APPROVAL SHEET

Title of Thesis: GPU Based Cloth Simulation on Moving Avatars

Name of Candidate: Yi Wang
Master of Science, 2005

Thesis and Abstract Approved: _____

Dr. Marc Olano
Assistant Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

CURRICULUM VITAE

Name: Yi Wang.

Permanent Address: Apt 203, Bldg 16, Fang Yuan Xiao Qu, Xinji, Hebei 052360, China.

Degree and date to be conferred: Master of Science, August 2005.

Date of Birth: Dec 15, 1977.

Place of Birth: Hebei, China.

Collegiate institutions attended:

University of Maryland, Baltimore County, M.S. Computer Science, 2005.

Beijing Institute of Technology, China, B.S. Computer Science, 2000.

Major: Computer Science.

Professional positions held:

Software Developer and Team Leader, NEC-CAS Software laboratory Inc. (July 2000 - July 2003).

Research Assistant / Teaching Assistant, CSEE Department, UMBC (September 2003 - May 2005).

ABSTRACT

Title of Thesis: GPU Based Cloth Simulation on Moving Avatars

Yi Wang, Master of Science, 2005

Thesis directed by: Dr. Marc Olano

In real-time graphics applications, scenes involving cloth motion are very common. Real-time cloth simulation on moving characters is still a challenging task because of the long computation time to solve the physically based cloth model on the CPU. To improve the cloth simulation performance, we propose a level of detail (LOD) method to move part of the cloth simulation work to the Graphics Processing Unit (GPU) on current graphics hardware. Current GPUs are highly parallel stream processors, which execute compute-intensive operations more efficiently than CPUs, but provide less flexible resource access and logic controls. We take advantage of both the two processing units and combine them into a cloth simulation pipeline. The cloth is modeled as a mesh hierarchy. Only the coarse mesh is stored in main memory and processed by the CPU. After the coarse mesh motion is determined by the CPU, it is sent to the GPU for further refinement. We present a set of GPU algorithms that run in multiple rendering passes to subdivide cloth meshes, detect cloth-environment and cloth-cloth collisions and do collision response on the GPU. Our approach can be generalized to handle collision deformation behaviors and can be easily integrated in real-time graphics applications.

GPU Based Cloth Simulation on Moving Avatars

by
Yi Wang

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2005

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Two decades of research in physically based cloth motion simulation has turned the animator's dream into reality. In recent production animations like Pixar/Disney's Monsters Inc., we saw realistic cloth motion [5, 7]. However it usually takes seconds, if not minutes, to generate each frame. This is unacceptable for real-time applications.

Because cloth is easily deformable, it is usually modeled as a mass-spring system instead of a single geometry. The high computation time of cloth motion simulation is mainly due to two reasons:

1. To determine the position of the cloth nodes in each step, a large differential equation system which represents the entire, fully connected cloth mesh, has to be solved.

2. To guarantee correct collision behavior, both the collisions between cloth and external objects and cloth self collisions have to be detected and corrected. Unrealistic

collision behavior may ruin the whole simulation.

Many real-time applications calculate the cloth motion by decomposing this equation system to achieve inaccurate but visually acceptable cloth behavior. Collision detection between cloth and external objects is usually performed on their bounding volumes instead of the actual geometry meshes. More accurate real-time collision detection is not yet supported as a general case.

As current GPUs (Appendix A) become more programmable, the GPU can do some work to relieve the host CPU. The motivation of our research is to utilize the parallel processing capability of GPUs to process the local cloth deformation in parallel. We leave the global motion calculation on the CPU where frequent branching and global data access are needed. In this way, we combine the CPU and the GPU into a cloth simulation pipeline. The refined cloth model is rendered directly on the GPU without reading back to the CPU.

Current GPUs are high performance parallel stream processors, which have been proven to be faster than CPUs on some compute-intensive operations. In 2004, Ian Buck et al. compared the performance of an ATI Radeon X800 GPU, a NVIDIA Geforce 6800 GPU and a 3 GHz Intel Pentium 4 processor with several representative algorithms from numerical applications[10]. On average, the GPUs have better performance of up to several times as fast in their tests. We expect a larger benefit from GPUs in the future since the advance of GPU technology exceeds Moore's Law [1]. However, current GPUs are not on-chip PRAMs (Parallel Random Access Machines). They mainly follow a stream architecture and provide less flexible resource access and logic controls.

To overcome the limitations of the GPU, our CPU algorithm eliminates most logical controls and localizes memory accesses for the GPU by calculating the interaction between the cloth and the whole environment. To minimize the CPU computation and the CPU-GPU communication, the cloth geometry is modeled as a mesh hierarchy, with the coarsest mesh being the top level.

In each time step, the position of the top-level mesh is calculated on the CPU. Collision between coarse mesh vertices and the environment is also detected and handled on the CPU. Then the coarse mesh is sent onto the GPU for refinement. The refinement process can run multiple passes, with the output of the first pass as the input to the second pass. During each pass, new cloth vertices are generated by hardware interpolation. Self intersection between cloth meshes and collision with external objects are handled. The cloth is rendered directly on graphics hardware once the refinement is finished.

This level of detail method is based on the observation that the movement of a coarse cloth mesh largely determines the global motion of the cloth. Although the cloth is a continuous material, local cloth deformations rarely propagate to the whole cloth. Ignoring the global effect of local cloth deformation is visually acceptable in most cases.

We organize the thesis as the following manner: we first introduce three areas of related work: cloth dynamics, collision detection, and real-time cloth simulation. We then introduce current GPU architecture and model it as a general purpose parallel processor. We give an overview of our CPU-GPU cloth simulation framework and discuss its advantages and limitations in Chapter 4. In Chapters 5 and 6, we explain our CPU algorithm and GPU

algorithm in detail respectively. We also address some important implementation issues following the algorithm description. Finally we evaluate our approach in Chapter 7 and give conclusions and future work in Chapter 8.

Technical terminologies and notations are listed and defined in Appendix A.

Chapter 2

Related Work in Cloth Simulation

In this chapter, we summarize previous work in cloth motion simulation according to the three key problems they address: cloth dynamics in unconstrained space, cloth collision problem and real-time cloth simulation.

2.1 Cloth Dynamics

Generally, there are three approaches to model cloth motion: geometric-based, continuum-based and particle-based. Geometry approaches [25, 37] are the earliest model. The cloth shape is approximated by curved primitives like catenaries. No explicit force or energy equation is used when calculating the motion of these models. Continuum-based approaches [14, 36] are based on continuum mechanics and produce the most accurate simulation result. A piece of cloth is assigned an energy function, with the behavior of

the fabrics predicted by calculating the minimum value of the energy function. The third category of approaches is based on a spring-connected particle system rather than a continuous sheet [2, 5, 6, 7, 9, 28]. Hence some ad hoc assumptions are made when calculating forces. The particle-based approaches produce good results in a relatively short time and are accepted by the majority of the current cloth simulation community.

In particle-based approaches the crossing points between warp and weft threads are represented by particles with mass. These particles interact with adjacent particles and environment through mechanical connections. These connections are represented by energy functions and are often simplified to linear springs. Figure 2.1 shows a mass-spring model for a piece of cloth. The stretch force, shear force and bend force are all represented by springs. In this way, the cloth could maintain its shape during the simulation process.

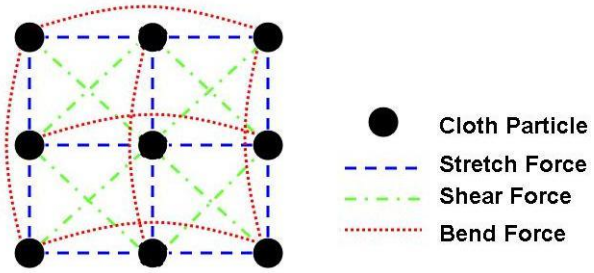


Figure 2.1: Mass-Spring Cloth Model.

In each time step, we calculate the particle positions by solving the spatial differential equation system:

$$\ddot{X} = M^{-1}\left(-\frac{\partial E}{\partial X} + F\right) \tag{2.1}$$

In Equation 2.1, X is the position vector of the cloth particles. M is a diagonal matrix representing the mass distribution of the cloth. E is a scalar function of X, representing

the cloth's internal energy. Matrix F represents the external forces (gravity, contact and friction and damping).

In practice, this equation system is solved by explicit or implicit numerical integration or a combination of these two. The explicit method assumes linear motion over each time step, i.e. the velocity $\dot{X}$ at the end of time step t is the same as at the beginning of time step t. This method is simple but it requires small time steps (proportional to the square root of the stiffness [28]) to guarantee stability and is ill suited for stiff forces like stretch and collision with solids. As an improvement, implicit methods require $\dot{X}$ at the end of time step t to be consistent with the current state X at time $t+1$. Since the position at the end of a time step t is not reached blindly, the implicit method tolerates larger time steps and hence is more stable than the explicit method. However, the explicit method is more simple and is acceptable in many cases.

Many real-time applications use Verlet Integration to do physics simulation [15]. Verlet integration is an explicit method and can be used in a decomposed equation system to drive the movement of cloth particles. We will discuss it in detail in Section 5.1.

2.2 Collision Detection and Collision Response

Compared to rigid body simulation, cloth simulation faces much harder collision detection and collision response problems. Unlike rigid bodies, a piece of cloth tends to collide with itself. To achieve correct simulation results, we need to detect two kinds of collisions:

cloth-environment collision and cloth-cloth collision. There is significant prior work on this topic [3, 7, 8, 9, 12, 29].

Collision detection techniques rely on the representation of the object to be detected. One common way is to represent both the cloth and the environment (human skin, chairs...) as triangle meshes. In this case, collision detection is usually performed by detecting vertex-triangle collisions and edge-edge collisions between two moving meshes. In order to avoid $O(n^2)$ comparisons (where n is the number of triangles), acceleration structures like bounding volumes are used. The most widely used bounding volumes are Bounding Spheres, Axis-aligned Bounding Boxes (AABB), and Oriented Bounding Boxes (OBB). Bounding volumes can be organized hierarchically to further reduce the calculation. For the cloth model a Binary Space Partitioning (BSP) tree can be obtained by recursively partitioning the cloth meshes according to their coordinates in the unfolded 2D texture space.

One successful non-real-time approach to handle cloth-cloth collision is Bridson et al's 2002 SIGGRAPH paper [8]. They handle collision, friction, and contact robustly even for complex structures of wrinkles and folds. They assume linear movement of cloth vertices in each time step and detect vertex-triangle and edge-edge collision by solving a cubic equation (described in Section 5.2). When cloth meshes contact each other, fast repulsion forces as well as static and kinetic friction are calculated using a simplified but physically plausible model. The computation time of this method is too long to be adopted in real-time applications.

There is much prior work on rigid body collision detection. Since our GPU collision detec-

tion method is an image-space method, it is worth mentioning some existing image-space collision detection methods that utilize graphics hardware [3, 4, 17, 18, 21]. The basic approach is to render all the objects from one view direction and generate a depth map. They extract object occlusion information from the depth map and find potentially colliding objects. They either read the depth image into the main memory and do detailed interference analysis on the CPU, or use hardware supported occlusion query (Appendix A) to get a brief answer of whether interference happens. Image-based methods are very efficient in finding potential collision sets. Our algorithm makes two significant improvement over these methods: we handle deformable bodies as well as collision response on the GPU. The differences between our method and existing ones are discussed in detail in Section 4.3.

In 2004, two similar GPU implementations of particle systems were published [23, 24]. These systems first sort the free-moving particles with a GPU bitonic sorting algorithm. Then they are able to detect and correct collisions between nearby particles on the GPU. These sorting-based methods can not solve the collision problem in cloth simulation because cloth particles form a single surface without self intersections. A state, where no collision happens between particles, may still be illegal if some particles move to the wrong side of the cloth.

2.3 Real-Time Cloth Simulation

Simulation speed is critical for real-time applications. Usually at least 10 frames per second is needed. The cloth dynamics and collision detection model have to be further simplified. Several pioneering works have appeared within the past five years to do fast cloth simulation [11, 28, 34, 38]. But none of them have come into wide application, either because real-time performance is only achieved in very limited cases or because the pre-computation is impractical for real-time applications.

In 2004, Cordier and Thalmann proposed an innovative data-driven approach to do real-time clothing simulation [11]. They pre-compute their cloth model on a fine mesh for a variety of human motions. At runtime, only a coarse cloth mesh is physically calculated. Initially the coarse mesh doesn't contain any wrinkle information. The simulator retrieves its pre-computed fine mesh data using the coarse mesh position and interpolates wrinkle shapes onto the coarse mesh. Their method achieved good visual results for movements that can be interpolated from the pre-computed motions, but it failed to work for movements outside the interpolation range. Their level of detail method is an inspiration for our research.

In 2004, NVIDIA released a demo that did simple cloth simulation on the GPU [38]. Their method uses two types of fragment shaders (Appendix A): a cloth motion shader and a collision detection shader. In the cloth motion shader, cloth vertices are advanced to the new position by Verlet Integration. The cloth position and spring forces are stored as 2D textures. In the collision detection shader, constraints are applied using a fragment shader.

The original demo can only handle collision with spheres. NVIDIA provided an improved version in the 2005 I3D conference [39]. Collision detection between cloth vertices and two basic geometry types are supported: ellipsoids and planes. Objects of each geometry type are stored into a 1D texture. For example, each ellipsoid is stored as nine floating point values in the texture. The first three values contain the center position. The following three values contain the orientation. The last three values contain the radius scales in three dimensions. The collision detection fragment shader reads objects from these 1D textures and detects interference with the cloth vertex in world space. NVIDIA's demo achieved very good performance for simple collision environments. Since frequent texture access is expensive on the GPU, their method doesn't scale well with the number of bounding volumes and could not handle precise collision detection with large geometry meshes.

Once collision is detected, each cloth vertex is pushed outside of the nearest surface. This pure geometry method may cause incorrect effects. Even though their method doesn't handle cloth self intersection, this phenomenon is not easily observed in their real-time demo. The reason is that the cloth shape of the self-intersecting part is complex and quickly changing. The observers don't have enough time to notice self intersections in real-time. The cloth motion captures the most attention.

Unlike NVIDIA's world space collision detection algorithm, our collision detection algorithm works in projected image space and hence scale well with the mesh complexity: only geometries projected onto the same pixel are detected for collision, most objects are culled by the transformation and rasterization process.

Chapter 3

Introduction to GPUs

In last chapter we have introduced several works based on GPU programming. In this chapter we briefly summarize the high level architecture of the GPU, model it as a general purpose parallel machine and discuss the limitations of this machine.

3.1 General GPU Pipeline

As indicated by the name, a Graphics Processing Unit or GPU is a chip designed for special purpose computation. Because of the unique nature of graphics computation, e.g. relatively fixed processing pipeline, highly parallel tasks and inherent SIMD nature, GPUs follow a quite different design philosophy from CPUs. Generally, current GPUs can be considered as an assembly line. Graphics data (or any kind of data) are preprocessed on the CPU and fed into the GPU pipeline. Data flows along the pipeline while being processed by a number

of stages until it reaches the end stage, the frame buffer. The frame buffer is automatically scanned by the video controller and the data is displayed on a monitor. Figure 3.1 shows the basic stages of the GPU pipeline. Notice that there are usually several copies of vertex processing units and fragment processing units working in parallel.

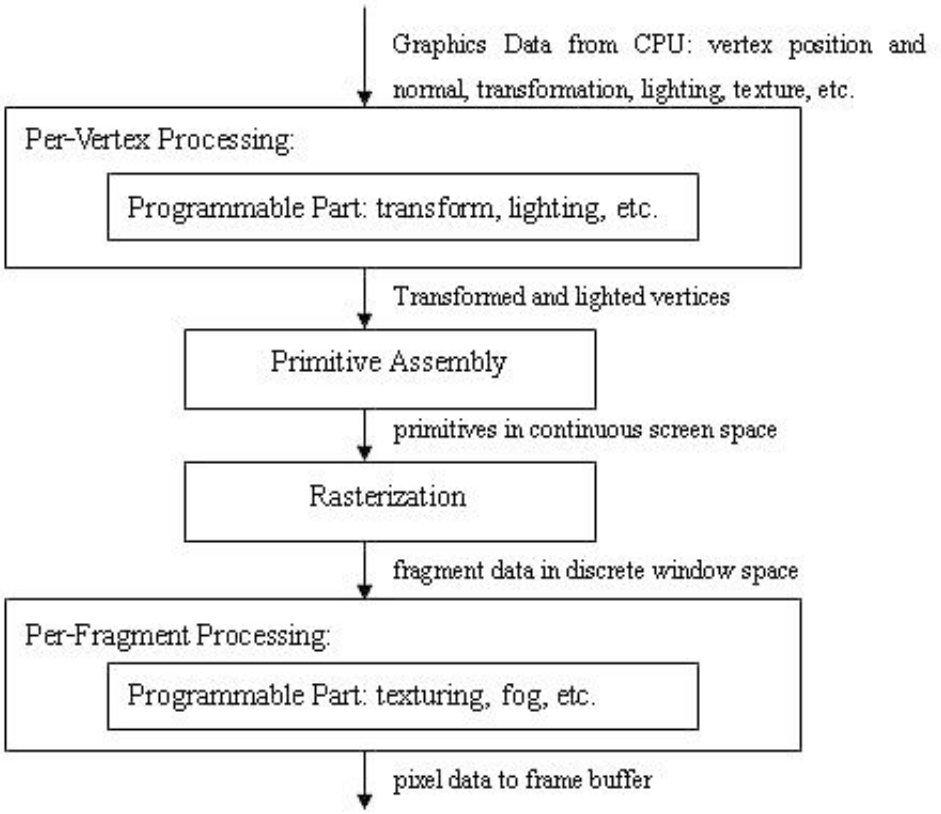


Figure 3.1: Processing flow of current graphics hardware pipeline. Part of the per-vertex processing stage and part of the per-fragment processing stage are programmable.

Within the past five years, some stages in the GPU processing pipeline have became programmable. The fixed transform and lighting units are replaced with vertex processors and fixed fog and texturing units are replaced with fragment processors. Vertex programs and fragment programs are loaded and swapped at run time to generate different shading effects. Initially, the aim of these extensions is to provide as flexible shading and texturing

functions as software renderers do. Later, people discovered that these processors could be programmed to do a variety of non-graphics computations [16, 20, 23, 30]. Because of these processors' high parallelism and deep pipelining, they are proven to out-perform CPU in those computation areas that have compute-intensive tasks with predictable memory accesses and infrequent branches.

3.2 Model GPU As a Parallel Processor

Cloth simulation is not the task that GPUs are originally designed for. By modeling the GPU as a general-purpose parallel processor, we can understand the GPU from a general purpose viewpoint. This will help us design an efficient scheme that avoids the shortcomings of the GPU and makes efficient utilization of its parallel execution units and other acceleration features.

Based on the high level architecture of the NVIDIA Geforce6 GPU [33], we abstract this GPU as a general purpose parallel processor shown in Figure 3.2 and design our GPU algorithm based on this abstract model.

In Figure 3.2, the upper set of parallel units (vertex processors) correspond to the programmable part of the per-vertex processing stage in Figure 3.1. The lower set of parallel units (fragment processors) correspond to the programmable part of the per-fragment processing stage in Figure 3.1. The serializer corresponds to the primitive assembly part in Figure 3.1. The interpolator corresponds to the rasterization part in Figure 3.1.

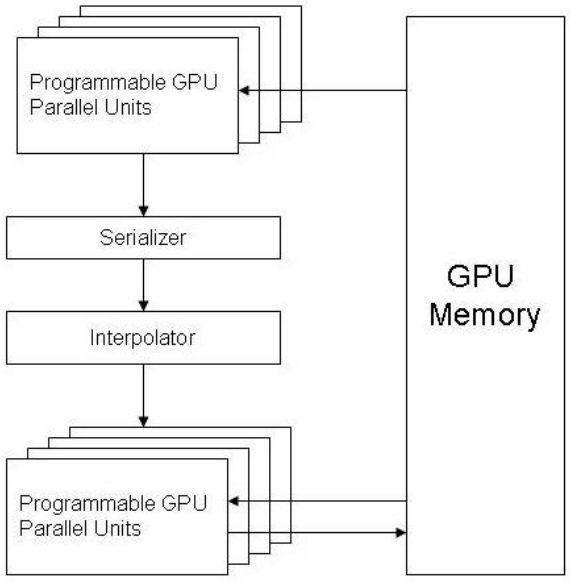


Figure 3.2: Model GPU as a Parallel Processor. The upper set of parallel units are the vertex processors. The lower set of parallel units are the fragment processors.

Figure 3.3 illustrates how data is processed on the GPU. The GPU first reads vertex data from the GPU memory, processes them with multiple parallel processing units. Then the processed vertices are grouped into triangles according to their connection information and interpolated into fragments. These fragments are processed by multiple parallel processing units and written to the framebuffer.

Before GPU execution, the CPU loads data and GPU programs to the GPU memory. Then under the control of the CPU, the vertex processors read instructions and data from the GPU memory, process the data as SPMD (Single Program Multiple Data) processors and write the result to the serializer. The vertex processor also calculates the output address (vertex location) for the fragment programs.

The data are processed in a serialized manner by some fixed functions and interpolated into a new set of data. The interpolator interpolates all the variables according to a special

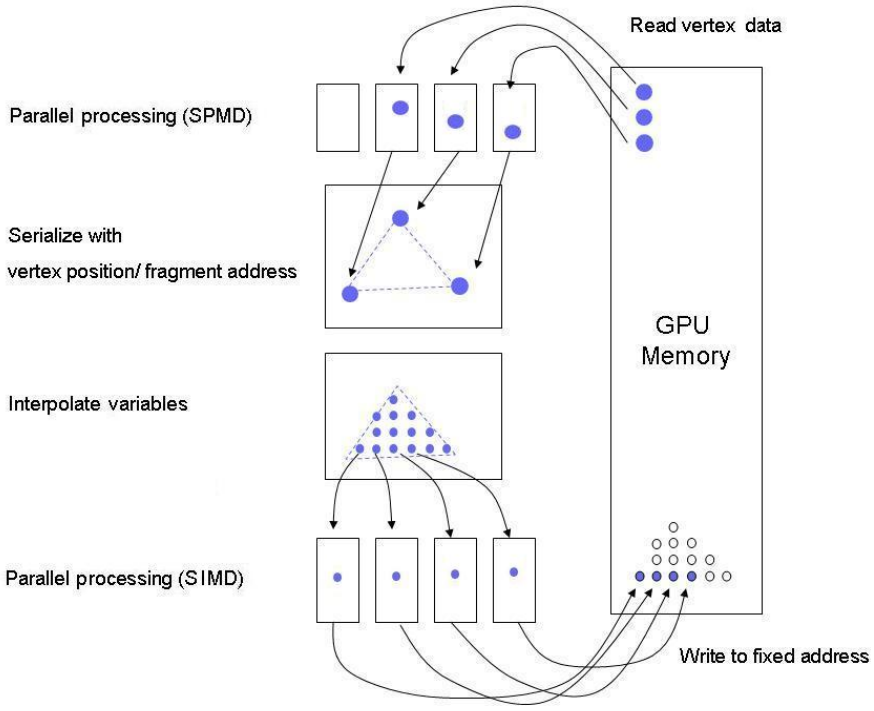


Figure 3.3: Data processing process of GPU.

variable (vertex position in window space).

The fragment processors take the new data as input, process them in a SIMD way and then write the output to a special GPU memory address. The fragment processors can't write to addresses other than the one calculated by the interpolator. The fragment processors can access global data in the GPU memory (by texture lookup). The output of the fragment processors can be used as the input of the vertex processor as well as the global data that can be accessed by later fragment programs.

We use GLSL [22], a C-like shading language to program the GPU. Unfortunately, we don't have as much freedom as CPU programming. The GPU has many limitations, which have to be kept in mind when writing GPU programs. We list the most important ones

below:

1. No random write. The output address of the fragment programs is calculated by a fixed unit or by the vertex program. The fragment program itself can't determine its output address. The vertex processor can not even write directly to the framebuffer. The only way to write the output of a vertex program to the framebuffer is to set the GPU in a special state that both the interpolator and the fragment processor pass the input through without changing it.

2. Limited number of output writes. Only a limited size of data can be written to the GPU memory. We have to use multiple rendering passes to output more data.

3. Limited global memory access. Only a limited number of GPU memory read operations are allowed.

4. Limited number of interpolators. Only a limited number of variables can be interpolated and passed from the vertex processor to the fragment processor.

5. Limited number of local variables. This limits the complexity of the GPU program.

Although multipass rendering methods (Appendix A) can be used to overcome these limitations, they lead to performance drops and precision problems.

Chapter 4

Design Overview

In this chapter we first introduce the basic idea of our CPU-GPU cloth simulation framework. Then we formalize the problem we want to solve. Based on the formalized problem, we discuss the whole approach in detail.

4.1 Problem Analysis

Our level of detail method is based on the observation that the movement of a coarse cloth mesh largely determines the global motion of the cloth. Although the cloth is a continuous material, local cloth deformations rarely propagate to the whole cloth. Ignoring the global effect of local cloth deformation is visually acceptable in most cases.

The cloth shape calculation process can be considered as a signal processing process. We can simulate the motion of the cloth well if the number of nodes (i.e. the sampling rate)

used to represent the cloth is proportional to the frequency of the external environment and hence can sample the environment well.

Consider an ideal case: a piece of cloth is falling in a vacuum and free space, only gravity is applied and all the springs are at rest condition. We can use a single node to represent the whole cloth since all the cloth nodes fall the same way. When part of the cloth hits a table surface, the cloth's motion can no longer be represented by a single node. The geometry detail of the table surface determines the number of nodes used to represent the cloth. A sparse cloth mesh can not follow any small underlying bumps on the table surface.

The actual environment usually contains objects of all frequencies. Big objects like a table surface shows up in low frequency space and small objects like bumps on the table surface, only appear in high frequency space. A reasonable way to determine the cloth shape is to first determine the global cloth motion by low frequency objects and then refine the cloth mesh according to high frequency details.

4.2 Problem Simplification

As indicated in the previous work, it is computationally expensive to simulate physically correct cloth motion in a general collision environment. For real-time applications, we aim at solutions that can generate visually a correct cloth motion in some typical situations, e.g. a cloak worn by a moving avatar.

We treat each piece of cloth as a 2D manifold surface, which well represents most clothes.

Cloth behavior is decided by two factors: the internal forces (mechanical properties of the cloth) and the external forces (gravity, repulsion force due to collision and friction due to contact). In most real-time applications the human body and other objects are represented by triangle meshes, so we try to solve collisions between the cloth triangle mesh and a triangle mesh environment.

For a visually correct solution, most interpenetrations between cloth and human body meshes should be removed since they can be easily observed. Self intersections between different parts of cloth are less obvious when the cloth is deforming very quickly. This fact is shown in NVIDIA's cloth simulation demo [38].

We make two assumptions for the collision environment:

1. The collision environment mesh is no finer than the finest cloth mesh. Our collision detection algorithm can be modeled as a geometric signal sampling process, which follows the Nyquist-Shannon sampling theorem. We use the cloth mesh to sample the collision environment. If the collision environment contains higher frequency geometries that are finer than the cloth mesh scale, the cloth mesh after collision correction may still penetrate into these geometries. Figure 4.1 shows such a case. Notice that the high frequency geometry is under-sampled by the cloth vertices and hence can not be detected.

2. The collision environment is composed of closed meshes. We use a depth-peeling algorithm to detect cloth vertices that are blocked from the observer by the environment mesh. In this case we require the collision environment to be composed of

closed meshes. Under this assumption, we can safely claim that a cloth vertex is outside the environment meshes if there are an even number of layers of environment mesh (not including cloth itself) between the observer and the cloth vertex. Figure 4.2 shows such a case. Cloth vertex A is outside the external objects and vertex B is inside the external objects.

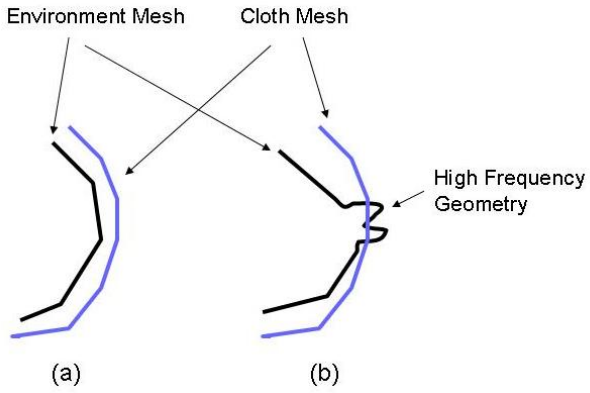


Figure 4.1: (a) Environment mesh is no finer than cloth mesh. (b) Environment mesh is finer than cloth mesh. Inter-penetration can not be detected.

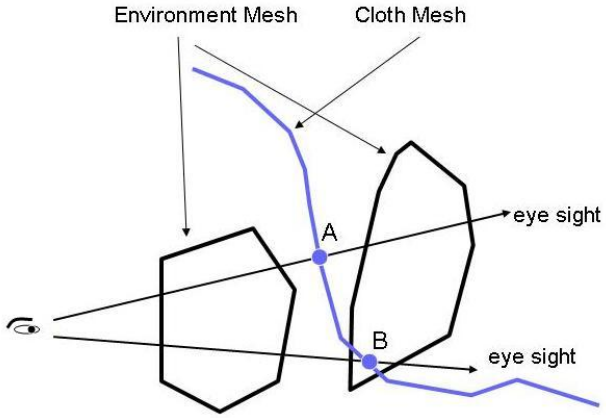


Figure 4.2: Environment mesh is closed. Two layers between observer and cloth node A, so A is outside environment mesh. Three layers between observer and cloth node B, so B is inside environment mesh.

4.3 Design Overview

Our method integrates two sets of cloth simulation algorithms. One set is executed on the CPU, the other on the GPU. For convenience, we will call them the CPU algorithm and the GPU algorithm respectively. The idea is to take advantage of both the two processors and combine them into a cloth simulation pipeline as shown in Figure 4.3.

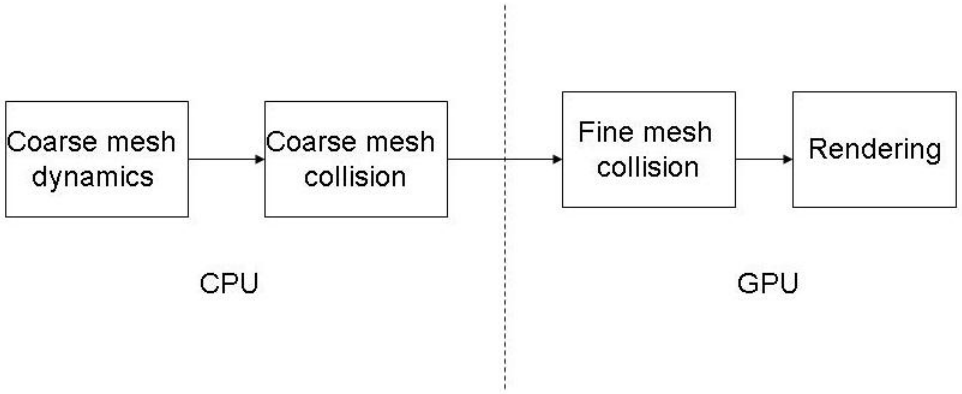
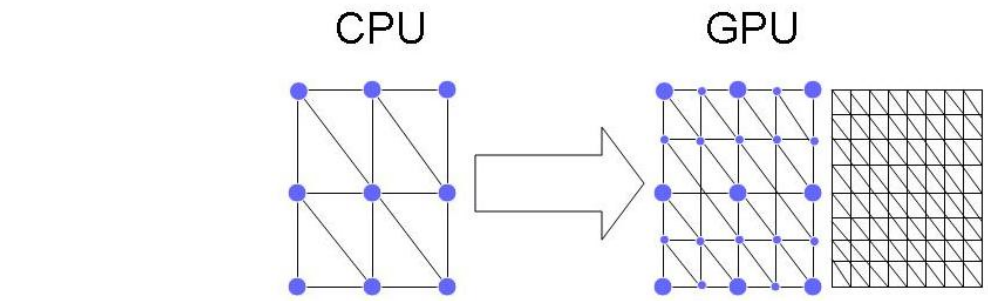


Figure 4.3: The CPU-GPU Cloth Simulation Pipeline.

The global motion of the cloth is calculated on the CPU, while the cloth local deformation and cloth rendering are done on the GPU. To overcome the limitations of the GPU (as mentioned in Section 3.2), the CPU algorithms eliminate most logical controls and localize memory accesses for the GPU by calculating the interaction between the coarse cloth mesh and the whole environment.

To minimize the computation on the CPU and the CPU-GPU communication, the cloth geometry is modeled as a mesh hierarchy, with the coarsest mesh being the top level. In each time step, only the top-level mesh is calculated on the CPU. The CPU first calculates the candidate position by the cloth dynamics equation. It then detects and corrects the

collisions between the coarse mesh vertices and the environment. Then the coarse mesh is sent onto the GPU for refinement. The refinement process can run multiple iterations, with the output of the first pass taken as the input to the second pass. During each pass, new cloth vertices are generated by hardware interpolation. Collisions with the environment are handled by an image-space GPU algorithm. The number of the output vertices becomes roughly four times that of the input. The cloth is rendered directly on graphics hardware once the refinement is finished. Figure 4.4 compares the functions of the CPU algorithms and GPU algorithms. The original 3×3 cloth mesh is refined by the GPU into 5×5 and 9×9 meshes. Inter-connections between cloth vertices are also updated.



Mesh Size:	coarse mesh	fine mesh
Space:	object space	image space
New Vertices:	no	yes
Cloth Dynamics:	yes	not yet
Collision Detection:	yes	yes
Self Collision Det.:	no	yes
Rendering:	no	yes
Precision:	high	low

Figure 4.4: Compare CPU and GPU algorithms.

Chapter 5

CPU Cloth Simulation

We only do the top-level coarse mesh calculation on the CPU. The position of the coarse mesh and the constraints will be sent to the GPU to generate finer meshes. Since the coarse mesh determines the movement of the whole cloth, we try to solve them accurately. This will not be a big burden for the CPU because the number of vertices in the coarse mesh is small (usually less than one hundred).

5.1 Cloth Dynamics

We model cloth as a mass-spring system. In each time step, we calculate the particle positions by solving the Spatial Differential Equation (Equation 2.1). When the number of cloth vertices is large, accurately solving this equation system is very slow. As in most real-time applications, we don't propagate spring forces to non-neighboring vertices in

each time step, so position calculation is localized and simplified. We further use Verlet Integration to approximate this equation. The Verlet Integration is shown in Equation 5.1.

$$X_{n+1} = X_n + D(X_n - X_{n-1}) + \frac{1}{2} * a * t^2 \tag{5.1}$$

D is the damping coefficient, which is usually very close to 1. a is the acceleration of the cloth node, which is calculated by dividing the force applied on this particle by the particle's mass. This equation assumes a constant speed damping and hence doesn't store the particles' speed.

This simplification may lead to some degree of spring-like behavior depending to the size of the cloth mesh. This is not a problem in our approach, since only the coarse mesh is calculated by the Verlet integration and the coarse mesh contains a small number of nodes.

External constraints have a higher priority than internal spring forces. If a cloth vertex is constrained by an external constraint, it neglects the internal spring forces applied to it. For example, the top row of the cloak nodes are stuck onto the avatar trunk mesh, so these cloth nodes' movement strictly follow the avatar's movement as shown in Figure 5.1. Their movement will propagate to other cloth nodes through the spring system.

Verlet integration advances all cloth vertices to a new candidate position by external constraints and internal spring forces. Some of these candidate position may be illegal. We then detect collisions between cloth nodes and the avatar skin mesh.

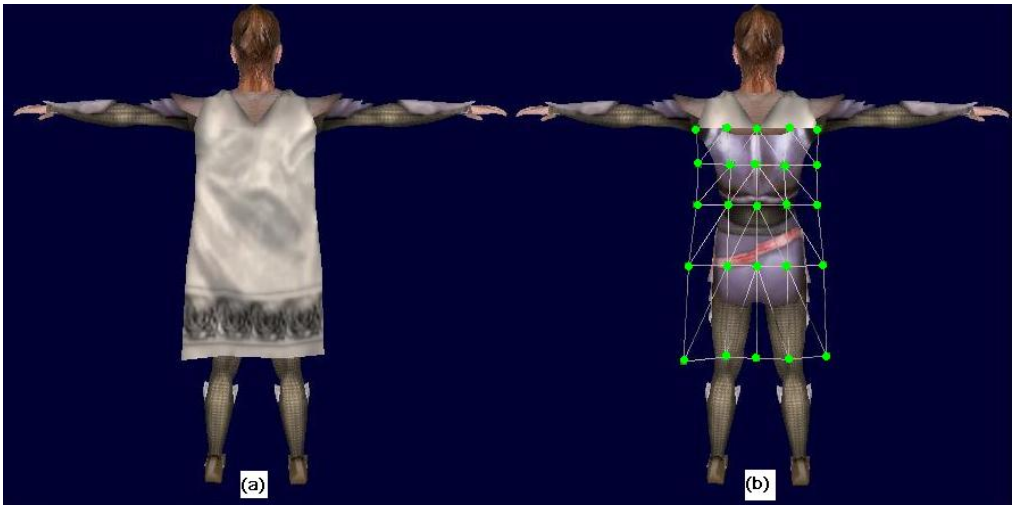


Figure 5.1: The avatar model and the cloak model. The green dots in Image (b) show the cloth particles. The top row of cloth particles are stick to the avatar mesh.

5.2 Collision Detection on CPU

As in most applications, we represent both the cloth and the environment (human skin, chairs...) as triangle meshes. In this case, collision detection is usually performed by detecting vertex-triangle collisions and edge-edge collisions between two moving meshes. In our coarse cloth mesh simulation on the CPU, we allow edge-edge collisions and only detect vertex-triangle collisions. Edge-edge collision will be solved on the GPU during the subdivision process.

We assume a linear trajectory for all the moving vertices during each time step. The vertex-triangle collision point under this assumption is found by solving a cubic equation that represents all the intersections of a line segment (trajectory of a moving vertex) and a sweep volume of a triangle (trajectory of a moving triangle). The diagram of the sweep volume and the cubic equation is shown in Figure 5.2. This equation satisfies when the four vertices A, B, C and P move onto the same plane.

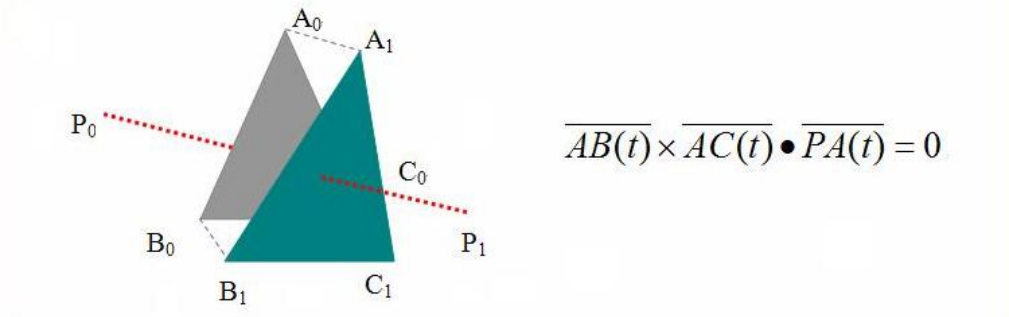


Figure 5.2: Intersection of a line segment and a sweep volume of a triangle. During time step t, triangle moves from (A_0, B_0, C_0) to (A_1, B_1, C_1), cloth vertex moves from P_0 to P_1.

In Figure 5.2. The cubic equation on the right is used to find moving vertex and triangle's intersection. A,B and C are the triangle vertices. P is the moving vertex.

This cubic equation may have multiple real solutions. We choose the one with the smallest *t* that falls into [0,1] range, since this is where the first intersection happens.

We used two bounding volume methods to reduce the collision detection cost:

1. An Oriented Bounding Box (OBB) is used to bound the set of triangles on the same body component. This reduces the number of lower level collision detections.

2. An Axis-Aligned Bounding Box (AABB) is used to bound the skin triangle sweep volumes. This reduces the chance of having to solve the cubic equation.

Collision correction raises another problem for real-time applications. Usually the environment is modeled as rigid body. So we want the cloth vertex to change its motion immediately where the trajectory intersection is detected. Otherwise inter-penetration may be observed in the next time step.

We handle collision in the following way: before collision with skin, cloth vertex moves along its original trajectory; after collision, it follows the skin's movement. So the whole trajectory is two connected line segments. We are actually assuming the cloth vertex will stick to the colliding body because of static friction. This is a simplified method, since in real life the friction may not be big enough to prevent the cloth from sliding. This method produces acceptable result because the collision point changes very quickly in a dynamic environment.

Chapter 6

GPU Cloth Simulation

This chapter describes the GPU algorithm in detail. We first give an overview of the algorithm and explain the working environment for the GPU algorithms. We describe the subdivision algorithm, the depth-peeling algorithm, and the penetration detection algorithm separately. Finally, we compare our GPU algorithm with existing GPU collision detection algorithms.

6.1 Terminology

We first define two terminologies that will be used to describe our GPU algorithm.

1. *Depth Layer*: We define the *depth layer* of a fragment F as the number of fragments that have smaller depth value than F's depth value in the post-projection space. Intu-

itively, depth layer indicates how many triangles block this fragment.

2. *Snapshot*: We define the *snapshot* as an image formed by projecting the scene onto a 2D viewport. A typical snapshot is the scene window shown on the screen. For dynamic scenes, the snapshots are different if they are taken at the same location and along the direction but at different time.

6.2 GPU Algorithm Overview

There are two types of data stored on the GPU. One is dynamic data, including the cloth mesh geometry image and the depth maps of the environment. The other is pre-computed static data, including the interconnection information (index array) of the fine mesh. Dynamic data is sent to the GPU in every cloth simulation time step after the CPU cloth simulation, while the static data is only loaded once at the beginning of the program.

At each time step, the GPU gets the position of the coarse cloth mesh as a vertex array from the CPU. The GPU cloth refinement is done in three steps. First, the coarse mesh is subdivided by the rasterizer and a finer mesh is generated. Second, we use a depth-peeling loop to find the depth layer values for all the cloth vertices. Finally, collision detection and correction is performed on all the newly generated vertices.

Figure 6.2 shows the flow diagram of our GPU algorithm. The control flow, which happens on the CPU, contains two levels of loops. The outer loop is the refinement loop. Each iteration refines the cloth mesh by one level. Once the cloth mesh is fine enough, the cloth

geometry image is copied from the on-chip texture buffer into the on-chip vertex array buffer and then rendered directly on the GPU (some hardware bypasses the copying process and renders the texture directly as a vertex array). The inner loop is the depth-peeling loop. Each iteration increments the depth layer value for each fragment by some amount (see Section 6.4 for details) and writes the result into the framebuffer. The depth-peeling pass also encodes the position, normal and depth layer value into output colors and stores them as textures. These data will be used as projective textures in the interpenetration detection and correction pass. The interpenetration detection pass analyzes the final depth layer map of the cloth mesh and corrects the detected interpenetrations.

After collision correction, we guarantee all the vertices in this level of cloth mesh are in a consistent state. We may still see penetrations in the rendered image as shown in Figure 6.1. This is because the cloth mesh is coarser than the human body mesh.

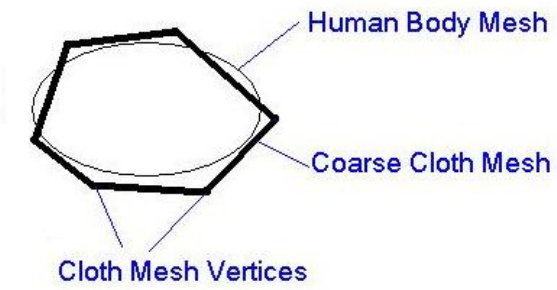


Figure 6.1: No collision detected in this level of cloth mesh, but newly interpolated cloth vertices may appear inside the environment mesh.

All we need to do is to subdivide the cloth mesh into a finer mesh and detect collisions in a higher frequency space. The GPU refines the mesh in multiple passes until it is fine enough to detect all the penetrations. Lastly, the GPU renders the refined mesh directly on the GPU

with a cloth lighting model.

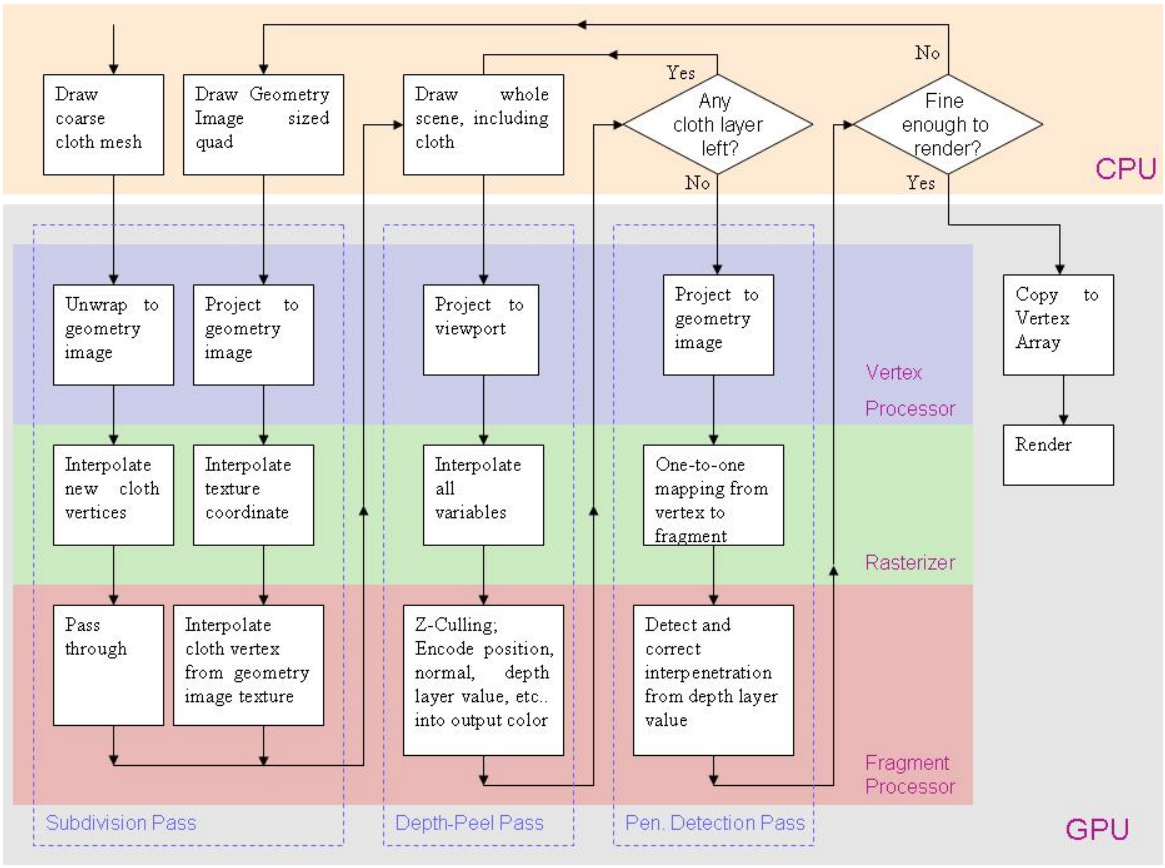


Figure 6.2: Flow Diagram of GPU cloth refinement.

Unlike the CPU algorithms which run in continuous (not sampled by the rasterizer) object space, our GPU algorithms run in sampled image space. What we see is fragments instead of triangles in this sampled space. Just as we can choose the most convenient view directions to view the world, we can also choose the most convenient image space to do computation. In our GPU algorithm, the subdivision and penetration detection pass run in geometry image space of the cloth, while the depth peeling pass runs in the projected image space of the scene. As long as we have a mapping function, we can conveniently map data from one space to the other.

We give an example to show the difference between object-space and image-space. To determine whether a cloth vertex is behind another object along the view direction, the continuous space method is to use the object surface's analytic equation to test whether the cloth vertex is inside the surface. With the image-space method, we project the object onto the view plane, sample (rasterize) its depth value into a 2D image and save the image. This sampled image contains less information than the original object model. Then we project the cloth vertex onto the view plane and use the vertex's projected (x,y) coordinates to retrieve a texel from the projective depth map. The comparison result between the retrieved depth value and the cloth vertex's projected z coordinate tells us which is in front. This projective texture (Appendix A) method is accelerated by graphics hardware.

6.3 Cloth Mesh Subdivision

We use different subdivision methods for the first refinement pass and the following passes.

During each time step, the input of the first refinement pass is a cloth vertex array from the CPU. This array contains both the position and geometry image (Appendix A) coordinates of the cloth vertices. This GPU subdivision algorithm requires the geometry image of the cloth model to be pre-calculated. This can be done by several existing algorithms [19, 26, 35]. Since our cloth model is a rectangle, we simply use its texture coordinates to map to the geometry image, which is also a rectangle.

If the original mesh contains $n \times n$ vertices, the GPU will draw a $(2n-1) \times (2n-1)$ image

and exchange vertex position and texture coordinates in the vertex shader. So the GPU rasterizer will assign the interpolated vertex position to the newly generated vertex (pixel in the render target) in the geometry image. The subdivided mesh will contain $(2n-1)\times(2n-1)$ vertices. The position of the new vertices are linearly interpolated from their neighboring parents. Figure 6.3 provides a diagram of the subdivision process. After each subdivision process, a newly generated vertex is either interpolated from two or four neighboring vertices from the upper level of cloth mesh. We call these upper level cloth vertices the *parents* of the new vertex.

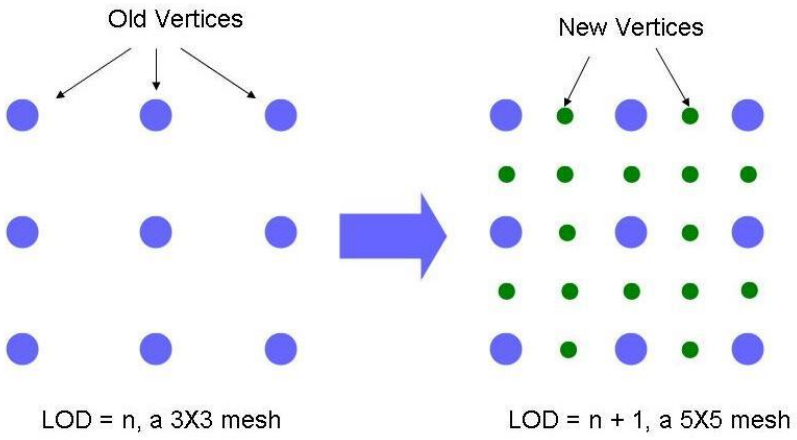


Figure 6.3: Surface subdivision on the GPU.

We use the rasterizer on the GPU to insert a new vertex between every pair of neighboring vertices in each column or row in the cloth geometry image. We render the cloth's geometry image onto the screen with each pixel covering one vertex, so all the cloth vertices can be processed by the fragment shader.

In the later refinement passes, the CPU no longer sends cloth vertex data. Instead, it draws a quad covering the geometry image of the cloth. The GPU uses the last pass's render result, which is a 2D cloth geometry image in the GPU memory, as a texture for the quad.

The actual geometry image coordinates of the cloth are then automatically generated by the rasterizer. This subdivision method reduces the CPU-GPU communication.

After the subdivision step, the new cloth vertex may penetrate into the environment mesh or the cloth mesh. The GPU algorithms should detect this case and correct it within the current subdivision pass. Otherwise, this wrong effect will propagate to the final image.

6.4 Depth-Peeling

Collision detection on the GPU is done by comparing the depth layer values between neighboring vertices in the cloth geometry image. The depth layer value is accumulated during the depth-peeling (Appendix A) process. The whole scene is peeled in a front-to-back order.

We store the depth value as well as the depth layer value in each snapshot. In each depth peeling pass, a fragment shader takes the snapshot of last depth-peeling pass as a projective texture and detects whether the cloth vertex shows up in the texture. If found, the depth layer value remains the same. Otherwise, it increases by 1 if the fragment is on an environment mesh. The shader can tell whether the fragment in this layer is on the environment or the cloth by the object ID (encoded into color) of the current fragment. It can also distinguish different parts of the cloth from the its geometry image coordinate.

The depth layer value accumulates until the cloth vertex is found in one snapshot. By the end every cloth vertex knows which depth layer it is in. Figure 6.4 shows the result of

depth-peeling process. From left to right, each depth peeling-pass generates one image. One obvious observation is that the right hand of the Cally model begins to appear after the third depth-peeling pass.

Figure 6.4: A series of images generated by depth peeling. All fragments (except background) in each image have the same depth layer value.

Since we only sample the snapshot for each cloth vertex, we don't need to render the cloth as a triangle mesh to generate a snapshot. Instead we draw cloth vertices as discrete points without inter-connections. This method reduces the time to generate a snapshot. Rendering as points also avoids instability problems for vertices on the cloth boundary.

6.5 Penetration Detection

Before each subdivision pass, we always guarantee the upper level of cloth mesh is consistent with the environment, i.e. all the cloth vertices in the coarse mesh are outside the human body volume (if only the human body is considered for collision detection). If a newly generated vertex is an odd number of layers away from all its neighboring parents, we consider this vertex to be in an inconsistent state and correct it.

We now describe the penetration detection algorithm in detail. For each newly generated vertex on the cloth mesh, the algorithm examines the depth layer value of this vertex as well as its eight immediate neighbors in the geometry image. We could estimate the local shape of the collision object from this 3×3 array of depth layer values, and find penetration accordingly. Old vertices needn't be examined because they are already corrected by previous runs of the algorithm.

For convenience, we suppose the environment object has at most two layers of meshes. Higher layer problems can be solved in a similar way. In a two-layer environment, a cloth vertex may has a depth layer value of 0, 1 or 2. Parent vertices can only be 0 or 2, since they are guaranteed to be outside the environment object. We enumerate all the combinations of depth layer values of these vertices and see which combination indicates an interpenetration.

For new vertices with two parents, only three combinations indicate an interpenetration as illustrated in Figure 6.5:

(a.1) The depth layer values of the two parents are both 0 and the new vertex is 1.

(a.2) Two parents are both 2 and the new vertex is 1.

(a.3) One parent is 0 and the other parent is 2. The new vertex is 1.

For the case where the two parents are both 0 and the new vertex is 2, we consider it to be a legal case since the new vertex is outside the environment object mesh. This case happens when a small object lies between the two parent cloth vertices in the projected image.

For new vertices with four parents, the situation is not as complex as it seems to be. Since

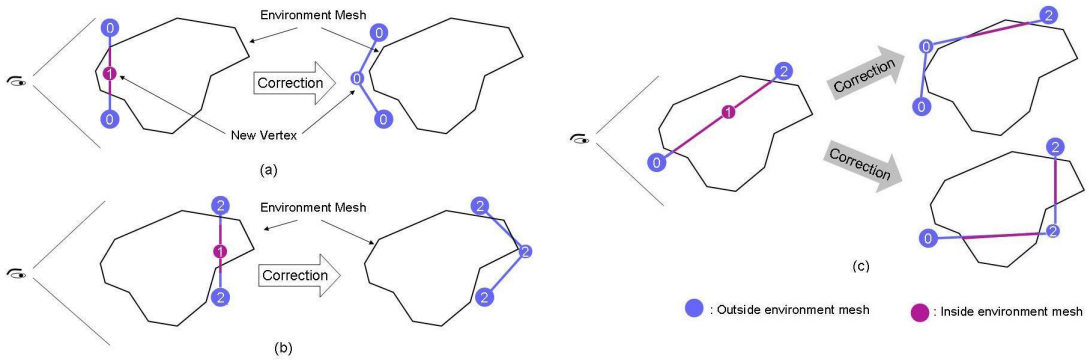


Figure 6.5: Collision detection and correction with two parents. (a): case a.1; (b): case a.2: (c): case a.3. Notice the two possible ways to correct case a.3 collision.

interpenetration only happens when new vertex has a depth layer value of 1, we can classify the combinations of the four parents into five cases as illustrated in Figure 6.6 and Figure 6.7:

(b.1) All the four parents are 0.

(b.2) All the four parents are 2.

(b.3) Three parents are 0 and one parent is 2.

(b.4) Three parents are 2 and one parent is 0.

(b.5) Two parents are 0 and two parents are 2.

We only retrieve the object from the 1D object list texture and detect world space collision when we found projected image space collision. This method works because world space collision may happen only if projected image space collision happened.

Based on the two assumptions introduced in Section 4.2, the above penetration detection algorithm can detect all the penetrations within the viewport resolution precision.

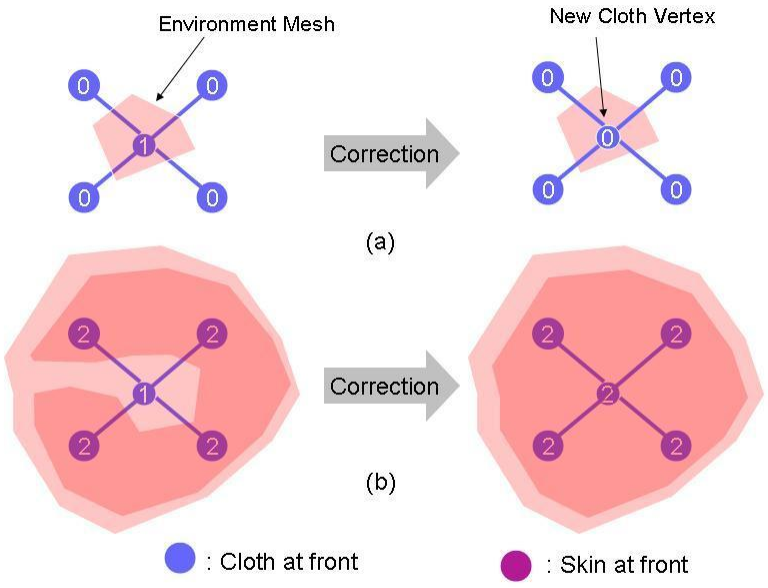


Figure 6.6: Collision detection and correction with four parents. (a): case b.1; (b): case b.2

6.6 Penetration Correction

Once penetration is found, we can use a variety of methods to push the cloth vertex outside the collision object. We list some of them:

1. Image-space Method: This is a simple method in which we push the cloth vertex one depth layer forward or backward to the object's surface position, and offset a little along the surface normal to represent the cloth thickness. This method is easy to implement for image-space collision detection approaches. Since the object surface position found by this method may be quite different than the actual collision position, this method sometimes causes wrong effects.

   The image-space method may meet ambiguity problems on choosing the depth layer

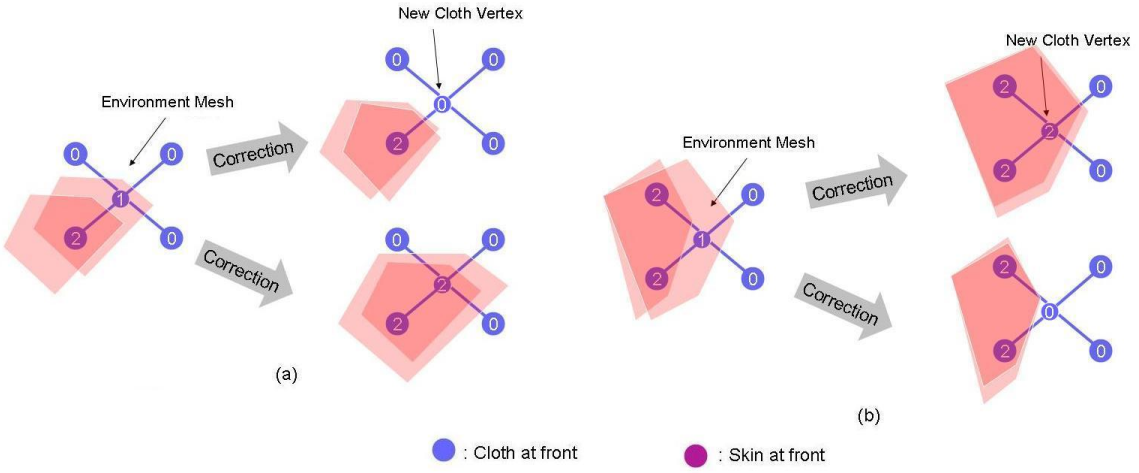


Figure 6.7: Collision detection and correction with four parents. (a): case b.3; (b): case b.5

to push the cloth. For example, the collision case (a.3) is ambiguous as shown in Figure 6.5. Cases (b.3), (b.4) and (b.5) are also ambiguous as shown in Figure 6.7. We can't tell which correction option is right because of the limited sampling rate of the 3×3 mesh. A simple way is to let the new vertex follow the majority and always correct to front for tied cases.

2. Object-space Method: This method finds the nearest vertex on the collision object surface and pushes the cloth vertex outside of that vertex. Obviously, this method still doesn't give physically correct answers all the time, but is easy to implement for object space collision detection methods. NVIDIA's cloth simulation demo used this method.

3. History-based Method: To support more accurate collision correction, it is necessary to consider the previous position of the cloth vertices. Figure 6.8 illustrates such an example. If we only consider the current cloth position, we can not tell which

correction direction is the right one. This method would be quite complex since both the movement of the cloth and the environment need to be considered. We will not discuss it in this thesis.

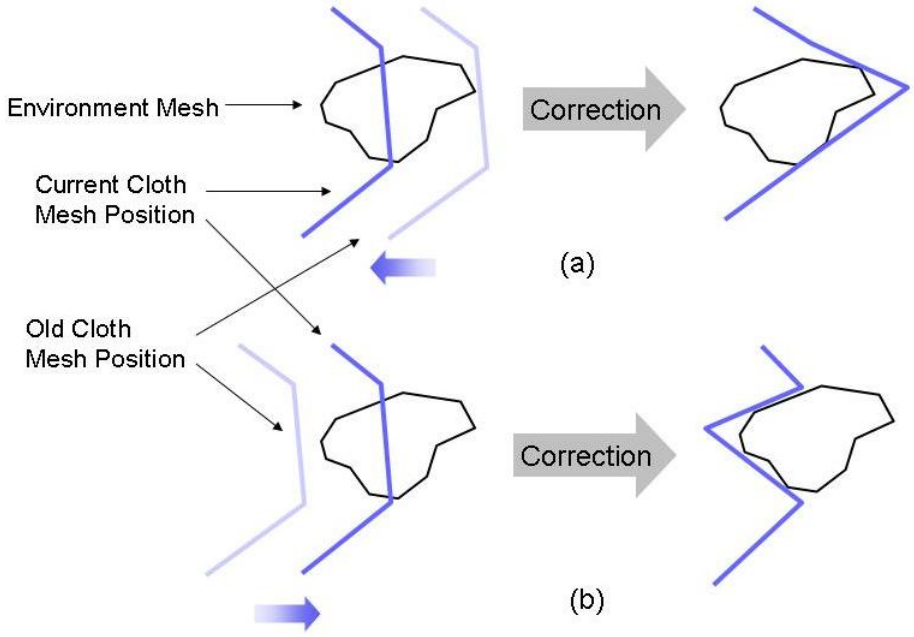


Figure 6.8: Need previous cloth position for correct collision response.

6.7 Advantages and Limitations

Compared with NVIDIA's world-space collision detection algorithm on the GPU, our collision detection algorithm works in projected image space and hence scales well with the mesh complexity: only geometries projected onto the same pixel are detected for collision, most objects are culled by the transformation and rasterization process.

The image-space penetration detection algorithm has another advantage: it knows which penetrations are visible and can choose to detect and correct only visible penetrations. For

example, our depth-peeling iteration can break if all the depth layer values are bigger than two. Since the cloth vertices will not be corrected more than one depth layer further, we can not see the corrected result anyway. This trick doesn't affect the cloth motion in the next time step because the coarse cloth motion is already determined.

Compared with existing image-space collision detection methods [3, 4, 17, 18, 21], our method has three major advantages:

1. We use the geometry image of the object (cloth) to sample the projected depth image for object interference. Existing work either read the depth image into the main memory and did detailed interference analysis on CPU, or used hardware supported occlusion query (Appendix A) to get a brief answer of whether interference happens. With the geometry image and projective texture, we can get more detailed information about where the collisions occur and handle many collisions on the GPU in parallel.

2. Our method deals with both collision detection and collision response for deformable objects. Our method can simulate collision deformation phenomena. Existing methods only detect collision or compute proximity between two existing meshes. Our method, however, can generate finer meshes dynamically during the collision detection process. The bonus is that only the coarse mesh need to be stored in main memory.

3. Our algorithm handles the viewport resolution problem well. All image-based methods inherently have viewport resolution problems. Interpenetrations between geome-

tries smaller than a pixel size in the projected image can not be detected. However, by keeping the collision detection resolution consistent with (or even higher than) the display resolution, our method guarantees the viewport resolution error will not be observed.

Our current algorithm has two major limitations on handling collisions:

1. Only corrects newly generated vertices.

   During each subdivision step, when a penetration is detected, current algorithm corrects the newly interpolated vertices and leaves the parent vertices at the initial position. The newly generated springs are not used for cloth dynamics calculation.

   This may cause severe deformation when the penetrating geometry is sharp. Figure 6.9 compares three different collision detection and correction results in a situation with sharp geometries. This example simulates the process of a piece of cloth falling onto a table in three time steps. Case (a) in Figure 6.9 uses the pure CPU method, which only detects collision for the coarse mesh. Case (b) and case (c) both detect collisions for the fine mesh as well as the coarse mesh. Case (b) uses our GPU method, which doesn't change the parent particles' movement according to new particles' movement. As shown in (b.2) of Figure 6.9), the parent particles make no response when their neighboring new particles hit some objects. Case (c) uses the same method as method (a) with a finer mesh. The parent particles' movement is affected by the new springs.

   One direct solution to this problem is to apply new cloth springs on the parent parti-

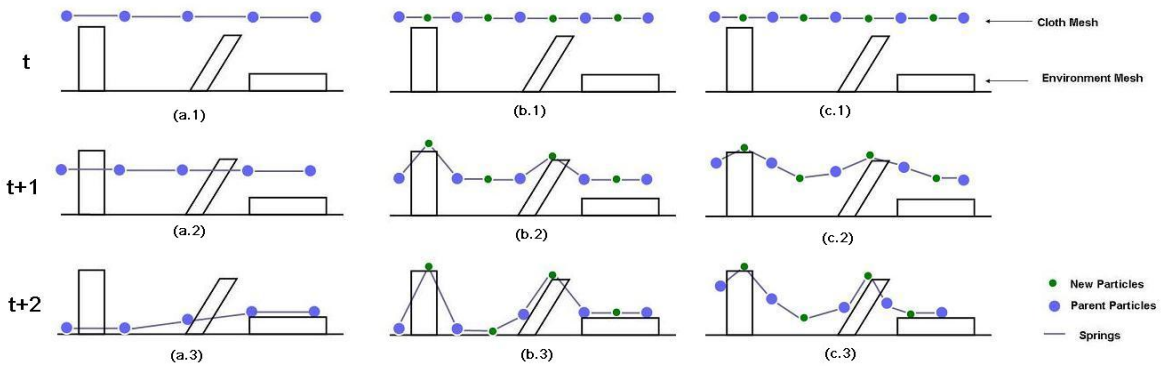


Figure 6.9: A piece of cloth falling onto a table. (a.1)-(a.3): Only parent particles are considered for collision. (b.1)-(b.3): Both parent particles and new particles are detected for collisions, but new particles' movement don't affect parent particles. Severe spring deformation can be observed. (c.1)-(c.3): Both new particles and parent particles are detected for collisions. Also use fine mesh springs to move parent particles.

cles using the GPU [39]. However, the moved parent vertices may penetrate into the environment mesh and need another iteration of collision detection.

2. Only corrects along the projection direction.

   Like most image-space methods, our method is also limited by the projection direction. The penetration detection algorithm reports the fact of penetration and the surface position where penetration is observed, instead of the actual collision position. If the penetration correction algorithm deforms the cloth mesh based only on view-dependent information, it won't work out physically correct results. To overcome this limitation, we could combine our image-space GPU algorithm with an history-based collision detection and correction algorithm. We use our image-space algorithm to find the collision object and use the history-based GPU algorithm to find the precise collision position. This hybrid algorithm could overcome the limitations of the image-space method and keep the collision detection cost low.

Chapter 7

Evaluation

We implemented our cloth simulation algorithms based on Cal3D Version 0.9.1, a real-time skeletal based character animation library. Cal3D provides a simple spring-mesh cloth model without any collision detection. The CPU cloth simulation algorithms are implemented by adding new collision detection functions to this library. The GPU cloth simulation algorithms are implemented as a set of OpenGL function calls and GLSL shaders. All the implementations run on NVIDIA's Geforce 6800Ultra GPU.

Since the Cal3D Cally demo resembles a typical case in real-time graphics applications, we use this demo as the evaluation environment.

7.1 Cal3D and Cally Model

Cal3D provides an avatar model (named Cally) that has several predefined actions: walking, stooping, running, staggering, and waving hands. Cal3D can perform animation blending. The avatar can execute multiple animations at the same time and Cal3D will blend them together smoothly.

We use this Cally model to represent the collision environment and a simple cloak model to represent the clothes worn on the avatar. The original cloak model is a 5×5 rectangular cloth mesh with the top row of vertices stuck to the body. This simple model simplifies our modeling process. However the collision detection calculation of the cloak is no cheaper than that of other clothes, e.g. T-shirts and trousers, because the cloak has more freedom on its movement and may collide with most parts of the body. The avatar is defined using triangle meshes, with movement modeled in a 3D modeling package.

The motion is calculated by traditional skeleton-driven deformation (SDD) method. The animation file defines a series of skeleton movements. The skin mesh vertices are interpolated from its nearby skeleton bones with different weights.

7.2 CPU Cloth Collision Detection Results

We calculate the motion of a 5×5 coarse cloth mesh on the CPU. The CPU handles the dynamics and the collisions with a very little performance drop. However the effect is

obvious: all the coarse mesh vertices are pushed out side the body. Figure 7.1 compares the cloth motion with and without CPU collision detection. After CPU collision correction, the cloth motion is largely correct. The local penetrations, whose scale is less than the coarse cloth mesh size, will be corrected on the GPU.

Figure 7.1: CPU Collision Detection and Correction Result. Image (a) and (b) shows the cloth motion before collision detection and image (c) and (d) shows the cloth motion after collision detection and correction. From the cloth mesh wireframes in (b) and (d), we can clearly see that the cloth vertices are corrected outside the body.

7.3 GPU Subdivision and Collision Correction Results

The GPU subdivision algorithm and collision correction algorithms can detect and correct collisions successfully and show good results in most cases. Figure 7.2 and Figure 7.3 compare the results with and without the GPU collision correction.

Depth-peeling finds cloth vertices that are behind part of the body but are still outside the closed body mesh. Figure 7.4 compares the results with depth-peeling and without depth-peeling:

Besides the cloak model, we tried a robe model as another cloth shape. Figure 7.5 shows the experiment result of the robe model. The robe is simplified as a cylinder surrounding the body. The upper image is after CPU collision detection and before GPU penetration correction. The lower image is after GPU penetration correction. Some penetration areas around the thigh are not corrected in the lower image. This is because the thigh triangle mesh is not strictly closed as the GPU algorithm requires.

7.4 Performance Evaluation

We evaluated our implementation on a machine with the following configuration:

OS: Windows XP Professional

CPU: Dual Intel P4 3.2 GHz

Main Memory: 1GB

Mesh Size	No Collision	CPU Collision	GPU Collision	Depth-Peeling
5×5	60	60	-	-
9×9	60	47-60	60	60
17×17	60	30-45	60	60
33×33	60	1-8	60	30
65×65	48	<1	52	30
129×129	14	<1	29	19

Table 7.1: Performance data for walking action. The numbers are measured in frames per second.

GPU: NVIDIA Geforce6800 Ultra GPU with 256MB Video Memory

The Cally avatar model used for collision detection test contains 2131 vertices. The performance is measured in frames per second (FPS).

To compare the GPU cloth simulation performance with the CPU, we also implemented a CPU cloth mesh subdivision algorithm. The CPU mesh subdivision algorithm interpolates vertex positions and vertex weights, adds necessary constraints and generates new springs for the finer mesh. The subdivision algorithm is called once a time when the user sets a mesh detail level. During each time step, the CPU dynamics calculation and the CPU collision detection are performed on the finer mesh.

We compared the CPU and the GPU performance for two different avatar actions: a walking action and a sneaking action. The sneaking action involves more frequent collisions between the cloak and the body. Table 7.1 compares the CPU and GPU subdivision and collision detection performance for a walking action. Table 7.2 compares the CPU and GPU subdivision and collision detection performance for a sneaking action.

From the data listed in the above two tables, we have the following observations:

Mesh Size	No Collision	CPU Collision	GPU Collision	Depth-Peeling
5×5	60	60	-	-
9×9	60	18-43	60	60
17×17	60	3-6	60	60
33×33	60	<1	60	29
65×65	48	<1	52	30
129×129	13	<1	29	19

Table 7.2: Performance data for sneaking action. The numbers are measured in frames per second.

1. The GPU performs faster than the CPU. By comparing the CPU performance with and without collision detection, we can see that collision detection consumes the major part of the GPU time. The dynamics calculation is not neglectable when the mesh size is larger than 65×65. The GPU algorithm achieves better performance by replacing the expensive CPU dynamics calculation and collision detection with a less precise but acceptable approximation.

2. The GPU performance is more stable than the CPU. This conclusion comes from the fact that the CPU FPS in the sneaking action is much lower than in the walking action, while the GPU FPS remains the same. The GPU performance is independent of the number of collisions, while the CPU performance depends heavily on the number of collisions.

3. The GPU performance drops when depth-peeling is enabled. This is because the whole scene has to be rendered during every depth-peeling pass.

Figure 7.2: Collision is successfully detected and corrected in the finest mesh. Observe that collision is not detected until the second subdivision pass. This is because the cloth mesh is too sparse before the second pass and the newly added vertices are all on the correct side of the skin mesh.

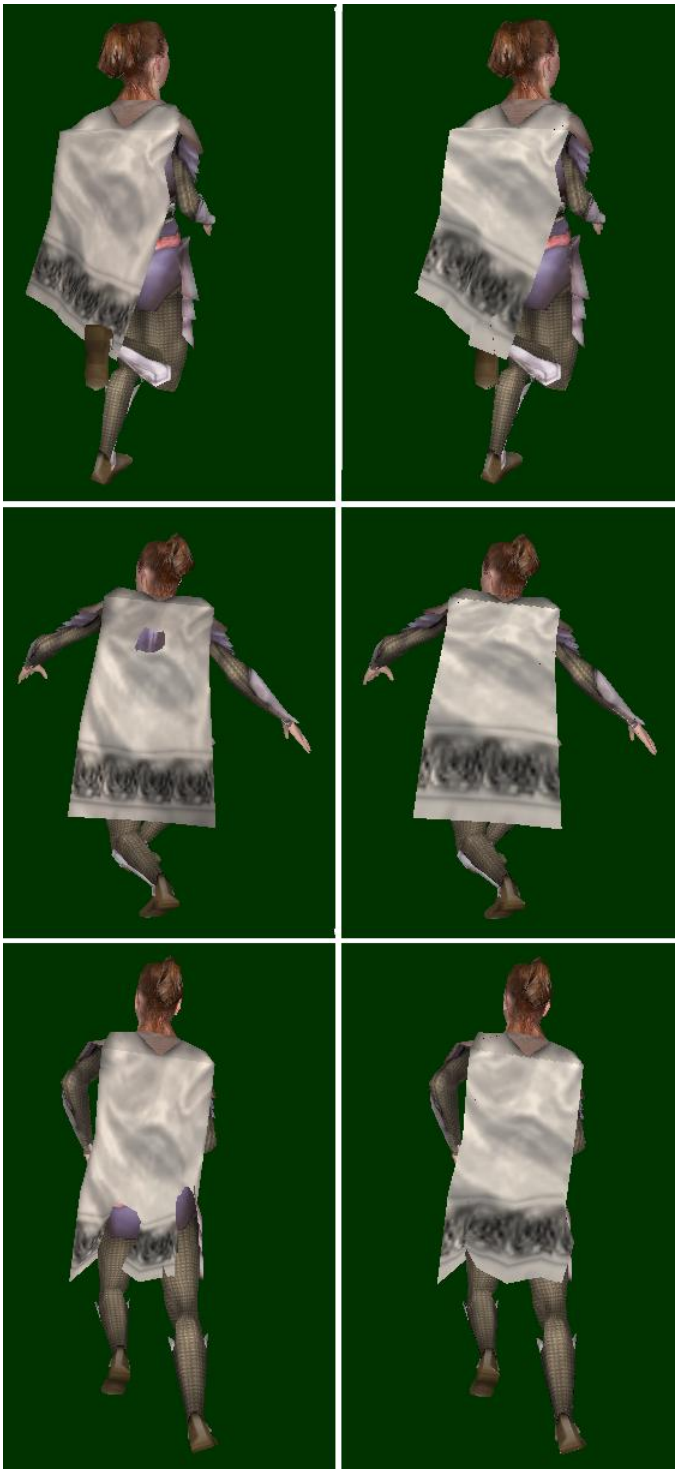


Figure 7.3: The images on the left show the scene with no GPU correction. The images on the right enable GPU collision correction.

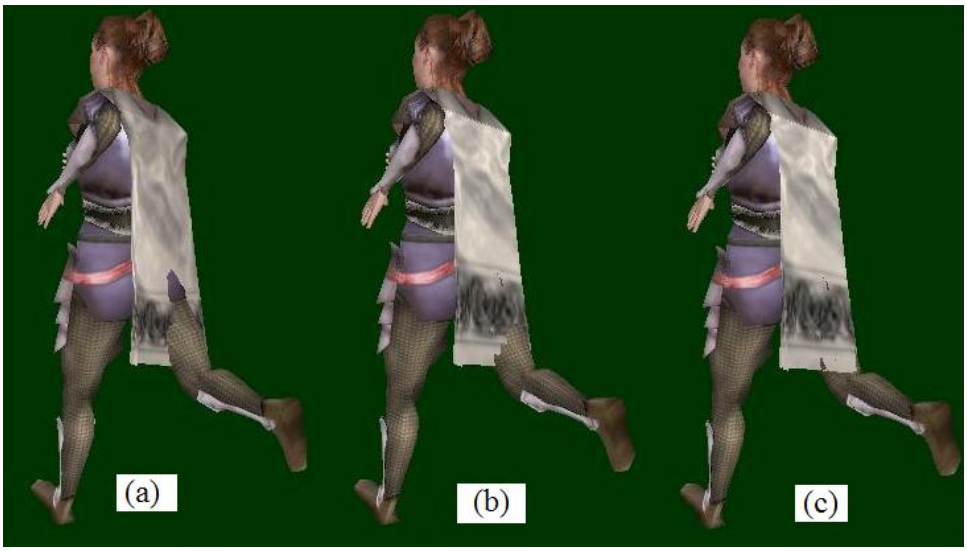


Figure 7.4: Depth-peeling Result. Image(a) is without GPU collision correction. Image(b) is with GPU collision correction but without depth-peeling. Image(c) is with both GPU collision correction and depth-peeling. In the last image, most vertices are pushed to the correct position, except for the vertex in the lower right corner of the original CPU mesh.

Figure 7.5: Collision detection result of the robe model.

Chapter 8

Discussion and Conclusions

In this work, we designed and implemented a framework to use both CPU and GPU to do real-time cloth simulation. The cloth mesh is organized hierarchically. Coarse mesh simulation is done on the CPU and fine cloth simulation is done on the GPU. The GPU algorithm integrated a surface subdivision method with an image-space collision detection method. Collision is detected and corrected during cloth surface subdivision process. Theoretical analysis as well as experiment results are given to support this framework.

Our CPU-GPU implementation also provides a platform to investigate more interesting collision detection and correction algorithms. There is plenty of future work possible in this CPU-GPU cloth simulation framework:

1. Implement history-based collision correction method on the GPU.

2. Allow upper level mesh modification during the collision correction process.

3. Displace the newly interpolated vertices with forces (spring forces, gravity...) and constraints in the fragment shader of the subdivision pass.

4. Cloth-cloth collision correction on the GPU.

5. Pre-filter the external objects before detecting collision in projected image space.

6. Use vertex shader and vertex texture to do penetration detection and correction, so we can combine the penetration detection pass and the next subdivision pass into a single pass.

APPENDIX A: Terminology, Notations and Techniques

This appendix defines the terms and techniques that are used in this thesis.

Terminology and Notations

1. *GPU*: Graphics Processing Unit. A special purpose processor loaded on computers to accelerate graphics calculations.

2. *Shader*: A shader is a program that uses some data, e.g. light directions and textures, as input and output colors to points on a surface. Most rendering systems generalize the term “shader” to refer to any programmable stage of the rendering pipeline. Shaders are executed by the CPU in some systems. However in real-time graphics systems like OpenGL and DirectX, shaders usually run on the GPU. In this thesis, the term “shader” refers to these GPU shaders.

3. *Geometry Image*: A geometry image is a 2D image created by remeshing an arbitrary 3D surface onto a regular structure [19, 26]. Once a 3D model is remeshed into a geometry image, it can be efficiently processed by graphics hardware because the GPU is optimized for processing homogeneous data arrays. For example, a geometry image can be subdivided on GPU to generate finer mesh and modified by fragment programs to change its shape. Geometry images can be stored as a floating point texture and accessed by the fragment shader in some recent GPUs. Figure 8.1 shows a subdivided cloth mesh and its corresponding geometry images.

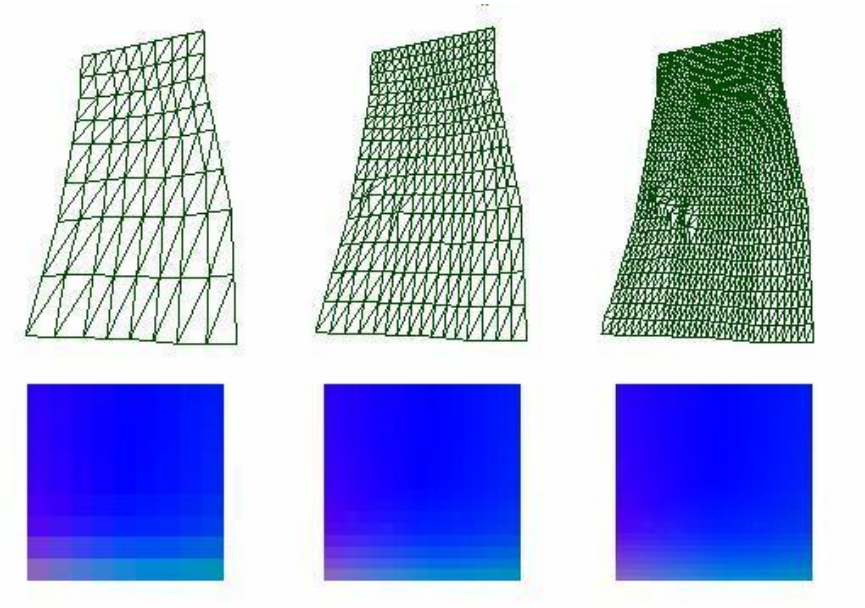


Figure 8.1: Three levels of cloth mesh hierarchy and their corresponding geometry images.

4. *Projective Texture*: Projective texture is a texture lookup method. When using projective textures, the texture coordinate of each object vertex is a function of the vertex position. This function uses a projection matrix to transform the vertex position to the vertex texture coordinate. Projective texture lookups are accelerated by current graphics hardware by a texture matrix.

GPU Techniques

Because of the limitations of current GPUs, it is often tricky to implement some basic algorithms, e.g. sorting and gathering, on the GPU. We describe several such GPU programming techniques that will be used in our GPU based cloth simulation algorithm.

1. *Multipass Rendering*: Render the scene geometry more than one passes on the GPU. During each rendering pass, the GPU reads the scene geometry, processes it with a sequence of operations and writes image into the framebuffer. Multipass rendering enables more operations than a single-pass rendering and enables more flexible memory access by using output image as textures in a new pass. Because limited resolution of the output image and limited bits per pixel, multipass rendering often meet precision problems.

2. *Occlusion_Query*: This is an OpenGL extension supported by current graphics hardware. The application can query the hardware for the number of pixels drawn by a primitive or a group of primitives [32]. It is actually a hardware gathering operation that can only be applied on the output of the fragment processors. This technique provides a way for the CPU to make logic control decisions according to the GPU's calculation result.

3. *Depth-Peeling*: Depth peeling is a technique used on GPUs to sort the objects in a scene according to their depth value [13, 27]. This is a multi-pass approach that extracts the k^{th}-closest objects of a scene in k passes. In the first pass, the scene is rendered and the object with the largest depth is stored as shadow map. In the follow-

ing pass, the scene is rendered again and the depth values of the objects are compared with the shadow map. The objects with greater or same depth as the shadow map are rejected. The result depth image is used as the shadow map in the next pass. The peeling process loops until Occlusion_Query indicates that no fragment is written into the render target.

4. *Render_To_Vertex Capability*: NVIDIA and ATI provided RENDER_TO_VERTEX capability in their newly released graphics hardware [26, 31]. This capability allows the output of fragment programs be used as vertex data, which is the input of the vertex processor. With this capability, the shape of the geometry can be modified locally on the GPU and data transmission between CPU and GPU is reduced.

Bibliography

[1] Kurt Akeley. *3D Graphics Hardware Architecture (SIGGRAPH Conference Real-Time Shading Course Notes)*, 2004.

[2] U. Ascher and E. Boxerman. On the modified conjugate gradient method in cloth simulation. In *The Visual Computer*. Springer-Verlag, 2003.

[3] G. Baciu and W. S. Wong. Image-based collision detection for deformable cloth models. In *IEEE Transactions on Visualization and Computer Graphics*, volume 10, pages 649–663, 2004.

[4] George Baciu and Wingo S. K. Wong. Image-based techniques in a hybrid collision detector. In *IEEE Transactions on Visualization and Computer Graphics*, volume 9, pages 254–271. IEEE Educational Activities Department, 2003.

[5] D. Baraff and A. Witkin. Large steps in cloth simulation. In *Proceedings of ACM SIGGRAPH 98*, pages 43–54. ACM Press, 1998.

[6] D. Baraff and A. Witkin. Rapid dynamic simulation. In D. House and D. Breen, editors, *Cloth Modeling and Animation*, pages 145–173. A.K. Peters, 2000.

[7] D. Baraff, A. Witkin, and M. Kass. Untangling cloth. In *ACM Transactions on Computer Graphics (ACM SIGGRAPH 2003)*, volume 22, pages 862–870, 2003.

[8] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. In *ACM Transactions on Computer Graphics (ACM SIGGRAPH 2002)*, volume 21, pages 594–603, 2002.

[9] R. Bridson, S. Marino, and R. Fedkiw. Simulation of clothing with folds and wrinkles. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA 2003)*, pages 28–36. ACM Press, 2003.

[10] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Computer Graphics*, volume 23, pages 777–786. ACM Press, 2004.

[11] F. Cordier and N. Magnenat-Thalmann. A data-driven approach for real-time clothes simulation. In *Proceedings of 12th Pacific Conference on Computer Graphics and Applications (PG 2004)*, 2004.

[12] J. Eischen and R. Bigliani. Continuum versus particle representations. In D. House and D. Breen, editors, *Cloth Modeling and Animation*, pages 79–122. A.K. Peters, 2000.

[13] C. Everitt. Interactive order-independent transparency. White paper. NVIDIA. 1999.

[14] C. Feynman. Modeling the appearance of cloth. Master's thesis, Massachusetts Institute of Technology, 1986.

[15] Harvey Gould and Jan Tobochnik. *An Introduction to Computer Simulation Methods*. Addison-Wesley, 1988.

[16] Naga K. Govindaraju, Michael Henson, Ming C. Lin, and Dinesh Manocha. Interactive visibility ordering and transparancy computations among geometric primitives in complex environments. In *Proceedings of ACM SIGGRAPH Symposium on interactive 3D graphics and games*, pages 49–56, 2005.

[17] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. Fast collision culling using graphics processors. In *ACM VRST*, 2004.

[18] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha. Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *Graphics Hardware*, pages 25–32. Eurographics Association, 2003.

[19] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer Graphics and Interactive techniques*, pages 355–361. ACM Press, 2002.

[20] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Graphics Hardware*, pages 92–101. Eurographics Association, 2003.

[21] Kenneth E. Hoff III, Andrew Zaferakis, Ming Lin, and Dinesh Manocha. Fast and simple 2D geometric proximity queries using graphics hardware. In *Proceedings*

of ACM SIGGRAPH 2001 Symposium on Interactive 3D Graphics, pages 145–148. ACM Press, 2001.

[22] John Kessenich, Dave Baldwin, and Randi Rost. *THE OPENGL SHADING LANGUAGE. Version 1.10.* OpenGL Architecture Review Board, 2004.

[23] Peter Kipfer, Mark Segal, and R. Westermann. Uberflow: A GPU-based particle engine. In *Graphics Hardware*. ACM SIGGRAPH/Eurographics, 2004.

[24] A. Kolb, L. Latta, and C. Rezk-salama. Hardware-based simulation and collision detection for large particle systems. In *Graphics Hardware*. Eurographics Association, 2004.

[25] T.L. Kunii and H. Gotoda. Singularity theoretical modeling and animation of garment wrinkle formation processes. In *The Visual Computer*, volume 6, pages 326–336, 1990.

[26] F. Losasso, H. Hoppe, S. Schaefer, and J. Warren. Smooth geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 138–145. Eurographics Association, 2003.

[27] Abraham Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. In *IEEE Computer Graphics Application*, volume 9, pages 43–55. IEEE Computer Society Press, 1989.

[28] M. Meyer, G. Debunne, M. Desbrun, and A. Barr. Interactive animation of cloth-like objects in virtual reality. In *Journal of Visualization and Computer Animation*, volume 12, pages 1–12, 2001.

[29] J. Mezger, S. Kimmerle, and O. Etzmuß. Hierarchical techniques in collision detection for cloth animation. In *Proceedings of Winter School of Computer Graphics (WSCG 2003)*, pages 322–329, 2003.

[30] K. Moreland and E. Angel. The FFT on a GPU. In *Graphics Hardware*, pages 112–119. Eurographics Association, July 2003.

[31] NVDIA. *GL_EXT_pixel_buffer_object. NVIDIA_OpenGL_Specs http://developer.nvidia.com/*, 2004.

[32] NVIDIA. *GL_ARB_occlusion_query. http://oss.sgi.com/projects/oglsample/registry/*, 2001.

[33] Matt Pharr and Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, March 2005.

[34] I. Rudomin and J. L. Castillo. Real-time clothing: geometry and physics. In *Journal of Winter School of Computer Graphics*, 2002.

[35] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 146–155. Eurographics Association, 2003.

[36] P. Volino, M. Courchesne, and N. Magnenat-Thalmann. Versatile and efficient techniques for simulating cloth and other deformable objects. In *Proceedings of ACM SIGGRAPH 95*, pages 137–144. ACM Press, 1995.

[37] Jerry Weil. The synthesis of cloth objects. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 49 – 53. ACM Press, 1986.

[38] M. Wloka. *Interactive Cloth Simulation (presentation), http://developer.nvidia.com/object/gdc2001_clothsim.html*, 2004.

[39] Cyril Zeller. *Cloth Simulation on GPU (I3D presentation), http://developer.nvidia.com/object/i3d_2005_presentations.html*, 2005.