

Praktikum Computer Animation

Hannes Schulz*

July 26, 2008

*University Freiburg, ACS, Matr. 2411732, schulzha@informatik.uni-freiburg.de

Contents

1	Introduction	3
2	Tetrahedron Meshes	3
2.1	Tetrahedron Mesh Representation	3
2.2	Algorithms on the Mesh Representation	6
3	Collision Detection	6
4	Collision Depth Estimation	6
5	Haptic Device Integration	8
6	Results	9
7	Conclusion	9

1 Introduction

In contrast to traditional computer input devices like keyboards and mice, haptic devices are much more fit to a 3D-environment in at least two ways. First, haptic devices provide a 3D-cursor which the user can move through 3D space. Mapping displayed to and from work space is made easier for the user than when she is using a mouse/keyboard mappings. Second, haptic devices help to create realistic simulations. They enable us to simulate part of our 3D world by providing intuitive tools to manipulate the simulated world.

A key feature for realistic simulations is the force-feedback functionality. The haptic device feels like some tool only if it interacts with the simulated environment in the way this particular tool would. It is therefore crucial to be able to determine the forces acting on a haptic device in a simulated environment, especially if this environment is non-trivial and/or changing.

This work describes an implementation of the components vital to creating a responsive 3D simulation for a haptic device. Section 2 deals with the efficient representation of tetrahedron meshes using templates and reverse indices. Section 3 describes how to find collisions using a hashing method. These collisions are then resolved by determining a consistent penetration depth and calculating an associated force, which is described in Section 4. Finally, in Section 5, we describe how to employ the calculated information to create an environment controllable through a haptic device.

2 Tetrahedron Meshes

2.1 Tetrahedron Mesh Representation

The tetrahedron mesh representation we use is similar to the one described in Celes et al. [2005].

The tetrahedron mesh is represented by objects of the class `Node` and `Tetra4`, which we will describe in the following.

A `Node` is the representation of a point in 3D. Each `Node` knows one associated `Tetra4` and its index within it. It may have additional attributes for rendering. We keep one node array per mesh, which contains each node necessary to describe the mesh exactly once.

A `Tetra4` is the representation of a tetrahedron. A tetrahedron consists of four 3D points. The storage of them would be redundant, so instead, we store references to the `Nodes` in the aforementioned array. We additionally store adjacency information within a `Tetra4`. As described in Celes et al. [2005], this information can be reduced to references to neighbouring tetrahedrons and two reverse indices. The latter can be squeezed into one byte. The tetrahedra are stored in one array per mesh.

We will now explain the critical concepts of our representation.

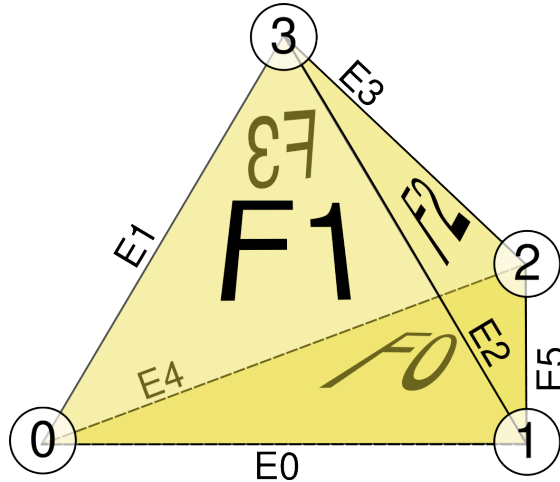


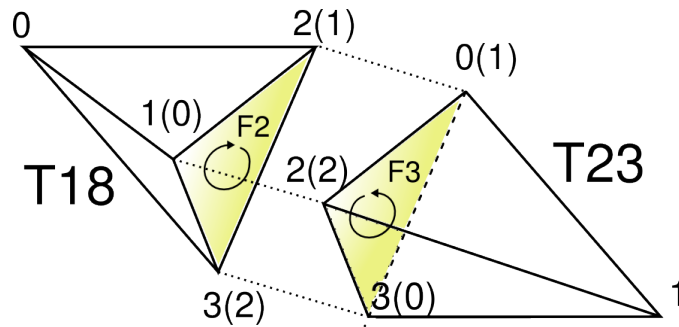
Figure 1: The **Tetra4** template. The order of nodes (0,...,3) determines the order of faces (F1,...,F3) and edges (E0,...,E5). The adjacency relationships within the tetrahedron are thus fixed.

Templates We can make use of the order of references to nodes stored in a **Tetra4**, which can provide us with unique references to faces, vertices and edges at no additional storage cost. To do so, we create a template of a tetrahedron by first, drawing a tetrahedron and second, numbering all vertices, faces and edges. The result is depicted in Figure 1. From this, we can derive adjacency information. For example, edge E0 is always connected to the faces F1 and F0. The template is represented as series of two-dimensional arrays mapping for example a face-ID to the IDs of vertices adjacent to the face. Please note that this information independent of any tetrahedron object.

Entity Uses We can now pass around references to faces, vertices and edges using a minimal representation, namely, a pointer to a **Tetra4** and a small number (in $\{0, \dots, 5\}$) denoting the index of the entity within this **Tetra4**. If, for example, we want to represent the face which is denoted by the 0th, first and third node (F1) referenced by this tetrahedron, the reference is a pair consisting of a reference to the tetrahedron and the number 1. These pairs are called entity uses, where an entity may be a face, vertex or edge.

Adjacency of Tetrahedra Within a **Tetra4**, adjacency is determined by the template. **Tetra4s** keep references to adjacent tetrahedra ordered by the number of the face they share. This information is not sufficient for quick access, though. For example, if we want to find the use of the face F0 in **Tetra4** t_1 in the neighbouring **Tetra4** t_2 , we would need to check the faces F0 to F3 of t_2 for equality with F0 in t_1 . As suggested in Celes et al. [2005], we shorten this process by remembering the ID of the particular face in t_2 which corresponds to F0 in t_1 . These IDs are called “reverse indices”.

Adjacency of Vertices We can derive even more information if we assume that in the template, vertices around a face are always listed clock-wise, as seen from the outside of the tetrahedron. This condition can be ensured when initializing the tetrahedron by switching the order of nodes if necessary. Then, the order of nodes in the neighboring tetrahedron t_2 are listed in reverse order to the nodes in our tetrahedron t_1 . Additionally, the second listing can be ring-shifted by some offset between zero and two. We now also remember this offset of nodes in F0 in t_1 to the nodes in the corresponding face in t_2 . As a result, we can directly (that is, in constant time) access the vertex-use vu of a vertex v in a neighbouring tetrahedron with respect to the face f using a function as depicted in Figure 2. This rotational offset nicely fits into the same byte which stores the ID of the adjacent face described above.



```

inline VertexUse Tetra4::getVertexMatch(int v, int f){
    Tetra4* neighbour = neighbours[f];
    char v_id_in_neighbouring_face =
        (3 - faceVertexNr[f][v]
         + nodeOffset[f]
        )%3;
    char v_id_in_neighbour =
        elemVertexNr[neighbouringFaceNr[f]][v_id_in_neighbouring_face];
    return VertexUse(neighbour, v_id_in_neighbour);
}

```

Figure 2: Bottom: Algorithm to determine the vertex-use corresponding to vertex v in a neighboring tetrahedron with respect to face f . Here, `neighbours` contains references to the neighboring tetrahedrons, `faceVertexNr` maps the vertex number ($0 \dots 3$) to the number of a vertex in a face ($0 \dots 2$), `elemVertexNr` does the reverse of `faceVertexNr`. The reverse indices are accessed via `nodeOffset` and `neighbouringFaceNr`. Top: Illustration of the above algorithm. Face F2 in T1 corresponds to F3 in T2 template. The face vertex numbers are written in brackets, the element vertex numbers without brackets. The order of face vertices is clockwise and denoted by circular arrows. The node-offset in this example is 2.

2.2 Algorithms on the Mesh Representation

It is now quite easy to specify algorithms which make use of the described representation. For example, we can search for all tetrahedrons which make use of a `Node` n by first, accessing the `Tetra4` t which n refers to, and second, iterating over all faces adjacent to n in t , restarting this process with the neighbours which we haven't seen yet. We implemented the enumeration of all facets around a vertex, all vertex uses of a vertex and all edges using a vertex. Since edges are used quite often (many edge uses for one edge), we also implemented a version where edges are only enumerated once. We decided to hide the enumeration details from the user using template callbacks. To demonstrate the capabilities of our implementation, we determined the normals of boundary nodes by averaging the normals of their adjacent faces. The necessary code (see `determineBoundary()`) is very intuitive. Additionally, the calculation smoothes the collision response results described below.

3 Collision Detection

Collisions are detected exactly as described in Teschner et al. [2005].

First, for each node $n = (x, y, z)$, the grid-cell is determined as

$$(\lfloor x/g_x \rfloor, \lfloor y/g_y \rfloor, \lfloor z/g_z \rfloor) = (i, j, k).$$

A hash-value for the grid cell is determined as

$$h = (i \cdot 73856093) \text{ xor } (j \cdot 19349663) \text{ xor } (k \cdot 83492791).$$

In a hash table, at position h , a reference to n is saved.

In a second step, for all tetrahedra t , all grid cells affected by the axis-aligned bounding box of t are traversed. The nodes at the gridcell-position in the hashtable are then checked whether they are inside the tetrahedron t as follows. First, since we do not know whether the nodes in the hashtable are from the current grid cell or another one with the same hash value, we have to check whether the node is in the grid cell at all. Second, we compute the barycentric coordinates of the node w.r.t. the tetrahedron. If it is inside, we remember the collision.

4 Collision Depth Estimation

For collision depth estimation, we implemented two algorithms. First, we implemented the extension of the collision detection algorithm presented above, as described in ?. Second, we implemented a variant which builds on the same ideas, yields exactly the same results, but is both simpler and faster than the mentioned method.

Our tests suggest that our method is superior to the one described in Heidelberger et al. [2004]. We will therefore only describe our method. We exemplarily visualize some important concepts in Figure 3.

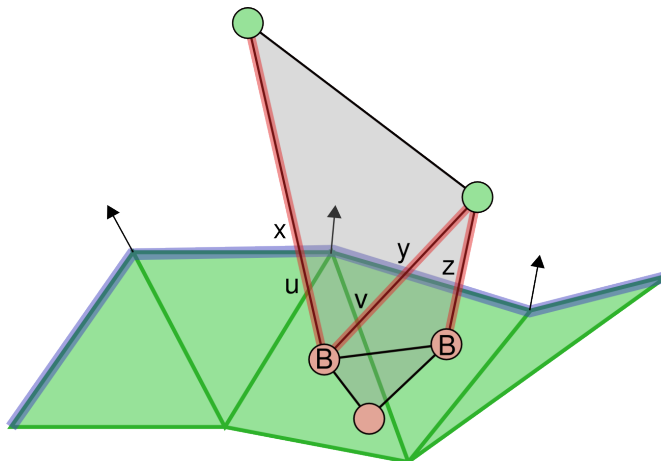


Figure 3: Collision Depth Estimation. Red circles denote nodes in collision. Nodes marked with “B” represent boundary nodes. Red lines denote boundary edges. The blue line is the object boundary of the green object, with arrows denoting the object normal. The intersections x , y and z determine the depth of the boundary points. To find them, the boundary edges are traced from the boundary nodes to the surface, if necessary via the intersections u , and v . At the barycentric coordinates of x , y and z the surface normals are averaged.

We start with the set of collisions from Section 3, that is a list of pairs (t, n) consisting of a tetrahedron t and a node n which lies inside t . We now iterate over all edges exiting n using the method described in Section 2.2. All edges which contain both, a node in collision and a node not in collision, are flagged as boundary edges, their colliding node is flagged as a boundary node.

We can now determine the depth of the boundary nodes n . In Heidelberg et al. [2004] this is done by hashing edges and boundary faces. We do not use hashing. Instead, we trace the boundary edges e to the surface by checking for intersections of e with all faces of t , the tetrahedron which n collided with. If an intersection with a boundary face is found, we can estimate the direction of the normal using barycentric coordinates of the intersection point as described in Heidelberg et al. [2004]. If, on the other hand, the intersection with a non-boundary face is found, we determine the connecting tetrahedron and restart the process, ignoring the face where we came from.

Once the depth of all border points is determined, we can propagate the depth information exactly as in the original algorithm.

The advantage of our method is that we do not depend on hashtable size or the number of boundary faces in the scenery. Instead, we can determine the collision depth in a time proportional to the number of collisions. Other limiting factors are the maximum collision depth and the distance of the boundary nodes (measured in tetrahedrons) to the surface.

5 Haptic Device Integration

The haptic device used for this work was a PHANTOM Omni device from SensAble Technologies. The device comes with two development APIs, one of which is very high-level and does collision response by itself. The low-level API was used for this work.

The documentation of the device provides code snippets for initialization and setup of the device as well as hints on how to efficiently send and receive force and position information. Speed is crucial, since delays result in non-intuitive response of the device, which is easily detectable by the human device handler. In the following, we will describe the points where we deviate from the documentation or where the documentation does not provide ready-to-go examples.

Our visualization framework is OpenGL with GLUT. As commonly chosen, visualization and force rendering is controlled using the idle-callback. Here, we chose to calculate new force twice as often as (visual) rendering takes place. The calculated force is placed in a global variable, where the haptic device callback function (which is called asynchronously) can pick it up for force rendering.

Before force calculation/visualization takes place, the new position of the device in the scenery has to be determined. Here, two options exist. One may either rotate the camera independent of the device movements (allocentric pen coordinates) or move the pen with the camera (egocentric pen coordinates). We chose the latter variant, since the device is most easily controllable in egocentric coordinates. For the implementation we need to know the current position and rotation of the camera C in homogeneous coordinates, which we can derive by multiplying the current model-view matrix M and the current trackball matrix T each time the camera position or orientation changed:

$$C = M \cdot T$$

We can then assume that the coordinates for the pen as told by the device are egocentric coordinates P_e . To determine the allocentric coordinates P_a of the pen necessary to draw the pen at the correct position in the scenery, we transform the coordinates using C :

$$P_a = P_e \cdot C$$

Of course, the forces calculated using the method described above must be converted similarly to become intuitive in egocentric space.

To support the illusion of a solid object, we move the pen according to the forces acting on it before drawing. Thus the response force increases when the user pushes harder, but there is no visual penetration.

6 Results

For our tests we used two objects, a virtual pen resembling the haptic device and a test object.

A simple cube served as a test object first. Here, the averaging of surface normals has to be turned off. The device response is quick and intuitive. For our other, more complex test objects (a cow and a teddy) we did not find a drop in performance.

It has to be noted, though, that the consistent penetration depth approach yields counter-intuitive results when the pen penetrates the object sideways. This is especially the case, when the tip or top of the pen is completely outside the object, and the long side of the pen is slowly pushed towards the object. As soon as the thin pen is completely inside the object, there is a strong force along the pen, while before there was only a soft force towards the side of it before. This becomes especially problematic when, as described above, the penetration is not visualized to create the illusion of a solid object. In this case, a simple shortest-distance-to-surface approach would have yielded more intuitive results.

There are at least two possible ways to solve this problem. First, one could calculate the response with respect to an object which is not displayed and has better properties than the pen, for example a sphere. The pen could then be drawn for visualization purposes. The sphere would need to be large enough to allow deep penetrations, which in turn means that fine structures cannot be explored anymore. Second, the entry point could be remembered and response forces projected onto the vector to the entry point. This yields a totally different approach than the one implemented in this work.

7 Conclusion

In this work, we implemented the necessary tools to create 3D sceneries with interacting objects. A crucial design criterion here is speed. With speed in mind, a small and fast mesh representation was selected. A hashing approach was chosen to detect collisions. We further implemented a method to determine a consistent penetration depth from the detected collisions and were able to simplify and speed up the implemented methods. Finally, we could demonstrate the realtime capabilities of our implementation using a haptic device. We find that the approach of consistent penetration depth estimation works well, but has problems with long, thin objects where more effort is needed to yield truly intuitive results.

References

Waldemar Celes, Glaucio H. Paulino, and Rodrigo Espinha. A compact adjacency-based topological data structure for finite element mesh representation. 64:1529–1556, 2005.

Bruno Heidelberger, Matthias Teschner, Richard Keiser, Matthias Mller, and Markus H. Gross. Consistent penetration depth estimation for deformable collision response. In Bernd Girod, Marcus A. Magnor, and Hans-Peter Seidel, editors, *VMV*, pages 339–346. Aka GmbH, 2004. ISBN 3-89838-058-0. URL <http://dblp.uni-trier.de/db/conf/vmv/vmv2004.html#HeidelbergerTKMG04>.

Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, Marie-Paule Cani, Franois Faure, Nadia Magnenat-Thalmann, Wolfgang Straer, and Pascal Volino. Collision detection for deformable objects. *Comput. Graph. Forum*, 24(1):61–81, 2005. URL <http://dblp.uni-trier.de/db/journals/cgf/cgf24.html#TeschnerKHZRFCFMSV05>.