will not effect render time.

In implementation, I used a three-dimensional array of lists to represent the subdivided space, G_X. These lists keep track of which polygons intersect each subdivision, and Z_X is simply the set of array elements that contain a non-empty list. For $\forall Z_i \in Z_x$ the list *S* of faces that intersect Z_i is non-empty. Then for $\forall T_j \in S$ there exist three vertices, v_1, v_2, v_3, and three corresponding normal vectors $\vec{n}_1, \vec{n}_2, \vec{n}_3$. Now consider the plane-space of the polygon, where any point on the polygon can be expressed as a linear combination of basis vectors $\vec{b}_1, \vec{b}_2$. Given that we already have three vertices on the plane, we can define $\vec{b}_1$ and $\vec{b}_2$ as $\vec{b}_1 = v_2 - v_1$ and $\vec{b}_2 = v_3 - v_1$. Therefore $\exists p \in \Re^3$ such that *p* is on the plane defined by T_j then:

$$p = k_1(v_2 - v_1) + k_2(v_3 - v_1) + v_1$$

where $k_1, k_2 \in \Re$ are coefficients. We can treat the normal vectors in a similar fashion. Consider an alternate vector space for the normal vectors on T_j. Any normal vector coming off of the plane defined by T_j can be defined as a linear combination of two basis vectors, $\vec{r}_1$ and $\vec{r}_2$. Using the corresponding given normal vectors we define $\vec{r}_1$ and $\vec{r}_2$ as $\vec{r}_1 = (\vec{n}_2 - \vec{n}_1)$ and $\vec{r}_2 = (\vec{n}_3 - \vec{n}_1)$. Therefore $\exists \vec{m} \in \Re^3$ such that $\vec{m}$ is a normal on T_j, then:

$$\vec{m} = k_1(\vec{n}_2 - \vec{n}_1) - k_2(\vec{n}_3 - \vec{n}_1) + \vec{n}_1$$

where $k_1, k_2 \in \Re$ are coefficients, and notably the same coefficients as those that define a point on the plane of T_j. Therefore, if we project the center point, (x_z, y_z, z_z) of Z_j on

to the plane by the method discussed previously. We can then find k_1 and k_2 that satisfy the linear combination for points on a plane, then we can plug k_1 and k_2 into the above linear combination to calculate $\vec{m}$, the normal vector at that point. The overall normal for Z_j is the average of the normal vectors calculated for each polygon that intersects Z_j.

3.1.7 Calculating Per-Sample UV Coordinates

Because texturing is a part of the experiment, the polygon-based input object will have a UV-coordinate specified with each vertex. A UV-coordinate it a two-dimensional coordinate that specifies where the associated vertex corresponds to the bitmap. The UV-coordinate for each sample point in the point-based representation must be computed in order to assign the correct color to that point. The method used is much like the method used to find the per-sample normal vectors.

For $\forall Z_i \in Z_X$ the list of polygons that intersect Z_i, S, is non-empty. Therefore let $\exists T_j \in S$ and $\exists u_1, u_2, u_3 \in \Re^2$ such that u_1, u_2, u_3 represent the UV-coordinates that correspond to the vertices of T_j. Any UV-coordinate on the plane defined by T_j can be defined as a linear combination of basis vectors $\vec{r}_1$ and $\vec{r}_2$. Given our input model we can define these basis vectors as $\vec{r}_1 = (u_2 - u_1)$ and $\vec{r}_2 = (u_3 - u_1)$. Therefore $\exists m \in \Re^2$ such that *m* is a UV-coordinate on T_j and:

$$m = k_1(u_2 - u_1) + k_2(u_3 - u_1) + u_1$$

where $k_1, k_2 \in \Re$ are coefficients. As with the per-sample normals the same conversion can take place. We can find the projected center point of Z_i and find k_1 and k_2 by evaluating the linear combination of vertices on the plane defined by T_j. The we can use these coefficients k_1 and k_2 to find *m*, where *m* is the UV-coordinate of the center point of Z_i projected onto the plane defined by T_j.

To find the color information at each sample point, Z_i, we must calculate the UV-coordinate of each polygon that intersects Z_i, and use it to find an appropriate color value. Then we must average together the different color values from the difference vertices to obtain out final per-sample color information.

3.1.8 Creating the Oct-tree

Now that we have sampled all the information needed from the polygon-based input object, we are ready to organize the sample points in a hierarchical fashion. For this we use an oct-tree structure. An oct-tree is a tree structure where the children divide the space of their parent into eight octants. Often the tree-structure itself provides enough geometric data to place the nodes of the tree in some space, so storing vertex values is unnecessary. Let Y_X be the parent node of the oct-tree, such that $Y_X = \{Y_1, Y_2, \mathsf{K}\ , Y_8\}$ where $\{Y_1, Y_2, \mathsf{K}\ , Y_8\}$ is an ordered set of sub-trees each containing their own ordered set of nodes. Any subtree may contain the empty set if it has no children nodes. That is, if the surface of the object does not pass through it's bounds. The exact location of the each

node is defined as follows: let the center of a subtree, Y_0, be $(x_0, y_0, z_0) \in \Re^3$; and let the dimensions of the subtree be $k \times k \times k$, where $k \in \Re$; then the geometry of it's child sub-trees are:

$Y_{0,1}$ is centered at $\left(x_0 + \frac{k}{4}, y_0 + \frac{k}{4}, z_0 + \frac{k}{4}\right)$, with dimensions $\frac{k}{2} \times \frac{k}{2} \times \frac{k}{2}$,

$Y_{0,2}$ is centered at $\left(x_0 - \frac{k}{4}, y_0 + \frac{k}{4}, z_0 + \frac{k}{4}\right)$, with dimensions $\frac{k}{2} \times \frac{k}{2} \times \frac{k}{2}$,

$Y_{0,3}$ is centered at $\left(x_0 + \frac{k}{4}, y_0 - \frac{k}{4}, z_0 + \frac{k}{4}\right)$, with dimensions $\frac{k}{2} \times \frac{k}{2} \times \frac{k}{2}$,

$Y_{0,4}$ is centered at $\left(x_0 - \frac{k}{4}, y_0 - \frac{k}{4}, z_0 + \frac{k}{4}\right)$, with dimensions $\frac{k}{2} \times \frac{k}{2} \times \frac{k}{2}$,

$Y_{0,5}$ is centered at $\left(x_0 + \frac{k}{4}, y_0 + \frac{k}{4}, z_0 - \frac{k}{4}\right)$, with dimensions $\frac{k}{2} \times \frac{k}{2} \times \frac{k}{2}$,

$Y_{0,6}$ is centered at $\left(x_0 - \frac{k}{4}, y_0 + \frac{k}{4}, z_0 - \frac{k}{4}\right)$, with dimensions $\frac{k}{2} \times \frac{k}{2} \times \frac{k}{2}$,

$Y_{0,7}$ is centered at $\left(x_0 + \frac{k}{4}, y_0 - \frac{k}{4}, z_0 - \frac{k}{4}\right)$, with dimensions $\frac{k}{2} \times \frac{k}{2} \times \frac{k}{2}$,

$Y_{0,8}$ is centered at $\left(x_0 - \frac{k}{4}, y_0 - \frac{k}{4}, z_0 - \frac{k}{4}\right)$, with dimensions $\frac{k}{2} \times \frac{k}{2} \times \frac{k}{2}$,

where the position in the ordered set of child nodes defines the exact geometry of each child node.

As discussed earlier the space around the object is divided $n \times n \times n$, into cubes; where $n = 8^k$. Let *k* be the highest level in our oct-tree. We can now just define the sampled space as the leaf nodes in the oct-tree, and those samples that do not intersect the surface S_X are empty nodes. From this base point we can fill in the parents of the non-null leaf nodes. The only geometric information that we need to store at each leaf node is the normal vector and the color information. To calculate the normal vector and color

information at each parent node, we take the average of the normal vectors and colors of each non-null child node.

We also need to specify a normal cone angle at each node for use in the rendering algorithm. This normal cone angle defines how much the node's children's normal vectors deviate from their parent's normal vector. For this I use a similar method to when I found the vertex in the polygon model that deviated the furthest from the center point. Let $\exists a \in \Re^3$ such that *a* is the furthest deviation in any dimension from the parent's normal vector; and $\exists \vec{n}_0 \in \Re^3$ that defines the normal vector for the parent node. Now let $C_n = \{\vec{n}_1, \vec{n}_2, \mathrm{K}\ , \vec{n}_s\}$, and $\exists \vec{n}_1, \vec{n}_2, \mathrm{K}\ , \vec{n}_s \in \Re^3$ be the normal vectors for each non-null child node. If there are no non-null child nodes, the current node is a leaf node, than $a = 0$. If there are some non-null child nodes, than $\exists \vec{n}_i \in C_n$ such that $h \in \vec{n}_i$ and $g \in \vec{n}_0$ where *g* is in the same dimension as *h*. Then let $d = h - g$ and $d \geq r - t$ for $\forall r \in \vec{n}_j$ and $\forall t \in \vec{n}_0$ where *r* and *t* share the same dimension, and for $\forall \vec{n}_j \in C_n$. Therefore *d* is the largest deviation from the parent's normal vector in any dimension. This is not an exact measure of the angle of largest deviation between the parent's normal vector and the children's normal vectors, but it is sufficient for the rendering algorithm.

3.2 Rendering a Point-Based Object

3.2.1 Overview

For the purposes of this project I decided to use as much graphics hardware support as possible to render a point-based object. Graphics hardware has been optimized for the rendering of polygon-based representations, so I am not able to fully utilize the hardware. However I do use the graphics pipeline to do much do much of the final transformation, lighting calculations, and final display of the point primitives. The hierarchical organization of the point-based object allows for an efficient rendering algorithm that adapts the detail to be rendered based on the amount of detail that will be visible on the final output.

We can take advantage of the hierarchical oct-tree structure we have organized the points in during rendering. The benefits of the hierarchical structure are: we can easily display the model at a coarse resolution if we need to, and we can perform backface culling and frustum culling at coarser resolutions, for a more efficient algorithm.

During rendering we perform a depth-first traversal of the oct-tree, because it requires the least overhead and is the most intuitive method for this application. Botsch et al. discussed the possibilities of using a breadth-first traversal, and showed that while it offered some benefits, it also cost a lot of overhead. Figure 3.3 shows the pseudocode for traversing the each node for rendering.

```
RenderNode
{
    if (node is empty)
       return;

   if (subtree is front facing)
   {
      Calculate the node's projected size;
      if (subtree is within the view frustum) and  (
      (node is a leaf node) or (the benefit of recursing
      further is low)  )
      {

         if (node is within the view frustum and the node
         is front facing.)
         {
            draw a point;
         }
      }

      Render each child node;
   }
}
```

Figure 3.3: Psuedocode for point-based rendering

We now examine the details of this algorithm.

3.2.2 Backface Culling

As stated earlier, we can cull off entire subtrees of the oct-tree by adapting basic culling procedures. Backface culling is a procedure that determines whether or not a primitive is turned towards the camera. In many cases it is beneficial to constrain rendering only to those primitives that face the camera, as the primitives that do not face the camera are often on the hidden side of objects and will not appear in a final image anyway.

To determine whether or not a primitive is back-facing let $\exists \vec{n} \in \Re^3$ such that $\vec{n}$ is the normal vector to the primitive. In this case the rendering primitives are the points

represented by each node in the oct-tree. Let M be a 4×4 matrix such that $\vec{n}M = \vec{e}$ where $\vec{e}$ is the normal vector with respect to the camera position. In other words, M is the transformation from world-space into camera-space. If the resulting camera-space vector, $\vec{e}$ has a negative z-component than the normal is facing back towards the camera and the normal is considered front-facing. Otherwise the normal is considered back-facing and we do not render the corresponding primitive.

Using the normal cone angle discussed earlier we can easily extend this operation to determine whether or not the cone of normal vectors is front of back-facing. After we determine $\vec{e}$, the camera-space normal vector of a primitive, we simply check to see if the z-coordinate of $\vec{e}$ is greater than the cone angle. If this is the case, than the entire cone of normal vectors is facing away from the camera and the entire sub-tree under the current node can be culled.

3.2.3 Calculating Point Size

Before we can discuss Frustum Culling we must first discuss the method we use to find the amount of space that each point will occupy on the screen. Here is an example where current graphics hardware systems do not provide an optimal environment for point-based rendering. The only way to specify a point-size in the OpenGL API is to specify the screen-size of the point. This requires the point-rendering algorithm to already know the projection of each point before it is sent through the pipeline.

Let $\exists p \in \Re^3$ such that p is the coordinate of a point, and let $\exists l \in \Re$ that represents the distance from the center point p to any side of the cube represented by the current node. Also let $\exists M, P$ that are 4×4 matricies. M represents the transformation from world-space to camera-space as discussed earlier, and P represents the transformation from camera-coordinates to clip-coordinates. Clip coordinates are used further in frustum culling, but for these purposes we use it as an intermediate stage on the way to projected coordinates. Let $p_c = (x_c, y_c, z_c, w_c)$ be a clip-coordinate, than:

$$p_p = \left(\frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c} \right)$$

represents the projected coordinate of our original point p.

We use this method to find the projected size of a point p by first calculating the z-coordinate, z_c, of p with respect to the camera. Next we let this z_c represent the distance of the point to the camera. In order to calculate the real distance of a point to the camera we would have to calculate:

$$d = \sqrt{x_c^{\,2} + y_c^{\,2} + z_c^{\,2}}$$

which would take a significant amount of computation if it had to be done for each point. Therefore we can shorten the computation by using just the z_c value and still get satisfactory results.

Because l simply represents the smallest distance from the center point, p, to the edge of the cube, we must compute $s = l \times \sqrt{3}$ so that s is the greatest length from p to

the edge of the cube. This way we can eliminate any holes that may occur in between points. Next we define a point *r* as $r = (s, 0, z_c)$. Now we take the clip-coordinates of *r* and then the projected coordinates of *r*, r_p . Because of how we defined *r* the *x*-coordinate of r_p is s_p the projected size of the original point *p*. We now have a value we can input to the hardware through the OpenGL interface to specify the size of any point in world space. We may, however, get a negative value for s_p. If this occurs, than the point *p* must have been behind the camera. If this occurs we cull this node off as part of the frustum culling.

We also use this point size to partially determine how many levels to traverse during any given rendering process. If the point size falls below a specified threshold, than we will draw the current point and not traverse any of the current node's children nodes. This allows us to not render any primitives that are not noticeable on the final output, therefore making this algorithm output-sensitive.

3.2.4 Frustum Culling

Frustum culling is not rendering any primitive that falls outside of the area being viewed by the camera. This cuts down on the total number of primitives that must be rendered, therefore increasing efficiency. In the case of point-based representations arranged into hierarchical data structures we can use this technique to further remove sub-trees from calculation and increase the efficiency of the rendering algorithm.

In order to determine whether or not a primitive lies within a view frustum we simply calculate the clip coordinates discussed above by taking any point *p* as well as the world-to-camera transformation *M* and the camera-to-clip transformation *P* and applying them to our original point *p* to get p_c the clip-coordinate of *p*. Now, given $p_c = (x_c, y_c, z_c)$, *w* and *h* are the width and height of the final viewport, and (x_0, y_0) is the center of the viewport, than the screen coordinates $p_s = (x_s, y_s)$ are:

$$\begin{pmatrix} x_s \\ y_s \\ z_s \end{pmatrix} = \begin{pmatrix} \left(\frac{w}{2}\right)x_c + x_0 \\ \left(\frac{h}{2}\right)y_c + y_0 \\ \left(\frac{1}{2}\right)z_c + \frac{1}{2} \end{pmatrix}$$

Now if $(0 \le x_s \le w) \wedge (0 \le y_s \le h) \wedge (0 \le z_s \le 1)$ than *p* is within the view frustum, otherwise *p* is not within the camera's view, and is culled off.

The most obvious way to extend this to determine whether or not an entire sub-tree is within the view frustum is to compute whether or not any of the corners of the sub-tree's bounding sphere occur within the view frustum. However this would involve the projection of eight points into screen-space.

A faster way to extend this operation to sub-trees is the use of the point's already calculated projected size. This way, given the screen-coordinates of a node's center point p_s, we can use the projected point size s_p so if:

$$(-s_p \le x_s \le w + s_p) \wedge (-s_p \le y_s \le h + s_p)$$

Than at least part of the sub-tree is within the view frustum, else the entire sub-tree can be culled off. Notice that we remove the evaluation of the z_s component. This is

because the point size does not correspond to the *z*-coordinate of the screen-space point, due to the fact that the *z*-dimension of the screen space is scaled differently than the *x* and *y* dimensions. Instead of spending extra computation to try and account for this scaling, we simply do not evaluate the *z* component, and we get satisfactory results.

3.2.5 Determining Traversal Depth

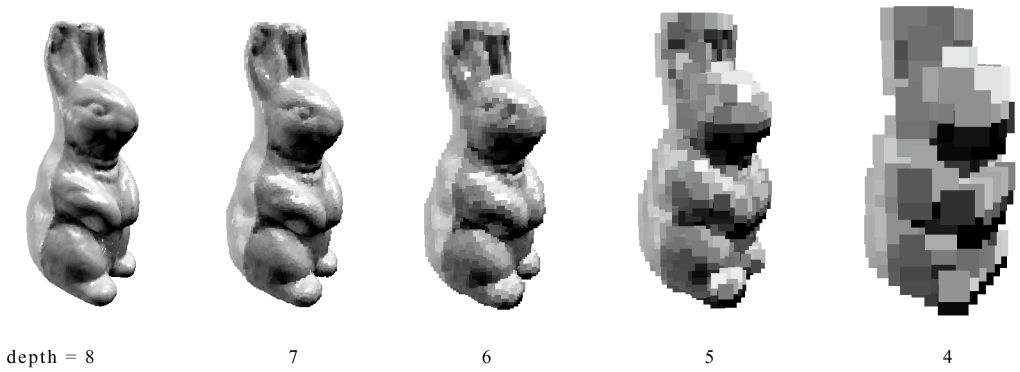

Figure 3.4: A surface rendered at different traversal depths

In addition to not traversing sub-trees of nodes that do not occupy a significant portion of the output, we can specify a maximum depth for the rendering algorithm to traverse down to. The purpose of this is to allow for fluid movement even when the amount of leaf nodes visible require so much computation that interactive frame rates are not possible. This is accomplished by setting a maximum frame speed parameter, $f_{\max}$, a minimum frame speed, $f_{\min}$, and a minimum traversal depth $k_{\min}$. It also keeps track of how far it it traversed the tree during the last render, k_0. The render function for the point representation takes in the amount of time needed to render the last node, f and whether

or not an event occurred such as the user requested a change in the viewpoint. The rendering function also outputs a Boolean value based on whether or not we can refine the model further without increasing frame speed over the threshold. Before we start traversing the oct-tree of the point representation we determine the traversal depth. Figure 3.5 shows the psuedocode for determining traversal depth. Using this method we can adjust the traversal depth so that we can adjust the level of detail to keep an interactive frame rate. This will result in very coarse models in scenes where many points must be rendered, however this is the tradeoff which we mean to evaluate.

```
Render Object
{
        refine = true;
        if (f < f_min)
        {
                increment k_min;
                k_0 = k_min;
        }
        if (f > f_max)
        {
                decrement k_0;
                refine = false;
        }

        if (an event occurred)
                k_0 = k_min;

        Traverse hierarchy to k_0;
        return refine;
}
```

Figure 3.5: Psuedocode for determining traversal depth

3.2.6 Splatting

In order to facilitate a comparison, I have selected the rectangular point-representations used by graphics hardware in order to trade-off some of the visual quality produced by point-based models for the increased speed that they can support.

Chapter 4: Results

4.1 Speed Results

A discussion of the computer performance results is essential to understanding the perception-based user-performance results discussed later. These tests were all performed on a Pentium 4, 2.0Ghz processor with 512MB of memory. The operating system was RedHat Linux 9.0. I chose to use Linux because its handling of the large amounts of memory required by the point-based representation was better than that of Windows XP. These computers used an Nvidia RIVA TNT 2 Model 64 Pro graphics card.

The point-based object representation used in this project allows for the amount of rendering primitives sent to the graphics hardware to vary based on how many primitives are viewable. Graphics hardware will perform this sort of operation on polygon primitives as well, however it hardware usually performs some calculations, such as transformations, on all primitives including those that are eventually excluded. Culling out large amounts of data in software reduces the amount of these general calculations the hardware must perform. Therefore, the speed needed to render objects represented by points varies greatly on how many points are viewable at any given

moment. Figure 4.1 shows the amount of time to render one frame at a given quantity of points. These results were obtained from the data collected from the subjects. It is a combination of data from each test. The error bars in Figure 4.1 are the standard deviation from the mean of each set of samples for a particular point quantity. They help to illustrate the variations in the rendering speed at each quantity level.

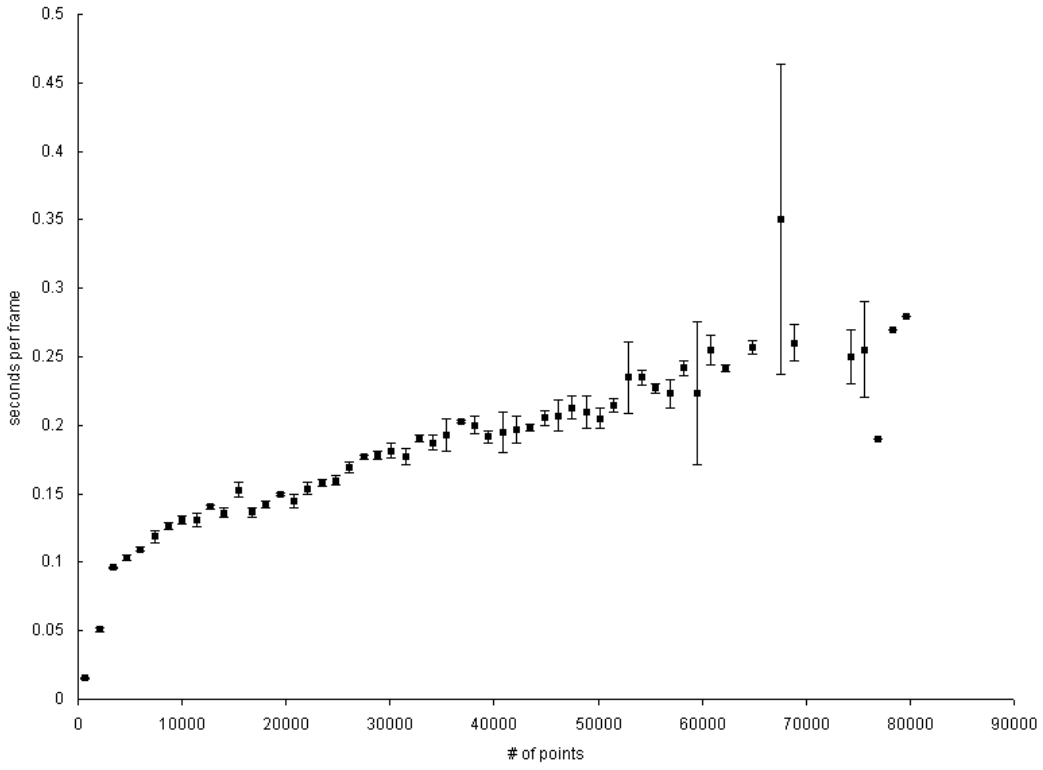

Figure 4.1: Per point quantity rendering speed results

These results are not surprising. The time needed to render a frame increases with the number of primitives sent to the graphics hardware. Figure 4.2, Figure 4.3, Figure 4.4, Figure 4.5 show the number of frames that rendered a certain number of points for each

type of test. Looking at these in conjunction with Figure 4.1 gives some idea as to the rendering performance during each test. Note the difference between the graphs of the Object Recognition tests and the Navigation tests.

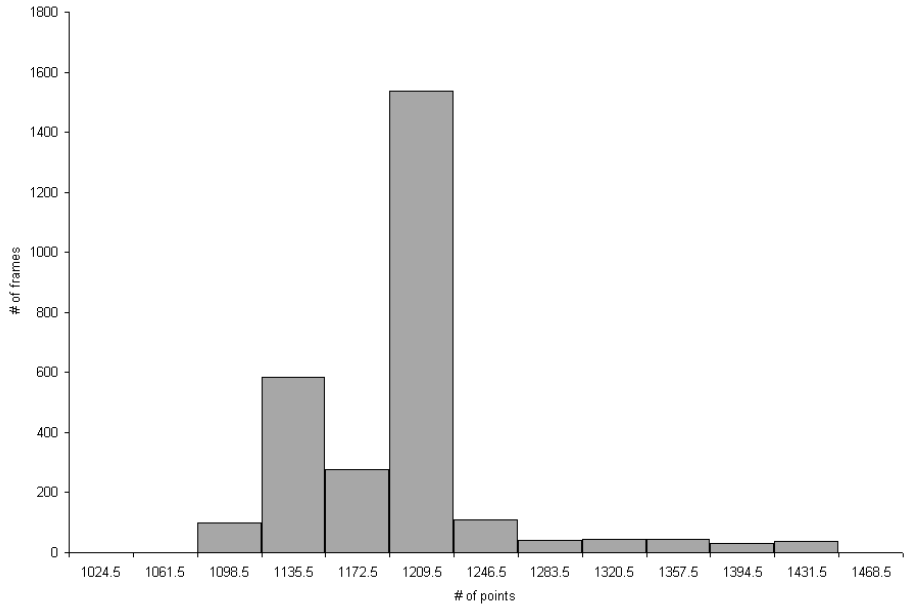

Figure 4.2: Distribution of points-per-frame for the Far Object Recognition Test

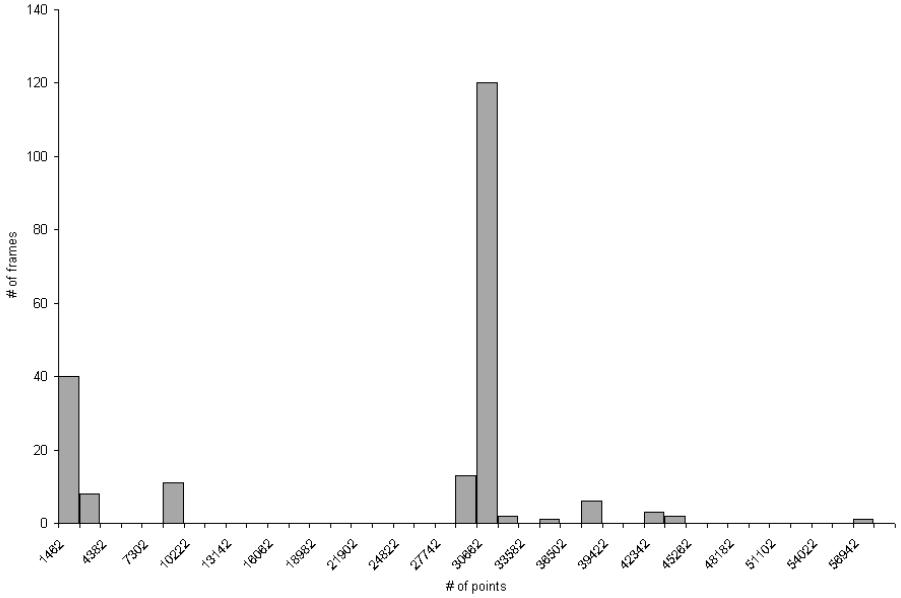

Figure 4.3: Distribution of points-per-frame for the Near Object Recognition Test

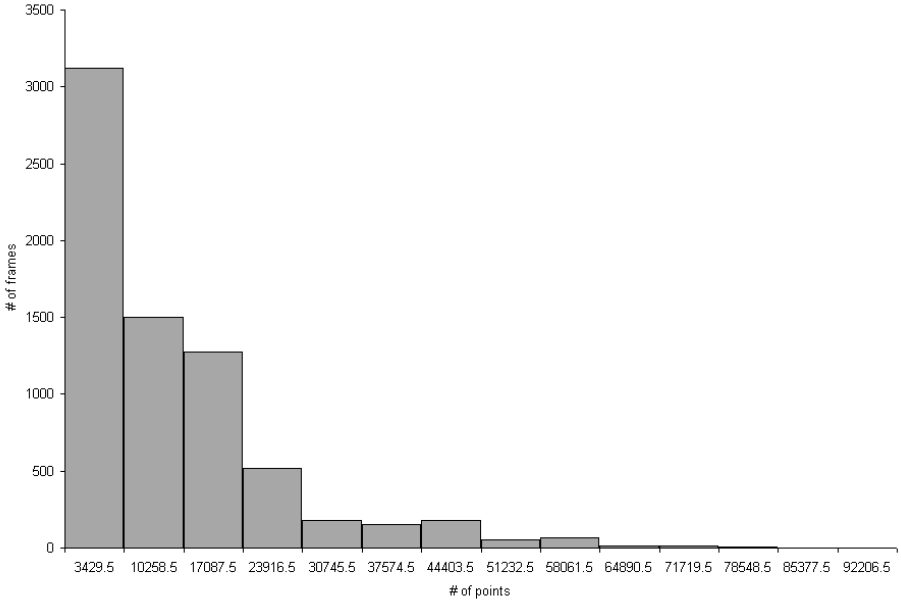

Figure 4.4: Distribution of points-per-frame for the Low-Speed Navigation Test

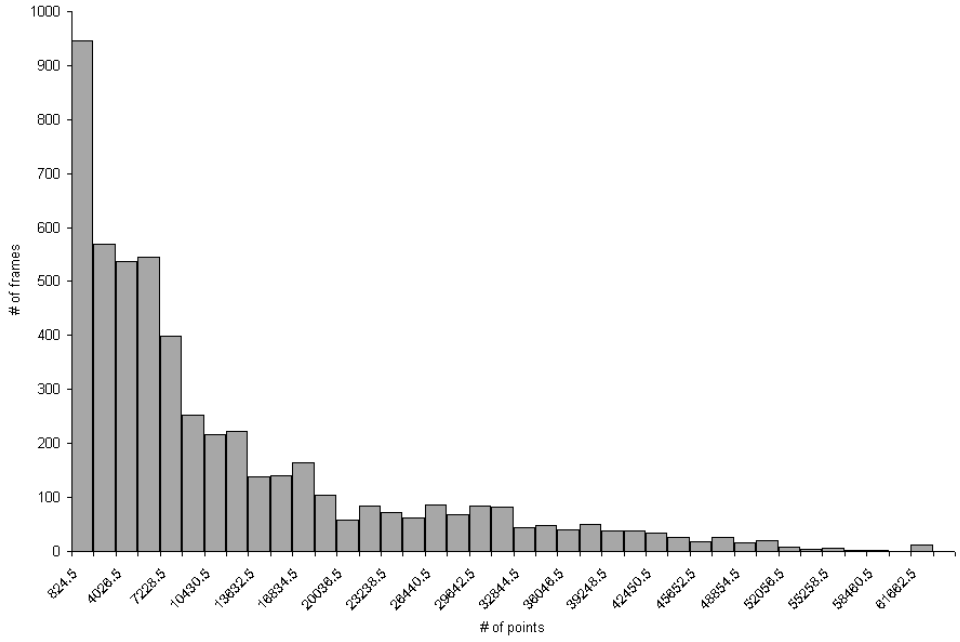

Figure 4.5: Distribution of points-per-frame for the High-Speed Navigation Test

The terrain models in the navigation tests require a large amount of points to display properly, because they are often rendered very close to the viewer. This leads to the navigation tests frequently rendering a large amount of points, as compared to the object recognition tests.

Trial		Average	Median	Minimum	Maximum	Standard Deviation
near:	T	0.05	0.05	0.02	0.08	0.011
	P	0.08	0.1	0.0	0.2	0.046
far:	T	0.16	0.16	0.08	0.19	0.019
	P	0.01	0.01	0.0	0.02	0.007
low-speed:	T	0.05	0.04	0.01	0.11	0.010
	P	0.13	0.12	0.01	0.91	0.038
high-speed:	T	0.04	0.04	0.01	0.07	0.012
	P	0.11	0.11	0.0	0.34	0.06

Table 4.1: Test performance results (seconds per frame)

Table 4.1 shows statistics about the overall rendering speed of the two different methods on each test. T indicates data for scenes using the polygon-based representation, and P represents data for scenes using the point-based representation. To estimate the difference between the mean rendering speeds of each method we use the following to construct a confidence interval:

$$(\bar{x}_1 - \bar{x}_2) \pm z_{a/2}\sqrt{\frac{{\sigma_1}^2}{n_1} + \frac{{\sigma_2}^2}{n_2}}$$

Where $\bar{x}_1$ and $\bar{x}_2$ are the two mean rendering speeds, σ_1 and σ_2 are the population variances of the different rendering speeds from each method, and n_1 and n_2 are the number of readings taken from each method. The $z_{a/2}$ term represents the margin of error that we want this confidence interval to represent. Specifically the $z_{a/2}$ represents the area under a normal distribution curve that represents the probability of the mean being at a certain point. The assumption here is that the sample sizes are large enough to apply the Central Limit Theorem to conclude that the possible differences between the means have a normal distribution. The two values produced by this equation are the upper and lower bounds of our confidence interval. For the purposes of this investigation we wish to construct a 99% confidence interval, therefore $z_{a/2} = 2.58$ for all our tests. This gives us a large margin of error, so that we can make very solid assertions about the data. Also, because our sample sizes are very large, we can use the standard deviations of each test to estimate σ_1 and σ_2.

For the Near Object Recognition test we collected 207 samples for the point-based rendering method and 314 samples for the polygon-based method. Using these and the values from the tables we have the difference of the average time needed for point-based rendering from the average time needed for polygon-based rendering is somewhere between 0.022 and 0.037. Therefore we can conclude that the polygon-based rendering speed is faster than the rendering speed of the point-based representations.

For the Far Object Recognition Test, I collected 2,845 and 296 samples for the point-based version and polygon-based version, respectively. The point-based version of this test collected far more samples than the polygon-based version because the samples are collected per-frame, and as we shall see, the point-based version was able to render many more frames per second than the polygon-based version. Under a 99% confidence interval, the difference between the polygon-based Far Object Recognition Test and its point-based counterpart is between 0.147 and 0.153. Therefore we can firmly assert that in the Far Object Recognition test the point-based rendering method produced much faster rendering speeds than that of the polygon-based test.

The results of the Far Object Recognition Test and the Near Object Recognition Test are due to the difference in output-sensitivity between the two representation schemes. Objects represented with points can employ a hierarchical data structure within the object to adjust the level of detail to render. Polygon-based objects typically have no such capability. The data suggests that polygon-based representations even incur a speed

penalty when being viewed from far away. This could possibly be due to complications in the rasterization of many polygons that are very closely grouped together.

The sample sizes for the Low-Speed Navigation test are 7,065 and 45,992 for the point-based version and polygon-based version respectively. Again, the reason for the difference in sample sizes is that the polygon-based method yielded higher rendering speeds and therefore rendered more frames, collecting more samples. The bounds for the 99% confidence interval of the difference between the polygon-based Low-Speed Navigation Test and its point-based version are 0.079 and 0.080. Therefore the polygon-based rendering method is much faster than the point-based rendering method when applied to the Low-Speed Navigation test.

For the High-Speed Navigation tests the samples size for the point-based method is 207, and the sample size for the polygon-based method is 24,020. Thus under a 99% confidence interval the difference between the average rendering speed for the polygon-based test and the average rendering speed for the point-based test is between 0.067 and 0.072. Therefore, in the High-Speed Navigation Test the polygon-based method yields faster rendering speeds.

4.2 Object Recognition Results

In order to evaluate the object recognition tests I have examined the naming time for the second execution of each test, treating the first execution as a practice run. I can do this because the two executions display different objects to the user, making the results

independent. The object to be identified in the far object recognition test was a dog, and the object in the near test was a rabbit. The testing program randomly gave each of the fifteen subjects a point-based object or a polygon-based object. Table 4.2 reports the number of subjects who accurately identified the object in each test. P stands for a test using the point-based object representation, and T stands for a test using the polygon-based representation.

Trial		Correct	Incorrect	Percent correct
near:	T	6	0	100%
	P	7	2	78%
far:	T	4	1	80%
	P	9	1	90%

Table 4.2: Object Recognition accuracy results

These results do not indicate a significant difference in overall accuracy between the two types of representations. Instead we must look at the amount of time each subject needed to identify an object. Table 4.3 shows statistics on how long it took subjects to identify the object presented in either tests. This table excludes those who incorrectly identified the object. Each unit in Table 4.3 represents a thousand clock cycles.

Trial		Average	Median	Minimum	Maximum	Standard Deviation
near:	T	892	980	480	1290	347
	P	2110	1760	1480	4010	889
far:	T	7157	7055	4760	9760	2492
	P	2861	2090	1010	6620	1987

Table 4.3: Object Recognition timing results (in thousands of clock cycles)

Looking first at the Near Object Recognition Test we can analyze whether or not the data sets are significantly different using a *t*-test of the difference between two means.

We cannot use the method from Section 4.1, because our sample size is small, and the Central Limit Theorem no longer applies. Instead we must calculate a test statistic *t*:

$$t = \frac{(\bar{x}_1 - \bar{x}_2) - D_0}{\sqrt{s^2\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}}$$

where $\bar{x}_1$ and $\bar{x}_2$ are the two population means and n_1 and n_2 are the two population sizes, and D_0 is our guess for the difference in the means. D_0 will be 0 because our null hypothesis will typically be that the two populations are the same. The variance of the population *s* is found by finding the weighted average between the two different standard deviations:

$$s^2 = \frac{(n_1 - 1)s_1^{\,2} + (n_2 - 1)s_2^{\,2}}{n_1 + n_2 - 2}$$

where s_1 and s_2 are the standard deviations of the two populations. Also, we must define a degrees of freedom variable, *df*:

$$df = n_1 + n_2 - 2$$

Now we can use our degrees of freedom to compare our calculated *t*-statistic and to the Student's *t* distribution to find the probability that our guess is correct.

For the Near Object Recognition Test, our null hypothesis is that the two means are essentially equal. Whereas the alternate hypothesis is that the mean for the point-based representation is significantly larger than that of the polygon-based test. For this test our samples sizes are 6 and 7 for the polygon-based version and the point-based

version respectively, resulting in a degrees of freedom of 11. Using the above formulas yields a *t*-value of 3.34, which is larger than the *t*-value for 0.005 percentile, meaning that there is less than a 0.5% probability that the means of the two data sets are essentially the same. Therefore we can confidently say that the mean naming time for the point-based object is significantly larger than the mean naming time for the polygon-based object. The subjects had a much harder time identifying the model, in this case a rabbit, from close up when it was rendering using points. This makes sense, because the point-based representation suffers from severe aliasing when viewed from up close. The object surface is not interpolated between samples, but it is roughly estimated using the closest sample value. Polygon-based objects, on the other hand, are interpolated and therefore yield a significantly better picture than the point-based models when viewed from a close distance.

Now we consider the Far Object Recognition Test. The sample size for the polygon-based test is 4 and the sample size for the point-based test is 9, therefore we have 11 degrees of freedom. Our null hypothesis is that the two means are the same. This time our alternate hypothesis is that the mean for the naming times of the point-based object is less than the mean for the naming times of the polygon-based object, as the data would suggest. Our calculated *t*-value is 3.14, which is larger than the *t*-value for the 0.005 percentile. Therefore the probability that the two data sets are the same is less than 0.5%. We can reject the null hypothesis and assert that subjects were able to identify the point-based object faster than the polygon-based object. This illustrates that

the output-sensitivity of the point-based representation enables for more efficient object representation when the user sees the object from far away.

Overall, these results support my hypothesis of how point-based representations will behave with respect to polygon-based representations. The interpolation of the polygon-based representations allow for better rendering for an object that is close to the viewpoint, whereas the point-based representation yields better results in the case where polygons become smaller than points, such as in an object far from the viewpoint. Now let us examine how the two object-representation methods compare when applied to interactive environments.

4.3 Low-Speed Navigation Test Results

The Low-Speed Navigation Test prompted the subject to navigate a smooth path through a rocky terrain from a start point to a finish point. Here I employ the method described in Section 2.4 to generate a rating of the subject's performance for this test. This test was performed twice in a row for each user, the first time using polygon-based representations and the second time using either polygon-based representations or point-based representations depending on a random number generated by the program. Table 4.4 shows the grades gathered by the tests. Test A uses polygon-based representations in both the control group and the variable group. Test B uses polygon-based representations in the control group and point-based representations in the variable group.

Trial		Average	Median	Minimum	Maximum	Standard Deviation
control:	A	63.6%	63.3%	46.2%	82.2%	11.5%
	B	72.5%	74.1%	58.7%	85.3%	10.4%
variable:	A	72.5%	74.6%	60.6%	79.4%	7.11%
	B	69.3%	74.4%	42.2%	81.2%	15.4%

Table 4.4: Low-Speed Navigation Test results

The values alone show some interesting possible trends. In the control group performance seems to have improved between test A and B. This is not surprising because the subject is presented with the same terrain to navigate, and they can navigate the terrain more easily the second time. The variable group seems to have started out well in test A, and slightly decreased in performance in test B. However, notice that test B's standard deviation is much larger, and that the minimum and maximum values of test B reach further extents. This suggests that people were still able to improve performance between the two different kinds of object representation, and others had a very difficult time with this task when it was using point-based rendering.

In order to statistically validate these observations we must perform a paired-difference analysis. Because test A and B present the same objects the two tests are not independent, so instead of examining the performance on each test, we must examine the difference in performance between the two tests.

Trial	Average	Median	Minimum	Maximum	Standard Deviation
control:	8.7%	7.3%	-4.4%	20.8%	8.1%
variable:	-4.3%	0.0%	-32.4%	13.8%	19.7%

Table 4.5: Improvement between Low-Speed Navigation Tests A and B

The sample size for the control group is 10, and the sample size for the variable group is 5. Consequently we have 13 degrees of freedom. Our null hypothesis is that the population means are the same; while our alternate hypothesis is that the difference in the averages for the control group is greater than the differences in the averages for the variable group. Using the calculations described in Section 4.2 we find a calculated *t*-value of 1.65, which lies between the 0.025 and 0.05 percentiles on the Student's *t* distribution. This means that there is a 2.5% to 5% chance that the difference in improvement between the control and variable groups is due to the variance in the samples and not due to the mean improvement of the variable group being less than the control group. This chance is small, but not insignificant; therefore we can only tentatively state that the point-based objects yield lower performance in this test than do the polygon-based objects, but we cannot assert this as a fact.

4.4 High-Speed Navigation Test Results

The High-Speed Navigation test presented the user with a canyon to fly a ship through. The set-up is the same as the Low-Speed Navigation test with two tests per subject and each subject is placed into either a control group or a variable group.

Trial		Average	Median	Minimum	Maximum	Standard Deviation
control:	A	72.6%	77.8%	34.4%	88.44%	21.9%
	B	80.2%	91.3%	37.5%	93.1%	24.0%
variable:	A	73.5%	71.6%	58.1%	89.8%	12.8%
	B	64.6%	63.5%	26.2%	92.4%	23.1%

Table 4.6: High-Speed Navigation Test results

The results from the High-Speed Navigation Test, shown in Table 4.6, are very similar to those collected from the Low-Speed Navigation Test from Table 4.4. They indicate that the point-based representations used in variable test B may have caused some subjects to perform worse, while other subjects were still able to improve their performance between tasks. However, there were three cases that had to be thrown out because a collision detection failure resulted in the program not being able to output data, and the grading method used to analyze the results had no way to take that into account. Two of these cases occurred in a scene that used point-based representations, and one case happened while using the polygon-based representations.

Trial	Average	Median	Minimum	Maximum	Standard Deviation
control:	7.6%	5.6%	2.9%	15.8%	5.0%
variable:	-8.9%	-7.0%	-41.7%	20.2%	23.6%

Table 4.7: Improvement between High-Speed Navigation Tests A and B

The resulting sample sizes for this test are 5 and 6 for the control and variable groups, respectively. We have 9 degrees of freedom. Using the same paired-difference analysis from the Low-Speed Navigation Test we compare the difference between tests A and B, shown in Table 4.7 with the calculations described in Section 4.2, we calculate a *t*-value of 1.522, which falls between the 0.05 and 0.10 percentiles on the Student's *t* distribution. This means that there is a 5%-10% percent chance that the difference in means of the two groups is due to the variance of the subjects' performances. Therefore, we cannot make any solid assumptions about the differences between the improvements in performance.

Chapter 5: Conclusions

From the results, one can clearly see that using polygon-based representations yield a scene that is much faster and clearer than one that uses point-based representations. From the standpoint of visual fidelity, all tests except the Far-Object Recognition Test pointed in favor of using polygon-based objects. Point-based representations especially fail in the case of terrains, large objects that are often seen partly close up. Because of the user's proximity to these kinds of objects, they require an especially dense sampling, and that large quantities of points be rendered. Looking at the difference in the quantity of points rendered between the Low-Speed Navigation Test, Figure 4.4, and the Near Object Recognition Test, Figure 4.3, I conclude that the large quantities of points rendered per frame are mainly responsible for the difference in rendering speeds between the two representations. Therefore any advances in improving the efficiency per-point of point-based rendering would significantly affect these results. Currently, graphics hardware is built with polygon-based rendering in mind, which can result in difficulties for non-polygon-based methods. If optimizations were made for point-based rendering in hardware, these results would change significantly [4].

The Far-Object Recognition Test stands out as an exception from the rest of the tests. The point-based version of the Far Object Recognition test yielded faster rendering speeds, Table 4.1, and better user performance, Table 4.3. The increase in speed is

undoubtedly due to the fact that point-based models are output-sensitive, and can reduce the number of primitives rendered as they become less significant on screen, Figure 4.2. This test affirms that there are currently some cases where using the point-based representation scheme in lieu of polygon-based objects offers an improvement in speed and clarity.

Using current consumer graphics hardware the point-based representation yields a significantly lower visual fidelity than that of polygon-based representations when applied to interactive scenes. However, these results still support the prediction that because polygons are not output sensitive, using point-based representations for three-dimensional models will eventually be advantageous.

Chapter 6: Future Work

The results indicate that the current form of point-based rendering performs well when viewing an object that the user does not have to get very close to. This algorithm suffers when it must render a terrain or landscape that the user must get very close to and interact with. Therefore a new method of rendering for point-based representations of terrain would greatly benefit this research. Such a method could take into account the distance of a sub-tree from the viewpoint to determine how many levels to render.

Another drawback to the point-based rendering algorithm proposed in this paper is that calculations done during tree-traversal involve floating-point values. If these values could be represented as integers or some other discrete quantities and only converted to floating point before being processed by the graphics hardware, the time needed for computation may reduce and provide better performance.

An advantage of point-based rendering is that the lighting calculations may be performed for each point primitive, providing for better detail representation across the objects surface. This effect is studied somewhat in this research, but several other implications exist; such as being able to easily vary the specular and ambient material attributes with each point.

The user-performance results suggest a strong correlation between rendering speed and the visual fidelity of interactive scenes. Further research in the strength of this correlation and the relationship between object clarity and visual fidelity of interactive

scenes would be beneficial for better determining how much of the tradeoff between rendering speed and object detail is perceived by the user.

Bibliography

[1] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Point Set Surfaces. In *Proceedings of the conference on Visualization '01*. IEEE Computer Society, 2001.

[2] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient High Quality Rendering of Point Sampled Geometry. In *Thirteenth Eurographics Workshop on Rendering*, pages 53-64. The Eurographics Association, 2002.

[3] Chandrajit Bajaj, Insung Ihm, Sanghun Park. 3D RGB Image Compression for Interactive Applications. In *ACM Transactions on Graphics,* v.20 n.1 pages10-38. ACM Press, 2001.

[4] Liviu Coconu and Hans-Christian Hege. Hardware-Accelerated Point-Based Rendering of Complex Scenes. In *Thirteenth Eurographics Workshop on Rendering*. The Eurographics Association, 2002.

[5] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, M. Howard. Towards an Interactive High Visual Complexity Animation System. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, pages 289-299. ACM Press, 1979.

[6] Changcheng Huang, Michael Devetsikiotis, Ioannis Lambadaris, A. Roger Kaye. Modeling and Simulation of Self-Similar Variable Bit Rate Compressed Video: A Unified Approach. In *Proceedings of the conference on Applications, technologies, architectures and protocols for computer communication*, pages 114-125. ACM Press, 1995.

[7] Arie E. Kaufman. Volume Visualization. In *ACM Computing Surveys*, v.28 n.1 pages 51-64. IEEE Computer Society Press, 1993.

[8] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, Gregory A. Turner. Real-Time, Continuous Level of Detail Rendering of Height Fields. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109-119. ACM Press, 1996.

[9] Marc Levoy and Turner Whitted. The Use of Points as a Display Primitive. Computer Science Department, The University of North Carolina, 1985.

[10] Charles Loop, T. D. DeRose. Generalized B-Spline Surfaces of Arbitrary Topology. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 347-356. ACM Press, 1990.

[11] Bui Tong Phong, Illumination for Computer Generated Pictures. In *Communications of the ACM*, v.18 n.6 pages 311-317. ACM Press, 1975.

[12] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, Markus Gross. Surfels: Surface Elements as Rendering Primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335-342. ACM Press, 2000.

[13] W. T. Reeves. Particle Systems—a Technique for Modeling a Class of Fuzzy Objects. In *ACM Transactions on Graphics,* vol. 2 issue 2, pages 91-108. ACM Press, 1983.

[14] Patrick Reuter, Ireneusz Tobor, Christophe Schlick, Sébastien Dedieu. Point Based Modeling and Rendering Using Radial Basis Functions. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Austalasia and South East Asia*, pages 111-118. ACM Press, 2003.

[15] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343-352 ACM Press, 2000.

[16] Mark Segal and Kurt Akeley. The OpenGL® Graphics System: A Specification (Version 1.5), 2003.
`http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf`

[17] Richard Szeliski and David Tonneson. Surface Modeling with Oriented Particle Systems. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 185-194. ACM Press, 1992.

[18] Demetri Terzopoulos, Hong Qin. Dynamic NURBS with Geometric Constraints for Interactive Sculpting. In *ACM Transactions on Graphics*, v.13 n.2 pages103-136. ACM Press, 1994.

[19] Greg Turk. Re-Tiling Polygonal Surfaces. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 55-64 ACM Press, 1992.

[20] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, Wolfgang Stra_er. The Randomized Z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 361-170. ACM Press, 2001.

[21] Benjamin Watson, Alinda Friedman, Aaron McGaffey. Using Naming Time to Evaluate Quality Predictors for Model Simplification. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 113-120. ACM Press, 2000.

[22] Benjamin Watson, Alinda Friedman, Aaron McGaffey. Measuring and Predicting Visual Fidelity. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 213-220. ACM Press, 2001.

[23] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, Markus Gross. Surface Splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371-378. ACM Press, 2001.