國立交通大學

多媒體工程研究所

碩 士 論 文

基於視角的連續自我碰撞偵測以及在圖形處理器上之加速

View-based Continuous Self-Collision Detection with Graphics Hardware Acceleration

研 究 生：鄭游駿

指導教授：黃世強 教授

中 華 民 國 一 百 年 七 月

基於視角的連續自我碰撞偵測以及在圖形處理器上之加速

View-based Continuous Self-Collision Detection with Graphics Hardware Acceleration

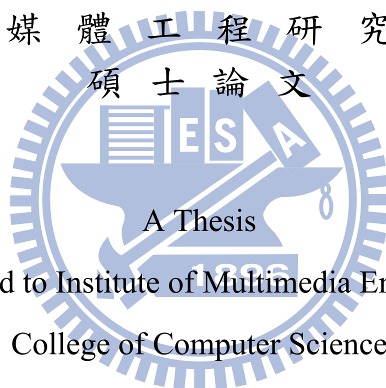
研　究　生：鄭游駿　　　　Student　：Yu-Chun Cheng

指 導 教 授：黃世強　　　　Advisor　：Sai-Keung Wong

國　立　交　通　大　學

多　媒　體　工　程　研　究　所

碩　士　論　文



A Thesis

Submitted to Institute of Multimedia Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2011

Hsinchu, Taiwan, Republic of China

中　華　民　國　一　百　年　七　月

基於視角的連續自我碰撞偵測以及在圖形處理器上之加速

研究生：鄭游駿　　指導教授：黃世強　教授

國 立 交 通 大 學

多 媒 體 工 程 研 究 所

摘要

在這篇論文中，我們提出了一個新的基於視角的方法來對可變形物體進行連續自我碰撞偵測，首先我們會利用一個點或是一條線段等簡單的幾何元件來當作基準，這些基準元件會被放置於可變形物體的內部，在整個物理模擬過程中，我們會確保它們不會穿過該可變形物體，接著，我們會利用這些基準元件來定義可變形物體中所有三角形的方向，根據每一個三角形的方向將整個可變形物體的三角形做分堆，放進不同的視角集合裡，這個過程稱為視角測試。如果所有的三角形都朝向基準元件，則該可變形物體中是沒有自我碰撞存在的，否則，我們將會對某些成對的視角集合做進一步地處理。視角測試的計算量遠小於傳統的方法，使用我們的方法來偵測自我碰撞，整體的效能是比較快的。此外，由於我們的方法適合於平行運算，所以我們進一步地利用圖形處理器來實作我們的方法，進而加速及改進整體的效能，實驗的數據顯示，利用我們的方法來偵測可變形物體的自我碰撞，其效能不管是在 CPU 上還是 GPU 上都是令人滿意的。

View-based Continuous Self-Collision Detection with Graphics Hardware Acceleration

Student: Yu-Chun Cheng Advisor: Sei-Keung Wong

National Chiao Tung University

Institute of Multimedia Engineering

Abstract

In this thesis, we propose a novel view-based approach for continuous self-collision detection with deformable triangle meshes. At first, we compute a simple geometric primitive, such as a point or a line segment. The primitive is put inside the deformable object, and we assume that it does not penetrate the object during the simulation. Then, the primitive is employed to determine the orientation of all triangles of the object. The triangles are divided into several view sets according to their orientation each frame, and this procedure is called *view test*. If all triangles face the primitives, then the object is self-collision free. Otherwise, self-collision is detected for certain pairs of the view sets. The computation of the view-based approach is lower than traditional methods, such as regular patches division and contour tests. Besides, our approach is suitable for parallel computing. We implement the view-based approach on GPUs with CUDA and improve the performance significantly. The experimental results show that the performance of the view-based approach for continuous self-collision detection is satisfied on both CPUs and GPUs.

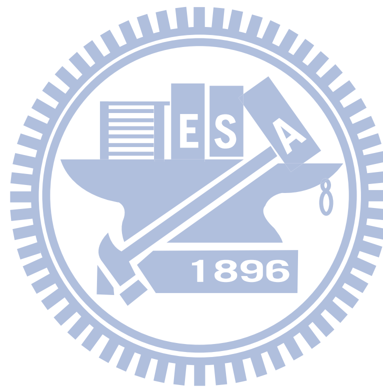
Acknowledgements

I would like to thank my advisor, Dr. Sai-Keung Wong, for his guidance, assistance, and inspirations. His suggestion is very helpful for me to obtain the idea and develop the framework of the thesis. In addition to support my work, he cared about our health and gave much useful advice of philosophy of life to us. I would also like to thank my thesis committee members, Dr. Ming-Te Chi and Dr. Wen-Chieh Lin, who evaluated my thesis.

I would like to thank all the labmates for their comments and help. They are so kind and cute so that I can learn in a cheerful environment. Last but not the least, I thank my parents for their unconditional support and patient. I hope it's my turn to take care of them.

Yu-Chun Cheng

July 2011



Contents

摘要 i

Abstract ii

Acknowledgements iii

Table of Contents iv

List of Figures vii

List of Tables x

1 Introduction 1

1.1 Motivation 4

1.2 Overview 4

1.3 Contribution 6

1.4 Organization 7

2 Related Work 8

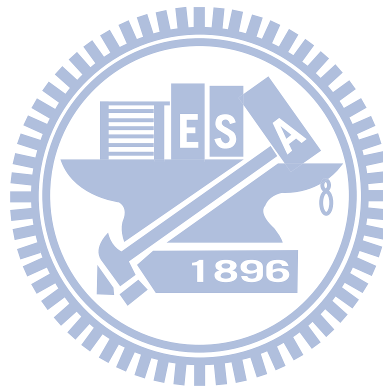
3 View-based Scheme 11

3.1 View-based Models 11

3.2 View Tests with a Point 13

3.3 View Tests with a Line Segment 17

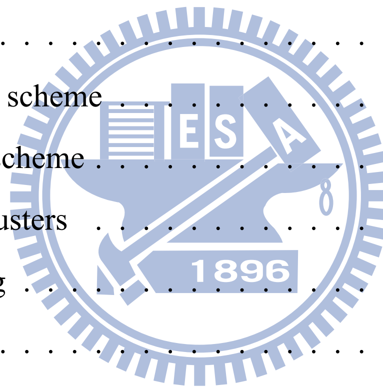
3.4 View-based Self-Collision Detection 25



3.5 View-Point Scheme . . . 29
3.6 View-Line Scheme . . . 32
3.7 Discussion . . . 34
3.7.1 View-based approach vs. traditional approach . . . 34
3.7.2 View-point vs. view-line . . . 35

4 Implementation on CPUs **37**
4.1 Preprocessing . . . 37
4.1.1 Acceleration structure construction . . . 37
4.1.2 View primitives computation . . . 39
4.1.3 Feature assignments . . . 40
4.1.4 Ghost triangles generation . . . 40
4.2 View Tests . . . 42
4.2.1 View-point scheme . . . 42
4.2.2 View-line scheme . . . 42
4.2.3 Triangle clusters . . . 43
4.3 Boundary Handling . . . 44
4.4 Traversal . . . 44
4.5 Elementary Tests . . . 47

5 Implementation on GPUs **49**
5.1 GPU Architecture . . . 49
5.2 Use of Data on GPUs . . . 52
5.2.1 Static data . . . 52
5.2.2 Dynamic data . . . 52
5.2.3 Global memory allocation in advance . . . 54
5.3 View Tests . . . 55
5.3.1 Vertex region determination . . . 56
5.3.2 Triangle type determination . . . 56
5.4 Boundary Handling . . . 57



5.5 Traversal 60

5.6 Elementary Tests 68

6 Results and Discussion **70**

6.1 Animation Benchmarks 70

6.2 Results on CPUs 77

6.3 Results on GPUs 82

6.4 Differences of Each Step on CPUs and on GPUs 84

6.5 Vertex Movement within a Frame 88

6.6 Improvement with Triangle Clusters 88

6.7 vBVHs Construction 88

6.8 Discussion 89

6.8.1 Closed and unclosed meshes 89

6.8.2 Ghost triangles with or without ghost vertices 90

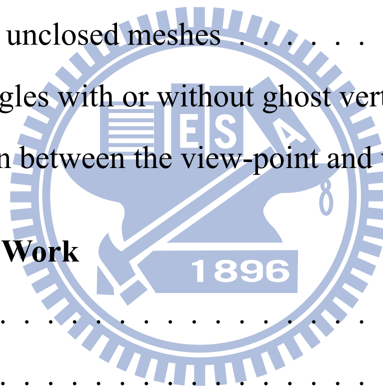
6.8.3 Comparison between the view-point and view-line scheme 91

7 Conclusion and Future Work **94**

7.1 Conclusion 94

7.2 Future Work 95

Bibliography **97**



List of Figures

1.1 Two types of collision detection. Let Δt be the time step. 3

3.1 The view-based point model. 12

3.2 The view-based line segment model. 12

3.3 Triangle orientation determination with a view-point at a certain time. . . 13

3.4 Triangle orientation determination with a view-point over a time interval. 16

3.5 Triangle orientation determination with a view-line at a certain time. . . . 17

3.6 Triangle orientation determination with a view-line at a certain time when the triangle lies in multiple regions. 20

3.7 Triangle orientation determination with a view-line at a certain time when the triangle lies in multiple regions. 20

3.8 Triangle orientation determination with a view-line over a time interval when the triangle moves across multiple regions. 22

3.9 Triangle orientation determination with a view-line over a time interval when the triangle moves across multiple regions. 25

3.10 A 2D closed curve with a ray shot outward. 26

3.11 2D closed curves with self-intersection. 27

3.12 A 2D unclosed curve with self-intersection. 29

3.13 An example of handling unclosed objects. 29

3.14 Negatively oriented edges according to the view-point and view-line schemes. 36

4.1 Ghost triangles with and without a ghost vertex. 41

4.2 Triangle clusters of a deformable object. 43

4.3 Marking process of vBVHs. 46

4.4 Skipped nodes in the vBVHs are adopted in order to compress the vBVHs. 47

5.1 Summary table of various GPU architecture. (The information is quoted from [NVI09].) 51

5.2 Level-order node list of the BVH. 53

5.3 Preprocess the triangle type list before performing boundary handling. . . 59

5.4 The triangle type list. 63

5.5 Preprocess the triangle type list before performing traversal. 64

5.6 Examples for history lists. 66

5.7 History list with many redundant nodes and reasonable nodes. 66

5.8 PCPs array preprocessing. 69

6.1 A series of snapshots of Ani. one. The first row and the second row are viewed from different viewpoints. 72

6.2 A series of snapshots of Ani. two. The first row and the second row are viewed from different viewpoints. 72

6.3 A series of snapshots of Ani. three. 73

6.4 A series of snapshots of Ani. four. 73

6.5 A series of snapshots of Ani. five. 74

6.6 A series of snapshots of Ani. six. 74

6.7 The snapshots of Ani. one and two in wireframe. 75

6.8 The snapshots of Ani. three with ghost triangles. 75

6.9 The snapshots of Ani. five with ghost triangles. 76

6.10 The snapshots of Ani. six with ghost triangles. 76

6.11 The numbers of triangles of all kinds of view sets with the view-point scheme for the six benchmarks. 80

6.12 The numbers of triangles of all kinds of view sets with the view-line scheme for the six benchmarks. 81

6.13 Comparisons of the number of negatively oriented triangles without considering violated triangles between the view-point scheme and the view-line scheme. 93

7.1 Boundary edges roll and lay down. 96

List of Tables

5.1 Salient features of device memory for devices of compute capability 1.x. The information is quoted from [NVI10a]. 50

6.1 Model complexities and information of ghost triangles for unclosed models. 71

6.2 Execution time (in ms) of each step with the view-point scheme on CPUs for continuous self-collision detection. 79

6.3 Execution time (in ms) of each step with the view-line scheme on CPUs for continuous self-collision detection. 79

6.4 Timing comparisons (in ms) between our view-based approach, AABB, 16-DOP, and ICCD for continuous self-collision detection. 79

6.5 Execution time (in ms) of performing traversal in the beginning to obtain the initial history nodes. 82

6.6 Execution time (in ms) of each step with the view-point scheme on GPUs for continuous self-collision detection. 82

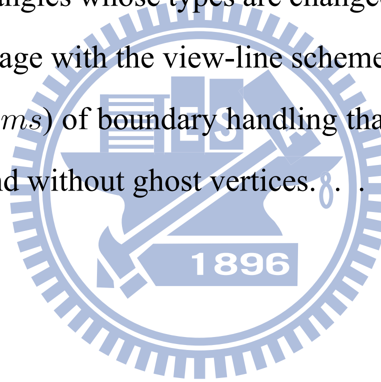
6.7 Execution time (in ms) of each step with the view-line scheme on GPUs for continuous self-collision detection. 83

6.8 Speed-up factors of each step with the view-point scheme using CPUs and GPUs. 83

6.9 Speed-up factors of each step with the view-line scheme using CPUs and GPUs. 83

6.10 Boundary handling with the view-line scheme by exact and inexact methods on CPUs. 86

6.11 Boundary handling with the view-line scheme by exact and inexact methods on GPUs. 86
6.12 Execution time (in ms) of performing traversal on GPUs with the view-line scheme for three different policies. 87
6.13 The number of vertices within a frame on average according to their movement. 88
6.14 The numbers of vertices and clusters of the deformable objects in the six benchmarks. 89
6.15 Execution time (in ms) of performing view tests in the view-line scheme with triangle clusters. 89
6.16 The cost of marking all of the nodes in the preprocessing stage. 89
6.17 The numbers of triangles whose types are changed between two consecutive frames on average with the view-line scheme. 90
6.18 Execution time (in ms) of boundary handling that the ghost triangles are constructed with and without ghost vertices. 91



Chapter 1

Introduction

Collision detection is an important and popular technique, and it is applied to several areas, such as computer graphics, virtual reality, physics simulation, cloth simulation, and computer animation. In general, objects are composed of triangle meshes in 3D space. We can employ the technique to detect collision for interactions between all the triangles of objects. So the movement of objects is reasonable and realistic. According to the types of objects, the complexity of collision detection can be quite different. In the object-space, the cost of performing collision detection is proportional to the number of triangles of objects. For rigid objects, it is easy to detect collision by using the bounding boxes such that swept volumes of the objects are covered. But for deformable objects, the computation is more complicated and expensive. Besides, deformable objects, such as cloth, may have a lot of self-collisions. We propose a novel view-based approach to perform self-collision detection with deformable objects in order to improve the performance.

At first, we can employ view primitives inside the object to determine the orientation of all triangles of deformable objects. And then the triangles are divided into several view sets according to their orientation. There are four view sets described as follows.

1. V^+: the triangles in V^+ face the view primitive.
2. V^-: the triangles in V^- are back to the view primitive.
3. V^0: the triangles in V^0 are coplanar to the view primitive at a certain time within a

frame.

4. V^v: the triangles in V^v are violated for unclosed meshes.

Note that we determine the triangles in V^+ to be positively oriented, and the triangles in V^- and V^0 to be negatively oriented. If all triangles face the view primitive, i.e. they are positively oriented, the deformable object is determined to be self-collision free. Otherwise, we can prove that there must be two triangles that one faces the view primitive and another is back to the view primitive, or these two triangle are both coplanar to the view primitive at the colliding position for a closed deformable object. So, we can detect self-collision for certain pairs of the view sets. On the other hand, we employ GPUs to improve the performance of self-collision detection because there are more processors on GPUs, and our approach is suitable to be performed in parallel.

Collision detection can be classified into two types, discrete collision detection and continuous collision detection. For discrete collision detection, we do not consider the movement trajectories of objects. The cost of computation is lower, and it is faster than continuous collision detection. But some collisions may be missed, and objects may pass through each other within a frame. Therefore, discrete collision detection is inaccurate. For continuous collision detection, the movement trajectories of objects are considered. We interpolate the movement trajectories of objects between two frames. If the time step of the simulation between two frames is smaller, we can obtain more accurate contact time of two colliding objects. But the cost of computation is higher.

We illustrate the two types of collision detection in Figure 1.1: (a) discrete collision detection, and (b) continuous collision detection. In Figure 1.1(a), the green ball is moving toward the red ball at time T_0. After a time step, we can obtain that the green ball and the red ball are overlapping at time T_1 by performing collision detection. Appropriate treatment will be performed for the two balls at time T_1. In fact, the collision has occurred within the time interval $[T_0, T_1]$. In Figure 1.1(b), the green ball is moving toward the red ball at time T_0. We compute the movement trajectory of the green ball by using its movement direction and velocity, and detect the collision between the two balls at time T', where $T' \in [T_0, T_1]$. In this case, the two balls may move oppositely after appropriate

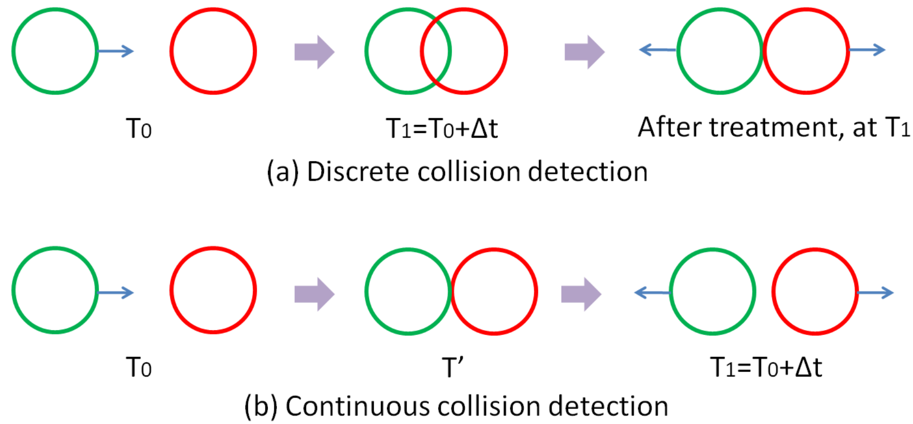


Figure 1.1: Two types of collision detection. Let Δt be the time step.

treatment at time T'.

Collision detection between different objects is called inter-object collision detection. However, collision detection for an object itself is called intra(self)-object collision detection. For rigid objects, suppose that they are independent, and there is no collision between themselves and the objects in the beginning. They may collide with or penetrate each other when moving. For deformable objects, such as cloth, collision occurs not only between the objects but also themselves. Therefore, self-collision detection should be handled for deformable objects. The computation of collision detection is quite expensive. A lot of techniques are developed to improve the performance of collision detection.

We can employ acceleration structures of objects to improve the performance of collision detection, such as bounding volume hierarchies (BVH). Each triangle is bounded by a bounding volume, and the acceleration structure is constructed by a top-down or a bottom-up manner. A leaf node of the BVH only contains a triangle. At the beginning of collision detection, we perform traversal for the BVHs of every two objects. The process of traversal is executed recursively by performing overlap tests of the bounding volumes until a pair of leaf nodes is reached. After that, we get a set of triangle pairs, called potentially colliding pairs. Finally, we perform elementary tests for the potentially colliding pairs to obtain the actual colliding triangle pairs.

1.1 Motivation

Nowadays, computer graphics and animation are widely applied in video games, movies, mobile products, and smart phones. Collision detection is a primary technique to simulate activity of objects. And the performance of collision detection should be not only accurate but also interactive. In this thesis, our focus is on continuous self-collision detection for deformable objects. The deformable objects are cloth, such as clothes worn on a character. For cloth, there may be numerous self-collisions. Hence, the computation is quite expensive, and it is a challenge to improve the performance of self-collision detection for cloth. On the other hand, the development of graphics processing units (GPUs) is rapid. GPUs are very suitable for computation of vectors in computer graphics. In addition, GPUs are employed for general purpose computation in recent years. Therefore, we employ GPUs to deal with a large number of computation of self-collision detection.

1.2 Overview

The main problem of self-collision is the high cost in checking the adjacent triangles. In fact, most adjacent triangles do not collide with each other. For the traditional method [VMT94], they compute normal cones and perform contour tests. A model is divided into several regular patches with hierarchical structures. The normal vectors of all triangles are bounded in each patch by computing a normal cone during the simulation phase. In the procedure, a lot of vectors and normalization are computed and the normal cones are bottom-up updated by merging the patches. Finally, contour tests are performed to check whether or not the model is self-collision free. Tang et al. [TCYM09] proposed a method to compute tightly bounded normal cones. However, the cost of normal cones computation is quite large due to computing additional vectors and normalization.

Hence, we want to eliminate the computation of normal cones and contour tests. We propose a novel view-based approach to perform continuous self-collision detection. Our method can be applied to deformable manifold triangle meshes, and we neither compute normal cones nor perform contour tests. Objects are determined whether or not they are

self-collision free by performing view tests. View tests are performed for all triangles by calculating dot products of the face normal vectors and the view-based vectors according to the view primitive and the triangles. The cost of performing view tests is much lower than computing the normal cones and performing contour tests. Our approach is suitable for column-liked models, such as dresses, pants, and shirts. In addition, the deformable objects should be closed. For unclosed triangle meshes, we can add some ghost triangles to enclose the boundaries. All edges are shared by two adjacent triangles for closed meshes. For unclosed meshes, there are some edges which only attach to one triangle. So, we can extract the boundary edges and generate the ghost triangles.

The view-based approach mainly has four steps, including view tests, boundary handling, traversal, and elementary tests. In the step of view tests, a deformable object is divided into several view sets based on their orientation related to a view primitive. The view primitive should be put inside the deformable object in the beginning, and we assume that it does not penetrate the deformable object during the simulation. All triangles of the deformable object are classified into three types include positive orientation, negative orientation, and violation. For closed meshes, the triangles can be positively oriented and negatively oriented. For unclosed meshes, the triangles can be positively oriented, negatively oriented, and violated. If all triangles of an object are positively oriented, then the object is self-collision free. Otherwise, further checks should be performed for certain pairs of the view sets.

After performing view tests, traversal is performed to collect potentially colliding triangle pairs based on negatively oriented triangles and violated triangles. If the number of negatively oriented and violated triangles is few, the cost of performing traversal for them is low. Finally, elementary tests are performed for the potentially colliding pairs.

In recent years, the techniques of GPUs are getting mature and popular. Parallel computing is employed massively in computer graphics. Actually, our view-based approach is suitable to be performed in parallel on GPUs. We use GPUs to accelerate our system with Compute Unified Device Architecture (CUDA). We evaluate our view-based approach with other methods. Our approach on CPUs outperforms other traditional meth-

ods, such as the approaches based on AABB [vdB99], K-DOP [KHM+98, MKE03], and ICCD [TCYM09]. Besides, our approach implemented on GPUs is 12 times faster than the one on CPUs on average.

1.3 Contribution

Our major contributions of the thesis are described as follows.

1. A novel view-based approach for continuous self-collision detection with lower computation is proposed.

2. Partial BVHs for all kinds of view sets are employed to perform traversal. We do not construct new BVHs for the view sets but mark the nodes of the original BVH to obtain the partial BVHs. The cost of marking nodes is low, and the cost of performing traversal for these partial BVHs is low.

3. The view-based approach is implemented on GPUs with CUDA. We implement two versions: CPUs and GPUs.

4. It is easy to implement the view-based approach on both CPUs and GPUs. The concept of the view-based approach is simple and straightforward. A deformable object is divided into several view sets by performing view tests with computation of dot products.

5. Self-collision detection with triangle-based traversal is performed on GPUs. The degree of parallelism for performing traversal is higher with the triangle-based method. In other words, traversal is performed based on each triangle on GPUs.

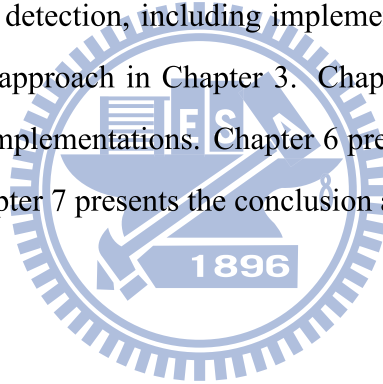
6. The view-based approach is suitable for different kinds of models. Deformable objects are most suitable for column-liked objects, such as dresses, pants, bags, and shirts. Besides, a piece of cloth, which can form a sphere-liked or a cube-liked region, is also suitable for the view-based approach.

7. If an object is determined to be self-collision free after performing view tests, traversal is not required to be performed and the computation is reduced significantly.

8. The performance of self-collision detection using the view-based approach is acceptable for an object with a lot of self-collision. We can extract a set of triangles from an object by performing view tests that self-collision occurs at these triangles. On CPUs, traversal is performed for the partial BVHs of the different view sets respectively. On GPUs, traversal is performed with the triangle-based method.

1.4 Organization

The remaining chapters of the thesis are organized as follows. Chapter 2 reports the related work about collision detection, including implementing on CPUs and on GPUs. We present our view-based approach in Chapter 3. Chapter 4 and 5 present the view-based algorithms and their implementations. Chapter 6 presents the experimental results and discussion. Finally, Chapter 7 presents the conclusion and future work.



Chapter 2

Related Work

Collision detection is an important technique in physics simulation. A comprehensive overview of collision detection for deformable objects can be found in the survey paper [TKH+05].

By a brute force method, we can perform self-collision detection for all triangle pairs of an object. If the number of triangles of an object is n, then we need to execute $\binom{n}{2}$ elementary tests of all triangle pairs. A lot of techniques can be employed to improve the performance of self-collision detection, including bounding volumes, spatial partitioning methods, and image-based methods. Bounding volume hierarchies are AABB trees [vdB99], OBB trees [GLM96], k-DOP [KHM+98], and Sphere [PG95]. By using BVH, it reduces the checking of triangle pairs that collision does not happen. The method employing BVH uses many bounding volumes with different sizes to bound an object. The entire object is enclosed with the largest bounding volume. Then, the object is divided into two similar parts and each part is bounded by another bounding volume. This is performed recursively until one or some triangles are enclosed with a bounding volume. BVH forms a structure of a tree. We can obtain a set of triangle pairs that every two triangles of a pair collide with each other by traversing the tree. Spatial partitioning includes octree [BT95] and BSP [Mel00]. The concepts of spatial partitioning and BVH are the same. The way of spatial partitioning is to partition the space into regular grids that bound the objects. The image-based methods rasterize meshes to framebuffer. The framebuffer includes stencil,

color, and depth buffer [BW02] [KP03] [VSC01].

Volino et al. [VMT94] proposed a breakthrough method for discrete self-collision detection. Originally, there is a lot of collision detection in checking adjacent triangle pairs. In general, most of adjacent triangles do not collide with each other. They divided a deformable object into several regular patches by computing normal cones, and performed contour tests to eliminate the collision check between adjacent triangle pairs. Provot [Pro97] extended the methods to handle continuous self-collision detection. After that, [WB05] [TCYM09] further improved the methods.

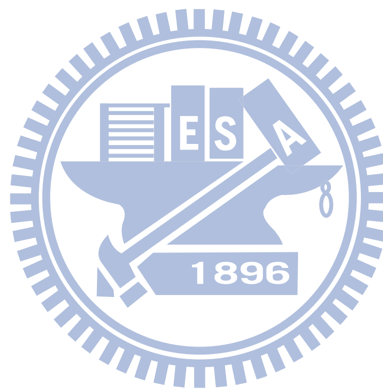
Recently, many parallel approaches were proposed for collision detection. Tang et al. [TMT09] proposed a parallel approach for collision detection with deformable models. The method performed incremental hierarchical computations and the hierarchies were built and updated each frame on multi-core CPUs. Kim et al. [KHH+09] combined the techniques of CPUs and GPUs to improve the performance of continuous collision detection. The main idea of the method is that traversal for BVHs is performed on CPUs, and elementary processing is performed on GPUs.

Due to the architecture of GPUs, they are more suitable than CPUs for performing parallel computation. Lauterbach et al. [LMM10] proposed an approach for rigid or deformable models. The method proposed by Lauterbach was to perform parallel computation for building, updating, and traversal of the hierarchies. Liu [LHLK10] implemented collision detection for massive moving rigid models. It changed the approach of Sweep and Prune (SaP) to the one of parallel SaP. And it reduced the number of false positives by using spatial subdivisions. Tang et al. [TMLT11] proposed a novel stream registration method to compute the triangle pairs that can have collision. And it reduced the overhead of the memory by using the deferred front tracking method.

Govindaraju et al. [GRLM03] used the technique of visibility-based culling to find out a potential collision set (PCS), and two-pass rendering algorithm to find out the part of collision. Due to the accuracy of image-based methods is affected by the resolution of images, Govindaraju et al. [GKJ+05] combined the method [GRLM03] with the technique of chromatic decomposition to improve the performance and the accuracy of collision

detection for deformable triangle meshes. Allard et al. [AFC+10] computed layered depth images on GPUs, and then used the information to handle the contacts of models.

In addition to physics simulation, collision detection is also used for motion planning. Pan et al. [PM11] implemented cluster and collision-packet traversal on GPUs to improve the performance of collision detection for motion planning.



Chapter 3

View-based Scheme

The main concept of the view-based approach is to divide all triangles of a deformable object into several view sets according to their orientation related to a view primitive. We classify the triangles into four view sets and three types, including positively oriented, negatively oriented, and violated. If all triangles of the object face the view primitive, i.e. they are positively oriented, then the object is self-collision free. Thus traversal does not be performed. Otherwise, self-collision detection is performed for certain pairs of the view sets.

We use a point and a line segment to be the view primitives. So, there are two view-based schemes. We introduce the view-point model and the view-line model at first, then explain how to perform view tests according to the different view primitives. After that, two kinds of view-based schemes are described in details. Finally, we discuss the differences between the view-based approach and the traditional approach, and between the view-point scheme and the view-line scheme.

3.1 View-based Models

We use two kinds of view primitives to determine the orientation of all triangles, including a point and a line segment. The view primitives are put inside the deformable object in the beginning. Besides, we assume that the view primitives do not penetrate the

deformable object during the simulation.

The view-based point model is demonstrated in Figure 3.1. Suppose that q is the view-based point, and it is put at the center of the model.

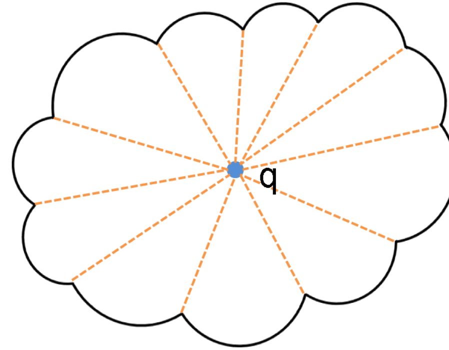


Figure 3.1: The view-based point model.

The view-based line segment model is demonstrated in Figure 3.2. Suppose that $\overline{q_0q_1}$ is the view-based line segment, and it is put inside the model. The space and the deformable object are divided into three regions, R_0, R_1, and R_2, based on the line segment $\overline{q_0q_1}$. Note that L_0 and L_1 are perpendicular to the line segment $\overline{q_0q_1}$. We can easily extend the model to 3D space that L_0 and L_1 are two parallel planes perpendicular to the line segment $\overline{q_0q_1}$.

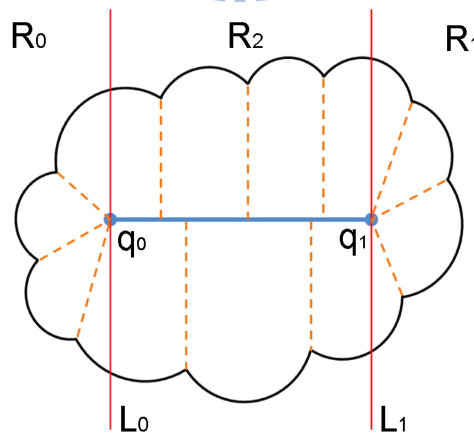


Figure 3.2: The view-based line segment model.

The view-based point is called *view-point*, and the view-based line segment is called *view-line*.

3.2 View Tests with a Point

We determine the orientation of all triangles of a deformable object according to the view primitive by performing view tests. In this section, we introduce performing view tests with a view-point.

At a certain time

At first, we perform the view test of a triangle at a certain time. Assume that there is a triangle $T(v_0, v_1, v_2)$, and $N = (v_1 - v_0) \times (v_2 - v_0)$ is the normal vector of the triangle. In addition, q is the view-point, as shown in Figure 3.3. We can choose any point of the triangle, for example vertex v_0 is chosen, and obtain a vector $v_0 - q$. Then, the view test is performed as follow.

$$VT_p = (v_0 - q) \cdot N \tag{3.1}$$

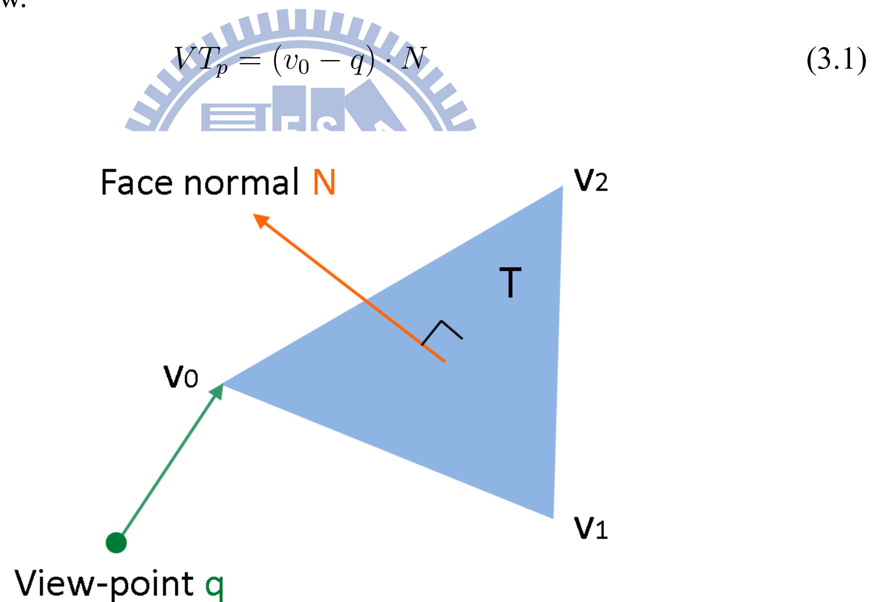


Figure 3.3: Triangle orientation determination with a view-point at a certain time.

Theorem 1. *Suppose that there is a triangle $T(v_0, v_1, v_2)$, and N is the normal vector of the triangle. q is a fixed view-point without intersecting with the triangle. If $(v_0 - q) \cdot N > 0$, then for any point v of the triangle T, $(v - q) \cdot N$ is always greater than 0.*

Proof. We prove it as follows.

1. If $\exists$ a point v' such that $(v' - q) \cdot N = 0$

   $\Rightarrow v' - q$ and N are perpendicular

   $\Rightarrow$ the triangle T and the point q are coplanar

   This is a contradiction to $(v_0 - q) \cdot N > 0$.

2. If $\exists$ a point v' such that $(v' - q) \cdot N < 0$

   Because the direction for normal vectors of all points of the triangle are the same, and $(v_0 - q) \cdot N > 0$, there must be a point v'' of the triangle such that $(v'' - q) \cdot N = 0$.

   $\Rightarrow v'' - q$ and N are perpendicular

   $\Rightarrow$ the triangle T and the point q are coplanar

   This is a contradiction.

Based on that, the theorem is proved. □

Corollary 1. *Suppose that there is a triangle $T(v_0, v_1, v_2)$, and N is the normal vector of the triangle. q is a fixed view-point without intersecting with the triangle. Then, for all points of the triangle T, the results of view tests are the same.*

By Corollary 1, we can classify the type of a triangle by performing view tests with any point of the triangle according to the view-point. Then, by Theorem 1 and Equation 3.1, we have three conclusions. For any point of a triangle T,

- If $VT_p > 0$, then the triangle faces the view-point.
- If $VT_p < 0$, then the triangle is back to the view-point.
- If $VT_p = 0$, then the triangle is coplanar to the view-point.

Over a time interval

We perform the view test of a triangle over a time interval $[0, \Delta t]$, where Δt is the time step of the simulation. During the simulation, all vertices move with linear velocities within a frame. Therefore, the position of a vertex v in $[0, \Delta t]$ should be $v^t = v_{bgn} + Vt$, where v_{bgn} is the initial position of v, v^t is the new position of v, V is the linear velocity of v, and $t \in [0, \Delta t]$.

Assume that there is a triangle $T(v_0, v_1, v_2)$, and $N = (v_1 - v_0) \times (v_2 - v_0)$ is the normal vector of the triangle in the beginning. After the time step Δt, the triangle T moves to $T'(v_0', v_1', v_2')$, and N' is the normal vector of the triangle T'. In addition, q is the viewpoint, as shown in Figure 3.4. We can choose any point of the triangle T, for example vertex v_0 is chosen, and perform the view test as follow.

$$VT_p(t) = v_0(t) \cdot N(t) \tag{3.2}$$

$VT_p(t)$ is a cubic function in the time domain, $N(t)$ is the time normal vector of the triangle T with quadratic form in the time domain, and $v_0(t)$ is a linear function in the time domain.

Hence, we want to compute $v_0(t)$ and $N(t)$. Suppose that the velocities of vertices v_0, v_1, and v_2 in $[0, \Delta t]$ are V_0, V_1, and V_2. As mentioned above, the position of v_0, v_1, and v_2 at a certain time in $[0, \Delta t]$ are

$$v_0^t = v_0 + V_0 \cdot t, t \in [0, \Delta t]$$

$$v_1^t = v_1 + V_1 \cdot t, t \in [0, \Delta t]$$

$$v_2^t = v_2 + V_2 \cdot t, t \in [0, \Delta t]$$

- $V_1(t)$ and $v_2(t)$ are similar to the result of $v_0(t)$.

$$\begin{aligned} v_0(t) &= (v_0 - q) + ((v_0^t - q) - (v_0 - q)) \\ &= (v_0 - q) + ((v_0 + V_0 \cdot t - q) - (v_0 - q)) \\ &= (v_0 - q) + ((v_0 - q) + V_0 \cdot t - (v_0 - q)) \\ &= (v_0 - q) + V_0 \cdot t \end{aligned}$$

- Let $B_s = v_1 - v_0$, $B_t = V_1 - V_0$, $C_s = v_2 - v_0$, $C_t = V_2 - V_0$.

$$\begin{aligned} N(t) &= (v_1^t - v_0^t) \times (v_2^t - v_0^t) \\ &= ((v_1 + V_1 \cdot t) - (v_0 + V_0 \cdot t)) \times ((v_2 + V_2 \cdot t) - (v_0 + V_0 \cdot t)) \\ &= ((v_1 - v_0) + (V_1 - V_0) \cdot t) \times ((v_2 - v_0) + (V_2 - V_0) \cdot t) \\ &= (B_s + B_t \cdot t) \times (C_s + C_t \cdot t) \\ &= (B_s \times C_t) \cdot t^2 + (B_t \times C_s + B_s \times C_t) \cdot t + B_s \times C_s \end{aligned}$$

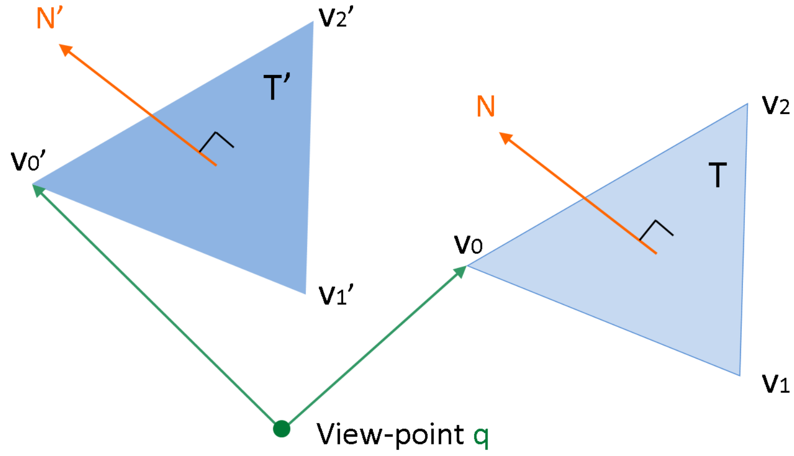


Figure 3.4: Triangle orientation determination with a view-point over a time interval.

Theorem 2. *Suppose that there is a triangle T, which moves from (v_0, v_1, v_2) to (v'_0, v'_1, v'_2) in $[0, \Delta t]$, and $N(t)$ is the time normal vector of the triangle T. q is a fixed view-point without intersecting with and passing through the triangle T in $[0, \Delta t]$. If $v_0(t) \cdot N(t) > 0$, then for any point v of the triangle, $v(t) \cdot N(t)$ is always greater than 0 in $[0, \Delta t]$.*

Proof. We prove it as follows.
If the triangle T does not pass through the view-point q, and $v_0(t) \cdot N(t) > 0$, then the triangle T always faces the view-point q in $[0, \Delta t]$. By Theorem 1, all the vertices of the triangle face the view-point q. Therefore, for any point v of the triangle, $v(t) \cdot N(t)$ is always greater than 0 in $[0, \Delta t]$. □

Corollary 2. *Suppose that there is a triangle T, which moves from (v_0, v_1, v_2) to (v'_0, v'_1, v'_2) in $[0, \Delta t]$, and $N(t)$ is the time normal vector of the triangle T. q is a fixed view-point without intersecting with and passing through the triangle T in $[0, \Delta t]$. Then, for all points of the triangle T, the results of view tests are the same in $[0, \Delta t]$.*

By Corollary 2, we can classify the type of a triangle by performing view tests with any point of the triangle over a time interval according to the view-point. Then, by Theorem 2 and Equation 3.2, we have three conclusions. For any point of a triangle T in $[0, \Delta t]$,

- If $VT_p(t) > 0$, then the triangle faces the view-point, and it is assigned to the view

set V_p^+.

- If $VT_p(t) < 0$, then the triangle is back to the view-point, and it is assigned to the view set V_p^-.
- If $VT_p(t)$ contains both positive and negative values, then the triangle is coplanar to the view-point at a certain time in $[0, \Delta t]$, and it is assigned to the view set V_p^0.

3.3 View Tests with a Line Segment

In this section, we introduce performing view tests with a view-line.

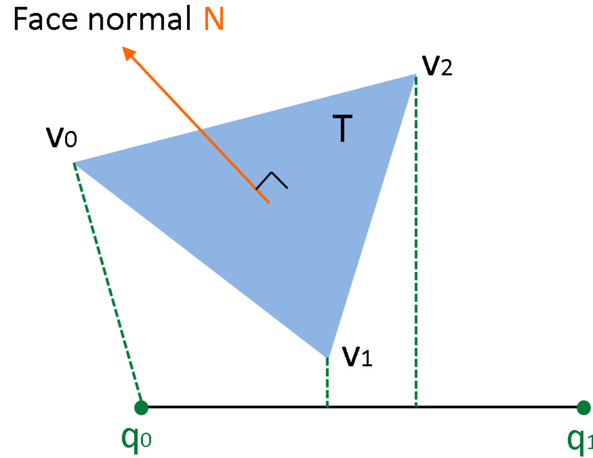


Figure 3.5: Triangle orientation determination with a view-line at a certain time.

At a certain time

At first, we perform the view test of a triangle at a certain time. Assume that there is a triangle $T(v_0, v_1, v_2)$, and $N = (v_1 - v_0) \times (v_2 - v_0)$ is the normal vector of the triangle. In addition, $\overline{q_0 q_1}$ is the view-line, as shown in Figure 3.5. We can choose a vertex v of the triangle T and obtain a vector $v - v_p$, where v_p is the check point on the view-line $\overline{q_0 q_1}$. Then, the view test is performed as follow.

$$VT_l = (v - v_p) \cdot N \tag{3.3}$$

v_p has three types according to the location of v.

1. If $v \in R_0$, then $v_p = q_0$.

2. If $v \in R_1$, then $v_p = q_1$.

3. If $v \in R_2$, then $v_p =$ the projective point of v on the view-line $\overline{q_0 q_1}$.

Therefore, view tests based on the view-point model and the view-line model are different. For the view-line model, we divide space and deformable objects into three regions, and the check points are different with respect to different vertices. Hence, before performing view tests of a triangle, we determine the regions for three vertices of a triangle.

For a vertex v of the triangle T, the projective point of v on the view-line $\overline{q_0 q_1}$ should be $v_p = q_0 + u \times (q_1 - q_0)$.

$$\begin{aligned}
& \overrightarrow{q_0 q_1} \cdot (v_p - v) \\
= \quad & (q_1 - q_0) \cdot (v_p - v) \\
= \quad & (q_1 - q_0) \cdot ((q_0 + u \times (q_1 - q_0)) - v) \\
= \quad & (q_1 - q_0) \cdot (q_0 - v) + u \times ((q_1 - q_0) \cdot (q_1 - q_0)) = 0 \\
\Rightarrow \quad & u = \frac{(v - q_0) \cdot (q_1 - q_0)}{(q_1 - q_0) \cdot (q_1 - q_0)}
\end{aligned}$$

Then, we can determine the regions for the vertex v based on the value of u.

- If $u < 0$, then $v \in R_0$.

- If $u > 1$, then $v \in R_1$.

- If $u \in [0, 1]$, then $v \in R_2$.

Theorem 3. *Suppose that there is a triangle $T(v_0, v_1, v_2)$, and N is the normal vector of the triangle. $\overline{q_0 q_1}$ is a fixed view-line without intersecting with the triangle. Then, for all vertices of the triangle T, the results of view tests are the same.*

Proof. We prove it as follows.

- Case 1: For any two vertices, if they are in the same region, R_0 or R_1, then it is proved by Corollary 1.

- Case 2: For any two vertices, if they are in the same region, R_2, then suppose that v_0 and v_1 are chosen, and v_{0p} and v_{1p} are the projective points of v_0 and v_1 on the view-line $\overline{q_0 q_1}$.

$$A = (v_0 - v_{0p}) \cdot N$$

$$B = (v_1 - v_{1p}) \cdot N$$

$$C = (v_0 - v_{1p}) \cdot N$$

$$D = (v_1 - v_{0p}) \cdot N$$

  By Corollary 1, we have the following results.
  The results of A and D are the same according to the fixed point v_{0p}.
  The results of B and C are the same according to the fixed point v_{1p}.
  The results of A and C are the same according to the fixed point v_0.
  The results of B and D are the same according to the fixed point v_1.
  Therefore, the results of $VT_{v_0} = (v_0 - v_{0p}) \cdot N$ and $VT_{v_1} = (v_1 - v_{1p}) \cdot N$ are the same.

- Case 3: For any two vertices, if one vertex belongs to R_0 and the other belongs to R_2, then suppose that v_0 and v_1 are chosen that $v_0 \in R_0$ and $v_1 \in R_2$, as shown in Figure 3.6. We can find a point m of the triangle that the projective point of m on the view-line $\overline{q_0 q_1}$ is q_0. The results of VT_{v_0} and VT_m are the same by case1. The results of VT_m and VT_{v_1} are the same by case2. So, the results of VT_{v_0} and VT_{v_1} are the same.

- case4: For any two vertices, if one vertex belongs to R_1, and the other belongs to R_2, then the result is similar to case3.

- case5: For any two vertices, if one vertex belongs to R_0 and the other belongs to R_1, then suppose that v_0 and v_1 are chosen that $v_0 \in R_0$ and $v_1 \in R_1$, as shown in Figure 3.7. Similar to case3, we can find two points m_0 and m_1 of the triangle that

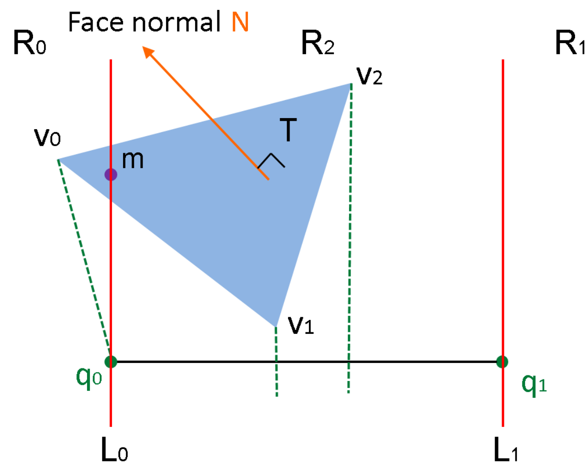
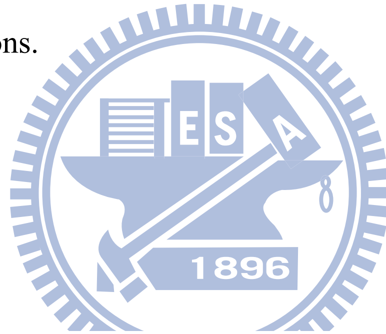


Figure 3.6: Triangle orientation determination with a view-line at a certain time when the triangle lies in multiple regions.



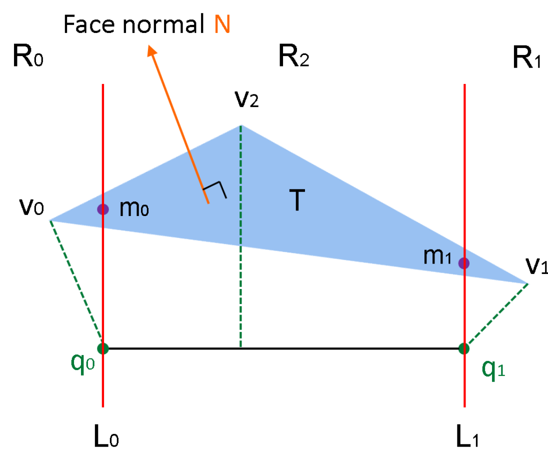


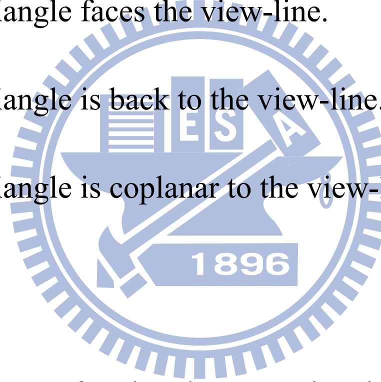
Figure 3.7: Triangle orientation determination with a view-line at a certain time when the triangle lies in multiple regions.

the projective points of m_0 and m_1 on the view-line $\overline{q_0q_1}$ are q_0 and q_1. The results of VT_{v_0} and VT_{m_0} are the same, and the results of VT_{m_1} and VT_{v_1} are the same by case1. The results of VT_{m_0} and VT_{m_1} are the same by case2. So, the results of VT_{v_0} and VT_{v_1} are the same.

According to the above results, it is proved. Hence, for all vertices of the triangle T, the results of view tests are the same. □

Therefore, we can classify the type of a triangle by performing view tests with any point of the triangle according to the view-line. In addition, the chosen vertex belongs to only one region at a certain time, so view tests are performed for just one time. By Theorem 3 and Equation 3.3, we have three conclusions. For any point of a triangle T,

- If $VT_l > 0$, then the triangle faces the view-line.
- If $VT_l < 0$, then the triangle is back to the view-line.
- If $VT_l = 0$, then the triangle is coplanar to the view-line.



Over a time interval

Next, we perform view tests of a triangle over a time interval $[0, \Delta t]$, where Δt is the time step of the simulation. During the simulation, all vertices move with linear velocities within a frame. Therefore, the position of a vertex v at a certain time in $[0, \Delta t]$ should be $v^t = v_{bgn} + Vt$, where v_{bgn} is the initial position of v, v^t is the new position of v, V is the linear velocity of v, and $t \in [0, \Delta t]$.

Assume that there is a triangle $T(v_0, v_1, v_2)$ and $N = (v_1 - v_0) \times (v_2 - v_0)$ is the normal vector of the triangle in the beginning. After the time step Δt, the triangle moves to $T'(v'_0, v'_1, v'_2)$, and N' is the normal vector of the triangle T'. In addition, $\overline{q_0q_1}$ is the view-line, as shown in Figure 3.8.

Before performing view tests of a triangle, we determine the regions that the triangle belongs to because the triangle can spread and move across multiple regions in $[0, \Delta t]$. The regions for a triangle are determined by its three vertices. We compute the regions

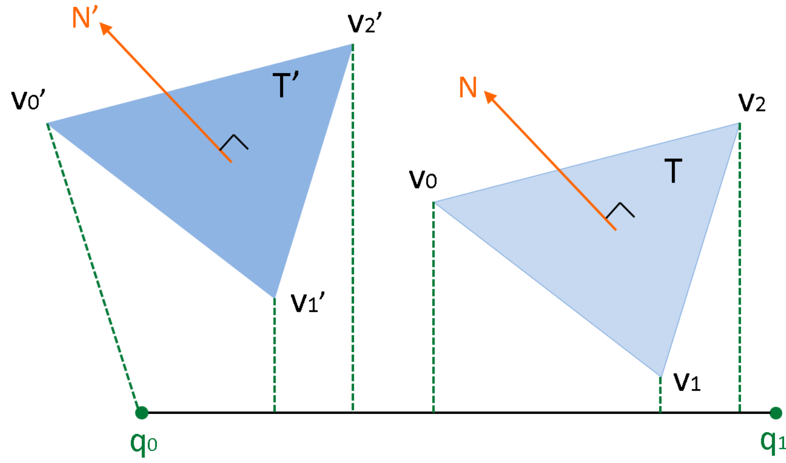


Figure 3.8: Triangle orientation determination with a view-line over a time interval when the triangle moves across multiple regions.

for three vertices, then the regions for the triangle are the union of the regions for its three vertices.

Suppose that there is a vertex of the triangle, which moves from v to v' in $[0, \Delta t]$. Then, the projective points of v and v' on the view-line $\overline{q_0 q_1}$ should be $v_p = q_0 + u_{bgn} \times (q_1 - q_0)$ and $v'_p = q_0 + u_{end} \times (q_1 - q_0)$. Because the vertices move linearly, $[u_{bgn}, u_{end}]$ is also linear. We can determine the regions that the vertices move across in $[0, \Delta t]$ according to the values of u_{bgn} and u_{end}.

Let R_{v_0}, R_{v_1}, and R_{v_2} are the regions that the three vertices v_0, v_1, and v_2 move across in $[0, \Delta t]$. So, the regions for the triangle are $R_{v_0} \cup R_{v_1} \cup R_{v_2}$. Note that the check points are different based on different vertices of the triangle. Actually, view tests are performed based on the vertices. Suppose that v is one of the vertices of the triangle T. Then, the view test is performed as follow.

$$VT_l(t) = v(t) \cdot N(t) \tag{3.4}$$

$VT_l(t)$ is a cubic function in the time domain, $N(t)$ is the time normal vector of triangle T with quadratic form in the time domain, and $v(t)$ is a linear function in the time domain for the vector variation from the vertex to the check points in $[0, \Delta t]$. Note that we compute $v(t)$ separately according to the region for the vertex. For example, if a vertex v

moves from R_0 to R_2 in $[0, \Delta t]$, then the check point is variable. So, the function of vector variation from the vertex to the check point is not linear. We should compute n kinds of $v(t)$, where n is the number of regions for the vertex in $[0, \Delta t]$.

Suppose that the velocities of vertices v_0, v_1, and v_2 in $[0, \Delta t]$ are V_0, V_1, and V_2. As mentioned above, the positions of v_0, v_1, and v_2 at a certain time in $[0, \Delta t]$ are

$$v_0^t = v_0 + V_0 \cdot t, t \in [0, \Delta t]$$

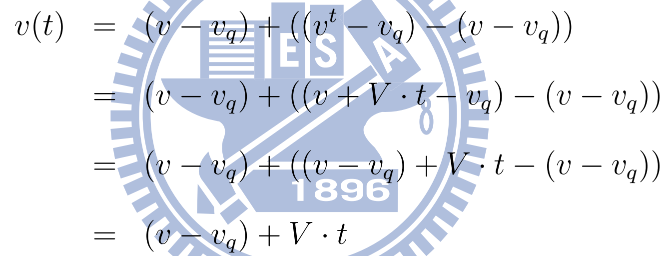
$$v_1^t = v_1 + V_1 \cdot t, t \in [0, \Delta t]$$

$$v_2^t = v_2 + V_2 \cdot t, t \in [0, \Delta t]$$

Then, $v(t)$ and $N(t)$ are computed as follows.

- For any vertex v, v moves across R_0 or R_1 in $[0, \Delta t]$.

$$\begin{aligned} v(t) &= (v - v_q) + ((v^t - v_q) - (v - v_q)) \\ &= (v - v_q) + ((v + V \cdot t - v_q) - (v - v_q)) \\ &= (v - v_q) + ((v - v_q) + V \cdot t - (v - v_q)) \\ &= (v - v_q) + V \cdot t \end{aligned}$$



v_q is q_0 or q_1 according to the location of v.

- For any vertex v, v moves across R_2 in $[0, \Delta t]$.

$$\begin{aligned} v(t) &= (v - v_{q0}) + ((v^t - v_{qt}) - (v - v_{q0})) \\ &= (v - v_{q0}) + ((v + V \cdot t - v_{qt}) - (v - v_{q0})) \\ &= (v - v_{q0}) + (v - v_{qt}) + V \cdot t - (v - v_{q0}) \\ &= (v - v_{qt}) + V \cdot t \end{aligned}$$

v_{q0} is the projective point of v in the beginning, and v_{qt} is the projective point of v at a certain time in $[0, \Delta t]$.

- Let $B_s = v_1 - v_0$, $B_t = V_1 - V_0$, $C_s = v_2 - v_0$, $C_t = V_2 - V_0$

$$
\begin{aligned}
N(t) &= (v_1^t - v_0^t) \times (v_2^t - v_0^t) \\
&= ((v_1 + V_1 \cdot t) - (v_0 + V_0 \cdot t)) \times ((v_2 + V_2 \cdot t) - (v_0 + V_0 \cdot t)) \\
&= ((v_1 - v_0) + (V_1 - V_0) \cdot t) \times ((v_2 - v_0) + (V_2 - V_0) \cdot t) \\
&= (B_s + B_t \cdot t) \times (C_s + C_t \cdot t) \\
&= (B_s \times C_t)t^2 + (B_t \times C_s + B_s \times C_t)t + B_s \times C_s
\end{aligned}
$$

Theorem 4. *Suppose that there is a triangle T, which moves from (v_0, v_1, v_2) to (v_0', v_1', v_2'), and $N(t)$ is the time normal vector of the triangle T. $\overline{q_0 q_1}$ is a fixed view-line without intersecting with and penetrating the triangle in $[0, \Delta t]$. Then, for all vertices of the triangle T, the results of view tests are the same in $[0, \Delta t]$.*

Proof. We prove it as follows.

- Case 1: For any two vertices, if they move in the same region in $[0, \Delta t]$, then the results of view tests are always the same by Theorem 3.

- Case 2: For any two vertices, if they move across multiple regions in $[0, \Delta t]$, the proof is described as follows. For example, in Figure 3.9, v_0 moves from R_2 to R_0, and v_1 and v_2 move inside R_2 in $[0, \Delta t]$. Suppose that the triangle T moves to T_t at time t, where $t \in [0, \Delta t]$. In $[0, t]$, all vertices move in the same region and the results of view tests are the same by Theorem 3. On the other hand, in $(t, \Delta t]$, We can always find a point m_t that the projective point of m_t on the view-line $\overline{q_0 q_1}$ is q_0. So, the results of view tests of v_0 are equal to the point m_t in $(t, \Delta t]$. And the results of view tests of v_1 and v_2 are equal to the point m_t in $(t, \Delta t]$. Therefore, the results of view tests of v_0, v_1, and v_2 are the same.

By Case 1 and Case 2, it is proved. Hence, for all vertices of the triangle T, the results of view tests are the same in $[0, \Delta t]$. □

Therefore, we can classify the type of a triangle by performing view tests with any point of the triangle over a time interval according to the view-line. View tests are performed for nr times, where nr is the number of regions that the chosen vertex moves

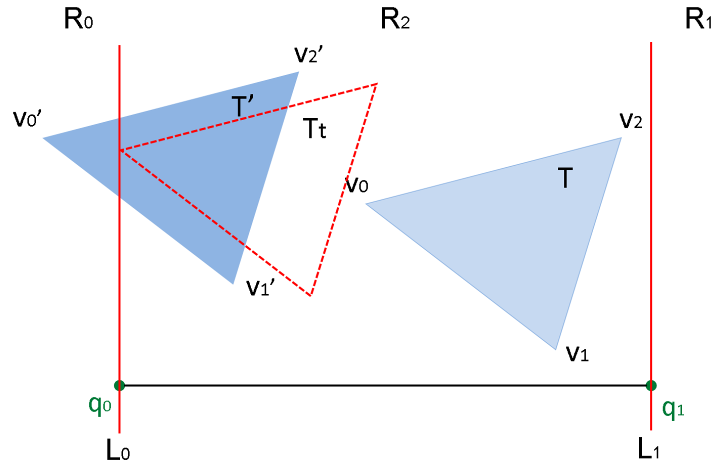


Figure 3.9: Triangle orientation determination with a view-line over a time interval when the triangle moves across multiple regions.

across in $[0, \Delta t]$. By Theorem 4 and Equation 3.4, we have three conclusions. Suppose that the result of view tests for a triangle T is $VT_l(t) = \cup VT_{li}(t)$, where $VT_{li}(t)$ are the results of view tests according to region R_i, $i = 0, ..., n-1$, and $1 \leq n \leq 3$. For any point of a triangle T in $[0, \Delta t]$,

- If $\forall VT_{li}(t) > 0$, then $VT_l(t) > 0$. So, the triangle faces the view-line, and it is assigned to the view set V_l^+.
- If $\forall VT_{li}(t) < 0$, then $VT_l(t) < 0$. So, the triangle is back to the view-line, and it is assigned to the view set V_l^-.
- If $VT_{li}(t)$ contain both positive and negative values, then the triangle is coplanar to the view-line at a certain time in $[0, \Delta t]$, and it is assigned to the view set V_l^0.

3.4 View-based Self-Collision Detection

If a deformable object has self-collision, then there are some properties for the triangles at the colliding position. Let's take a look on a 2D closed curve, for example. Suppose that there is a 2D closed curve, and there is no self-intersection. In fact, the curve is called a Jordan curve, as shown in Figure 3.10. By the Jordan curve theorem, the curve divides

the 2D space into two regions, an interior region and an exterior region. For example, in Figure 3.10, the interior region is indicated by green color, and the exterior region is indicated by blue color.

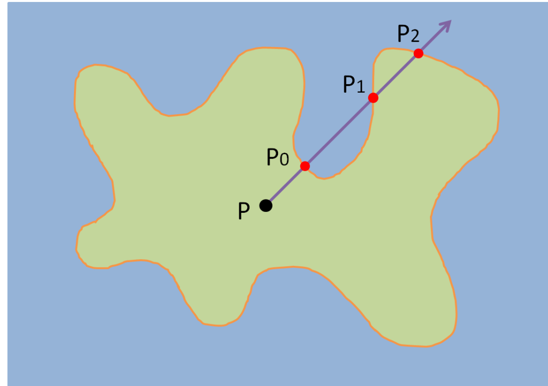


Figure 3.10: A 2D closed curve with a ray shot outward.

Now, we shoot a ray from a fixed point P in the interior region to any point in the exterior region, as shown in Figure 3.10. We can observe that the ray must intersect with the curve for odd number of times. Initially, the ray lies in the interior region. After the first time the ray intersects with the curve, the ray lies in the exterior region. However, when the ray intersects with the curve again, the ray must lie in the interior region. In short, after the ray intersecting with the curve, the ray lies in the different region.

Suppose that the normal vectors of the curve point to the exterior region. And there is a ray R with direction d shot from the point P to the exterior region and intersected with the curve at P_0, P_1, and P_2 orderly, as shown in Figure 3.10. P_0 is the first intersection point. In this case, $d \cdot N_{P_0} \geq 0$, where N_{P_0} is the normal vector of the point P_0. So, we can determine that the point P_0 faces the point P. Next, P_1 is the second intersection point. In this case, $d \cdot N_{P_1} \leq 0$, where N_{P_1} is the normal vector of the point P_1. So we can determine that the point P_1 is back to the point P. Finally, P_2 is the third intersection point. In this case, $d \cdot N_{P_2} \geq 0$ again, where N_{P_2} is the normal vector of the point P_2. So, we can determine that the point P_2 faces the point P.

For a 2D closed curve with no self-intersection, we can now conclude that if a ray shoots from a point P inside the curve and intersects with some points on the curve. When

an intersection occurs between the ray and the curve, the regions that the ray lies and the orientation of the intersection points according to P are opposite.

Now, for a 2D closed curve with self-intersection, suppose that the colliding point is called p_c. A ray is shot from the point P in the interior region to p_c, as shown in Figure 3.11. At the colliding position, the ray should intersect with the curve for greater than or equal to two times. Therefore, there are at least two points that one faces P and another is back to P at the colliding position. In addition, there is a special case. At the colliding position, the normal vectors of the points are perpendicular to the direction of the ray, i.e. $d \cdot N_i = 0$, where i is the number of points at the colliding position.

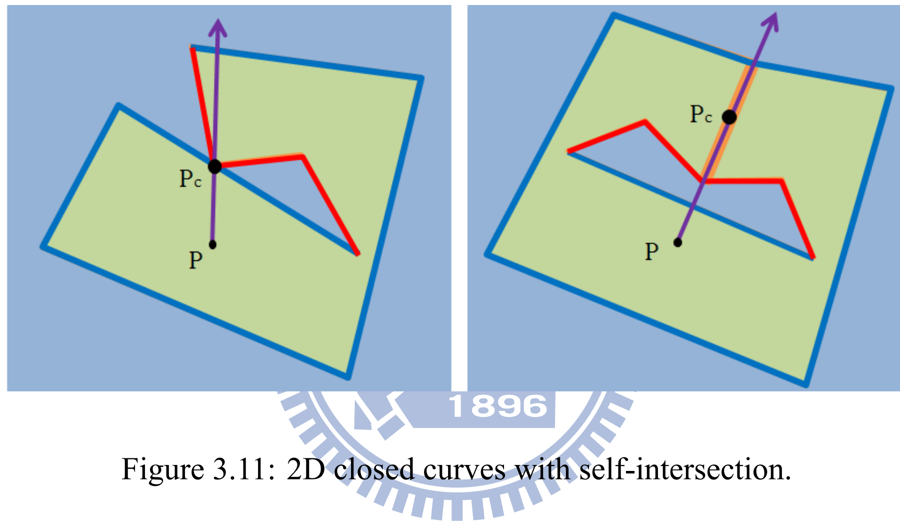


Figure 3.11: 2D closed curves with self-intersection.

Theorem 5. *Suppose that there is a 2D closed curve with self-intersection. A point P is put in the interior region of the curve. Then, at the colliding position, there are two points p_x and p_y that $(d \cdot N_x) \cdot (d \cdot N_y) < 0$ or both N_x and N_y are perpendicular to d, where d is the direction of a ray shot from the point P to the exterior region of the curve, and N_x and N_y are the normal vectors of p_x and p_y.*

Proof. If we shoot a ray from the point P to the colliding position, then there are two points at least at the colliding position. By the Jordan curve theorem, the location, which the ray passes through, should be change when the ray intersects with the curve, and the orientation should be opposite for any two consecutive intersection points. Now, we have two points, suppose they are p_x and p_y, at least at the colliding position, so, the orientation

of p_x and p_y are opposite according to P, and $(d \cdot N_x) \cdot (d \cdot N_y) < 0$. Besides, for the special case, Both N_x and N_y can be perpendicular to d. $\square$

We can extend the results to a 2D closed curve with a line segment in the interior region. We divide the space and the curve into three regions. Originally, we create a ray shot from the point P in the interior region to the exterior region. Now, we create a ray shot from the check point on the line segment to the curve. The check point is determined by the region that the curve belongs to. We can further extend the results to 3D closed triangle meshes with a point or a line segment inside the meshes.

Corollary 3. *Suppose that there is a 3D closed model with triangle meshes, and a point q lies inside the meshes. If self-collision occurs, then there are at least two triangles at the colliding position that one faces the point q, and another is back to the point q, or the normal vectors of these two triangles are both perpendicular to the vector from q to themselves.*

Corollary 4. *Suppose that there is a 3D closed model with triangle meshes, and a line segment $\overline{q_0q_1}$ lies inside the meshes. If self-collision occurs, then there are at least two triangles at the colliding position that one faces the line segment $\overline{q_0q_1}$, and another is back to the line segment $\overline{q_0q_1}$, or the normal vectors of these two triangles are both perpendicular to the vector from the check points to themselves.*

Handling unclosed meshes

The view-based approach is suitable for closed meshes. For unclosed meshes, there are some modification of the view-based approach. For example, for a 2D unclosed curve, the boundaries of the curve may roll, as shown in Figure 3.12. We can observe that the edges a and b both face the view-point q, and they collide with each other at point p_c. This is a contradiction to Theorem 5. So, we collect the edges which are determined to be violated. Similarly, for a 3D model, we collect the triangles which do not conform to Corollary 3 and 4. These triangles are called violated triangles.

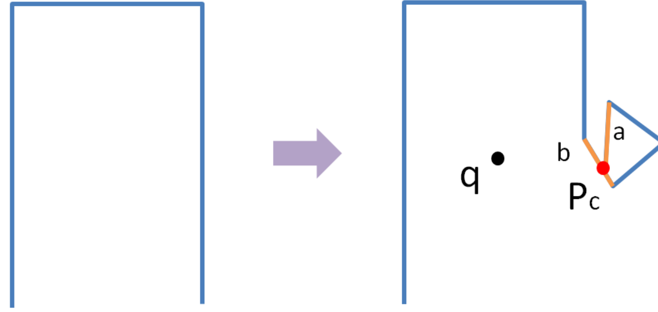


Figure 3.12: A 2D unclosed curve with self-intersection.

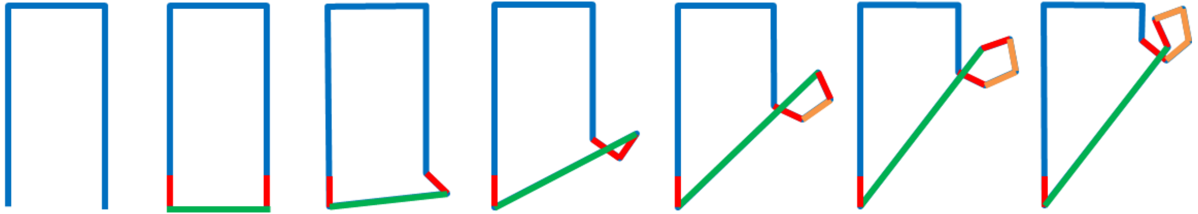


Figure 3.13: An example of handling unclosed objects.

For example, suppose that there is a 2D unclosed curve with a gap in the beginning, as shown in Figure 3.13. The curve is composed of several line segments. We create a ghost edge to enclose the gap, which is indicated by green color. The edges, which collide with the ghost edge, are indicated by red color. We collect these red edges into a violated view set V^v. Finally, the view set V^v contains the red edges and orange edges. In other words, V^v contains the edges which overlap with or pass through the ghost edge.

3.5 View-Point Scheme

For the view-point scheme, we divide the triangles of a deformable object into three view sets at first, including V_p^+, V_p^-, and V_p^0. In addition, there is an additional view set, V_p^v, for unclosed meshes. We classify the types of all triangles according to the method described in section 3.2. For all triangles, view tests are performed by $VT_p(t) = v(t) \cdot N(t)$, where $v(t)$ is a linear function in the time domain according to the movement trajectory of one of the vertices, and N(t) is the time normal vector with quadratic form in the time domain.

Triangles in V_p^+ are determined to be positively oriented. Triangles in V_p^- and V_p^0 are determined to be negatively oriented. Triangles in V_p^v are determined to be violated.

Theorem 6. *Suppose that there is a deformable object M_c and a view-point q. The view-point is put inside the object in the beginning, and we assume that the view-point does not pass through the deformable object during the simulation. If all triangles of M_c are positively oriented according to the view-point in $[0, \Delta t]$, then M_c is self-collision free.*

Proof. We prove it as follows.
If self-collision occurs at a certain time in $[0, \Delta t]$. Then, at the colliding position, there are two triangles at least that one faces the view-point q, and another is back to the view-point q, or the normal vectors of these two triangles are both perpendicular to the vectors from the view-point q to themselves. If a triangle is back to the view-point, then $VT_p(t) < 0$. If a triangle is coplanar to the view-point, then $VT_p(t) = 0$. This is a contradiction because the results of view tests of all triangles are $VT_p(t) > 0$. So, if all triangles of M_c are positively oriented according to the view-point q, then M_c is self-collision free. □

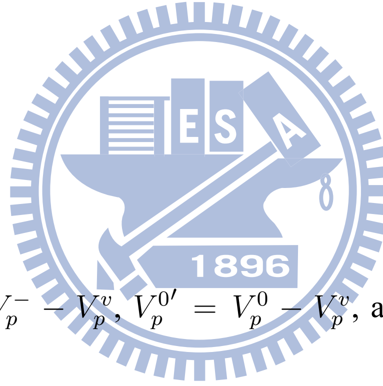
If a deformable object is determined to be self-collision free after performing view tests, then we do not perform traversal. Otherwise, we perform traversal for certain pairs of the view sets. By Corollary 3, for closed meshes, we observe that triangles in V_p^+ and V_p^- may collide with each other. And triangles in V_p^0 may collide with all triangles. Because that, for any triangle T in V_p^0, when T faces the view point at a certain time, it may collide with triangles which are back to the view point. When T is coplanar to the view point at a certain time, it may collide with other coplanar triangles. When T is back to the view point at a certain time, it may collide with the triangles which face the view point. Actually, there are several triangles at a colliding position, and two positively oriented triangles may collide with each other, or a positively oriented triangle may collide with a coplanar triangle. We just need to detect the collision with two triangles but not all colliding triangle pairs at the colliding position. We have proved that there must be two triangles at least that one faces the view point, and another is back to the view point, or the normal vectors are both perpendicular to the vectors from the view point to themselves.

So, if a deformable object is not self-collision free, we perform traversal for the following pairs to collect potentially colliding pairs.

- (V_p^+, V_p^-)
- (U, V_p^0)

U is the union set of all triangles of the deformable object. For unclosed meshes, after performing view tests, we divide all triangles into three view sets. After that, violated triangles are collected by performing boundary handling. We pick out violated triangles from the three view sets. In other words, violated triangles are special and independent, and they may collide with all triangles. We perform further checks for the following pairs to collect potentially colliding pairs.

- $(V_p^{+\prime}, V_p^{-\prime})$
- $(U', V_p^{0\prime})$
- (U, V_p^v)



$V_p^{+\prime} = V_p^+ - V_p^v$, $V_p^{-\prime} = V_p^- - V_p^v$, $V_p^{0\prime} = V_p^0 - V_p^v$, and $U' = U - V_p^v$. Note that $V_p^{+\prime} \cap V_p^{-\prime} \cap V_p^{0\prime} \cap V_p^v = \phi$.

The view-point scheme

1. Compute a point inside the deformable object M_c in the preprocessing stage.
2. Divide all triangles of the deformable object into three view sets, including V_p^+, V_p^-, and V_p^0. For unclosed meshes, there is an additional view set, V_p^v.
3. If all triangles are in the set V_p^+, then the deformable object M_c is self-collision free.
4. Otherwise, we need to perform further checks for the pairs of (V_p^+, V_p^-) and (U, V_p^0). For unclosed meshes, we need to perform further checks for the pairs of $(V_p^{+\prime}, V_p^{-\prime})$, $(U', V_p^{0\prime})$, and (U, V_p^v).

3.6 View-Line Scheme

For the view-line scheme, we divide the triangles of a deformable object into three view sets at first, including V_p^+, V_p^-, and V_p^0. In addition, there is an additional view set, V_l^v, for unclosed meshes. We classify the types of all triangles according to the method described in section 3.3. For all triangle, view tests are performed by $VT_l(t) = v(t) \cdot N(t)$, where $v(t)$ is a linear function in the time domain according to the movement trajectory of one of the vertices, and N(t) is the time normal vector with quadratic form in the time domain.

Triangles in V_l^+ are determined to be positively oriented. Triangles in V_l^- and V_l^0 are determined to be negatively oriented. Triangles in V_l^v are determined to be violated.

Theorem 7. *Suppose that there is a deformable object M_c and a view-line $\overline{q_0q_1}$. The space and the object are divided into three regions, R_0, R_1, and R_2. The view-line is put inside the object in the beginning, and we assume that the view-line does not penetrate the deformable object during the simulation. If all triangles of M_c are positively oriented according to the view-line in $[0, \Delta t]$, then M_c is self-collision free.*

Proof. We sketch our proof as follows.
Suppose that we divide the triangles into three sets, S_0, S_1, and S_2 according to the three regions, R_0, R_1, and R_2. We can show the triangles in each set S_i are self-collision free. Then, we can show that it is collision free between two sets of the sets.

Consider that there are triangles in one set, which have collision. Then assume that these two triangles are T_0 and T_1. We can show that this is not possible as there must be the third triangle which is negatively oriented with respect to the view-line according to the Jordan curve theorem. This is a contradiction to our assumption.

Consider that there are two triangles collide between two sets. Assume that these two triangles are T_0 and T_1. Then T_0 is a triangle in one set, and T_1 is a triangle in another set. However, T_0 and T_1 must also belong to the same set as there is at least one vertex of a triangle belong to both set; otherwise, these two triangles cannot collide. However, we have already shown that it is collision free in a set. □

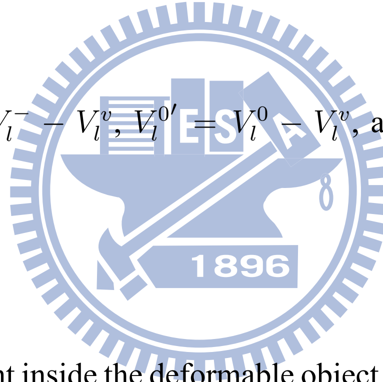
If a deformable object is self-collision free, then we do not perform traversal. Otherwise, by Corollary 4, we perform further checks for the following pairs to collect potentially colliding pairs.

- (V_l^+, V_l^-)
- (U, V_l^0)

U is the union set of all triangles of the deformable object. For unclosed meshes, we perform further checks for the following pairs to collect potentially colliding pairs.

- $(V_l^{+\prime}, V_l^{-\prime})$
- $(U', V_l^{0\prime})$
- (U, V_l^v)

$V_l^{+\prime} = V_l^+ - V_l^v$, $V_l^{-\prime} = V_l^- - V_l^v$, $V_l^{0\prime} = V_l^0 - V_l^v$, and $U' = U - V_l^v$. Note that $V_l^{+\prime} \cap V_l^{-\prime} \cap V_l^{0\prime} \cap V_l^v = \phi$.



The view-line scheme

1. Compute a line segment inside the deformable object M_c in the preprocessing stage.
2. Divide all triangles of the deformable object into three view sets, including V_l^+, V_l^-, and V_l^0. For unclosed meshes, there is an additional view set, V_l^v.
3. If all triangles are in the set V_l^+, then the deformable object M_c is self-collision free.
4. Otherwise, we need to perform further checks for the pairs (V_l^+, V_l^-) and (U, V_l^0). For unclosed meshes, we need to perform further checks for the pairs of $(V_l^{+\prime}, V_l^{-\prime})$, $(U', V_l^{0\prime})$, and (U, V_l^v).

3.7 Discussion

3.7.1 View-based approach vs. traditional approach

For the traditional approach proposed by [VMT94], an object is divided into several regular patches, called normal cones. A normal cone is defined to be a triangle group that all triangles in the group satisfy the condition: there is a vector V such that $N_i \cdot V > 0$, where N_i are the normal vectors of the triangles.

During the simulation, normal cones are computed for all the leaves of the BVH of the object in the beginning. Then, normal cones of internal nodes are computed by merging adjacent normal cones until the normal cones of two adjacent nodes cannot be merged. In the procedure, there is a lot of computation of vectors and normalization. After that contour tests are performed to check whether or not the model is self-collision free. For example, suppose that there is a patch that contains n triangles, T_1, ..., T_n. The normal vectors of all triangles are N_1, ..., N_n. If we can find a vector V that $N_i \cdot V > 0$, where $i = 1, ..., n$, then V is called the normal of the patch and the patch form a normal cone. On the other hand, we project the contour of the patch onto a plane which is perpendicular to the vector V. The projective contour in the plane is called C. The patch is determined to be self-collision free if the following two conditions are satisfied: (1) $\exists V$ such that $N_i \cdot V > 0, \forall N_i$, where $i = 1, ..., n$, and n is the number of triangles of the patch; (2) There is no intersection for C.

Algorithm 1 and 2 show the procedures of self-collision detection with the view-based scheme and the traditional approach for a closed deformable object. We can observe that there are mainly four differences.

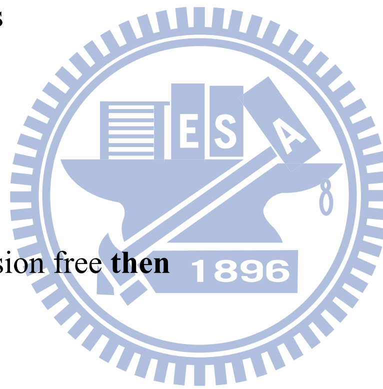
1. View primitives are computed in the preprocessing stage for the view-based scheme.
2. For the view-based scheme, we adopt performing view tests to determine whether or not the deformable object is self-collision free. For the traditional approach, they adopt computing normal cones and performing contour tests to determine whether or not the deformable object is self-collision free.

3. If an object is determined to be self-collision free after performing view tests, then traversal does not be performed. But for the traditional approach, in fact, traversal, normal cones computation, and contour tests are performed at the same time.

4. For the view-based scheme, if objects are not self-collision free, the vBVHs of all kinds of view sets are constructed. Actually, we do not reconstruct BVHs of the view sets, but extract the partial BVHs, named vBVHs, by marking the nodes of the original BVH according to the view sets. We just need to update the nodes of vBVHs that the nodes contain the triangles whose types are changed between two consecutive frames.

Algorithm 1 Self-collision detection with the view-based scheme.

1: ***Preprocessing phase***
2: compute view primitives
3: construct BVH
4: ***Simulation phase***
5: perform view tests
6: **if** the object is self-collision free **then**
7:     **return**
8: **end if**
9: build vBVHs of positively oriented and negatively oriented triangles
10: perform traversal for the vBVHs to collect potentially colliding pairs
11: perform elementary tests



E S A
1896

3.7.2 View-point vs. view-line

There are two differences between the view-point scheme and the view-line scheme.

1. The view-point scheme is simple, and only one point is required. In the step of view tests, vertex region determination does not be performed. So, for the view-point scheme, the execution time of performing view tests is faster than the view-line scheme.

Algorithm 2 Self-collision detection with the traditional approach

1: ***Preprocessing phase***

2: construct BVH

3: ***Simulation phase***

4: perform traversal for the BVH to collect potentially colliding pairs with normal cones computation and contour tests

5: perform elementary tests

2. In Figure 3.14, the red edges are negatively oriented, and the black edges are positively oriented according to the view primitive. We can observe that the number of negatively oriented edges with the view-line scheme is less than the view-point scheme. So, for the view-line scheme, the execution time of performing traversal is faster than the view-point scheme.

If the number of negatively oriented triangles is similar after performing view tests with the view-point and view-line scheme, the performance is better with the view-point scheme. In general, the performance of the view-line scheme is better than the view-point scheme. The model deforms dynamically, and we cannot predict the shape of the deformable object. So, we can decide which scheme we should use dynamically according to the number of negatively oriented triangles.

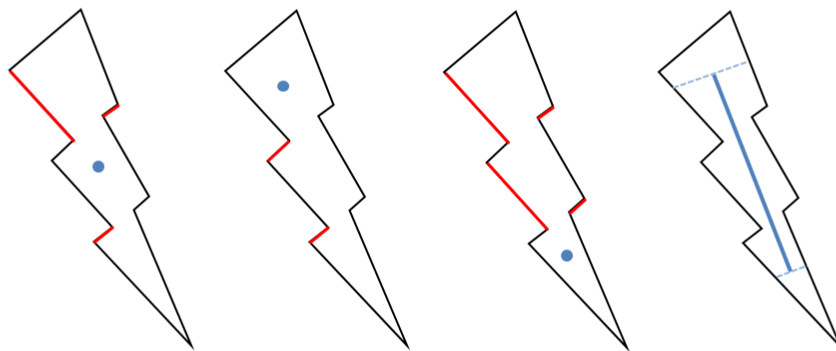
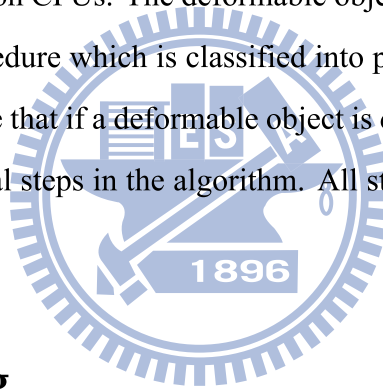


Figure 3.14: Negatively oriented edges according to the view-point and view-line schemes.

Chapter 4

Implementation on CPUs

Continuous self-collision detection is performed with the view-based approach for deformable triangle meshes on CPUs. The deformable objects can be closed or unclosed. Algorithm 3 shows the procedure which is classified into preprocessing phase and simulation phase. We can observe that if a deformable object is determined to be self-collision free, then we can skip several steps in the algorithm. All steps are described in details in the following sections.



4.1 Preprocessing

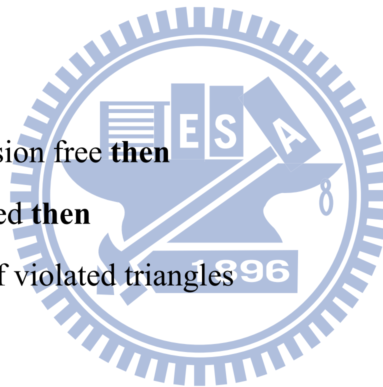
There are four parts in the preprocessing stage, including acceleration structure construction, view primitives computation, feature assignments, and ghost triangles generation for unclosed meshes.

4.1.1 Acceleration structure construction

The BVHs of all objects are constructed to accelerate performing traversal. The process is performed recursively with top-down manner. At first, the entire object is enclosed with the largest bounding volume. Then, the object is divided into two similar parts, and each part is bounded by another bounding volume. For a certain node, if all triangles are grouped in one part, and another is empty after dividing, we take one of triangles to the

Algorithm 3 Algorithm of self-collision detection with the view-based approach

1: ***Preprocessing phase***
2: construct BVH
3: compute view primitives
4: feature assignments
5: **if** the object is unclosed **then**
6:     fill holes with ghost triangles
7:     construct BVHs of ghost triangles
8: **end if**
9: ***Simulation phase***
10: perform view tests
11: **if** the object is self-collision free **then**
12:     **if** the object is unclosed **then**
13:         clear the view set of violated triangles
14:     **end if**
15:     **return**
16: **end if**
17: **if** the object is unclosed **then**
18:     perform boundary handling
19:     build vBVH of violated triangles
20: **end if**
21: build vBVHs of positively oriented and negatively oriented triangles
22: perform traversal for the vBVHs to collect potentially colliding pairs
23: perform elementary tests



empty part. This is performed recursively until one triangle is enclosed with a bounding volume. The type of bounding volumes we use is k-DOP [KHM+98] with degree 16. Other types of bounding volumes are allowed, such as axis-aligned bounding boxes [vdB99], oriented bounding boxes, and spheres [JP04].

4.1.2 View primitives computation

There are many types of view primitives. In the thesis, we use a point and a line segment for the view-based approach. We should compute the point and the line segment lying inside a deformable object in the preprocessing stage. We can calculate the center of all vertices to be the view-point. On the other hand, we construct the OBB of the object. Then, we get three principle axes according to the orientation of the OBB. The longest axis with proper length is chosen to be the view-line. We employ the method presented by [GLM96] to construct the OBB, and the view-line computation is briefly described as follows.

1. Compute the sum of all vertices and the mean value μ.
2. Generate a covariance matrix C based on μ.
   $C_{jk} = \frac{1}{n}\sum_{i=1}^{n} \overline{p_j^i}\,\overline{p_k^i}$, where n is the number of vertices, p^i is one of the vertices, and $\overline{p^i} = p^i - \mu = (p_1^i, p_2^i, p_3^i)$
3. Compute the eigenvalues and eigenvectors of the covariance matrix. The covariance matrix is symmetric, so we employ [Smi61, Ebe06] to solve the eigensystem.
4. Extract the longest axis of the eigenvectors and determine a proper length based on the bound of the OBB.

Our testing models contain two types of objects, rigid or semi-rigid objects and deformable objects. The deformable objects may interact with the rigid objects but never pass through the rigid objects. View primitives are computed according to the rigid objects or the deformable objects, and they are put inside the deformable objects. In order to

ensure that the view primitives do not penetrate the deformable objects during the simulation, we compute the view primitives according to the rigid objects generally. Because the deformable objects never pass through the rigid objects, the view primitives always lie inside the deformable objects. For a semi-rigid object, such as a moving character, we can compute the view primitives attached to the skeleton. Because the skeleton of the semi-rigid object does not penetrate the object, we can ensure that the view primitives lie inside the deformable object during the simulation.

4.1.3 Feature assignments

Self-collision occurs when the distance of any two points of the object is less than δ_d, where δ_d is a user-defined threshold. Initially, the distance of the two points is $d^0(p_0, p_1) > \delta_d$. After a simulation time step Δt, $d(p_0, p_1) \leq \delta_d$, then we determine these two points collide with and penetrate each other. We can set δ_d as the thickness of cloth. For continuous self-collision detection, we compute the exact contact time of two colliding triangles within a frame. There are two cases to handle, including vertex-triangle and edge-edge. And there are 15 elementary pairs of two triangles, including 6 vertex-triangle pairs and 9 edge-edge pairs. In other words, for a pair of triangles, we should compute elementary tests for 15 times. In order to reduce the computation of elementary tests, we perform feature assignments [WB06, CTM08] in the preprocessing stage. A triangle contains 6 features, including 3 vertices and 3 edges. For example, an edge, named E, belongs to two adjacent triangles, and we just assign the edge to one of the triangles. Then, if these two triangles both collide with another triangle T', we just need to perform the elementary tests of E between one of the triangles and T'.

4.1.4 Ghost triangles generation

Our view-based approach is suitable for closed and unclosed triangle meshes. The meshes prefer to be closed. For unclosed meshes, we fill the holes of the object with ghost triangles in the preprocessing stage. Note that ghost triangles should not collide with the

origin object. The original meshes and the ghost triangles are independent, and they form a closed object. There are two ways to fill the holes, as shown in Figure 4.1. The first way is to connect the vertices of a hole orderly. We can observe that the ghost triangles are irregular. The second way is to create a ghost vertex, and generate ghost triangles based on the vertex. The location of ghost vertices is good to be the center of all boundary vertices. Ghost triangles with a center ghost vertex are more regular, and it is simple to generate ghost triangles with a ghost vertex. Besides, we can adjust the location of ghost vertex to ensure that ghost triangles do not collide with the object in the beginning. Finally, BVHs of ghost triangles based on different holes are constructed. We briefly describe the method of generating ghost triangles as follows.

1. Extract the boundary edges by winged edge structure.

2. Sort the boundary edges and get a order list of edges.

3. For each hole, generate ghost triangles orderly according to the list of edges with or without a ghost vertex.

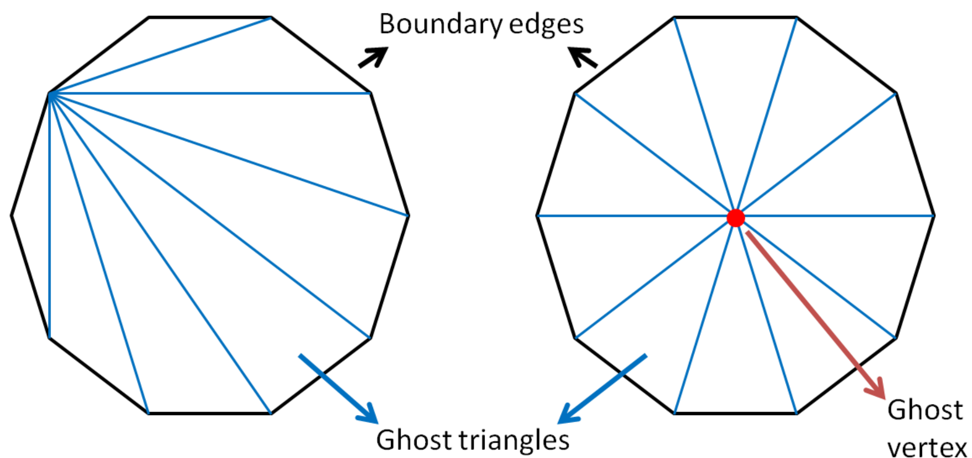


Figure 4.1: Ghost triangles with and without a ghost vertex.

4.2 View Tests

In the step of view tests, we determine the types of all triangles, including vertex region determination and triangle type determination. They are performed sequentially.

4.2.1 View-point scheme

For the view-point scheme, the vertex region determination is not required. We determine the type of a triangle by the result of $VT_p(t) = (v - q)(t) \cdot N(t)$, where v is any point of the triangle, q is the view-based point, and $N(t)$ is the time normal vector of the triangle. In $[0, \Delta t]$, the movement trajectory of v is linear, q is fixed, and $N(t)$ is a quadratic function in the time domain. Hence, $VT_p(t)$ is a cubic function in the time domain. We can compute the range of the cubic function in $[0, \Delta t]$ by an approximate min-max method [TCYM09].

$$N(t) = Dt_2 \cdot t^2 + Dt_1 \cdot t + Ds \quad (4.1)$$

$$(v - q)(t) = Vs + Vt \cdot t, Vs = v_{bgn} - q, Vt = (v_{end} - q) - Vs \quad (4.2)$$

$$(v-q)(t) \cdot N(t) = (Dt_2 \cdot Vt) \cdot t^3 + (Dt_2 \cdot Vs + Dt_1 \cdot Vt) \cdot t^2 + (Dt_1 \cdot Vs + Ds \cdot Vt) \cdot t + Ds \cdot Vs \quad (4.3)$$

v_{bgn} is the initial position of v, and v_{end} is the terminal position of v within a frame.

4.2.2 View-line scheme

We determine the types of all triangles, and $VT_l(t) = (v - q)(t) \cdot N(t)$ is computed, where v is one of the vertices of a triangle, q is the check point, and $N(t)$ is the time normal vector of the triangle. At first, we choose one of the vertices, v, and determine the regions for the vertex. $(v - q)(t)$ is computed in the step of vertex region determination according to related regions of the vertex v.

- For the region R_0, $Vs = v_{bgn} - q_0, Vt = (v_{end} - q_0) - Vs$.
- For the region R_1, $Vs = v_{bgn} - q_1, Vt = (v_{end} - q_1) - Vs$.

- For the region R_2, $Vs = v_{bgn} - proj_{v_{bgn}}$, $Vt = (v_{end} - proj_{v_{end}}) - Vs$.

v_{bgn} is the initial position of v, v_{end} is the terminal position of v, $proj_{v_{bgn}}$ and $proj_{v_{end}}$ are the projective points of v_{bgn} and v_{end} on the view-line within a frame.

Therefore, if the movement trajectory of a vertex is across multiple regions within a frame, $(v-q)(t)$ is not linear. We should compute $(v-q)(t)$ separately. So, view tests are performed for nr times, where nr is the number of regions that the chosen vertex moves across within a frame.

4.2.3 Triangle clusters

In the view-line scheme, we determine the regions for all vertices. Now, we construct triangle clusters of a deformable object in the preprocessing stage, as shown in Figure 4.2. We choose a vertex first such that the number of triangles attached to the vertex is largest. Then, all triangles adjacent to the chosen vertex form a cluster. This is performed until all the triangles of the object have assigned to a cluster. For a single triangle cluster, if there are nt triangles, we can determine the regions for these nt triangles according to the center vertex in common. Suppose that the deformable object contains nv vertices and ns triangle clusters, then we need to execute the vertex region determination for ns times instead of nv times. Because $ns < nv$, we can improve the performance.

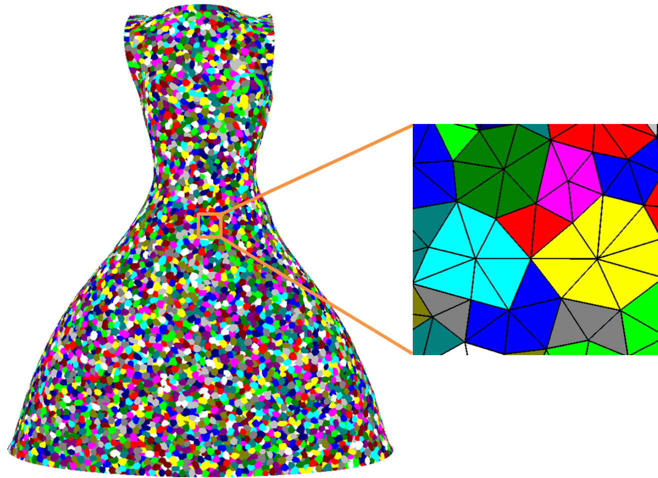


Figure 4.2: Triangle clusters of a deformable object.

4.3 Boundary Handling

For unclosed meshes with self-collision, boundary handling is performed. In the step of boundary handling, we collect the triangles which are violated to the view-based approach, and then further checks are performed for the violated triangles in the step of traversal.

We employ ghost triangles to collect violated triangles. Traversal between the BVHs of ghost triangles and the whole object is performed to collect the violated triangles. At first, we collect potentially colliding pairs between ghost triangles and the whole object. Then, elementary tests are performed for the potentially colliding pairs to get the exactly colliding triangles. We assign the violated triangles to the view set V^v. The process is performed sequentially according to the holes of the object. The violated triangles are collected until the model is self-collision free. In other words, the violated triangles keep their types until the model is self-collision free. When the model is self-collision free, the view set V^v is released, and the violated triangles are set to be positively oriented. In short, the set V^v contains the triangles which overlap with or pass through ghost triangles.

1896

4.4 Traversal

After performing view tests and boundary handling, we get several view sets. If a deformable object is not self-collision free, further checks are performed for some pairs of the view sets. Suppose that U is the union set of all triangle of the deformable object.

- For closed objects with the view-point scheme, there are three view sets, V_p^+, V_p^-, and V_p^0. If the deformable object is not self-collision free, further checks are performed for the pairs of (V_p^+, V_p^-) and (U, V_p^0).
- For unclosed objects with the view-point scheme, there are four view sets, V_p^+, V_p^-, V_p^0, and V_p^v. If the deformable object is not self-collision free, further checks are performed for the pairs of $(V_p^{+\prime}, V_p^{-\prime})$, $(U', V_p^{0\prime})$, and (U, V_p^v).
- For closed objects with the view-line scheme, there are three view sets, V_l^+, V_l^-, and

V_l^0. If the deformable object is not self-collision free, further checks are performed for the pairs of (V_l^+, V_l^-) and (U, V_l^0).

- For unclosed objects with the view-line scheme, there are four view sets, V_l^+, V_l^-, V_l^0, and V_l^v. If the deformable object is not self-collision free, further checks are performed for the pairs of $(V_l^{+\prime}, V_l^{-\prime})$, $(U', V_l^{0\prime})$, and (U, V_l^v).

However, traversal is performed for the BVHs of the view sets. We do not construct new BVHs of all view sets, but extract partial BVHs of the original BVH by marking the nodes of the original BVH. For example, if a node of the BVH contains a triangle, which is assigned to the view set V_p^+, then the node is marked for the view set V_p^+. The partial BVHs are called vBVHs. For each view set, there is a related vBVH, and nodes in the vBVHs of different view sets may be repeated. Traversal is performed for the vBVHs to collect potentially colliding pairs.

For closed meshes, there are three types of flags, P, N, and Z, in a node of the original BVH. For unclosed meshes, there are four types of flags, P, N, Z, and V, in a node of the original BVH.

1. P: If there is a triangle, which is assigned to the view set V_p^+ or V_l^+, in a node, then the flag P of the node is set to be true.

2. N: If there is a triangle, which is assigned to the view set V_p^- or V_l^-, in a node, then the flag N of the node is set to be true.

3. Z: If there is a triangle, which is assigned to the view set V_p^0 or V_l^0, in a node, then the flag Z of the node is set to be true.

4. V: If there is a triangle, which is assigned to the view set V_p^v or V_l^v, in a node, then the flag V of the node is set to be true.

We mark the flags of a node of the BVH according to the kinds of triangles within the node. Initially, the flag P of all nodes are set to be true, and other kinds of flags are set to be false. We mark all the nodes according to the different view sets sequentially. For

a view set, the marking process starts from the leaf nodes of the BVH, and bottom-up traversal is performed recursively. When the process reaches a node, which is already marked with same feature, then the recursion is terminal. So, we have three vBVHs for closed meshes and four vBVHs for unclosed meshes. For example, Figure 4.3(a) shows the original BVH, and Figure 4.3(b) shows the marking results of two different view sets. In other words, we now have two vBVHs, as shown in Figure 4.3(c) and (d). Actually, the cost of marking process is low. At first, we extract the triangles whose types are changed between two consecutive frames, and update the vBVHs according to these triangles in the runtime phase. For example, if a triangle changes its type and it is assigned from V_p^+ to V_p^-, then we need to update the vBVHs of the view sets V_p^+ and V_p^- according to the triangle.

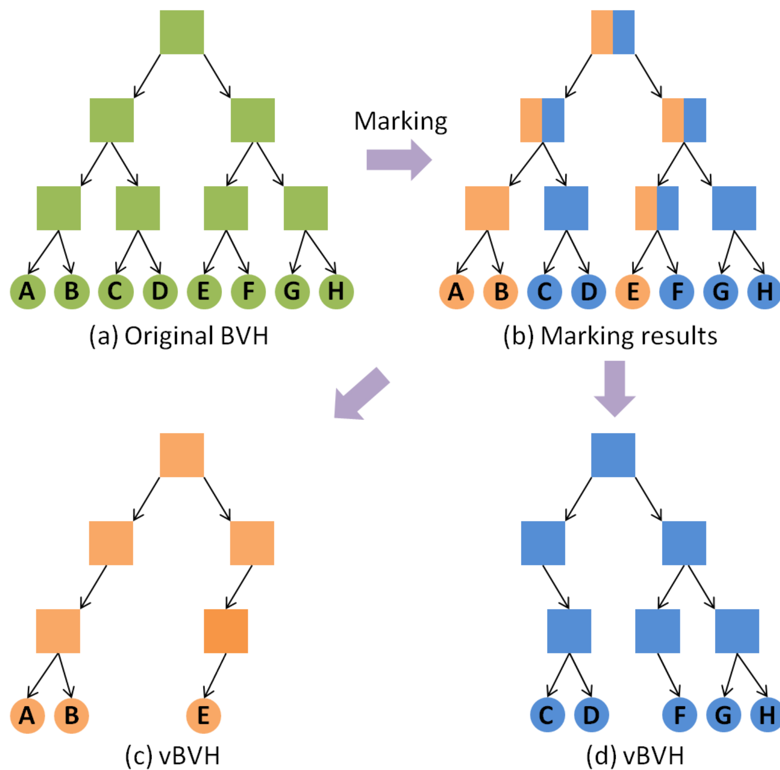


Figure 4.3: Marking process of vBVHs.

After getting the vBVHs, we compress the vBVHs further to improve performing traversal for these vBVHs. For example, Figure 4.4(c) and (d) are the results of the compact vBVHs simplifying from the original vBVHs, which are shown by Figure 4.4(c) and

(d). We implement the process of compression by the following method. Traversal is performed after getting the vBVHs and started from the root of the vBVHs. When the process of traversal encounters a node, which only has a single child, we skip the node instead of performing overlap tests, and then check its child directly, as shown in Figure 4.4. The cost of checking the number of children of a node is less than performing bounding volume overlap tests. The checking process is performed until a node with two children or a leaf node is reached. The number of overlap tests for the vBVHs is much less than full traversal for the original BVH.

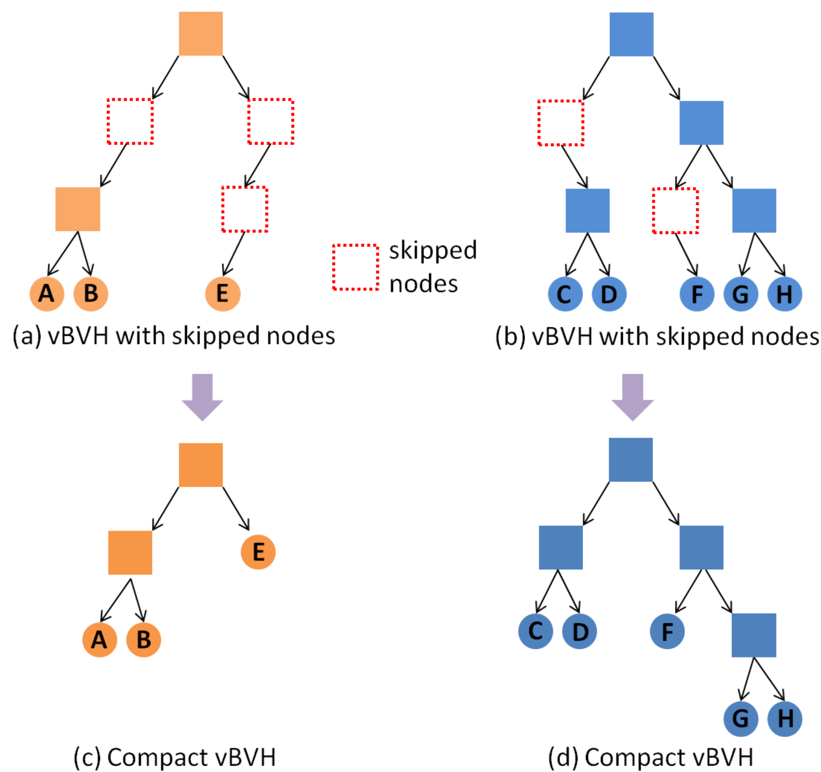
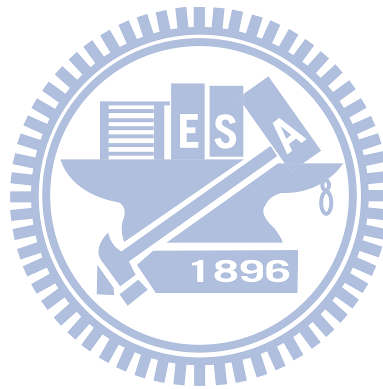


Figure 4.4: Skipped nodes in the vBVHs are adopted in order to compress the vBVHs.

4.5 Elementary Tests

After performing traversal for the vBVHs, we get a set of potentially colliding pairs. Elementary tests [TCYM09] are performed for the potentially colliding pairs sequentially with feature assignments [WB06].

For any two triangles, if they collide with each other, then the colliding point may occur between a vertex and a triangle or between two edges. There are fifteen elementary pairs of two triangles, including six vertex-triangle pairs and nine edge-edge pairs. So, we need to compute elementary tests fifteen times for each potentially colliding pair. We compute the time when the two features are coplanar for each elementary pair. Then shortest distance between the two features is computed. The vertices and edges of triangles move with constant velocities, so we can detect the collision by their movement trajectories. If two features collide with each other within a frame, we can obtain a contact time. For all effective contact time of fifteen elementary pairs, the smallest value is chosen. In addition, we can reduce the computation of elementary tests by using feature assignments.



Chapter 5

Implementation on GPUs

5.1 GPU Architecture

On graphics processing units (GPUs), the number of processors is more than multi-core platforms, and it supports higher degree of parallelism. In recent years, there are more and more researches of computer graphics implemented by GPUs. The view-based approach for continuous self-collision detection is suitable for executing in parallel. We implement the approach on GPUs with CUDA.

A graphics chip is composed of several multiprocessors. A multiprocessor is composed of 8 processors, which are divided into two groups. So, each group contains four processors and forms a SIMP unit. A GPU process, called a kernel, is executed by a grid, which is formed by several blocks. A block is mapped to a multiprocessor, and a lot of threads can be executed in a block. Every 32 threads are grouped together and formed a warp, which is the smallest processing unit in CUDA. 32 threads in a warp are executed in parallel at a certain time. Besides, a multiprocessor also contains special function units, registers, shared memory, constant memory, and local memory.

On GPUs, there are several types of memory with different performance [NVI10b]. Table 5.1 shows the property of all memory. We can observe that register and shared memory are on chip, and the access speed are the fastest. But the number of registers and the size of shared memory are limited. So, we use these kinds of memory efficiently. In

Memory 1	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	n/a	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	n/a	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Table 5.1: Salient features of device memory for devices of compute capability 1.x. The information is quoted from [NVI10a].

general, variables which are accessed frequently or shared by all threads in the same block should be stored by using shared memory. We can copy the data from global memory to shared memory at the beginning of kernel execution.

The development of GPUs is rapid. Design mechanisms are different from different generation of GPU architectures. Our GPU processes are performed by NVIDIA's devices of Fermi architecture. Figure 5.1 shows the comparisons between three GPU generations. Fermi architecture is much different from previous GPUs. There are several significant improvements described as follows.

1. The processors are called CUDA cores, and the number of CUDA cores is much more than before.

2. Compute capability of double precision is more than 8 times faster, and single precision is more than 2 times faster. Besides, fused multiply-add (FMA) computation replaces multiply-add (MAD) computation.

3. The number of special function units is double. And there are 16 CUDA cores in a multiprocessor instead of 8 CUDA cores. Therefore, two warps can be scheduled at the same time in a block.

4. For global memory and local memory, L1/L2 cache is supported.

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Figure 5.1: Summary table of various GPU architecture. (The information is quoted from [NVI09].)

5. 64KB of RAM are provided for shared memory and L1 cache. We can schedule the size manually by two policies. 16KB for shared and 48KB for L1 cache; 48KB for L1 cache and 16KB for shared memory. So, the size of shared memory we can use is larger.

6. Fermi supports single-error correct and double-error detect (SECDED) error correcting code (ECC). The data can be maintained stably.

7. __device__ functions support recursion. Note that __device__ functions are only called by devices.

There are two important issues that should be considered in parallel computing on GPUs. The first issue is about the work load balancing for the threads, and each thread should access data systematically so as to improve the degree of parallelism. The second issue is that the calculated results of each thread are stored systematically in parallel according to the related thread id. In order to handle these two issues, we employ fundamental parallel functions, including parallel prefix sum, parallel sort, and parallel reduction [HOS+07, NVI].

5.2 Use of Data on GPUs

5.2.1 Static data

```
int4     *m_dFaceVertIdx;
int      *m_dNumFeaturePoint;
int      *m_dNumFeatureEdge;
int      *m_dEdgeVertIdx;
```

Before performing the process on GPUs, we copy the data from main memory to GPU memory. These data are static and just need to be copied once, including vertex indices and features of all triangles. For all triangles, we store three vertex indices into an array and copy to GPU memory so as to access vertex positions. A triangle contains six elementary primitives, including three vertices and three edges. Feature assignments are performed to decide the number of primitives for each triangle in the preprocessing stage. The data of vertices and edges are stored orderly based on the features.

5.2.2 Dynamic data

```
float4   *m_dVertArr;
float4   *m_dVelArr;
float    *m_dFaceBVbound;
float    *m_dkDopNodeList;
float    *m_dGhostNodeList;
```

Because our experiments are performed based on benchmarks, we need to load and update the information of models and copy these data from main memory to GPU memory each frame, including vertex positions, vertex velocities, and bounding volume information.

Data of BVH can be stored in a one-dimension array. We cannot use pointers to access nodes of BVH on GPUs. We store the BVH into a node list by level-order, as shown in Figure 5.2. For a single node, we store the data include bounding volume information, a leaf flag, the index of its left child in the node list (if it is an internal node), the index of the triangle in the node (if it is a leaf node). Note that the right child of a node must be adjacent to the left child in the node list. The bounding volume information is dynamic, and it should be updated each frame, but the index of the left child or the triangle is static. Suppose that there are n nodes in the BVH, then we allocate $20 \times n$ spaces for the node list that 20 is a multiple of 4. The bounding volume information of each triangle is contained in the node list, but these data are used frequently. So, we store the bounding volume information of all triangles additionally into an array.

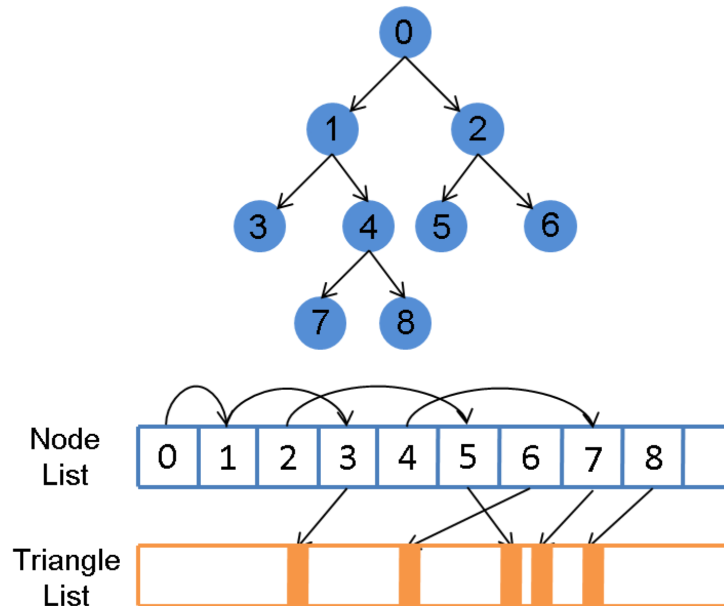
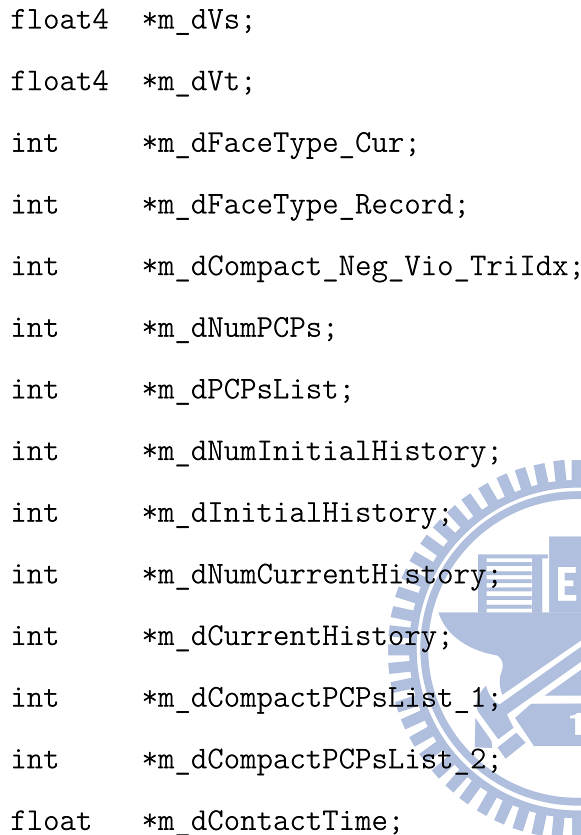


Figure 5.2: Level-order node list of the BVH.

5.2.3 Global memory allocation in advance

```
float4  *m_dVs;
float4  *m_dVt;
int     *m_dFaceType_Cur;
int     *m_dFaceType_Record;
int     *m_dCompact_Neg_Vio_TriIdx;
int     *m_dNumPCPs;
int     *m_dPCPsList;
int     *m_dNumInitialHistory;
int     *m_dInitialHistory;
int     *m_dNumCurrentHistory;
int     *m_dCurrentHistory;
int     *m_dCompactPCPsList_1;
int     *m_dCompactPCPsList_2;
float   *m_dContactTime;
```



We need to allocate a set of GPU memory in the preprocessing stage in order to store data in runtime. Thus, we can reduce the memory time for memory allocation in runtime.

1. We store linear vector variation from a vertex to check points for all vertices in $[0, \Delta t]$ into m_dVs[] and m_dVt[].

2. We use m_dFaceType_Cur[] to store temporary types of all triangles after performing view tests and boundary handling.

3. Before performing traversal, we compress the triangle type list and collect the indices of negatively oriented and violated triangles into m_dCompact_Neg_Vio_TriIdx[].

4. After performing view tests and boundary handling, the types of all triangles should be updated from m_dFaceType_Cur[] to m_dFaceType_Record[].

5. m_dNumPCPs[] records the number of potentially colliding pairs.

6. m_dPCPsList[] records the potentially colliding pairs of all triangles. We allocate 16 spaces per triangle. So, the size of m_dPCPsList[] is $16\times$ the number of all triangles.

7. We perform traversal in the preprocessing stage, and store the colliding situation into m_dInitialHistory[] for each triangle. We allocate 512 spaces per triangle. So, the size of m_dInitialHistory[] is $512\times$ the number of all triangles.

8. m_dNumInitialHistory[] records the number of nodes for the initial traversal based on each triangle.

9. We record the current results of traversal into m_dCurrentHistory[] and store the number of history nodes into m_dNumCurrentHistory[] every frame. We allocate 512 spaces per triangle. So, the size of m_dCurrentHistory[] is $512\times$ the number of all triangles.

10. Before performing elementary tests, we compress the PCPs list based on each triangle. We allocate global memory dynamically according to the number of potentially colliding pairs, and collect potentially colliding pairs into m_dCompactPCPsList_1[] and m_dCompactPCPsList_2[].

11. m_dContactTime[] records contact time of all potentially colliding pairs, and these data are copied back to main memory.

5.3 View Tests

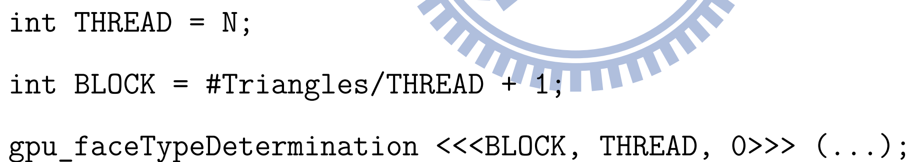
In the step of view tests, we want to determine the types of all triangles. View tests are performed by two parts, including vertex region determination and triangle type determination. The processes are performed sequentially.

5.3.1 Vertex region determination

```
int THREAD = N;
int BLOCK = #Vertices/THREAD + 1;
gpu_vertexRegionDetermination <<<BLOCK, THREAD, 0>>> (...);
```

The activity regions for all vertices in $[0, \Delta t]$ are determined. The GPU process and CPU process are the same. But one thread per vertex is created to execute the process. So, totally nv threads are created and N threads per block, where nv is the number of vertices.

5.3.2 Triangle type determination



```
int THREAD = N;
int BLOCK = #Triangles/THREAD + 1;
gpu_faceTypeDetermination <<<BLOCK, THREAD, 0>>> (...);
```

The types of all triangles are determined according to their orientation related to the view primitive in $[0, \Delta t]$. The GPU process and CPU process are the same. But one thread per triangle is created to execute the process. So, totally nt threads are created and N threads per block, where nt is the number of triangles.

For closed meshes, we divide all triangles into three view sets, The types of all triangles are computed every frame. For unclosed meshes, we divide all triangles into four view sets. For a certain frame, view tests are performed for non-violated triangles to determine the types that violated triangles are skipped. We keep the types of violated triangles until the model is self-collision free.

On the other hand, we do not employ triangle clusters to improve the performance on GPUs. The main improvement of employing triangle clusters is to reduce the execution time of vertex region determination. On CPUs, vertex region determination is performed sequentially according to each vertex, and the execution time of vertex region determination is proportional to the number of vertices. But on GPUs, vertex region determination is performed in parallel for all vertices. The impact of the number of vertices on the performance of vertex region determination is much less on GPUs. So, we do not adopt triangle clusters to improve the performance.

5.4 Boundary Handling

For unclosed meshes, we need to perform boundary handling. We extract the triangles which collide with or pass through ghost triangles and record these triangles until the object is self-collision free.

```
int THREAD = N;
int BLOCK = #Triangles/THREAD +1;
gpu_boundaryHandling <<<BLOCK, THREAD, 0>>> (...);
```

Violated triangles will keep their types until the object is self-collision free. So, we do not perform boundary handling for violated triangles. There are two policies. The simplest method is to create one thread per triangle. So, totally nt threads are created, and N threads per block, where nt is the number of triangles. Boundary handling is performed for each thread, but threads responsible for violated triangles do not execute any computation. In other words, the process just need to perform boundary handling and update the types for non-violated triangles.

```
int THREAD = N;
```

```
int BLOCK = #Triangles with non-violated/THREAD +1;
gpu_boundaryHandling <<<BLOCK, THREAD, 0>>> (...);
```

By another method, we will compress the triangle type list and collect the indices of non-violated triangles before performing boundary handling. So, we get a compact array which records the indices of non-violated triangles. One thread per non-violated triangle is created. So, totally nv threads are created, and N threads per block, where nv is the number of non-violated triangles. Compressing the triangle type list can reduce the number of threads, and the workload of each thread is balanced. But there are some penalties for compressing.

The steps of compression are described as follows. At first, we divide the triangle type list into nc chunks that 8 elements per chunk and nc threads are created. Then, we need to compute the number of non-violated triangles for each chunk and store the results in an array A. Note that the size of elements in A is nc. Next, we calculate the exclusive prefix sum for the array A and store the results in another array B. The elements in the array B indicate the start index in the compact array for each chunk. Finally, we store the indices of non-violated triangles in the compact array according to the prefix sum in the array B, as shown in Figure 5.3. The elements in the compact array indicate the triangle indices which are non-violated.

In the step of boundary handling, each thread is responsible for traversing between the target triangle and the BVHs of ghost triangles. For example, if there are nh holes in the deformable object, then the target triangle is traversed for these nh BVHs of ghost triangles sequentially. The process is ended as long as the target triangle collides with any ghost triangle. In fact, we determine whether or not the target triangle collides with a ghost triangle by their bounding boxes instead of triangles. If we compute collision detection by triangles exactly, the performance is bad. There are two reasons. On GPUs, it is good to perform simple and consistent computation in a kernel. Performing traversal with exact collision detection contains the computation of overlap tests for bounding volumes and elementary tests of triangles. It is more complicated, and the ability of parallel computing

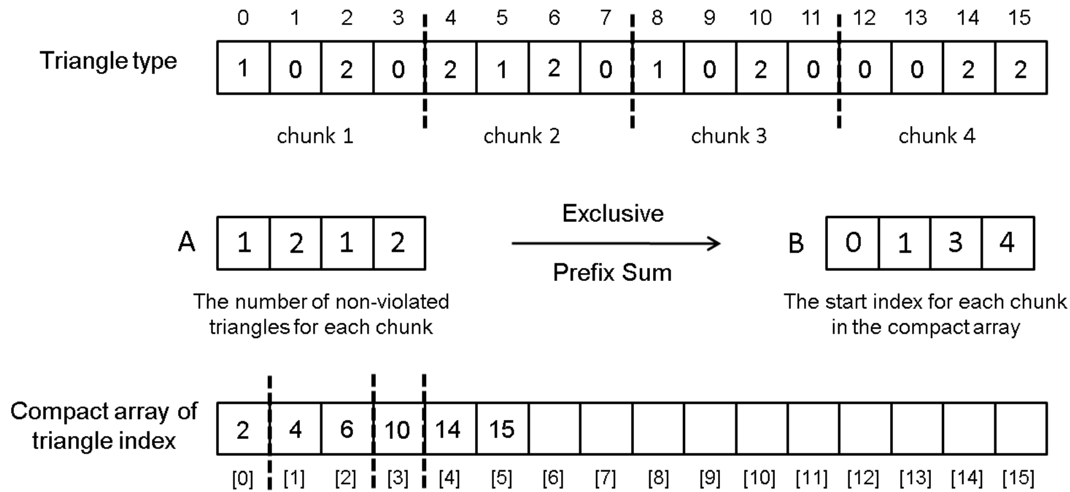


Figure 5.3: Preprocess the triangle type list before performing boundary handling.

is reduced. Another reason is that there are a lot of global memory accesses. We need to access the data of bounding volumes and triangle information at the same time, and the data in the cache is not consistent. The number of violated triangles with inexact collision detection by bounding volumes is larger than exact collision detection by triangles. And then, in the step of traversal, the computation and the number of potentially colliding pairs are larger. But the entire performance is better.

On CPUs, the cache is larger and the computation ability of processors is better, so we perform boundary handling with exact collision detection by triangles.

We employ shared memory to store the data of bounding volumes of each triangle. In the stage of traversal for each triangle, there are a lot of overlap tests, and the data of bounding volume of target triangles are used frequently. So, we employ shared memory to store the data, as shown in Algorithm 4. For each thread in a block, it copies the data from the global memory to shared memory itself in the beginning.

In the stage of traversal, we employ stack to perform traversal instead of recursion, as shown in Algorithm 4. Fermi GPU supports recursion, but the performance of recursion is slower. For a certain node of BVH, there are three cases.

- Case 1: There is no overlapping between the bounding volume of the target triangle and the node. The process is ended.

Algorithm 4 Algorithm of coping the data of bounding volumes into shared memory

```
1:  int tid = threadIdx.x;
2:  int i = blockIdx.x*blockDim.x+threadIdx.x;
3:  __shared__ float s_BVbound[N*16]; // 16-Dops
4:  if i < #Triangles then
5:      for k = 0 to 7 do
6:          S_BVbound[tid*16+k] = g_FaceBVbound[i*16+k];
7:          S_BVbound[tid*16+8+k] = g_FaceBVbound[i*16+8+k];
8:      end for
9:      ...
10: end if
```

- Case 2: The bounding volume of the target triangle collides with the node, and the node is not a leaf. Continue to traverse for the left child of the node and store the index of the right child in a stack. Because we store the BVH in a node list by level-order, the right child must be adjacent to the left child. And the index of the right child is the index of the left child plus one.

- Case 3: The bounding volume of the target triangle collides with the node and the node is a leaf. The target triangle is violated.

If there is no overlapping between the bounding volume of the target triangle and the node or the node is a leaf, we get the index of the next tested node from the stack. The process is ended when the stack is empty.

5.5 Traversal

On CPUs, we usually employ some acceleration structures, like BVH and kd-trees, to improve the performance of traversal. For collision detection, we take two roots of BVHs and start performing traversal of two objects. If these two roots collide with each other, we continue to traverse for four pairs based on these two roots. The process is ended until the

Algorithm 5 Algorithm of traversal for boundary handling with a stack

```
1:  int stack[SIZE]
2:  BVT = bounding volume of the target triangle
3:  current node = the root of the BVH of a certain hole
4:  while true do
5:      if current node and BVT overlap then
6:          if current node is a leaf then
7:              The target triangle is violated.
8:          else
9:              current node = left child of current node
10:             Push(stack, current node index + 1)
11:             continue
12:         end if
13:     end if
14:     if stack is empty then
15:         break
16:     end if
17:     current node = Pop(stack)
18: end while
```

considered nodes are both leaf nodes. However, on GPUs, if we start to perform traversal from the roots of BVHs, the degree of parallelism is bad for the first few steps. For example, the first step of traversal is computed by only one thread, and the second step of traversal is computed by four threads if the roots overlap. So, we do not perform traversal in accordance with the way on CPUs. In general, traversal is performed according to each triangle in order to increase the degree of parallelism. In other words, traversal on GPUs is performed with a triangle-based manner. We can store potentially colliding pairs of each triangle individually in an array with respect to the triangle id. We have four policies to perform traversal on GPUs.

```
int THREAD = N;
int BLOCK = #Triangles/THREAD + 1;
gpu_traversal <<<BLOCK, THREAD, 0>>> (...);
```

The simplest method is to create one thread per triangle. So, totally nt threads are created, and N threads per block, where nt is the number of triangles. Traversal is performed for each thread, but threads responsible for positively oriented triangles do not execute any computation. In other words, the process just needs to perform traversal for negatively oriented and violated triangles. We employ stack to perform traversal. Besides, we store the bounding volume bounds of each triangle in shared memory to reduce global memory accesses. However, each thread needs to perform traversal from the root to leaves for each triangle, and there are a lot of overlap tests. For a single thread on GPUs, the computation ability is worse than CPUs, so the performance is not good.

```
int THREAD = N;
int BLOCK = #Triangles with negatively oriented and violated/THREAD + 1;
gpu_traversal <<<BLOCK, THREAD, 0>>> (...);
```

By the second method, We preprocess the triangle type list before performing traversal. We compress the triangle type list and collect the indices of negatively oriented and violated triangles. So, we get a compact array which records the indices of negatively oriented and violated triangles. After performing view tests and boundary handling, we get an array which indicates the types of all triangles. For example, Figure 5.4 shows the results, where positively oriented triangles are indicated by 0, negatively oriented triangles are indicated by 1, and violated triangles are indicated by 2. To compress the triangle type list can reduce the number of threads we need, and the workload of each thread is balanced. But some penalties are needed for compressing. After compressing, we create nnv threads to perform traversal in parallel, where nnv is the number of negatively oriented and violated triangles. For each triangle, it needs to be traversed to the entire BVH.

1	0	1	0	2	1	2	0	1	0	1	0	0	0	2	2

Figure 5.4: The triangle type list.

The steps of compression are similar to what we do in the step of boundary handling. At first, we divide the triangle type list into nc chunks and 8 elements per chunk, and nc threads are created. Then, we need to compute the number of negatively oriented and violated triangles for each chunk and store the results in an array A. Next, we calculate the exclusive prefix sum for the array A and store the results in another array B. The elements in the array B indicate the start index in the compact array for each chunk. Finally, we store the indices of negatively oriented and violated triangles in a compact array according to the prefix sum in the array B, as shown in Figure 5.5. The elements in the compact array indicate the triangle indices, which are negatively oriented or violated.

The following two policies are based on the front-based decomposition algorithm presented by Tang et al. [TMT09]. There is a bounding volume test tree (BVTT) which contains all the overlap test pairs during traversal. Some nodes in BVTT are marked by front. Front nodes indicate the terminal points for traversing in the previous frame. In the next frame, traversal is performed from these front nodes instead of roots. If the quality

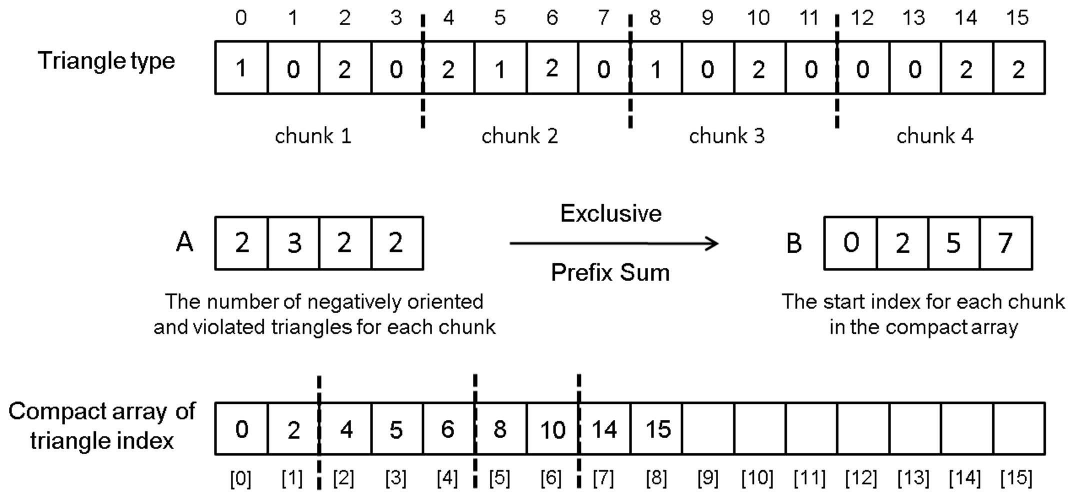


Figure 5.5: Preprocess the triangle type list before performing traversal.

of the BVTT front is over than a threshold, the BVTT front have to be rebuilt.

```
int THREAD = N;
int BLOCK = #Triangles with negatively oriented and violated/THREAD + 1;
gpu_traversal <<<BLOCK, THREAD, 0>>> (...);
```

By the third policy, nnv threads are created, where nnv is the number of negatively oriented and violated triangles. Each thread is responsible for traversing between the target triangle and the BVH of the original object. We do not build BVTT because our traversal is performed based on each triangle. We record a list of nodes according to each triangle each frame, that the nodes contain two types. One collides with the target triangle, which contains leaf nodes, and another does not collide with the target triangle, which contains internal nodes and leaf nodes. These nodes are called history nodes. The list is composed of $512 \times n$ elements, where n is the number of triangles. So, for each triangle, we allocate 512 spaces to store the history nodes. Initially, the list is composed of the root nodes of the BVH. It means that, for all triangles, traversal should be started from the root. After the initial stage, there is a list of pairs for each triangle. So, traversal is performed for the history nodes in the next frame. The algorithm is shown in Algorithm 6.

Algorithm 6 Algorithm of traversing with history nodes

```
1:  int stack[SIZE]
2:  int round = the number of history nodes of the target triangle
3:  BVT = bounding volume of the target triangle
4:  for i = 0 to round-1 do
5:      current node = the related element in the history list
6:      if current node and BVT do not overlap then
7:          Push(HistoryList, current node index)
8:          continue
9:      end if
10:     if current node is a leaf then
11:         Push(PCPList, the index of the triangle within current node)
12:         Push(HistoryList, current node index)
13:         continue
14:     end if
15:     current node = left child of current node
16:     Push(stack, current node index + 1)
17:     while true do
18:         if current node and BVTT overlap then
19:             if current node is a leaf then
20:                 Push(PCPList, the index of the triangle within current node)
21:                 Push(HistoryList, current node index)
22:             else
23:                 current node = left child of current node
24:                 Push(stack, current node index + 1)
25:                 continue
26:             end if
27:         end if
28:         if stack is empty then
29:             break
30:         end if
31:         current node = Pop(stack)
32:     end while
33: end for
```

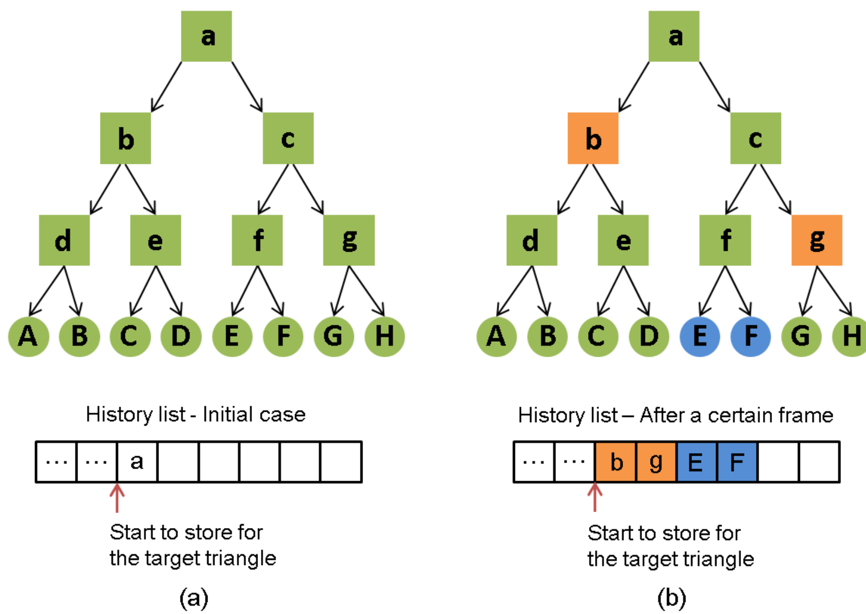


Figure 5.6: Examples for history lists.

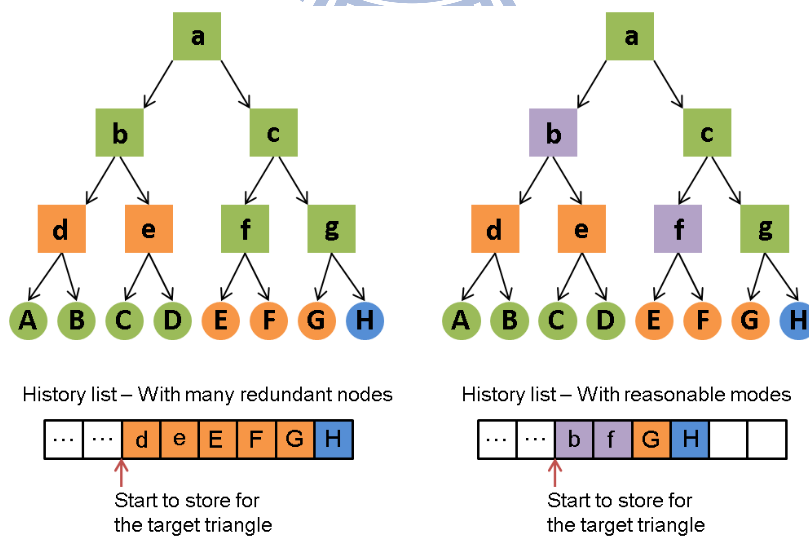


Figure 5.7: History list with many redundant nodes and reasonable nodes.

Figure 5.6(a) shows that the initial BVH. We record the root node for the target triangle in the history list. In Figure 5.6(b), traversal between the target node and the initial BVH is terminated at the nodes which are indicated by orange and blue color. The bounding volume of the target triangle collide with the blue nodes, E and F, and does not collide with the orange nodes, b and g. We record the nodes b, g, E, and F in the history list for the target triangle. In the next frame, the target triangle starts to perform traversal from nodes b, g, E, and F sequentially instead of the root, a. The performance is better than the previous methods. There are two reasons. The first reason is that, for a target triangle, the average number of performing overlap tests when traversing from its history nodes is less than traversing from the root. The second reason is that the colliding results of a target triangle is probably similar to the previous frame. So, we can improve the performance of performing traversal using history nodes. But as time goes by, history nodes would become inappropriate for performing traversal, and the number of history nodes for each triangle would increase. So, there are a lot of overlap tests, including unnecessary tests. For example, the history nodes may contain two sibling nodes which both not overlap with the target triangle. We should perform overlap tests for their parent instead of two sibling nodes, as shown in Figure 5.7. We can improve this problem by the following solution. When the number of history nodes increases over than a threshold, we can restart to traverse from the root node instead of the history nodes in the next frame.

```
int THREAD = N;
int BLOCK = #Triangles with negatively oriented and violated/THREAD + 1;
gpu_traversal <<<BLOCK, THREAD, 0>>> (...);
```

For the fourth policy, we need to preprocess traversal for each triangle and record the colliding situation between the initial BVH and each triangle. In fact, in the initial case, there is no self-collision between all triangles except for adjacent triangles. We want to record the initial history nodes and start to perform traversing from these nodes instead of the root node. We can observe that the number of performing overlap tests from the initial

history nodes is less than the root node. Besides, there is one more condition to discard the history nodes. Traversal is performed from the initial history nodes for every few frames. The performance is the best for the fourth policy.

So, there are two conditions that history nodes should be released.

1. When the number of history nodes is too large over a threshold.
2. For every few frames.

5.6 Elementary Tests

After performing traversal, we can get a set of potentially colliding pairs (PCPs) for each triangle. PCPs based on each triangle are stored in an array which consists of $16 \times nt$ elements, where nt is the number of triangles. So, for each triangle, we allocate 16 spaces to store PCPs. We perform elementary tests of the PCPs with two different policies.

```
int THREAD = N;
int BLOCK = #Triangles/THREAD + 1;
gpu_elementaryTest <<<BLOCK, THREAD, 0>>> (...);
```

Because we store the PCPs in an array according to each triangle, nt threads are created to perform elementary tests, where nt is the number of triangles. Each thread is responsible for computation of a set of PCPs belongs to a triangle. There is a problem that there may be 0 to 16 PCPs for a triangle. The workload of each thread is different. So we sort the array which contains the data of the number of PCPs for each triangle, and assign work to threads according to the number of PCPs to get the better workload balance. The performance is improved, but some penalties are needed for sorting.

```
int THREAD = N;
```

```
int BLOCK = #PCPs/THREAD + 1;
gpu_elementaryTest <<<BLOCK, THREAD, 0>>>(...);
```

By another way, we create np threads to perform elementary tests, where np is the number of PCPs. So, each thread is responsible for one PCP. We need to preprocess the PCPs array before performing elementary tests. Initially, the PCPs are distributed in an array based on each triangle, and we have the data of the number of PCPs for each triangle in an array, called numPCPs array. The exclusive prefix sum of numPCPs array is computed. Then, we can employ nt threads to collect the PCPs for each triangle in parallel, where nt is the number of triangles. Finally, we get a compact array consists of the PCPs without redundant space, as shown in Figure 5.8.

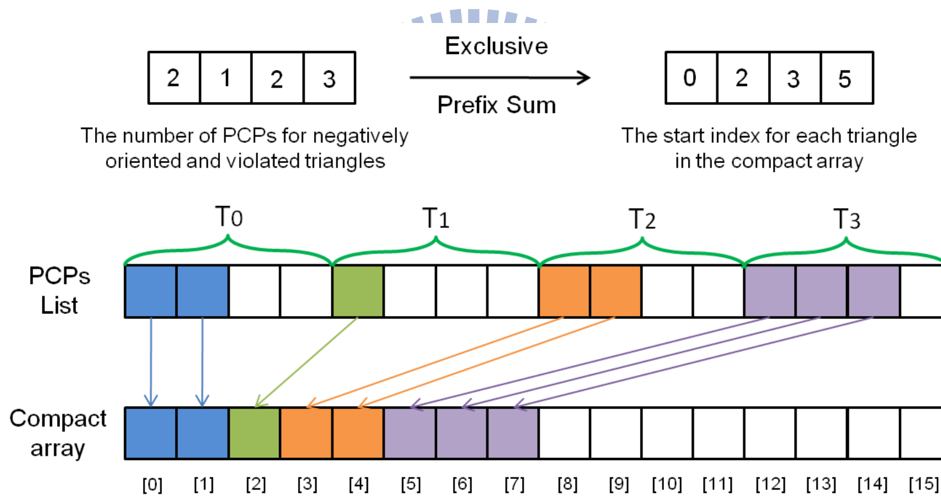


Figure 5.8: PCPs array preprocessing.

We use two methods of elementary tests on GPUs, including Interval Newton and Cubic Solver [TCYM09]. On CPUs, the process of Interval Newton is recursive, and recursive functions are supported by Fermi GPUs. All the processes on CPUs and on GPUs are the same. Additionally, for Cubic Solver, we need to solve the cubic function exactly, and all the processes on CPUs and on GPUs are the same. Besides, we employ feature assignments [WB06] to reduce the computation of elementary tests.

Chapter 6

Results and Discussion

We implemented the view-based approach using both CPUs and GPUs. The experiments were performed on an Intel(R) Core i7 CPU 870 @ 2.93GHz 2.93GHz 4 GB main memory and one thread is used, and on NVIDIA Geforce GTX 480 graphics card that 2.0 compute capability is supported. On the other hand, the experiments were classified into two types.

1. We performed the entire procedure of continuous collision detection includes inter-object and self-object using CPUs.
2. We performed continuous inter-collision detection contains BVH update using CPUs and continuous self-collision detection using GPUs.

The type of bounding volume is K-DOPs with degree 16. Execution time of kernel functions was measured by the CUDA compute visual profiler.

6.1 Animation Benchmarks

Our experiments were performed on six animation benchmarks. The benchmarks are generated by the dynamic simulation system created by our lab. Each benchmark contains a semi-rigid or rigid object and a deformable cloth. Table 6.1 shows the model complexities, and the information of holes and ghost triangles for unclosed models. Figure 6.1,

6.2, 6.3, 6.4, 6.5, and 6.6 show a series of snapshots for the six animation benchmarks. In addition, the animation benchmarks are described as follows.

Ani.	Closed	#Frames	Semi-rigid and Rigid Objects		Deformable Objects			
			#Tri. (K)	#Vert. (K)	#Tri. (K)	#Vert. (K)	#Holes	#Ghost Tri.
One	Yes	1176	40	20	12	6	-	-
Two	Yes	721	40	20	49	25	-	-
Three	No	1167	34	17	50	25	4	549
Four	No	971	34	17	50	25	4	549
Five	No	1568	19.2	9.6	76	39	2	320
Six	No	1166	34	32	80	40	1	800

Table 6.1: Model complexities and information of ghost triangles for unclosed models.

- **Ani. One:** A bunny and a ball with low complexity. The bunny is semi-rigid and moves up and down inside the deformable ball. The deformable ball falls and interacts with the bunny.

- **Ani. Two:** A bunny and a ball with high complexity. Similar to Ani. One, but the number of triangles of the deformable ball is larger.

- **Ani. Three:** A character wears a dress in a windy environment. The character is rigid. There are a lot of wrinkles for the deformable dress.

- **Ani. Four:** A walking character wears a dress.

- **Ani. Five:** A rod and a cloth. The rod is rough and its shape bends gradually. There is a cylinder-liked deformable cloth which covers and interacts with the rod. As time goes by, the rod is bending and the deformable cloth slips along the surface of the rod.

- **Ani. Six:** A board and a cloth in a windy environment. The square board is rigid. There is a deformable cloth spreads on the board. The cloth is affected by wind and interacts with the board.

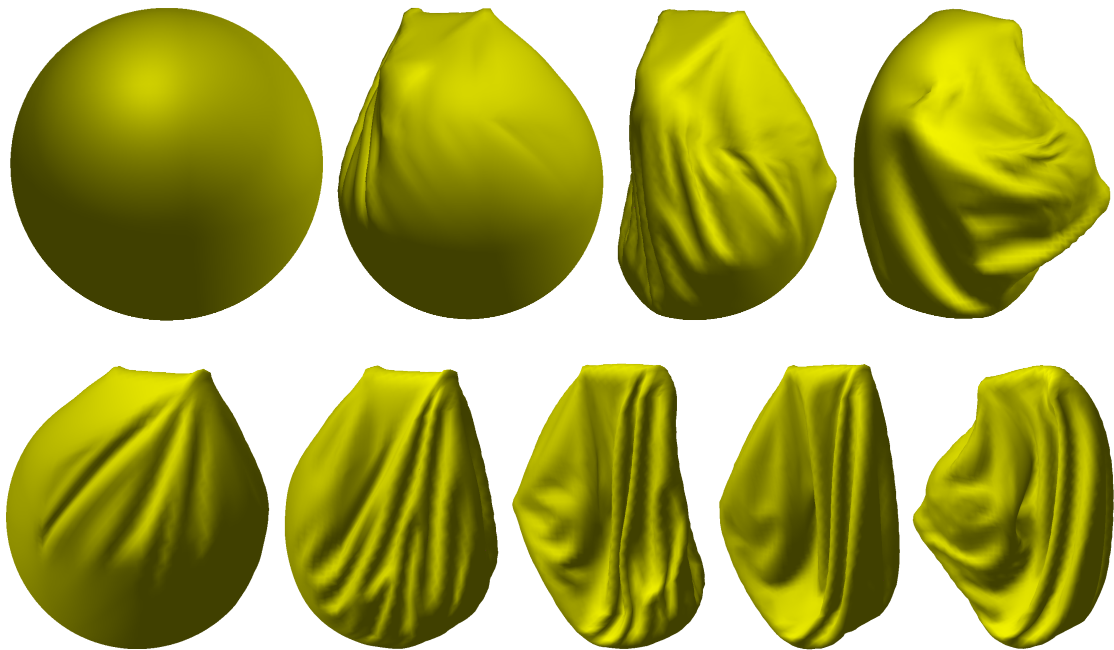


Figure 6.1: A series of snapshots of Ani. one. The first row and the second row are viewed from different viewpoints.

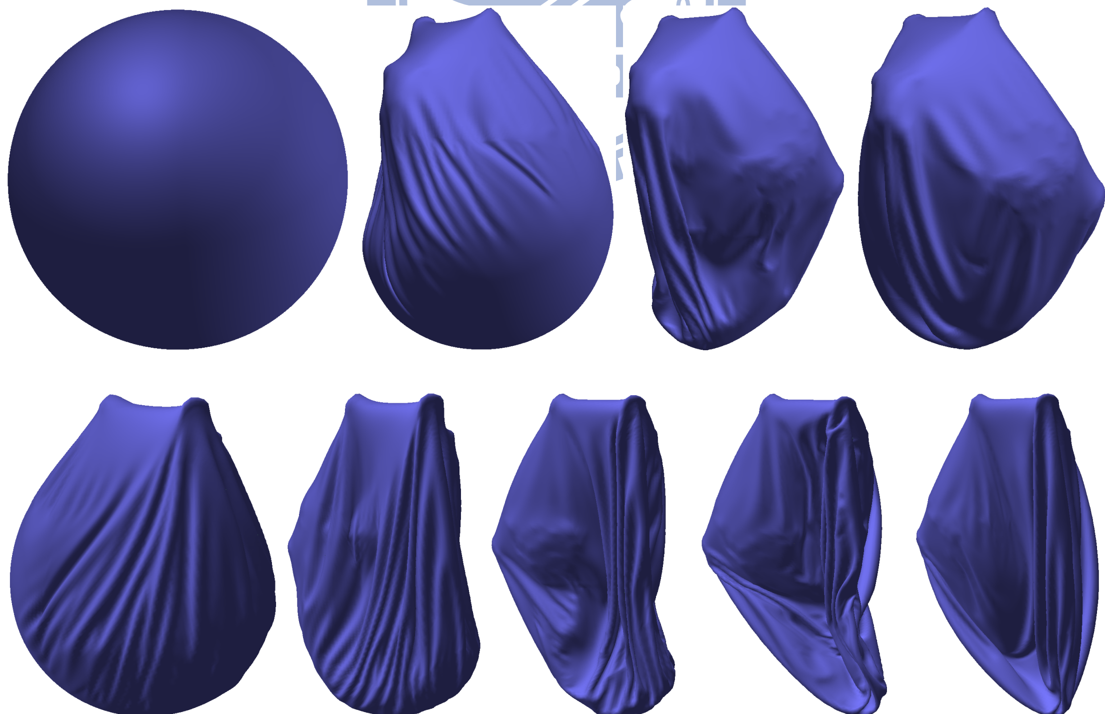


Figure 6.2: A series of snapshots of Ani. two. The first row and the second row are viewed from different viewpoints.

Figure 6.3: A series of snapshots of Ani. three.

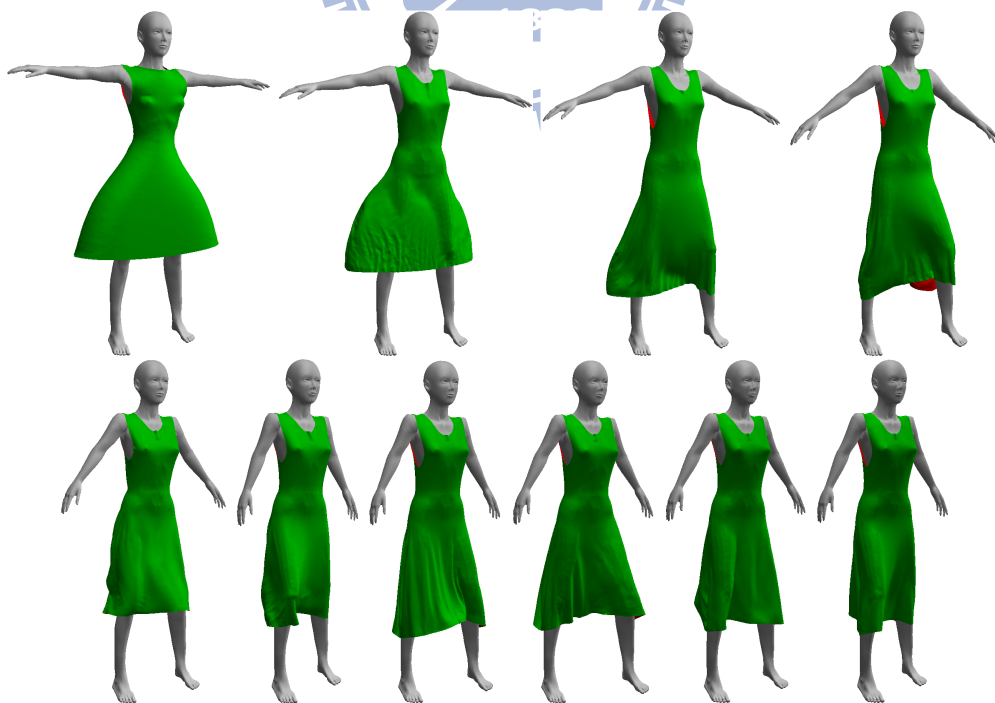


Figure 6.4: A series of snapshots of Ani. four.

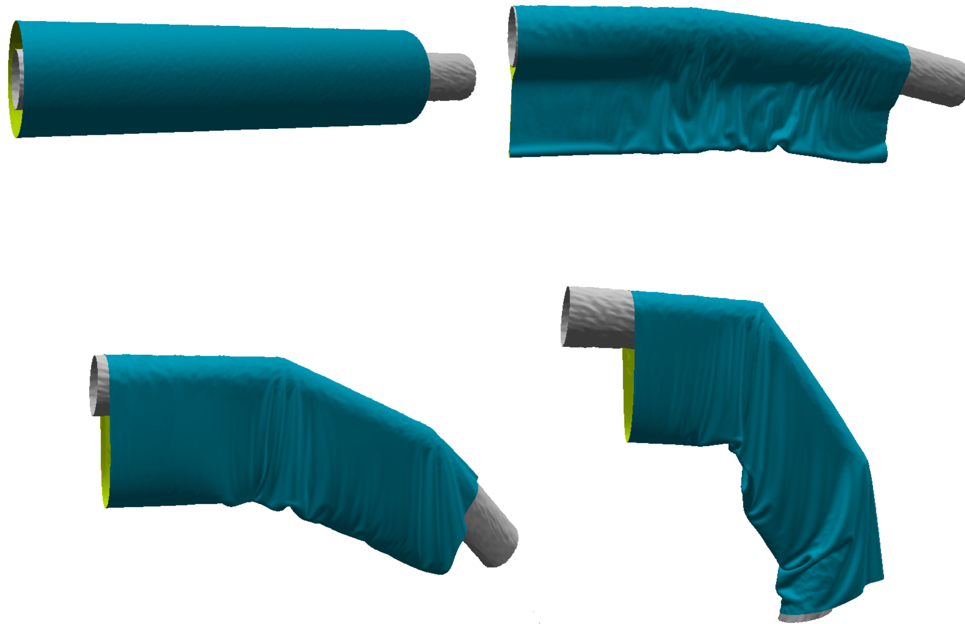


Figure 6.5: A series of snapshots of Ani. five.

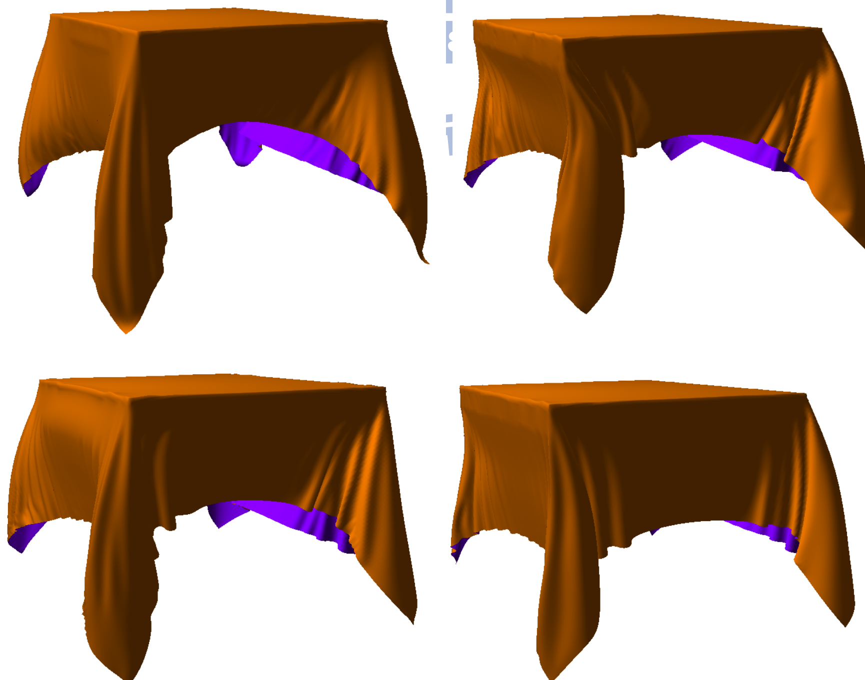


Figure 6.6: A series of snapshots of Ani. six.

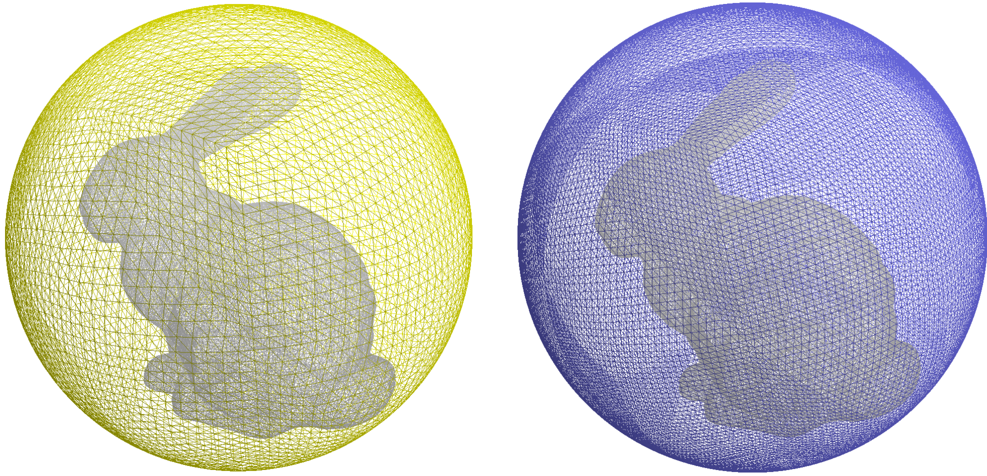


Figure 6.7: The snapshots of Ani. one and two in wireframe.

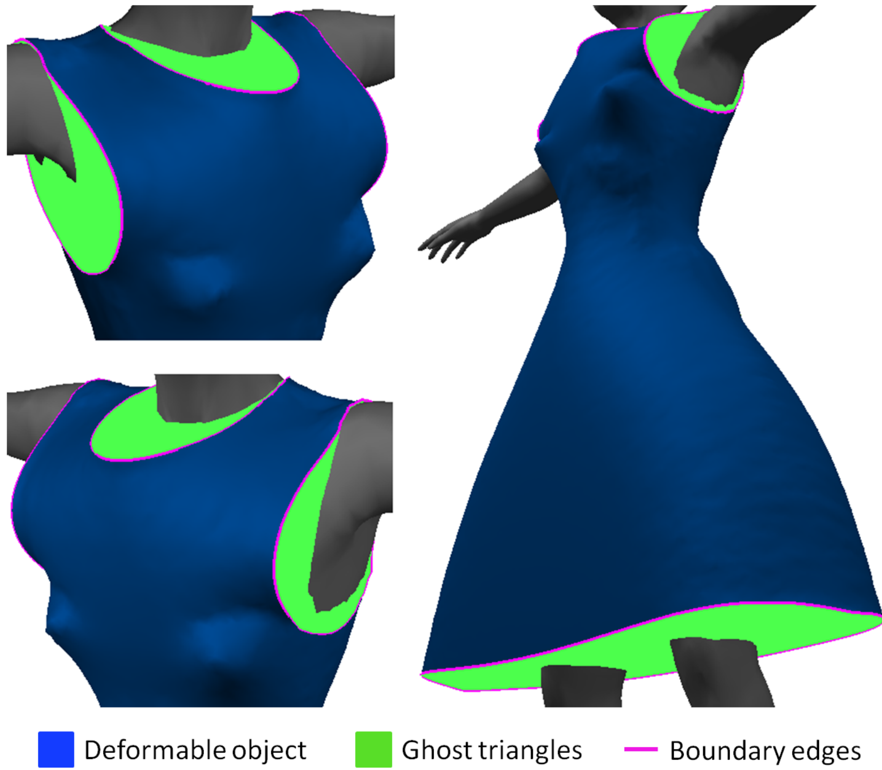


Figure 6.8: The snapshots of Ani. three with ghost triangles.

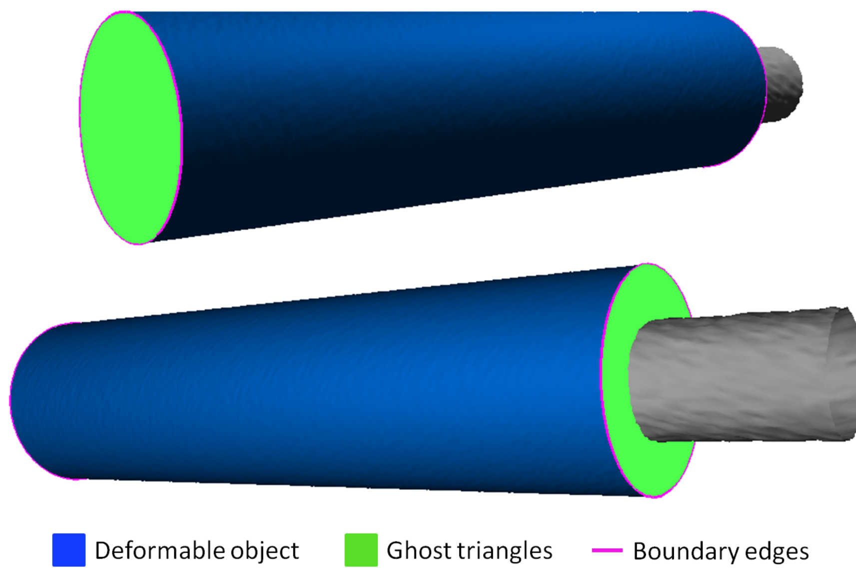


Figure 6.9: The snapshots of Ani. five with ghost triangles.

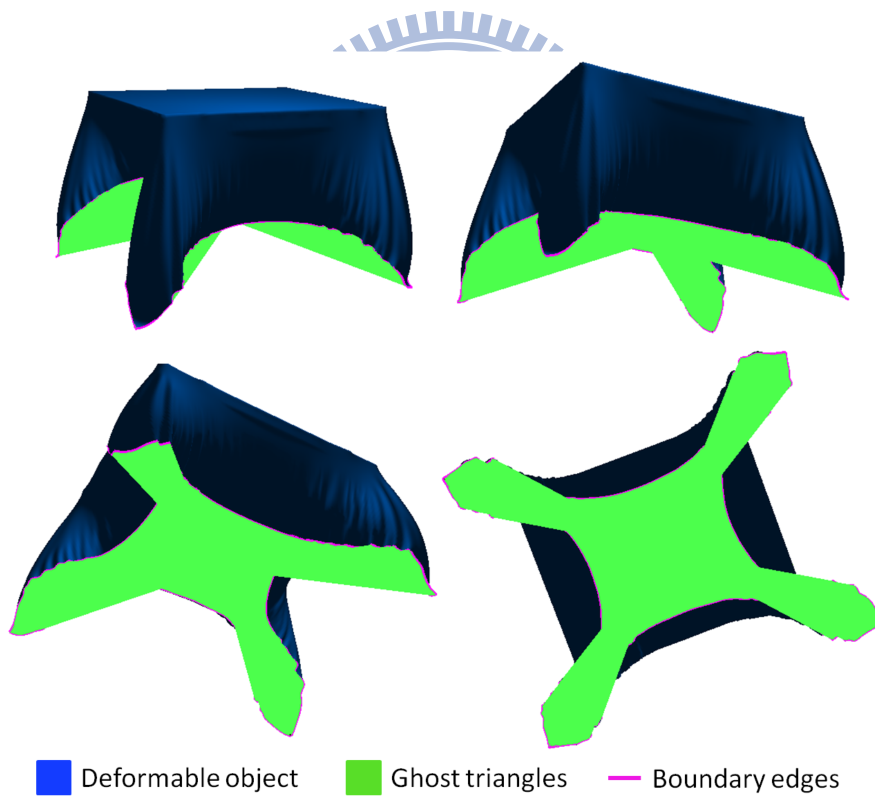


Figure 6.10: The snapshots of Ani. six with ghost triangles.

The deformable objects in Ani. one and two are closed, as shown is Figure 6.7. As mentioned above, Ani. one and two are similar, but the number of triangles of the cloth in Ani. two is more than Ani. one. The deformable objects in Ani. three, four, five, and six are unclosed. Figure 6.8, 6.9 and 6.10 show the snapshots of Ani. three, five, and six that the deformable objects are indicated by blue color, and the ghost triangles are indicated by green color. The information of ghost triangles in Ani. three and four are the same.

6.2 Results on CPUs

Table 6.2 and 6.3 show the execution time of each step with the view-point and view-line schemes on CPUs for continuous self-collision detection. Table 6.4 shows timing comparisons between our approach, AABB, k-DOP, and ICCD for continuous self-collision detection. For the view-point scheme, the speedup factors are 6.1x $\sim$ 27.7x, 2.3x $\sim$ 4.4x, and 2x $\sim$ 3.7x compared to AABB, K-DOP, and ICCD. For the view-line scheme, the speedup factors are 6.5x $\sim$ 28.6x, 2.2x $\sim$ 4.2x, and 2x $\sim$ 3.5x compared to AABB, K-DOP, and ICCD. We can observe that the performance of the view-line scheme is a little better than the view-point scheme on average.

Figure 6.11 and 6.12 show the numbers of triangles of all kinds of view sets with the view-point scheme and the view-line scheme for the six benchmarks.

There are several factors that make impacts on the performance, such as the types of simulation objects, the number of holes, the number of ghost triangles, and the deformation of objects.

- For closed meshes, boundary handling does not need to be performed, and there are no violated triangles. So, the number of triangles that needs to be handled in the step of traversal is less. Thus, the performance of handling closed meshes is better than unclosed meshes on average. For example, the performance of handling Ani. one and two is better.

- There are more ghost triangles if the number of holes for deformable objects is more. And the primary factor that makes a great impact on the performance is the

number of ghost triangles. From Table 6.1, we can observe that the number of holes in Ani. three, four, and five are larger than the number of holes in Ani. six. But the number of ghost triangles in Ani. six is more than other animation benchmarks. Consequently, the performance of handling Ani. six is improved less by our view-based approach than others.

- From Figure 6.3, we can observe that the deformable object deforms intensely. The number of negatively oriented and violated triangles is large. The performance of handling Ani. three is improved least by our view-based approach.

- Among the four benchmarks with unclosed meshes, Ani. five is improved most by our view-based approach. From Table 6.1, we observe that the number of ghost triangles is least so as to result in low cost of extracting violated triangles in the step of boundary handling. Besides, from Figure 6.5, we can observe that the boundaries of the deformable object slide and deform slightly, so that the number of violated triangles is few.

- The execution time of boundary handling is proportional to the number of triangles, the number of ghost triangles, and the deformation of the deformable object. From Table 6.2 and 6.3, we can observe that the execution time of boundary handling is longer if the number of ghost triangles is larger. On the other hand, the numbers of triangles in Ani. three and four are the same. But the execution time of boundary handling for Ani. three is longer than Ani. four. From Figure 6.3, we can observe that the deformation of the deformable object is severer in Ani. three.

- We can observe that the performance of performing view tests is quite different with the view-point scheme and the view-line scheme. The view-point scheme is simple and the step of vertex region determination is not required, so the performance is better. But the performance of performing traversal with the view-point scheme is worse because the number of negatively oriented triangles is larger.

Ani.	View Tests	Boundary Handling	vBVH Update	Traversal	Elementary Tests	Total
One	0.98	-	<0.01	0.83	<0.01	1.88
Two	4.17	-	<0.01	4.37	0.73	9.27
Three	4	4.26	0.16	8.4	0.67	17.49
Four	4	2.28	0.19	6.43	0.13	13.03
Five	5.72	0.53	0.13	6.13	0.62	13.13
Six	5.94	7.1	<0.01	6.56	<0.01	19.6

Table 6.2: Execution time (in ms) of each step with the view-point scheme on CPUs for continuous self-collision detection.

Ani.	View Tests	Boundary Handling	vBVH Update	Traversal	Elementary Tests	Total
One	1.18	-	<0.01	0.83	<0.01	2.01
Two	4.65	-	<0.01	4.38	0.75	9.78
Three	4.75	4.24	0.11	6.6	0.7	16.4
Four	4.75	2.22	0.11	4.72	0.13	11.93
Five	6.77	0.54	0.11	5.15	0.14	12.71
Six	7.13	7.37	<0.01	6.14	<0.01	20.64

Table 6.3: Execution time (in ms) of each step with the view-line scheme on CPUs for continuous self-collision detection.

Ani.	AABB	16-DOP	ICCD	View-point	View-line	Speed-up(point)	Speed-up(line)
One	21.84	7.13	7	1.88	2.01	11.6x,3.8x,3.7x	10.8x,3.5x,3.5x
Two	131.78	41.03	30.88	9.27	9.78	14.2x,4.4x,3.3x	13.5x,4.2,3.2x
Three	106.25	40.04	35.2	17.49	16.4	6.1x,2.3x,2x	6.5x,2.4x,2.1x
Four	97.24	31.48	31.9	13.03	11.93	7.5x,2.4x,2.4x	8.2x,2.6x,2.7x
Five	364.04	45.54	42.2	13.13	12.71	27.7x,3.5x,3.2x	28.6x,3.6x,3.3x
Six	343.27	44.62	42.15	19.6	20.64	17.5x,2.3x,2.2x	16.6x,2.2x,2x

Table 6.4: Timing comparisons (in ms) between our view-based approach, AABB, 16-DOP, and ICCD for continuous self-collision detection.

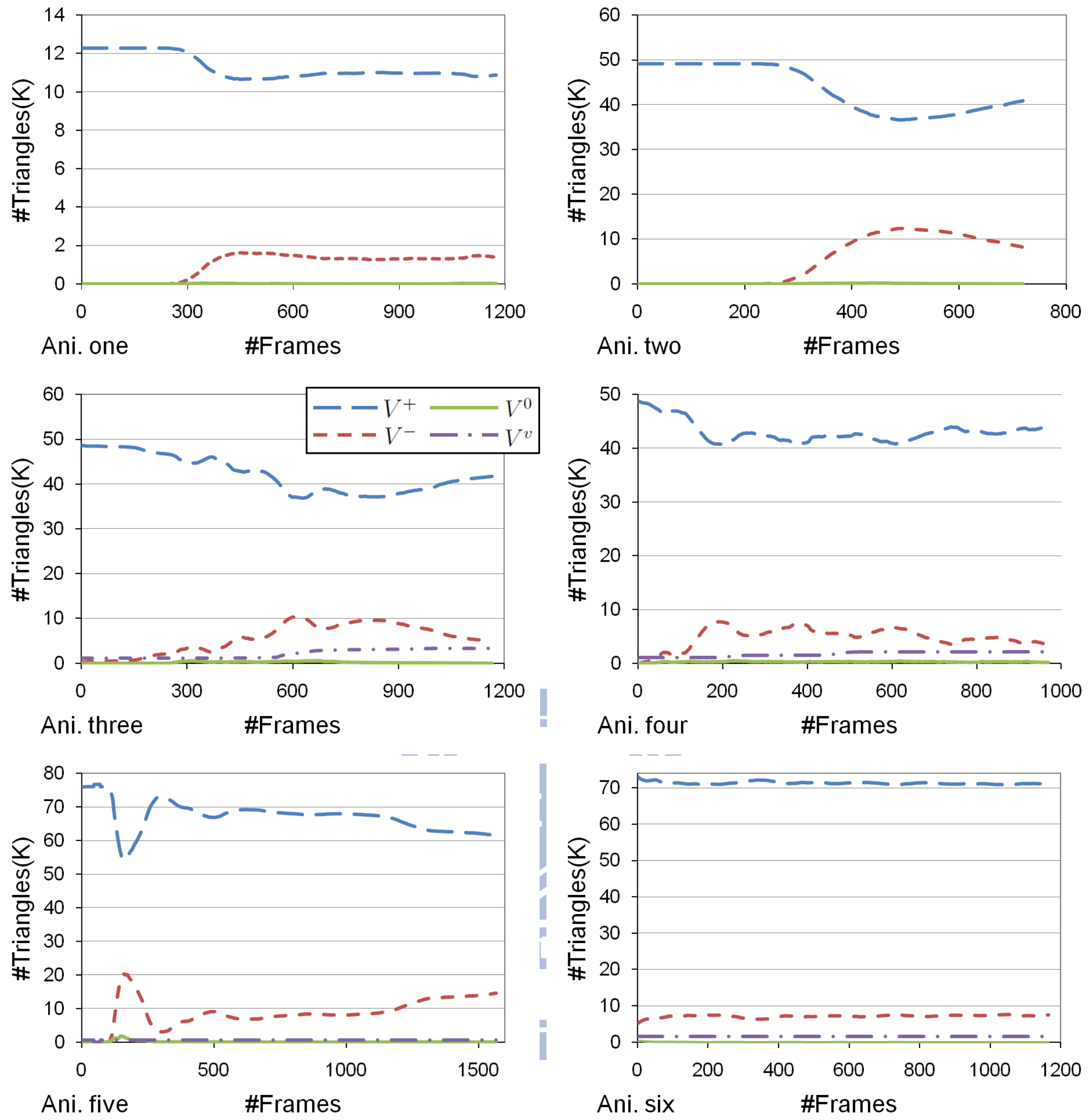


Figure 6.11: The numbers of triangles of all kinds of view sets with the view-point scheme for the six benchmarks.

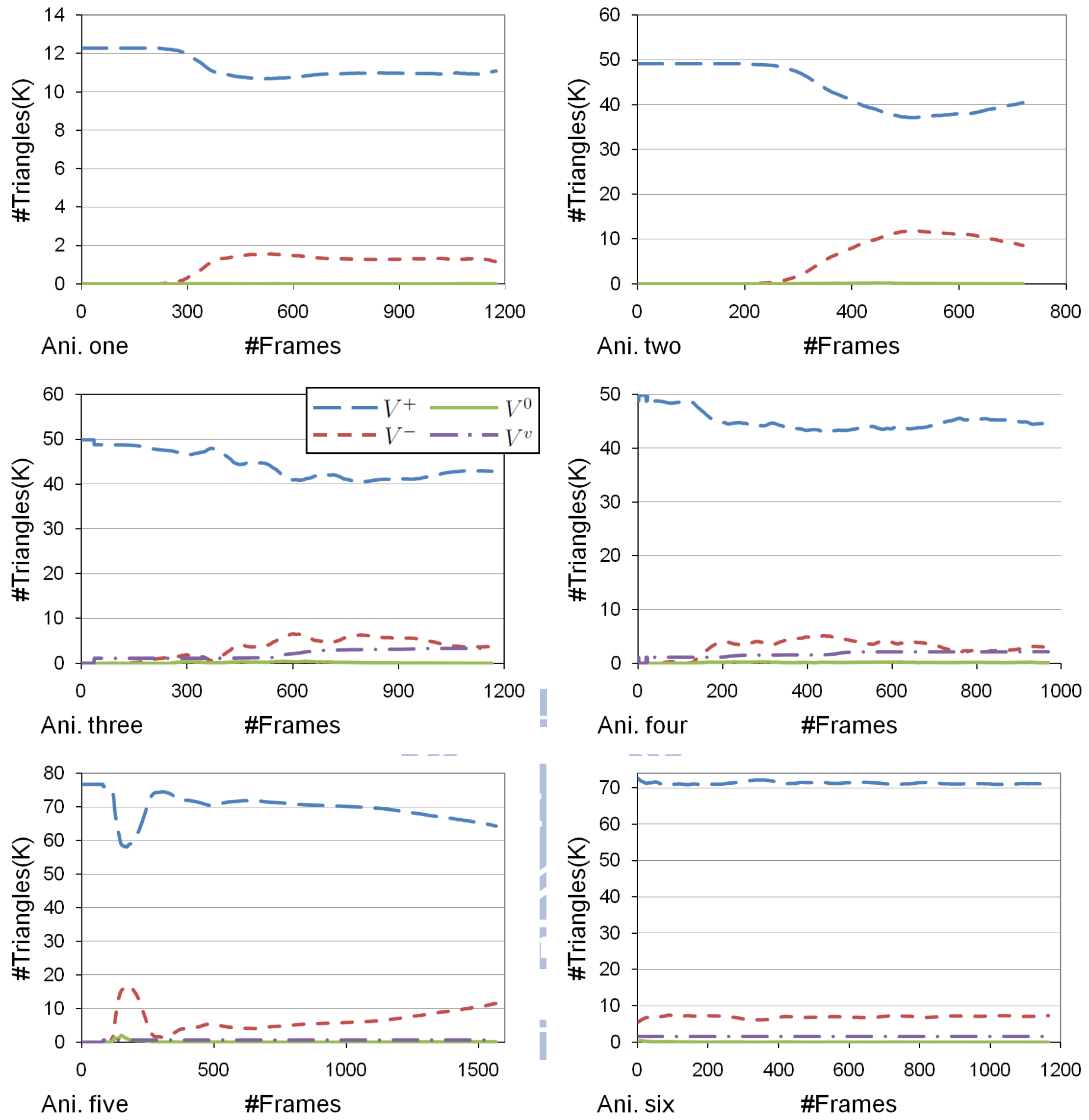


Figure 6.12: The numbers of triangles of all kinds of view sets with the view-line scheme for the six benchmarks.

6.3 Results on GPUs

Table 6.5 shows the execution time of performing traversal in the beginning to obtain the initial history nodes. We release the data of history nodes for every few frames and restart to perform traversal from the initial traversal results. Table 6.6 and 6.7 show the execution time of each step with the view-point and view-line scheme on GPUs for continuous self-collision detection. The elements of others contain operations of compression, parallel scan, and update of the triangle type list. Table 6.8 and 6.9 show the speed-up factors of each step with the view-based approach using CPUs and GPUs for continuous self-collision detection.

In the step of traversal, history nodes are adopted to improve the performance. As mentioned previously, there are two conditions that history nodes should be released, and traversal is performed from the initial traversal history nodes in the next frame. When the number of history nodes is too large over than a threshold, we should release the history nodes. By experiments, the threshold is defined to be 128. On the other hand, we should release the history nodes for every 25 frames.

Ani.	one	two	three	four	five	six
time (ms)	3.2	12.4	15.35	15.34	22.15	21.09

Table 6.5: Execution time (in ms) of performing traversal in the beginning to obtain the initial history nodes.

Ani.	View Tests	Boundary Handling	Traversal	Elementary Tests	Others	Total
One	0.03	-	0.2	0.04	0.01	0.28
Two	0.09	-	0.63	0.11	0.02	0.83
Three	0.12	0.29	1.01	0.12	0.03	1.57
Four	0.12	0.27	0.79	0.04	0.03	1.25
Five	0.07	0.22	0.76	0.04	0.05	1.14
Six	0.08	0.61	0.7	0.02	0.05	1.46

Table 6.6: Execution time (in ms) of each step with the view-point scheme on GPUs for continuous self-collision detection.

Ani.	View Tests	Boundary Handling	Traversal	Elementary Tests	Others	Total
One	0.03	-	0.19	0.04	0.01	0.27
Two	0.08	-	0.64	0.11	0.02	0.85
Three	0.13	0.29	0.88	0.12	0.03	1.45
Four	0.13	0.27	0.62	0.04	0.04	1.1
Five	0.07	0.21	0.6	0.04	0.06	0.98
Six	0.09	0.6	0.76	0.05	<0.01	1.5

Table 6.7: Execution time (in ms) of each step with the view-line scheme on GPUs for continuous self-collision detection.

Ani.	View Tests	Boundary Handling	Traversal	Elementary Tests	Total
One	32.7x	-	4.2x	-	6.7x
Two	46.3x	-	6.9x	6.8x	11.2x
Three	33.3x	14.7x	8.3x	5.8x	11.1x
Four	33.5x	8.4x	8.1x	3.3x	10.4x
Five	81.7x	2.4x	8.1x	3.5x	11.5x
Six	74.3x	11.6x	9.4x	-	13.4x

Table 6.8: Speed-up factors of each step with the view-point scheme using CPUs and GPUs.

Ani.	View Tests	Boundary Handling	Traversal	Elementary Tests	Total
One	39.3x	-	4.4x	-	7.4x
Two	58.1x	-	6.8x	6.8x	11.5x
Three	36.5x	14.6x	7.5x	5.8x	11.3x
Four	36.5x	8.2x	7.6x	1.9x	10.8x
Five	96.7x	2.6x	8.6x	2.8x	13x
Six	79.2x	12.3x	8.1x	-	13.8x

Table 6.9: Speed-up factors of each step with the view-line scheme using CPUs and GPUs.

6.4 Differences of Each Step on CPUs and on GPUs

The procedures of our view-based approach are divided into four steps, including view tests, boundary handling (if the deformable object is unclosed), traversal, and elementary tests. And we discuss the differences between implementing on CPUs and on GPUs for the four steps.

View tests

We determine the types of all triangles of the deformable object by performing view tests. And we divide view tests into two steps, including vertex region determination and triangle type determination. The implementation of view tests on CPUs and on GPUs are the same. But we take advantage of more processors on GPUs to improve the performance of performing view tests.

On CPUs, we use one thread to perform the two steps of view tests sequentially. In the step of vertex region determination, the process is performed sequentially according to each vertex. After determining the regions of all vertices, we compute the types of all triangles according to each triangle sequentially in the step of triangle type determination.

On GPUs, we create nv threads in the step of vertex region determination and nt threads in the step of triangle type determination, where nv is the number of all vertices, and nt is the number of all triangles of the deformable object. So, the computation of a vertex or a triangle is handled by a thread, and the processes are executed in parallel. The computation of view tests is the simplest in the view-based approach.

Boundary handling

For unclosed meshes, we perform boundary handling to extract violated triangles each frame. We perform traversal for the BVHs of ghost triangles and the deformable object.

On CPUs, we perform traversal sequentially according to the BVHs of all holes in the deformable object. In other words, performing traversal in the step of boundary handling on CPUs is BVH-based. If a triangle collides with ghost triangles, then the triangle is

determined to be violated. We keep the types of all violated triangles until the deformable object is self-collision free. Note that we compute exactly whether or not collision occurs between the deformable object and ghost triangles.

On GPUs, we also perform boundary handling sequentially according to the BVHs of all holes. But nt threads are created, where nt is the number of triangles of the deformable object. Each thread is responsible for the computation between a triangle and the BVHs of ghost triangles. So, the degree of parallelism is higher. In other words, performing traversal in the step of boundary handling on GPUs is triangle-based. On the other hand, when traversal is performed, we collect the potentially colliding pairs, but elementary tests are not performed for these potentially colliding pairs. We determine whether or not two triangles collide with each other by the bounding boxes of the two triangles and determine the type of the triangle according the result. If we perform elementary tests exactly for the potentially colliding pairs of the deformable object and ghost triangles, the computation is more complicated in the kernel function, and there are more global memory accesses.

Table 6.10 and 6.11 show the results on CPUs and on GPUs. On CPUs, it is better to perform boundary handling with exact collision detection for Ani. three, four, and five so as to reduce the number of violated triangles and the computation time of performing traversal. But for Ani. six, the hole of the deformable object is huge, and the computation of boundary handling can be reduced a lot with inexact collision detection. So, the performance of whole procedure for Ani. six is better with inexact collision detection. On GPUs, we can observe that if we compute collision detection exactly in the step of boundary handling, the number of violated triangles is smaller, and the execution time of performing traversal is shorter, but the execution time of boundary handling increases significantly. Therefore, the performance is reduced for the whole procedure.

Traversal

We perform traversal to collect potentially colliding pairs. The situation is similar to the step of boundary handling.

On CPUs, after performing view tests and boundary handling, we get three kinds of

1896

Ani.	Exactness				Inexactness			
	Boundary handling	#Violated triangles	Traversal	Total	Boundary handling	#Violated triangles	Traversal	Total
Three	4.24	2032	6.6	16.41	2.22	5548	10.9	19.26
Four	2.22	1690	4.72	11.93	1.45	3898	8.32	15.55
Five	0.54	622	5.15	12.71	0.39	1367	8.21	14.71
Six	7.37	1593	6.14	20.64	3.56	3116	8.01	18.75

Table 6.10: Boundary handling with the view-line scheme by exact and inexact methods on CPUs.

Ani.	Exactness			Inexactness		
	Boundary handling	Traversal	Total	Boundary handling	Traversal	Total
Three	1.43	0.74	2.43	0.29	0.86	1.47
Four	0.88	0.53	1.64	0.27	0.62	1.1
Five	1.08	0.51	1.73	0.21	0.6	0.98
Six	0.91	0.71	1.78	0.6	0.76	1.5

Table 6.11: Boundary handling with the view-line scheme by exact and inexact methods on GPUs.

view sets for closed meshes and four kinds of view sets for unclosed meshes. Then, we build vBVHs according to all kinds of view sets. After that, we perform traversal for the vBVHs sequentially. In other words, performing traversal on CPUs is BVH-based.

On GPUs, we get a set of triangles which are negatively oriented or violated after performing view tests and boundary handling. Traversal is performed for negatively oriented and violated triangles. nnv threads are created, where nnv is the number of triangles, which are negatively oriented or violated. Thus, the computation of performing traversal of a triangle is handled by a thread. Note that we do not need to build vBVHs for all kinds of view sets. We perform traversal between the negatively oriented and violated triangles and the BVH of the deformable object. In other words, performing traversal on GPUs is triangle-based. Besides, we employ the front-based method presented by Tang et al. [TMT09] to improve the performance of performing traversal, but we do not construct the bounding volume test tree. We record a list of nodes for each triangle every frame that traversal is terminated at these nodes. So, we do not need to perform traversal from the root of the BVH in the next frame but start from the history nodes. In short, our approach

is triangle-based, and the degree of parallelism is high.

Table 6.12 shows the execution time of performing traversal on GPUs with the view-line scheme for three different policies. For the first policy, we perform traversal without using history nodes. For the second policy, we employ history nodes to improve performing traversal. After releasing the history nodes, traversal is performed and started from the root node of the BVH. For the last policy, traversal is performed and started from the initial traversal history nodes when the history nodes are released. We can observe that the performance of the third policy is the best, and the performance of the first policy is the worst because the cost of traversing for the BVH from the root each frame is too expensive.

Ani.	without using history nodes	using history nodes	
		restart from root	restart from initial history nodes
One	0.66	0.23	0.19
Two	1.93	0.72	0.64
Three	3.05	1.05	0.88
Four	2.25	0.77	0.62
Five	2.31	0.73	0.6
Six	2.85	0.86	0.76

Table 6.12: Execution time (in ms) of performing traversal on GPUs with the view-line scheme for three different policies.

Elementary tests

After collecting potentially colliding pairs, we perform elementary tests for these potentially colliding pairs. We check the collision of two triangles in a colliding pair. The implementations of elementary tests on CPUs and on GPUs are the same. But we take advantage of more processors on GPUs to improve the performance of performing elementary tests.

On CPUs, the process is performed sequentially by one thread for all potentially colliding pairs. On GPUs, np threads are created, where np is the number of potentially colliding pairs. So, the computation of a colliding pair is handled by a thread, and np

threads are executed in parallel.

6.5 Vertex Movement within a Frame

Table 6.13 shows the number of vertices within a frame according to their movement on average. We can observe that most of vertices move in the same region within a frame. The number of vertices, which move across two regions, is less than 30 for all animation benchmarks. Therefore, in the step of view tests, we can choose any vertex of a triangle to perform view tests with the view-line scheme, and the number of performing view tests of a triangle is nearly one on average.

Region	R_0	R_1	R_2	R_0<->R_2	R_1<->R_2
Ani. one	684	1995	3459	2	3
Ani. two	2553	8677	13317	11	18
Ani. three	3010	6929	15269	<1	17
Ani. four	10423	6434	8352	11	5
Ani. five	5445	16616	16489	4	5
Ani. six	12805	12898	14691	3	2

Table 6.13: The number of vertices within a frame on average according to their movement.

6.6 Improvement with Triangle Clusters

Table 6.14 shows the number of vertices and clusters of the deformable objects. The number of clusters is about one-third of the number of vertices on average. Table 6.15 shows the execution time of performing view tests in the view-line scheme with triangle clusters. We can observe that the performance is improved a little with triangle clusters.

6.7 vBVHs Construction

On CPUs, we build vBVHs based on all kinds of view sets each frame. In fact, we neither reconstruct the BVHs of all view sets nor remark all the nodes.

Ani.	one	two	three	four	five	six
#Vertex	26148	44580	42330	42330	48240	73125
#Cluster	3256	13048	10970	10970	19200	20100

Table 6.14: The numbers of vertices and clusters of the deformable objects in the six benchmarks.

Ani.	one	two	three	four	five	six
without triangle clusters	1.18	4.65	4.76	4.75	6.77	7.13
with triangle clusters	1.11	4.36	4.43	4.45	6.17	6.33

Table 6.15: Execution time (in ms) of performing view tests in the view-line scheme with triangle clusters.

Suppose that we mark the nodes of the BVH of the deformable object with P, N, Z, and V, where P for the view set V^+, N for the view set V^-, Z for the view set V^0, and V for the view set V^v.

At first, all of the nodes are marked with P in the preprocessing stage because the number of triangles in the view set V^+ is the most. Table 6.16 shows the cost of marking all of the nodes with P. Besides, there is a little number of triangles whose type is changed between two consecutive frames, as shown in Table 6.17. We just need to update the vBVHs according to the triangles whose types are different from itself between two consecutive frames. The computation is low, and the cost is less than $0.2ms$. Note that we keep the type of violated triangles until the deformable object is self-collision free.

Ani.	One	Two	Three	Four	Five	Six
time (ms)	0.88	4.2	5.71	5.78	7.71	7.65

Table 6.16: The cost of marking all of the nodes in the preprocessing stage.

6.8 Discussion

6.8.1 Closed and unclosed meshes

The differences between closed and unclosed meshes with the view-based approach for continuous self-collision detection are described as follows.

Ani.	to V_l^+	to V_l^-	to V_l^0	to V_l^v	V_l^+ to others	V_l^- to others	V_l^0 to others	Total
One	3	4	7	0	4	3	7	28
Two	28	40	67	0	40	28	67	270
Three	72	74	147	78	72	147	3	593
Four	77	78	146	4	84	75	146	610
Five	61	67	128	<1	69	60	128	513
Six	35	42	70	1	43	35	70	296

Table 6.17: The numbers of triangles whose types are changed between two consecutive frames on average with the view-line scheme.

1. For unclosed meshes, we generate ghost triangles and construct the BVHs in the preprocessing stage.

2. We perform boundary handling for unclosed meshes to extract violated triangles. Collision detection is performed for the BVHs of deformable objects and ghost triangles.

3. The number of triangles, which are handled in the step of traversal, for unclosed meshes is larger than the number of triangles for closed meshes because there are violated triangles additionally for unclosed meshes. Actually, if deformable objects do not deform severely or holes in the object is small, then the number of violated triangles is few, and the performance is better, such as Ani. five.

6.8.2 Ghost triangles with or without ghost vertices

Our algorithm is suitable for closed and unclosed triangle meshes. We add some ghost triangles for unclosed triangle meshes to fill the holes. As mentioned above, there are two ways to generate ghost triangles. The first way is to connect the vertices that are around the holes. The second way is to add ghost vertices, and then connect the boundary vertices with the ghost vertices. In practice, we implement the second way to generate ghost triangles. There are two reasons. First, it is simple to determine the order of all vertices, and we can generate ghost triangles according to the boundary vertices orderly. So it is easy to implement. Second, the ghost triangles created by the first way are irregular,

and the performance of performing traversal for the BVH is poorer. The spatial locality of the BVH of ghost triangles with ghost vertices is better. Table 6.18 shows the execution time of boundary handling that the ghost triangles are constructed with and without ghost vertices. For CPU, boundary handling is performed by exact collision detection. For GPU, boundary handling is performed by inexact collision detection.

Ani.	CPU		GPU	
	with	without	with	without
Three	4.24	28.3	0.29	0.51
Four	2.22	14.7	0.27	0.6
Five	0.54	2.92	0.21	0.23
Six	7.37	16.8	0.6	1.12

Table 6.18: Execution time (in ms) of boundary handling that the ghost triangles are constructed with and without ghost vertices.

6.8.3 Comparison between the view-point and view-line scheme

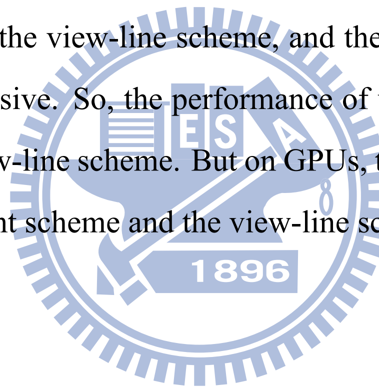
In general, the view-line scheme is better than the view-point scheme. There are two differences between the view-point scheme and the view-line scheme.

1. The cost of computation with the view-line scheme in the step of view tests is more expensive than the view-point scheme. For the view-point scheme, we just need to compute the start position and terminal position of all vertices within a time interval $[0, \Delta t]$. And then we can get the linear information for the movement trajectories of all vertices. For the view-line scheme, we divide the space and the deformable object into three regions. We determine the regions of all vertices and compute the related linear information of movement trajectories of all vertices based on the related regions. If a vertex moves across multiple regions in $[0, \Delta t]$, then view tests are performed for multiple times.

2. In general, the number of negatively oriented triangles with the view-line scheme is less than the view-point scheme. So, the execution time of performing traversal

is faster with the view-line scheme. Actually, we cannot predict the results. We should choose a better policy according to the deformation of the object and the situation in the previous frame. Figure 6.13 shows the comparisons of the number of negatively oriented triangles between the view-point scheme and the view-line scheme. In this case, we do not consider the violated triangles. In other words, we record the number of negatively oriented triangles and do not subtract the violated triangles.

From Table 6.2, 6.3, 6.6, and 6.7, we can observe that the types of view primitives make a big impact on performance of performing view tests on CPUs. For the view-point scheme, each vertex belongs only to one region, and the computation of vertex region determination is lower. However, a chosen vertex of a triangle may move across multiple regions within a frame with the view-line scheme, and the computation of vertex region determination is more expensive. So, the performance of view tests with the view-point scheme is faster than the view-line scheme. But on GPUs, the performance of performing view tests with the view-point scheme and the view-line scheme is similar.



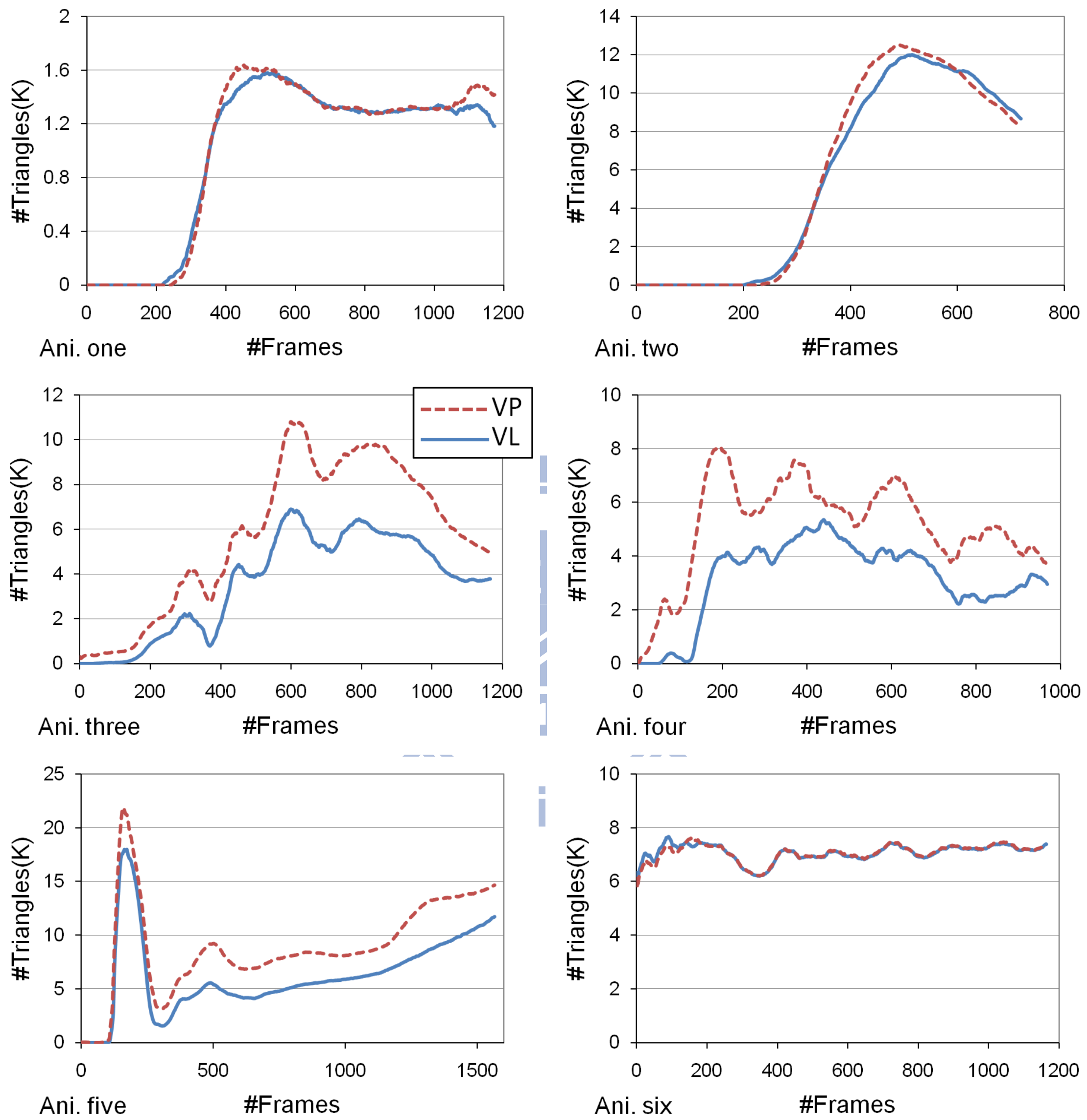


Figure 6.13: Comparisons of the number of negatively oriented triangles without considering violated triangles between the view-point scheme and the view-line scheme.

Chapter 7

Conclusion and Future Work

We give our conclusion and future work in this section.

7.1 Conclusion

We propose a novel view-based approach for continuous self-collision detection with deformable manifold triangle meshes. The view-based approach is suitable for closed triangle meshes. For unclosed triangle meshes, we generate ghost triangles in the preprocessing stage to fill the holes and enclose the triangle meshes. We check the orientation of all triangles of the deformable object based on the view primitives by performing view tests to confirm the object is self-collision free. There are two types of view primitives, including a point and a line segment. The view primitives are put inside the deformable objects in the beginning, and we assume that the view primitives do not penetrate the deformable object during the simulation.

On CPUs, if the object is determined to be not self-collision free after performing view tests, the vBVHs are constructed for all the view sets. Performing traversal for the vBVHs is efficient to collect potentially colliding pairs. Note that performing traversal on CPUs is BVH-based. Finally, we perform elementary tests to check the potentially colliding pairs so as to find the actual colliding triangle pairs.

On GPUs, if the object is not self-collision free, we collect a set of triangles which are

negatively oriented and violated. Traversal is performed for the BVH of the deformable object and a triangle, and totally nnv threads are executed in parallel, where nnv is the number of negatively oriented and violated triangles. Performing traversal on GPUs is triangle-based. Finally, we perform elementary tests to check the potentially colliding pairs by np threads executed in parallel so as to find the actual colliding triangle pairs, where np is the number of potentially colliding pairs.

The experimental results show that our algorithm is more efficient than other methods for continuous self-collision detection on CPUs. Besides, performing the view-based approach on GPUs is more efficient than performing on CPUs. There are some procedures of the view-based approach that are different on CPUs and on GPUs. Because that the degree of parallelism is higher on GPUs, and we can execute the process further in parallel.

7.2 Future Work

If the deformable object deforms severely, and the rigid object moves every frame, we should compute view primitives every frame. We can employ GPUs to compute view primitives.

Besides, we can observe that the influence of violated triangles for performing traversal is great. The types of violated triangles are kept until the object is self-collision free. But the boundaries of the meshes may sometimes roll in the beginning and lay down later, and the violated triangles become normal, as shown in Figure 7.1. The black edges pass through the ghost edge again and become normal. But the curve is still not self-collision free according to the results of performing view tests. So the black edges are still assigned to the violated view set. In fact, when the boundaries of the meshes are no longer rolled, the view set of violated triangles should be released even though the object is not self-collision free. Then, we can reduce the cost of performing traversal for unclosed triangle meshes.

We want to handle skeleton deformable objects with the view-based approach. For

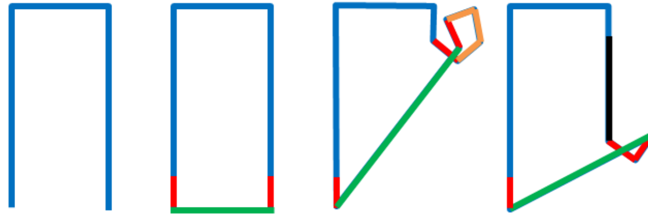


Figure 7.1: Boundary edges roll and lay down.

skeleton deformable objects, triangle orientation cannot be determined appropriately according to a single view primitive. We should divide the objects and generate related view primitives for each part. For example, a character can be divided into five parts mainly, including four limbs and the body. We can generate five view primitives and perform view tests separately.

Finally, we want to extend the view-based approach to handle continuous inter-collision detection. For some kinds of models, the view-based approach is also suitable. For example, if there are two objects. One of them lies in the other one, and a view primitive is put inside the interior object. During the simulation, these two object do not penetrate each other, and then we can observe that inter-collision should occur at two triangles with the same orientation based on the view primitives. Finally, we will implement the whole procedure of collision detection, including BVHs update, on GPUs.

Bibliography

[AFC$^+$10] J. Allard, F. Faure, H. Courtecuisse, F. Falipou, C. Duriez, and P.G. Kry. Volume contact constraints at arbitrary resolution. *ACM Transactions on Graphics (TOG)*, 29(4):82, 2010.

[BT95] S. Bandi and D. Thalmann. An adaptive spatial subdivision of the object space for fast collision detection of animated rigid bodies. In *Computer Graphics Forum*, volume 14, pages 259--270, 1995.

[BW02] G. Baciu and W.S.-K. Wong. Hardware-assisted selfcollision for deformable surfaces. In *ACM Symposium on Virtual Reality Software and Technology*, pages 129--136, 2002.

[CTM08] S. Curtis, R. Tamstorf, and D. Manocha. Fast collision detection for deformable models using representative-triangles. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 61--69, 2008.

[Ebe06] D.H. Eberly. Eigensystems for 3×3 symmetric matrices (revisited). Technical report, Geometric Tools Inc, 2006.

[GKJ$^+$05] N.K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M.C. Lin, and D. Manocha. Interactive collision detection between deformable models using chromatic decomposition. *ACM Transactions on Graphics*, 24(3): 991--999, 2005.

[GLM96] S. Gottschalk, MC Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171--180, 1996.

[GRLM03] N. Govindaraju, S. Redon, M. Lin, and D. Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25--32, 2003.

[HOS+07] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. CUDPP homepage, 2007.

[JP04] D.L. James and D.K. Pai. Bd-tree: output-sensitive collision detection for reduced deformable models. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 393--398, 2004.

[KHH+09] D. Kim, J.P. Heo, J. Huh, J. Kim, and S. Yoon. HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs. *Proceedings of the Korea Advanced Institute of Science and Techaology*, 28(7):1791--1800, 2009.

[KHM+98] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE transactions on Visualization and Computer Graphics*, 4(1):21--36, 1998.

[KP03] D. Knott and K.D. Pai. Cinder: Collision and interference detection in real-time using graphics hardware. In *Graph. Interface*, 2003.

[LHLK10] F. Liu, T. Harada, Y. Lee, and Y.J. Kim. Real-time collision culling of a million bodies on graphics processing units. In *ACM Transactions on Graphics (TOG)*, volume 29, page 154. ACM, 2010.

[LMM10] C. Lauterbach, Q. Mo, and D. Manocha. gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. In *Computer Graphics Forum*, volume 29, pages 419--428, 2010.

[Mel00] S. Melax. Dynamic plane shifting bsp traversal. In *Graphics interface*, pages 213--220, 2000.

[MKE03] J. Mezger, S. Kimmerle, and O. Etzmuss. Hierarchical Techniques in Collision Detection for Cloth Animation. *Journal of WSCG*, 11(2):322--329, 2003.

[NVI] NVIDIA Corporation. GPU computing SDK.

[NVI09] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Fermi. 2009.

[NVI10a] NVIDIA Corporation. CUDA C best practices guide version 3.2. 2010.

[NVI10b] NVIDIA Corporation. NVIDIA CUDA C programming guide version 3.1.1. 2010.

[PG95] I.J. Palmer and R.L. Grimsdale. Collision Detection for Animation using Sphere-Trees. In *Computer Graphics Forum*, volume 14, pages 105--116, 1995.

[PM11] J. Pan and D. Manocha. GPU-based Parallel Collision Detection for Real-Time Motion Planning. *Algorithmic Foundations of Robotics IX*, pages 211--228, 2011.

[Pro97] X. Provot. Collision and Self-collision Handling in Cloth Model Dedicated to Design Garments. In *Graphics Interface*, pages 177--189, 1997.

[Smi61] Oliver K. Smith. Eigenvalues of a symmetric 3×3 matrix. *Communications of the ACM*, 4(4):168, 1961.

[TCYM09] Min Tang, Sean Curtis, Sung-Eui Yoon, and Dinesh Manocha. ICCD: Interactive Continuous Collision Detection between Deformable Models Using Connectivity-Based Culling. *IEEE Transactions on Visualization and Computer Graphics*, 15(4):544--557, 2009.

[TKH+05] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, et al. Collision detection for deformable objects. In *Computer Graphics Forum*, pages 61--81, 2005.

[TMLT11] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. Collision-streams: Fast GPU-based collision detection for deformable models. In *I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 63--70, 2011.

[TMT09] M. Tang, D. Manocha, and R. Tong. Multi-core Collision Detection between Deformable Models. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pages 355--360, 2009.

[vdB99] G. van den Bergen. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics, GPU, and Game tools*, 2(4): 1--14, 1999.

[VMT94] P. Volino and N. Magnenat-Thalmann. Efficient Self-collision Detection on Smoothly Discretized Surface Animations using Geometrical Shape Regularity. In *Computer Graphics Forum*, pages 155--166, 1994.

[VSC01] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast Cloth Animation on Walking Avatars. *Computer Graphics Forum*, 20(3):260--267, Sep. 2001.

[WB05] W.S.K. Wong and G. Baciu. Dynamic Interaction between Deformable Surfaces and Non-smooth Objects. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):329--340, 2005.

[WB06] W.S.K. Wong and G. Baciu. A Randomized Marking Scheme for Continuous Collision Detection in Simulation of Deformable Surfaces. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 181--188, 2006.