# UNIVERSITY OF FREIBURG

## DEPARTMENT OF COMPUTER SCIENCE
## COMPUTER GRAPHICS GROUP

HAPTIC RENDERING

Computer Animation Course
Final Report

**Prepared By**
Nadir AKINCI
Gizem KAYAR

July 23, 2008

## *TABLE OF CONTENTS*

# 1. INTRODUCTION

Haptic, comes from a Greek word (Haphe), means pertaining to the sense of touch [4]. This technology allows users to feel the touch via applied forces, vibrations and motions. In our project, we have used the Phantom Omni Haptic Device produced by SensAble Technologies that is provided by the department.

The stages of our project can be classified into five parts as seen below.

**First stage** loads objects into the scene (see Sec. 2.1).
**Second stage** performs collision detection (see Sec. 2.2).
**Third stage** computes penetration depths for colliding objects (see Sec. 2.3).
**Fourth stage** applies deformable object dynamics (see Sec. 2.4).
**Fifth stage** handles the haptic device - scene interaction (see Sec. 2.5).

Further information about the project such as visualization and challenges confronted are presented in (Sec. 3) and (Sec. 4), respectively.

# 2. STAGES OF THE PROJECT

## *2.1- Loading the Objects*

Loading the objects into the scene is the very first step in the project. Associated *.dcstructure* files of the objects keep the data about object vertices and tetrahedrons. We also added force related constants to the beginning of each file used. However, the number of vertices and tetrahedrons vary from one object to another and can be very large for some objects as illustrated in Table 1. For realistic simulations, we observed that the simulated objects should be composed of nearly same sized tetrahedrons. In Table 1., we listed some of the objects used in our simulations.

| Object | No. of Vertices | No. of Tetrahedrons |
|---|---|---|
| Sphere_hr | 125 | 320 |
| Thickrod | 81 | 160 |
| Cube | 27 | 40 |
| Cow | 3068 | 9873 |
| Sneakers | 166 | 419 |

**Table 1.** Number of vertices and tetrahedrons for different objects

### 2.1.1- Surface Extraction

As traversing around the object, it can be easily seen that inside tetrahedrons have four face neighbors where surface tetrahedrons have less than four. This situation gives us an idea about the algorithm of extracting surface tetrahedrons, which means, the number of face neighbors for each tetrahedron should be found first. We check all the tetrahedrons with each other if they share any common vertex index. If the number of shared vertices is three, then they are face neighbors. After detecting all the neighborhood information, we use the number of face neighbors for any tetrahedron to check whether it is a surface tetrahedron or not.

## *2.2- Collision Detection*

Collision detection is the beginning of the application stage in this project. An efficient collision detection algorithm improves the speed of a simulation dramatically. Hence we have used the algorithm proposed in the paper [1]. According to this algorithm, we have first hashed vertices into the hash grid, and then we have hashed the tetrahedrons. In the penetration depth estimation step, we have also hashed intersecting edges and checked those edges for intersection with the outer faces of the tetrahedrons.

### 2.2.1- AABB Computation

Computing the AABBs of the tetrahedrons provides fast rejection in collision tests and used in hashing procedure of tetrahedrons as explained in Sec. 2.2.2. We have also computed AABBs of intersecting edges to speed up intersection tests.

### 2.2.2- Hashing Procedure

#### 2.2.2.1- Vertex Hashing

Any vertex position (x,y,z) should be first discretized according to algorithm presented in [1]. We have discretized the position using formula seen below.

discretized (x, y, z) = (i, j, k) = ( floor(x / CellSize), floor(y / CellSize),floor(z / CellSize) )

Where CellSize is the grid cell size of the hash grid.

Then, the discretized vertex coordinates are sent to hash function which is presented in (CS1).

```
int getGridCellId(int i, int j, int k)
{
        const float nxy = numCellsX * numCellsY;
        return i + j * numCellsX + k * nxy;
}
```

**(CS1)**

numCellsX is the number of cells in X direction and numCellsY is the number of cells in Y direction.

As a consequence of those steps, the vertex is associated with an appropriate hash cell. After inserting all the vertices into a hash table with corresponding cell value, the second step computes the respective cells for all tetrahedrons as presented in [1].

#### 2.2.2.2- Tetrahedron Hashing

First of all, minimum and maximum values of tetrahedron AABBs are discretized and the associated hash table indices are found. The cells between discretized minimum and maximum hash indices are traversed to find whether there are any vertices in given cell positions or not. If so, intersection test between tetrahedron AABB and vertex is performed. When test finds any intersection between tetrahedron AABB and vertex, then Barycentric coordinates are used for actual vertex in tetrahedron test as indicated in [1]. A vertex is a colliding vertex if it is inside a tetrahedron.

#### 2.2.2.3- Intersecting Edge Hashing

For all intersecting edges, AABBs are computed and hashed into the hash grid. Intersection is checked against only the surface tetrahedrons' outer faces in the corresponding hash grid cell.

## *2.3- Consistent Penetration Depth Estimation*

Computing consistent collision response is one of the major tasks in simulation environments. In our project, the main task is handling the collisions between haptic device and the scene objects. For this reason, after applying collision detection, we have applied the penetration depth estimation method of [2]. The stages of this phase are:

- Detecting colliding points (Sec. 2.3.1).
- Classifying border points, determining intersecting edges, calculating exact intersection points, computing weight functions, corresponding penetration depths and directions (Sec. 2.3.2).
- Propagating penetration depths and directions for all colliding points (Sec. 2.3.3).

### 2.3.1- Detecting Colliding Points

Colliding points are detected using the algorithm we have described in Sec. 2.2. We are keeping a collision state flag for each vertex for this reason. Our vertex state structure is illustrated in (CS2).

```
struct VertexState
{
        vec3* vertex;          // pointer to vertex
        u32 vIndex;             // vertex index
        TetraMesh* mesh;  // pointer to the owner mesh

        bool borderState;    // is this a border vertex?
        BorderVertex* borderVert;      // if true, holds pointer to BorderVertex          (CS2)
        bool colState;         // is the vertex colliding
        Tetrahedron* colTetra;          // if true, holds pointer to the colliding tetra.

        vector<u32> vertexTetras;      // tetras that use this vertex
        vector<u32> vertexNeighbors;              // local vertex neighbors of the vertex
};
```

### 2.3.2- Border Points, Intersecting Edges and Exact Intersection Points

#### 2.3.2.1- Border Points

Border points are the colliding points which have at least one non-colliding vertex neighbor. So as to optimize computations, it is essential to pre-compute the vertex neighbors of each vertex and keep them in a vector. After computing complete vertex neighborhood for the mesh, border points are detected by checking the collision state flags of each vertex neighbors. Once the vertex is detected as a border vertex, its border state flag is assigned true (see (CS2)) and it is pushed to borderVertices vector that keeps the BorderVertex (CS3) objects in the Scene class.

It is important to keep the processed state flag of the corresponding border vertex which means that all the intersecting edges that contain the given border vertex are processed, all the weight related calculations are done and we reach to the final. In order to calculate weight for any given border point, we keep the variables sumLower and sumUpper in our structure in addition to penDepth and penDirection, which is explained in detail in Sec. 2.3.2.3.

```
struct BorderVertex
{
        VertexState* bVertState;  // pointer to the vertex state
        float penDepth;  // penetration depth
        vec3 penDir;  // penetration direction                    (CS3)
        bool processed;  // vertex is processed or not

        float sumLower;      // internal variables
        float sumUpper;
};
```

#### 2.3.2.2- Intersecting Edges

The next step is identifying the intersecting edges whose vertices include one non-colliding and one border point. These edges are computed easily while finding the border vertices. For each intersecting edge, we keep its border and outside vertices, colliding tetrahedron, face normal and exact intersection point informations (CS4).

```
struct InterEdgeTetra
{
        BorderVertex* borderVertex;  // pointer to border vertex
        vec3* outsideVertex; // pointer to outside vertex
        Tetrahedron* tetra; // pointer to the intersected tetrahedron
        vec3 iPoint;  // holds the computed intersection point          (CS4)
        vec3* normal;  // holds pointer to the normal of intersected face
        bool processed;  // all non-degenerate edges are processed
};
```

#### 2.3.2.3- Exact Intersection Points

The final and the most challenging part of this phase for us is finding the exact intersection point between intersecting edges and collided tetrahedrons.

In the beginning of this phase, we have implemented a method which propogates the intersecting edge from its colliding tetra through the surface tetra that contains the exact intersection point. The only disadvantage of this method is traversing all the neighbour tetrahedrons starting from the collided tetra through the final, surface tetra if the penetration is too deep. Algorithm had to make a decision as picking the proper next tetra through the surface. While trying to find the exact intersection point, we have confronted with many degenerate cases if the intersecting edge is going completely through an arbitrary edge or through the corner vertices of the collided mesh.

After some time, we have changed our method to a more stable and comparatively fast version. In this method, firstly, we find the AABB of the corresponding intersecting edge and we detect all the grid cells that contain this AABB. In the second step we search all the tetrahedrons that lie inside these grid cells and we consider any tetrahedron only if it is a surface tetrahedron. We check these surface tetrahedrons' outside faces to find the exact intersection point so we don't need to handle any degenerate cases that we had to handle as in the previous method.

Once we find the exact intersection point of the intersecting edge which means the edge is processed, we calculate the weight function for the associated border vertex and the exact intersection point, to approximate penetration depth and direction. We use sumLower and sumUpper variables for weight accumulation purposes.

After calculating each sumLower and sumUpper values for corresponding border vertex and its intersecting edges, then we compute the final penetration depth and direction as presented in [3] and set the vertex as processed.

### 2.3.3- Propagation

In propagation phase, all the colliding but not processed, non-border vertices are processed. Hence, any colliding point with a processed border vertex neighbor is considered as a new border vertex and all the steps that explained in Sec. 2.3.2 (except edge intersection) are carried out on them also. At the end of this stage, all colliding points have a penetration depth and direction.

## *2.4- Dynamics*

We also added mass-spring object dynamics to our project and employed the following features

- Gravitational force
- Spring forces
- Volume preservation force, using the method of Teschner et al. [3]
- Verlet / Euler-Cromer integration schemes
- World collision and friction

## *2.5- Interaction*

We used two callback functions for haptic control. One is for getting the position of the haptic device, which is called by the user. And the other is an asyncronous callback which is running in another thread applies force to the haptic device continuously. Our haptic force computation methodology is accumulating all penetration forces applied to the haptic object (which can be an arbitrary object) from the other objects.

If haptic device is not connected, mouse emulation mode is activated automatically. In mouse emulation mode, mouse movement results in motion in xy plane where the mouse wheel results in motion in z direction.

The keyboard mapping of the program functions are as follows;

| | |
|---|---|
| **Up** | Moves camera forward. |
| **Down** | Moves camera backward. |
| **Left** | Rotates camera left. |
| **Rİght** | Rotates camera right. |
| **A** | Slides camera left. |
| **D** | Slides camera right. |
| **Enter** | Simulation switch. |
| **Backspace** | Enables visual debugging mode. |
| **W** | Wireframe mode switch. |
| **H** | Hidden mode switch (for debugging purposes). |
| **L** | Lighting switch. |
| **G** | Voxel grid drawing switch. |
| **B** | AABB draw switch. |
| **P** | Switches to mesh picking mode (after picking the object, switches back to normal). |
| **V** | Switches to four-view mode (for scene building purposes). |

# 3. VISUALIZATION AND ADDITIONAL FEATURES

We have used OpenGL for visualization. For setting up OpenGL, we have used the library GLFW (An OpenGL Framework) which is also a nice framework like GLUT. We directly visualized outer faces of the tetrahedrons in our simulations inside a simple rectangular world. Additional features of our program can be listed as;

- User is able to position the objects in the scene using the program itself and can save up to 10 scenes for later use. A scene builder mode is created for that purpose. The saved scenes are kept in discrete scene files for later use.
- User may pick any arbitrary object as the haptic object in the simulation moment.
- An object may be flagged as passive which means that it is not simulated.
- Border points, penetration vectors, wireframe meshes, AABBs of tetrahedrons and also hash grid itself can be visualized by enabling the visual debugging mode of our program.

# 4. CHALLENGES CONFRONTED

The most time consuming part of this project was finding exact intersection points which is explained in detail in Sec. 2.3.2.3. Our first method for this stage caused many degenerate cases and we spent more than two weeks to be able to handle them. At the beginning, we tried to solve this problem by adding epsilon values to some critical calculations that compute line-triangle intersections. This solution to the problem had cut off up most of the degenerate cases but caused new ones. We have also tried those critical calculations in double precision but it did not help either. Hence we completely written a new method which handles all the degenerate cases (explained in Sec. 2.3.2.3).

## 5. REFERENCES

[1] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, M. Gross, "**Optimized Spatial Hashing for Collision Detection of Deformable Objects**," *Proc. Vision, Modeling, Visualization VMV'03,* Munich, Germany, pp. 47-54, Nov. 19-21, 2003
[2] B. Heidelberger, M. Teschner, R. Keiser, M. Mueller, M. Gross, "**Consistent Penetration Depth Estimation for Deformable Collision Response,**" *Proc. Vision, Modeling, Visualization VMV'04*, Stanford, USA, pp. 339-346, Nov. 16-18, 2004.
[3] M. Teschner, B. Heidelberger, M. Mueller, M. Gross, "**A Versatile and Robust Model for Geometrically Complex Deformable Solids**," *Proc. Computer Graphics International CGI'04*, Crete, Greece, pp. 312-319, June 16-19, 2004.
[4] http://en.wikipedia.org/wiki/Haptic