# RAY-DISC INTERSECTION BASED RAY TRACING FOR POINT CLOUDS

by

Ozan Yücel

BS. in Electronics and Communication Engineering, Yıldız Technical University, 2005

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Systems and Control Engineering
Boğaziçi University

2009

# ACKNOWLEDGEMENTS

I would like to express my deep-felt gratitude to my advisor, Dr. Ali Vahit Şahiner for his advice, encouragement, enduring patience and constant support. I enjoyed every piece of this project thanks to his motivation and guidance. As a result my researches and developments became smooth and rewarding for me.

I am also grateful for the existence and love of my family which supported me during my studies.

# ABSTRACT

## RAY-DISC INTERSECTION BASED RAY TRACING FOR POINT CLOUDS

This thesis presents a method for direct ray tracing of point sampled surfaces. This allows to render high quality images using unstructured point-sampled data. An oriented disc is placed at each point and rays are tested for intersection with these discs. In case of intersection, all points within a fixed distance of the ray are used to interpolate the position, normal and any other attributes.

Ray-Disc intersection tests are accelerated using a hierarchical data structure namely an octree. This provides a computation complexity of O(n log n). Also searching for intersected octree nodes around ray is much more faster than scanning whole point data. All of the intersected discs can be located inside the nodes which are intersected by the ray.

For data without point normals, required normals are calculated from point data by fitting a tangent plane to each point.

An improvement is proposed for adapting disc radius to the local point density by determining k nearest neigbours. Using variable disc radius fills empty holes within the point cloud without loss of detail. Variable disc radius approach is also used in normal estimation process where the k nearest neighbours are used in calculations instead of neighbours within a fixed radius.

# ÖZET

## NOKTA KÜMELERİ İÇİN IŞIN-DİSK KESİŞİMİ TABANLI IŞIN İZLEME

Bu tez noktalardan oluşmuş geometrileri doğrudan ışın izlemeyle görselleştirmeyi sağlayan bir metod sunmaktadır. Böylece düzensiz nokta kümelerinden yüksek kaliteli görüntüler elde edilir. Bu yöntemde her noktaya bir disk yerleştirilir ve ışınlar bu disklerle kesişim testine sokulur. Kesişim olması durumunda ışından belirli bir uzaklıktaki noktalar da kullanılarak konum, normal gibi özellikler için interpolasyon yapılır.

Işın-Disk kesişim testleri octree gibi hiyerarşik bir data yapısı kullanılarak hızlandırıldı. Octree kullanmak hesaplamaları O(n log n) karmaşıklığına indirger. Ayrıca ışın çevresindeki kesişime uğramış octree hücrelerini bulmak, bütün nokta kümesini taramaktan çok daha hızlıdır. Bütün kesişime uğramış diskler bu octree hücrelerinin içinde bulunabilir.

Normalleri olmayan noktalar içinse, her bir noktaya tanjant düzlemi koyarak ihtiyaç duyulan normaller hesaplandı.

Bu yöntemi geliştirmek için disk yarıçapları en yakın k kadar komşu noktayı kullanarak değiştirildi ve lokal nokta yoğunluğuna uyumlu hale getirildi. Değişken disk yarıçapı kullanmak nokta kümesindeki boşlukları detay seviyesinde kayıp olmadan doldurdu. Değişken disk yarıçapı yaklaşımı normal hesaplamasına da uyarlandı. Hesaplamalarda sabit bir komşuluk yarıçapı içindekiler yerine en yakın k kadar komşu nokta kullanıldı.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1. INTRODUCTION

The thesis presents a direct rendering method for point sampled geometry known as point clouds using a ray tracing based technique. Point clouds represent the underlying surface with point data consisting of (x,y,z) coordinates and other attributes such as normal, color and material information for each point.

The most common sources for collecting point cloud data are 3D scanners and laser range finders[3]. Collected data is used for many purposes, medical imaging, creating 3D CAD models for manufactured parts and various inspection, visualization, animation applications[4].

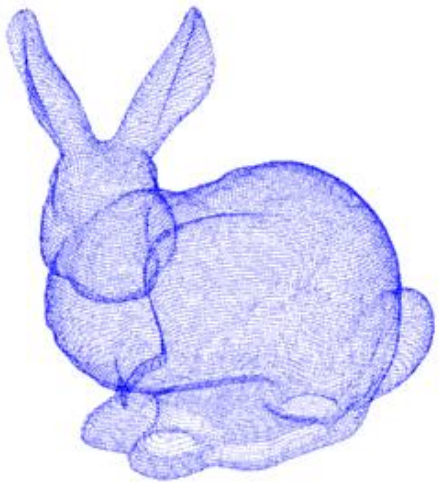


Figure 1.1. Point cloud data for Stanford Bunny

To use standard techniques for rendering, surface reconstruction of the point cloud is required using a method like Marching Cubes[1].

Points are presented by disc shaped primitives and ray tracing is carried out via disc-ray intersections. Also a normal estimation method based on plane fitting is proposed for the discs.

Direct rendering techniques allow rendering to be performed without the explicit reconstruction of the underlying surface. In the approach employed in the thesis, two improvements are proposed for generating the discs and estimating the disc normals. Instead of using fixed size discs, we have employed a nearest neighbourhood set approach where disc radius is variable. Again a nearest neighbour approach is used in normal estimation while choosing the points to which the plane is to be fitted. Experimantations on face data captured with 3D scanners show that we can obtain improved rendering results especially in areas where the data is sparse.

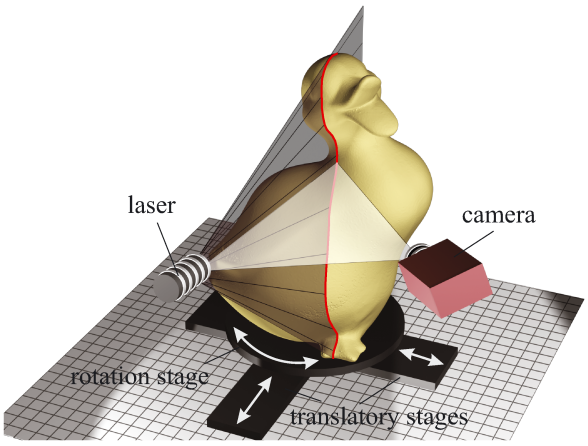



Figure 1.2. Point cloud generation using a laser stripe

Experiments have been carried out on Stanford University's bunny, dragon and budha point cloud data sets [5] and face data gathered from 3D scanners at Bogazici University's Department of Computer Science[6].

This thesis is divided in to 6 chapters starting with introduction in chapter 1.

Chapter 2 introduces a literature survey for rendering methods of point sampled geometry. Several methods of surface reconstruction and direct rendering are presented to state the differences and advantages between them. Also ray tracing fundamentals are explained.

In chapter 3, Schaufler and Jensen's Method which is used in our application is explained. We describe how the intersections are calculated by using discs and interpolation of the several intersection result for the same ray.

Normal estimation is also examined in this chapter to use in the absence of the point normals. We represent several methods to estimate normals and also get in to the detail of least square minimization method which we have used in out experiments.

In addition acceleration methods for point based ray tracing is introduced such as spatial data structures. We explain how to use octree and insert points into it. Also searching for an intersection in octree is described by using top to bottom traversal method.

In chapter 4, experiments and results with different ray tracing and normal estimation parameters are provided. This experiments shows the effect of disc, cylinder radius values used in ray tracing method and normal estimation radius on rendered results. Rendering outputs of different point clouds and variability effects are shown in this chapter.

Final chapter provides evaluation of the results and conclusions. Also the plans for the future work is presented.

# 2. RENDERING METHODS FOR POINT SAMPLED GEOMETRY

## 2.1. Surface Reconstruction

These methods are generally based on reconstructing a continuous surface which consists of polygons. This methods require extra overhead before the actual rendering. On the other hand rasteration is widely preferred after surface reconstruction which provides fast results because of using hardware accelerated gpu libraries.

Triangles or polygons may be created from points by using various methods such as Marching Cubes[1]. The Marching Cubes algorithm takes eight neighbor locations at a time forming an imaginary cube, then determines the polygons needed to represent the part of the iso-surface that passes through this cube. The individual polygons are then attached into the desired surface.

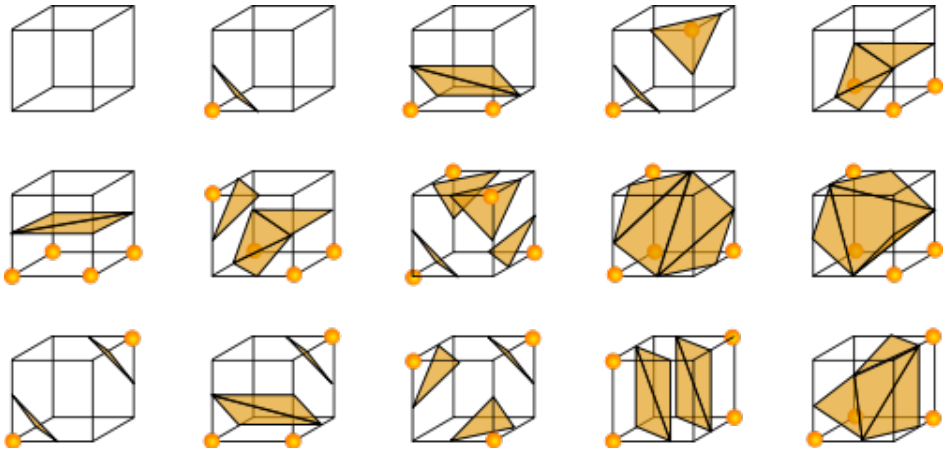


Figure 2.1. Marching cubes triangulation for different point configurations[1]

The basic principle behind the marching cubes algorithm is to subdivide space into a series of small cubes. The algorithm then instructs us to 'march' through each of the cubes testing the corner points and replacing the cube with an appropriate set of polygons. The sum total of all polygons generated will be a surface that approximates the one the data set describes.

Another surface reconstruction method is greedy triangulation which is suitable for triangulation of simple polygons. The objective is to minimize the total edge length in the triangulation. This is achieved by an iterative process that selects the shortest available internal diagonal at each stage. Each of these edges must be tested for intersection with the other edges in the triangulation. In the worst case, all diagonals will be considered.

Radial Sweep is a more advanced technique where central point in set of given points is connected to other points radially. In the next step triangles are formed as radial edges are connected together. Then convex hull of points is constructed and points from their angles are arranged.

Some of these triangles may have not good connections. For optimization of these triangles, their neighboring triangles that have a common edge are found. Then a tetrahedral is formed from each two triangles its diagonals are computed. Finally large diagonal is replaced with the short one. This procedure is repeated until no edge changes.

One of the well-known surface reconstruction methods is Delaunay triangulation[7]. The algorithm begins by generating a triangle that contains all the points from which the triangles can be obtained in order to guarantee that all the points will lie within the triangulation. The points are inserted within the triangulation one at a time; each inserted point implies making a series of changes in edges shared by two triangles, until all the triangles that contain the inserted point have been updated.

## 2.2. Direct Rendering

It is not necessary to construct a surface from point clouds. Points can be rendered by shooting rays and intersecting them with local mathematical definitions which are calculated on the run. Although this can be slower than the triangle based rendering methods, passing surface reconstruction process and directly rendering the scene provides shorter process times for complex geometries.

### 2.2.1. Ray Tracing

Ray tracing is computer based image generation method which traces path of light from eyes to the objects creating 2D pictures of 3D world. In ray tracing, a ray of light is traced in a backwards direction. That is, we start from the eye or camera and trace the ray through a pixel in the image plane into the scene and determine what it hits. The pixel is then set to the color values returned by the ray.

In real world rays are generated by light sources. Hence the direction of rays are from light sources towards the intersected objects and eye. In ray tracing it is not practical to trace millions of rays generated by light sources. Most of the rays will not hit the eye and contribute to the image of the scene. Intersection computation for these rays increases amount of time to produce an image. Instead if the rays are shot from eyes towards the scene the number of rays will decrease dramatically to a limit which is defined by the number of pixels in the image.

This requirement also gives us the definition of ray generation. A ray is defined by $R(t) = R_0 + t * R_d$ which means an origin point and a direction vector is needed. We use the eye coordinates as the origin of the ray and the pixel coordinates as the direction point. Direction vector is calculated by $P_{direction} - P_{origin}$. Then the vector is normalized and used in the ray equation.

When we want to find the color of a pixel associated with a light ray, we need to find all the different light sources contributed to it. To aid this discussion we can

divide light rays into four classes:

- Pixel rays or eye rays : Carries light directly to the eye through a pixel on the screen.
- Illumination rays or shadow rays : Carries light from a light source to an object surface.
- Reflection rays : Carries light reflected by an object.
- Transparency rays : Carries light passing through an object.

The fundamental idea is to find out what light is arriving at a particular point on a surface and then proceed onward to the eye. This eye ray carries the illumination information of the surface point. The illumination of the point is a cumulative effect of the illumination, reflection and transparency rays. These rays are generated by eye ray - object intersection using the surface and ray definitions.

Calculating illumination starts with an ray object intersection. The ray is the eye ray which is generated from eye towards the pixel. The general idea behind ray-object intersections is to put the mathematical equation for the ray into the equation for the object and determine if there is a real solution. If there is a real solution then there is an intersection and we must return the closest point of intersection and the normal at the intersection point.

Before tracing illumination, reflection or transparency rays a shadow ray is generated from intersection point towards the light sources to detect if there is a clear path between them. If the there are no lights intersected by the shadow rays the intersection point is not illuminated and the pixel is colored in black. Otherwise if the point is not in shadow, further calculations are performed using the intersected object's material properties such as diffusion, reflection and transparency coefficients.

If the object is reflective a reflected ray is generated which is tested against all of the objects in the scene. If the reflected ray hits an object then a local illumination model is applied at the point of intersection and the result is carried back to the first

intersection point. If the intersected object is transparent, then a transmitted ray is generated and tested against all the objects in the scene. As with the reflected ray, if the transmitted ray hits an object then a local illumination model is applied at the point of intersection and the result is carried back to the first intersection point. The reflected rays can generate other reflected rays that can generate other reflected rays, etc. Recursion continues until a predefined depth level is reached. This is called recursive ray tracing.

### 2.2.2. Point Based Ray Tracing

Fundamentally ray tracing a point cloud is no different from ray tracing mathematically defined primitives such as spheres, planes or triangles[8]. Shooting ray and calculating intersection is the main process of point based ray tracing[9]. Intersection calculation provides information of surface and reflection normals which can be used in Phong Illumination Model. Also the color and material properties of the points are used in the Phong equation.

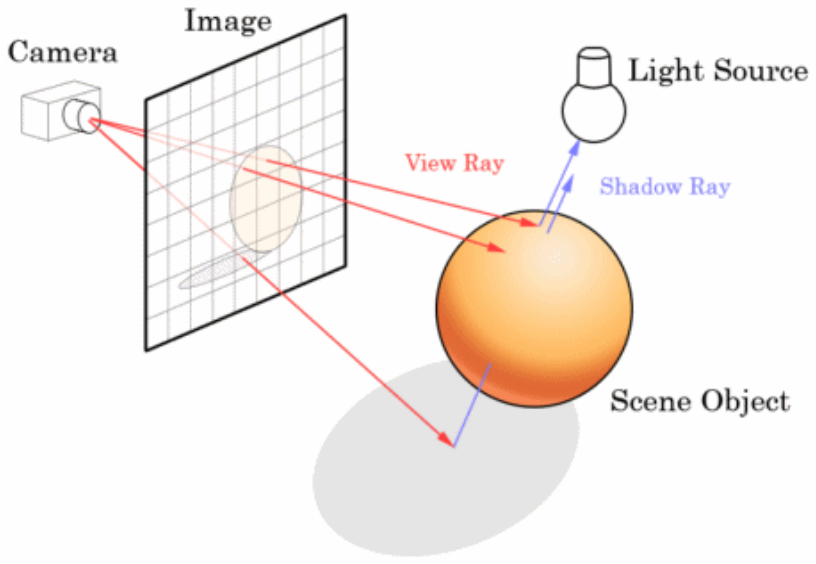



Figure 2.2. Ray Tracing[2]

Ray tracing starts with a scene definition. Scene consists of 2D output screen, eye, objects and lights. Shooting ray from the eye to each pixel in the screen and tracing this ray simulates a light ray in the opposite direction. If this ray intersects an object

and reflects to a light source then there is illumination for that pixel. Illumination depends on surface normal and material properties of intersected object, light sources and intersecting ray angle to the surface. All these parameters are represented in Phong equation.

$$I_p = k_a i_a + \sum_{lights} (k_d(L.N)i_d + k_s(R.V)^{\alpha} i_s) \tag{2.1}$$

- $k_s$: specular reflection constant, the ratio of reflection of the specular term of incoming light.
- $k_d$: diffuse reflection constant, the ratio of reflection of the diffuse term of incoming light.
- $k_a$: ambient reflection constant, the ratio of reflection of the ambient term present in all points in the scene rendered.
- $\alpha$: is a shininess constant for this material, which is larger for surfaces that are smoother and more mirror-like. When this constant is large the specular highlight is small.
- L: The direction vector from the point on the surface toward each light source.
- N: The normal at this point on the surface.
- R: The direction that a perfectly reflected ray of light would take from this point on the surface.
- V: The direction pointing towards the viewer.

In our application a disc is inserted to the scene for each point which has the same center and normal coordinates with the related point. Rays can be intersected with several discs along the way. The normals and intersection points of this different intersections are interpolated in order to provide the actual information.

Computing an intersection with the point geometry is split into two parts: detecting if an intersection has occurred and computing the actual point and normal of

the intersection.

2.2.2.1. Adamson and Alexa's Ray Tracing Method For Point Clouds. The $C^{\infty}$ surface definition of this method[10][11] is consistent and it can be shown to define smooth manifold surfaces from point clouds if certain natural sampling criteria are fulfilled. Due to these properties and its ease of implementation, several researchers have used this definition for ray-tracing point models. Complex mathematical operations make this method harder to implement and slower compared to other on the run methods.

Let's say point samples of surface S are $P = \{p_i \epsilon R^3\}, i\epsilon\{1, ..., N\}$, the neighborhood of P is the union of a set of balls centered at the points $p_i$:

$$\mathbf{B} = \bigcup_i B_i, B_i = \{x | \|x - p_i\| < r_B, x\epsilon R^3\} \tag{2.2}$$

Both surface S as well as the approximation function $\hat{\mathrm{S}}$ are contained by B neighborhood. For the approximation function $\hat{\mathrm{S}}$ two more functions will be defined on the neighborhood: the weighted average and the normal direction. The weighted average function is defined by $a : B \rightarrow B$ which maps each point x onto the weighted average of the contributing points. The normal direction function is defined by $n : B \rightarrow R^3$ assigning a normal to each point in the neighborhood of the points. With these information the approximating surface $\hat{\mathrm{S}}$ can be defined as

$$\hat{S} = \{x\epsilon B | f(x) = n(x)^T (x - a(x)) = 0\} \tag{2.3}$$

Assuming the points $p_i$ is assigned with the normals $n_i$ and h as the support

radius of the points, the weighted average function a and the normal direction function n is:

$$a(x) = \frac{\sum_{i=1}^{N} \Theta(\|x - p_i\|/h) p_i}{\sum_{i=1}^{N} \Theta(\|x - p_i\|/h)}, \; n(x) = \frac{\sum_{i=1}^{N} \Theta(\|x - p_i\|/h) n_i}{\| \sum_{i=1}^{N} \Theta(\|x - p_i\|/h) n_i \|} \tag{2.4}$$

If the points do not have surface normals, they can be computed by using the methods we discussed or a different mapping n based on the weighted covariance directions in x can be defined.

Smooth and positive weight functions should be chosen which have negative first derivatives. To avoid discontinuities the points beyond radius of support h should be ignored by making $\Theta$ return zero at distance h. Otherwise, both a(x) and n(x) will be effected immediately by the points entering the radius of support. For more smoothness, at least a zero first derivative at distance h should exist. A truncated Gaussian function can be used with a scaling parameter $\sigma$ to ensure that the weights go to zero when approaching the radius of support.

2.2.2.2. Splat Based Ray Tracing. Splats can be used as surface representation and rays can be intersected with the ellipses[12]. The resulting surface is $C^{-1}$ continuous and linear, which causes severe artifacts especially under complex lighting situations. This method differs from Schaufler and Jensen's [13] by the lack of interpolation for intersections. This is the reason why the surface is not smooth.

2.2.2.3. Splat Based Rasterization. The first step in splat rendering is splat rasterization by determining which image pixels are covered by the projected splats[14]. A splat is created for each point in the set. Since there is no native support for splat primitives in current graphics libraries, splats have to be represented by other drawing primitives like points or triangles. A special elliptical alpha-texture is mapped on to

that triangle. Then all fragments outside the splat are assigned a zero alpha value. Finally splat is rendered using one of the Flat, Gouraud or Phong Shading methods.

# 3. IMPLEMENTATION

## 3.1. Disc Intersection Based Ray Tracing Method For Point Clouds

We have used a disc intersection based ray tracing technique[13] that uses only a local sampling of the point sampled geometry skipping the time consuming surface reconstruction step. This method intersects a cylinder around the ray with the discs and then computes the intersection as a weighted average of discs whose centers are inside the cylinder[13]. Although this approach produces high-quality images, the resulting geometry depends on the particular rays used for intersecting the surface. Therefore, this ray-surface intersection algorithm does not define a consistent surface.

### 3.1.1. Ray Disc Intersection

Disc is the only primitive we used in our intersection tests. Rays are checked for intersection with the discs inserted in each point. Ray - disc intersection is basically same as ray - plane intersection with a difference of disc radius limitation. Ray should be intersected with the disc plane and the intersection point should reside inside the disc radius.

For ray - plane intersection we need geometric definitions of both ray and plane. A plane is defined by the equation: Ax + By + Cz + D = 0, or the vector [A B C D]. A, B, and C, define the normal to the plane. If $A^2 + B^2 + C^2 = 1$ then the unit normal $P_n = [ABC]$. If A, B, and C define a unit normal, then the distance from the origin [0 0 0] to the plane is D.

A ray is defined by:

$R_0 = [X_0, Y_0, Z_0]$ $R_d = [X_d, Y_d, Z_d]$

so $R(t) = R_0 + t * R_d, t > 0$

R(t) should be inserted into the plane equation to determine if there is an intersection with the plane:

$$A(X_0 + X_d * t) + B(Y_0 + Y_d * t) + (Z_0 + Z_d * t) + D = 0$$

which leads to:

$$t = -(A * X_0 + B * Y_0 + CZ_0 + D)/(A * X_d + B * Y_d + C * Z_d)$$

$$t = -(P_n * R_0 + D)/(P_n * R_d)$$

First we need to compute $P_n R_d = V_d$. If $V_d = 0$ then the ray is parallel to the plane and there is no intersection. If $V_d > 0$ then the normal of the plane is pointing away from the ray. We use one-sided planar discs and stop if $V_d > 0$.

Now second dot product $V_0 = -(P_n * R_0 + D)$ and $t = V_0/V_d$ should be computed. If t ¡ 0 then the ray intersects plane behind origin else intersection point is calculated as:

$$P_i = [X_i Y_i Z_i] = [X_0 + X_d * tY_0 + Y_d * tZ_0 + Z_d * t]$$

Now we usually want surface normal for the surface facing the ray, so if $V_d > 0$ then reverse sign of ray.

After finding the intersection point of the plane we should check if the point is inside the disc radius. A disc is defined by plane equation, center $(X_c Y_c Z_c)$ and a radius r.

If magnitude of $[X_i Y_i Z_i] - [X_c Y_c Z_c]$ is smaller than r the point is inside the disc radius and the disc is intersected by the ray.

### 3.1.2. Intersection Detection

The ray is surrounded by a cylinder and only those nodes in the octree that intersect the cylinder are traversed[13]. Intersection test continues with the discs in the intersected nodes. The discs residing in the cylinder are added to list for interpolation operations.

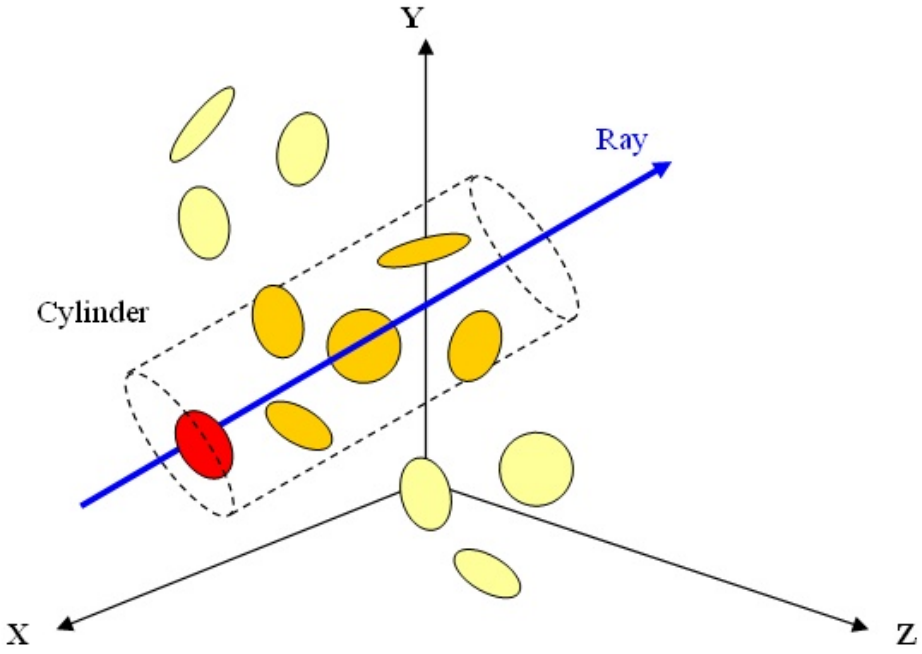



Figure 3.1. Intersection Detection

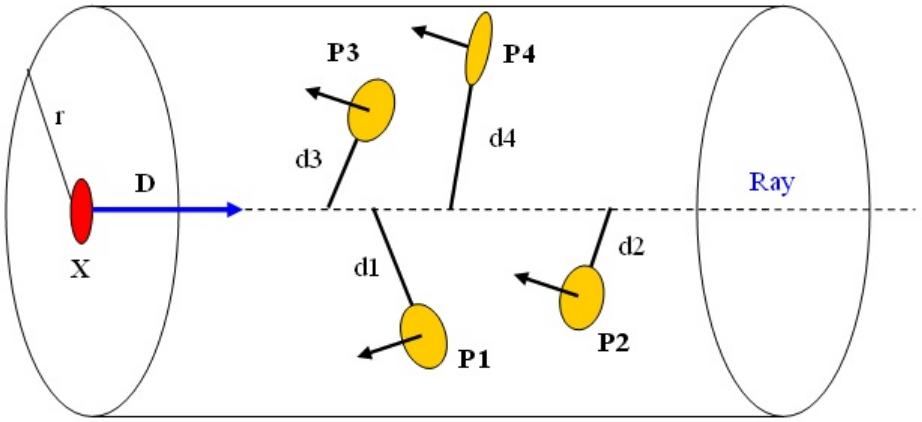



Figure 3.2. Interpolation

### 3.1.3. Interpolation

The actual intersection point, normal and other attributes will be interpolated from the points within the cylinder[11][15]. In particular, the normal of the intersection point is formed by weighting the normals of the points in the cylinder based on the point's distance from the ray. The same weighting is applied to find the actual intersection point along the ray.

We intersect the planes formed by each point and its normal with the ray's parametric representation, ray = X + t . D, where X and D are the origin and the direction of the ray.

The final intersection point is found by interpolating the computed intersection location and normal values. In general, any attribute associated with the points can be interpolated this way.

$$attrib = \frac{\sum_i attrib_i * (r - d_i)}{\sum_i (r - d_i)} \tag{3.1}$$

Points will only be considered if their distance from the ray $d_i$ is smaller than r. Note that the distance is not measured orthogonal to the ray but in the plane of the point and its normal to create the circular disks around each point.

This intersection point computation is slightly view dependent. The final position of the surface will vary with the direction of the incoming ray.

```
for each pixel in image do
  Create ray from eyepoint passing through this pixel
  Initialize NearestT to INFINITY
  Find intersected nodes by ray
  for every disc in the intersected nodes do
    if ray intersects this disc then
      if t of intersection is less than NearestT then
        Set NearestT to t of the intersection
      end if
    end if
  end for
  if there is no hit then
    Fill this pixel with background color
  else
    Create cylinder starting from the intersection point
    for every disc in the intersected nodes do
      if disc is inside the cylinder then
        add disc to interpolation equation
      end if
    end for
    Interpolate points for the final point and normal
    Use Interpolated Intersection Data to compute shading function
    Fill this pixel with color result of shading function
  end if
end for
```

Figure 3.3. Pseudocode for Ray Tracing Algorithm

## 3.2. Normal Estimation

Some data collection hardware do not provide point normals as part of the data. In such cases they can be estimated using reliable normal estimation methods. There are several approaches that employ the techniques suc as using surface reconstruction, numerical optimizations and Voronoi diagrams for this purpose. Vertex calculations provide reliable normal information but requires surface reconstruction which is costly. Variations of numerical approaches work well even when point clouds are contaminated with noise. A variation of the Voronoi based method is also reported for noisy point clouds[16]. Usually it is the application requirements that determine the type of the approach to be employed.

### 3.2.1. Normal Estimation with Surface Reconstruction

Surface reconstruction methods supplies polygon information that can be used to calculate normals[7]. These polygons are defined by faces containing vertices which are arranged in an anti-clockwise order around the face. In order to calculate the normal 3 non colinear vertices are needed. For quads or any other irregular shaped faces we just need to choose any 3 vertices. The result will be the same because all points are on the same plane.

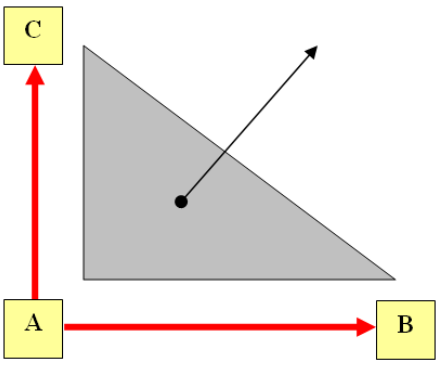



Figure 3.4. Normal calculation for a triangle

First we need to calculate the vectors between vertices. Considering 3 vertices A, B and C, required vectors can be calculated easily as:

$$Vector1 = VertexB - VertexA \tag{3.2}$$

$$Vector2 = VertexC - VertexA \tag{3.3}$$

To calculate the normal we need to calculate the cross product of these vectors. This will give us a vector that is positioned perpendicular to the respective plane. We need to normalize this vector to get the final normal vector.

Since it requires surface reconstruction in overall this method is computationaly costly.

### 3.2.2. Plane Fitting methods

These methods are used to calculate normals without explicitly reconstructing a surface[16]. A widely used approach is to find a proper set of points in the local neighborhood for a point p and then compute a plane that best fits to these chosen points. The normal of the plane is taken as the estimated normal at p. This basic plane fitting method has been made more effective with improvements.

Hoppe [17] presented an algorithm where the normal at each point is estimated as the normal to the fitting plane obtained by applying the total least square method to the k nearest neighbors of the point in the point cloud. Specifically for a point p and its k nearest neighbors $\{p_i\}_{i=1}^{k}$, they find the fitting plane $n^T x = c$ for p by minimizing the error term $e(n, c) = \sum_{i=1}^{k}(n^T p_i - c)^2$ under the constraint $n^T n = 1$. It should be noticed that the normals computed by fitting planes are unoriented.

This method can be improved in two different ways. Fitting plane for a point p should respect the nearby points more than the distant points in the point cloud. Hence the neighboring points are assigned different weights based on their distances to p. The smaller the distance of a sample from p, the bigger the weight it has. In other

words, the error term can be defined as $e(n,c) = \sum_{i=1}^{k}(n^T p_i - c)^2\theta(\|p_i - p\|)$ , where $\theta()$ is a weighting function. In their implementation, the weighting function is taken as Gaussian, i.e., $\theta(\|p_i - p\|) = e^{-(\frac{\|p_i - p\|}{h^2})}$, where $h^2$ is chosen to be one third the square distance between p and its k-th nearest neighbor. This method is called as weighted plane fitting method.

Regarding to Mitra, Nguygen and Guibas, [18] a proper selection of the value of k is important to obtain a good normal estimation[17]. Using the same value of k at all points could give biased fitting especially at places where samples are arbitrarily dense. Hence, instead of using k nearest neighbors of the point, they consider the samples within a ball of certain radius r. Under the assumption that the noise has zero mean and standard deviation $\sigma_n$, they could get a bound on the angle between the estimated normal and the true normal with a probability almost one. An optimal radius r can be obtained by minimizing this bound, which has the following expression in three dimensional case provided the probability is $1 - \epsilon$:

$$r = (\frac{1}{\kappa}(c_1\frac{\sigma_n}{\sqrt{\epsilon\rho}} + c_2\sigma_n^2))^{\frac{1}{3}} \tag{3.4}$$

where r is the local sampling density, k is the local curvature and $c_1$ and $c_2$ are some constants. The actual algorithm takes $\sigma_n$ as user input and evaluates r in an iterative manner. Initially r and k are evaluated based on the k (= 15) nearest neighbors and then the radius r is obtained from the equation. Once one gets the neighborhood size $r$, $\rho$ and $\kappa$ are reevaluate based on the samples within this neighborhood to get a better estimation of $r$, $\rho$ and $\kappa$. They claim that three iterations in general are enough to obtain good estimations for all the quantities. The above method is called as the adaptive plane fitting method.

3.2.2.1. Normal Estimation with Least Square Minimization. Let's say n is the normal which will be calculated in q location. Surface around q is defined by the points around q. These points should be as close as possible to the tangent in q. Then a tangent plane around q can be handled as a least squares problem. Assume a plane $H(x) : n^T q = n^T p_i, \|n\| = 1$ is searched passing through q that minimizes the squares $(n^T(q - p_i))^2$.

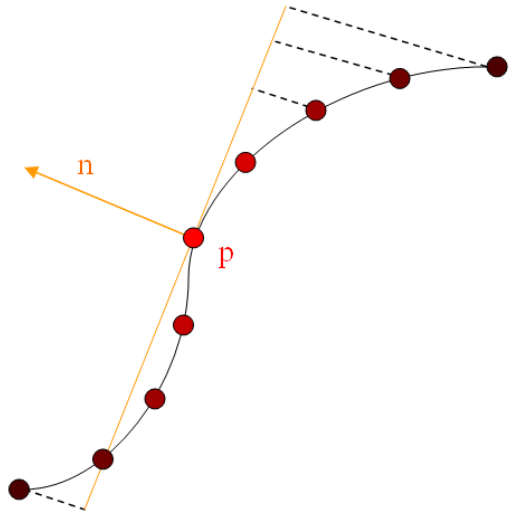



Figure 3.5. Estimating normal close to point q

However, only points $p_i$ closer to q should be taken into account. These points can be found by looking k-nearest neighbors of q or using a locally supported weight function $\theta$ to weight closer points. A hat function can simulate k-nearest neighbor approach with suitable radius for $\theta$. A locally weighting part should be added to the function for final calculation. When all combined, n is defined by the following minimization formula:

$$min_{\|n\|=1} \sum_i (n^T(p_i - q))^2 \theta(\|p_i - q\|). \tag{3.5}$$

Quadratic constraint $\|n\| = 1$ makes this a nonlinear optimization problem. To

solve this equation, he outer product matrices $(p_i - q)(p_i - q)^T$ will be used and the function will be minimized as;

$$m(n) = n^T(\sum_i (p_i - q)(p_i - q)^T \theta(\|p_i - q\|))n. \tag{3.6}$$

Following eigenvalue/eigenvector decomposition of the sum of outer products will be the next step:

$$\sum_i (p_i - q)(p_i - q)^T \theta = E(\lambda_0, \lambda_1, ...)E^T. \tag{3.7}$$

This decomposition shows that in transformed coordinates $E^T n$ the equation $m(n) = n^T E(\lambda_0, \lambda_1, ...)E^T n$ has only pure quadratic terms. Also each of these quadratic terms has an eigenvalue as coefficient which are $\lambda_0 <= \lambda_1, ....$ The minimum of $m(n)$ can be found among all unit-length vectors for $E^T n = (1, 0, 0, ...)^T$. This gives the result of $n = e_0$ which is the eigenvector of the smallest eigenvalue.

Another important point is this estimation will only give a normal direction, not an orientation. The correct orientation should be derived from inside-outside information generated using scanning of the object. In our application we assume that all normal z values must be in positive direction because of scanning in one direction. If a normal has a negative z value it is reversed for correct orientation[7][16].

3.2.2.2. Weight Function. $\theta(\delta) = \delta^{-u}, u \in \mathbb{N}$ or the Gaussian $\theta(\delta) = exp(\delta^2/h^2)$ can be selected as the radial function $\theta$. This are global calculation functions making every point influencing each other.

Locally supported radial functions should be used for reaching local solutions. Let's say $\epsilon$ is a distance parameter for determine local points. If the distance between two points are farther than $\epsilon$ the function $\theta$ return 0 which ignores the influence of the other point like $\delta > \epsilon \Rightarrow \theta(\delta) = 0$. This means only the points with a distance of $\epsilon$ will be used in the normal calculation for that point.

### 3.2.3. Big Delaunay ball method

For a "noise-free" point set, Amenta [19] proposed a Voronoi based method for estimating normals[16]. For a given set of points $P \subset \mathbb{R}^3$, let Vor P and Del P denote the Voronoi diagram and its dual Delaunay triangulation of P respectively. Denote the Voronoi cell for a point p as $V_p$. Amenta and Bern [19] showed that the line through p and the furthest Voronoi vertex in $V_p$, called its pole, can approximate the normal at p up to orientation. However this property does not hold for noisy samples. Dey and Goswami [16] extended the idea of poles to the noisy samples.

A ball can be called as Delaunay if its boundary circumscribes a Delaunay tetrahedron, or equivalently has a center at a Voronoi vertex v and has a radius $\|v - p\|$ where $v \in V_p$. By definition, the Delaunay balls are maximally empty. The Delaunay balls with poles at their centers are called polar balls. The observation of Amenta and Bern [17][19] can be interpreted in terms of the polar balls as follows. If p is a sample point on the boundary of a polar ball B, the segment joining p and the center of B estimates the normal direction at p. Dey and Goswami observed that, under some reasonable noise model, certain Delaunay balls remain relatively big and can play the role of polar balls. This suggests an algorithm for estimating the normals for the noisy point cloud. Redefine the pole for a point $p \in P$ as the furthest vertex of its Voronoi cell whose dual Delaunay ball is big. Similar to the "noise-free" case, the normal line at p can be approximated by the line through p and its pole. We call this algorithm Big Delaunay Ball algorithm.

The key to the BDB algorithm is to identify the big Delaunay balls. The big Delaunay balls are identified by comparing their radii with the nearest neighbor dis-

tances of the incident samples. Specifically, for a point $p \in P$, let $\lambda_p$ denote its average nearest distances to the five nearest neighbors of p in P. We call a Delaunay ball big if its radius is larger than $c\lambda_p$ for at least one of its incident points $p \in P$ where c is an user defined parameter. A small value for c makes the algorithm sensitive to the noise since the small Delaunay balls are identified as big. On the other hand, a large value for c makes less Delaunay balls marked as big. As a result, more points have no big Delaunay ball incident on them and hence no normal can be estimated for these points.

## 3.3. Acceleration

### 3.3.1. Using Data Structures

Hierarchical data structures are important representation techniques in the domains of computer vision, image processing, computer graphics, robotics, and geographic information system. In computer graphics space partitioning data structures are popular to handle ray object intersections faster and efficiently. There are different types of data structures like BSP-trees, Kd-trees and octrees[20][21].

BSP also known as Binary Space partitioning trees recursively subdivides space into hyperplanes. This method is useful for dividing polygons into groups which can be identified as being in front or back of that separating plane. Since point clouds do not provide a defined shape or convex sets such as polygons this method is not useful for partitioning in our application.

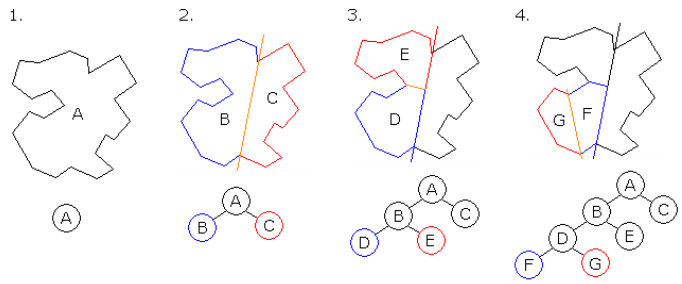



Figure 3.6. BSP Partitioning of a polygon

Kd-trees also divide space by using hyperplanes like BSP trees with only one difference being that separating planes used for partitioning are not defined by the shape. Instead they are predefined to be axis aligned for sustaining coherence among the objects. For point cloud applications every node in a k-d tree is a k-dimensional point. These points create a new node dividing the tree in one dimension. For example if for a particular x split all points which has smaller x value than the node are in the left sub-tree and points with larger x values will be in the right sub-tree. While this k-d tree provides fast searching for a particular point, we concerned distance based neighborhood instead of one point searching. This situation lead us using octrees which

also divide space with planes but not using points as node boundaries. Octree divides node into equal sub-nodes encapsulating the points. Uniform distribution of octree nodes is suitable for our algorithm which looks for points in a radius.

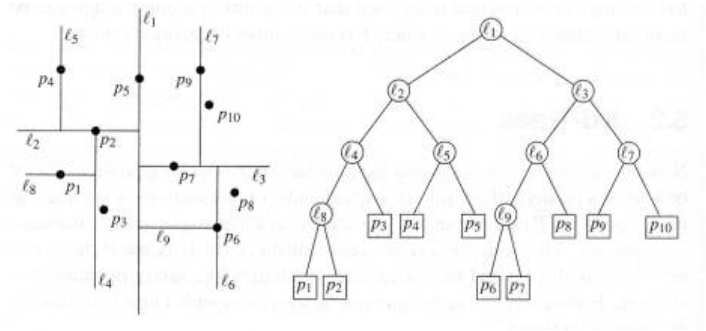



Figure 3.7. Kd-tree Decomposition

### 3.3.2. Octree

An octree is a tree data structure which partitions a three dimensional space by recursively subdividing it into eight octants. Subdividing starts with rood node whose geometric representation is a boundary cube containing all of the primitives. Nodes containing primitives called leaf nodes. In generation process if a leaf node contains more than a predefined number of primitives it is divided into new nodes. This process contains until all the leaf primitives are below the predefined threshold. Non-leaf nodes which do not contain primitives are not subdivided and not taken into account in the traversing process[21].

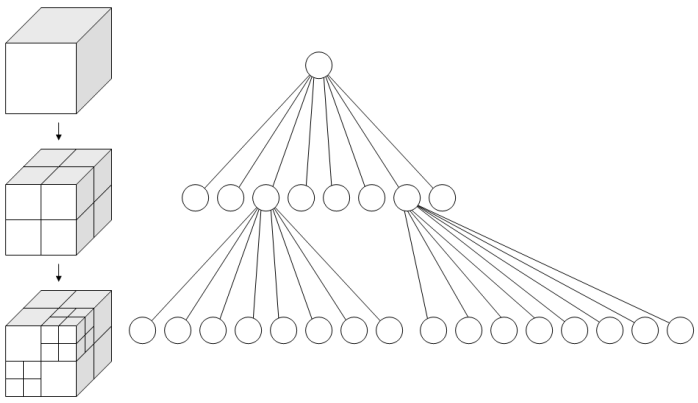


Figure 3.8. Octree Subdivision

Subdivisions can be performed in two ways; uniform and adaptive sub-division. In uniform sub-division all nodes of the octree are subdivided regardless of the number of primitives associated with a particular node. This type of division is ideal for uniformly distributed primitives such as volumetric data which is not the case in our application. In adaptive subdivision a node is subdivided if the number of primitives associated with that node exceeds a threshold value. Adaptive subdivision is ideal for non-uniform data distributions such as boundaries of an object. In ray tracing applications only the surface of the object will be used in the calculations and also the data we used like Stanford bunny and face geometries are only surface scanned point clouds. Hence the adaptive octree is the optimum choice for storing our point clouds.

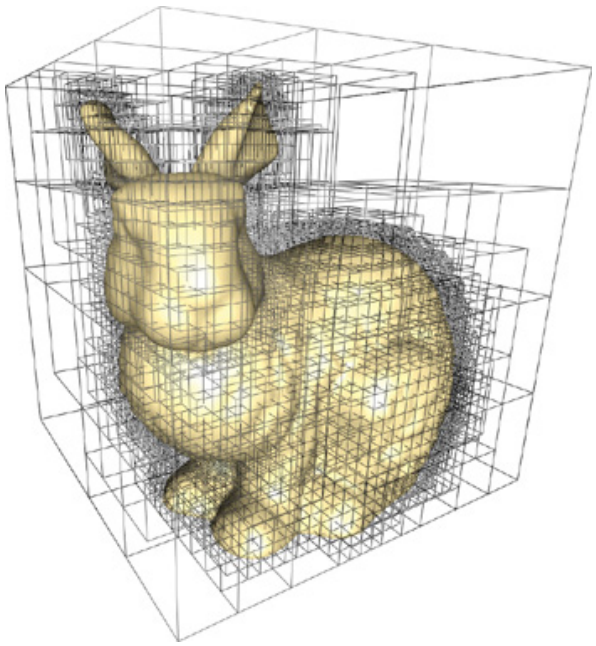


Figure 3.9. Adaptive octree subdivision for Stanford bunny

As mentioned before [21] in order to process computations of ray-splat intersections efficiently, we have choosen to use an octree for storing the point discs. Discs could also be created at the run-time for each intersected node to minimize memory usage but it would also increase the process load which is a more important resource for our application. The generation of the octree and the insertion of the discs is done in one step. Therefore a disc is inserted to scene for each point with a center of the point location and point normal. Octree generation process start with calculating the boundary volume for root node. The points with maximum and minimum (x,y,z) val-

ues specify the root boundaries. Starting with an empty octree represented by the root that describes the bounding box of the entire point cloud, we iteratively insert each disc into that leaf cell that contains the center of the disc. As soon as one leaf cell would contain more than a given small number of discs, the leaf cell gets subdivided into eight equally-sized subcells. The discs that were stored in the former leaf cell get adequately distributed among its children, which are the new leaf cells. This first phase is as simple as generating an octree for points. The iteration stops once all discs have been inserted.

The octree is used in ray tracing by checking if the ray is intersecting the boundaries of the root node. If not no further calculations will be made for that ray. When an intersection occurs child node boundaries will be used to test the intersection. This progress will continue for deeper nodes until the actual objects are found. The intersected octree nodes are found by traversal algorithms.

There are two main octree traversal algorithms which are bottom-up and top-down[21]. Bottom-up is traversing starts at first terminal node intersected by ray and using neighbor-finding to obtain the next node[22]. Top-down is starting from the root node and going down recursively to the terminal voxel. We are using the top-down method which is looking for if the rays hits a (leaf) cell of the octree. We take intersected node and its neighbours into consideration to check for intersection of the ray with the discs stored within that cells.

Table 3.1. Octree usage effect on performance

| Number of Points | Octree Usage | Number Of Intersections | Render Duration |
|---|---|---|---|
| 453 | No | 72297500 | 25sec. |
| 453 | Yes | 1357314 | 02sec. |

## 3.4. Implementation Details

Our code is written using C++ language in Borland C++ Builder 6.0 environment. Coding is organized with related classes for maintaining object oriented design. Win32 and VCL components are used in graphical user interface making the application Windows Operating System dependent.

OpenGL library is included in the project in order to display 2D output image created with the ray tracing computation. OpenGL library is also used to display a preview image of the point cloud. This is done by drawing a square in each point and rendering with GPU providing fast camera changes with user interaction. This interactions are changing viewing direction with mouse movement while holding right mouse button and moving camera in left, right, forward, backward, up, down directions with keyboard input. Ray tracing is started with this camera information modified by user interaction.

Other open source libraries are Stanford University's read - write functions for PLY file format and newmat11 for matrix calculations and eigen value decomposition used in normal estimation method.

### 3.4.1. Graphical User Interface

Graphical user interface of our application consists of one form containing OpenGL output window, main menu, progress bar and a status bar for displaying general information about the ray tracing process such as point count, disc and cylinder radius and render time.

The functions of the main menu items are;

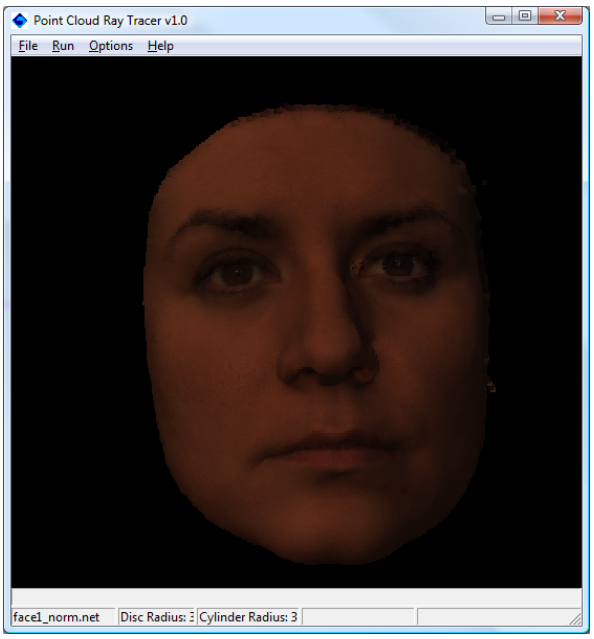
- File ⇒ Open : Open input data file for loading point clouds in to the scene. Supports .NET and .PLY file formats.
- File ⇒ Save As : Saves the generated output image of the ray tracing computa-

tion as bitmap file format.

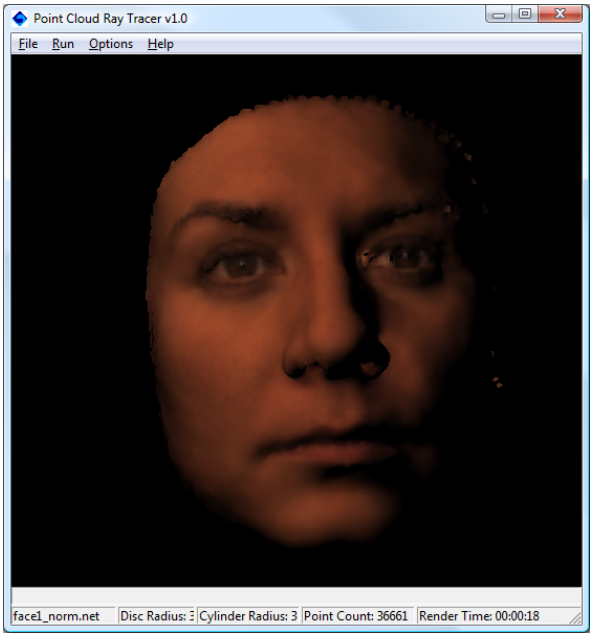
- Run ⇒ Render : Starts the ray tracing computation using the current point cloud.
- Run ⇒ Calculate Normals : Calculated the normals of the current point cloud.
- Run ⇒ Save Visible Points : Saves the current point cloud into a new file ignoring the invisible points. Used for filtering surface point clouds like face data.

- Options ⇒ Allow Preview : Shows the preview image of the current point cloud using OpenGl library if selected.

- Help ⇒ About : Opens the about window showing the application information.




(a) Point cloud preview with OpenGL




(b) Generated image after ray tracing

Figure 3.10. Graphical User Interface


(a) File Menu


(b) Run Menu


(c) Options Menu


(d) Help Menu

Figure 3.11. Menu Details

# 4. EXPERIMENTATION

We used several file formats for point cloud input such as VP, NET, PLY. PLY known as Polygon File Format or the Stanford Triangle Format is designed by Stanford University [5] to store three dimensional data from 3D scanners. Objects in PLY format are defined as a list of nominally flat polygons. A variety of properties can be stored including: color and transparency, surface normals and texture coordinates. We have supported PLY format in our application for Stanford Bunny. Polygon information is only used for normal calculation. We extracted the point coordinates for applying our method. There is no color information for our bunny model or any other attributes.

Facial point data we gathered from Bogazici University [6] are vp and net files. These file formats contain point and texture coordinates which matches with a separate bitmap file. There is no normal information or polygon indexes to calculate normals easily. So we used least square minimization method to estimate normals of the points and write this data into the same file.

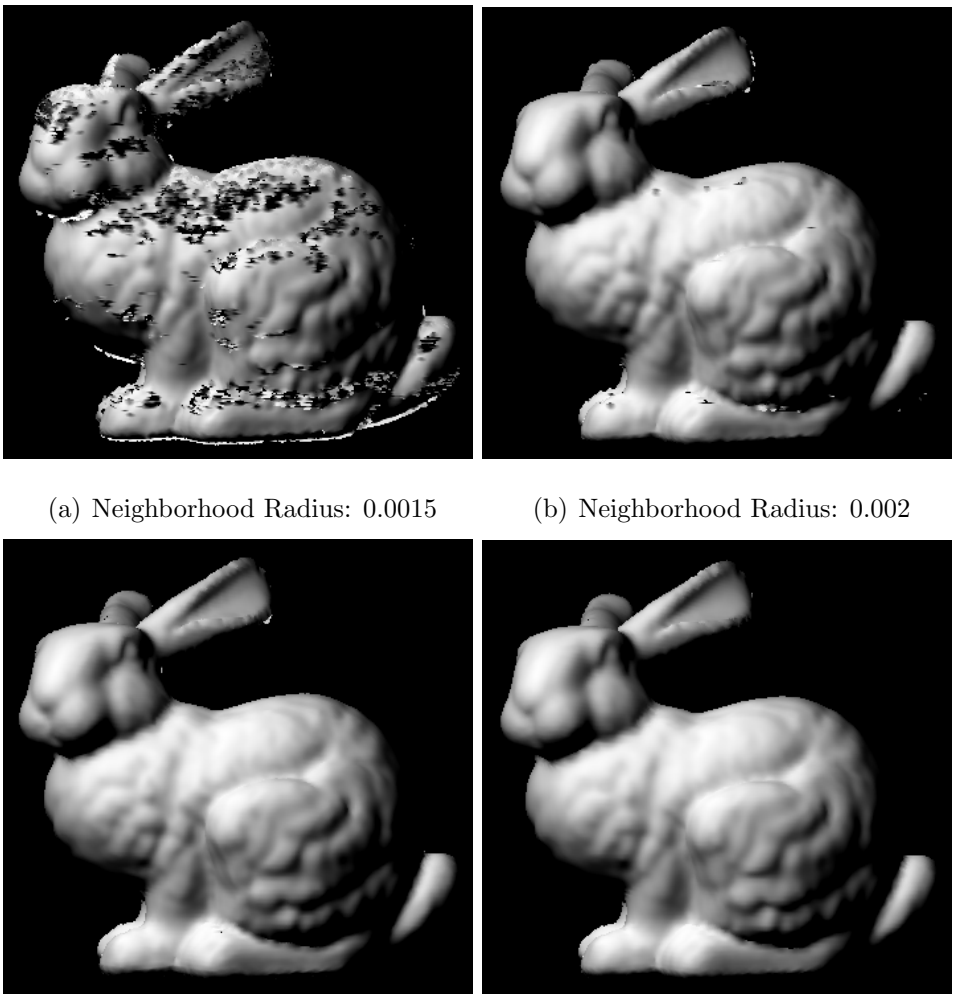
## 4.1. Normal Estimation

### 4.1.1. Normal Estimation Using Fixed Neighbourhood Radius

As mentioned before if the normals are not part of the data, they should be estimated. Out method is least square minimization which fits a plane to the points in the neighborhood of each point. This method uses local calculation by ignoring points with a distance bigger than a predefined radius. If a radius smaller than a optimum value is used error rate in normal estimation will increase because of insufficient number of point samples in the neighborhood. In contrast if a too large radius is used smoothness will increase decreasing the level of detail.

Table 4.1. Bunny Data Normal Calculation Error Rates By Using Fixed Radius

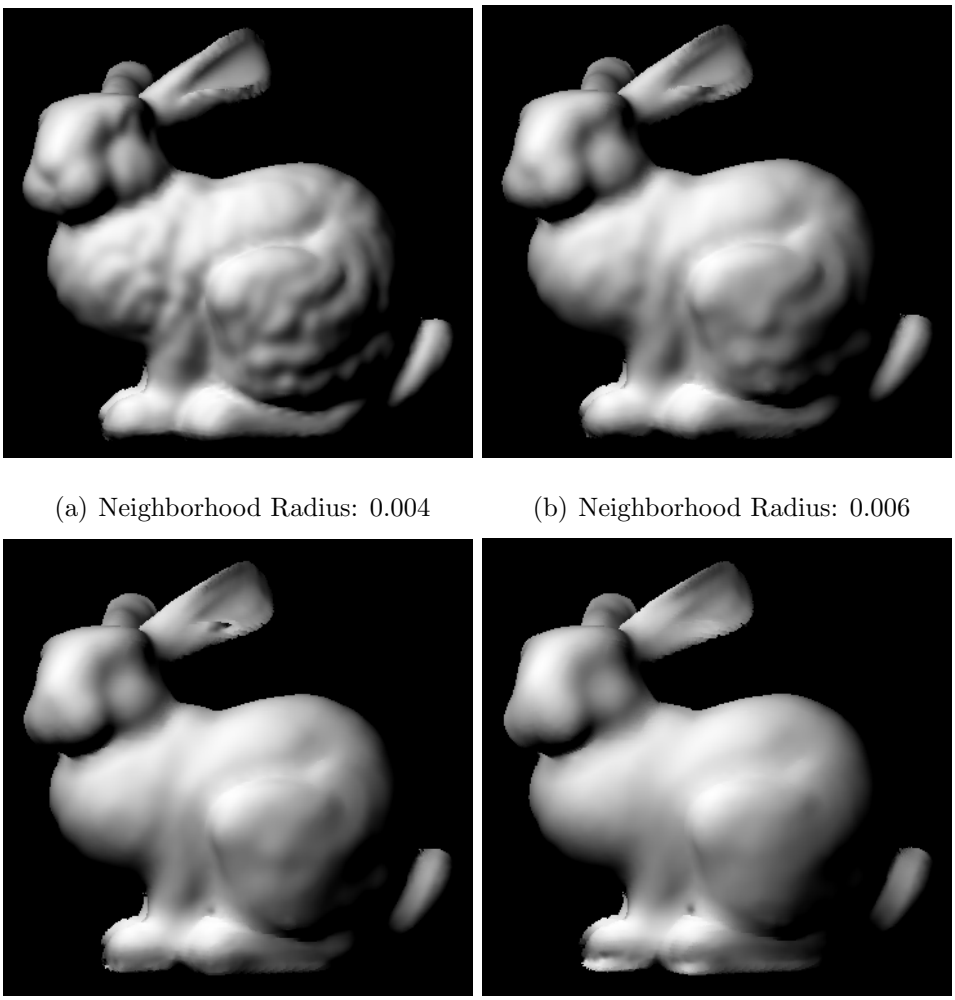
| Radius | 239 Points | 994 Points | 4261 Points | 19509 Points |
|---|---|---|---|---|
| 0,002 | 1,90749510 | 1,88196900 | 1,504502800 | 0,091319650 |
| 0,003 | 1,90749510 | 1,82637200 | 0,301746700 | 0,084886208 |
| 0,004 | 1,90749510 | 1,63725400 | 0,047003265 | 0,093005635 |
| 0,005 | 1,85897270 | 0,99347919 | 0,054752789 | 0,103418350 |
| 0,006 | 1,78151460 | 0,33011973 | 0,060911361 | 0,114707000 |
| 0,020 | 0,28632677 | 0,26563179 | 0,275710940 | 0,330883800 |



(a) Neighborhood Radius: 0.0015 (b) Neighborhood Radius: 0.002

(c) Neighborhood Radius: 0.0025 (d) Neighborhood Radius: 0.003

Figure 4.1. Estimating normals with different neighborhood radii 1



(a) Neighborhood Radius: 0.004

(b) Neighborhood Radius: 0.006

(c) Neighborhood Radius: 0.008

(d) Neighborhood Radius: 0.010

Figure 4.2. Estimating normals with different neighborhood radii 2

### 4.1.2. Normal Estimation Using Variable Neighbourhood Radius

Here we provide figure and table results of normal estimation by choosing variable neighbourhood radii which is done by getting k nearest neighbours of the points for the calculations. Mean square errors ($MSE(P) = \frac{1}{n}\sum_{i=1}^{n}(p_i - p)^2$) between original and estimated normals with different k values are shown in Tables 4.2, 4.3, 4.4. It is seen that too small or too big k values increases error rate regardless of the point cloud characterisctics and point counts. Also error rates decreases proportionally when the number of points in the point cloud increases.

Minimum error rates can be achieved by choosing k value between 5 and 9. For point clouds with higher number of points k value should be chosen larger to lower the errors. It is also seen in the Tables 4.5, 4.6, 4.7 that processing time is proportional with the k index. Comparison results show that using variable neighbourhood radius instead of a fixed provides more adaptivity among different point clouds.

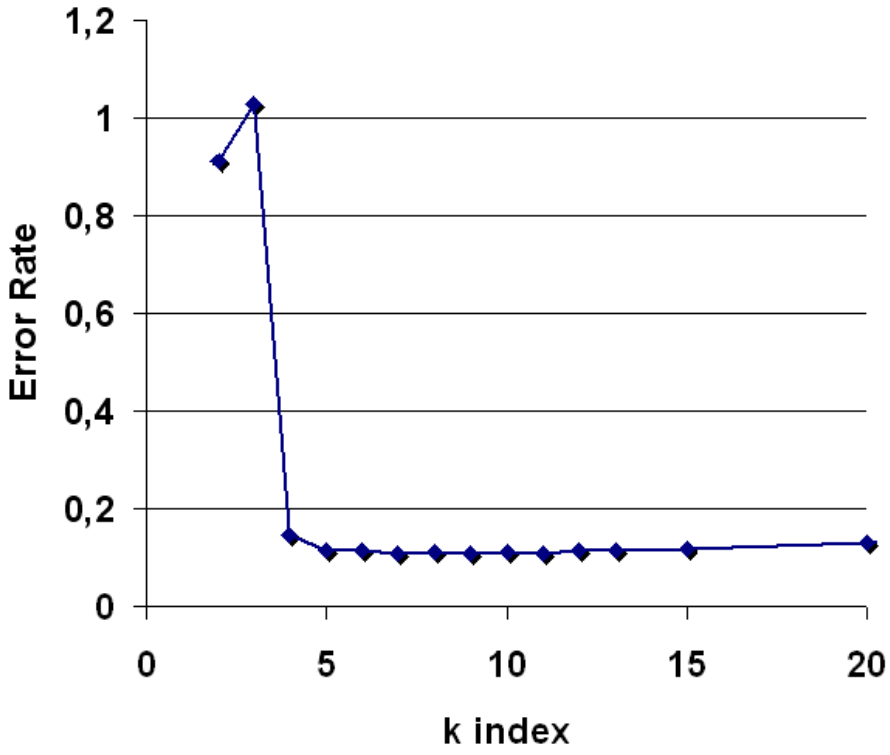



Figure 4.3. Normal Estimation Error Rate

Table 4.2. Bunny Data Normal Estimation Error Rates By Using Variable Radius

| k | 239 Points | 994 Points | 4261 Points | 19509 Points |
|---|---|---|---|---|
| 2 | 1,11608730 | 1,16747390 | 1,232299000 | 1,24051450 |
| 5 | 0,26041976 | 0,10566889 | 0,051443763 | 0,098280884 |
| 10 | 0,41923645 | 0,16671158 | 0,057377376 | 0,089128956 |
| 15 | 0,38791528 | 0,19912653 | 0,073489256 | 0,089931242 |
| 20 | 0,46224821 | 0,22952819 | 0,088552125 | 0,092159577 |
| 60 | 0,49191537 | 0,33576241 | 0,188117030 | 0,118682910 |
| 100 | 0,59222138 | 0,41123930 | 0,235927330 | 0,149603380 |
| 200 | 0,84743029 | 0,48253393 | 0,315448490 | 0,214325500 |

Table 4.3. Dragon Data Normal Estimation Error Rates By Using Variable Radius

| k | 2719 Points | 11804 Points | 51145 Points | 220421 Points |
|---|---|---|---|---|
| 2 | 1,26231120 | 1,224159400 | 1,251614200 | 1,473059700 |
| 5 | 0,15590234 | 0,078749813 | 0,035893843 | 0,055084959 |
| 10 | 0,18369356 | 0,084152423 | 0,037773214 | 0,020904943 |
| 15 | 0,23707913 | 0,100514720 | 0,046740592 | 0,024500033 |
| 20 | 0,29008231 | 0,122099710 | 0,053451680 | 0,028916527 |
| 60 | 0,51106435 | 0,212383360 | 0,103276220 | 0,055007156 |
| 100 | 0,58332855 | 0,333414880 | 0,142242820 | 0,078352347 |
| 200 | 0,65242016 | 0,513580080 | 0,204455810 | 0,117209320 |

Table 4.4. Budha Data Normal Estimation Error Rates By Using Variable Radius

| k | 3719 Points | 16983 Points | 76376 Points | 295919 Points |
|---|---|---|---|---|
| 2 | 1,31090330 | 1,31184850 | 1,304974200 | 1,475505600 |
| 5 | 0,23570126 | 0,13286060 | 0,060644791 | 0,069094434 |
| 10 | 0,25673980 | 0,16916777 | 0,065123558 | 0,034926001 |
| 15 | 0,29353914 | 0,20541918 | 0,083588004 | 0,039028596 |
| 20 | 0,31311327 | 0,22734129 | 0,102299410 | 0,044769209 |
| 60 | 0,47915909 | 0,36817670 | 0,216293990 | 0,089859903 |
| 100 | 0,56269336 | 0,43100801 | 0,295557101 | 0,132474930 |
| 200 | 0,61903310 | 0,55055439 | 0,385448310 | 0,227266550 |

Table 4.5. Bunny Data Normal Estimation Durations

| k | 239 Points | 994 Points | 4261 Points | 19509 Points |
|---|---|---|---|---|
| 2 | ∼00:00:01 | ∼00:00:01 | ∼00:00:01 | 00:00:01 |
| 5 | ∼00:00:01 | ∼00:00:01 | ∼00:00:01 | 00:00:01 |
| 10 | ∼00:00:01 | ∼00:00:01 | ∼00:00:01 | 00:00:02 |
| 15 | ∼00:00:01 | ∼00:00:01 | ∼00:00:01 | 00:00:03 |
| 20 | ∼00:00:01 | ∼00:00:01 | ∼00:00:01 | 00:00:03 |
| 60 | ∼00:00:01 | ∼00:00:01 | 00:00:01 | 00:00:08 |
| 100 | ∼00:00:01 | ∼00:00:01 | 00:00:03 | 00:00:14 |
| 200 | ∼00:00:01 | 00:00:01 | 00:00:06 | 00:00:29 |

Table 4.6. Dragon Data Normal Estimation Durations

| k | 2719 Points | 11804 Points | 51145 Points | 220421 Points |
|---|---|---|---|---|
| 2 | ∼00:00:01 | 00:00:01 | 00:00:04 | 00:00:13 |
| 5 | ∼00:00:01 | 00:00:01 | 00:00:04 | 00:00:17 |
| 10 | ∼00:00:01 | 00:00:01 | 00:00:05 | 00:00:23 |
| 15 | ∼00:00:01 | 00:00:01 | 00:00:06 | 00:00:30 |
| 20 | ∼00:00:01 | 00:00:02 | 00:00:08 | 00:00:37 |
| 60 | 00:00:01 | 00:00:05 | 00:00:22 | 00:01:32 |
| 100 | 00:00:01 | 00:00:08 | 00:00:36 | 00:02:30 |
| 200 | 00:00:04 | 00:00:18 | 00:01:18 | 00:05:28 |

Table 4.7. Budha Data Normal Estimation Durations

| k | 3719 Points | 16983 Points | 76376 Points | 295919 Points |
|---|---|---|---|---|
| 2 | ∼00:00:01 | 00:00:01 | 00:00:04 | 00:00:20 |
| 5 | ∼00:00:01 | 00:00:01 | 00:00:05 | 00:00:23 |
| 10 | ∼00:00:01 | 00:00:01 | 00:00:07 | 00:00:31 |
| 15 | ∼00:00:01 | 00:00:02 | 00:00:10 | 00:00:42 |
| 20 | ∼00:00:01 | 00:00:02 | 00:00:12 | 00:00:49 |
| 60 | 00:00:01 | 00:00:07 | 00:00:32 | 00:02:06 |
| 100 | 00:00:02 | 00:00:12 | 00:00:50 | 00:03:19 |
| 200 | 00:00:05 | 00:00:25 | 00:01:48 | 00:06:59 |

Table 4.8. Normal Estimation Error Rate Comparison Results of Fixed and Variable Neighbourhood Radii on Bunny Data

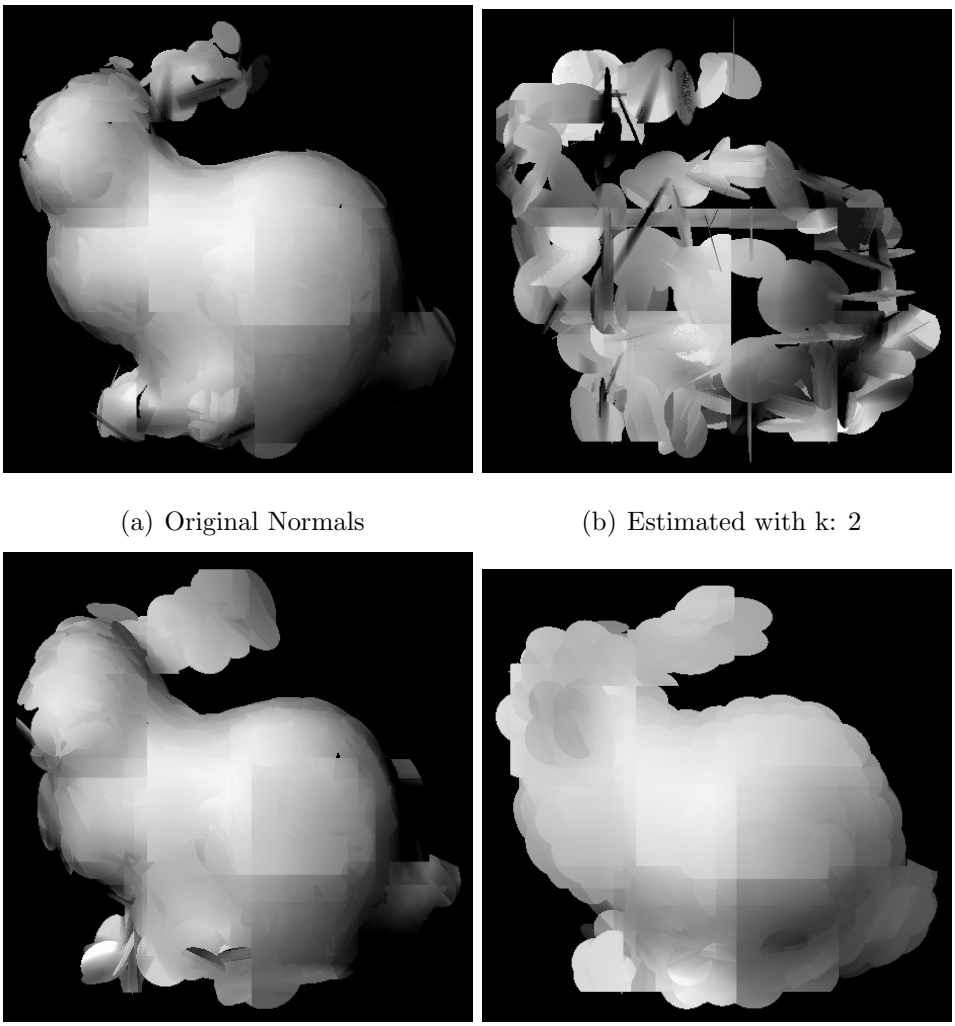
| Normal Estimation Type | 239 Points | 994 Points | 4261 Points | 19509 Points |
|---|---|---|---|---|
| Fixed Radius (r: 0,003) | 1,90749510 | 1,82637200 | 0,301746700 | 0,084886208 |
| Variable Radius (k: 5) | 0,26041976 | 0,10566889 | 0,051443763 | 0,098280884 |

Table 4.9. Normal Estimation Error Rate Comparison Results of Fixed and Variable Neighbourhood Radii on Dragon Data

| Normal Estimation Type | 2719 Points | 11804 Points | 51145 Points | 220421 Points |
|---|---|---|---|---|
| Fixed Radius (r: 0,003) | 1,41884180 | 0,075594805 | 0,067989603 | 0,104416620 |
| Variable Radius (k: 5) | 0,15590234 | 0,078749813 | 0,035893843 | 0,055084959 |

Table 4.10. Normal Estimation Error Rate Comparison Results of Fixed and Variable Neighbourhood Radii on Budha Data

| Normal Estimation Type | 3719 Points | 16983 Points | 76376 Points | 295919 Points |
|---|---|---|---|---|
| Fixed Radius (r: 0,003) | 0,50730735 | 0,17642291 | 0,196050090 | 0,255177440 |
| Variable Radius (k: 5) | 0,23570126 | 0,13286060 | 0,060644791 | 0,069094434 |



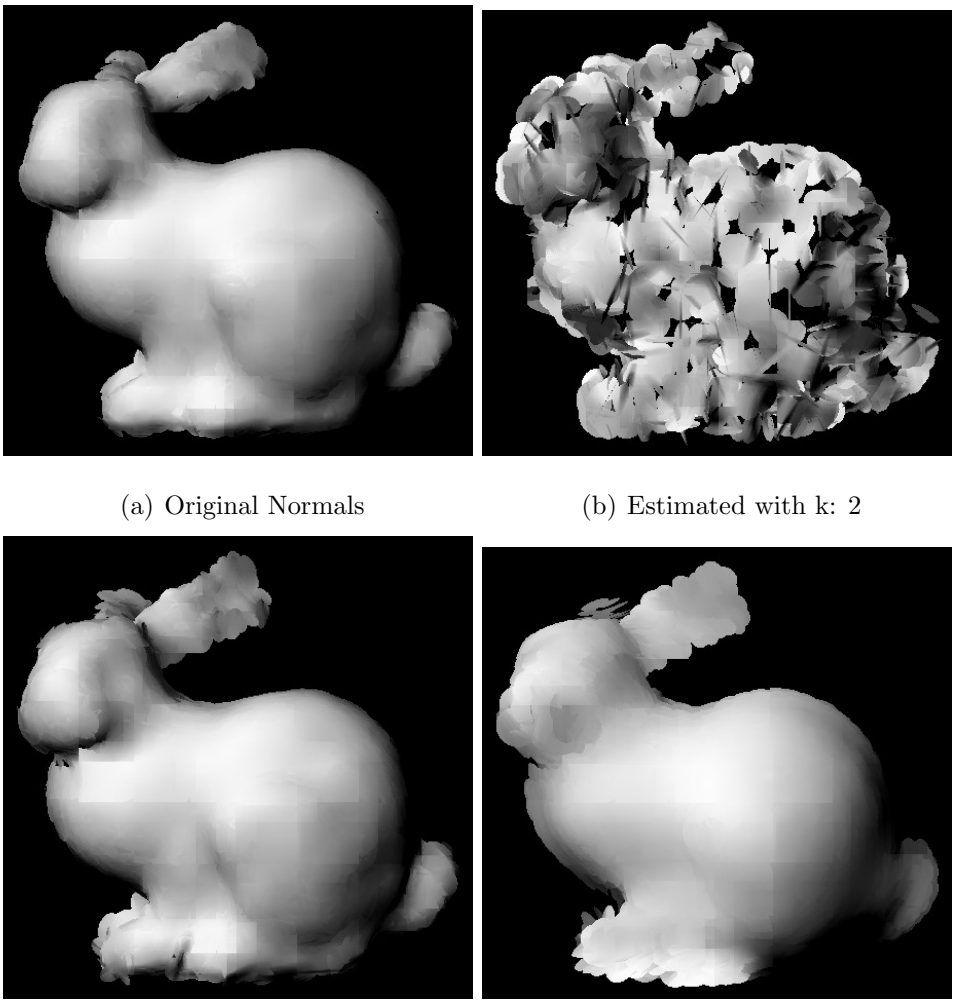
(a) Original Normals

(b) Estimated with k: 2

(c) Estimated with k: 5

(d) Estimated with k: 60

Figure 4.4. Estimating Bunny data normals with 500 points and variable neighborhood radii



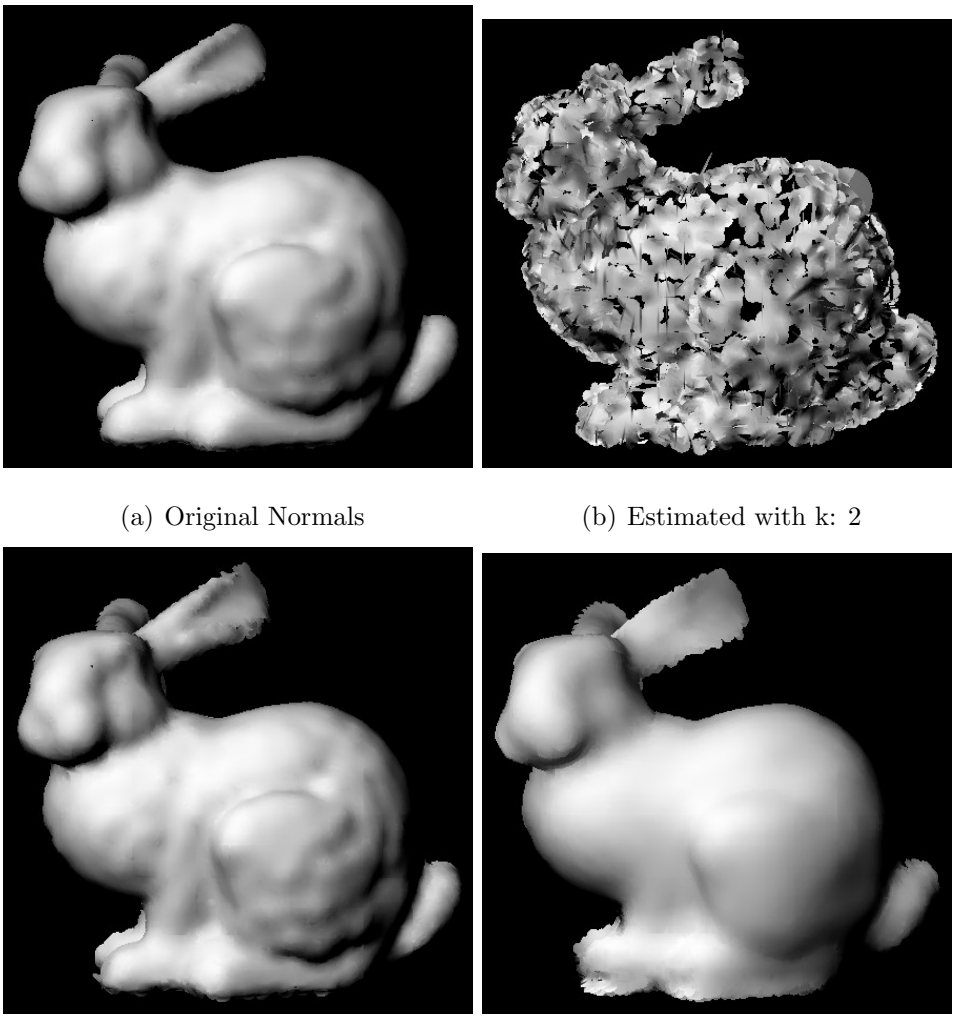
(a) Original Normals

(b) Estimated with k: 2

(c) Estimated with k: 5

(d) Estimated with k: 60

Figure 4.5. Estimating Bunny data normals with 2000 points and variable neighborhood radii



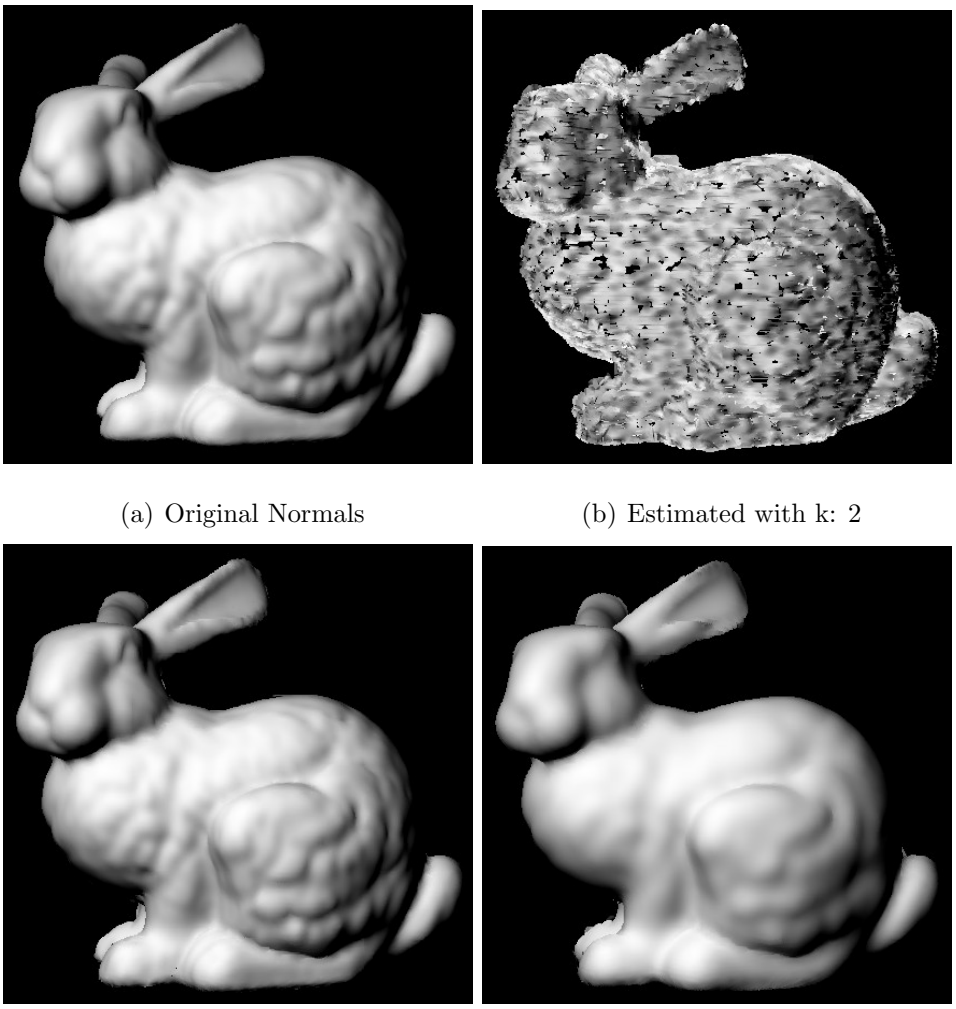
(a) Original Normals

(b) Estimated with k: 2

(c) Estimated with k: 5

(d) Estimated with k: 60

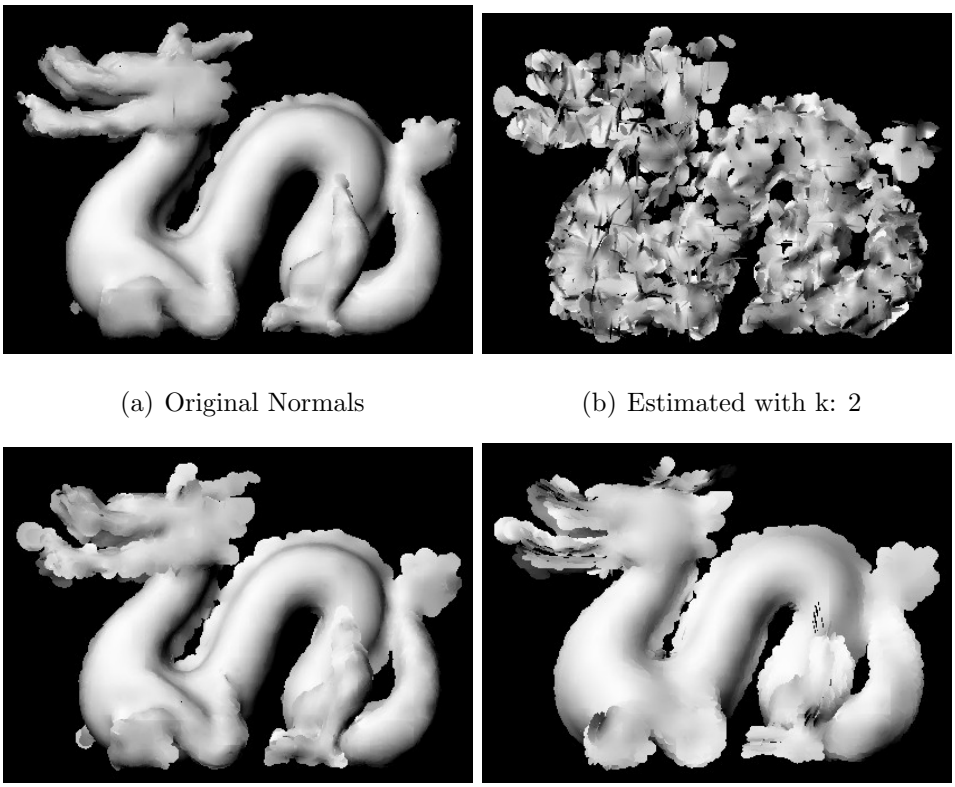
Figure 4.6. Estimating Bunny data normals with 8000 points and variable neighborhood radii



(a) Original Normals (b) Estimated with k: 2

(c) Estimated with k: 8 (d) Estimated with k: 60

Figure 4.7. Estimating Bunny data normals with 36000 points and variable neighborhood radii



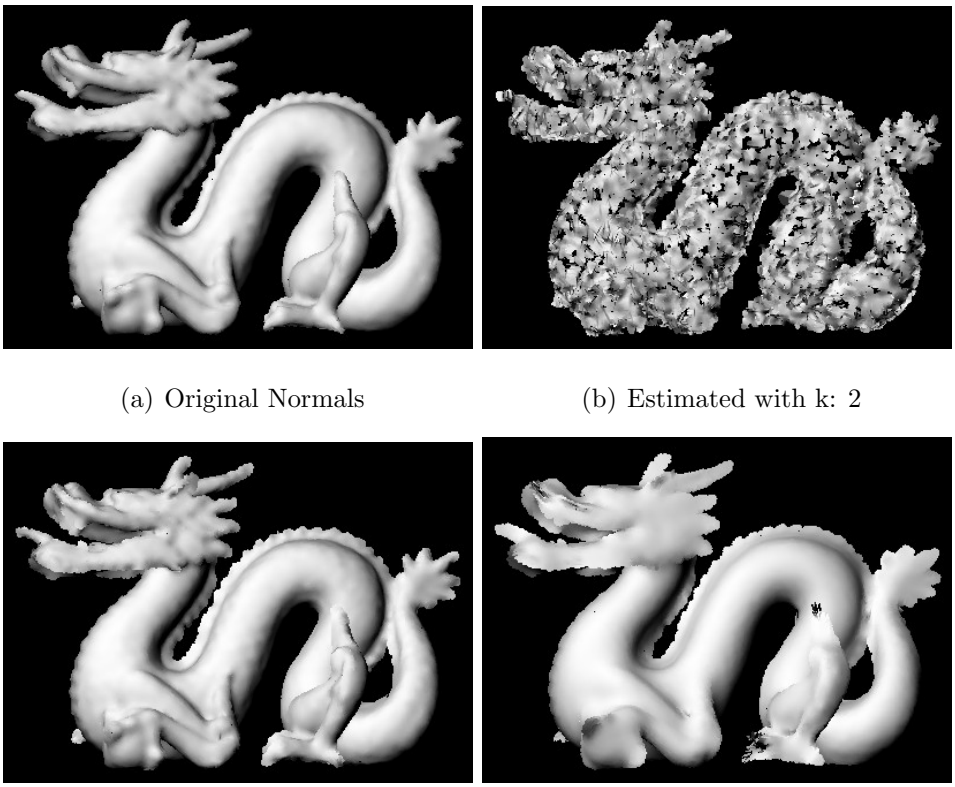
(a) Original Normals

(b) Estimated with k: 2

(c) Estimated with k: 5

(d) Estimated with k: 60

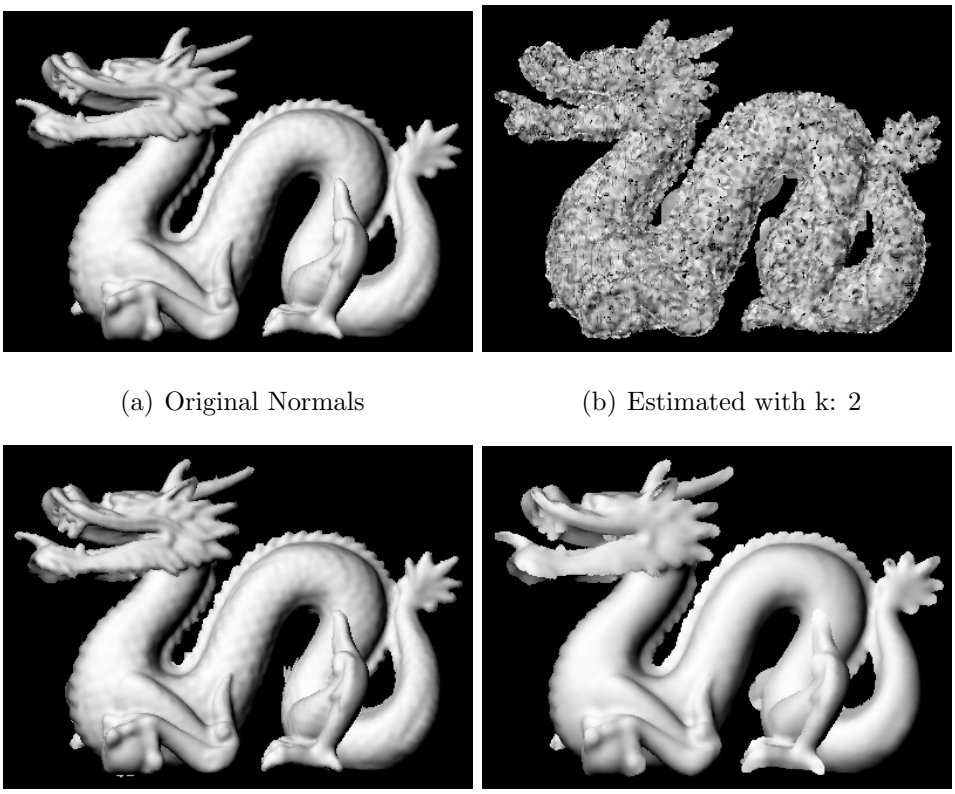
Figure 4.8. Estimating Dragon data normals with 11k points and variable neighborhood radii



(a) Original Normals (b) Estimated with k: 2

(c) Estimated with k: 5 (d) Estimated with k: 60

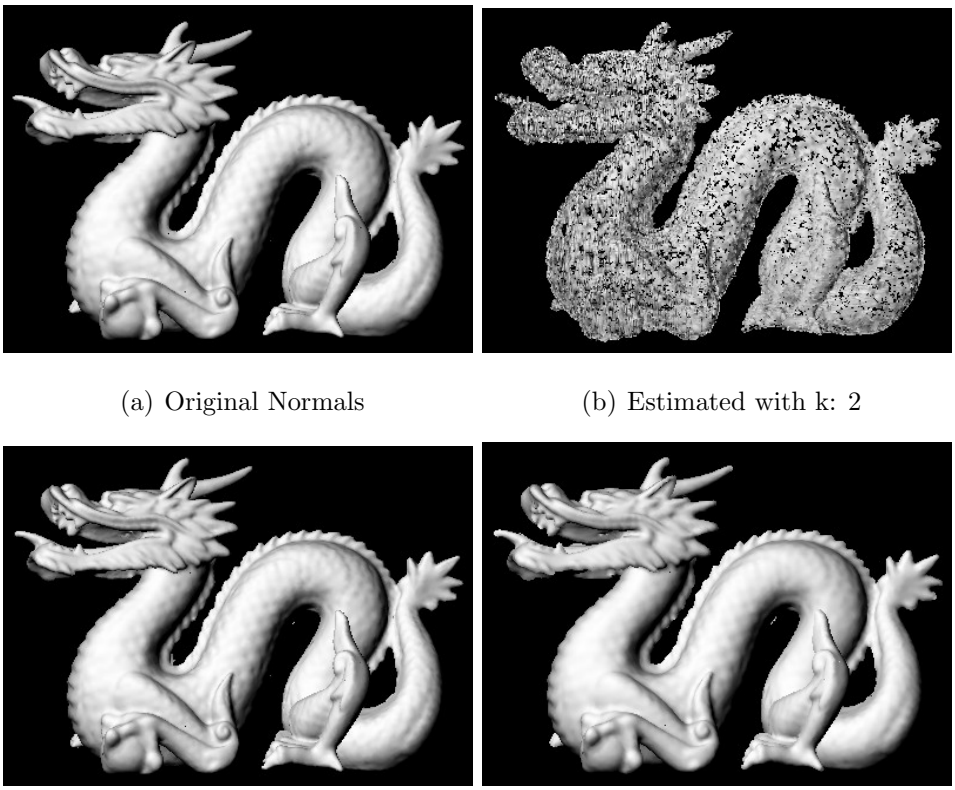
Figure 4.9. Estimating Dragon data normals with 22k points and variable neighborhood radii



(a) Original Normals (b) Estimated with k: 2

(c) Estimated with k: 7 (d) Estimated with k: 60

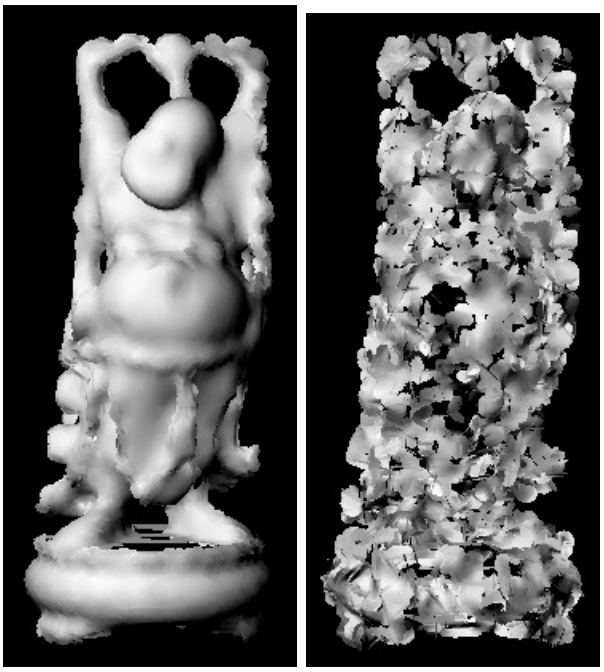
Figure 4.10. Estimating Dragon data normals with 100k points and variable neighborhood radii



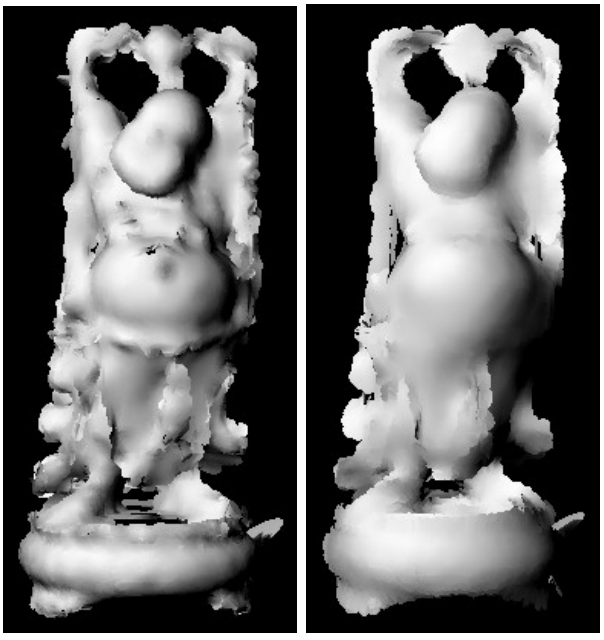
(a) Original Normals (b) Estimated with k: 2

(c) Estimated with k: 9 (d) Estimated with k: 60

Figure 4.11. Estimating Dragon data normals with 437k points and variable neighborhood radii

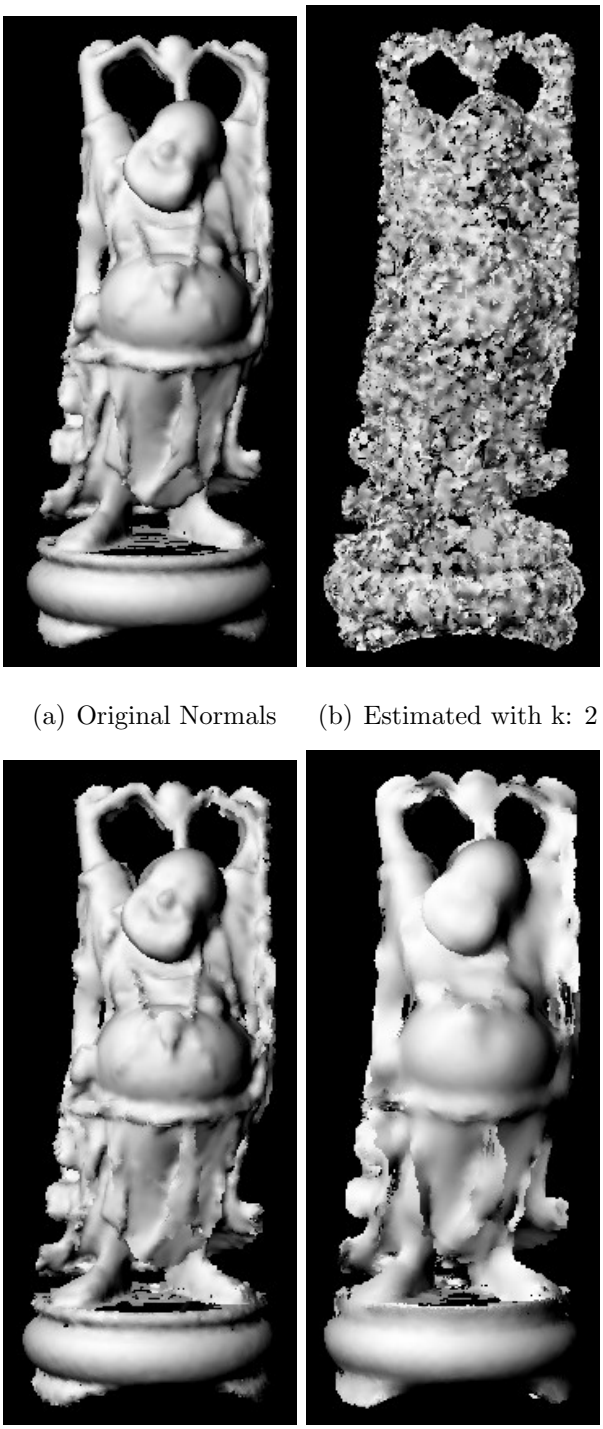


(a) Original Normals (b) Estimated with k: 2



(c) Estimated with k: 5 (d) Estimated with k: 60

Figure 4.12. Estimating Budha data normals with 7k points and variable neighborhood radii



(a) Original Normals (b) Estimated with k: 2

(c) Estimated with k: 5 (d) Estimated with k: 60

Figure 4.13. Estimating Budha data normals with 32k points and variable neighborhood radii

(a) Original Normals (b) Estimated with k: 2

(c) Estimated with k: 5 (d) Estimated with k: 60

Figure 4.14. Estimating Budha data normals with 144k points and variable neighborhood radii

(a) Original Normals (b) Estimated with k: 2

(c) Estimated with k: 9 (d) Estimated with k: 60

Figure 4.15. Estimating Budha data normals with 543k points and variable neighborhood radii

## 4.2. Ray Tracing

### 4.2.1. Stanford Data Set

We have used various point cloud samples from Stanford University which contains different number of points. The list of the data set is shown in Table 4.11.

Table 4.11. Stanford Data Set Information

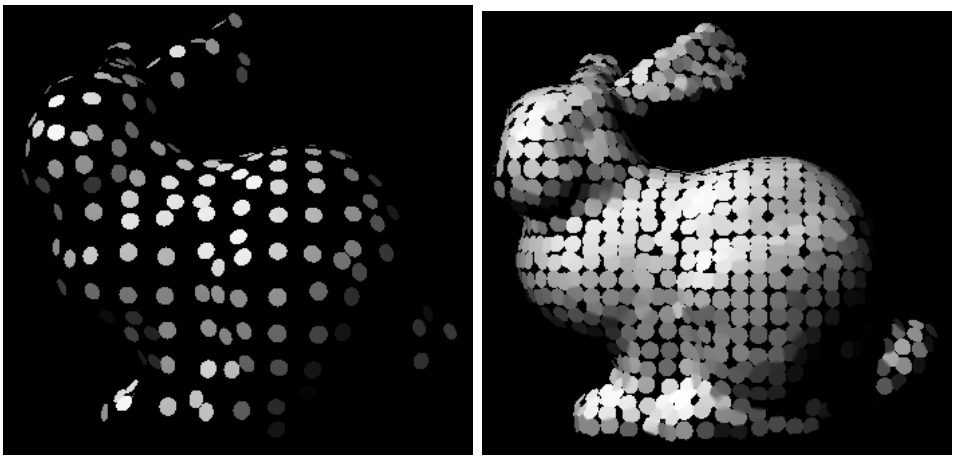
| | | | | |
|---|---|---|---|---|
| Bunny | 453 Points | 1889 Points | 8171 Points | 35947 Points |
| Dragon | 5205 Points | 22998 Points | 100250 Points | 437645 Points |
| Budha | 7108 Points | 32328 Points | 144647 Points | 543652 Points |

Nevertheless it is impossible to take infinite point samples and too much number of points slows down ray tracing process. So an optimum point cloud should be chosen for enough level of detail. We have mostly used the bunny of 35947 points for experiments which was sufficient.

Another ray tracing parameter is the radius of the discs we have used for each point. These discs are the only objects intersected by rays and effects the results directly. If disc radii are too small there will be gaps between points which are not hit by rays and not illuminated. Optimum disc radius should be as big as the maximum gap between points.

If the radius becomes larger than this optimum value, results will not have significant changes. This is the result of the method we used. Distance between points also between disc centers are not effected by radius. When an intersection occurs no matter which disc is intersected, only the discs close to the intersection point will be processed. Close discs are the ones whose centers are inside the cylinder created around the ray. Therefore the number of discs in the cylinder and interpolated attributes will not change by the disc radius.

This condition is true for surface point clouds. The cylinder length depends on the



(a) 453 points, Disc r: 6, Cylinder r: 6



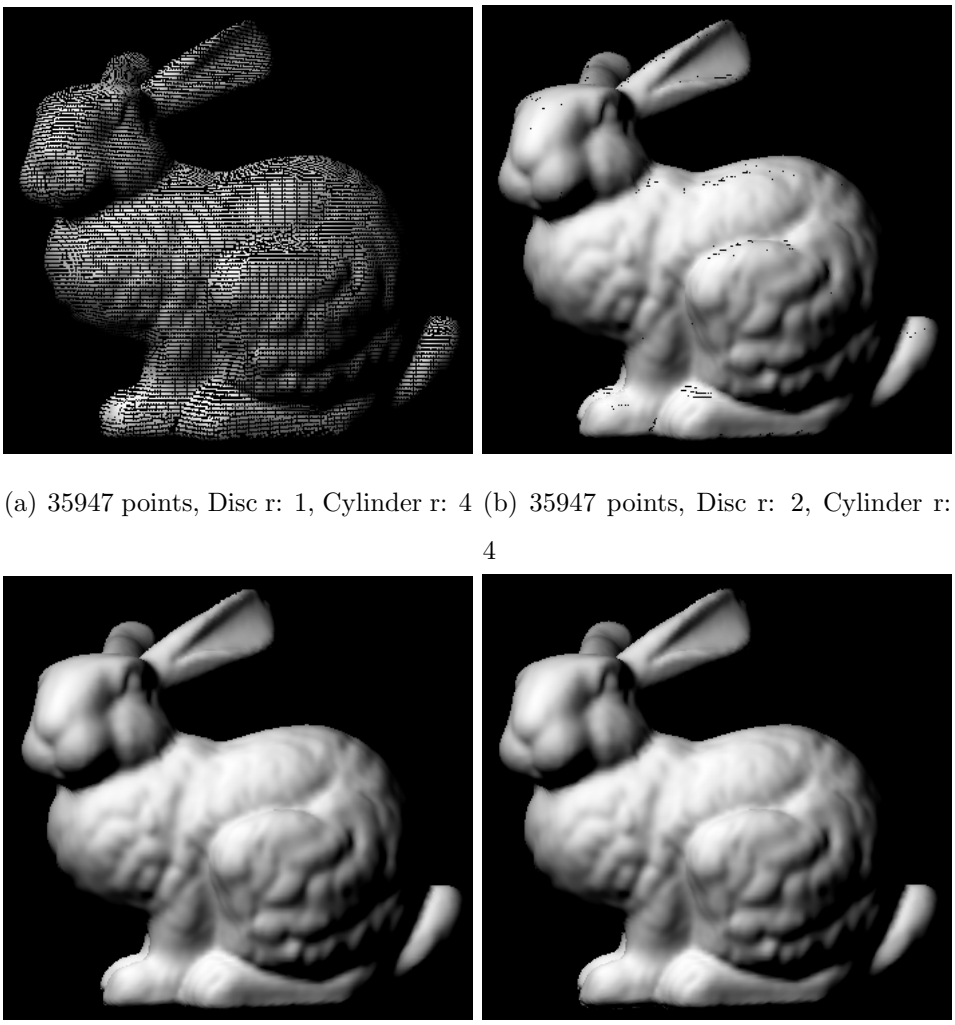
(b) 1889 points, Disc r: 6, Cylinder r: 6

(c) 8171 points, Disc r: 5, Cylinder r: 6

(d) 35947 points, Disc r: 3, Cylinder r: 4

Figure 4.16. Rendering Stanford bunny with different number of points

disc radius and if the disc radius increases too much the discs behind the intersection point will be included to interpolation.



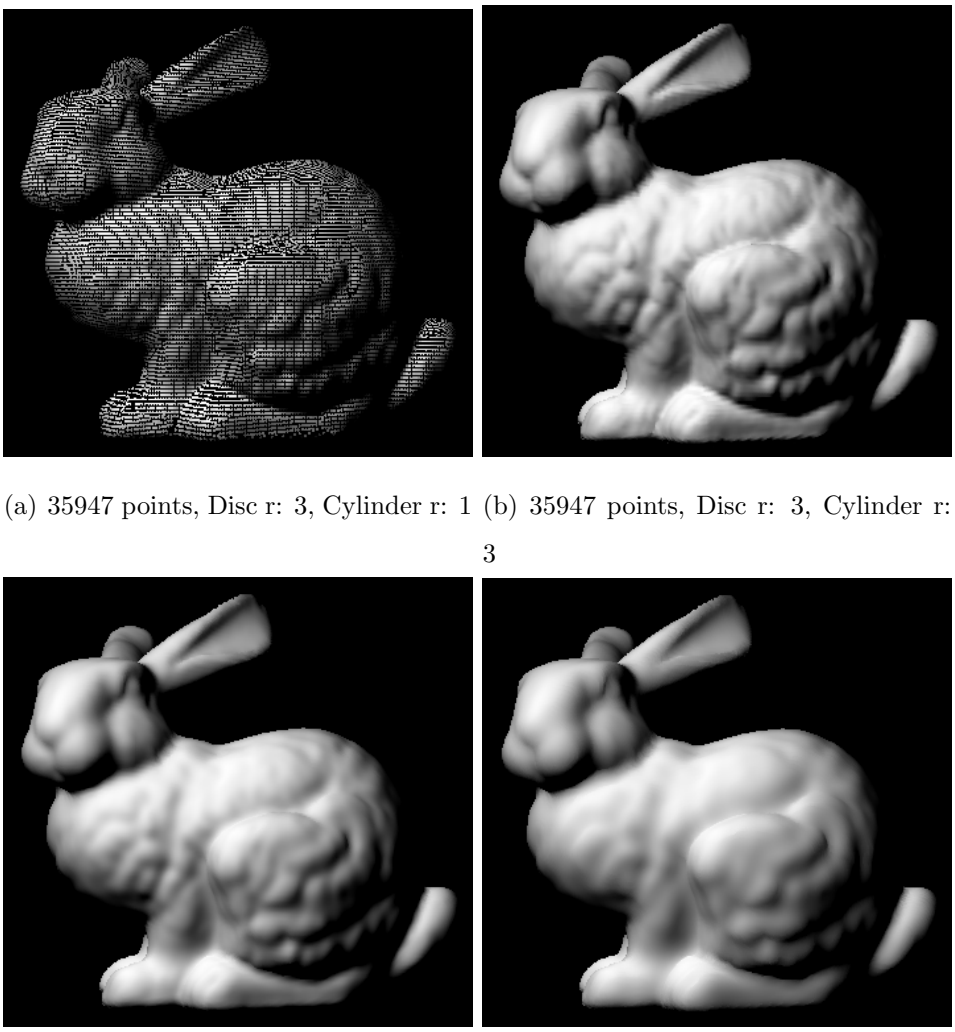
(a) 35947 points, Disc r: 1, Cylinder r: 4 (b) 35947 points, Disc r: 2, Cylinder r: 4

(c) 35947 points, Disc r: 3, Cylinder r: 4 (d) 35947 points, Disc r: 4, Cylinder r: 4

Figure 4.17. Rendering Stanford bunny with different disc radii

As mentioned before a cylinder is created around ray when an intersection occurs. This cylinder starts from the intersection point and has a length of disc radius to cover only the surface discs. Cylinder radius directly effects the number of discs that will be interpolated causing smoothness.

If the cylinder radius too small there will not be enough interpolation in order to provide smoothness. If it is smaller than the disc radius the edge points on the intersected disc will not be included to calculation. This will cause gaps between points on low density point clouds as if the disc radius is the same as the cylinder radius.



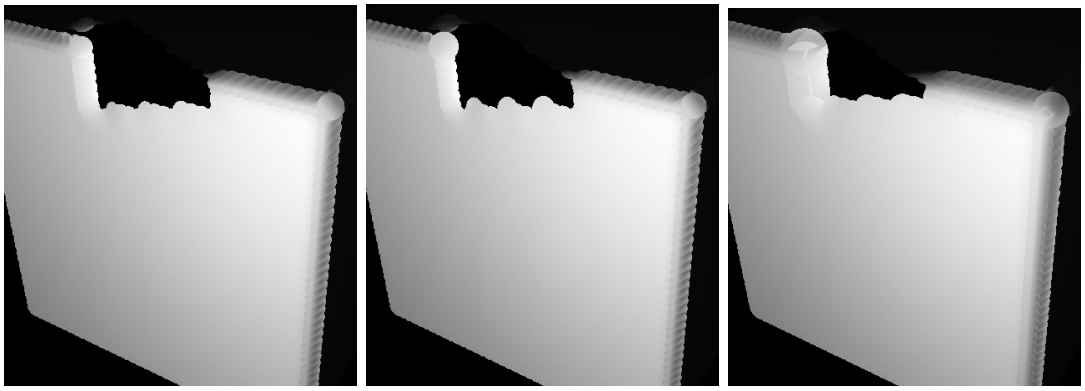
(a) 35947 points, Disc r: 3, Cylinder r: 1 (b) 35947 points, Disc r: 3, Cylinder r: 3

(c) 35947 points, Disc r: 3, Cylinder r: 6 (d) 35947 points, Disc r: 3, Cylinder r: 10

Figure 4.18. Rendering Stanford bunny with different cylinder radii

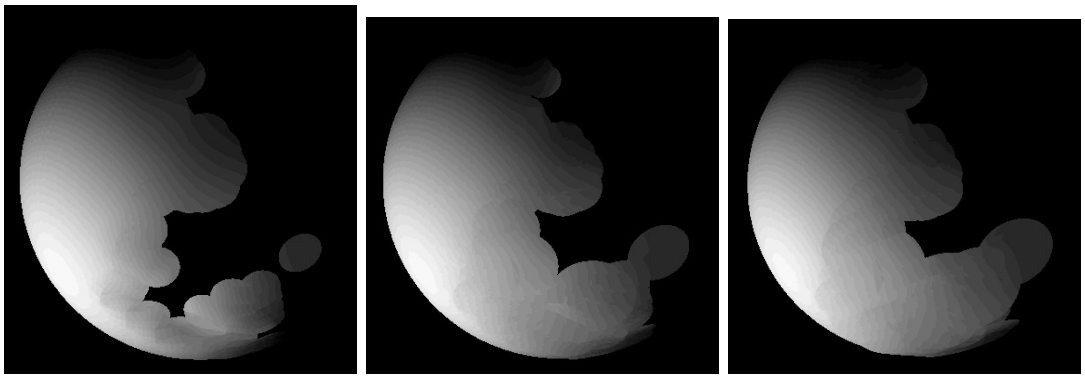
On the other hand if the cylinder radius becomes to large, the increased smoothness will lower the level of detail. There is no optimum cylinder radius since the smoothness - detail balance is a relative issue. For most of the point clouds we have selected cylinder radius same as the disc radius which has given sufficient level of detail and smoothness.

4.2.1.1. Using Variable Disc Radii. In this section experimentation results collected by using variable disc radius are presented. 3 different objects with 4 different number of points are used to observe the differences. It is seen that disc radius variability provides continuous surfaces while keeping details if the k index is selected properly. Too small k values results in holes between points and too large k values causes smoothness in dense areas. Quality difference between fixed and variable disc radii is especially seen in point clouds with higher number of points.



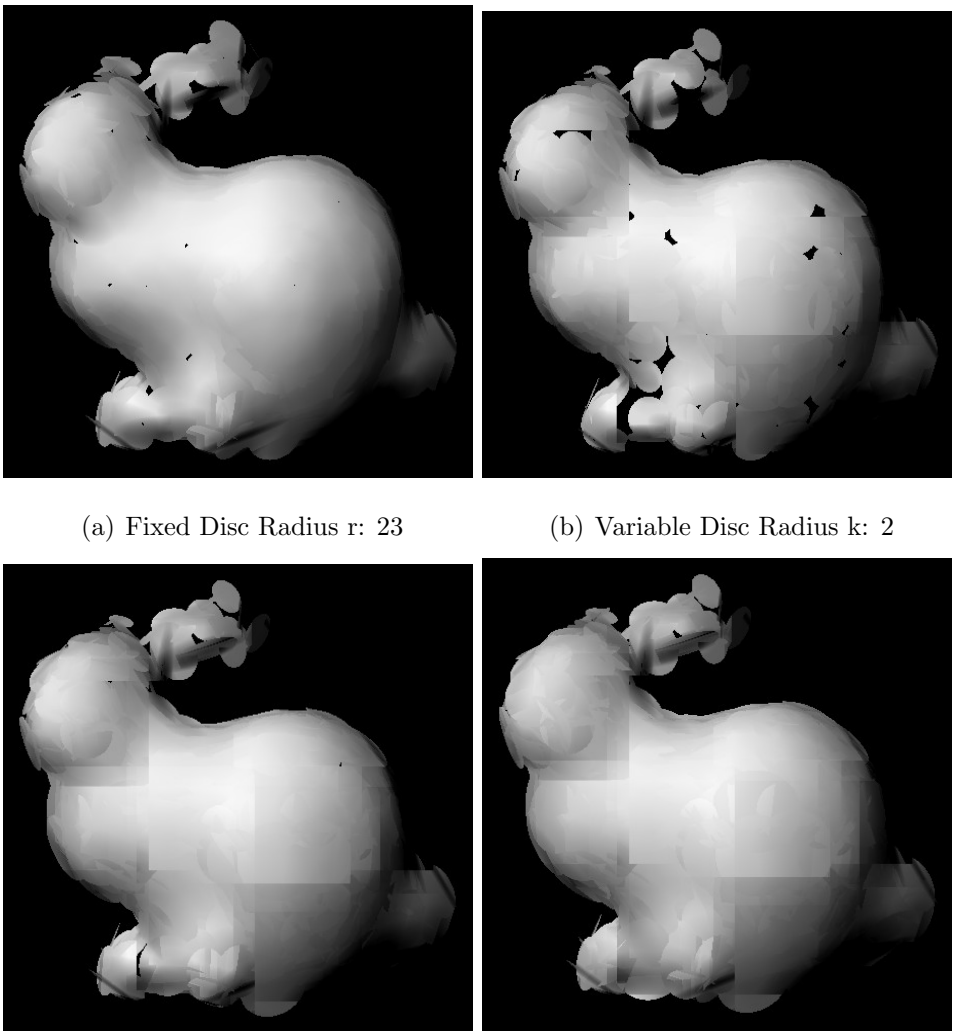
(a) Fixed Disc Radius (b) Variable Disc Radius k: 4 (c) Variable Disc Radius k: 7

Figure 4.19. Ray tracing box with a hole with variable disc radii



(a) Fixed Disc Radius (b) Variable Disc Radius k: 3 (c) Variable Disc Radius k: 6

Figure 4.20. Ray tracing sphere with a hole with variable disc radii



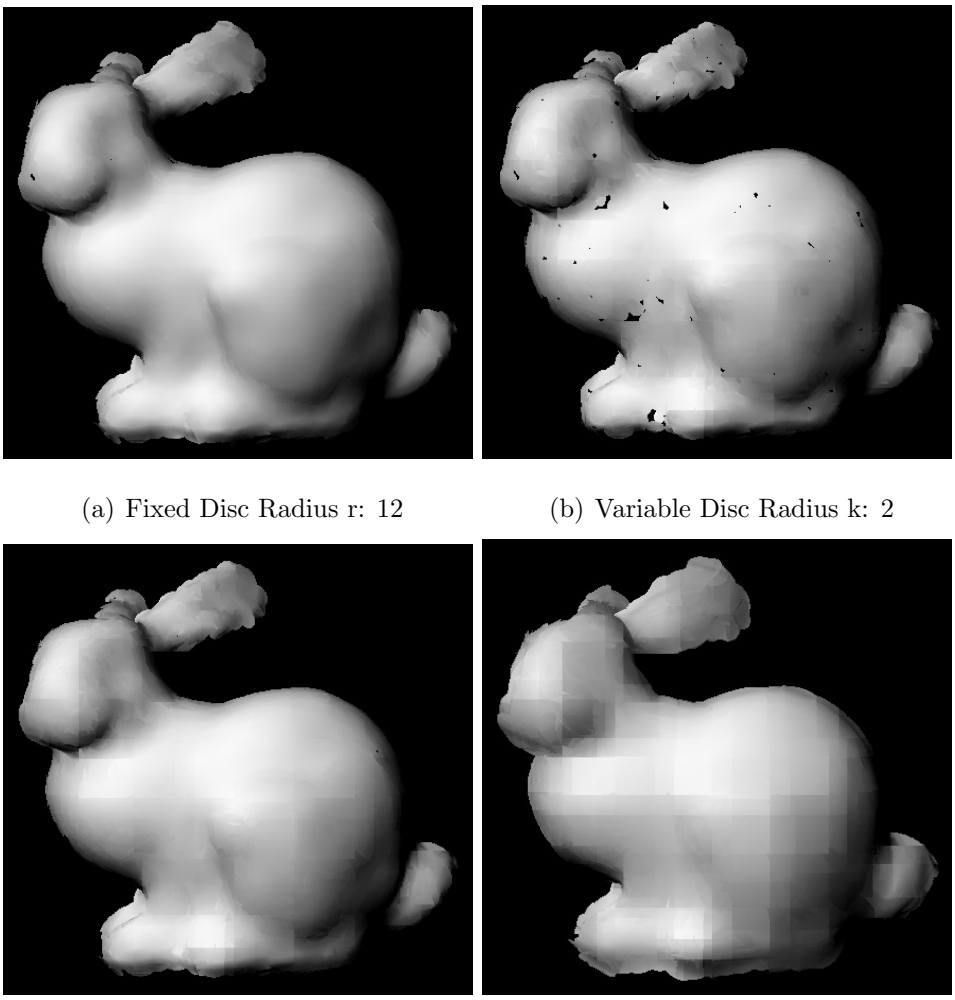
(a) Fixed Disc Radius r: 23

(b) Variable Disc Radius k: 2

(c) Variable Disc Radius k: 3

(d) Variable Disc Radius k: 4

Figure 4.21. Ray tracing Bunny Data with 453 points and variable disc radii



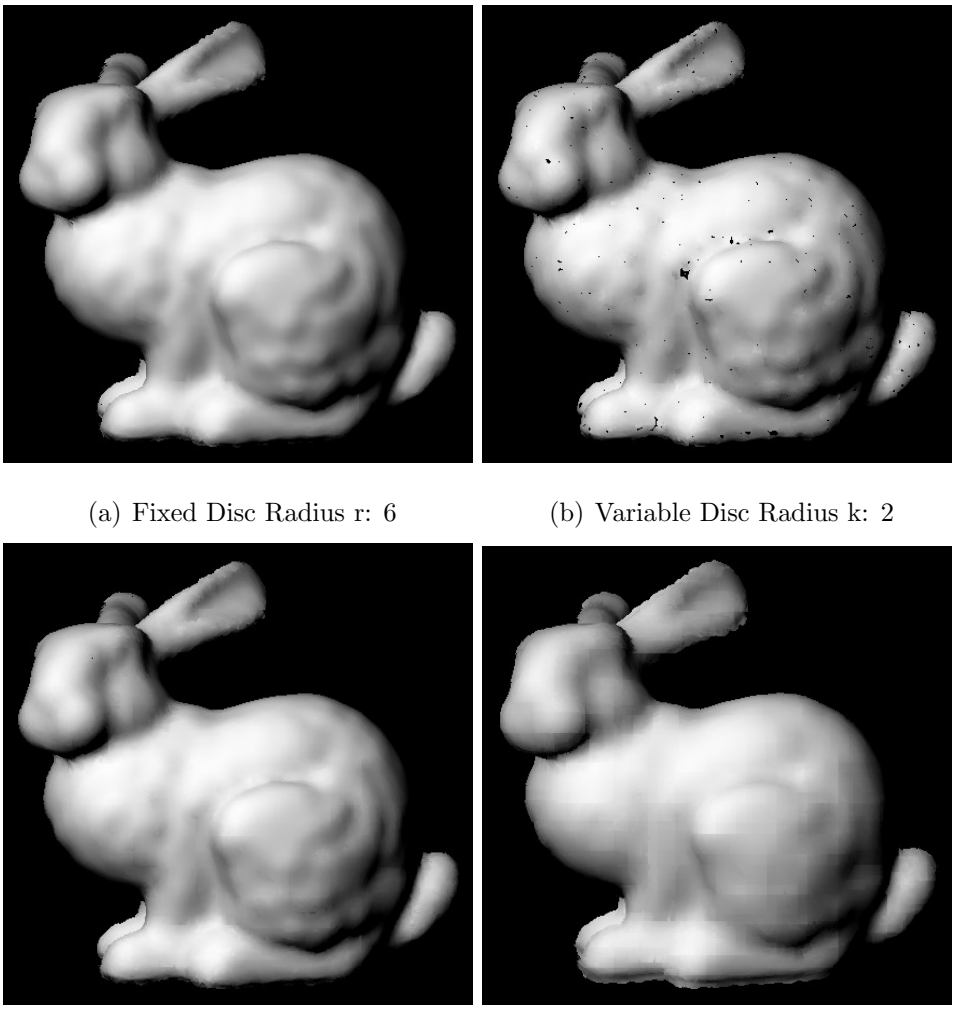
(a) Fixed Disc Radius r: 12

(b) Variable Disc Radius k: 2

(c) Variable Disc Radius k: 3

(d) Variable Disc Radius k: 10

Figure 4.22. Ray tracing Bunny Data with 1889 points and variable disc radii



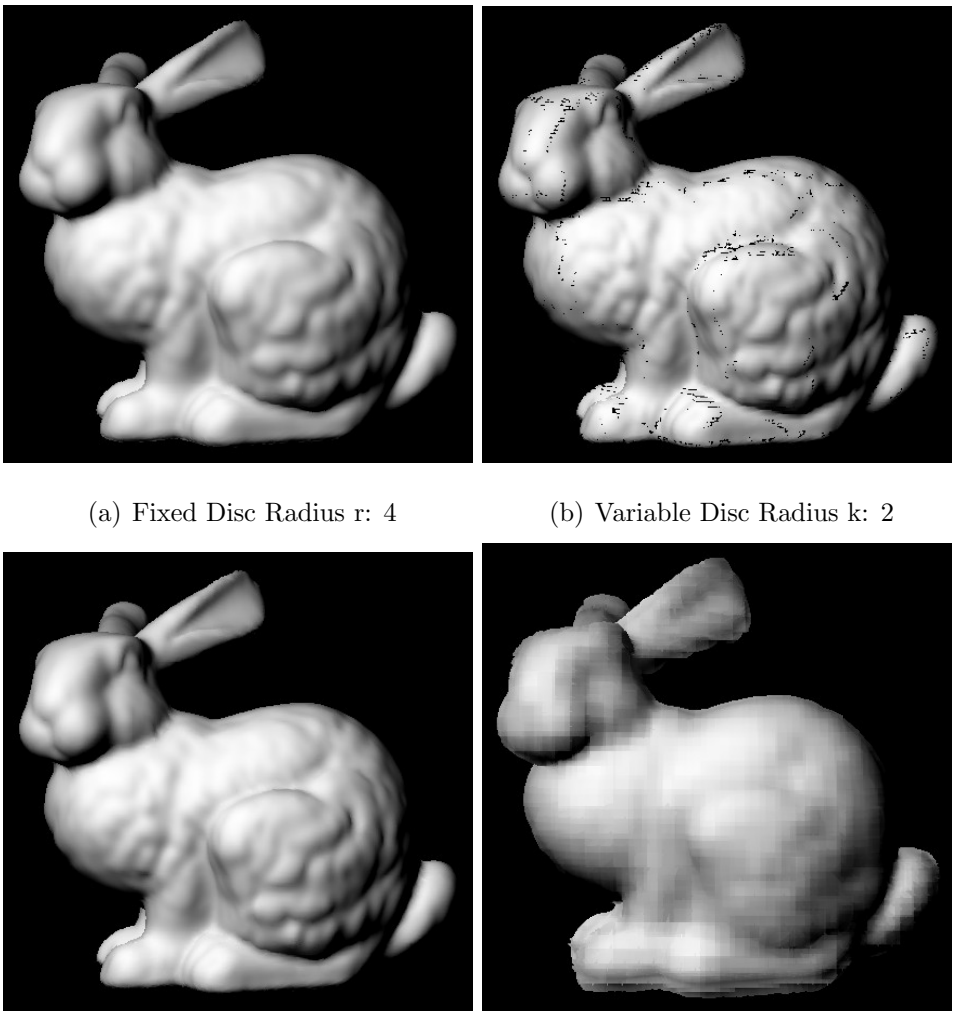
(a) Fixed Disc Radius r: 6

(b) Variable Disc Radius k: 2

(c) Variable Disc Radius k: 3

(d) Variable Disc Radius k: 10

Figure 4.23. Ray tracing Bunny Data with 8171 points and variable disc radii



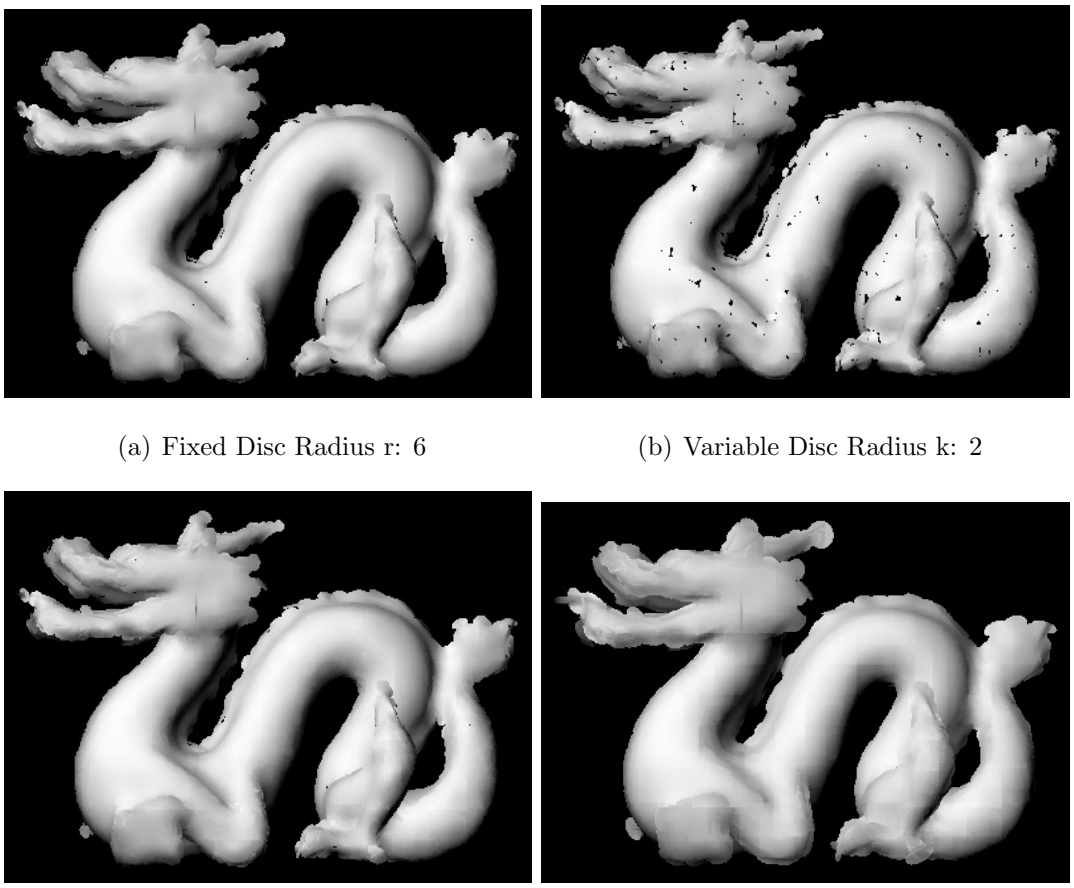
(a) Fixed Disc Radius r: 4

(b) Variable Disc Radius k: 2

(c) Variable Disc Radius k: 4

(d) Variable Disc Radius k: 100

Figure 4.24. Ray tracing Bunny Data with 35947 points and variable disc radii



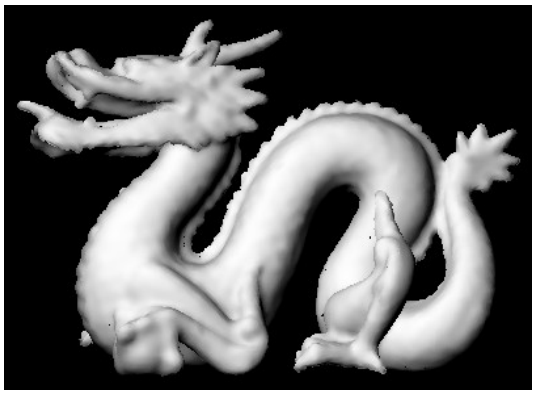
(a) Fixed Disc Radius r: 6

(b) Variable Disc Radius k: 2

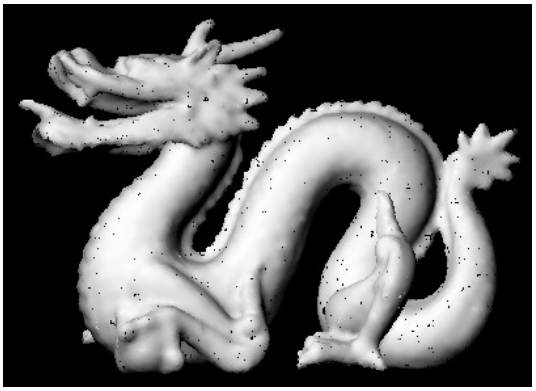
(c) Variable Disc Radius k: 3

(d) Variable Disc Radius k: 7

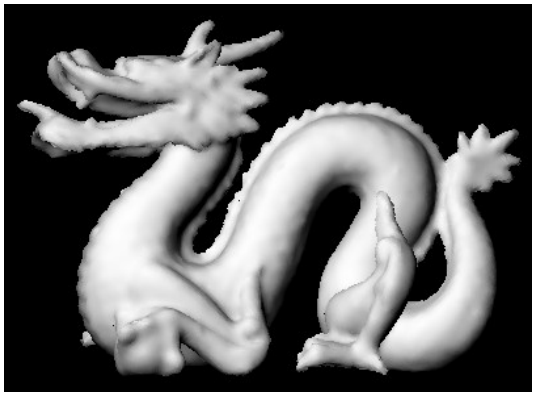
Figure 4.25. Ray tracing Dragon Data with 11k points and variable disc radii



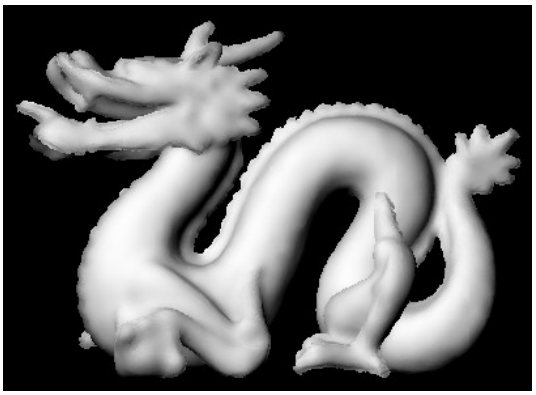
(a) Fixed Disc Radius r: 3



(b) Variable Disc Radius k: 2

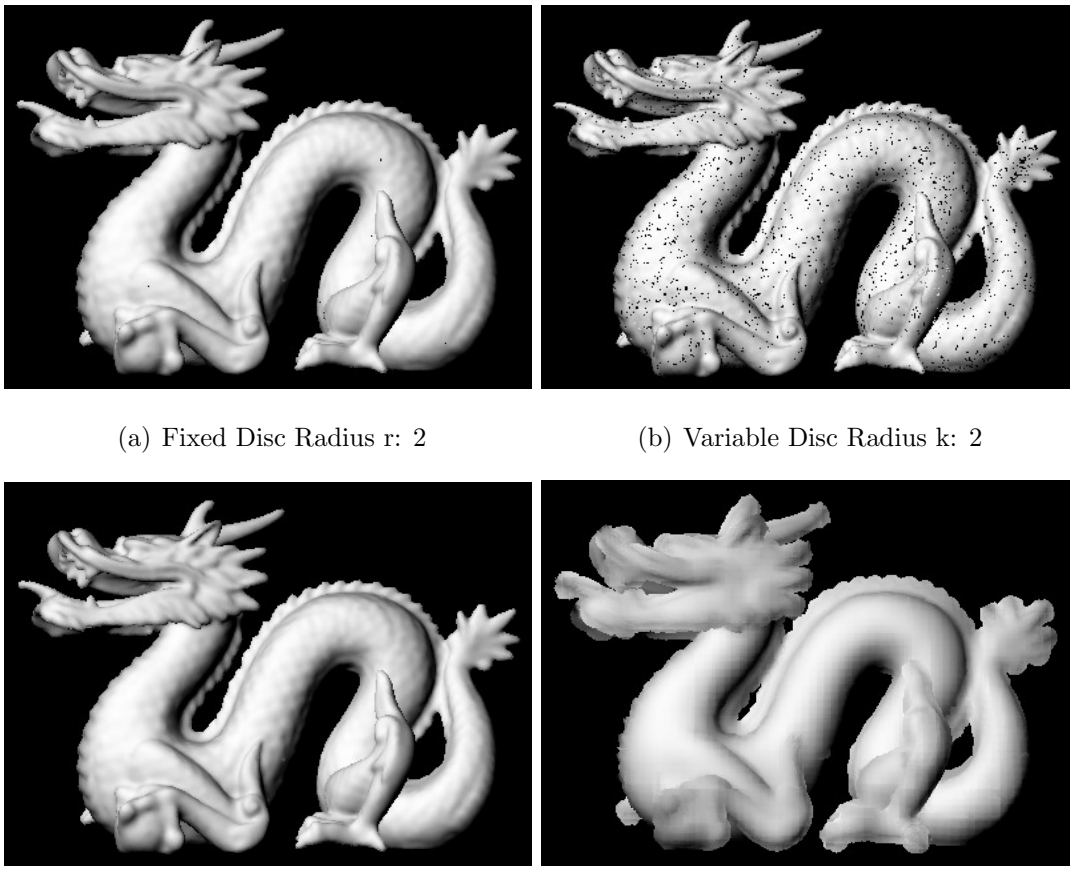


(c) Variable Disc Radius k: 3



(d) Variable Disc Radius k: 7

Figure 4.26. Ray tracing Dragon Data with 22k points and variable disc radii



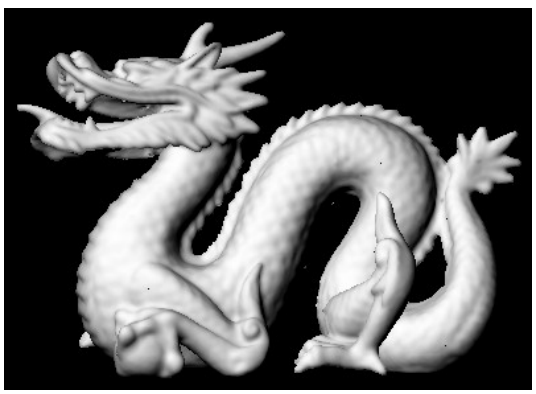
(a) Fixed Disc Radius r: 2

(b) Variable Disc Radius k: 2

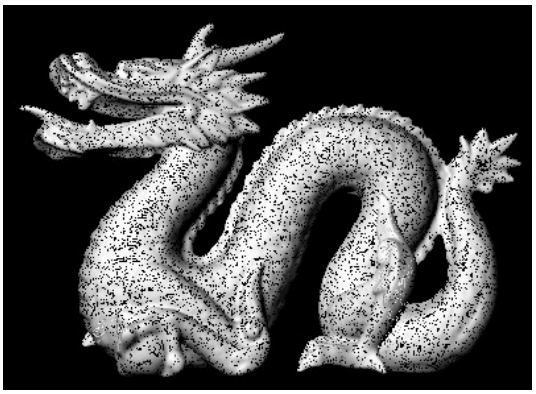
(c) Variable Disc Radius k: 6

(d) Variable Disc Radius k: 100

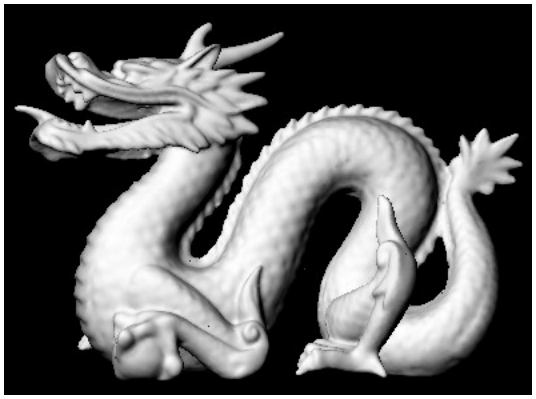
Figure 4.27. Ray tracing Dragon Data with 100k points and variable disc radii



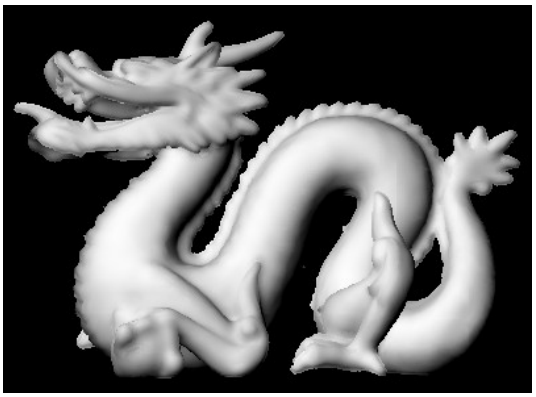
(a) Fixed Disc Radius r: 1.8



(b) Variable Disc Radius k: 2

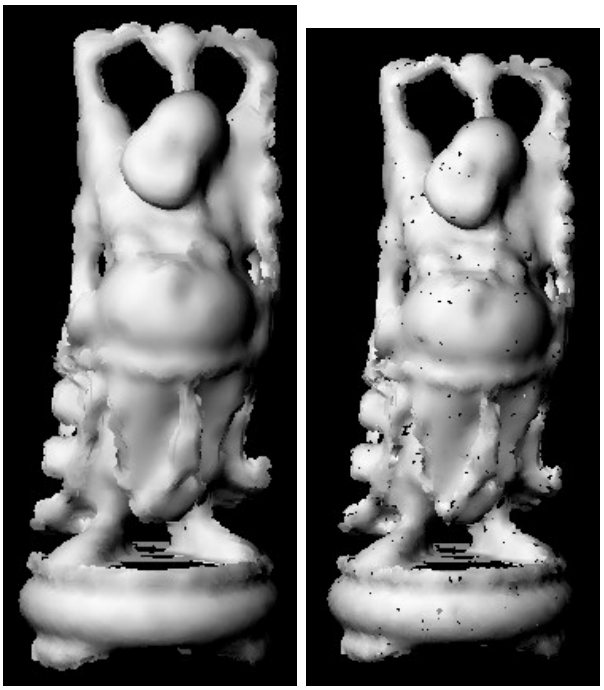


(c) Variable Disc Radius k: 9



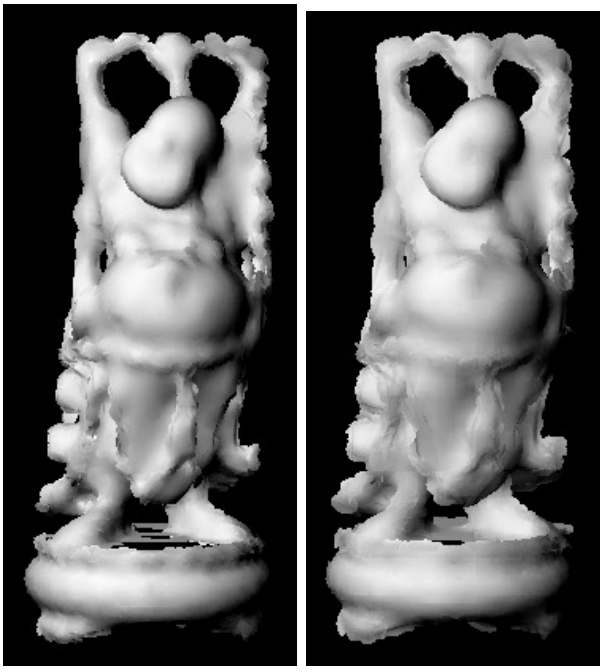
(d) Variable Disc Radius k: 100

Figure 4.28. Ray tracing Dragon Data with 437k points and variable disc radii



(a) Fixed Disc Radius r: 5

(b) Variable Disc Radius k: 2



(c) Variable Disc Radius k: 3

(d) Variable Disc Radius k: 7

Figure 4.29. Ray tracing Budha Data with 7k points and variable disc radii

(a) Fixed Disc Radius r: 3

(b) Variable Disc Radius k: 2

(c) Variable Disc Radius k: 3

(d) Variable Disc Radius k: 7

Figure 4.30. Ray tracing Budha Data with 32k points and variable disc radii

(a) Fixed Disc Radius r: 2

(b) Variable Disc Radius k: 2

(c) Variable Disc Radius k: 5

(d) Variable Disc Radius k: 100

Figure 4.31. Ray tracing Budha Data with 144k points and variable disc radii

(a) Fixed Disc Radius r: 1.8

(b) Variable Disc Radius k: 2

(c) Variable Disc Radius k: 12

(d) Variable Disc Radius k: 100

Figure 4.32. Ray tracing Budha Data with 543k points and variable disc radii

### 4.2.2. Face Data Set

After making experiments by using Stanford bunny, we applied the same method for the 3D scanned face data obtained from the Bogazici University Department of Computer Engineering[6]. Adjustments for point coordinates, disc and cylinder radii is required due to different data ranges. Other difference is points normals which are not part of the dace data unlike the bunny.

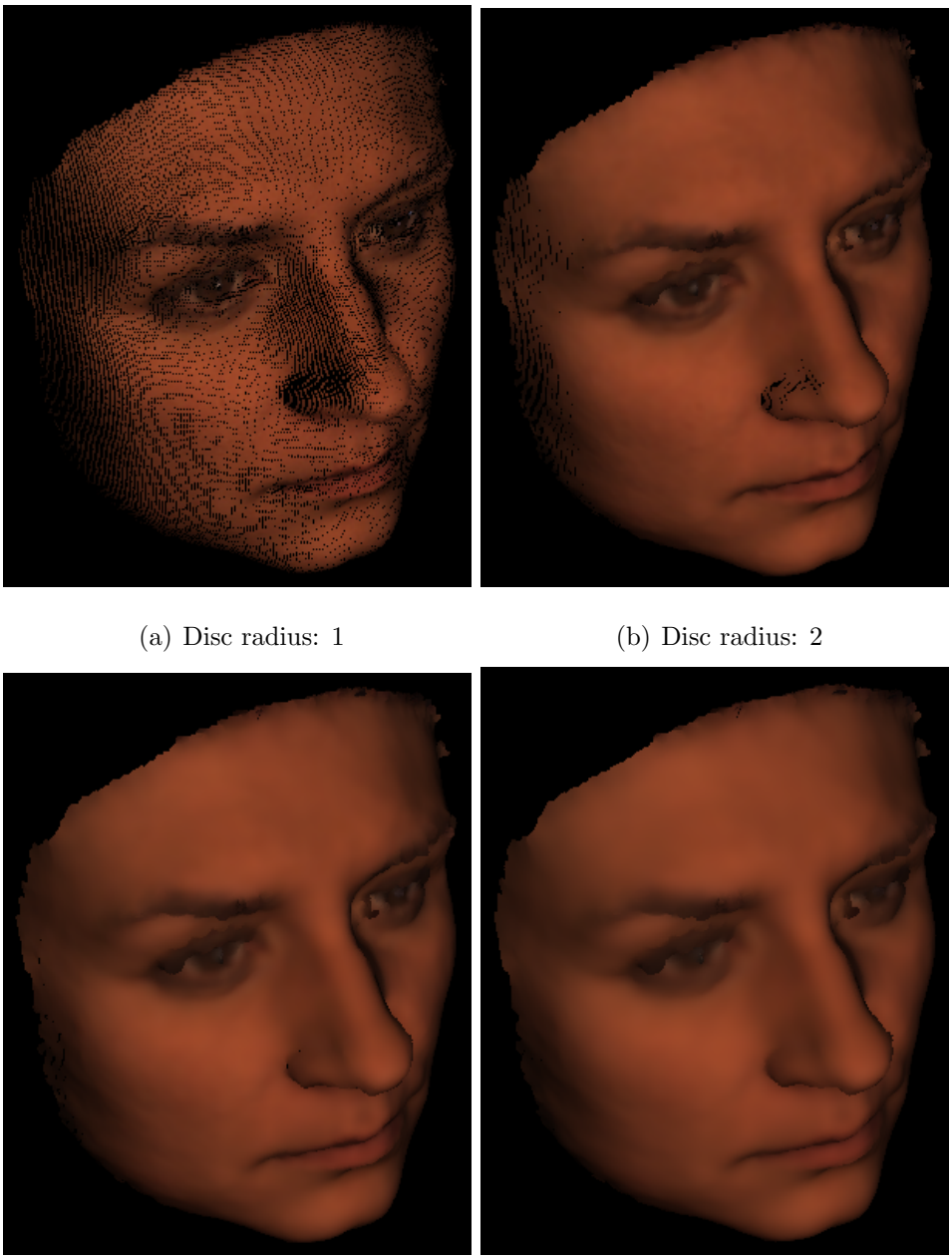
We have used least square minimization method to estimate the point normals. The results are satisfactory except some artifacts due to noise and different detail levels among the face data. Irregular point distribution in areas such as moustaches eye brows causes errors in normal estimation.

Irregular distribution of point data also effects the ray tracing method. We use the same disc radius for whole point cloud which causes gaps among low density areas. If the disc and cylinder radii is increased smoothness and detail loss is also increased.

Figure 4.33. Ray tracing face data 1 with fixed disc radius

Figure 4.34. Ray tracing face data 2 with fixed disc radius

Figure 4.35. Ray tracing face data 3 with fixed disc radius



(a) Disc radius: 1

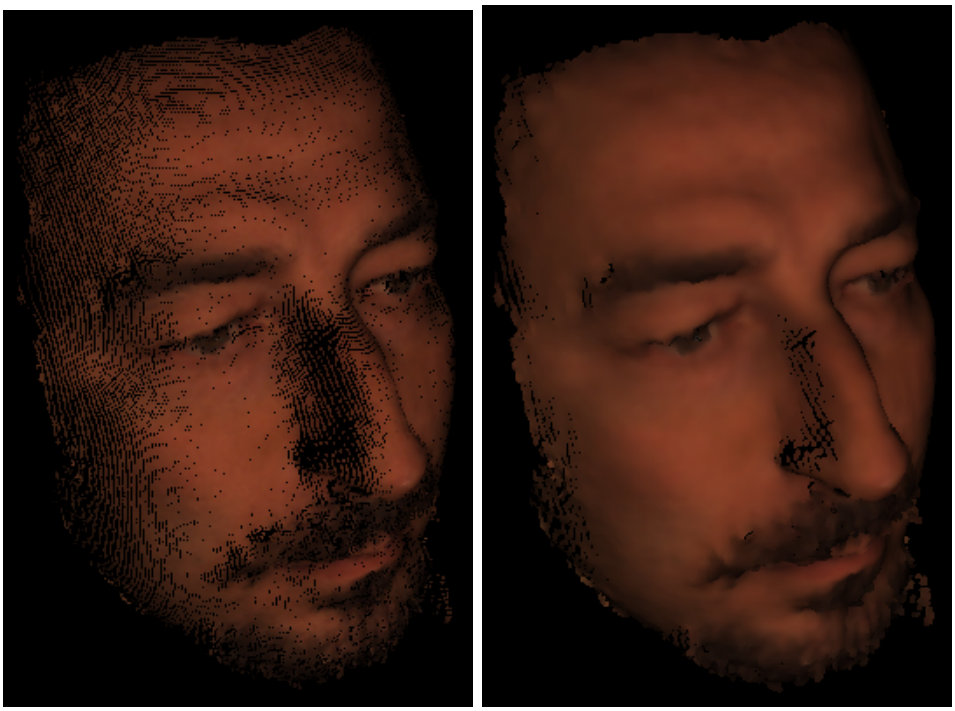
(b) Disc radius: 2

(c) Disc radius: 3

(d) Disc radius: 4

Figure 4.36. Ray tracing face data 1 with different fixed disc radii

Figure 4.37. Ray tracing face data 1 with variable disc radius



(a) Disc radius: 1



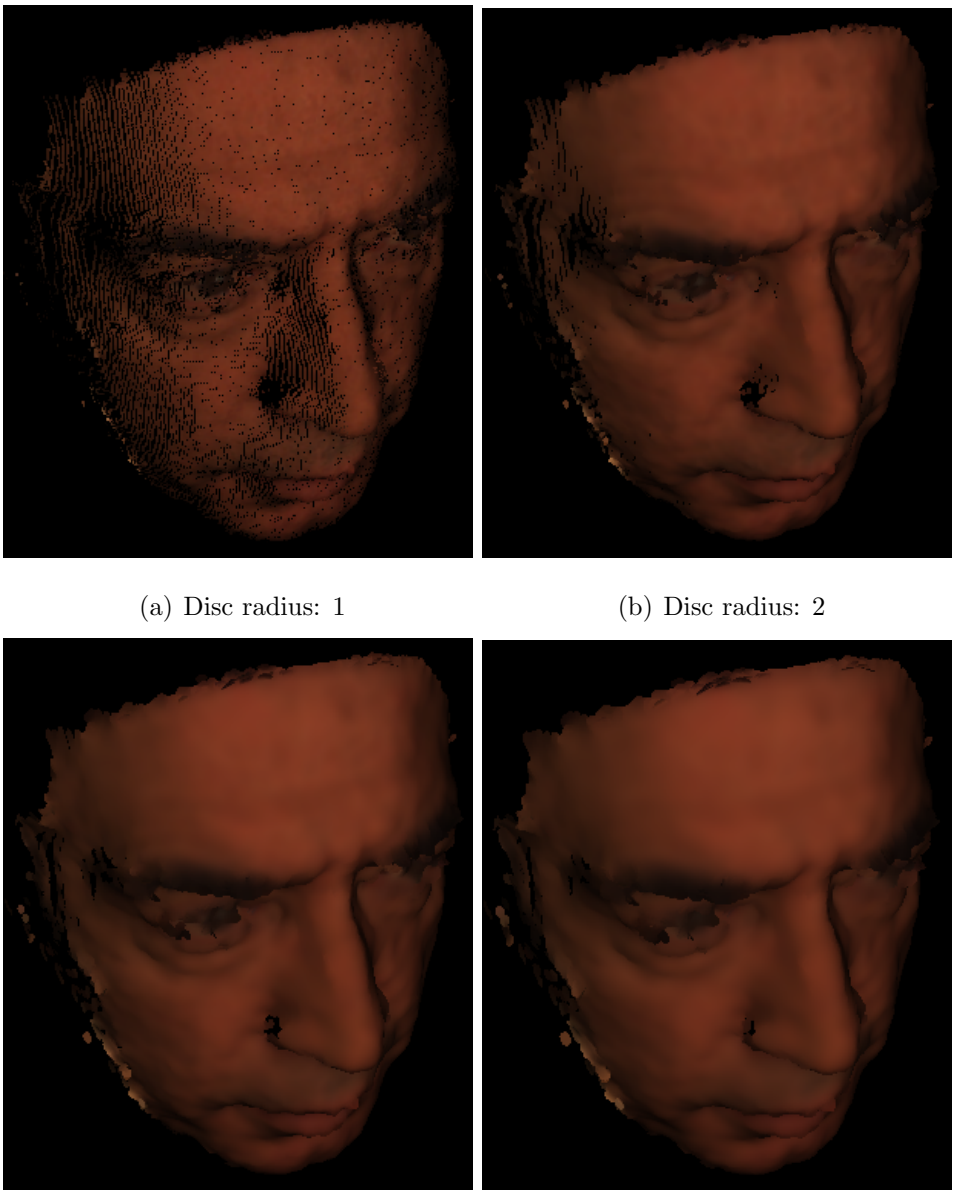
(b) Disc radius: 2

(c) Disc radius: 3

(d) Disc radius: 4

Figure 4.38. Ray tracing face data 2 with different fixed disc radii

Figure 4.39. Ray tracing face data 2 with variable disc radius



(a) Disc radius: 1

(b) Disc radius: 2

(c) Disc radius: 3

(d) Disc radius: 4

Figure 4.40. Ray tracing face data 3 different fixed disc radii

Figure 4.41. Ray tracing face data 3 with variable disc radius

# 5. CONCLUSIONS

In this thesis we have presented methods for directly rendering point sampled geometry. Rendering method is based on ray tracing which uses discs as intersection primitives to represent points. Interpolation is executed for the neighbour points around the intersection which is determined by creating a cylinder.

Important parameters are number of points, disc and cylinder radius. While disc radius directly effects the intersection tests like filling holes between the points, cylinder radius effects the amount of interpolation among the neighbour points.

Also normals can be estimated with least square minimization method if they are not part of the data. Neighborhood radius is also important in normal estimation. Small radiuses are more prone to errors because of insufficient point samples. Also selecting a larger radius than optimum will cause losing details and result in a smooth surface.

Adaptive octree is an ideal data structure for ray tracing point surfaces which are uniform and consists of boundary points. Octree generation and finding neighbor points by using bottom-up traversal method in both ray tracing and normal estimation decreases process duration slightly.

Experimentations with various parameter values showed that increasing disc and cylinder radius makes the image output smoother. On the other hand if the disc radius is not large enough to fill the gaps between points, there will be disorders because of empty areas on the surface. Also if the cylinder radius is not large enough the interpolation will cause discontinuities between illumination of the discs. Choosing varible disc radius improves rendering quality by filling holes in sparse areas and preserving details in denser parts. This approach is also useful for normal estimation where point distribution is non-uniform.

Our experimentations showed that it is not necessary to reconstruct surface from points in order to obtain fast and high quality images. Direct rendering of point clouds using disc intersection based ray tracing is an efficient way to get satisfactory outputs.

Directly ray tracing point clouds gives efficient results especially in complex geometries. In future we plan to develop a more flexible method which can be used for different models by choosing neighbourhood value of k depending on the point cloud characteristics. This will provide full adaptivity for rendering and normal estimation calculations without the need for user input.

# REFERENCES


1. Shu, R., C. Zhou, and M. S. Kankanhalli, "Adaptive Marching Cubes", Technical report, National University of Singapore Institute of Systems Science, Kent Ridge, Singapore 0511, 1995.

2. "Wikipedia, Ray tracing", `http://en.wikipedia.org/`.

3. Gonzlvez, P. R., D. G. Aguilera, and J. G. Lahoz, "From Point Cloud To Surface: Modeling Structures In Laser Scanner Point Clouds", Technical report, Land and Cartography Engineering Department, Univ. of Salamanca, Spain, 2007.

4. Munoz, D., N. Vandapel, and M. Hebert, "Automatic 3-D Point Cloud Classification Of Urban Environments", Technical report, Carnegie Mellon University, Pittsburgh, PA, 15213, 2008.

5. "Stanford University, The Stanford 3D Scanning Repository", `http://graphics.stanford.edu/data/3Dscanrep/`.

6. "Bogazici University, The Bosphorus Database", `http://bosphorus.ee.boun.edu.tr`.

7. Gross, M. and H. Pfister, *Point-based Graphics*, Morgan Kaufmann, 2007, pages 294-313.

8. Pfister, H., M. Zwicker, J. van Baar, and M. Gross, "Surfels: Surface elements as rendering primitives", *Proceedings of SIGGRAPH 2000*, USA, 2000.

9. Grossman, J. P. and W. J. Dally, "Point Sample Rendering", Technical report, Rendering Techniques '98, Springer, Wien, Vienna, Austria, 1998.

10. Alexa, M., J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva, "Computing and Rendering Point Set Surfaces", *Proceedings of IEEE Visualization*,

2001.

11. Adamson, A. and M. Alexa, "Ray Tracing Point Set Surfaces", Technical report, TU Darmstadt, GRIS Fraunhoferstr. 5, 64283 Darmstadt, 2002.

12. Linsen, L., K. Muller, and P. Rosenthal, "Splat-based Ray Tracing of Point Clouds", Technical report, International University Bremen, Bremen, Germany, 2003.

13. Schaufler, G. and H. W. Jensen, "Ray Tracing Point Sampled Geometry", Technical report, Massachusetts Institute of Technology, Stanford University, 2000.

14. Diankov, R. and R. Bajcsy, "Real-Time Adaptive Point Splatting For Noisy Point Clouds", Technical report, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 2001.

15. Gumerov, N. A. and R. Duraiswami, "Fast Radial Basis Function Interpolation Via Preconditioned Krylov Iteration", Technical report, University of Maryland, College Park, USA, 2005.

16. Dey, T. K., G. Li, and J. Sun, "Normal Estimation for Point Clouds: A Comparison Study for a Voronoi Based Method", Technical report, The Ohio State University, Columbus OH, USA, 2005.

17. Hoppe, H., T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Surface reconstruction from unorganized points", *Proceedings of ACM SIGGRAPH*, 1992.

18. Mitra, N. J., A. Nguyen, and L. Guibas, "Estimating Surface Normals in Noisy Point Cloud Data", Technical report, Stanford Graphics Laboratory, Campus Drive, Stanford, USA, 2003.

19. Amenta, N., M. Bern, and M. Kamvysselis, "A new Voronoi-based surface reconstruction algorithm", *Proceedings of SIGGRAPH 98*, July 1998.

20. Revelles, J., C. Urena, and M. Lastra, "An Efficient Parametric Algorithm for Octree Traversal", Technical report, University of Granada, Spain, 1998.

21. Frisken, S. F. and R. N. Perry, "Simple and Efficient Traversal Methods for Quadtrees and Octrees", Technical report, Mitsubishi Electric Research Laboratories, 2001.

22. Arya, S., D. M. Mount, N. S. Netanyahu, R. Silverman, , and A. Y, "An optimal algorithm for approximate nearest neighbor searching Fixed dimensions", *Journal of the ACM*, 1999.