Using subdivision surfaces and PN triangles for game applications

Pu, Guannan

2009

Pu, G. (2009). Using subdivision surfaces and PN triangles for game applications. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/21079>

<https://doi.org/10.32657/10356/21079>

NANYANG
TECHNOLOGICAL
UNIVERSITY

Using subdivision surfaces and PN triangles for game applications

A thesis submitted to the Nanyang Technological University

In fulfillment of the requirement for the degree of

Master of Engineering

Pu Guannan

B. Eng. Computer Engineering (Honours)

Nanyang Technological University
Division of Computer Science
School of Computer Engineering
2009

Abstract

Polygonal meshes (in particular, triangular meshes) are commonly used to represent and render models and objects in game applications. Due to speed and storage limitation, usually only coarse meshes and simplified meshes are used for rendering. To generate more details from a coarse mesh, several techniques can be applied, like normal mapping, displacement mapping, subdivision surfaces and other techniques.

This thesis investigates efficient and adaptive mesh refinement schemes for rendering triangular meshes in modern games. Two schemes are proposed in this thesis:

First, a subdivision scheme which applies Loop subdivision over multiple meshes is proposed to provide mesh surface refinement suitable for game applications. This scheme can overcome the difficulties of using subdivision in game applications. The implementation has been used in real commercial game engine.

Second, a novel surface tessellation scheme based on PN (point normal) triangles is proposed. The new technique can perform fast adaptive tessellation and rendering using GPU shader processors, which achieves smoothness and speed in the same time. The scheme reduces the number of triangles used and it also reduces memory consumption and complexity compared to previous existing methods.

Acknowledgement

I would like to give deepest thank to my supervisor Dr. Zheng Jianmin who spent a lot of his time on directing and advising my research project. I owe thousand thanks for his inspiration, consultation and continuous encouragement. Without his helps and thoughts the project could not start. His advices are the fundamental of this project.

I would like to thank Dr. Seah Hock Soon who has given me a lot of help and advices during my final year. He helped me to keep working on with a lot of efforts.

I would like to thank my previous colleagues Parimal Aswani and Sundar Raman, for their continuous support and guidance throughout the duration of my project. They provided many thoughtful ideas and great helps to me and kept me on the right track.

I would like to thank Rick Baltman, Mark De Simone and Doug McNabb from 2XL Games, Inc., Phoenix, USA, for their insightful suggestions based on strong industrial background.

Finally I would like to thank my family for their moral support in those difficult times.

Table of Contents

Abstract	1
Acknowledgement	2
Table of Contents	3
Table of Figures	6
Table of tables	9
Chapter 1 Introduction	10
1.1 Motivation	10
1.2 Objective and scope	13
1.3 Contribution	14
1.4 Outline	15
Chapter 2 Background and related work	16
2.1 Subdivision surfaces	16
2.1.1 Chaikin's algorithm	16
2.1.2 Subdivision surfaces	18
2.1.3 Loop subdivision	20
2.2 Curved PN triangles	22
2.2.1 Bézier triangle	22
2.2.2 Curved PN triangles	23
2.3 GPU tessellation	29
2.3.1 Graphics Processing Unit	29
2.3.2 General-Purpose computation on Graphics Processing Units	29
2.3.3 Shader	30
2.3.4 Level of details	32

2.3.5 Generic mesh refinement on GPU	33
2.3.6 Adaptive mesh refinement on GPU	34
2.4 Other real-time refinement techniques	37
Chapter 3 Loop subdivision over multiple meshes for games	38
3.1 Loop subdivision surfaces over multiple meshes	38
3.1.1 Surface matching.....	38
3.1.2 Smooth surface matching.....	42
3.2 Implementation of Loop subdivision over multiple meshes.....	43
3.2.1 Winged-edge data structure	43
3.2.2 Modified Winged-edge data structure.....	44
3.2.3 Loop subdivision over a single mesh.....	48
3.2.4 Loop subdivision over multiple meshes with different textures	49
Chapter 4 New adaptive GPU PN triangles tessellation	52
4.1 Curved PN triangles on GPU.....	52
4.1.1 Overview	52
4.1.2 Design	53
4.1.3 Implementation	55
4.1.4 Examples	57
4.1.5 Performance	63
4.1.6 Observation	64
4.2 Adaptive Curved PN triangles on GPU	64
4.2.1 Overview	64
4.2.2 Depth-tagging.....	66
4.2.3 Face matching	70
4.2.4 New adaptive tessellation algorithm	71
4.2.5 Performance	81

Chapter 5 Conclusion.....	89
References.....	91

Table of Figures

Figure 1.1 Screenshot from Doom, id Software, 1993[2].....	11
Figure 1.2 Screenshot from Gears of War, Epic Games, 2006 [6]	11
Figure 1.3 Screenshot from Uncharted Drake's Fortune, Naughty Dog, 2007 [7].....	12
Figure 2.1 Example of Chaikin's Algorithm	17
Figure 2.2 Subdivision example	18
Figure 2.3 Illustration of Loop Subdivision.....	21
Figure 2.4 Example of Loop subdivision.....	21
Figure 2.5 Bézier triangles: a cubic patch with its control nets	23
Figure 2.6 The control points of a triangular Bézier patch form a control net	24
Figure 2.7 Normal components of curved PN triangles	25
Figure 2.8 Input points and normals	26
Figure 2.9 Normal calculation of PN triangles[8]	26
Figure 2.10 Curved PN triangle tessellation in the graphics pipeline.....	28
Figure 2.11 Vertex shader.....	31
Figure 2.12 Pixel shader	32
Figure 2.13 A discrete LOD example	32
Figure 2.14 Illustration of generic GPU refinement process[47]	33
Figure 2.15 Depth tag configuration of adaptive mesh refinement[48].....	35
Figure 2.16 The matrix of adaptive refinement patterns[48]	36
Figure 2.17 Architecture of adaptive refinement scheme[48]	36
Figure 3.1 A single surface contains multiple textures.....	39
Figure 3.2a Multi-texture surface models in games (rocks)	39
Figure 3.2b Visual cracks when subdivision is applied on multi-texture surface	40
Figure 3.3 Crack solution.....	41

Figure 3.4 Result of the average approach.....	41
Figure 3.5 Illustration of the extension approach	42
Figure 3.6 Winged-edge data structure [61]	44
Figure 3.7 Class diagram of modified winged edge data structure.....	45
Figure 3.8 Traversing surrounding vertices within the Winged-edge data structure.....	48
Figure 3.9 Loop subdivision implementation	49
Figure 3.10 Smooth extension implementation	50
Figure 3.11 Averaging matching	51
Figure 3.12 Smooth extension	51
Figure 4.1 Refinement patterns.....	53
Figure 4.2 PN triangles technique on GPU.....	54
Figure 4.3 Original PN triangles technique	55
Figure 4.4 GPU curved PN triangles implementation	56
Figure 4.5 GPU curved PN triangles on a cube	59
Figure 4.6a GPU curved PN triangles on a sphere, depth level 1.....	60
Figure 4.6b GPU curved PN triangles on a sphere, depth level 14	60
Figure 4.6c GPU curved PN triangles on a sphere, depth level 40.....	61
Figure 4.7a GPU curved PN triangles on a teapot, depth level 1	61
Figure 4.7b GPU curved PN triangles on a teapot, depth level 2	62
Figure 4.7c GPU curved PN triangles on a teapot, depth level 10	62
Figure 4.8 Adaptive PN triangles.....	66
Figure 4.9 Coefficients of Bézier Triangle	67
Figure 4.10 Calculate the distance for edge.....	68
Figure 4.11 Four control points of an edge.....	68
Figure 4.12 Calculate the distance for surface.....	69
Figure 4.13 Face Matching	71

Figure 4.14 Adaptive PN triangles original teapot model wireframe	75
Figure 4.15 Adaptive PN triangles original teapot model solid rendering	75
Figure 4.16 Adaptive PN triangles refined teapot model wireframe	76
Figure 4.17 Adaptive PN triangles teapot refined model solid rendering	76
Figure 4.18 Adaptive PN triangles: tolerance = 0.01, teapot wireframe	77
Figure 4.19 Adaptive PN triangles: tolerance = 0.01, teapot solid rendering.....	77
Figure 4.20 Adaptive PN triangles: original torus knot wireframe	78
Figure 4.21 Adaptive PN triangles: original torus knot solid rendering.....	78
Figure 4.22 Adaptive PN triangles refined torus knot model wireframe.....	79
Figure 4.23 Adaptive PN triangles refined torus knot model wireframe.....	79
Figure 4.24 Adaptive PN triangles: tolerance = 0.01, torus knot wireframe	80
Figure 4.25 Adaptive PN triangles: tolerance = 0.01, torus knot wireframe	80
Figure 4.26 Adaptive PN triangles: original human head.....	81
Figure 4.27 Adaptive PN triangles: original human head wireframe	81
Figure 4.28 Adaptive PN triangles: tolerance = 0.258, human head	82
Figure 4.29 Adaptive PN triangles: tolerance = 0.258, human head wireframe.....	82
Figure 4.30 Adaptive PN triangles: original soldier model	83
Figure 4.31 Adaptive PN triangles: original soldier model wireframe.....	83
Figure 4.32 Adaptive PN triangles: tolerance = 0.88, soldier model.....	84
Figure 4.33 Adaptive PN triangles: tolerance = 0.88, soldier model wireframe	84
Figure 4.34 Comparison of FPS between adaptive scheme and uniform scheme	87
Figure 4.35 Comparison of GPU calculations between adaptive scheme and uniform scheme.....	87

Table of tables

Table 3.1 Edge table[28].....	44
Table 4.1 GPU curved PN triangles performance	63
Table 4.2 Adaptive GPU PN triangle performance	85

Chapter 1

Introduction

This chapter looks at the motivation for this project, its objective and scope, my contributions and outline of the thesis.

1.1 Motivation

A video game is an electronic game played by means of images on a video screen and often emphasizing fast action [1]. Visual effect is the key feature of video game and the graphics quality determines how players can interact with the game world. Visual effect of computer games has been improved greatly since the advancement of computer hardware and computer graphics theory.

Graphics of early games is very poor due to the limitation from both hardware and computer graphics techniques. A screenshot from best selling game Doom [2], shown in Figure 1.1, illustrates the early game graphics. Low polygon count models and simple textures are used in retro games. Storage was very limited for early games. For example, Atari 2600 [3] games have a maximum 4KB cartridge storage space [4].

Figure 1.1 Screenshot from Doom, id Software, 1993[2]

Modern games require immersive and realistic environment and atmosphere. Today most games contain gigabytes game files and require DVD or even Blu-ray [5] disc to store game data. Most of this huge storage space is used to store high resolution models and high quality media files. High resolution models contain high polygons counts, which make graphics look much better than games of last decades. High polygon count models can make the game world look very impressive together with dynamic lighting, high detailed texture and artificial intelligence. Figure 1.2 and Figure 1.3 are screenshots from modern computer games, Gears of War [6] and Uncharted Drake's Fortune [7].

Figure 1.2 Screenshot from Gears of War, Epic Games, 2006 [6]

Figure 1.3 Screenshot from Uncharted Drake's Fortune, Naughty Dog, 2007 [7]

From these images we can tell that the high resolution models are naturally more attractive than the low resolution ones. The capability of graphics hardware has increased tremendously in the past twenty years, which makes rendering of super high quality models possible. These high quality models can either be modeled beforehand or be real-time generated. However, highly detailed models will take up significant amount of storage space.

For modern game development, the storage space needed is increasing rapidly. The end user will face a rather long installation time and huge amount of disk space being occupied. The storage problem can be quite troublesome especially in console game development. Most console games would not require any installation to play, and the game will be loaded directly from an optical storage drive. The larger model packs for each environment will require longer loading time. If the game contents are forced to be divided into several discs, players need to frequently switch discs.

To address this storage issue, we need to find a real-time surface tessellation scheme which can generate highly detailed models in real-time for game applications. Thus we can achieve more impressive image quality without worrying about the storage consumption. Meanwhile real-time surface tessellation can free the game modelers from constructing highly detailed models.

1.2 Objective and scope

As discussed above, games have become more realistic and aesthetic looking with higher density of polygons. The main objective of this project is to develop efficient schemes to perform surface tessellation in modern games. The schemes should allow a game programmer to render a smooth surface easily.

Such schemes are not easy to achieve. Within the scope of game development, there are three limitations: (I) memory capacity for storing data structures, (II) calculation speed and (III) compatibility to real game development. To address these issues, we have to find suitable surface tessellation schemes for industrial usage. The surface tessellation schemes must be efficient to fit limited memory and fast enough for real-time rendering.

Game applications are quite different from animations and movies. All graphics in game excluding cut scenes are needed to be rendered in real-time. Many surface subdivision schemes are available for increasing the number of triangles and improving the smoothness of coarse models, but most of them are cost intensive and unsuitable for performance oriented game applications. For game developments where the rendering is required to be done in real-time, subdivision surfaces are very hard to implement. Traditional subdivision surfaces schemes are not applicable to be used by real-time applications like games. Due to the complexity of geometrical and topological

reconstructions accompanied with subdivision surfaces and the speed limitation, game developers have never applied subdivision in commercial games. In addition, in game applications 3D objects are often composed of many meshes, but subdivision algorithms usually apply to a single mesh. If we simply apply subdivision algorithms to each of the meshes, this likely results in cracks.

In this thesis, we will investigate the use of subdivision surfaces and fast smooth surface rendering techniques like Curved PN triangles [8] in the development of video games.

1.3 Contribution

The contribution of the thesis includes two schemes for generating smoother surfaces per memory space in game applications:

The first scheme is to apply Loop subdivision with Hoppe's extension [9] over multiple meshes. Game surfaces contain multiple patches, and traditional subdivision surfaces will generate cracks between these patches. This problem has been solved by our proposed scheme.

The second scheme is a new technique to perform mesh refinement using a combination of GPU and curved PN triangles, which is an efficient solution to achieve fast speed with subdivision-like quality. This scheme interprets triangular mesh models as curved PN triangles and provides a simple, adaptive approach to tessellate PN triangles. The method greatly reduces graphics memory consumption and storage space. It can be very easy to be implemented in game application. In this scheme, a smooth surface is defined by a set of control points, or just a coarse mesh. The shape of the surface is similar to the original coarse mesh, but containing a lot more details. A game programmer or general

programmer need not understand the underlying mathematics of the developed program to implement the scheme, and it works for arbitrary triangular surfaces which are used in most game applications.

1.4 Outline

The rest of the thesis is organized as follows: Chapter 2 gives a literature review, in which legacy subdivision schemes are discussed and curved PN triangles and a GPU tessellation scheme are also described. A scheme to apply Loop subdivision over multiple meshes is proposed in Chapter 3 and two schemes for tessellating curved PN triangles schemes on GPU are proposed in Chapter 4. A conclusion is given in Chapter 5.

Chapter 2

Background and related work

In this chapter, basic subdivision theory and traditional subdivision schemes are discussed. Necessary fundamental concepts such as Bézier triangles and PN triangles, and GPU tessellation are reviewed.

2.1 Subdivision surfaces

2.1.1 Chaikin's algorithm

The theory of surface subdivision can be traced back to Chaikin's algorithm [10]. Chaikin introduced a simple scheme for generating a smooth curve, which is the result of applying recursively corner-cutting to an input polygon.

Specifically, Chaikin's algorithm can be stated as follows:

Given a control polygon $\{P_0, P_1, \dots, P_n\}$, we refine this control polygon by generating a new polygon consisting of a sequence of control points $\{Q_0, R_0, Q_1, R_1, \dots, Q_{n-1}, R_{n-1}\}$, where each new pair of points Q_i, R_i is to be taken at a ratio of $\frac{1}{4}$ and $\frac{3}{4}$ between the endpoints of the line segment $\overline{P_i P_{i+1}}$. That is: $Q_i = \frac{3}{4}P_i + \frac{1}{4}P_{i+1}$, $R_i = \frac{1}{4}P_i + \frac{3}{4}P_{i+1}$. The new points

form a new control polygon, which is a refinement of the original control polygon. Figure 2.1 is an example illustrating Chaikin's algorithm.

It has been shown that the curve generated by Chaikin's algorithm is equivalent to a quadratic B-spline curve [11, 12]. Therefore, Chaikin's method actually provides a simple subdivision curve drawing mechanism without the need of analytical definition of B-splines.

Figure 2.1 Example of Chaikin's Algorithm

2.1.2 Subdivision surfaces

Representing complicated shapes is a fundamental problem in geometric design and game modeling. Building data structures and algorithms for generating, representing and manipulating complicated geometric shapes is a difficult problem. In SIGGRAPH '98, Pixar unveiled a short animated film, *Geri's game* [13]. The short film was a great success and defined how the new generation animation should be. The stunning visual effect and high detailed quality are the result of subdivision surfaces.

The idea of Chaikin's algorithm can be extended to surfaces, producing subdivision surfaces. The basic idea of subdivision is as follows: Subdivision defines a smooth curve or surface as the limit of a sequence of successive refinements [14]. A subdivision surface scheme, in the field of 3D computer graphics, is a method of representing a smooth surface via the specification of a coarse polygon mesh. This polygon mesh is known as a control net. The smooth surface can be calculated from the coarse mesh as the limit of a sequence of iterative processes of subdividing polygonal faces. The subdivision surface itself is perfectly smooth, unlike the control net. A simple example of subdivision is shown in Figure 2.2. The model in the left is the original model, and the rest two models are subdivided models.

Figure 2.2 Subdivision example

Speaking of subdivision surfaces, the technology is not new. The first two schemes, which are Doo-Sabin [15] and Catmull-Clark [16] schemes, were proposed in 1978. Many different schemes have been developed since then. Subdivision is mainly used in computer graphics. In particular, general subdivision surfaces schemes are mainly used in applications where the rendering is done offline. For example, subdivision surfaces can be used to generate highly detailed models from coarse meshes and to create cut scenes.

The benefit of subdivision method to game development is not only limited to the modeling phase. The most important reason why we want to apply subdivision surfaces in game is to generate more details from a coarse mesh. When a subdivision surface is applied in game applications, the game designers will not need to worry about how close the camera could be. The scalable geometry technique is an important part of developing next generation game engines. With subdivision surfaces, game designers can create low resolution models and the game engine will render high resolution ones by subdivision.

However, to use subdivision surfaces in real-time rendering for game applications is challenging. There is no standardized hardware for implementing subdivisions except for several specialized hardware including SGI hardware [17]. No current standard hardware includes a generic tessellation unit which could enable a GPU based surface refinement technique. The difficulty is mainly due to the lack of geometry generation. Geometry shader [18] introduced in DirectX 10 [19] and Shade Model 4.0 [20] could have some contribution but it is still very limited. Current generation GPU is optimized to render a large collection of geometric models, but it cannot handle the work to generate more polygons or vertices.

Therefore most of current schemes are to generate the refined model in CPU and memory, and then the whole mesh will be sent to GPU for rendering.

2.1.3 Loop subdivision

Loop subdivision is an approximating face-split scheme for triangular meshes proposed by Charles Loop [21]. The scheme is based on the three-directional box spline, which produces surfaces that are C^2 -continuous everywhere except at extraordinary points, where only G^1 (first-order geometric continuity, which means the tangents to each curve at the point of connection have the same direction) can be achieved. An extension to Loop subdivision scheme [9] was proposed with special rules defined for edges. The boundary rules were further improved, and new rules were introduced for concave corners and normal modification [22] was proposed later.

An illustration is shown in Figure 2.3. Given a triangular mesh, first for each edge, we compute the midpoint using the four adjacent vertices: $V_{mid} = \frac{3}{8}V_0 + \frac{3}{8}V_1 + \frac{1}{8}V_2 + \frac{1}{8}V_3$, where V_0, V_1 are the two vertices on the edges and V_2, V_3 are the two vertices on sides of the edge. After we insert these midpoints, we replace each old vertex by:

$$V' = V(1 - \alpha_n) + \sum_{i=1}^n (V_i \frac{\alpha_n}{n}), \text{ where } \alpha_n = \frac{1}{64} \left(40 - \left(3 + 2 \cos \left(\frac{2\pi}{n} \right) \right)^2 \right), \text{ and } n \text{ is the}$$

valence of the vertex. Then all these newly inserted vertices will be connected to form the new subdivision surface.

Figure 2.3 Illustration of Loop Subdivision

Figure 2.4 shows the example of Loop subdivision. The leftmost model is the original model, and the rest three models are generated from three steps of subdivision.

Figure 2.4 Example of Loop subdivision

Loop subdivision scheme is quite useful in game applications, because game models are usually presented as triangle models.

2.2 Curved PN triangles

In this section, Bézier triangles and curved PN triangles are explained.

2.2.1 Bézier triangle

Bézier triangles were first introduced by de Casteljau in 1963 [23], which are an extension of the Bézier curve ideas to surfaces. Bézier triangles are defined for triangular patches, which are direct generalization of his algorithm on curves, which was published in 1959 [24].

The control net of a Bézier triangle is of a triangular structure. In the cubic case, the control net consists of vertices:

$$\begin{array}{c}
 b_{030} \\
 b_{021} b_{120} \\
 b_{012} b_{111} b_{210} \\
 b_{003} b_{102} b_{201} b_{300}
 \end{array}$$

For all control points b_{ijk} , we can find that all subscripts sum to 3. In general, the control net of a Bézier triangle of degree n consists of $(n+1)(n+2)/2$ vertices. Figure 2.5 gives an example of a cubic patch with its control net.

For simplicity, we denote the point b_{ijk} by b_I , where $I = (i, j, k)$. Also let $e_1 = (1, 0, 0)$, $e_2 = (0, 1, 0)$, $e_3 = (0, 0, 1)$ and $|I| = i + j + k$. Then the de Casteljau algorithm [25] is described as follows:

Given: A triangular array of points $b_I \in E^3; |I| = n$ and a point p in E^2 with barycentric coordinates u . The **barycentric coordinates [26] of p** are the coordinates defined with

respect to the three vertices b_{00n}, b_{0n0} and b_{n00} of the triangle. For example if a point $p = (u \cdot b_{00n} + v \cdot b_{0n0} + w \cdot b_{n00})$, then we say (u, v, w) are the barycentric coordinates of p .

Figure 2.5 Bézier triangles: a cubic patch with its control nets

Set:

$$b_I^r(\mathbf{u}) = u b_{I+e_1}^{r-1}(\mathbf{u}) + v b_{I+e_2}^{r-1}(\mathbf{u}) + w b_{I+e_3}^{r-1}(\mathbf{u}), \quad |I| = n - r, \mathbf{u} = (u, v, w)$$

where $r = 1, \dots, n$ and $b_I^0(\mathbf{u}) = b_I$.

Then $b_0^n(\mathbf{u})$ is the point on the Bézier triangle corresponding to value $\mathbf{u}$.

2.2.2 Curved PN triangles

Alex Vlachos et al.[8] introduced a technique called Curved PN triangles, for increasing the smoothness of triangular meshes. The basic idea is to replace flat triangles with curved patches with higher order normal variation. Specifically, given three vertices and three vertex normals of a flat triangle, this technique will create a cubic Bézier triangle to replace the original flat triangle's geometry and a quadratically varying normal for Gouraud shading.

Basic form of curved PN triangle

The geometry of a curved PN triangle is defined by a cubic Bézier patch shown in Figure 2.6.

Figure 2.6 The control points of a triangular Bézier patch form a control net

The vertex on the curved PN triangle corresponding to barycentric coordinates (u, v, w) with $w = 1 - u - v$ is:

$$\begin{aligned}
 b(u, v, w) &= \sum_{i+j+k=3} b_{ijk} \frac{3!}{i!j!k!} u^i v^j w^k \\
 &= b_{300} u^3 + b_{030} v^3 + b_{003} w^3 \\
 &\quad + 3b_{210} u^2 v + 3b_{120} u v^2 \\
 &\quad + 3b_{201} u^2 w + 3b_{102} u w^2 \\
 &\quad + 3b_{012} v w^2 + 3b_{021} v^2 w \\
 &\quad + 6b_{111} w u v
 \end{aligned}$$

The coefficients b_{ijk} of the Bézier patch are grouped by:

- Vertex coefficients: $b_{300}, b_{030}, b_{003}$
- Tangent coefficients: $b_{210}, b_{120}, b_{021}, b_{012}, b_{102}, b_{201}$

- Center coefficients: b_{111}

An interesting aspect of the curved PN triangle is that the normal component is independent from the geometric component. The normal component of a curved PN triangle is a quadratic Bézier triangle whose control net is shown in Figure 2.7.

Figure 2.7 Normal components of curved PN triangles

The quadratic Bézier triangle is defined as:

$$\mathbf{n} : R^2 \rightarrow R^3, w = 1 - u - v, \quad u, v, w \geq 0$$

$$\begin{aligned} \mathbf{n}(u, v, w) &= \sum_{i+j+k=2} n_{ijk} u^i v^j w^k \\ &= n_{200} u^2 + n_{020} v^2 + n_{002} w^2 \\ &\quad + n_{110} uv + n_{011} vw + n_{101} uw \end{aligned}$$

Determining PN triangles

The input data for a curved PN triangle are three vertices P_i and three normals N_i as shown in Figure 2.8.

Figure 2.8 Input points and normals

The Bézier triangles are determined by the following rules:

- Spread the b_{ijk} uniformly over the flat triangle, and place b_{ijk} in the intermediate position $(iP_1 + jP_2 + kP_3)/3$
- Keep corner control points unchanged, so triangles will match in border
- For each corner, project the two tangent coefficients closest to the corner into the tangent plane defined by the normal at the corner. Here projection means to find the closest point on the plane to the point.
- Move the center coefficient from its intermediate position V to the average of all six tangent points and continue its motion in the same direction for $1/2$ the distance already traveled.
- Since linearly varying normals ignore inflections in the geometry as illustrated in left image of Figure 2.9, the quadratic normal variation should be used (see the right of Figure 2.9).

Figure 2.9 Normal calculation of PN triangles[8]

Thus the formulae for the geometry control points of the PN triangle are:

$$\begin{aligned}
 b_{300} &= P_1, \\
 b_{030} &= P_2, \\
 b_{003} &= P_3, \\
 w_{ij} &= (P_j - P_i) \bullet N_i, \\
 b_{210} &= (2P_1 + P_2 - w_{12}N_1)/3, \\
 b_{210} &= (2P_2 + P_1 - w_{21}N_2)/3, \\
 b_{210} &= (2P_2 + P_3 - w_{23}N_2)/3, \\
 b_{210} &= (2P_3 + P_2 - w_{32}N_3)/3, \\
 b_{210} &= (2P_3 + P_1 - w_{31}N_3)/3, \\
 b_{210} &= (2P_1 + P_3 - w_{13}N_1)/3, \\
 E &= (b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201})/6, \\
 V &= (P_1 + P_2 + P_3)/3, \\
 b_{111} &= E + (E - V)/2
 \end{aligned}$$

The normal PN triangle is defined by:

$$\begin{aligned}
 n_{200} &= N_1, \\
 n_{020} &= N_2, \\
 n_{002} &= N_3, \\
 v_{ij} &= 2 \frac{(P_j - P_i)(N_i + N_j)}{(P_j - P_i)(P_j - P_i)}, \\
 n_{110} &= h_{110}/\|h_{110}\|, h_{110} = N_1 + N_2 - v_{12}(P_2 - P_1), \\
 n_{011} &= h_{011}/\|h_{011}\|, h_{011} = N_2 + N_3 - v_{23}(P_3 - P_2), \\
 n_{101} &= h_{101}/\|h_{101}\|, h_{101} = N_1 + N_3 - v_{31}(P_1 - P_3)
 \end{aligned}$$

Curved PN triangles rendering

The original technique[8] introduced was implemented on hardware graphics pipeline.

The hardware implementation steps are shown in Figure 2.10. This scheme can be transformed to software simulation easily.

Figure 2.10 Curved PN triangle tessellation in the graphics pipeline

Observation

The technique of Curved PN triangles provides a fast solution to improving the visual quality of triangle-based real-time rendering. The speed is decent and the image quality is acceptable. However, curved PN triangles technique as an extension of OpenGL is only provided for special hardware like ATI 8500. Without hardware support, the software emulation speed could be slow. General game developers must make sure that the techniques are available to most hardware.

2.3 GPU tessellation

2.3.1 Graphics Processing Unit

The technical definition of a GPU is "a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second" [27]. During the first two decades in the 1970s and 1980s after the birth of Graphics accelerators, GPU was only used to implement 2D primitives and later provide 2D acceleration for windows operating system. Since the early and mid 1990s, real-time 3D graphics was becoming common in computer and consoles games; however most of the processing was done by the CPU. Meanwhile 3D graphics chips were introduced to accelerate 3D functions. OpenGL and DirectX were introduced by SGI and Microsoft respectively. These friendly APIs helped developers to implement powerful applications easily.

August 31, 1999 [28] marks the introduction of the Graphics Processing Unit (GPU) for the PC industry. A GPU can be built into a video card or it can be integrated into the motherboard. GPUs can now be found on every modern desktop or notebook computers.

Modern GPUs are mostly used to perform calculations related to 3D computer graphics, such as texture mapping, rendering polygons, geometric calculations and anti-aliasing etc. Among these functions provided by GPU, programmable shader is the most important feature of modern GPU.

2.3.2 General-Purpose computation on Graphics Processing Units

The latest graphics architectures provide tremendous memory bandwidth and computational horsepower, with fully programmable vertex and pixel processing units

that support vector operations up to full IEEE floating point precision [29]. GPGPU (General-Purpose computation on Graphics Processing Units) has become an important part of both academia and industry.

Since GPU provides significant advantage in parallel processing, a number of important algorithms and schemes have been introduced in recent years. For example, particle systems [30, 31], fluid simulation [32-34], cloud dynamics [35, 36], cloth simulation [37], photon mapping [38], collision [39, 40], Global Illumination [41] and general computations [42, 43].

2.3.3 Shader

A shader in the field of computer graphics is a set of software instructions, which is used primarily to calculate rendering effects on graphics hardware with a high degree of flexibility. Shaders are used to program the graphics processing unit (GPU) programmable rendering pipeline, which has mostly superseded the fixed-function pipeline that allows only common geometric transformation and pixel shading functions. With shaders, customized effects can be used [44]. The most frequently discussed and used shaders are vertex shaders and pixel shaders.

A vertex shader replaces part of the geometry stage of graphics pipeline. Vertex shaders take input vertex data and use specified kernel to process each vertex. The processing kernel can also use input variables from applications to operate on the vertices. Each vertex processor always produces a single transformed vertex from the source vertex information. Figure 2.11 illustrates how vertex shaders work.

Figure 2.11 Vertex shader

Pixel shaders are similar to vertex shaders; they can also use input variables from application to calculate the final color of certain pixel and the input from GPU is merely fragments values. Pixel shaders determine the color of pixel. Figure 2.12 illustrates the mechanism of pixel shaders.

Figure 2.12 Pixel shader

GPU processing can be very fast because most modern GPUs have a multi-core design, and there are hundreds of shader units in a single GPU chip, which greatly improve the efficiency of processing.

2.3.4 Level of details

In computer graphics, *level of detail* (LOD) is a technique that controls the complexity of a 3D object representation based on the distance between object and the viewer or according to some other metrics such as object importance, eye-space speed or position [45, 46]. This *level of detail* technique increases the efficiency of rendering by decreasing the workload on graphics pipeline stages.

A discrete LOD example is shown in Figure 2.13. For distant objects, we use low-detailed models. For close-up objects, we use high-detailed models.

Figure 2.13 A discrete LOD example

Subdivision surfaces will make LOD (level of details) implementation quick and easy. Depending on which level the current subdivision needs to be at the game engine can generate the refined model on the fly. Using subdivision in LOD can significantly reduce the memory usage, loading time and workloads of game designers, because only original coarse meshes are loaded in memory (normal LOD implementation requires a model tree

for each model to be loaded into memory). Ordinary LOD can only store a limited number of models, so the graphics will eventually look blocky when camera is close enough. In contrast subdivision provides continuous transformation and can have as many levels as required.

2.3.5 Generic mesh refinement on GPU

Tamy Boubekeur et al. [47] proposed a simple but effective way to implement various mesh refinement schemes using GPU.

The main idea of this approach is to use a single refinement pattern in GPU for each polygon shape that exists in the mesh. The refinement pattern will be sent to GPU once for all as a vertex buffer. Let A_T be the set of attributes of a coarse triangle T . A_T contains vertex positions, vertex normals and other information such as colors. The algorithm is:

```
render (mesh  $M$ )
```

```
  for each coarse triangle  $T$  of  $M$  do
```

```
    place  $A_T$  as uniform variables for the vertex shader
```

```
    draw refine pattern
```

Figure 2.14 Illustration of generic GPU refinement process[47]

The illustration is shown in Figure 2.14. Every triangle in the coarse mesh (a) will be rendered using the refinement pattern (b) to generate the final outcome (c).

This generic mesh refinement is very straightforward but very useful in real applications such as games. We can apply this approach for curved PN triangles. However, the generic mesh refinement is not suitable for LOD design. This is because it uses only one specified pattern at one time.

2.3.6 Adaptive mesh refinement on GPU

Tamy Boubekur et al. also proposed an improved version of their generic mesh refinement. In paper [48], an adaptive refinement scheme was presented to implement refinement. The basic idea of this improved scheme is to generate depth tags first for the models and then it chooses adaptive refinement patterns according to the depth tags.

The depth tag or the LOD level is determined by the application on the CPU side. The depth tag of each vertex should be recomputed at each frame if application requires. The face depth-tag is computed as the average of the surrounding edge depth-tags, while all the edge depth-tags are computed simply by taking the arithmetic average of the two adjacent vertex depth-tags. Figure 2.15 illustrates two types of depth tag configuration. One is to use distance to determine the depth and the other is to use curvature to determine the depth.

Figure 2.15 Depth tag configuration of adaptive mesh refinement[48]

The adaptive refinement patterns are pre-computed and stored in GPU. For triangular meshes, the total number of all possible tessellation patterns is l^3 if the maximum tessellation level is l . Figure 2.16 illustrates the matrix of adaptive refinement patterns. This matrix stores all the possible refinement patterns that may be used. The matrix of adaptive refinement patterns contains all possible combinations of different depth levels. For example if the required depth level is 10, the matrix will contain 1000 different refinement patterns.

Figure 2.16 The matrix of adaptive refinement patterns[48]

The whole process is: First the application pre-computes all the possible tessellation patterns for the deepest refinement level specified and stores these patterns into GPU memory. Second the depth-tags are recomputed at each frame according to application control, and then corresponding refinement patterns will be used according to each triangle's depth-tags. The workflow architecture is illustrated in Figure 2.17.

Figure 2.17 Architecture of adaptive refinement scheme[48]

The main drawback of this approach is the size of graphics memory consumed by adaptive refinement patterns. As the refinement level goes up, the required space will have a cubic growth which greatly limits the depth level. In general game application and animation rendering software, the maximum depth should be more than ten. And this will require a rather large matrix containing 1000 different adaptive refinement patterns.

2.4 Other real-time refinement techniques

Many works have been done in recent years to compute subdivision surfaces using GPU. For example, Loop subdivision surfaces [49, 50] based on [51], Catmull-Clark subdivision surfaces [52]. To achieve subdivision surfaces on GPU is still cost-intensive, Charles Loop et al. proposed an approximating method of using bicubic patches [53] (using patching to approximate Catmull-Clark subdivision surfaces was first proposed by [54]). Denis Kovacs et al. improved Loop's scheme [55] with creases and corners. Bolz et al. [56] and Shiue et al. [51] proposed techniques for direct evaluation of subdivision surfaces on GPUs. Most of these approaches require additional subdivision steps or multiple passes [55, 57]. In SIGGRAPH 2009, Ni et al. [58] provided a course note to discuss how to use smooth surfaces that facilitate modeling and animation.

Game industry shows great interest in subdivision surfaces and other surface refinement techniques in recent years. The first study on how to apply subdivision surfaces on game was conducted by Sharp [59]. In this article Sharp studied butterfly subdivision scheme [60] and Catmull-Clark subdivision scheme. Sharp concluded there was no dearth of information on subdivision surfaces for direct use in game industry. Game developers need a simple solution to achieve smooth surfaces in game applications other than depending on existing functionality of modeling application.

Chapter 3

Loop subdivision over multiple meshes for games

This chapter describes an approach of applying Loop subdivision surfaces over multiple meshes. The implementation of Loop subdivision based on commercial game engine is presented.

3.1 Loop subdivision surfaces over multiple meshes

3.1.1 Surface matching

Mesh surfaces used in game applications are usually low detailed coarse mesh. Loop subdivision can be used to improve the smoothness of mesh surfaces in games. However, conventional subdivision surfaces use a single control mesh for the whole surface. But in actual game modeling, an object in game can consist of multiple meshes. The left image of Figure 3.1 and Figure 3.2a illustrate objects containing multiple meshes.

Figure 3.1 A single surface contains multiple textures

A simple approach is to apply Loop subdivision on each mesh independently, but this usually causes visible cracks as shown in right image of Figure 3.1 and Figure 3.2b. This is because after subdivision each patch will shrink and cannot match each other's border. To overcome this problem, we can construct a single control mesh for the entire surface to prevent the crack being generated. However this requires a lot of additional work for game modelers.

Figure 3.2a Multi-texture surface models in games (rocks)

Figure 3.2b Visual cracks when subdivision is applied on multi-texture surface

Another possible approach is to mark the border edges between different patches in the initialization step, then apply subdivision to each patch, and rejoin the border edges back after each subdivision. We call it “averaging” approach which is illustrated in Figure 3.3 and the result is shown in Figure 3.4. The averaging approach is simple. It does not need to find a single control mesh. However this approach will decrease the continuity order and usually give G^0 along the border, which is not favorable.

The basic technique of the averaging approach is to record all the vertices which are at the exactly same position during model loading. These vertices will be recorded as a linked list, **pPrevCorrelatedVertex** and **pNextCorrelatedVertex**, which are the specific data members to store them. After the subdivision process is completed, all these vertices will be accessed, and the average of them will be calculated. The average value will be assigned back to all vertices. Hence the gaps problem is solved. The result image may contain sharp edges since the joint will be only G^0 .

Figure 3.3 Crack solution

Figure 3.4 Result of the average approach

We denote the **inner border** as the sharing border of different patches. Then the implementation steps are shown below:

- 1: Mark all vertices and edges sharing the inner border
- 2: Make a linked list for each group of vertices on the inner border which have the same position

- 3: Apply subdivision on each patch, and the newly generated vertices will inherit the border sharing information and form their own linked list to calculate the average.
- 4: For each inner border vertices linked list, calculate the average of all vertices, and then assign this average vertex to all vertices in the linked list

3.1.2 Smooth surface matching

Note that the averaging approach gives only G^0 continuity along the inner border. To overcome this deficiency, here we propose a new approach, called the extension approach, which is to extend adjacent border edges by one layer. That is, for each patch of an object surface, we enlarge the triangular mesh by one layer, which is obtained from the mesh of the neighboring patches. Then we apply Loop subdivision to each extended triangular mesh independently. After the required subdivisions are done, the triangle strips [61] on the outer-most layer will be removed, and the remaining parts will form the refined mesh. The illustration is shown in Figure 3.5.

Figure 3.5 Illustration of the extension approach

Based on the property of Loop subdivision, the extension approach will automatically maintain the continuity of loop subdivision C^2 except for the extraordinary points where G^1 can be achieved. The comparison of averaging and extension methods will be showed in next section; we can easily find that the extension approach is smoother.

3.2 Implementation of Loop subdivision over multiple meshes

In this research, we have implemented Loop subdivision in a commercial racing game engine which was used to create the game, Baja Edge of Control [62]. This XBOX360/Playstation 3 game has been released in 2008 by THQ (Toy Head-Quarters Inc.). To best suit general game design, the implementation does not require remodeling of the meshes. So basically this implementation process could be applied in any game development with only minor modifications. In the following, we describe the details of the implementation.

3.2.1 Winged-edge data structure

The winged-edge data structure is a description of the connections between vertices, edges and polygons. Its most common use is for describing geometric polygon meshes. The implementation [63] given by Bruce G. Baumgart is a data structure composed of small blocks of words containing pointers and data in graphics and simulation. Figure 3.6 illustrates the Winged-edge data structure.

First we take a look at edge a which has two incident vertices X and Y . Please note that all edges in Winged-edge data structure are directional and edge a starts from vertex X

and ends at vertex Y. The left face is 1 and the right face is 2. The ordering is clockwise viewed from outside of the solid, so on face 2, the predecessor and successor of edge **a** are edge **e** and edge **c** for right traverse.

Figure 3.6 Winged-edge data structure [64]

The edge table of the direction edge *a* from X to Y is shown in Table 3.1, which shows the important information of edge **a**.

<i>Edge</i>	<i>Vertices</i>		<i>Faces</i>		<i>Left Traverse</i>		<i>Right Traverse</i>	
<i>Name</i>	<i>Start</i>	<i>End</i>	<i>Left</i>	<i>Right</i>	<i>Pred</i>	<i>Succ</i>	<i>Pred</i>	<i>Succ</i>
<i>a</i>	X	Y	1	2	<i>b</i>	<i>d</i>	<i>e</i>	<i>c</i>

Table 3.1 Edge table[64]

Each edge entry contains information: edge name, start vertex and end vertex, left face and right face, the predecessor edge and successor edge when traversing its left face, and the predecessor edge and successor edge when traversing its right face. Clockwise ordering is used for traverse. We only need to record one face and one traverse in actual implementation of Winged-edge data structure.

3.2.2 Modified Winged-edge data structure

To provide better support for our loop subdivision implementation, we modify the classic winged-edge data structure. The key idea of this modification is to make the Winged-Edge data structure be more suitable to our approach. The advantages of this modified

Winged-edge data structure are: (I) simplicity and efficiency for Loop subdivision in games, (II) compatibility to real game development. The class diagram of the modified winged-edge data structure is shown in Figure 3.7. The extra information added in each class is discussed below. The newly introduced data members are highlighted in ***bold italic*** type.

Figure 3.7 Class diagram of modified winged edge data structure

For each single face, we only need to record one edge, which we use a pointer **pEdge** to record, among all the edges of this face. The next face is just a pointer to point to the next face in the linked list which is only used for recording and traversing all the faces, and there is no topological meaning of this **pNextFace**.

For each single vertex, two Boolean values are used. One is used to tell whether the vertex is on the border or not, and the other one is used to record the status of subdivision. The value **newPoint** will be set to false before during the initialization of subdivision, and true will be given to the newly generated points. The data member **vertexData** is used to stored information such as actual position, texture coordinates and color etc. Similar to **Face**, we also use a pointer **pNextVertex** to maintain a linked list of vertices for easy traversing of all vertices. The rest members, **pPrevCorrelatedVertex**, **pNextCorrelatedVertex** and **pAverageDummy**, will be discussed in later section.

For each edge, we record the standard information like start vertex, end vertex, previous edge, next edge and one face. The **pPair** is used to record the edge which has the same collection of start vertex and end vertex but that the direction of pair edge is opposite to original edge. Similar to that for the vertices, we also use two Boolean values to record border information and subdivision changes.

This modified version of winged-edge data structure is implemented primarily to apply Loop subdivision on the racing game engine of Baja Game[62]. Two functions of **WingedEdge** class, **FirstInsert** and **SecondInsert**, are used to subdivide surface. The **FirstInsert** function will insert all the newly generated vertices into **WingedEdge** data structure. The **SecondInsert** will connect all these newly generated vertices and form new edges and new faces.

Basic function of modified Winged-edge data structure

To implement Loop subdivision, the data structure should be able to provide some basic functions like finding all adjacent vertices of a vertex. We will show how our modified Winged-edge data structure works.

To find all the surrounding vertices of one vertex, we can start from the edge pointer of the vertex. Figure 3.8 illustrates how we can do the traversing. The pseudo code is:

```
1: For each vertex v, get its edge pointer ptr_edge
2: Make two pointers for traverse: tmp_edge, tmp_vertex
3: tmp_edge = ptr_edge, tmp_vertex = v
4: If v is not on border
    Do
        tmp_edge = tmp_edge->pPairEdge->pNextEdge
        tmp_vertex = tmp_edge->endVertex
        while(tmp_edge != ptr_edge)
5: If v is on border
    While(tmp_edge is not on border)
        tmp_edge = tmp_edge->pPairEdge->pNextEdge
        tmp_vertex = tmp_edge->endVertex
    tmp_edge = ptr_edge->prev
    While(tmp_edge is not on border)
        tmp_edge = tmp_edge->pPairEdge->pPrevEdge
        tmp_vertex = tmp_edge->endVertex
```

Figure 3.8 Traversing surrounding vertices within the Winged-edge data structure

3.2.3 Loop subdivision over a single mesh

We first discuss the implementation of Loop subdivision over a single mesh. The basic procedure is shown in Figure 3.9. We build the Winged-edge data structure during the loading of game models. For each single surface, we build two copies of Winged-edge data structure. The original copy is stored and kept unchanged, and we only do subdivision operations on the working copy of the Winged-edge data structure. The original copy of Winged-edge data structure is used to easily rollback to base LOD level.

The distance from camera to model is used to control LOD. When camera position is modified or the game requires higher details, the game engine triggers the subdivision. The working copy of Winged-edge data structure will be subdivided first, and then we generate a new triangle list from the newly subdivided Winged-edge data structure copy and send the triangle list to GPU memory for rendering.

Figure 3.9 Loop subdivision implementation

3.2.4 Loop subdivision over multiple meshes with different textures

The averaging matching is discussed in Chapter 3.1.1. We just apply subdivision on each different patch and then calculate the average values of border vertices. These mean values will be assigned back to border vertices and different patches are connected with each other. The result is shown in Figure 3.11.

In the implementation of the smooth extension, we made some storage optimization so we have no need to store those extra triangle strips for each patch. The optimized scheme is illustrated in Figure 3.10. For every vertex on the border, we mark the adjacent vertex as its shadow pair. When we apply subdivision on whole surface, we use the shadow pair vertex to trace all the vertices on the adjacent patch. So after subdivision all patches are virtually connected to each other.

We denote the **shadow pair edge** as the pair edge sharing the inner border, the steps are shown below:

- 1: Mark all vertices and edges sharing the inner border
- 2: Assign the edge sharing the inner border as the shadow pair edge of original edge
- 3: When we apply subdivision on each patch, if the vertex is on the inner border, we use the shadow pair information to find those additional vertices information on the adjacent patch.

Figure 3.10 Smooth extension implementation

The comparison of these two schemes is shown in Figure 3.11 and Figure 3.12. We can clearly see the sharp edges produced by the *averaging matching method* (marked by red arrows in Figure 3.11) have been replaced by smoother edges (marked by red arrows in Figure 3.12).

Figure 3.11 Averaging matching

Figure 3.12 Smooth extension

Chapter 4

New adaptive GPU PN triangles tessellation

This chapter investigates an alternative solution to achieving smooth mesh in real-time other than the traditional subdivision method. We propose to use PN triangles. First we discuss how to apply the Curved PN triangles on general GPU, and then we will propose a novel adaptive scheme to render PN triangles using GPU.

4.1 Curved PN triangles on GPU

4.1.1 Overview

To generate complex geometric models is very important in computer graphics. Two techniques for generating complex geometry are subdivision surfaces and displacement mapping. However both of these two techniques are not fast enough for real-time rendering because current generation graphics hardware does not support real-time geometry generation, and to generate those extra details in CPU is not feasible because of the processing speed limitation. Basically subdivision surfaces and displacement mapping are mainly used in offline rendering such as animation and movie or modeling software such as 3D Studio Max.

The technique “curved PN triangles”[8] is a low cost solution to improve the visual quality of 3D objects in game applications. However this technique is only available to

specific hardware for example ATI Raedon 8500. In this section, we will discuss the implementation of PN triangles technique using generic GPU refinement which is discussed in Chapter 2. Our implementation is a combination of two techniques introduced in Chapter 2, curved PN triangles and Generic mesh refinement on GPU[47]. The main idea is to use GPU to render curved PN triangles.

4.1.2 Design

The design emphasizes on GPU to accelerate rendering speed in favor of game applications. The control mesh generation follows the same step as by [8], which has been discussed in Chapter 2. The construction of the control mesh will be done during the initialization period. The refinement pattern shown in Figure 4.1 will be used to merely record the topology. Here, to simplify the implementation, we merely use triangle lists rather than triangle strips.

Figure 4.1 Refinement patterns

GPU shaders will be used to process mesh refinement instead of doing it in the CPU. The processing time will be reduced due to parallel capability of the GPU. The process is illustrated in Figure 4.2. The actual refinement mesh generation will be calculated using GPU shaders. The basic idea here is to put the refinement calculation of each triangle into GPU. We compare this approach with the original PN triangles implementation[8] which is shown in Figure 4.3 and we can see that this implementation reduces the memory copy

from CPU to GPU and accelerate the calculation by transferring the calculation from CPU to GPU.

Figure 4.2 PN triangles technique on GPU

Figure 4.3 Original PN triangles technique

4.1.3 Implementation

The implementation can be illustrated from the steps below.

- 1: Generate refinement patterns in GPU memory
- 2: Construct PN triangles control net information
- 3: Adjust LOD, when LOD changes point working refinement pattern pointer to suitable refinement pattern available in GPU
- 4: For each triangle, send the control points/normals into GPU
- 5: Render specified refinement pattern using the PN triangle's Information

Figure 4.4 gives a more clear illustration. When the level of details for the current triangle is 3, we select the third refinement pattern in the GPU memory, and render every triangle of the coarse mesh using the selected refinement pattern.

Figure 4.4 GPU curved PN triangles implementation

In particular, the following shader code segment shows how we implemented the Curved PN triangles in GPU's vertex shader. The control net information is passed as input, and for each PN triangles we have 16 float3 variables which are ten control points $v[10]$ and six control normal points $n[6]$. The heaviest calculation of curved PN triangles technique is now handled by GPU, the code segment of our implementation is using DirectX's

HLSL [65]. The TEXCOORD0 (Texture Coordinate 0) c is a float array whose size is three, and the float array stores barycentric coordinates of each vertex.

```
Transform.fx

//shader effect file for GPU PN triangles

.

//input

uniform extern float3 v[10];
uniform extern float3 n[6];

OutputVS TransformVS(float3 c : TEXCOORD0)
{
    OutputVS outVS = (OutputVS)0;

    float3 temp = v[0] * pow(c.x,3) + v[1] * pow(c.y,3) + v[2] *
pow(c.z,3)

                                + 3* pow(c.x,2) * (c.y * v[3] + c.z * v[4])
                                + 3* pow(c.y,2) * (c.x * v[5] + c.z * v[6])
                                + 3* pow(c.z,2) * (c.x * v[7] + c.y * v[8])
                                + 6* v[9]*c.x*c.y*c.z;

    outVS.posH = mul(float4(temp, 1.0f), gWVP);

    return outVS;
}

.
```

4.1.4 Examples

Here we provide some examples to demonstrate our algorithm. Figure 4.5 shows the algorithm applied to a cube. From these images we can find that the curved PN triangles maintains the shape of the cube, which is very similar to the result of butterfly subdivision

[60] — an interpolating subdivision scheme. Figure 4.6 shows the implementation applied to a sphere. Figure 4.7 shows the implementation applied to a teapot where solid rendering is used. The result of the cube model shows one weakness of our current approach, which is our method can not define wanted sharp edges. All sharp edges will be removed so useful details may be discarded.

Frames Per Second = 1423.00
Milliseconds Per Frame = 0.7027
Triangle Count = 12
Vertex Count = 12
Debug Info: Level: 1

Frames Per Second = 1376.00
Milliseconds Per Frame = 0.7267
Triangle Count = 4332
Vertex Count = 2280
Debug Info: Level: 19

Frames Per Second = 465.00
Milliseconds Per Frame = 2.1505
Triangle Count = 167088
Vertex Count = 84252
Debug Info: Level: 118

Figure 4.5 GPU curved PN triangles on a cube

Frames Per Second = 443.51
Milliseconds Per Frame = 2.2548
Triangle Count = 360
Vertex Count = 1080
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 6
Debug Info: Level: 1
Lighting: per Vertex

Figure 4.6a GPU curved PN triangles on a sphere, depth level 1

Frames Per Second = 382.90
Milliseconds Per Frame = 2.6116
Triangle Count = 70560
Vertex Count = 43200
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 6
Debug Info: Level: 14
Lighting: per Vertex

Figure 4.6b GPU curved PN triangles on a sphere, depth level 14

Frames Per Second = 99.16
Milliseconds Per Frame = 10.0849
Triangle Count = 576000
Vertex Count = 309960
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 6
Debug Info: Level: 40
Lighting: per Vertex

Figure 4.6c GPU curved PN triangles on a sphere, depth level 40

Frames Per Second = 197.70
Milliseconds Per Frame = 5.0581
Triangle Count = 992
Vertex Count = 2976
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 2
Debug Info: Level: 1
Lighting: per Vertex

Figure 4.7a GPU curved PN triangles on a teapot, depth level 1

Frames Per Second = 194.49
Milliseconds Per Frame = 5.1417
Triangle Count = 3968
Vertex Count = 5952
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 2
Debug Info: Level: 2
Lighting: per Vertex

Figure 4.7b GPU curved PN triangles on a teapot, depth level 2

Frames Per Second = 194.83
Milliseconds Per Frame = 5.1326
Triangle Count = 99200
Vertex Count = 65472
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 2
Debug Info: Level: 10
Lighting: per Vertex

Figure 4.7c GPU curved PN triangles on a teapot, depth level 10

This algorithm in general generates satisfactory results as shown in Figure 4.5, Figure 4.6 and Figure 4.7. The test GPU used is ATI Raedon 4850. The rendering speed is decent while the surface is smooth. Since curved PN triangles only require local information, the speed of rendering is quite decent even when the LOD level is very high. The examples show we can achieve roughly subdivision quality rendering in real-time rendering.

4.1.5 Performance

This implementation has been analyzed by comparing the rendering frame rate per second (FPS) of two meshes, a cube and a teapot, either by using original PN triangles or by using GPU PN triangles. To simulate original PN triangles method, we can just use the software processing of DirectX. Our configuration for the testing is a PC equipped with Intel Core 2 DUO E8400 3.0GHz, 4G DDR2 RAM and an ATI HD4850 which is a commercial level graphics card. The result of this test is shown in Table 4.1. The left column shows the model name, LOD level, number of triangles and number of vertices, and the rest two columns show the rendering frame rate per second (FPS) of these two approaches.

Model LOD t=triangle count, v= vertex count	Original PN FPS	GPU PN FPS	GPU/ Original
Cube Level 1, t = 12, v = 36	3265.53	3605.98	110.4%
Cube Level 10, t = 1200, v = 792	2541.42	3485.43	137.1%
Cube Level 100, t = 120000, v = 61812	215.98	1385.31	641.4%
Teapot Level 1, t = 992, v = 2976	133.89	204.18	152.5%
Teapot Level 10, t = 99200, v = 65472	70.60	204.16	289.2%
Teapot Level 100, t = 9920000, v = 5109792	3.11	30.47	979.7%

Table 4.1 GPU curved PN triangles performance

The result of this experiment clearly shows that the GPU PN triangles approach is significantly faster than the original method, and the relative speed is dramatically increased when LOD level becomes higher and the total number of vertices and triangles increases. The huge speed increase is due to the use of the GPU where all the intensive calculations are being processed, and performing the mesh refinement on GPU also reduces the amount of data needed to be transferred from CPU to GPU.

4.1.6 Observation

It can be seen that the combination of curved PN triangles and generic mesh refinement can produce a fairly good result. The implementation itself is very straightforward and easy to code. The program can be used on arbitrary standalone graphics card.

The main limitation of this implementation is that the LOD scheme is not adaptive. This will produce popping effects [66] and also waste processing power. For surfaces containing points with high curvature and also points with low curvature, we should use different tessellation levels. In next section we will propose a novel scheme to solve this problem.

4.2 Adaptive Curved PN triangles on GPU

4.2.1 Overview

Tamy Boubekur and Christophe Schlick have improved their generic mesh refinement method to incorporate adaptive refinement[48], which we have described in Chapter 2. However, this adaptive refinement scheme cannot be applied when very high LOD is required, since the needed graphics memory has a cubic growth rate against the maximum

depth level. To address this issue, here we present a novel scheme for adaptive refinement and GPU PN triangles.

The adaptive PN triangle scheme presented in this section offers the following properties:

- Only limited graphics memory will be required, and the actual amount is the same as our GPU PN triangles implementation in Chapter 4.1. For each LOD, we only need one refinement pattern in GPU. Thus for N levels, we will store N refinement patterns in GPU. All these refinement patterns are generated in the initialization stage.
- For each frame, the control mesh and current required LOD level will be transferred from CPU to GPU. The bandwidth needed from CPU to GPU is very low.
- All calculations are processed in GPU using vertex shaders.

The workflow of our scheme is described in Figure 4.8. We have added several parameters like face depth and edge depth, which will be generated during the construction of PN triangles control mesh. The key idea here is to use these depth values to select suitable refinement patterns and to modify the pattern to match the borders during rendering, so we can reduce the total number of refinement patterns required. The technical details are explained in the following sections.

Figure 4.8 Adaptive PN triangles

4.2.2 Depth-tagging

The purpose of depth-tagging is to determine the required LOD for each edge and each face. The depth values of triangles and edges are generated during the initialization stage. For each triangle on the surface, one face depth and three edges' depth will be generated,

and these depth values will be used to represent the curvature level of the surface and determine needed LOD level during rendering.

To compute depth values, we need the control points of the Bézier triangles, whose construction has been discussed in Chapter 2.2. In particular, we need the 10 point coefficients of the Bézier triangle. The six normal coefficients are not needed. Figure 4.9 shows the ten point coefficients generated from a triangle.

Figure 4.9 Coefficients of Bézier Triangle

Edge depth

We denote the error level by ε . For every curve segment $P(t)$, $t \in [a, b]$, define $h(t)$ to be the distance between the point $P(t)$ on the curve and its corresponding point on the straight line segment connecting $P(a)$ and $P(b)$, as shown in Figure 4.10. Then we have [67]:

$$h(t) \leq \frac{\max |P''(t)| \Delta t^2}{8} \quad (1)$$

Therefore if $\frac{\max |P''(t)| \Delta t^2}{8} \leq \varepsilon$, $h(t) \leq \varepsilon$ for all $t \in [a, b]$, which means that we can

replace the curve segment by the straight line segment.

Figure 4.10 Calculate the distance for edge

Now we determine the **edge depth for an edge** of a Bézier Triangle:

Figure 4.11 shows the four control points Q_0, Q_1, Q_2, Q_3 of an edge.

Let $M = 6 * \max(\|Q_2 - 2Q_1 + Q_0\|, \|Q_3 - 2Q_2 + Q_1\|)$. Then $\max |P''(t)| \leq M$.

Define an integer m as the edge depth. If we let $\Delta t = \frac{1}{m}$, and substitute M and m into

(1), we have:

$$\frac{M \left(\frac{1}{m} \right)^2}{8} \leq \varepsilon \quad (2)$$

$$\text{Thus } m \geq \sqrt{\frac{M}{8\varepsilon}} \quad (3)$$

Figure 4.11 Four control points of an edge

Face depth

Similarly, for a C^2 continuous parametric surface $r(u, v)$ shown in Figure 4.12, define

$h(u, v)$ to be the distance from the point $r(u, v)$ to flat bottom surface. We have [67]:

$$h(u, v) \leq \frac{\max |r_{uu}| \Delta u^2 + 2 \max |r_{uv}| \Delta u \Delta v + \max |r_{vv}| \Delta v^2}{8} \quad (4)$$

If $r(u, v)$ is a Bézier Triangle, we define M_1, M_2, M_3 :

$$\begin{aligned} M_1 &= 6 \max \|P_{i+2,j,k-2} - 2P_{i+1,j,k-1} + P_{ijk}\| \\ M_3 &= 6 \max \|P_{i,j+2,k-2} - 2P_{i,j+1,k-1} + P_{ijk}\| \\ M_2 &= 9 \max \|P_{i+1,j+1,k-2} - P_{i+1,j,k-1} - P_{i,j+1,k-1} + P_{ijk}\| \end{aligned} \quad (5)$$

$$\text{Then } h(u, v) \leq \frac{M_1 \Delta u^2 + 2M_2 \Delta u \Delta v + M_3 \Delta v^2}{8}$$

We define n as the **face depth**. If we let $\Delta u = \Delta v = \frac{1}{n}$, we have

$$\frac{(M_1 + 2M_2 + M_3)}{8n^2} \leq \varepsilon \quad (6)$$

$$\text{Thus } n \geq \sqrt{\frac{(M_1 + 2M_2 + M_3)}{8\varepsilon}}$$

Figure 4.12 Calculate the distance for surface

4.2.3 Face matching

The depth tags generated for faces and edges in Section 4.2.2 have the property that the edge depths are always smaller than or equal to the face depth for each Bézier triangle. During the rendering, each Bézier triangle will be rendered using the refinement pattern r_n , where n is the face depth of the Bézier triangle. For two adjacent patches with different face tags, cracks will be generated if we do not do any modification. For example, the left two refined meshes in Figure 4.13 are generated from two triangles with face depth 4 and face depth 3. To avoid the crack caused by different face tags we need to modify the edges to match these refined meshes. The modification is done based on the edge's depth value. If the shared edge of these two triangles has an edge depth of 2, we will remap those vertices on the border to new locations, as illustrated in Figure 4.13 (right).

In general, for face depth n and edge depth m , we need to map $i = 0, 1, \dots, n-1, n$ to $j = 0, 1, \dots, m-1, m$. We try to minimize the difference of $\frac{i}{n}$ and $\frac{j}{m}$. Letting

$mi - nj = 0$ suggests

$$j = \left\lfloor \frac{m}{n} i + 0.5 \right\rfloor \quad (7)$$

Note in this step, there is neither reconnection nor removal of vertices. All the work required is just to modify the position of existing border vertices.

Figure 4.13 Face Matching

4.2.4 New adaptive tessellation algorithm

The implementation of the new algorithm can be described below:

- 1: Generate refinement patterns in GPU memory
- 2: Construct PN triangles control net information
- 3: Generate edges' depths and face depth for each Bézier triangle
- 4: Check whether tolerance needs to be adjusted in each frame,
for example when LOD level changes.
- 5: For each triangle, send the control points/normals into GPU
- 6: Use the face depth level refinement pattern modified by edges'
depth to render the refined mesh.

In the following shader code segment, we show how our scheme performs the modification of refinement patterns according to the Bézier triangles' edge depth. The shader code is quite straightforward, which remaps the border vertices from face depth to edge depth, and the rest of the vertices will just be passed along.

```
Transform.fx

//shader effects file for Adaptive GPU PN triangles

//input

uniform extern float3 v[10];

uniform extern float3 n[6];

uniform extern int faceDepth;

uniform extern int edgeDepth[3];

uniform extern bool adaptive;

.

OutputColorVS PerVertexVS(int4 c : TEXCOORD0)

.

// modified coefficients

    if(c.x == 0) // u = 0
    {
        c.y = floor( (edgeDepth[0] * c.y)/faceDepth +0.5 );
        c.z = edgeDepth[0] - c.y;
        coe.x = ((float)c.x) / edgeDepth[0];
        coe.y = ((float)c.y) / edgeDepth[0];
        coe.z = ((float)c.z) / edgeDepth[0];
    }
    else if(c.y == 0) // v = 0
    {
        c.z = floor( (edgeDepth[1] * c.z )/faceDepth +0.5);
        c.x = edgeDepth[1] - c.z;
        coe.x = ((float)c.x) / edgeDepth[1];
        coe.y = ((float)c.y) / edgeDepth[1];
        coe.z = ((float)c.z) / edgeDepth[1];
    }
    else if(c.z == 0) // w = 0
```

```

{
    c.x = floor( (edgeDepth[2] * c.x )/faceDepth +0.5);
    c.y = edgeDepth[2] - c.x;
    coe.x = ((float)c.x) / edgeDepth[2];
    coe.y = ((float)c.y) / edgeDepth[2];
    coe.z = ((float)c.z) / edgeDepth[2];
}
else
{
    coe.x = ((float)c.x) / faceDepth;
    coe.y = ((float)c.y) / faceDepth;
    coe.z = ((float)c.z) / faceDepth;
}
.

```

The input values are ten control points and six control normals which are the same as those we used in the uniform GPU PN triangles implementation. The additional depth information will be used to modify the coefficients, i.e. the barycentric coordinates, of each vertex. Note that one vertex shader will only process one vertex each time. These generated coefficients will be used to calculate the new positions of vertices in the refinement pattern; the rest are implemented in the same way as what we have discussed in Chapter 4.1.

We use a coarse teapot model and a torus knot model to show how our scheme works. Figure 4.14 and Figure 4.15 are rendered using the original model. The tolerance is set to 10000. Figure 4.16 and Figure 4.17 are rendered with tolerance set to 0.98. Figure 4.18 and Figure 4.19 are rendered with tolerance set to 0.01. The max face depth is 80 when tolerance is set to 0.01. Figure 4.20 to Figure 4.25 are the test results of our adaptive

method applied on the torus know model. Currently our scheme does not provide extra work for creases and corners, so visual artifacts can be seen in Figure 4.25. We also tested our algorithm on a human head model and a soldier model. The results are shown from Figure 4.26 to Figure 4.33.

Frames Per Second = 191.98
Milliseconds Per Frame = 5.2089
Triangle Count = 474
Vertex Count = 1422
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 1
Debug Info: Error: 10000.000000
Lighting: per Vertex

Figure 4.14 Adaptive PN triangles original teapot model wireframe

Frames Per Second = 192.37
Milliseconds Per Frame = 5.1984
Triangle Count = 474
Vertex Count = 1422
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 1
Debug Info: Error: 10000.000000
Lighting: per Vertex

Figure 4.15 Adaptive PN triangles original teapot model solid rendering

Frames Per Second = 181.99
Milliseconds Per Frame = 5.4947
Triangle Count = 9780
Vertex Count = 8415
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 9
Debug Info: Error: 0.850707
Lighting: per Vertex

Figure 4.16 Adaptive PN triangles refined teapot model wireframe

Frames Per Second = 182.36
Milliseconds Per Frame = 5.4837
Triangle Count = 9780
Vertex Count = 8415
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 9
Debug Info: Error: 0.850707
Lighting: per Vertex

Figure 4.17 Adaptive PN triangles teapot refined model solid rendering

Frames Per Second = 26.05
Milliseconds Per Frame = 38.3852
Triangle Count = 681739
Vertex Count = 366509
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 80
Debug Info: Error: 0.010000
Lighting: per Vertex

Figure 4.18 Adaptive PN triangles: tolerance = 0.01, teapot wireframe

Frames Per Second = 38.95
Milliseconds Per Frame = 25.6728
Triangle Count = 681739
Vertex Count = 366509
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 80
Debug Info: Error: 0.010000
Lighting: per Vertex

Figure 4.19 Adaptive PN triangles: tolerance = 0.01, teapot solid rendering

Figure 4.20 Adaptive PN triangles: original torus knot wireframe

Figure 4.21 Adaptive PN triangles: original torus knot solid rendering

Frames Per Second = 347.53
Milliseconds Per Frame = 2.8775
Triangle Count = 5271
Vertex Count = 4503
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 9
Debug Info: Error: 2.631281
Lighting: per Vertex

Figure 4.22 Adaptive PN triangles refined torus knot model wireframe

Frames Per Second = 345.64
Milliseconds Per Frame = 2.8932
Triangle Count = 5271
Vertex Count = 4503
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 9
Debug Info: Error: 2.631281
Lighting: per Vertex

Figure 4.23 Adaptive PN triangles refined torus knot model wireframe

Frames Per Second = 28.37
Milliseconds Per Frame = 35.2483
Triangle Count = 1116256
Vertex Count = 581891
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 134
Debug Info: Error: 0.010000
Lighting: per Vertex

Figure 4.24 Adaptive PN triangles: tolerance = 0.01, torus knot wireframe

Frames Per Second = 42.90
Milliseconds Per Frame = 23.3091
Triangle Count = 1116256
Vertex Count = 581891
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 134
Debug Info: Error: 0.010000
Lighting: per Vertex

Figure 4.25 Adaptive PN triangles: tolerance = 0.01, torus knot wireframe

Frames Per Second = 107.76
Milliseconds Per Frame = 9.2800
Triangle Count = 1778
Vertex Count = 5334
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 1
Debug Info: Error: 10000.000000
Lighting: per Vertex

Figure 4.26 Adaptive PN triangles: original human head

Frames Per Second = 106.94
Milliseconds Per Frame = 9.3515
Triangle Count = 1778
Vertex Count = 5334
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 1
Debug Info: Error: 10000.000000
Lighting: per Vertex

Figure 4.27 Adaptive PN triangles: original human head wireframe

Frames Per Second = 99.96
Milliseconds Per Frame = 10.0035
Triangle Count = 42939
Vertex Count = 35192
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 16
Debug Info: Error: 0.257669
Lighting: per Vertex

Figure 4.28 Adaptive PN triangles: tolerance = 0.258, human head

Frames Per Second = 99.55
Milliseconds Per Frame = 10.0448
Triangle Count = 42939
Vertex Count = 35192
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 16
Debug Info: Error: 0.257669
Lighting: per Vertex

Figure 4.29 Adaptive PN triangles: tolerance = 0.258, human head wireframe

Frames Per Second = 109.95
Milliseconds Per Frame = 9.0951
Triangle Count = 1733
Vertex Count = 5199
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 1
Debug Info: Error: 10000.000000
Lighting: per Vertex

Figure 4.30 Adaptive PN triangles: original soldier model

Frames Per Second = 108.06
Milliseconds Per Frame = 9.2540
Triangle Count = 1733
Vertex Count = 5199
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 1
Debug Info: Error: 10000.000000
Lighting: per Vertex

Figure 4.31 Adaptive PN triangles: original soldier model wireframe

Frames Per Second = 105.37
Milliseconds Per Frame = 9.4900
Triangle Count = 23694
Vertex Count = 22478
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 10
Debug Info: Error: 0.882460
Lighting: per Vertex

Figure 4.32 Adaptive PN triangles: tolerance = 0.88, soldier model

Frames Per Second = 103.97
Milliseconds Per Frame = 9.6185
Triangle Count = 23694
Vertex Count = 22478
C:Enable Wireframe face hiding
Q: Mode
E: Wireframe
Z:Lighting
1234: Switch Model
MN: change level/error
,: fast change
Max Adaptive FDepth: 10
Debug Info: Error: 0.882460
Lighting: per Vertex

Figure 4.33 Adaptive PN triangles: tolerance = 0.88, soldier model wireframe

4.2.5 Performance

This section provides some experimental result to show the performance of the proposed adaptive tessellation scheme.

Adaptive scheme vs. uniform scheme

Our Adaptive GPU PN triangles scheme has been analyzed by comparing the rendering frame rate per second (FPS) of a teapot mesh, either by using Uniform GPU PN triangles or by using Adaptive GPU PN triangles. The configuration is the same to what we used in uniform GPU PN triangles implementation. The result of this test is shown in Table 4.2. The left column shows the maximum face depths which determine the flatness of mesh, and the rest two columns show the rendering frame rate per second and also the face counts and vertices count (how many vertices and triangles are processed by GPU) of these two approaches.

Model Face Depth	Uniform GPU PN FPS	Adaptive GPU PN FPS
Teapot Face Depth = 1	204.62	194.35
	F = 992, V = 2976	F = 992, V = 2976
Teapot Face Depth = 10	204.65	182.48
	F = 99200, V = 65472	F = 23582, V = 19767
Teapot Face Depth = 20	204.66	178.51
	F = 396800, V = 229152	F = 97802, V = 67059
Teapot Face Depth = 40	167.53	174.40
	F = 1587200, V = 854112	F = 361940, V = 209196
Teapot Face Depth = 60	80.72	173.70
	F = 3571200, V = 1875872	F = 833802, V = 459191
Teapot Face Depth = 80	46.87	173.18
	F = 6348800, V = 3294432	F = 1469165, V = 793068

Table 4.2 Adaptive GPU PN triangle performance

The line chart in Figure 4.20 shows the comparison of FPS values between adaptive GPU PN triangles and uniform GPU PN triangles. From this chart we can observe that adaptive GPU PN triangles scheme is significantly faster than the uniform one when the curvature is high. But the adaptive scheme is slower than the uniform one when maximum face depth is small. This is due to the additional data transferring from CPU to GPU and extra GPU calculation required in our scheme. So the improvement is not notable if the size of refinement patch is small.

Figure 4.21 is the comparison of needed GPU calculations between adaptive GPU PN triangles and uniform PN triangles. We can find the adaptive scheme will require fast less calculations than the uniform one.

Figure 4.34 Comparison of FPS between adaptive scheme and uniform scheme

Figure 4.35 Comparison of GPU calculations between adaptive scheme and uniform scheme

New adaptive scheme vs. other adaptive scheme

Regarding memory consumption, our scheme is very efficient. For any N level maximum face depth required by curvature, we only need N refinement patches while the approach in[48] requires N^3 refinement patches. For our adaptive scheme, we can easily use several thousand LOD. This kind of high LOD cannot be achieved by the other adaptive scheme[48]. For example, our scheme only needs 10 refinement patches for level 10 face depth but the other scheme[48] needs 1000 refinements patches.

Chapter 5

Conclusion

In this thesis, first we investigate how to apply Loop subdivision in real game development. A scheme to apply Loop subdivision over multiple meshes is presented to provide subdivision in game applications. The proposed approach overcomes the difficulties of using subdivision in game applications, and the implementation has been tested in commercial racing game engine.

Second, to achieve smoothness and speed in the same time, two methods are proposed. One is a uniform PN triangle based on GPU, and the other one is a novel adaptive PN triangles scheme based on GPU. The second scheme is based on curved PN triangles and GPU mesh refinement. The main improvement of our approach is that it greatly reduces graphics memory consumption, and all the calculations are processed in GPU. Along with this scheme, we introduce a depth tagging scheme and a method to construct our adaptive refinement patterns. By using the adaptive GPU PN triangles, the rendering speed of smooth surface is improved. Our adaptive scheme allows fast real-time mesh refinement on general GPU and supports very high LOD (say, 100). It requires relative low memory. For a 100 levels LOD, only 100 refinement patterns are required in our method, while in the previous method, one million refinement patterns are needed.

As for future work, we will apply the idea of adaptive PN triangles to Loop subdivision surfaces. So the adaptive scheme can generate smoother surfaces. We will also look into how to keep wanted sharp edges, for example, the cube model should remain as cube form after applying our tessellation method. Our current approach lacks of the ability to identify which sharp edge to keep, so surface details may be discarded wrongly. The future work should propose a suitable solution to handle creases and corners. We will also work on how to improve this scheme to make it more suitable for game designers, so game designers and game programmers can implement and use this scheme without too much effort and adaption (to change the models). These are the promising directions for future work.

References

1. Merriam-Webster's Collegiate Dictionary Tenth Edition. 2001, Merriam-Webster, Inc.
2. idSoftware. *Doom II*. 2009 <http://www.idsoftware.com/games/doom/doom2/>.
3. Kent, S.L., *The Ultimate History of Video Games: From Pong to Pokemon--The Story Behind the Craze That Touched Our Lives and Changed the World*. 2001.
4. Montfort, N. and I. Bogost, *Racing the Beam: The Atari Video Computer System*. 2009: The MIT Press. 184.
5. Zink, M., P. Starner, and B. Foote, *Programming HD DVD and Blu-ray Disc*. 2008: McGraw-Hill, Inc. 384.
6. MicrosoftGameStudios. *Gears of War*. 2009 <http://gearsofwar.xbox.com>.
7. NaughtyDogs. *Uncharted Drake's Fortune*. 2009 <http://www.us.playstation.com/uncharted/>.
8. Vlachos, A., et al., *Curved PN triangles*, in *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001, ACM.
9. Hoppe, H., et al., *Piecewise smooth surface reconstruction*, in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 1994, ACM.
10. Chaikin, G., *An algorithm for high speed curve generation*. Computer Graphics and Image Processing, 1974(3).
11. Forrest, A.R., *Notes on Chaikin's Algorithm*. Computational Geometry Project Memo CGP74/1, 1974.

12. Riesenfeld, R., *On Chaikin's algorithm*. Computer Graphics and Image Processing, 1975(4(3)): p. 304-310.
13. Dufilho, K., *Geri's game*, in *ACM SIGGRAPH 98 Electronic art and animation catalog*. 1998, ACM: Orlando, Florida, United States.
14. Zorin, D. and P. Schröder, *Subdivision for Modeling and Animation*, in *ACM SIGGRAPH 2000 courses*. 2000, ACM: New Orleans, Louisiana, USA. p. 194.
15. Doo, D. and M. Sabin, *Behaviour of recursive division surfaces near extraordinary points*, in *Seminal graphics: poineering efforts that shaped the field*. 1998, ACM. p. 177-181.
16. Catmull, E. and J. Clark, *Recursively generated B-spline surfaces on arbitrary topological meshes*, in *Seminal graphics: poineering efforts that shaped the field*. 1998, ACM. p. 183-188.
17. Pulli, K. and M. Segal, *Fast rendering of subdivision surfaces*, in *ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96*. 1996, ACM: New Orleans, Louisiana, United States.
18. Blythe, D., *The Direct3D 10 system*. ACM Trans. Graph., 2006. **25**(3): p. 724-734.
19. Jones, W., *Beginning DirectX 10 Game Programming*. 2007: Course Technology Press. 384.
20. Oneppo, M., *HLSL shader model 4.0*, in *ACM SIGGRAPH 2007 courses*. 2007, ACM: San Diego, California.
21. Loop, C., *Smooth subdivision surfaces based on triangles* in *Department of Mathematics*. 1987, University of Utah.
22. Biermann, H., A. Levin, and D. Zorin, *Piecewise smooth subdivision surfaces with normal control*, in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, ACM Press/Addison-Wesley Publishing Co.

23. Casteljau, P.d., *Courbes et surfaces a poles*, in *Technical report*, A. Citroen. 1963: Paris.
24. Casteljau, P.d., *Outillages méthodes calcul*, in *Technical report*, A. Citroen. 1959: Paris.
25. Farin, G., *Curves and surfaces for CAGD: a practical guide*. 2002: Morgan Kaufmann Publishers Inc. 499.
26. Semple, J.G. and G.T. Kneebone, *The Algebra of Geometry: Cartesian, Areal and Projective Coordinates*. 1998.
27. NVIDIA, C. *Graphics Processing Unit (GPU)* 2009 <http://www.nvidia.com/object/gpu.html>.
28. NVIDIA, C. *GeForce 256*. 1999 <http://www.nvidia.com/page/geforce256.html>.
29. Luebke, D., et al., *GPGPU: general purpose computation on graphics hardware*, in *ACM SIGGRAPH 2004 Course Notes*. 2004, ACM: Los Angeles, CA.
30. Kipfer, P., M. Segal, and R. Westermann, *UberFlow: a GPU-based particle engine*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2004, ACM: Grenoble, France.
31. Harada, T., S. Koshizuka, and Y. Kawaguchi, *Sliced data structure for particle-based simulations on GPUs*, in *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*. 2007, ACM: Perth, Australia.
32. Enright, D., S. Marschner, and R. Fedkiw, *Animation and rendering of complex water surfaces*. *ACM Trans. Graph.*, 2002. **21**(3): p. 736-744.
33. Adams, B., et al., *Adaptively sampled particle fluids*, in *ACM SIGGRAPH 2007 papers*. 2007, ACM: San Diego, California.

34. Müller, M., D. Charypar, and M. Gross, *Particle-based fluid simulation for interactive applications*, in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2003, Eurographics Association: San Diego, California.
35. Harris, M.J., et al., *Simulation of cloud dynamics on graphics hardware*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2003, Eurographics Association: San Diego, California.
36. Harris, M.J., et al., *Simulation of cloud dynamics on graphics hardware*, in *ACM SIGGRAPH 2005 Courses*. 2005, ACM: Los Angeles, California.
37. Zeller, C., *Cloth simulation on the GPU*, in *ACM SIGGRAPH 2005 Sketches*. 2005, ACM: Los Angeles, California.
38. Purcell, T.J., et al., *Photon mapping on programmable graphics hardware*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2003, Eurographics Association: San Diego, California.
39. Govindaraju, N.K., et al., *CULLIDE: interactive collision detection between complex models in large environments using graphics hardware*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2003, Eurographics Association: San Diego, California.
40. Govindaraju, N.K., et al., *Fast continuous collision detection among deformable models using graphics processors*. *Comput. Graph.*, 2007. **31**(1): p. 5-14.
41. Sloan, P.-P., et al., *Image-Based Proxy Accumulation for Real-Time Soft Global Illumination*, in *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*. 2007, IEEE Computer Society.
42. Ripollés, O. and M. Chover, *Technical Section: Optimizing the management of continuous level of detail models on GPU*. *Comput. Graph.*, 2008. **32**(3): p. 307-319.

43. Krüger, J. and R. Westermann, *Linear algebra operators for GPU implementation of numerical algorithms*, in *ACM SIGGRAPH 2005 Courses*. 2005, ACM: Los Angeles, California.
44. Wikipedia. *Shader*. 2009 <http://en.wikipedia.org/wiki/Shader>.
45. Wikipedia. *Level of details*. 2009 http://en.wikipedia.org/wiki/Level_of_detail.
46. Clark, J.H., *Hierarchical geometric models for visible-surface algorithms*. SIGGRAPH Comput. Graph., 1976. **10**(2): p. 267-267.
47. Boubekur, T. and C. Schlick, *Generic mesh refinement on GPU*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2005, ACM: Los Angeles, California.
48. Boubekur, T. and C. Schlick, *A Flexible Kernel for Adaptive Mesh Refinement on GPU*. 2008, HAL - CCSD.
49. Kim, M. and J. Peters, *Realtime loop subdivision on the GPU*, in *ACM SIGGRAPH 2005 Posters*. 2005, ACM: Los Angeles, California.
50. Müller, K. and S. Havemann, *Subdivision Surface Tessellation on the Fly using a versatile Mesh Data Structure* in *EUROGRAPHICS 2000*. 2000.
51. Shiue, L.-J., I. Jones, and J. Peters, *A realtime GPU subdivision kernel*. ACM Trans. Graph., 2005. **24**(3): p. 1010-1015.
52. Kazakov, M., *Catmull-Clark subdivision for geometry shaders*, in *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. 2007, ACM: Grahamstown, South Africa.
53. Loop, C. and S. Schaefer, *Approximating Catmull-Clark subdivision surfaces with bicubic patches*. ACM Trans. Graph., 2008. **27**(1): p. 1-11.

54. Peters, J., *Patching Catmull-Clark meshes*, in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, ACM Press/Addison-Wesley Publishing Co.
55. Kovacs, D., et al., *Real-time creased approximate subdivision surfaces*, in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, ACM: Boston, Massachusetts.
56. Bolz, J. and P. Schröder, *Rapid evaluation of Catmull-Clark subdivision surfaces*, in *Proceedings of the seventh international conference on 3D Web technology*. 2002, ACM: Tempe, Arizona, USA.
57. Bunnell, M., *Adaptive Tessellation of Subdivision Surfaces With Displacement Mapping*, in *GPU Gems 2*. 2005.
58. Ni, T., et al., *Efficient substitutes for subdivision surfaces*, in *ACM SIGGRAPH 2009 Courses*. 2009, ACM: New Orleans, Louisiana.
59. Sharp, B. *Subdivision Surface Theory*. 2000
http://www.gamasutra.com/view/feature/3177/subdivision_surface_theory.
60. Dyn, N., D. Levine, and J.A. Gregory, *A butterfly subdivision scheme for surface interpolation with tension control*. ACM Trans. Graph., 1990. 9(2): p. 160-169.
61. Shreiner, D., et al., *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition) (OpenGL)*. 2005: Addison-Wesley Professional.
62. 2XLGames. *Baja Edge of Control*. 2009 <http://www.bajagame.com/>.
63. Bruce, G.B., *Winged edge polyhedron representation*. 1972, Stanford University.
64. Shene, C.-K. *The Winged-Edge Data Structure*. [cited;
<http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/model/winged-e.html>].

65. Microsoft. *HLSL High Level Shading Language for DirectX*. 2009
[http://msdn.microsoft.com/en-us/library/bb509561\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx).
66. Hoppe, H., *Progressive meshes*, in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, ACM.
67. Zheng, J. and T.W. Sederberg, *Estimating tessellation parameter intervals for rational curves and surfaces*. *ACM Trans. Graph.*, 2000. **19**(1): p. 56-77.