Feature Adaptive Ray Tracing of Subdivision Surfaces

by

Shujian Ke

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2017 by the
Graduate Supervisory Committee:

Ashish Amresh, Chair
John Femiani
Javier Gonzalez-Sanchez

ARIZONA STATE UNIVERSITY

May 2017

ABSTRACT

Subdivision surfaces have gained more and more traction since it became the standard surface representation in the movie industry for many years. And Catmull-Clark subdivision scheme is the most popular one for handling polygonal meshes. After its introduction, Catmull-Clark surfaces have been extended to several eminent ways, including the handling of boundaries, infinitely sharp creases, semi-sharp creases, and hierarchically defined detail. For ray tracing of subdivision surfaces, a common way is to construct spatial bounding volume hierarchies on top of input control mesh. However, a high-level refined subdivision surface not only requires a substantial amount of memory storage, but also causes slow and inefficient ray tracing. In this thesis, it presents a new way to improve the efficiency of ray tracing of subdivision surfaces, while the quality is not as good as general methods.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

In computer graphics, ray tracing is one of the most well-known algorithms for rendering photorealistic pictures. The concept of ray tracing is quit simple. All you need to do is to find the intersection of a line with an object and then to shade the point of intersection. It is very powerful. You can render almost any type of object.

However, a chief drawback of ray tracing is that it requires much more computational resources than the current rasterization based approach for real-time rendering. Over the past decades, tons of researchers have tried to realize real-time ray tracing. Many improved algorithms, libraries and more efficient ray tracing supported hardware have been presented, e.g., Embree Wald *et al.* (2014) and OptiX Parker *et al.* (2010). But despite the impressive algorithms and a brilliant improvement of hardware performance, the current situation is far away from satisfaction. Without doubt, full-featured real-time ray tracing will be a main challenge for many years to come.

And recently, subdivision surfaces have been a new trend for representing a smooth surface. Catmull-Clark subdivision scheme Catmull and Clark (1978) is a famous subdivision scheme that can handle polygonal meshes. It has been the standard surface representation in the movie industry for many years. For example, it has been used in Pixar's short film *Geri's Game* DeRose *et al.* (1998) and in *Toy Story 2*, and in all following feature films from Pixar. Since the introduction in 1978, Catmull-Clark surfaces have been extended to several eminent ways, including the handling of boundaries Nasri (1987), infinitely sharp creases Hoppe *et al.* (1994), semi-sharp creases DeRose *et al.* (1998), and hierarchically defined detail Studios (2005). The semi-sharp creases are especially important as they can handle realistic edges while

using less memory.

For ray tracing of subdivision surfaces, a common way is to construct spatial bounding volume hierarchies on top of input primitives. However, a high-level refined subdivision surface not only requires a substantial amount of memory storage, but also causes slow and inefficient ray tracing. According to recently research on this field, there are two categories for ray tracing of subdivision surfaces: pre-tessellation and direct intersection. The basic idea of pre-tessellation is to use a caching scheme to avoid memory bottleneck. Christensen *et al.* (2003) first introduced using ray differentials to construct a multi-resolution geometry cache (MRGC) for tessellated faces, which combines adaptive multi-resolution techniques and caching to avoid redundant subdivision. Benthin *et al.* (2015) proposed a fixed sized lazy-build evaluation cache specifically designed for ray tracing subdivision surfaces on many-core architecture.

In contrast to pre-tessellation, direct intersection approaches require very little memory for storing additional data. However, a major problem of these approaches is that they cannot directly support displacement mapping, which is very important for industry. Benthin *et al.* (2007) presented the efficient direct ray tracing of subdivision surfaces using packet. Tejima *et al.* (2015) proposed a direct ray tracing method for feature adaptive Catmull-Clark subdivision surfaces using fast and robust Bzier patch intersection.

Chapter 2

GEOMETRY

Linear algebra is the foundation of computer graphics; hence I start with a section on this area to introduce the basic math. It will make you feel comfortable with those ideas in order to understand the rest of this thesis.

2.1 Points, Vectors, Matrices and Normals

In computer graphics world, a vector usually means a direction in three-dimensional space. And a point, on the other hand, is a position in a three-dimensional space. The representation of point and vector are of course similar:

$\mathbf{v} = \langle \mathbf{x}, \mathbf{y}, \mathbf{z} \rangle$,

where x, y, z are real numbers.

Figure 2.1 shows an example of two points and one vector.

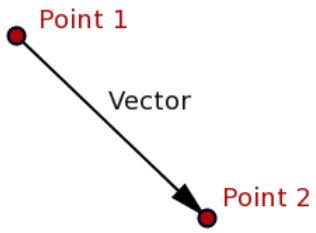

Figure 2.1: Point and Vector Example.

A normal is a term to describe the orientation of a surface of an object at a point on that surface. It is a vector with special meaning.

Matrix is a two-dimensional array of numbers which uses the standard notation

$m \times n$ where m and n are two numbers that represent rows and columns of the matrix. Here is an example of 4×4 matrix:

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix},$$

For computer graphics, we are interested in 3×3 or 4×4 matrices and the following sections will introduce how to use matrices multiplication to do linear transformation.

2.2 Basic Math Operation

A vector not only can be seen as the direction from A to B but also can represent the distance between A and B. This is given by computing the length of a vector:

$$\mathbf{L} = \sqrt{\mathbf{V_x} * \mathbf{V_x} + \mathbf{V_y} * \mathbf{V_y} + \mathbf{V_z} * \mathbf{V_z}},$$

the vectors length is sometimes also called magnitude.

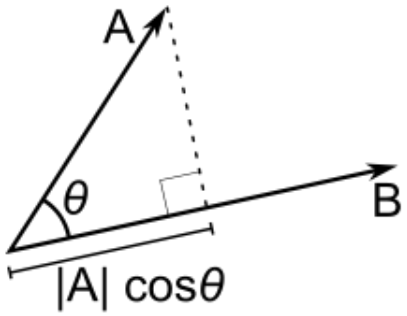

Figure 2.2: Dot Product

Dot product between two vectors is an important concept and common operation in computer graphics because the result of this operation relates to the angle between the two vectors. You can get the dot product by the following equation:

$$A \cdot B = A_x * B_x + A_y * B_y + A_z * B_z,$$

or

$$A \cdot B = \|A\| * \|B\| * \cos\theta,$$

where θ is the angle between A and B.

Figure 2.2 shows an example of dot product.

The cross product is also an operation on two vectors, but unlike the dot product which returns a number, the cross product returns a vector. And this vector is perpendicular to the other two vectors(need picture). You can use the following formula to calculate the cross product:

$A \times B = \langle A_y * B_z - A_z * B_y, A_z * B_x - A_x * B_z, A_x * B_y - A_y * B_x \rangle$

Figure 2.3 shows an example of cross product.

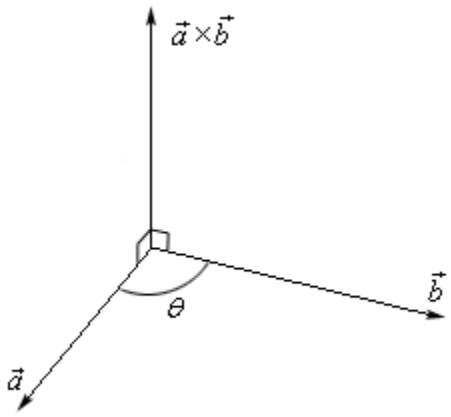

Figure 2.3: Cross Product

2.3 Transformation

In this part, I will start to combine all the things we have learned on points, vectors and matrices together, and I will show you how matrices work. The most important role the matrices play in computer graphics is to do linear transformations, which include scaling, rotation and translation. For all of those transformations, we will need the matrix multiplication.

2.3.1 Multiplication

In mathematics, matrix multiplication is a binary operation that generates a matrix from two matrices, which need to have compatible sizes in order to multiplied with each other. For example, the matrices of size m x p and p x n can be multiplied with each other. And in computer graphics world, we primarily focus on 4×4 matrices.

Here is what this multiplication looks like:

$$[x \quad y \quad z] * \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} x * m_{11} + y * m_{21} + z * m_{31} \\ x * m_{12} + y * m_{22} + z * m_{32} \\ x * m_{13} + y * m_{23} + z * m_{33} \end{bmatrix}$$

2.3.2 Identity Matrix

The identity matrix, or unit matrix, is a $n \times n$ square matrix with ones on the main diagonal and zeros elsewhere. It is denoted by I_n:

$$I_n = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The result of a matrix multiplied by the identity matrix is itself.

2.3.3 Scaling Matrix

If you have noticed the formula for multiplication above, you can see that the coordinates of the point are respectively multiplied by the coefficients m_{11} for x, m_{22} for y, and m_{33} for z. When we set those coefficients to 1 and all the others to 0, we get the identity matrix. However, when these coefficients other than 1, they act as a multiplier on the point's coordinates. The scaling matrix can be written as:

$$S = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix},$$

where S_x, S_y, and S_z are real numbers.

For example, we have a point P at position (1, 3, 5). If we set $S_x = 1$, $S_y = 2$,

and $S_z = 3$, then the new generated point will be at position (1, 6, 15).

2.3.4 Rotation Matrix

For the rotation matrix, I just simply present the three matrices for rotating around three axises, respectively. I won't spend too much time on how they come out and the logic behind those matrices. The only thing we need to know is how to use those matrices to transform the 3D points and vector. Here are what these matrices look like:

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

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

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

Now you know how to rotate points around individual axis, and it is also possible to put three matrices together to create more complex rotations. For example, if you want to rotate a point around the y-axis first, and then the z-axis, we can create two matrices using R_y and R_z and combine them using matrix multiplication $R_y * R_z$ to create a R_{yz} matrix:

$$R_{yz} = R_y * R_z.$$

2.3.5 Translation Matrix

For translation matrix, it is a little bit complex, which needs a 4×4 matrix. The transformation looks like that:

$$P = [x \quad y \quad z \quad 1] * \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} = \begin{bmatrix} x * m_{11} + y * m_{21} + z * m_{31} + m_{41} \\ x * m_{12} + y * m_{22} + z * m_{32} + m_{42} \\ x * m_{13} + y * m_{23} + z * m_{33} + m_{43} \\ x * m_{14} + y * m_{24} + z * m_{34} + m_{44} \end{bmatrix}$$

The upper 3×3 matrix will be used to scale and rotate points, and the fourth row will be used to translate point.

2.3.6 Transpose and Inverse

The transpose of a matrix is an operator which reflects a matrix over its diagonal. It usually denotes like: M^T. And formally, the i th row, j th column element of M^T is the j th row, i th column element of M:

$[M^T]_{ij} = [M]_{ji}$

If M is an $m \times n$ matrix, M_T is an $n \times m$ matrix. Let's take a 4×4 matrix as an example:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$

Then:

$$M^T = \begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix}$$

In linear algebra, an $n \times n$ square matrix M is called invertible if there exists an $n \times n$ square matrix N that:

$AB = BA = I_n,$

where I_n denotes the $n \times n$ identity matrix.

If there is such equation, the matrix N is called the inverse of A, denoted by A^{-1}. Matrix inversion is very useful when you want to move something back into its original place. For example, you have a point P, and a transformation matrix M. You want to move P to a new position P_1 with this matrix M. You will get:

$P_1 = PM,$

then you want to cancel this movement, you can simply multiply an inverse of this matrix:

$P = P_1 M^{-1} = PMM^{-1} = PI = P.$

Chapter 3

RAY TRACING

In computer graphics, ray tracing is one of the most well-known algorithms for rendering photorealistic pictures. The concept of ray tracing is quit simple. All you need to do is to find the intersection of a line with an object and then shading the point of intersection. It is very powerful. You can render almost any type of object.

The following sections briefly introduce the basic concept of ray tracing.

3.1 Viewing

The viewing part is quite of straightforward. It is just like that you stand in front of a window, which has size of $n \times m$ centimeters . And you are viewing the outside from this window. The image you can see is the image our ray tracing algorithm tries to simulate. Figure 3.1 shows how to use mathematical symbol to represent this process

3.2 Intersection

The intersection finds the closest object between the ray and all of the objects in the world. What we need are the ray direction and the ray start position. And what we want are the distance to the closest intersection and the color of the object at the point of intersection. It will return null if the ray doesn't hit any object.

3.2.1 Triangle

For triangle intersection, the Moller-Trumbore ray-triangle intersection algorithm Möller and Trumbore (2005) has been used. It is considered today a fast method

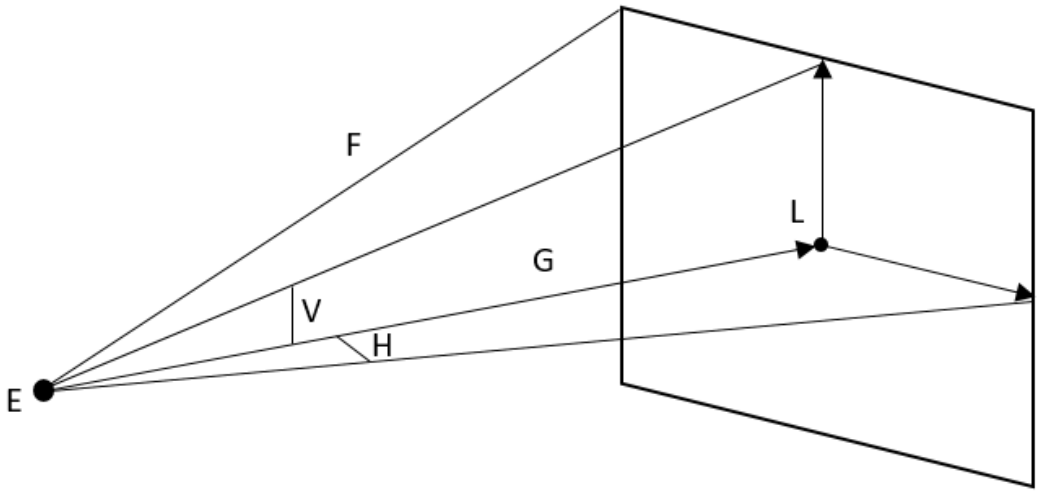

Figure 3.1: Viewing

for calculating the intersection of a ray and a triangle in three dimensions without needing precomputation of the plane equation of the plane containing the triangle. Here just presents the pseudocode of the algorithm and does not go into details about how it works.

Figure 3.2 shows an example of ray triangle intersection.

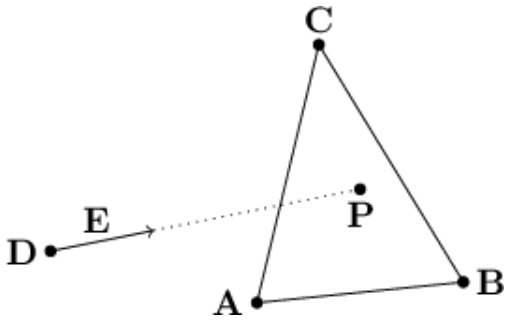

Figure 3.2: Ray Triangle Intersection

```
bool   ray_triangle_intersection(
    const Vec3f v0,
    const Vec3f v1,
```

```
    const Vec3f v2 // Triangle Vertices
    const Vec3f rayorig,
    const Vec3f raydir, // Ray info
    float &t, float, &u, float &v // Output value) {

    // two edges from v0
    Vec3f e0 = v1 - v0;
    Vec3f e1 = v2 - v0;
    // calculate determinant
    Vec3f P = cross_product(raydir, e1);
    float det = dot_prodcut(P, e0);
    // if det is near zero, ray lies in the plane of triangle
    // or ray is parallel to the plane of triangle
    // if det is negative, no culling
    If ( det > -EPSILON && det < EPSILON ) return false;
    float inv_det = 1.0f / det;
    Vec3f T = rayorig - v0;
    // calculate u
    u = dot_product(T, P) * inv_det;
    // check if it lies outside the triangle
    If ( u < 0.f || u > 1.f ) return false;
    Vec3f Q = dot_product(T, e0);
    v = dot_product(D, Q) * inv_det;
    // check if it lies outside the triangle
    If ( v < 0.f || u + v > 1.f ) return false;
    // distance between ray origin and triangle
    t = dot_product(e1, Q) * inv_det;
    return true;
}
```

3.2.2 Axis Aligned Bounding Box

The box is aligned with the axis of the coordinate system is called axis aligned box. And it is also called bounding box because it is often used for a bounding box as an accelerated structure. Williams *et al.* (2005) have presented an efficient and robust ray-box intersection algorithm. You can check the details about their algorithm and here just shows the pseudocode:

```
class Ray {
public:
    Ray(Vector3 &o, Vector3 &d) {
        origin = o;
```

```
        direction = d;
        inv_direction = Vector3(1/d.x(), 1/d.y(), 1/d.z());
        sign[0] = (inv_direction.x() < 0);
        sign[1] = (inv_direction.y() < 0);
        sign[2] = (inv_direction.z() < 0);
    }
    Vector3 origin;
    Vector3 direction;
    Vector3 inv_direction;
    int sign[3];
};

// Optimized method
bool Box::intersect(const Ray &r, float t0, float t1) const {

    float tmin, tmax, tymin, tymax, tzmin, tzmax;
    tmin = (bounds[r.sign[0]].x() - r.origin.x())
        * r.inv_direction.x();
    tmax = (bounds[1-r.sign[0]].x() - r.origin.x())
        * r.inv_direction.x();
    tymin = (bounds[r.sign[1]].y() - r.origin.y())
        * r.inv_direction.y();
    tymax = (bounds[1-r.sign[1]].y() - r.origin.y())
        * r.inv_direction.y();
    if ( (tmin > tymax) || (tymin > tmax) ) return false;
    if (tymin > tmin) tmin = tymin;
    if (tymax < tmax) tmax = tymax;
    tzmin = (bounds[r.sign[2]].z() - r.origin.z())
        * r.inv_direction.z();
    tzmax = (bounds[1-r.sign[2]].z() - r.origin.z())
        * r.inv_direction.z();
    if ( (tmin > tzmax) || (tzmin > tmax) ) return false;
    if (tzmin > tmin) tmin = tzmin;
    if (tzmax < tmax) tmax = tzmax;
    return ( (tmin < t1) && (tmax > t0) );
}
```

Figure 3.3 shows an example of ray box intersection.

3.3 Shading

Shading is to calculate the color of a point on an objects face. For a basic ray tracer, ambient, diffuse, specular, reflection and refraction are included, and all lights are white light for simplifying the calculation. The shading calculation needs the

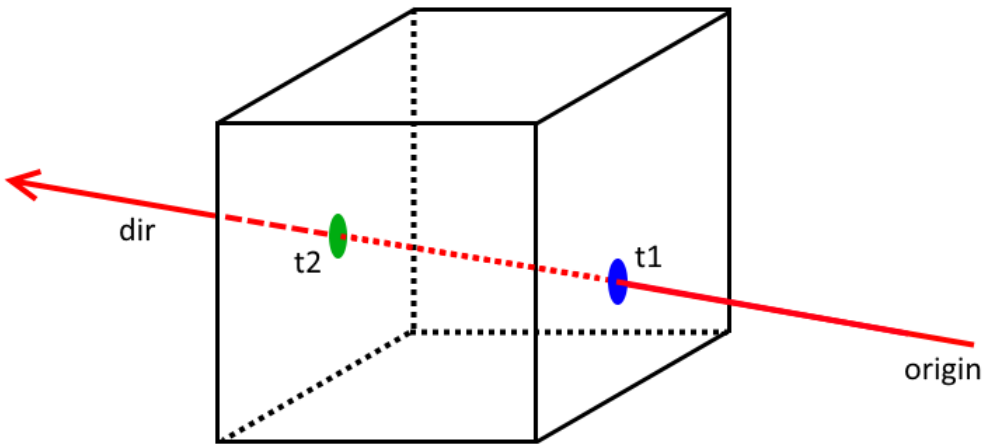

Figure 3.3: Ray Box Intersection

point of intersection on the object, the surface property of intersected object, the ray origin and direction, distance to the intersection, and the normal at the point. For the surface property of an object, there are surface ambient color, surface diffuse color, surface specular color, reflection factor, refraction factor, etc.

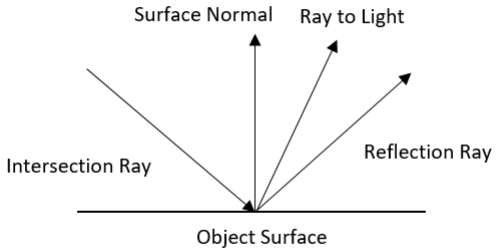

Figure 3.4: Surface Property

3.3.1 Ambient, Diffuse, and Specular

The ray showing in Figure 3.4 and the object surface property determine the color of the point being shaded. The intersection ray and the surface normal determine

the reflection ray. The formula is that the angle between the intersection ray and the surface normal equals to the angle between the surface normal and the reflection ray.

In the first place, ambient color of this object is the initial color for the point being intersected. For each light source, the shading method will compute the diffuse color and specular color and add them to the color of the point being intersected. At each iteration, the diffuse color and specular color are added to the color of the surface. The color of the surface becomes brighter as each iteration is processed.

Figure 3.5 shows an example of specular shading.

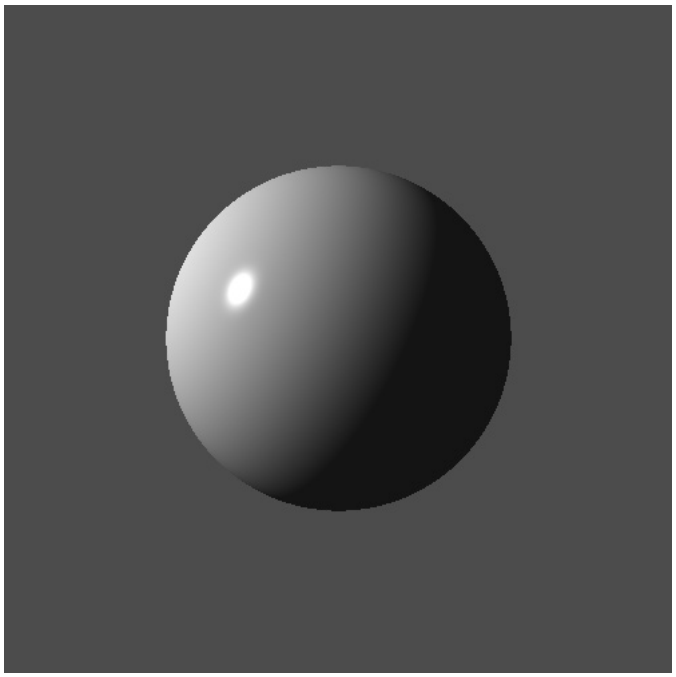

Figure 3.5: Specular Example

3.3.2 Shadow, Reflection, and Transparency

Shadows can give very impressive depth cues and make realistic image. Shading means to find the color at a point on one object's surface, while shadows refers to blocking the light which would have fallen on the surface.

It is very easy to integrate shadows into the ray tracer. For each intersected point, if there is at least one object stay between the point and the light source, this light source does not contribute to the shading process of this point.

For reflection and transparency, the Figure 3.6 shows the relationship with original ray, reflection ray and transparency ray.

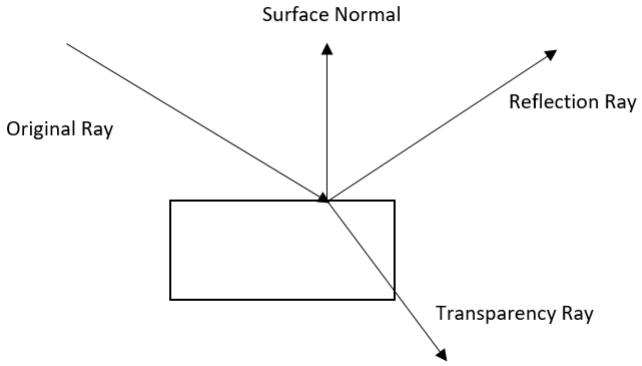

Figure 3.6: Reflection and Transparency

Unlike the original ray whose start position is at the camera, the start position of both reflection ray and transparency ray is at the point being intersected. The formula for reflection color looks like it:

$Color_p = Color_p + R * Color_r$

where

$Color_p$ is the color already calculated for the intersected point hit by original ray.

R is the reflect factor of the surface being intersected, which ranges from 0.0 for nonreflecting to 1.0 for perfect reflecting.

$Color_r$ is the color of the intersected point hit by reflection ray. It will use background color, which is defined by user, if the reflection ray didnt hit any object.

Figure 3.7 shows an example of reflection. Transparency works in a similar way

to reflection.

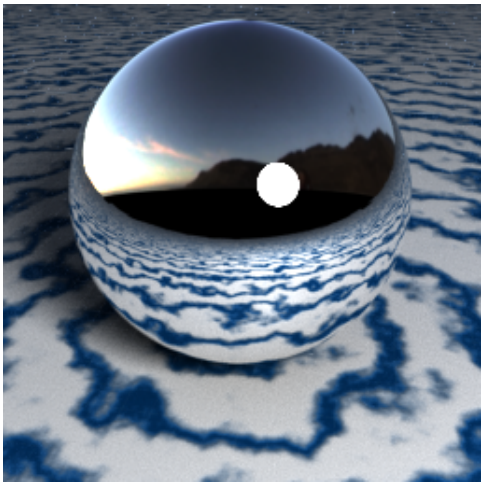

Figure 3.7: Reflection Example

3.4 Space Subdivision and Bounding Volume

Checking every ray for intersection with every triangle of objects becomes extremely expensive for a scene with tons of objects. One way to deal with this problem is to put bounding volume around complex objects. If the ray hit the bounding volume, it continues checking the intersection of the object within this bounding volume. There are some common types of bounding volume.

A bounding box is a cuboid that contains the object. In many cases, the bounding box is aligned with the axes of the coordinate system, which is known as an Axis-Aligned Bounding Box (AABB). And an Oriented Bounding Box (OBB) is called when it refers an arbitrary bounding box. AABBs are much easier to test for intersection than OBBs, but it cant deal with the situation that when the model is rotated, which requires it to be recomputed.

Axis-Aligned Bounding Box is used here and the intersection of ray-box is introduced above in intersection section. There are other types of bounding volume like

bounding cylinder, bounding sphere, minimum bounding rectangle, etc, which are not introduced in detail due to the limited space.

Figure 3.8 shows an example of bounding box.

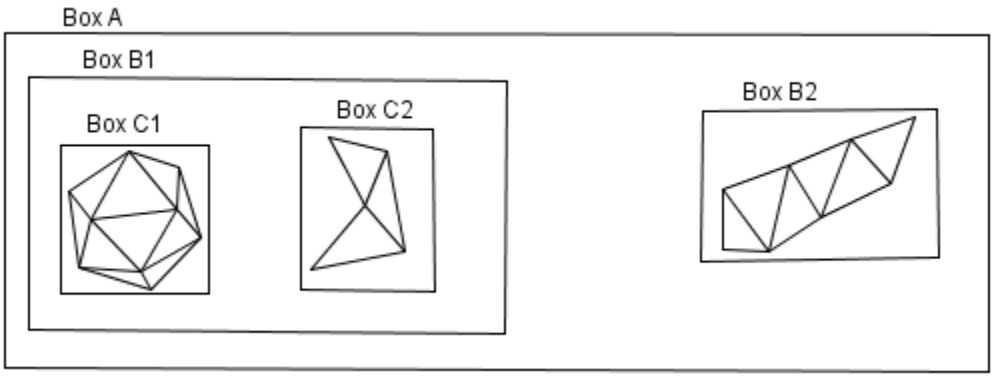

Figure 3.8: Bounding Box Example

Algorithms for the generation of bounding volumes led to the idea of space subdivision. Even though each object has its bounding volume, it is also inefficient to check the intersection with all objects in the scene. The idea of space subdivision is that objects are checked for intersection with a ray only when the object and the ray are in the same region of the space.

A k-dimensional (k-d) tree Bentley (1975) is used here as a space-partitioning data structure. The k-d tree is a binary tree, and the k value depends on the objects it contains. And here the k value equals to three because the objects in real world are three dimensional objects. Each node in k-d tree has a list of objects, the bounding volume including all of those objects, left child node and right child node.

Figure 3.9 shows an example of k-d tree.

The algorithm for building a k-d tree is that on the top node, it has all the objects in the scene and a bounding volume as a space that includes all objects. Then it recursively subdivides the node space into two parts, known as half-spaces. The basic

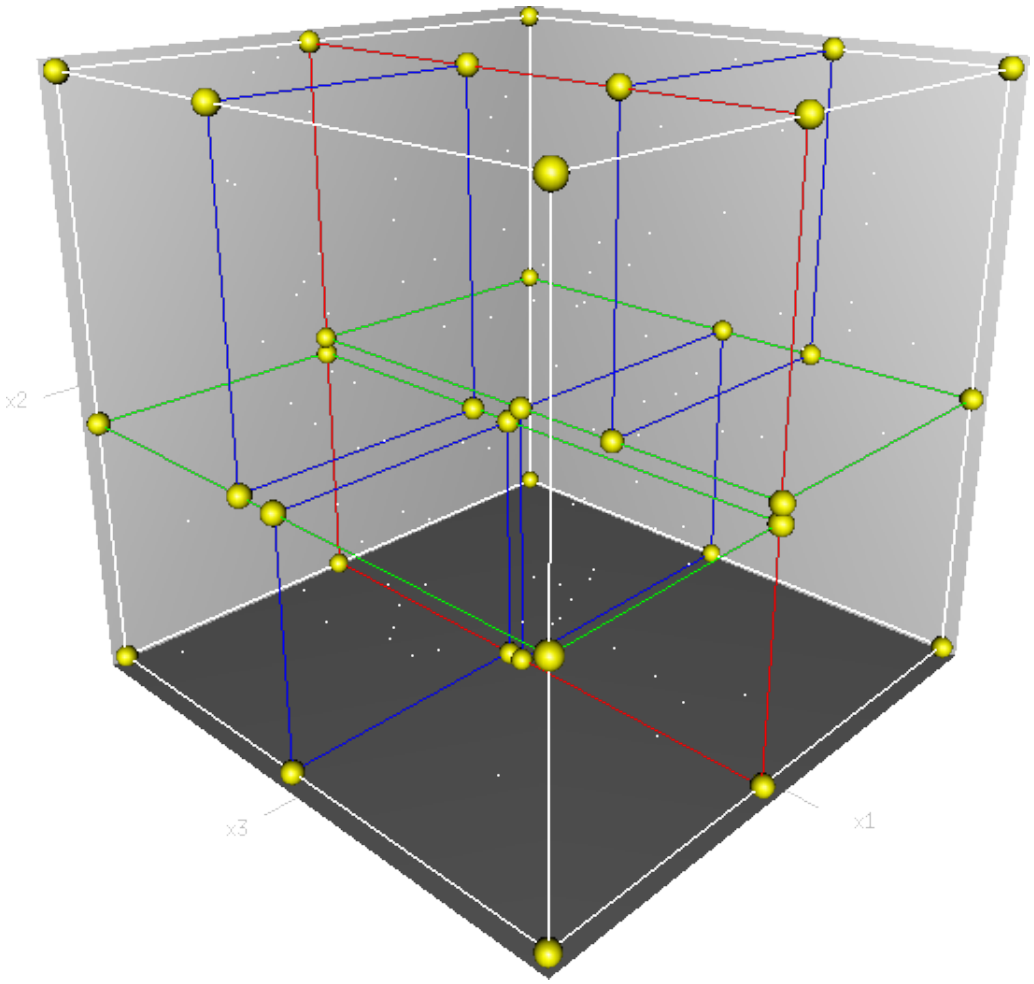

Figure 3.9: K-D Tree Example

rule for subdividing the space is that every node in the tree is associated with one of the k-dimensions, and the node space is subdivided according to that dimensions axis. For example, in 3-D world, if x axis is chosen, the objects whose center points x coordinate is smaller than the x coordinate of the average of center points of all objects go to the left child node, and others go to the right child node. Then how to choose which axis to subdivide is the problem. In this case, a simplest method is used that it depends on the depth. For example, for depth 1, 2, 3, 4, it chooses x, y, z, and x respectively because three dimension is used here. Below is the pseudocode

for the data structure:

```
struct  KDNode{
    AABB box;
    KDNode* left_child;
    KDNode* right_child;
    vector<object*> objs;
}
```

and the pseudocode for build k-d tree:

```
void build_kdtree(vector<objs *> objs, int depth) {
    if (depth > max_depth || objs.size() == 1) return;
    node.box = build_AABB(objs);
    node.objs = objs;
    int axis = depth mod 3;
    calculate average by axis from objs;
    node.left_child = build_kdtree(objects.axis < average,
        depth + 1);
    node.left_child = build_kdtree(objects.axis >= average,
        depth + 1);
}
```

Chapter 4

SUBDIVISION SURFACES

Subdivision surfaces are very useful in defining smooth, continuous, crackless surfaces. In addition, it also provides infinite level of detail, which means you can generate as many triangles as you wish. Figure 4.1 shows an example of a surface being subdivided. There are many other advantages that the subdivision rules are very simple and the implementation is straightforward.

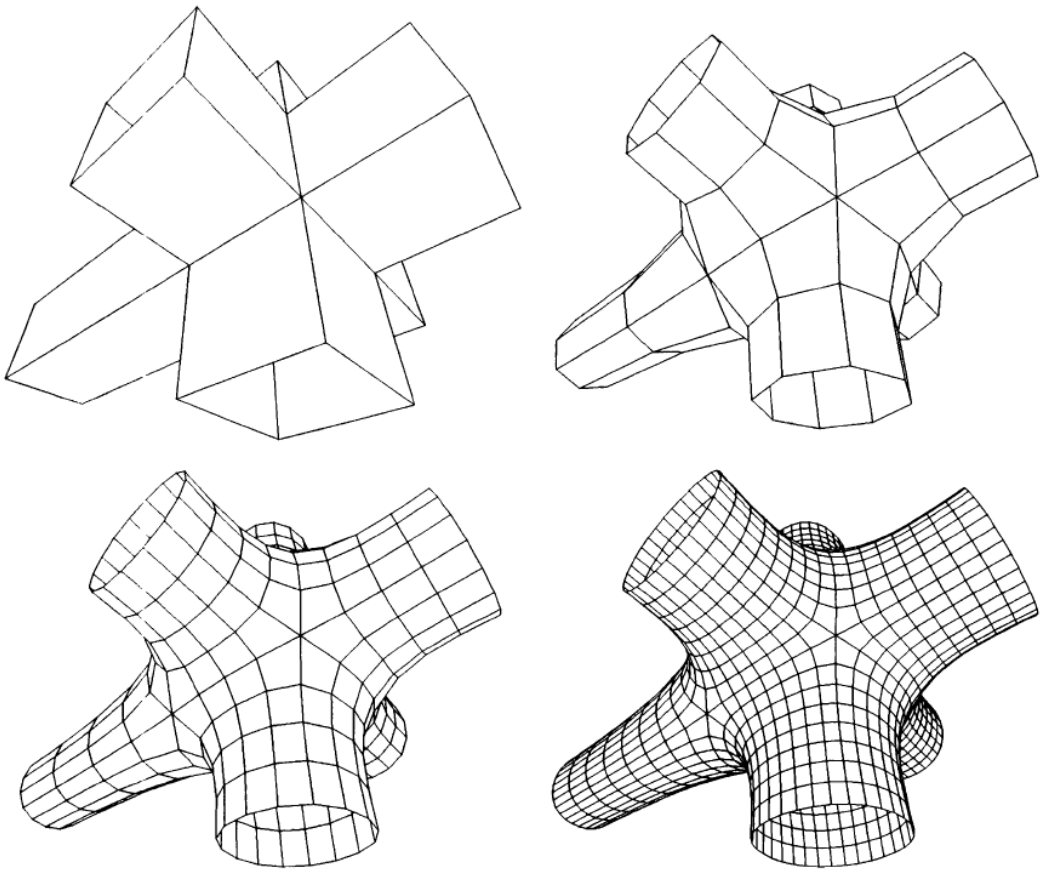

Figure 4.1: Subdivision

There is a two-phase process for the subdivision of surfaces. The first phase, called *refinement phase*, starts with a coarse input mesh, called the *control mesh*. It generates new vertices and connects them to create new, smaller triangles. The second phase, called smoothing phase, computes the new positions for some of all vertices in the mesh. Figure 4.2 shows an example of the two-phase process. This is how a subdivision scheme works by these two phases. For different subdivision schemes, in the first phase, the control mesh can be split in different ways, and in the second phase, the subdivision rules for placing new positions for the vertices will give different characteristics like the level of continuity.

A subdivision scheme can be differentiated by whether it is stationary, where it is uniform, and whether it is triangle-based or polygon-based. A scheme uses the same rules at every subdivision step is a stationary scheme, while a nonstationary may change the rules depending on the current step. A uniform scheme use the same rules for every vertex and edge, while a nonuniform scheme may use different rules for different vertices and edges. A triangle-based scheme only works on triangle meshes which only generates triangles, while a polygon-based schemes operates on arbitrary polygons.

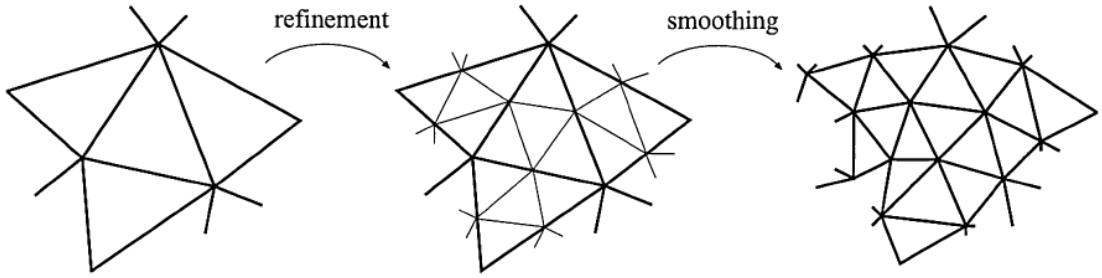

Figure 4.2: Two-Phase Process

In this chapter, it will first introduce some basic concepts about curves and surfaces, like Bézier and B-spline. Then several subdivision schemes will be presented.

4.1 Curve and Surface Representation

A curve in 3D can be expressed in the parametric form as:

$x = x(t)$, $y = y(t)$, $z = z(t)$, $t_1 \leq t \leq t_2$,

where the point (x, y, z) of the curve can be expressed as functions of a parameter t within a closed interval $t_1 \leq t \leq t_2$.

Similar to the curve, a parametric representation of the surface patches can be expressed as functions of the parameters u and v in a closed rectangle:

$x = x(u, v)$, $y = y(u, v)$, $z = z(u, v)$, $u_1 \leq u \leq u_2$, $v_1 \leq v \leq v_2$.

The parametric representation is mainly used, and there are other two ways to represent surfaces: implicit and explicit methods.

An implicit surface is defined as:

$f(x, y, z) = 0$.

And if an implicit surface can be solved for one of the variables as a function of the other two, the explicit surface is obtained:

$z = F(x, y)$.

There are many disadvantages and advantages of different methods of surface representation. And in this thesis, I won't go into detail about the difference among those representation methods, and I will focus on parametric form throughout this chapter.

4.2 Bézier Curves and Surfaces

Before introducing the Bézier curves/surfaces, it is important to know Bernstein polynomials first. The Bernstein polynomials are defined as:

$B_{i,n}(t) = \frac{n!}{i!\,(n-i)!}(1-t)^{n-i}t^i, \qquad i = 0,...,n.$

The Bernstein polynomials have several important properties:

- Non-negativity: $B_{i,n} \geq 0, \quad 0 \leq t \leq 1, \;\; i = 0, ..., n$
- Partition of unity: $\sum_{i=0}^{n} B_{i,n}(t) = 1$
- Symmetry: $B_{i,n}(t) = B_{n-i,n}(1-t)$.

Figure 4.3 shows an example of Bernstein polynomials with a degree equals to three. The x axis stands for t, and y axis is the value of Bernstein polynomials.

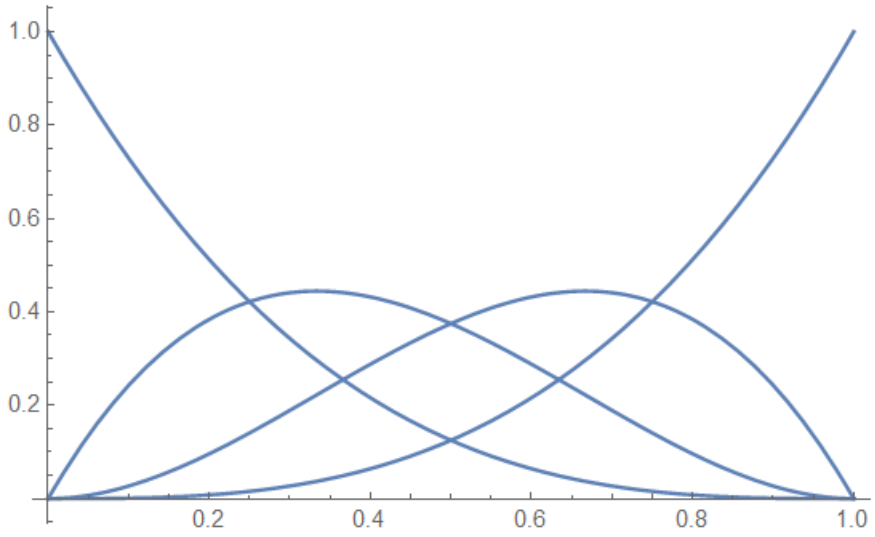

Figure 4.3: Bernstein Polynomials with Degree N = 3

A Bézier curve is a parametric curve that uses Bernstein polynomials as a basis. A degree of n Bézier curve is defined as:

$x(t) = \sum_{i=0}^{n} b_i B_{i,n}(t)$,

where the coefficients b_i are the Bézier control points which form the Bézier polygon.

Figure 4.4 shows an example of cubic Bézier curve.

And a Bézier surface is formed by the similar formula:

$x(u, v) = \sum_{i=0}^{m} \sum_{j=0}^{n} b_{ij} B_{i,m}(u) B_{j,n}(v), \quad 0 \leq u, v \leq 1.$

Figure 4.5 shows an example of cubic Bézier patch.

Bézier curves and surfaces have the following properties:

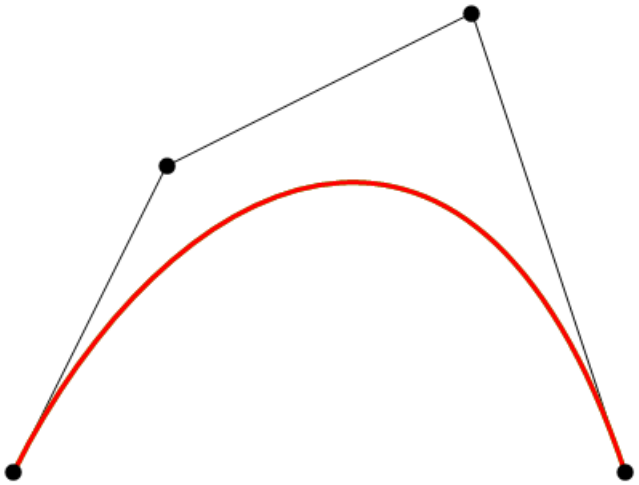

Figure 4.4: Bézier Curve

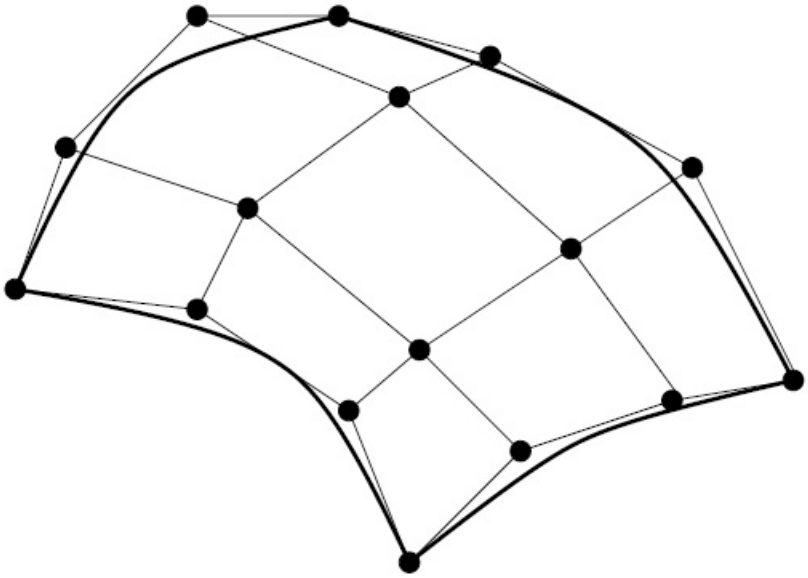

Figure 4.5: Bézier Patch

• Geometry invariance property: the invariance of Bézier curve and surface under rotation and translation of its control points is assured by the Partition of unity property of Bernstein polynomials.

• End points geometric property: the first and last control points are the endpoints of the curve. It is the same like $b_0 = x(0), b_n = x(1)$.

• Convex hull property: the Bézier curves and surfaces are in the convex hull of the control polygon.

4.3 B-spline Curves and Surfaces

The Bézier curves and surfaces have some disadvantages. First, the degree is directly related to the number of control points. Therefore, it requires a high degree in order to have a more complexed curve or surface. Second, changing any control point changes the shape of the entire curve or surface. Those disadvantages lead to the introduction of B-spline curve and surface.

B-spline curves is defined by piecewise polynomial basis functions. A set of non-descending breaking points defines a knot vector:

$T = (t_0, t_1, ..., t_{K-1}), \quad t_0 \leq t_1 \leq t_2 \leq ... \leq t_{K-1}.$

And given a know vector T, the B-spline basis functions, $N_i^k(t)$, are defined as:

$$N_i^1(t) = \begin{cases} 1 & \text{for } t_i \leq t < t_{i+1} \\ 0 & otherwise \end{cases},$$

for $k = 1$, and

$N_i^k = \frac{t - t_i}{t_{i+k-1} - t_i} N_i^{k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1}^{k-1}(t),$

for $k > 1$ and $i = 0, 1, ..., n$.

A B-spline curves is defined as follow:

$x(t) = \sum_{D-1}^{i=0} d_i N_i^n(t),$

where d_i is called de Boor points or control points, and $D = K - n + 1$.

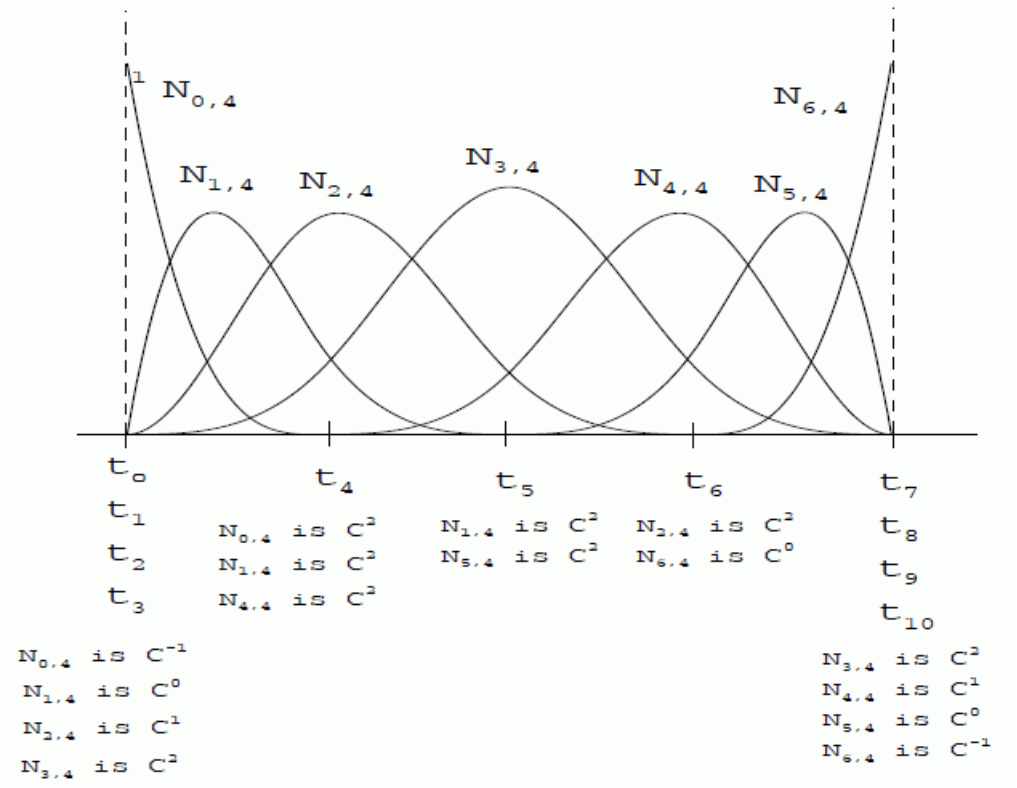

Figure 4.6: B-Spline Basis Functions

Parameter values within range t_{n-1} and t_{K-n} used for evaluating a B-spline curve. And Figure 4.7 shows an example of B-spline curve.

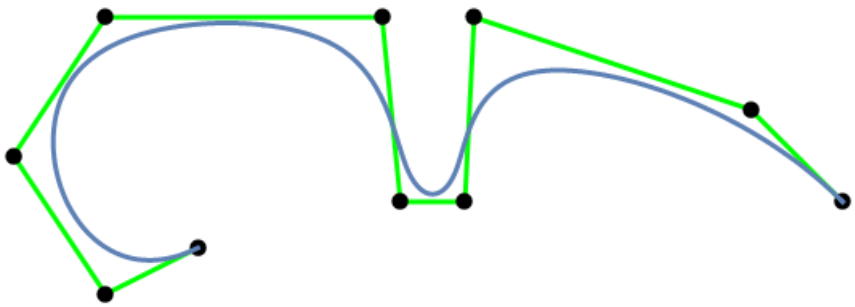

Figure 4.7: B-Spline Curve

B-spline surface $x(u, v)$ is defined as:

$$x(u,v) = \begin{bmatrix} N_0^m(u) & \cdots & N_{D-1}^m(u) \end{bmatrix} \begin{bmatrix} d_{0,0} & \cdots & d_{0,E-1} \\ \vdots & & \vdots \\ d_{D-1,0} & \cdots & d_{D-1,E-1} \end{bmatrix} \begin{bmatrix} N_0^n(v) \\ \vdots \\ N_{E-1}^n(v) \end{bmatrix},$$

where D is defined by a set of knot vector and control points along the u direction, while E is defined along v direction.

For information about this field, please refers to Farin (2002).

4.4 Catmull-Clark Subdivision

Catmull-Clark subdivision scheme is a famous subdivision scheme that can handle polygonal meshes. It is a generalization of a recursive bicubic B-spline patch subdivision algorithm. It has been the standard surface representation in the movie industry for many years. The example of Catmull-Clark subdivision is shown in Figure 4.8. It takes a control mesh as input and generate a new refined mesh containing more faces, edges, and vertices. A smooth limit surface is obtained by repeating this process.

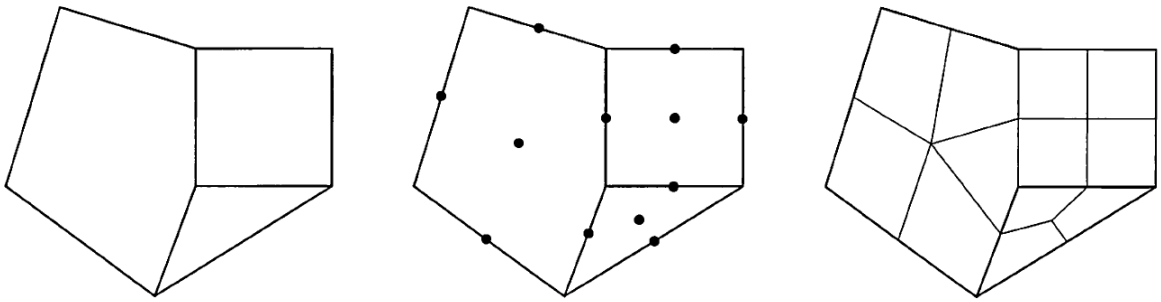

Figure 4.8: Catmull Clark 1

The subdivision rules of this algorithm are quite simple. The following notations are used to describe this algorithm. A vertex $\mathbf{v^k}$ with n surrounding edge points $\mathbf{e}_i^k$, where $i = 0 \ ... \ n-1$. The k means the subdivision depth. For example, the control mesh has all vertices with $k = 0$. After it executes the subdivision rules for

the control mesh once, all vertices has $k = 1$. Then for each face, a new face point $\mathbf{f}^{k+1}$ is generated at the center of the face. Given this, the rules for vertices and edges are:

— $v^{k+1} = \frac{n-2}{n} v^k + \frac{1}{n^2} \sum_{j=0}^{n-1} e_j^k + \frac{1}{n^2} \sum_{j=0}^{n-1} f_j^{k+1}$

— $e_j^{k+1} = \frac{v^k + e_j^k + f_{j-1}^{k+1} + f_j^{k+1}}{4}$

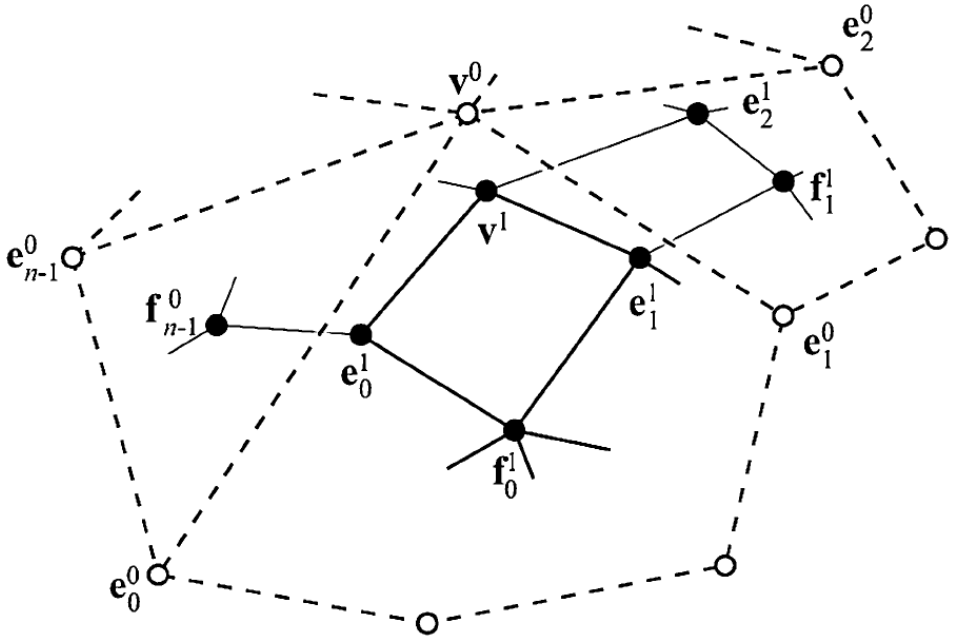

Figure 4.9: Catmull Clark 2

4.5 Feature Adaptive Subdivision Surfaces

Catmull-Clark subdivision surfaces are a generalization of regular bicubic B-splines patches. Nasri (1987) introduced boundary subdivision rules as an extension for Catmull-Clark scheme. Hoppe *et al.* (1994) applied those additional rules to the edges of non-boundary patches to generate creases. In addition, DeRose *et al.* (1998) proposed a semi-sharp crease feature that allow modeling edges with tighter radius of curvature while using less memory.

4.5.1 Subdivision Rules

According to DeRose *et al.* (1998), a crease is defined by adding a sharpness to an edge. Subdividing a sharp edge creates two child edges which have the sharpness value of their parent minus one. A vertex v_j is described as a crease vertex when it connects with exactly two crease edges e_j and e_k. The following rules are used (crease rule):

— $e_j^{i+1} = \frac{1}{2}(v^i + e_j^i)$

— $v_j^{i+1} = \frac{1}{8}(e_j^i + 6v^i + e_k^i)$

If a vertex connects to three or more sharp edges or located on a corner then it has $v^{i+1} = v^i$ (corner rule).

Figure 4.10 shows an example of a semi-sharp crease.

Nießner *et al.* (2012) provided modified scheme in order to deal with fractional smoothness and propagate sharpness properly where $e.s$ is the value of sharpness of an edge and smooth rule means Catmull-Clark rule:

— Face points are always at the center of the surrounding points

— e with $e.s = 0 \rightarrow$ smooth rule

— e with $e.s \geq 1 \rightarrow$ crease rule

— e with $0 \leq e.s \leq 1 \rightarrow (1 - e.s) \cdot e_{smooth} + e.s \cdot e_{crease}$

Then they introduced the vertex sharpness to handle vertices, where $v.s$ is the average of all incident edge sharpnesses and k is the number of edges around vertex v with $e.s > 0$:

— v with $k < 2 \rightarrow$ smooth rule

— v with $k > 2 \wedge v.s \geq 1.0 \rightarrow$ corner rule

— v with $k > 2 \wedge 0 \leq v.s \leq 1 \rightarrow (1 - v.s) \cdot v_{smooth} + v.s \cdot v_{corner}$

— v with $k = 2 \wedge v.s \geq 1.0 \rightarrow creaserule$

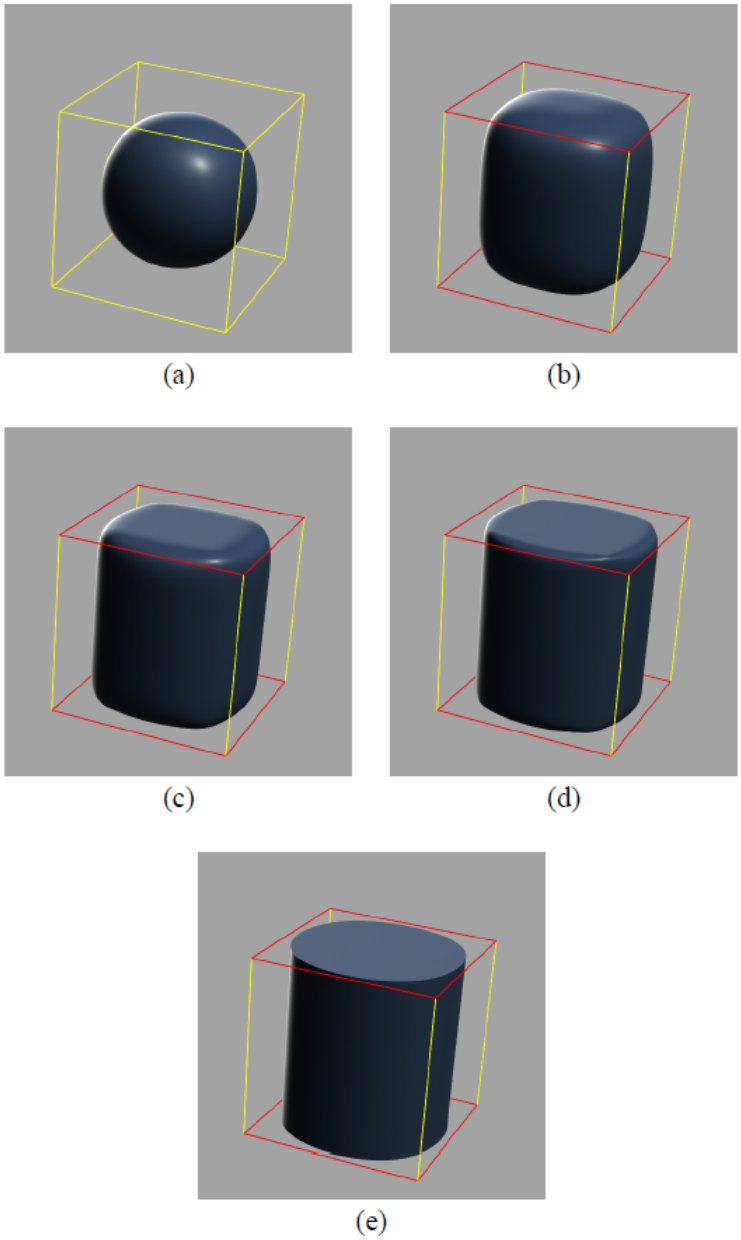

Figure 4.10: The Crease Sharpnesses for (a), (b), (c), (d), and (e) Are 0, 1, 2, 3, and Infinite, Respectively

— v with $k = 2 \wedge 0 \leq v.s \leq 1 \rightarrow (1 - v.s) \cdot v_{smooth} + v.s \cdot v_{crease}$

The semi-sharp creases are especially important as they can handle realistic edges while using less memory. For example, the model in Figure 4.12 was created with

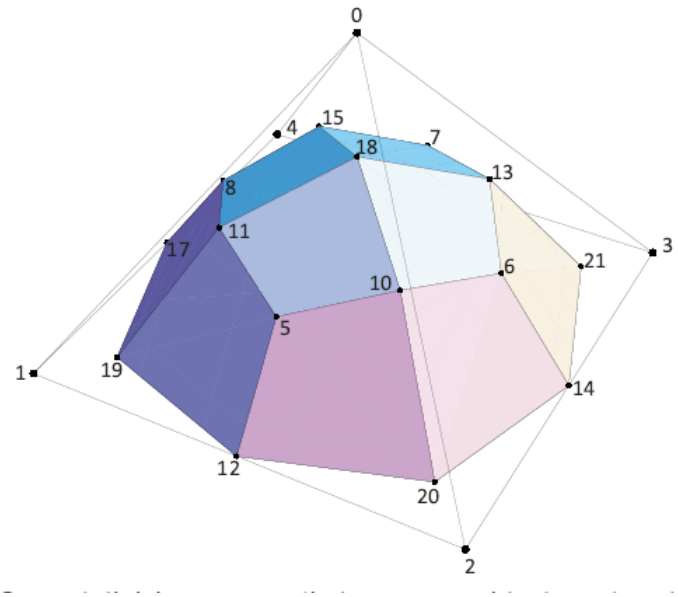

Figure 4.11: One Subdivision Step Spplied on a Pyramid Where the Edges of the Base Plane Are Tagged Sharp.

semi-sharp creases which only requires a few tag data per creased edge. Without semi-sharp creases, it would require a base mesh with more vertices, faces, and edges to achieve a similar shape.

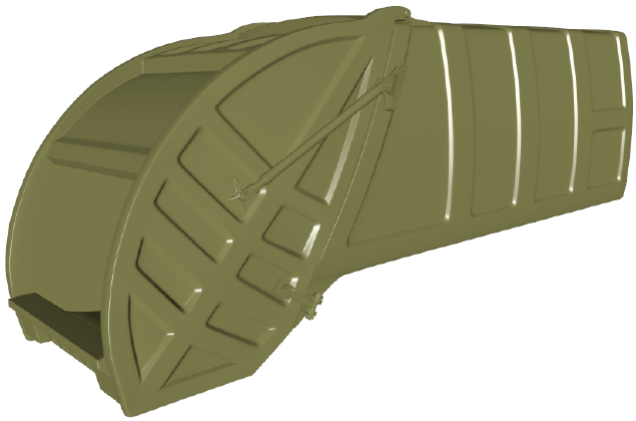

Figure 4.12: Disney Model

4.5.2 Feature Adaptive Subdivision

As mentioned before, the limit surface of Catmull-Clark subdivision can be described by a collection of regular bicubic B-spline patchesm, where there are infinitely many patches around extraordinary vertices and creases as shown in 4.13.

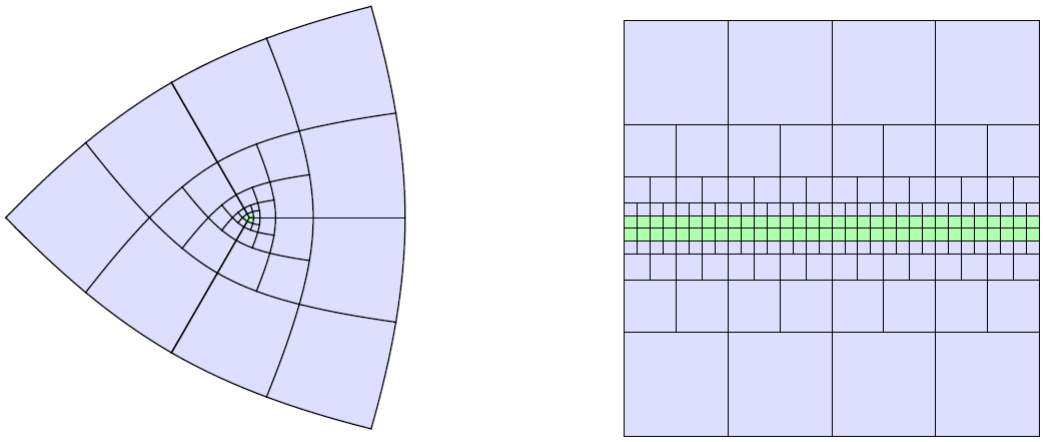

Figure 4.13: Bicubic Patches Around EV(left) and Creases(right)

In feature adaptive subdivision, there are two types of faces. A face is regular only if it is a quad with all regular vertices, if none of its edges or vertices are tagged as sharp, and there are no hierarchical edits that would influence the shape of the limit patch. In all other cases the face is recognized as irregular.

The feature adaptive subdivision identifies regular faces at each stage of subdivision, rendering them directly as bicubic B-splines using hardware tessellation. Irregular faces are refined, and the process repeats at the next finer level.

4.5.3 Patch Construction

For each subdivision level, there are two kinds of patches: full patches and transition patches. Full patches are patches that share edges with patches of the same

subdivision level. Regular full patches are passed throught the hardware tessellation pipeline and rendered as bicubic B-splines, and irregular patches are refined at next level.

Transition patches are patches that are adjacent to a patch from the next subdivision level. They are required to be regular. And where faces with different subdivision levels meet, T-junction occur. In order to obtain crack-free renderings, special methods should be taken. One way to do that is to use compatible power of two tess factors. However, this will be a severe limitation for the tessellation unit. In order to avoid this limitation, Nießner *et al.* (2012) proposed a method to split each transition patch into several subpatches by analyzing the arrangement of the adjacent patches. According to their analysis, there are only 5 distinct cases as shown in 4.14.

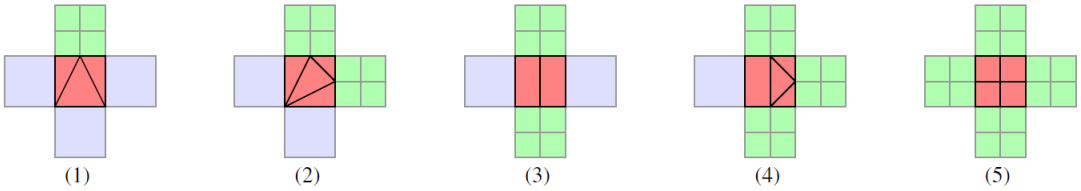

Figure 4.14: Five Transition Patch Cases

4.6 Adaptive Quadtrees Subdivision Surfaces

Even though the feature adaptive subdivision can subdivide irregular faces, and those with special features, to yield multiple patches that can be directly processed by tessellation hardware, which yielding an accurate surface, it has several disadvantages. First, each irregular input face maps to multiple tessellator input primitives, the number of which correlates with the subdivision level at run time. This complicates the application logic used to assign tessellation rates and submit primitives. Second, where faces with different subdivision levels meet, T-junctions occur, which must be fixed with the introduction of transition patches, which makes the distribution of

vertices on the tessellated limit surface less uniform (Figure 4.15).

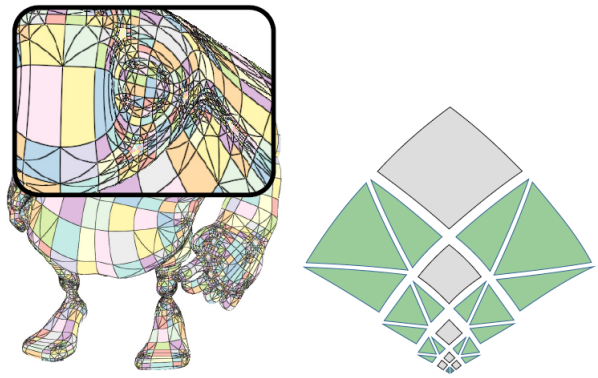

Figure 4.15: Transition Patches Arrangement

Brainerd *et al.* (2016) proposed a novel end-to-end subdivision algorithm that incorporates subdivision surface tessellation into the graphics pipeline. The most important method in their algorithm is to submit one tessellator primitive per input quadrilateral face, as in approximate schemes, but to pre-compute sufficient data to perform accurate surface evaluations equivalent to feature adaptive subdivision. Compared to the state of the art, our approach is both simpler and faster, and integrates with existing vertex and fragment shaders used for polygonal models.

4.6.1 Overview

The input of their algorithm is a Catmull-Clark base mesh, which is defined by its topology, feature tags, and a set of base vertices, which supports all common extensions of Catmull-Clark subdivision, including boundaries and both hard and semi-sharp crease tags on vertices and edges.

When a base mesh is first loaded, we process each face (and its 1-ring neighborhood) to create (or reuse) a subdivision plan. The subdivision plan is a data structure that represents a feature-adaptive subdivision hierarchy for the face, down to some

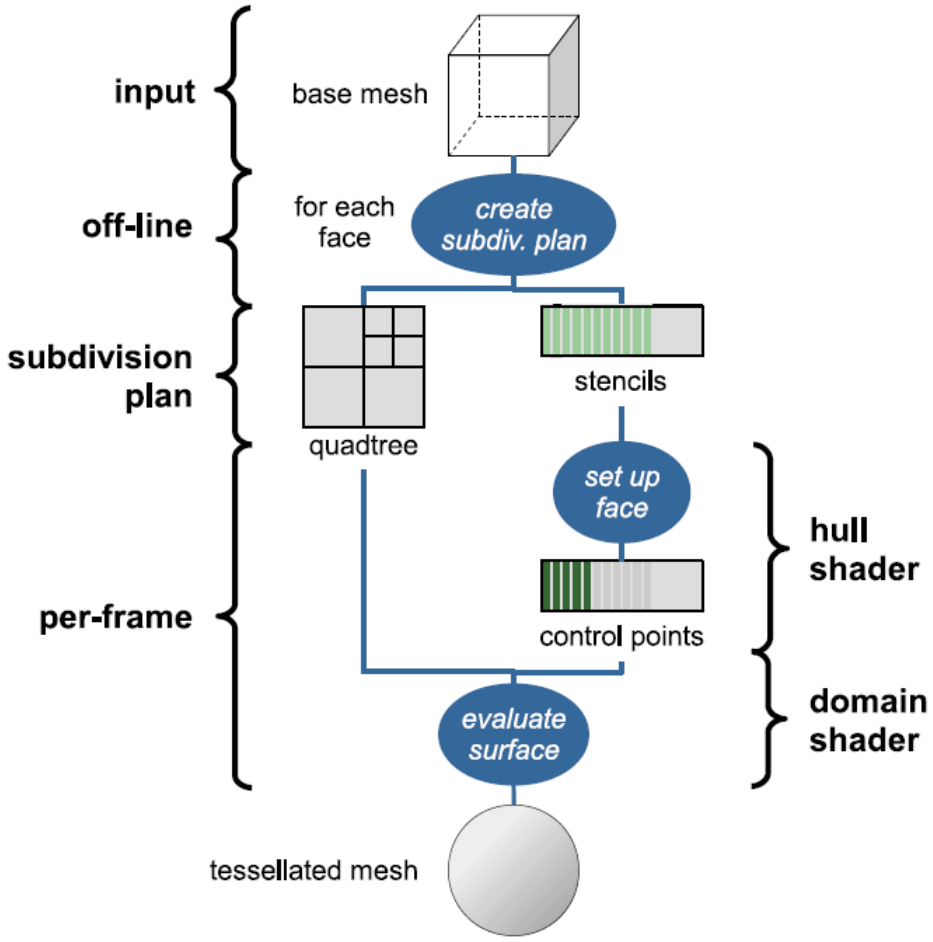

Figure 4.16: Adaptive Quadtrees Subdivision Overview

fixed maximum depth. Specifically, it comprises:

- a quadtree of the adaptive subdivision hierarchy of the face
- an ordered list of stencils for control points required by subdivided faces. Each stencil represents a control point as a weighted sum over base vertices in the 1-ring of the face.

Figure 4.17 shows an example of the quadtree structure

After that, they submit one primitive to the hardware tessellator for each quad

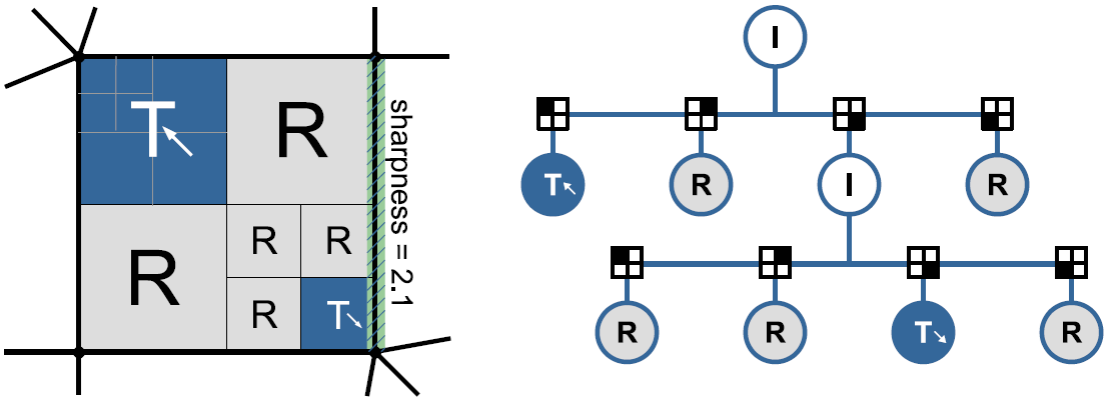

Figure 4.17: Quadtree Structure

face of the base mesh, irrespective of regularity, special features, or valence. This single primitive may represent a hierarchy of adaptively-subdivided faces, each with their own control points.

For the surface evaluation, the hardware tessellator produces a set of tessellation vertices, each associated with a parametric location within the u, v domain of the face. The surface evaluation step computes position and tangents of the limit surface at each of these locations. Surface evaluation maps to the hardware domain shader. Evaluation traverses the quadtree in the subdivision plan to find the directly-evaluable sub-face in which the u, v location lands. In the common case, this is a regular sub-face, which is evaluated as a B-spline surface, using control points output by the subdivision stage.

Chapter 5

RAY TRACING OF SUBDIVISION SURFACES

As the subdivision surfaces have been the standard modeling primitive, the demand for efficient rendering algorithms increases for both real time and photorealistic display. While interactive rendering has been addressed and near-interactive ray tracing has come within reach, efficient ray tracing of subdivision surfaces has only received little attention.

The ray tracing of subdivision surfaces existing so far can be classified in two categories Müller *et al.* (2003). The first one is to tessellate the surface before the ray tracing process and render the resulting polygons. The second method is that the intersection test is done directly with subdivision surface.

For the first category, the surface is refined uniformly or adaptively according to the curvature of the surface. Then the resulting polygons are rendered by a ray tracer. In order to obtain a high quality image (e.g., visible at reflections and silhouettes of objects and their shadows), a sufficient fine tessellation is necessary. However, because the refinement is executed independently from the scene, the complete, dense object must be refined, which means those parts that are not necessary for the computation of ray tracing are subdivided. This leads to a great number of polygons taking into account the exponential growth during a subdivision step. In order to deal with this problem, a reasonable subdivision level for the control mesh must be chosen depending on, e.g., the distance to the camera.

In the second category, the refinement of the control mesh is done automatically during the ray tracing. Therefore only parts of the control mesh with contribution to the light transport are refined. Due to the fact that in ray tracing, the rays

intersect geometry in a coherent way, Christensen *et al.* (2003) proposed using ray differentials to construct multiresolution caches for tessellated faces. They called such rays coherent rays. In ray tracing process, the specular reflection and refraction rays from flat or slightly curved surface are usually coherent; shadow rays from point lights, spot lights, directional lights, and small area lights are coherent. In this situation, it is possible to ray trace complex scenes: only the directly visible parts and the reflected, refracted, and shadow-casting areas need to be kept in tessellated form at any given time. Therefore, a geometric caching system is admirable for efficient store and trigger the tessellated faces.

5.1 Multiresolution Geometry Caching

Christensen *et al.* (2003) introduced an efficient algorithm to render ray tracing and global illumination effects in very complex scenes — scenes that are so complex that a fully tessellated representation of all objects would need more memory than is available. They utilized the relation between ray differentials and ray coherency to deal with classic and distribution ray tracing in complex scene. The key is that there are two types of rays:

1. Specular reflection and refraction rays from surfaces with low curvature and shadow rays to point-like light sources. These rays have small differentials, and require high accuracy and fine tessellation. These rays are usually coherent, so using a geometry cache with relatively small capacity (few entries) works well.

2. Specular reflection and refraction rays from highly curved surfaces and rays from wide distribution ray tracing. These rays have large differentials, do not require high accuracy, and can use a coarse tessellation. These rays are incoherent, so they require a cache with large capacity (many entries) and/or entries that are fast to recompute.

Figure 5.1 shows parallel rays specularly reflected by flat, convex, and concave surfaces.

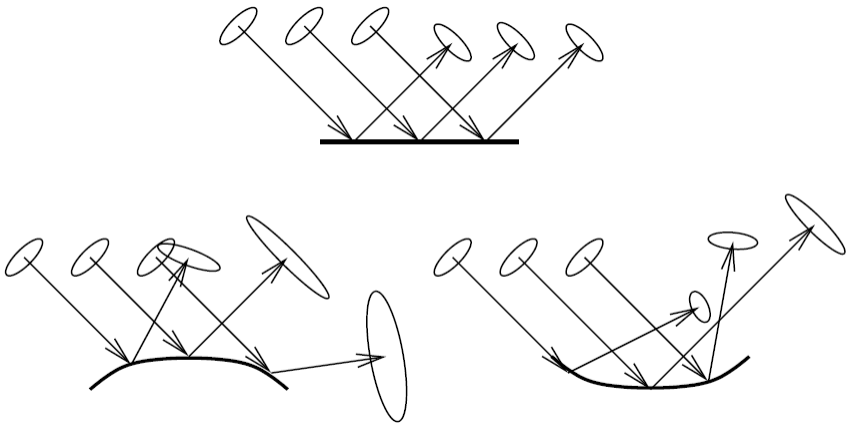

Figure 5.1: Specular Reflection

According to this relation, they present a multiresolution geometry caching scheme with separate caches for coarsely, medium, and finely tessellated surfaces. This exploits the different coherencies of various types of rays, and their different accuracy requirements. It is interesting to note that this scheme results in an automatic level-of-detail representation of the tessellation.

5.2 Lazy-Build Tessellation Caching

Benthin *et al.* (2015) proposed a lazy-build caching scheme to efficiently handle the highly tessellated surfaces while also utilizing today's many core architecture. According to their method, they lazily tessellate patches only when necessary, and utilize adaptive subdivision to efficiently evaluate the underlying surface representation. The core idea of their approach is a shared lazy evaluation cache, which triggers and maintains the surface tessellation. In addition, they also combine the caching scheme with SIMD-optimized subdivision primitive evaluation and fast hierarchy con-

struction over the tessellated surface.

5.2.1 Patch Generation

For ray tracing subdivision surfaces, they dynamically tessellate Catmull-Clark patch primitives. Then the tessellation is used to determine the ray intersection. For the patch construction, they follow a similar data flow to the feature-adaptive subdivision algorithm proposed by Nießner *et al.* (2012), where the input data is given as a coarse mesh over polygons (typically quad-dominant) with additional data arrays defining features like tessellation level, edge creases, vertex creases, and holes.

According to feature adaptive subdivision, to achieve as few patches as possible, they adaptively subdivide the patch only as long as the face is not a regular face. Finally, B-Spline patches are used for the regular faces, and Gregory patches are used to approximate patches with irregular faces. If accuracy of the surface is a concern, feature-adaptive subdivision could easily be continued to reduce the size of Gregory patches without any modification to the rest of our approach.

They build a top-level hierarchy over the original set of bicubic B-Spline and Gregory patches. While they build the top-level hierarchy at the beginning, each patch also maintains its own local BVH. A local BVH is traversed once a ray reaches a leaf of the top-level BVH. In contrast to the top-level BVH, all local hierarchies are built on-demand during ray traversal and their data is stored and managed by the tessellation cache.

5.2.2 Ray Patch Intersection

In order to intersect a ray with a subdivision model, they first traverse the ray through the top-level hierarchy. If a ray reaches a leaf, they then compute the intersection between the ray and the associated patch. To this end, they evaluate the

patch at a uniformly-spaced set of 2D domain locations, thus obtaining a regular grid of vertices. Displacements are applied to grid vertices. They then construct a local BVH4 over the corresponding tessellation, enabling fast ray traversal. Finally, when a ray reaches a leaf of a local BVH4, triangle intersection tests are performed.

5.2.3 Shared Lazy-Build Cache

Even though patch tessellation and local BVH4 construction is fast, performing them every time a ray intersects a patch will severely impact performance, as the associated construction cost is much higher than that of traversing a ray through the local BVH4 afterwards. As ray distributions typically exhibit spatial and temporal coherence, caching previously generated tessellation and BVH4 data is therefore an efficient approach to save redundant calculation.

To this end, they propose a new lazy-build evaluation cache specifically designed for many-core architectures. The cache operates as a globally-shared segmented FIFO (first-in first-out) cache that can efficiently store irregular-sized tessellation and BVH4 data under heavy multi-threaded conditions. As long as the cache is filling up, the scheme behaves exactly like a lazy hierarchy build with unbounded memory storage.

5.3 Feature Adaptive Ray Tracing

Unlike previous work, I propose a new way to improve the efficiency of ray tracing subdivision surfaces, while it has lower quality. According to feature adaptive subdivision, there are many patches around crease edges and extraordinary vertices, while there are less on the regular faces, which means the geometry around those places are complex and for the regular region, it is kind of smooth. Can we use this idea for the ray tracing? We cast more rays on the area with more patches and less for the regular part which have less patch. It is because those area have complex geometry

structure. It needs more rays to generate accurate result. And the smooth area are not.

A common ray tracing algorithm casts rays for each pixel on the image to the object world. But it is very expensive to calculate the color of all pixels. My work is trying to find a way to reduce the number of rays it needs to compute. The overview of my algorithm is that it first divides the image pixels into different groups by certain algorithm. For one group, it chooses several pixels to cast rays and then uses an interpolation algorithm to assign color for other pixels.

The following sections introduce all necessary features for my algorithm.

5.3.1 Half-Edge Data Structure

For the subdivision scheme, it requires a lot of operations that need to search adjacent edges, vertices and faces. In order to efficiently support adjacent queries for subdivision surfaces, a half-edge structure is needed based on the input edge data.

This data structure is called half-edge because instead of storing the edges of meshes, it stores half-edges. For an edge that connects two vertices, it splits into two half-edges, called pair, which have opposite directions. Figure 5.2 shows an example.

As you can see, every face is formed by a circular linked list around it. And for each half-edge, it has a pointer to its pair, the one with opposite direction.

It might look something like this:

```
struct HE_edge {
    HE_vert* source_vert;
    HE_edge* pair;
    HE_face* face;
    HE_edge* next;
};
```

For vertices, it has a vertex array to store all vertices information so that in the half-edge data structure, it only stores the index of the vertex. And it also has a

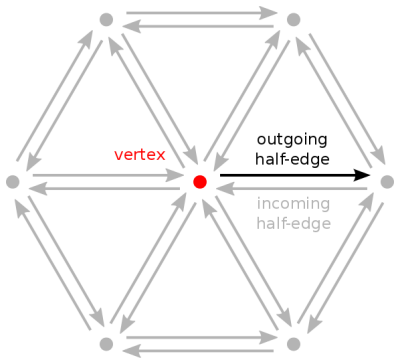

Figure 5.2: Half-Edge Data Structure

pointer to which half-edge it connects.

```
struct HE_vert {
    int index;
    HE_edge* edge;
};
```

For faces, it is quite simple which only needs a pointer to the first half-edge.

```
struct HE_face {
    HE_edge* edge;
};
```

The first frequently used query is to iterate over the half-edges adjacent to a face.

The pseudocode below shows how to do that:

```
HE_edge* edge = face->edge;
do {
    // do something
    edge = edge->next;
} while(edge != face->edge)
```

The second one is to iterate over the half-edges adjacent to a vertex.

```
HE_edge* edge = vert->edge;
do {
    // do something
    edge = edge->pair->next;
} while(edge != vert->edge)
```

5.3.2 *Patch Generation*

A patch is a face with its 1-ring vertices, which shows in Figure 5.3. And due to the property of Catmull-Clark subdivision, the 1-ring is the only information required to subdivide a face.

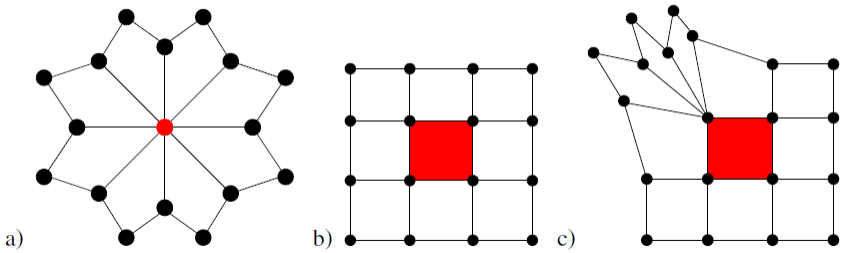

Figure 5.3: 1-Ring Vertices

The algorithm to fetch the 1-ring around a face is easy with the help of half-edge data structure. For a given face, the outer loop iterates all vertices, and the inner loop adds all faces around this vertex. Here is the psuedocode:

```
// f is the given face
HE_edge* e = f->edge;
do {
    HE_vert* v = e->source_vert;
    HE_edge* e_v = v->edge;

    do {
        add(e_v->face); // will remove repeated faces
        e_v = e_v->pair->next;
    } while (e_v != v->edge);
    e = e->next;
} while (e != f->edge);
```

After all 1-rings per input face are converted to a set of patches, they are able to be subdivided by using feature adaptive subdivision to generate a more refined mesh. The subdivision scheme has been discussed in chapter 4.

5.3.3 Ray-Patch Intersection

K-d tree and axis-aligned bounding box are used here as the acceleration structures for the ray-patch intersection. Once a ray hits a patch, it replaces the underlying faces with triangles, and executes ray-triangle intersection which has been discussed before as well.

5.3.4 Groups of Pixels

Once the model has been loaded, the next step is to divide the pixel map into different groups. The way using here is according to the first object the ray hits from this pixel. When the normal ray tracing is performed, it casts ray from each pixel to the world. For this method, it has an array to store the patch index which is the closest patch to the view plane. If the projection from object world to the pixel map is done, a breadth-first search is used to group the pixels that hit the same patch which means their patch index should be the same.

1												
1	1	1										
1	1	1	1									
1	1	1	1			3	3	3	3	3	3	3
					3	3	3	3	3	3	3	
						3	3	3	3	3		
							3	3				
2	2	2										
2	2	2	2									
	2	2	2	2								
		2	2	2								

Figure 5.4: Example of Groups of Pixels

```
for(i = 0; i < height; i++) {
    for(j = 0; j < width; j++) {
```

```
            if (pixel(i, j) is not added before) {
                Q.enqueue(pixel(i, j));
                while (Q is not empty) {
                    current = Q.dequeue();
                    int x = current.x;
                    int y = current.y;

                    // find adjacent pixels with the same patch index

                    // pixel on top of current

                    if (pixel(x-1, y) is not added before
                        && patch indices are same) {
                        Q.enqueue(x-1, y);
                    }
                    // the same for other three directions
                    ...

                }

            }

        }
}
```

5.3.5 Interpolation

For each group of pixels, a bilinear interpolation is used to approximate the color of the pixels. At first, the algorithm is to find the top, bottom, left and right pixels that has the same patch index. Then it casts rays for those four pixels just like the normal ray tracing. Finally, it uses the color of the four pixels and bilinear interpolation to calculate the color of the rest pixels in this groups.

5.3.6 Result

Table 5.1 shows the number of rays this algorithm casts in order to render the image, Table 5.2 shows the total time needed for rendering, and Table 5.3 shows the interpolation time and ray casting time for feature adaptive ray tracing. The image

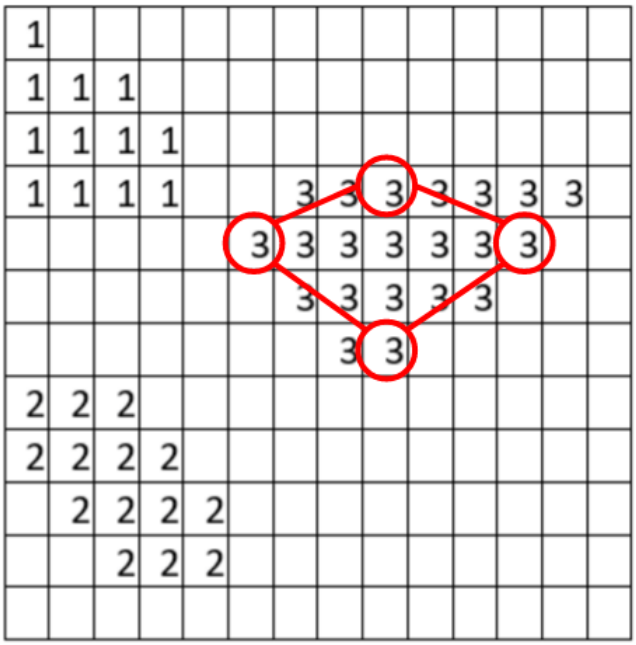

Figure 5.5: Bilinear Interpolation

size is 800 × 600.

Table 5.1: Rays Count

Depth	Feature Adaptive Ray Tracing	Normal Ray Tracing	Total Pixels
0	10115	95408	480000
1	40235	93983	480000
2	74409	93625	480000

With the increase of subdivision level, there are more rays needed for rendering. It is because the interpolation is based on the same patch. For a refined mesh with higher subdivision level, there are more patches.

And figure 5.6 - 5.8 show the comparison of feature adaptive ray tracing and the normal ray tracing.

Table 5.2: Rendering Time

Depth	Feature Adaptive Ray Tracing (s)	Normal Ray Tracing (s)
0	3.1	11.5
1	20.4	30.2
2	113.3	135.4

Table 5.3: Interpolation and Ray Casting Time

Depth	Feature Adaptive Ray Tracing (s)	Interpolation (s)	Rays Casting (s)
0	3.1	0.6	2.5
1	20.4	2.4	18
2	113.3	3.3	110

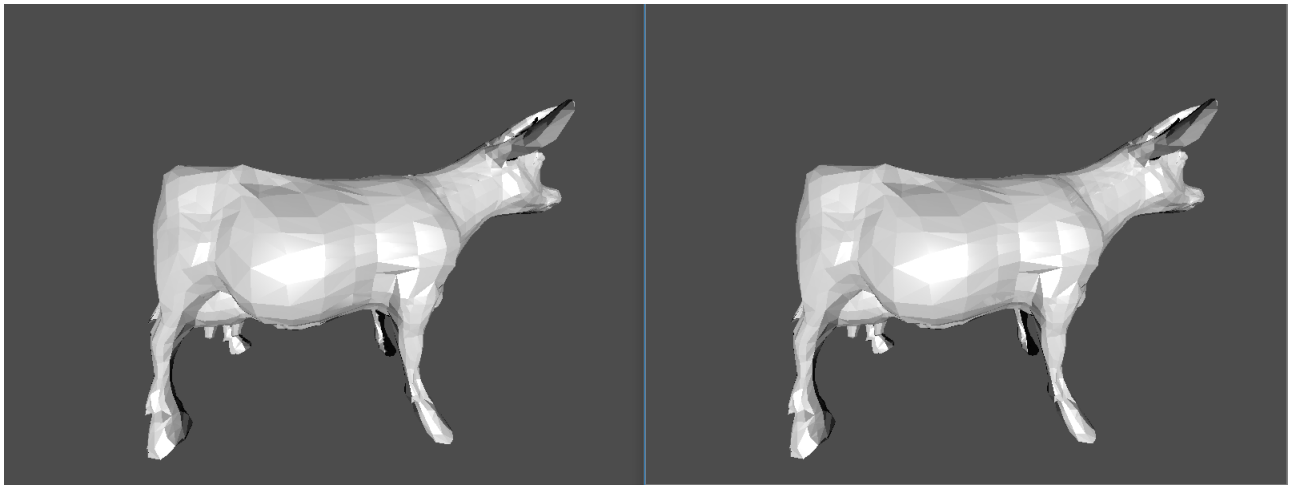

Figure 5.6: Subdivision Level 0 (Left Is the Feature Adaptive Ray Tracing, and Right Is The Normal Ray Tracing)

5.3.7 Image Quality Assessment

In computer graphics world, there is an intensive producer of visual content like images, videos, 3D data, etc. Therefore the need for controlling and evaluating the

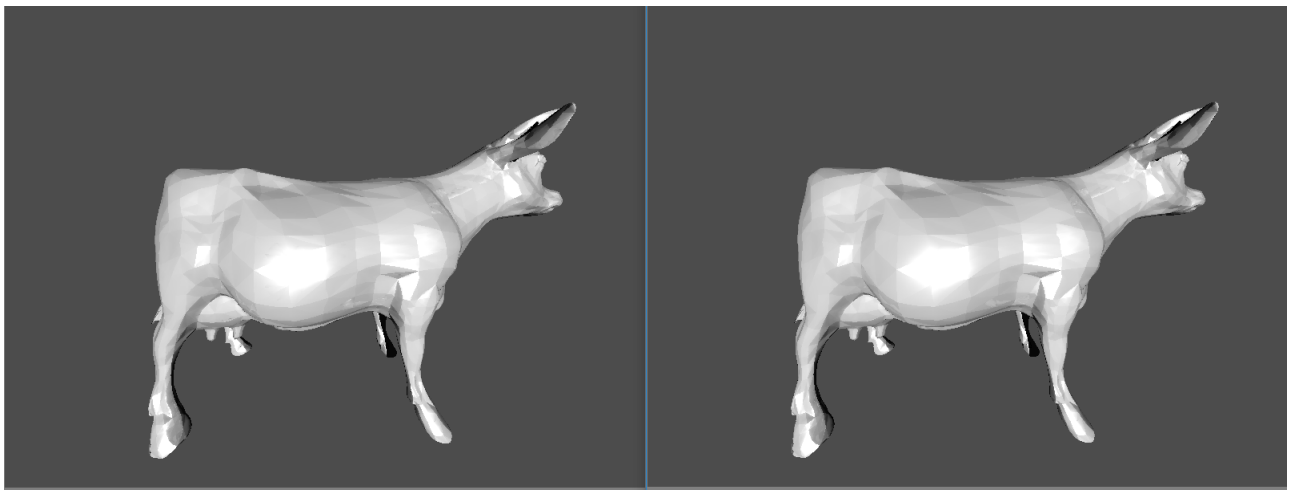

Figure 5.7: Subdivision Level 1 (Left Is the Feature Adaptive Ray Tracing, and Right Is The Normal Ray Tracing)

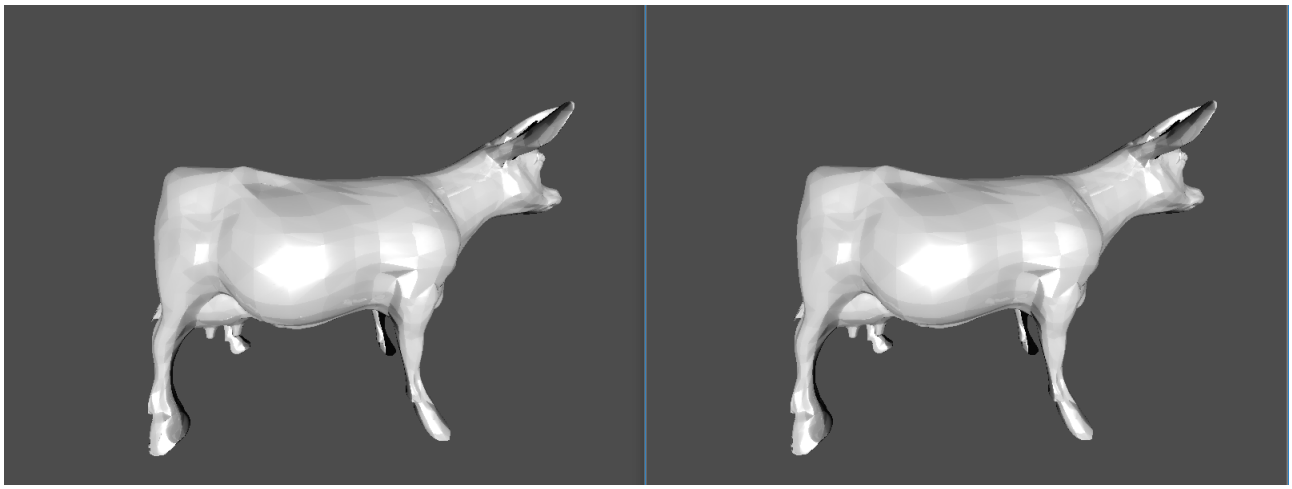

Figure 5.8: Subdivision Level 2 (Left Is the Feature Adaptive Ray Tracing, and Right Is The Normal Ray Tracing)

quality of these graphical data is necessary regardless of the application. How to compare and evaluate the quality of images is important for researchers who want to prove how good their algorithm is. Lavoué and Mantiuk (2015) present the existing objective quality metrics, the subjective quality experiments as well as an evaluation and comparison of their performance.

For the need of my algorithm, I choose the HDR-VDP-2 metric to compare and evaluate between the rendering effect of feature adaptive ray tracing and normal ray tracing.

HDR-VDP-2 is the visibility (discrimination) and quality metric capable of detecting differences in achromatic images spanning a wide range of absolute luminance values proposed by Mantiuk *et al.* (2011). Although the metric originates from the classical Visual Difference Predictor proposed by Daly (1992), and its extension HDR-VDP proposed by Mantiuk *et al.* (2005), the visual models are very different from those used in those earlier metrics. The metric is also an effort to design a comprehensive model of the contrast visibility for a very wide range of illumination conditions.

Figure 5.9 to 5.11 shows the results by using HDR-VDP-2 metric to compare the feature adaptive ray tracing and normal ray tracing.

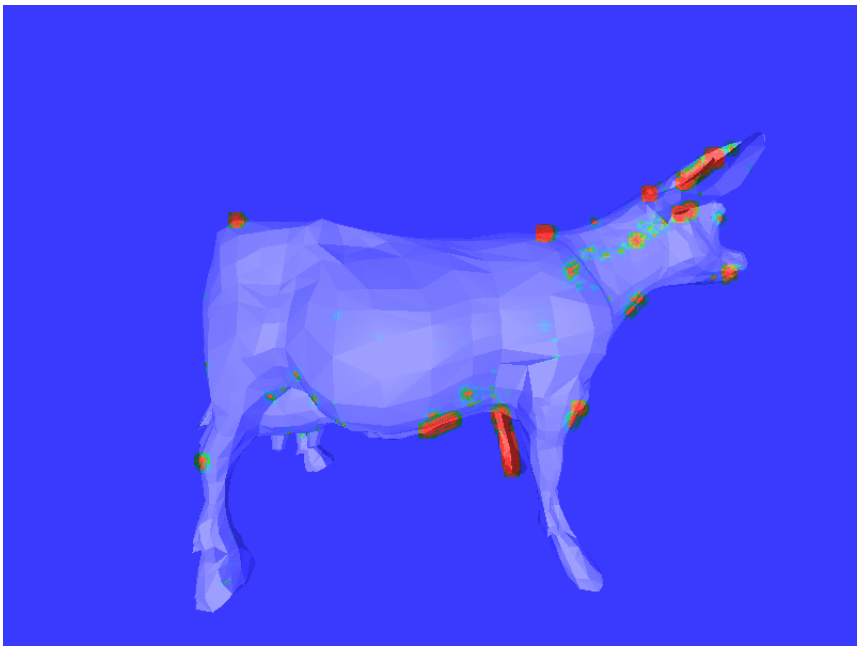

Figure 5.9: HDR for Depth = 0

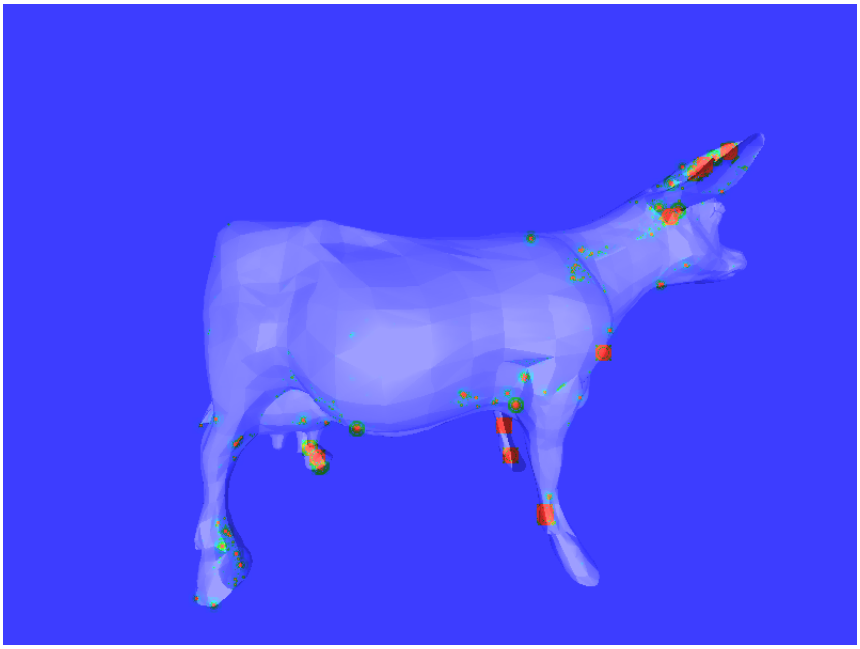

Figure 5.10: HDR for Depth = 1

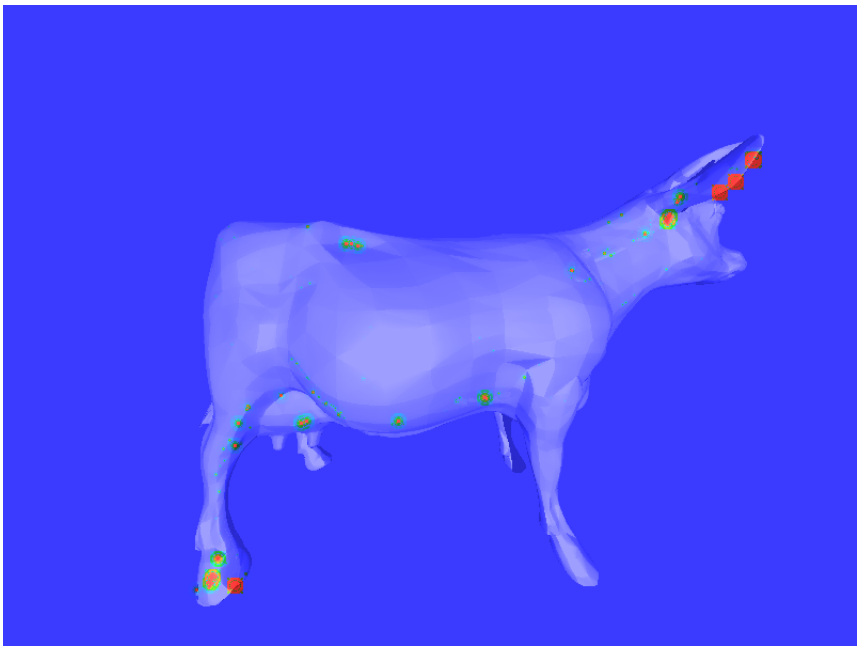

Figure 5.11: HDR for Depth = 2

Chapter 6

FUTURE

The algorithm presented here is a prototype which just shows the idea of feature adaptive ray tracing. There are many aspects that needs to improve. At the first place, the ray tracer described above is only a small subset of what can be done with ray tracing. More topics can be integrated into this algorithm.

—Supersampling and Antialiasing: there will be 'jaggies' along the edges of objects if it only sends one ray per pixel. Supersampling is a method to send multiple rays per pixel, and it deals with which rays to send and how to combine those colors. There are impressive researches that have done in this fields, like Whitted (2005) and Lee *et al.* (1985).

—Shading Model: The shading model used in this thesis is a very basic one. There are many different shading models that can describe the world more accurately, like Cook and Torrance (1982) and Hall *et al.* (1983).

Secondly, for ray tracing of subdivision surfaces, using a cache system to dynamic tessellate the control mesh is significant important due to the memory I/O problems. And for the algorithm presented in this thesis, it is very easy to support dynamic tessellation. Benthin *et al.* (2015) proposed a lazy-build caching system with SIMD-optimized subdivision primitive evaluation and fast hierarchy construction. In addition, they have integrated their work into Embree Wald *et al.* (2014) that makes interactive ray tracing of subdivision surfaces available.

In the third place, apart from the feature adaptive subdivision surfaces, an adaptive quadtrees structure subdivision surfaces has been presented by Brainerd *et al.* (2016). Their method subdivides the u, v domain of each face ahead of time, and

generates a quadtree structure. It yields a more uniformed surface and needs fewer primitives. It is also possible to integrate their method to this algorithm.

And finally, in this algorithm, it uses a bilinear interpolation to interpolate the color for the pixels in same group. I think there should have a more accurate model other than bilinear interpolation. Even though the result in this thesis is not bad, it may seems unsatisfiable when the scene becomes more complex. It still needs further research to make this idea robust and efficient for all situations. For example, instead of using bilinear interpolation, a convex hull can be used to do the interpolation more accurately.

Benthin, C., S. Boulos, D. Lacewell and I. Wald, "Packet-based ray tracing of catmull-clark subdivision surfaces", SCI Institute, University of Utah, Technical Report (2007).

Benthin, C., S. Woop, M. Nießner, K. Selgrad and I. Wald, "Efficient ray tracing of subdivision surfaces using tessellation caching", in "Proceedings of the 7th Conference on High-Performance Graphics", pp. 5–12 (ACM, 2015).

Bentley, J. L., "Multidimensional binary search trees used for associative searching", Communications of the ACM **18**, 9, 509–517 (1975).

Brainerd, W., T. Foley, M. Kraemer, H. Moreton and M. Nießner, "Efficient gpu rendering of subdivision surfaces using adaptive quadtrees", ACM Transactions on Graphics (TOG) **35**, 4, 113 (2016).

Catmull, E. and J. Clark, "Recursively generated b-spline surfaces on arbitrary topological meshes", Computer-aided design **10**, 6, 350–355 (1978).

Christensen, P. H., D. M. Laur, J. Fong, W. L. Wooten and D. Batali, "Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes", in "Computer Graphics Forum", vol. 22, pp. 543–552 (Wiley Online Library, 2003).

Cook, R. L. and K. E. Torrance, "A reflectance model for computer graphics", ACM Transactions on Graphics (TOG) **1**, 1, 7–24 (1982).

Daly, S. J., "Visible differences predictor: an algorithm for the assessment of image fidelity", in "SPIE/IS&T 1992 Symposium on Electronic Imaging: Science and Technology", pp. 2–15 (International Society for Optics and Photonics, 1992).

DeRose, T., M. Kass and T. Truong, "Subdivision surfaces in character animation", in "Proceedings of the 25th annual conference on Computer graphics and interactive techniques", pp. 85–94 (ACM, 1998).

Farin, G. E., *Curves and surfaces for CAGD: a practical guide* (Morgan Kaufmann, 2002).

Hall, R., D. P. Greenberg *et al.*, "Testbed for realistic image synthesis.", IEEE COMP. GRAPHICS APPLIC. **3**, 8, 10–20 (1983).

Hoppe, H., T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer and W. Stuetzle, "Piecewise smooth surface reconstruction", in "Proceedings of the 21st annual conference on Computer graphics and interactive techniques", pp. 295–302 (ACM, 1994).

Kuchkuda, R., "An introduction to ray tracing", Theoretical Foundations of Computer Graphics and CAD, Italy (1987).

Lavoué, G. and R. Mantiuk, "Quality assessment in computer graphics", in "Visual Signal Quality Assessment", pp. 243–286 (Springer, 2015).

Lee, M. E., R. A. Redner and S. P. Uselton, "Statistically optimized sampling for distributed ray tracing", in "ACM SIGGRAPH Computer Graphics", vol. 19, pp. 61–68 (ACM, 1985).

Mantiuk, R., S. J. Daly, K. Myszkowski and H.-P. Seidel, "Predicting visible differences in high dynamic range images: model and its calibration", in "Electronic Imaging 2005", pp. 204–214 (International Society for Optics and Photonics, 2005).

Mantiuk, R., K. J. Kim, A. G. Rempel and W. Heidrich, "Hdr-vdp-2: a calibrated visual metric for visibility and quality predictions in all luminance conditions", in "ACM Transactions on Graphics (TOG)", vol. 30, p. 40 (ACM, 2011).

Möller, T. and B. Trumbore, "Fast, minimum storage ray/triangle intersection", in "ACM SIGGRAPH 2005 Courses", p. 7 (ACM, 2005).

Müller, K., T. Techmann and D. Fellner, "Adaptive ray tracing of subdivision surfaces", in "Computer Graphics Forum", vol. 22, pp. 553–562 (Wiley Online Library, 2003).

Nasri, A. H., "Polyhedral subdivision methods for free-form surfaces", ACM Transactions on Graphics (TOG) **6**, 1, 29–73 (1987).

Nießner, M., C. Loop, M. Meyer and T. Derose, "Feature-adaptive gpu rendering of catmull-clark subdivision surfaces", ACM Transactions on Graphics (TOG) **31**, 1, 6 (2012).

Parker, S. G., J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison *et al.*, "Optix: a general purpose ray tracing engine", in "ACM Transactions on Graphics (TOG)", vol. 29, p. 66 (ACM, 2010).

Studios, P. A., "The renderman interface version 3.2. 1", URL: http://renderman. pixar. com/view/rispec **70** (2005).

Tejima, T., M. Fujita and T. Matsuoka, "Direct ray tracing of full-featured subdivision surfaces with bezier clipping", Journal of Computer Graphics Techniques (JCGT) **4**, 1, 69–83 (2015).

Wald, I., S. Woop, C. Benthin, G. S. Johnson and M. Ernst, "Embree: a kernel framework for efficient cpu ray tracing", ACM Transactions on Graphics (TOG) **33**, 4, 143 (2014).

Whitted, T., "An improved illumination model for shaded display", in "ACM Siggraph 2005 Courses", p. 4 (ACM, 2005).

Williams, A., S. Barrus, R. K. Morley and P. Shirley, "An efficient and robust ray-box intersection algorithm", in "ACM SIGGRAPH 2005 Courses", p. 9 (ACM, 2005).