Technische Universität Kaiserslautern
University of Kaiserslautern

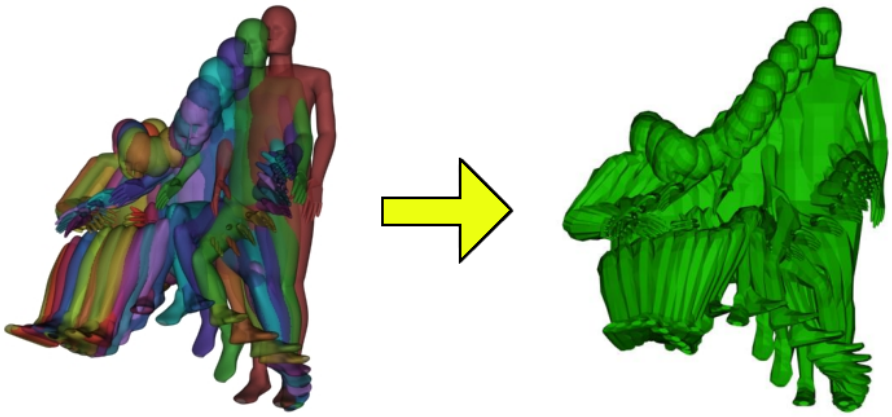
Fachbereich Informatik
Department of Computer Science

Bachelorarbeit

Merging Triangulated Meshes

von

Andreas Berres
a_berres@informatik.uni-kl.de



Erster Fachprüfer:	Prof. Dr. Hans Hagen
Zweiter Fachprüfer:	Juniorprof. Dr. Achim Ebert
Betreuer:	Dipl.-Inform. Andreas Divivier
Eingereicht am:	2. Juni 2009

Bachelor Thesis

Merging Triangulated Meshes

Andreas Berres

2. Juni 2009

Abstract

This bachelor thesis deals with the problem of merging triangulated meshes. The input data is provided by RAMSIS, a system that is used for vehicle occupant simulation. It consists of humanoid models, which are divided into groups for different limbs.

The algorithm to solve this problem takes two objects in the `obj` file format and produces a third, merged object, which consists of the outer hull of the union of both objects, but does not contain any triangles inside its hull. All triangles from one object, which intersect the other object, are subdivided. Numerous types of intersections will have to be discussed for this. Then, all subtriangles that lie on the hull of the merged object are kept, whereas the ones inside the merged object are removed.

To implement this decision, the algorithm classifies vertices according to their location, which signals, where the vertex lies with respect to the object it does not belong to. Based on this classification, we can then decide which triangles to save and which to discard.

Diese Bachelorarbeit befasst sich mit dem Problem der Verschmelzung von Dreiecksnetzen. Die Eingabedaten werden von RAMSIS, einem System zur Insassensimulation für Fahrzeuge, bereitgestellt. Sie bestehen aus menschenähnlichen Modellen, die in Gruppen für verschiedene Körperteile unterteilt sind.

Der beschriebene Algorithmus bekommt als Eingabe zwei Objekte im `obj`-Format und erzeugt ein drittes, verschmolzenes Objekt, welches aus der Hülle der vereinigten Objekte besteht und keine Dreiecke innerhalb seiner Hülle enthält. Alle Dreiecke eines Objekts, die das andere Objekt schneiden, werden unterteilt. Hierzu müssen wir eine Vielzahl an verschiedenen Fällen berücksichtigen. Dann behalten wir alle Teildreiecke, die zur Hülle des verschmolzenen Objekts gehören und entfernen jene, die innerhalb des Objekts liegen.

Während der Berechnungen klassifiziert der Algorithmus alle Knoten nach ihrer Orts-Markierung. Diese signalisiert, wo sich der Knoten im Verhältnis zu dem Objekt, zu dem er nicht gehört, befindet. Aufgrund dieser Klassifizierung können wir nun entscheiden, welche Dreiecke wir speichern, und welche wir verwerfen wollen.

Ich erkläre hiermit, die vorliegende Bachelorarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, den 2. Juni 2009

Contents

1 Motivation

The purpose of this bachelor thesis is to merge triangulated meshes, in a similar way to the union operator for Constructive Solid Geometry (CSG)[3, p. 93]: the outer hull of the union of two or more bodies (or meshes in our case) is kept, while all inner parts are removed. A simple example of this is shown in Figure 1.1. However, opposed to CSG, the objects to be dealt with are not solid but only the hull representations (i.e. the skin) of geometric objects. Therefore, the result should also be a hull.

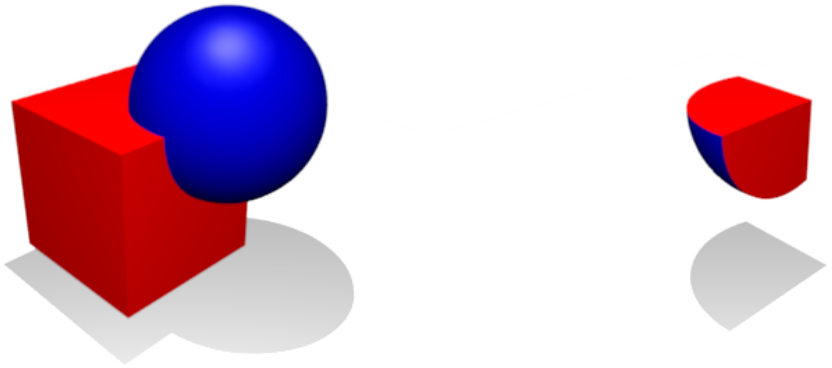


Figure 1.1: $A \cup B = (A + B) \setminus (A \cap B)$, where $A + B$ contains all elements of both meshes/bodies and $A \cap B$ contains the common elements. (*Image source:* http://en.wikipedia.org/w/index.php?title=Constructive_solid_geometry&oldid=279820325)

This union operation is applied to humanoid meshes that are used by the RAMSIS system. The RAMSIS project was initiated and funded by the entire German automotive industry [2, p. 4] and allows engineers to generate realistic CAD representations (manikins) of the human body. **RAMSIS** is an acronym for **Rechnergestütztes Anthropologisch-Mathematisches System zur Insassen-Simulation** (Computer Aided Anthropological Mathematical System for Occupant Simulation) [2].

During the manikin generation process, the relevant body dimensions, namely body height, ratio of sitting height over body height ("proportion") and waist circumference, can be adjusted and the remaining measurements are calculated accordingly, based on correlations in the built-in anthropometric databases [7, pp. 4–5]. This way, various

different body types can be generated, as depicted in Figure 1.2. The generated manikins can then be used to take advantage of the RAMSIS system's main feature, the Automatic Posture Calculation. The user can define various tasks like "put the left foot on the clutch pedal" or "look at the rear mirror". Then, an optimisation algorithm calculates how a human would most probably carry out the tasks [2, p. 9].

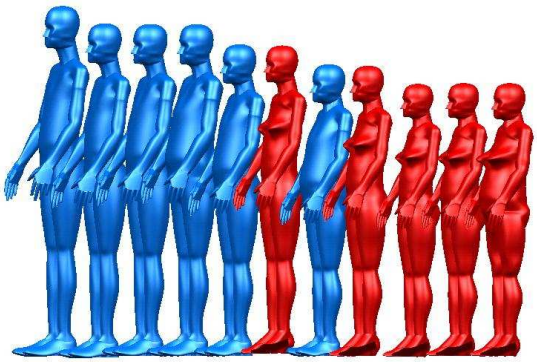


Figure 1.2: Manikins with different physical features; the blue ones are male and the red ones are female. (*Image source:* [2])

To improve interior vehicle ergonomy, the RAMSIS users have to know how much space several vehicle occupants need for carrying out typical tasks, such as those mentioned earlier. For this, a set of manikin postures for carrying out these typical tasks are created. Then, the skin meshes for the manikins in their corresponding postures are merged to compose a larger united object. Here, more intermediate movement objects lead to a smoother representation of movement and, when manikins are merged, to a less bumpy surface.

All manikin objects are divided into 52 subgroups, which contain one body part each. Figure 5.1 illustrates, how the manikins are subdivided. In Section 5.1.2, we will see, how this division can be of use for improving the performance of the algorithm.

The task for this work was to develop an algorithm, which can merge triangulated meshes. In this work, I will describe my approach to solve this problem. I will start by giving a short overview over my algorithm and relevant background information in Chapter 2 and summarising related works in Chapter 3. Chapter 4 describes the theoretical foundations used for the merging algorithm. In Chapter 5, the various parts of the algorithm are described in detail and Chapter 6 contains a description of the main classes of the program. Finally, Chapter 7 shows the results obtained with the algorithm, describes the problems that had to be solved, and it gives an outline of possible topics for future works.

2 Introduction

There are a few things that should be clarified before we begin. The following sections will define terminology (Section 2.1), give an outline of the merging algorithm (Section 2.2) and specify properties, which meshes have to fulfil (Section 2.3). Afterwards, Section 2.4 will give a brief introduction to the file format we use.

2.1 Terminology

Let us begin by introducing some terms, which are used throughout this thesis.

There are three data structures:

Vertex is a data structure which is used for both 3D points and triangle's corners.

Triangles consist of three vertices. They have a centroid, a normal and an area, which is greater than zero.

Vector is a 3D vector, which is defined by its direction, orientation and length.

The vertex class is used for a number of different purposes. To make it easier to distinguish between them, let us use the terms as follows:

Vertex is used for triangle's corners.

Point refers to a newly calculated point, which may or may not become a triangle's corner vertex.

Corner is an alternative name for vertex.

Centroid is a triangle's barycenter. It is computed from the corners A, B and C as $\frac{A+B+C}{3}$.

The amount of data that has to be handled is large and it may vary. I could not simply use arrays to store the vertices and triangles, because this would have resulted in having to change array sizes, leading to bad performance. Instead, I needed a list-like structure, which allows us to append elements at any time. The Standard Template Library (STL) provides a couple of structures resembling lists, including `list` and `vector`. According

to [1], the `vector` allows to access members more quickly than the `list` and both can append elements in constant time, so I decided to use the `vector`. Since I have a `Vector` class to use for 3-dimensional vectors, it would be confusing to use the term "vector" for both. Instead, I will use the term

Vector for the 3D `Vector` class, and

List for STL vectors.

These lists always store pointers to objects. However, writing "the pointer to the vertex" instead of "vertex" on every occasion would make this work quite uncomfortable to read, so **list** actually means the pointer to the object.

In the previous pages, we already saw, that there are objects, which are divided into groups. Naturally, both are meshes, but the term **mesh** is a bit too generic, so let us refrain from using it too often. When it is used, it refers to the whole objects rather than only a group.

The **bounding box** of an object is the smallest cuboid with edges, that are parallel to the axes of the coordinate system, which contains the object.

2.2 Main Idea

The main idea of the merging algorithm is to do exact calculations of intersections and to use tags for vertices and triangles, telling the program how to handle them.

Vertices have one of four location tags, which are calculated during the *in-out-test* in relation to the object they do not belong to: **out** if they are outside, **on** if they are on the surface, **in** if they are inside, and **unknown** as an initial state.

Triangles only have two status tags: **check** if we assume they will be kept and **remove** if we know we do not need them for the final object.

The intersection test, which checks whether or not two triangles intersect, is optimised using bounding boxes, the manikins' subdivision into groups, and the triangle statuses. If it is conducted, we first calculate and evaluate the triangles' intersection points. Then we triangulate both triangles, depending on the number of intersection points and where they lie with respect to the triangle. Finally, we classify the new triangles. The flow chart in Figure 2.1 visualises this procedure.

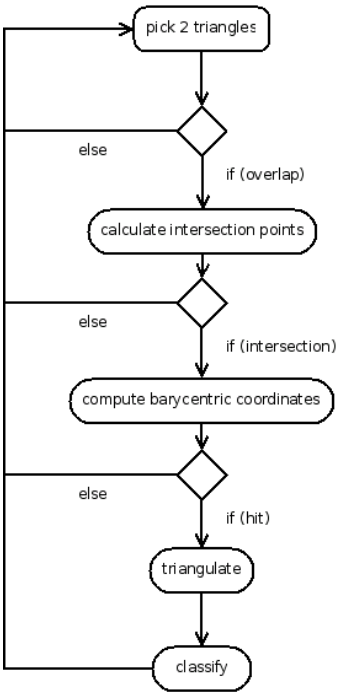


Figure 2.1: Flow chart for the core of the algorithm.

2.3 Required Mesh Properties

It is sufficient to provide an algorithm for two meshes, because if there are more meshes, we can start with two, and then merge the remaining meshes iteratively. However, in order to allow this, the algorithm requires certain properties of the mesh:

Coherence The surface of the objects is coherent[1] iff it does not contain any holes and there are no degenerated triangles.

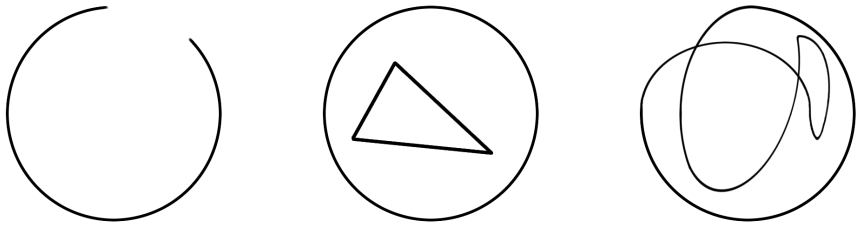
Hollowness There are no triangles inside the outer hull of the objects.

Uniqueness Each point of the object lies on exactly one face, edge or point; there are no overlaps between surfaces.

Figure 2.2 depicts examples violating these properties. The object in Figure 2.2a has a hole, so it is not coherent. The one in Figure 2.2b contains a triangle inside its hull, hence it violates the hollowness property. Figure 2.2c is an example of self-penetration:

[1] There can be several objects, which are not connected, but every separate part has to be coherent in itself.

there are two points in which the object penetrates its own boundary, so these points belong to more than one element of the object at the same time and are therefore not unique. Additionally, it is not hollow, either.



(a) Not coherent. (b) Not hollow. (c) Not unique.

Figure 2.2: Examples of property violations.

2.4 The obj File Format

The `obj` format is an easy-to-handle, open file format, which was introduced by Wavefront and which is now supported by many 3D modelling applications. There is a human-readable ASCII and a binary version of the format [8]. We use the ASCII version since it is more intuitive.

In general, a file has the following structure:

```
# Simplified obj file for a tetrahedron
 o Tetrahedron
 v  1.0 -1.0 -1.0
 v -1.0 -1.0  1.0
 v  1.0  1.0  1.0
 v -1.0  1.0 -1.0
 f 1 3 4
 f 2 4 3
 f 1 4 2
 f 1 2 3
```

The first character of each line gives the type of the element to be represented. It is succeeded by one or more elements, which contain the values, and which are separated by whitespaces.

A new object is declared as `o` <*objectName*>. Alternatively, a group can be declared as `g` <*groupName*>. Both versions can be used later in the program.

r0.4

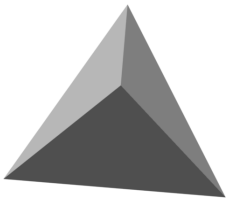


Figure 2.3: Rendered tetrahedron.

`v` x y z creates a vertex with the coordinates x, y and z.
`f` a b c creates a face with a, b, c as vertex numbers for vertices A, B and C.

Both vertices and faces are numbered consecutively. Please note that counting starts at 1 (for both vertices and faces), because we will need an offset with respect to vertex indices when creating new `obj` files, as well as corrections of the index during parsing.

The overall order of vertex and face definitions is arbitrary. However, the order of vertex coordinates is fixed, and the order of vertices for each face is restricted to a counter-clockwise order, as it defines the normal.

Beyond that, the `obj` format provides a lot more functions, such as assigning materials, defining vertex normals and texture vertices etc. We do not use these functions, as they are not relevant for our algorithm.

3 Related Works

Smith and Dodgson [6] use a very interesting approach in their paper about merging polygonal (rather than triangulated) meshes:

Their approach and mine comply with each other in so far as "[t]he operations at the heart of the basic Boolean operation are tests that determine, [...] whether a vertex of one input structure lies in the interior of the other input structure, and whether an edge of one input structure intersects a [face] of the other." [6, p. 150].

Another similar component to our method is the set of restrictions they apply to their meshes:

Shape boundary closure There must not be any breaks in the boundary surface. This corresponds to our algorithm's *coherence* property.

Shape enclosure Intersecting boundary components are forbidden as they can cause self-penetrations and elements which lie inside an object. This is equivalent to our *uniqueness* and *hollowness* properties.

However, the way in which these operations are implemented differs strongly. The authors break down the problem from 3D into 2D and 1D space. They divide their algorithm's operations into seven levels, as depicted in Figure 3.1. The third level operations determine, "whether a vertex of one solid lies inside or outside the other solid, and whether an edge of one solid intersects a [face] of the other solid" [6, p. 153], so they are the equivalents of our algorithm's *in-out-test* and *intersection test*.

Level 0 to 2 are the basis for level 3: On level 2, two elements are projected down into the xy-plane and compared in 2D space. It is used to test, whether two edges' projections intersect and if a vertex lies on a face. Level 1 only considers the x value and its operations tests, whether two edges overlap and whether a vertex lies on an edge. Level 0 is rather irrelevant, since all vertices coincide in 0D space, but mentioned nonetheless.
Levels 4 to 6 contain operations, which determine the resulting structure dependent on the operator (union, intersection or difference) that is to be used. After these steps, the polygons have to be subdivided into triangles along so-called *internal edges* (newly created ones, which run through the polygon's face) if a triangulated mesh is required [6, p. 158].

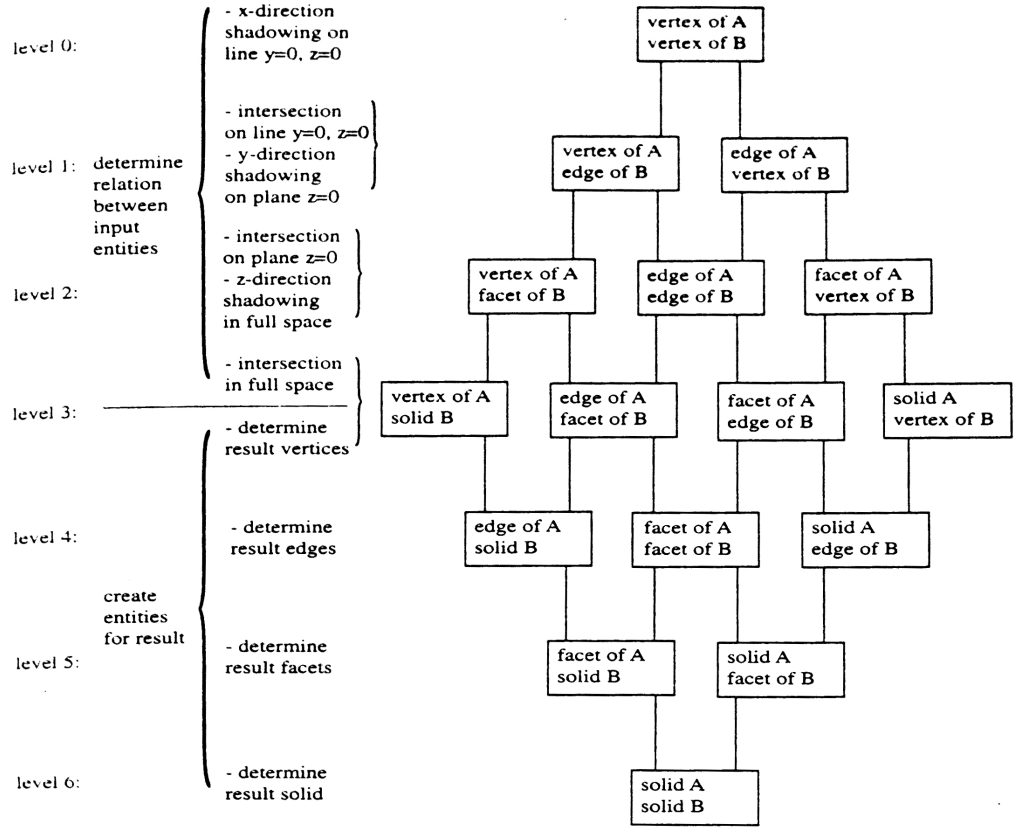


Figure 3.1: Smith and Dodgson's operation hierarchy. Every pair of elements sharing a box is compared with each other.

Through this hierarchical organisation, the authors can reduce search space:
For the in-out-test this means, that if a vertex does not lie inside the x-interval of an edge, it will not lie within the face that was reduced to this edge. However, if it does, we can test if its projection on the xy-plane lies inside the projection of the face. And if the vertex passes both stages, we can check the vertex location in 3D space.
For the intersection test, it means, that if two edges, which are projected onto the x-axis, overlap, we proceed to level 2. If their projections onto the xy-plane also intersect, we also test the 3D case.

After the algorithm, a "data-smoothing post-process" is applied. It simplifies the mesh and removes both gaps and slivers. However, Smith and Dodgson also note that this process is not fully robust as it can both fail to resolve arithmetic errors or fail to terminate [6, p. 162].

Smith and Dodgson claim that their method "is an advance because of the guarantee

of correct connectivity in the result of the boolean operation" [6, p. 150]. This suggests that previous approaches to solve this task had even bigger problems.

4 Theoretical Foundations

Before immersing ourselves in the algorithm, let us first lay down the foundations for the calculations. In the course of the algorithm, we will have to intersect triangles with each other. There are several ways, in which this intersection can happen, so Section 4.1 will give an overview on these possibilities. Moreover, as we will see in Section 4.1, we will have to calculate intersection points in two different ways. These are explained in Sections 4.2 and 4.3. Section 4.4 will introduce the barycentric coordinates, which play a decisive role in the algorithm, as they let us determine, whether or not a point lies on a triangle's face. Finally, Section 4.5 will outline how we can optimise triangulations.

4.1 Intersection of Triangles

The intersection and triangulation of triangles is implemented in a recursive way. This has to be done because, in theory, there could be an arbitrarily large number of triangles all intersecting one triangle. Thus we would have to define a triangulation for every number of vertices for every distribution on the face and the edges. Since we never know how many points there really are, we would have to define triangulation for an infinite number of intersection points and end up with an unnecessarily complicated algorithm or accept undefined cases.

When we intersect two triangles with each other, we have to distinguish between two possibilities: either the triangles lie in the same plane, or they lie in different planes.

For triangles in different planes, we intersect the first triangle's three edges with the second triangle's face, so we can find up to three intersection points. However, if we find an intersection for each of the three edges, one of the first triangle's vertices must lie on the second triangle's face and thus, is found as an intersection point for both adjacent edges. Then, there can only be one additional intersection point and we only find two distinct intersection points.

If they lie in the same plane, we intersect all pairs of edges from different triangles. Theoretically, we can find three intersection points per edge, but again, only two of them are distinct. Therefore, we can get a maximum of six intersection points, occuring if every edge intersects two edges of the other triangle. For triangles lying in the same

plane, there are some restrictions concerning the distribution of intersection points to edges and faces:

- If there is only one intersection point, it has to lie on an edge.
- In case there are two, they both have to lie on edges.
- For one intersection point on the face, we need at least two points on edges.
- For two intersection points on the face, we need at least one on an edge.
- Intersection points on the face can only occur if the point is a vertex of the other triangle. Since every triangle has three vertices, there can be a maximum of three intersection points on the face.
- If there are three intersection points on the face, there cannot be any intersection points on edges, otherwise we would get a polygon with more than three vertices.
- In case there are two intersection points on the face, we can have up to two points on edges. Each of them has to lie between one of the vertices on the face and the vertex outside the triangle.

For the triangulation of a subdivided triangle, we have to distinguish between intersections on the triangle's edges and those on its face (excluding the edges). With this distinction, we obtain a rather large number of different cases:

One Intersection Point

If we find one intersection point, it can either lie on an edge or on the face of the triangle.

One point on the edge (4.1a): This happens if the triangles' edges touch each other in one point (blue triangle), or if one triangle's vertex lies on the other triangle's edge (green triangle). As we can see, the blue version is not possible for triangles in the same plane, as there would have to be another intersection point.

One point on the face (4.1b): This occurs if one triangle's vertex lies on the other triangle's plane. We can see, that due to geometric constraints, this case is not applicable for triangles lying in the same plane. It needs its own triangulation, because we need one more triangle than for the previous case.

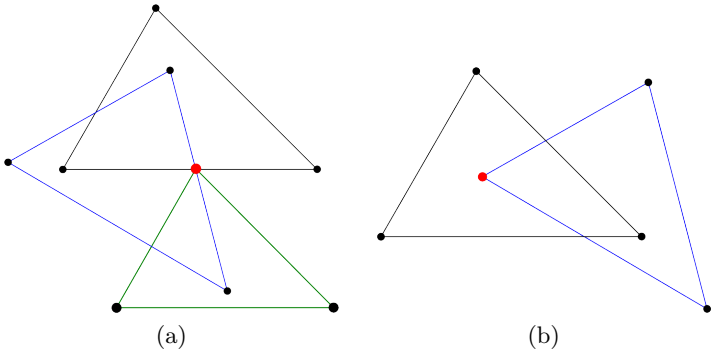


Figure 4.1: One intersection point.

Two Intersection Points

For two intersection points, all combinations of placement on an edge and on the face are possible. If two points are on edges, we have to consider both the case that they are on the same edge and that they are on different edges.

Both points on the same edge (4.2a): This can occur if either the first triangle's face intersects the second triangle's edge (blue triangle) or if the first triangle's edge lies within the second triangle's (green subtriangle). Again, the blue version cannot be used for triangles lying in one plane.

Each point on a different edge (4.2b): We can get this if the triangles intersect each other at the edges. It is only possible for triangles in different planes, as otherwise, the vertex adjacent to the two intersecting edges would have to lie on the surface.

One point on an edge and one point on the face (4.2c): This is the case if the triangles lie in different planes and have one edge running through the other's face if the first triangle has its vertex on the second triangle's face and the second triangle intersects the first triangle's face.

Both points on the face (4.2d): If one triangle's edges intersect the other triangle's face, we get two points on the face. This is another case, that is not applicable to triangles in the same plane.

Three Intersection Points

If we have three intersection points, we get all combinations of placements on edges and the face.

Each point on a different edge (4.3a): This occurs if one triangle lies inside the other triangle and has its vertices on that triangle's edges.

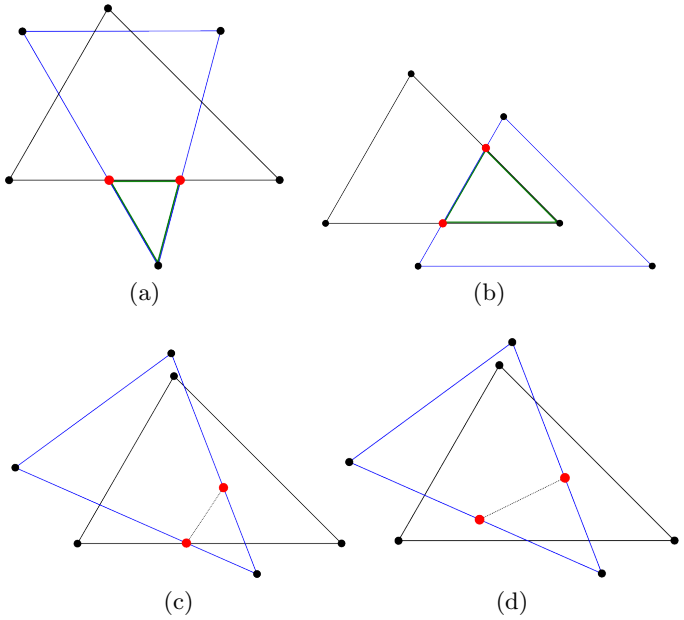


Figure 4.2: Two intersection points.

Two points on the same edge and one on a different one (4.3b): There are to possibilities how this can happen. Either the triangle lies inside the other and has two vertices on one and the third on a different edge (green subtriangle), or the triangles overlap and one triangle's vertex lies on the other's face (blue triangle).

Two points on the same edge and one on a different one (4.3c): There are to possibilities how this can happen. Either the triangle lies inside the other and has two vertices on one and the third on a different edge (green subtriangle), or the triangles overlap and one triangle's vertex lies on the other's face (blue triangle).

Two points on the same edge and one on the face (4.3d): There are two constellations, for which this can occur. Either one triangle lies inside the other and has two vertices on one edge (green subtriangle), or the triangles overlap and one triangle's vertex lies on the other's face (blue triangle).

One point on an edge and two on the face (4.3e): To let this happen, one triangle must lie inside the other and have one of its vertices on an edge and two vertices on the face.

All points on the face (4.3f): This triangle lies completely inside the other triangle and has its vertices on the that triangle's face.

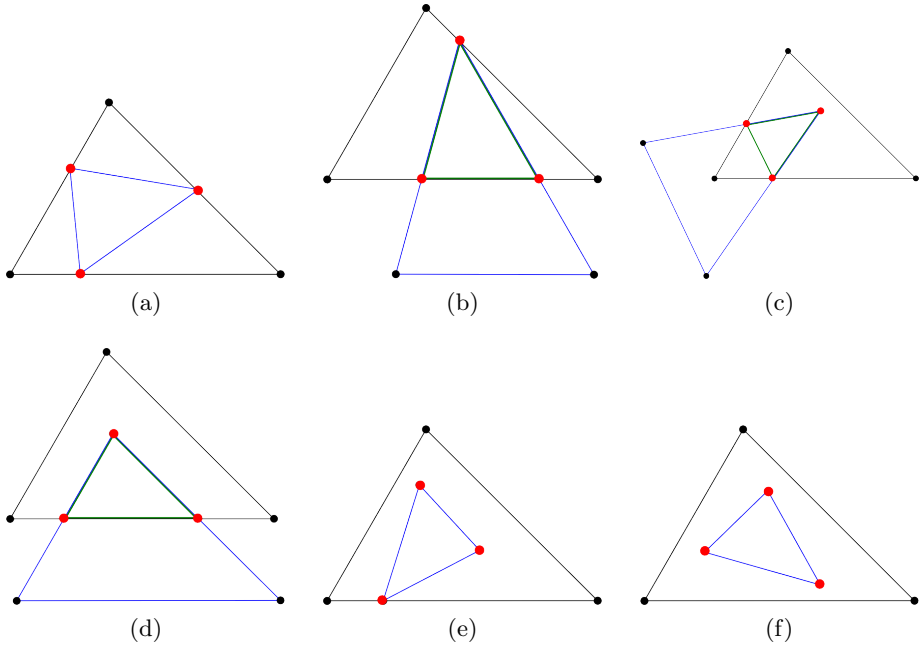


Figure 4.3: Three intersection points.

Four Intersection Points

For four intersection points, we cannot use all combinations: Every combination with three or four points on the face is impossible, as we would need quadrangles rather than triangles to reconstruct faces. So, only combinations with two or less points on the surface are possible.

Two points on one edge and the other two on different edges (4.4a): The two of the points lying on different edges of one triangle are vertices of the other triangle. The other triangle's third vertex has to lie outside the other triangle.

Two pairs of points sharing one edge each (4.4b): For this case, all vertices of one triangle lie outside the face of the other triangle, and the intersection points occur where the triangles' edges intersect.

Two points on one edge, another point on a different edge and one point on the face (4.4c): This occurs if one vertex lies on the face, another lies on the edge, which contains only one point, and the third lies outside the triangle.

Two points on different edges and two points on the face (4.4d): This happens for two vertices on the face and the third outside the triangle, behind one of the other triangle's corners.

Two points on the same edge and two points on the face (4.4e): For two vertices on the face and one on the other side of an edge, we get this case.

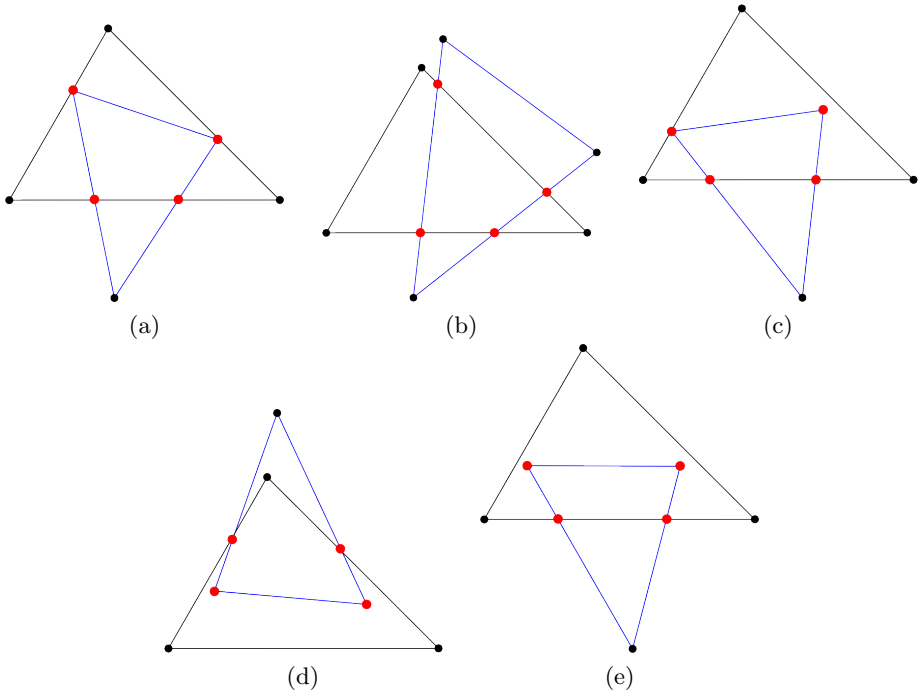


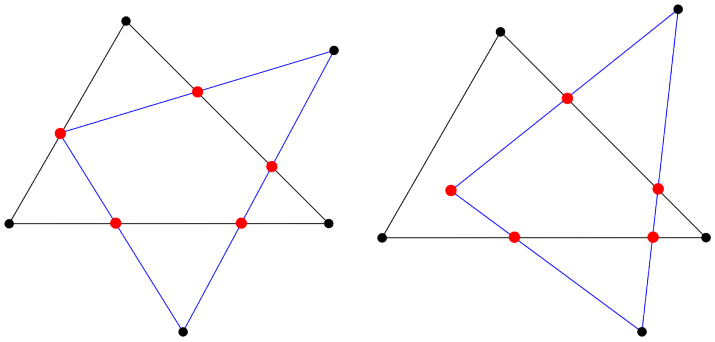
Figure 4.4: Four intersection points.

Five Intersection Points

For five intersection points, only a small selection of combinations can occur: For four points, we already had to restrict the number of points on the face to two. For five intersection points we have to reduce the number even further to one point only, because triangles with two points on the face would need three points on edges. This, however, is impossible as there is only one vertex left if two of them already lie on the face. This leaves us with two edges intersecting the other triangle. As we could see in the previous case, this gives us only two points on edges.

Two pairs on points sharing one edge each and one point on the third edge (4.5a): We can get this if one vertex lies on an edge and the other two lie outside the triangle.

Two pairs on points sharing one edge each and one point on the face (4.5b): This can occur if one vertex lies on the face and the other two lie outside the triangle.



(a) (b)

Figure 4.5: Five intersection points.

Six Intersection Points

A triangle with six intersection points can be triangulated in exactly one way.

Three pairs on points sharing one edge each (4.6): This is the case if every edge has two intersection points with edges from the other triangle.

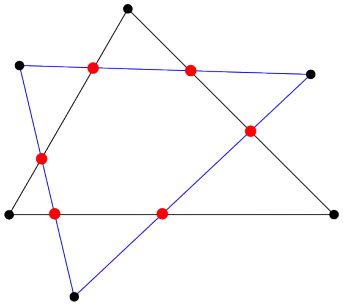


Figure 4.6: Six intersection points.

4.2 Intersection of Two Edges

Let $g = \vec{a} + s \cdot \vec{l}$, $h = \vec{b} + t \cdot \vec{m}$ be two lines we want to intersect with $\vec{a}, \vec{b}$ as location vectors of the start vertex and $\vec{l}, \vec{m}$ as vectors from the start to the end vertex of the edges.

According to [5, p. 67], if $g \nparallel h$ we can compute the intersection point from the equation $g = h$ as follows:

$$g = h$$

$$\vec{a} + s \cdot \vec{l} = \vec{b} + t \cdot \vec{m}$$

$$s \cdot \vec{l} = t \cdot \vec{m} + (\vec{b} - \vec{a})$$
$$s = \frac{b_x - a_x}{l_x} + t \cdot \frac{m_x}{l_x}$$
$$t = \frac{a_y - b_y}{m_y} + s \cdot \frac{l_y}{m_y}$$
$$t = \frac{a_y - b_y}{m_y} + (\frac{b_x - a_x}{l_x} + t \cdot \frac{m_x}{l_x}) \cdot \frac{l_y}{m_y}$$
$$t = \frac{\frac{a_y - b_y}{m_y} + \frac{b_x - a_x}{l_x}\frac{l_y}{m_y}}{1 - \frac{m_x}{l_x} \cdot \frac{l_y}{m_y}}$$

This gives us a 2-dimensional intersection point in the xy-plane. To get a 3D point, we can check if $a_z + s \cdot \vec{l}_z = b_z + t \cdot \vec{m}_z$. If the equation is satisfied, we get our intersection point P with $\vec{p} = \vec{a} + s \cdot \vec{l} = \vec{b} + t \cdot \vec{m}$; otherwise g and h are skew lines.

Now, we have to check if $0 \leq s \leq 1$, $0 \leq t \leq 1$ holds, because otherwise the edges do not intersect, as the point is outside their range on the line. If it does, P's vertex location is set to `on` since it lies on the surface of both objects by definition.

Optimisation using the Distance of Two Lines

We can achieve a small improvement of performance if we check the distance of g, h prior to calculating s and t, as defined in [5, p. 67]:

$$d(g,h) = \left| (\vec{b} - \vec{a}) \cdot \frac{\vec{m} \times \vec{l}}{||\vec{m} \times \vec{l}||} \right|$$

If $d > 0$, g and h are skewed and we can skip the calculation. If $d = 0$, there is an intersection of g and h, so we can compute an intersection point. This saves us the check for the z coordinate of the equation.

Optimisation for Edges in the Same Plane

If the lines lie in the same plane, they cannot be skew lines. Thus, we do not have to check the z coordinate or the distance of the lines. For our algorithm, this can always be used, as we only use this intersection test for triangles lying in the same plane.

4.3 Hesse Normal Form

Let $\vec{n} = (n_x, n_y, n_z)^T$, $|\vec{n}| = 1$ be the normal of the triangle with that needs to be tested. Further, let $A(a_x, a_y, a_z)$ be a vertex of the triangle (more generally: a known point on the surface) and $P(x, y, z)$ an arbitrary point of the surface (i.e. our intersection point) with the location vectors $\vec{a}, \vec{p}$ and let $d = \vec{n} \bullet \vec{a}$ be the distance of the plane to the origin of the coordinate system.

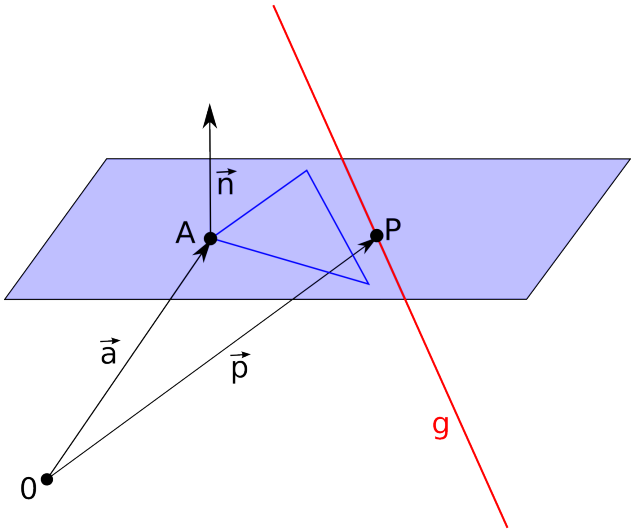


Figure 4.7: Line g intersects the plane in which the triangle lies in the point P.

Then the Hesse normal form of a plane can be defined as

$$\vec{n} \bullet \vec{p} - \vec{d} = 0 \quad \Leftrightarrow \quad n_x x + n_y y + n_z z - d = 0$$

(cf. [5, p. 66]). Given a straight line $g(t) = \vec{b} + t \cdot \vec{m}$ line, e.g. along a triangle's edge, we can now put in these values for P and calculate the t for which the plane and the line intersect:

$$n_x(b_x + t \cdot m_x) + n_y(b_y + t \cdot m_y) + n_z(b_z + t \cdot m_z) - d = 0$$

$$\Leftrightarrow b_x n_x + b_y n_y + b_z n_z + t(m_x n_x + m_y n_y + m_z n_z) - d = 0$$

$$\Leftrightarrow t_{result} = -\frac{b_x n_x + b_y n_y + b_z n_z - d}{m_x n_x + m_y n_y + m_z n_z}$$

If this t fulfils $0 \leq t \leq 1$, the edge intersects the plane in the intersection point P with $\vec{p} = \vec{b} + t_{result}\vec{m}$.

4.4 Barycentric Coordinates

Barycentric coordinates describe the position of a point P in relation to a simplex. For this algorithm, we only need the planar case, namely a triangle with corner vertices Q_i.

As depicted in Figure 4.8, the point P subdivides the original triangle t into three new subtriangles t_i, $i \in \{1, 2, 3\}$. The barycentric coordinates b_i can be written as $B(b_1, b_2, b_3)$ with

$$b_i = \frac{A_i}{A} \quad \text{and} \quad b_1\vec{q}_1 + b_2\vec{q}_2 + b_3\vec{q}_3 = \vec{p}$$

and A, A_i defined as

$$A = \frac{||\vec{e}_1 \times \vec{e}_2||}{2},$$
$$A_i = \frac{||\vec{p}_{i-1} \times \vec{p}_{i+1}||}{2}$$

with $i \equiv n \mod 3$, $n \in \{1, 2, 3\}$, as there are only three points.

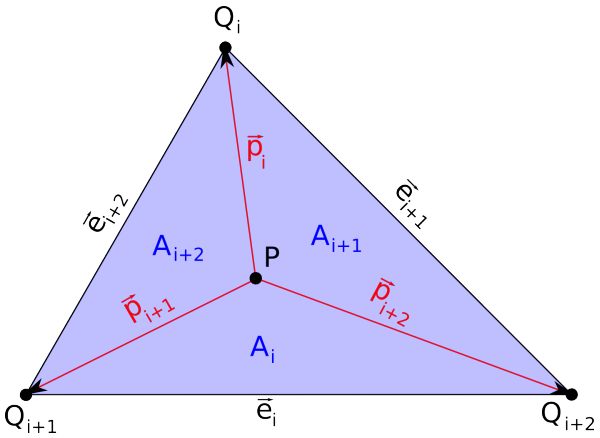


Figure 4.8: Triangle labelled for the barycentric coordinates of P.

For this example, the barycentric coordinates are $B(0.3, 0.4, 0.3)$.

Barycentric coordinates, as introduced in [4, pp. 34–37], are used in the algorithm due to a couple of very convenient properties they possess:

1. P lies inside the triangle iff $\sum_{i=1}^{3} b_i = 1 \wedge \forall_i 0 \leq b_i \leq 1$, else $\sum_{i=1}^{3} b_i > 1 \vee \exists_i (b_i < 0 \vee b_i > 1)$.
2. P lies on an edge iff $\exists_i \ (b_i = 0 \wedge \forall_{j \neq i} \ (b_j \neq 0) \wedge \sum_{j \neq i} b_j = 1)$.
3. P lies on a vertex iff $\exists_i \exists_j ((b_i = 0 \wedge b_j = 0) \wedge \exists_{k \neq i, k \neq j} (b_k = 1))$.

We can use property 1 to determine, whether or not the point calculated in 4.3 lies inside a triangle or not. If it does, we know that the two triangles that were tested intersect.

Moreover, we can also use this property for the rating of triangles, which will be discussed in detail in Section 5.5. It is possible to use this in 3-dimensional space, because the tetrahedron we receive, when connecting all pairs of triangle vertices with a point that lies outside the plane, will have a greater surface on all upper triangles than on the base.

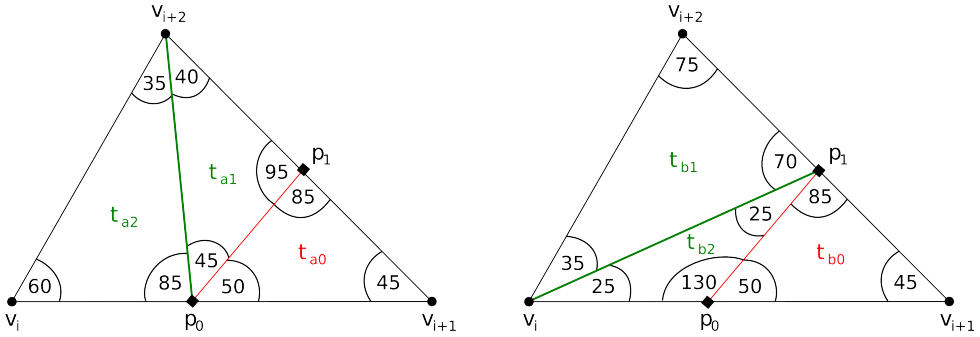
The properties 2 and 3 are used for the triangulations.

If the point lies on an edge, the triangulation needs to be done in a different way than if it lies on the face, since e.g. for a triangulation with one intersection point, one of the three subtriangles would have the area 0, hence we need only two rather than three triangles. For other triangulations, similar restrictions exist.

If the point lies on a vertex of the triangle, which has to be triangulated, it is not used for the triangulation, since two out of three subtriangles would have the area 0. Whenever this occurs, the vertex is removed from the list of points, with which the triangle has to be subdivided.

4.5 Optimisation of Triangulations

Sometimes, there are possibilities to optimise a triangulation. One possibility occuring quite often is given if there are quadrangles in the subdivided triangle after adding the vital edges. An example is given in Figure 4.9. The subtriangle t_0 has to be separated, but the remaining part can be triangulated in two ways.



(a) Possibility A (b) Possibility B

Figure 4.9: Optimisation for subquadrangles in triangulations.

During our optimisation, we are looking for $max(min(angles(t_{a1}, t_{a2})), min(angles(t_{b1}, t_{b2})))$ with t_{ai} from possibility A, t_{bj} from possibility B. If the maximum is found for t_{ai}, the triangulation is chosen as in Figure 4.9a, otherwise it is chosen as in Figure 4.9b.

For this example we get:

$$max(min(angles(t_{a1}, t_{a2})), min(angles(t_{b1}, t_{b2})))$$

$$max(min(95°, 40°, 45°, 35°, 60°, 85°), min(70°, 75°, 35°, 25°, 130°, 25°))$$
$$max(35°, 25°)$$
$$35°$$

In other words, triangulations with larger smallest angles are preferred to those with narrower smallest ones with the goal of avoiding slivers. Thus, for our example, the triangulation in Figure 4.9a is certainly better than the triangulation in Figure 4.9b, since its smallest angle is 35° rather than 25°.

5 Merging Algorithm

First, the given files are read, parsed and preprocessed. Then, we check if the two objects overlap. If they do, we continue, otherwise we skip the algorithm and proceed to the saving of our data. If the objects' bounding boxes overlap, we test all pairs of groups from different objects for overlaps and, if we find one, detect, which triangles are impacted, i.e. they lie partly or completely inside the overlapping area of the bounding boxes.

Now, all pairs of triangles from the sets of impacted ones are intersected, and if there are intersection points, the triangles (which are not tagged `remove`) are triangulated with the points. All intersection points that have been used are stored in the `allPoints` list for later use.

For reasons outlined in Section 5.4, we triangulate all triangles with intersection vertices, if they have not yet been tagged `remove`. Afterwards, we conduct the in-out-test for every triangle's centroid and tag the triangle according to its value.

Finally, we save all triangles that are neither tagged `remove` nor already part of the `finalTriangles` list (that is, they are not equal to a triangle in the list).

5.1 Reduction of Search Space

There are several ways, in which we can reduce the number of triangles we have to evaluate. We can make use of the objects' bounding boxes, the manikins' subdivision into groups and the triangles' status tag.

By **search space**, the number of triangles which have to be evaluated is meant. In general there is a huge amount of data that has to be processed.

5.1.1 Exploiting Bounding Boxes

One generally applicable way to reduce the search space is to clip the set of evaluated triangles to the overlapping part of their objects' bounding boxes.

All triangles, which are partly or completely inside this overlap, i.e. which have at least

Algorithm 1 Simplified pseudocode version of the algorithm for inputs $file_1$ and $file_2$.

```
objhandler1→read (file_1)
objhandler1→read (file_2)
objhandler1→preprocess (object_1)
objhandler2→preprocess (object_2)
// if the objects' bounding boxes overlap
if (calculateOverlap (boundBox (object_1), boundBox (object_2)))
    // combine all pairs of groups from different objects
    for (group_k1 ∈ object_1, group_l2 ∈ object_2)
        // if the groups' bounding boxes overlap
        if (calculateOverlap (boundBox (group_k1), boundBox (group_l2)))
            // reduce search space for group_k1
            impacted_k1 = detectImpactedTriangles (group_k1)
            // reduce search space for group_l2
            impacted_l2 = detectImpactedTriangles (group_l2)
            // check all pairs of impacted triangles from different objects for intersections
            for (t_i ∈ impacted_k1, t_j ∈ impacted_l2)
                ⟨intersectionPoints⟩ = calculateIntersectionPoints (t_i, t_j)
                // if any intersection points have been found, calculate subtriangles
                if (⟨intersectionPoints⟩ ≠ ⟨ ⟩)
                    triangulator→triangulate (t_i, ⟨intersectionPoints⟩)
                    triangulator→triangulate (t_j, ⟨intersectionPoints⟩)
                    // add all used intersection points to allPoints
                    ⟨allPoints⟩ →push_back(subset (⟨intersectionPoints⟩))
            // check all triangles for intersections with previous points if they are not removed yet
            // and determine their centroid's location; if the location is in, remove the triangle
            for (t_i ∈ group_k1)
                if (intersects (t_i, ⟨allPoints⟩) ∧ status (t_i) ≠ remove)
                    triangulator→triangulate (t_i, subset (⟨allPoints⟩))
                objhandler1→in-out-test (centroid (t_i), object_2)
                if (location (centroid (t_i)) = in)
                    status (t_i) = remove
            for (t_j ∈ group_l2)
                if (intersects (t_j, ⟨allPoints⟩) ∧ status (t_j) ≠ remove)
                    triangulator→triangulate (t_j, subset(⟨allPoints⟩))
                objhandler2→in-out-test (centroid (t_j), object_1)
                if (location (centroid (t_j)) = in)
                    status (t_j) = remove
        // save all triangles which are still part of one of the objects
        for (t_i ∈ object_1)
            if (status (t_i) ≠ remove ∧ t_i ∉ ⟨finalTriangles⟩)
                save (t_i)
        for (t_j ∈ object_2)
            if (status (t_j) ≠ remove ∧ t_j ∉ ⟨finalTriangles⟩)
                save (t_j)
else
    // save all triangles from both objects
    for (t_i ∈ object_1)
        save (t_i)
    for (t_j ∈ object_2)
        save (t_j)
```

one vertex inside the overlap, are evaluated further. All other triangles are completely outside the overlapping bit and can be ignored for the current pair of groups. These triangles will be considered again as soon as their group is tested against a third group.

5.1.2 Exploiting the Groups

As seen in Figure 5.1, in the OBJ files generated by the RAMSIS system, every manikin is divided into 52 groups containing different body parts, i.e. head, left lower leg or the right hand's index finger's first phalanx. It is possible to exploit this fact: If the bounding boxes of two groups overlap, further testing will be conducted. However, in the majority of cases, they do not overlap, so there cannot be an intersection. Thus, the pair can be discarded and we can save a lot of computation time.

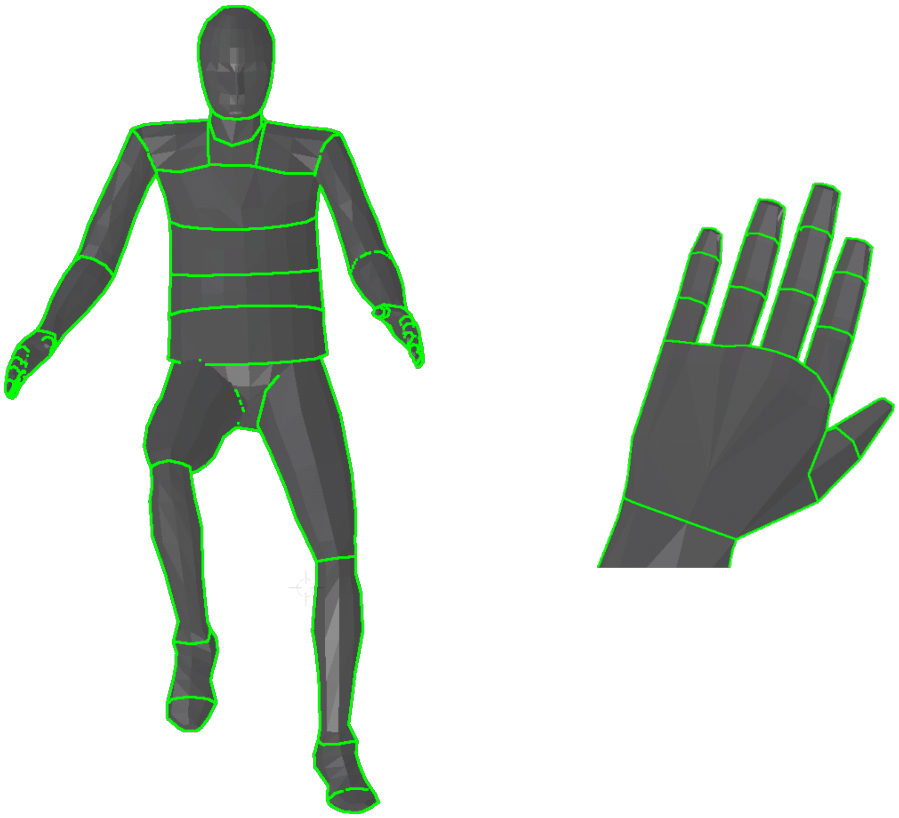
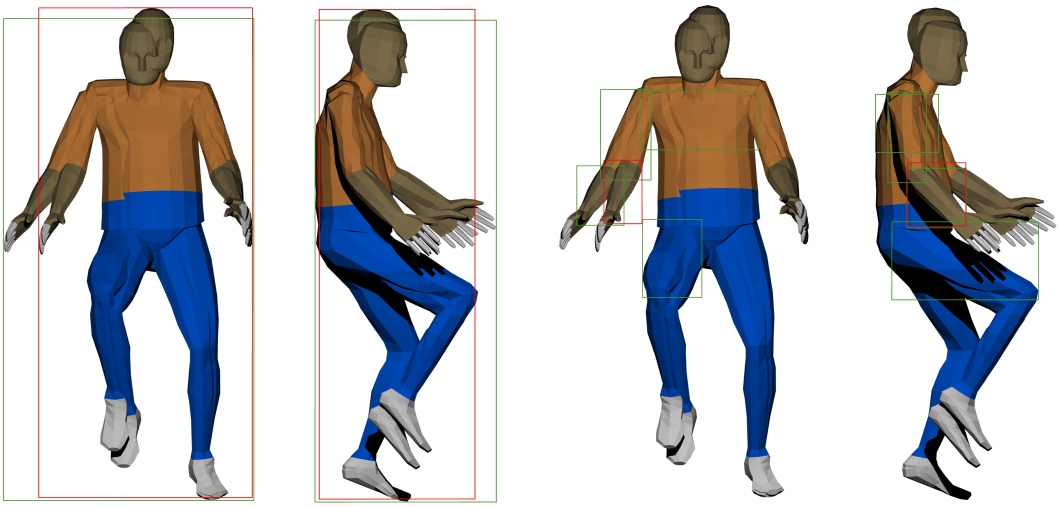


Figure 5.1: Subdivision of a full manikin and of its hand.

Exploiting this knowledge, we can reduce the search space quite drastically. The manikin in Figure 5.1 has a total of 2729 triangles. If we wanted to merge this manikin object with another one of a similar size, we would have to test all pairs of triangles between the two sets of triangles for intersections, which would result in $2729^2 = 7447441$ intersection tests at once. Two consecutive snapshots of an action tend to have large overlapping

areas, as seen in figures 5.2a and 5.2b, therefore we can get quite close to this upper bound. Unfortunately, this is very likely to happen, since we want to evaluate vehicle occupants' movements. To do this properly, we need a good approximation of the movement, and this is obtained with many samples which are close to each other. The two objects in Figure 5.2 are from two such snapshots.



(a) front view for objects (b) side view for objects (c) front view for groups (d) side view for groups

Figure 5.2: The overlap of two objects for consecutive snapshots of movement is very large. If we restrict the evaluations to groups with overlapping bounding boxes, the number of evaluations is a lot smaller.

By contrast, if only groups are intersected, most pairs of groups are discarded for calculation because their bounding boxes do not overlap. In figures 5.2c and 5.2d, the one object's forearm is evaluated. Its bounding box is drawn in red. In addition, all bounding boxes of the other object's groups, which could potentially overlap with the forearm's are drawn in green. As we can see, only the upper arm and the forearm intersect. The thigh's bounding box is very close, but it does not intersect the forearm's. If it did, there would not be any triangles inside the overlapping area, so we would not have to intersect any triangles either (cf. Section 5.1.1).

The biggest amount of tests for two groups would have to be conducted when testing the two head groups, which have 382 triangles each; this would result in a maximum of $382^2 = 145924$ tests at once. All other groups have 111 triangles at most, with a median of 36 triangles.

The global worst case would result in the same number of evaluations as if we used the whole objects, however, this is highly unlikely since it would mean that all groups' bounding boxes, and therefore all limbs of the body, would overlap.

Iterative Application

If the algorithm is applied iteratively, it should not be executed in parallel with different objects, as we would not be able to exploit the groups when merging objects that have already been merged before. For iterative implementation, the area, in which the impacted triangles for a step can lie, is at most as big as the bounding box of the currently tested group of the fresh object, so the algorithm will still perform fairly well.

5.1.3 Exploiting the Status Tags

Finally, the status tag of each triangle offers a very straightforward possibility to reduce the search space. If a triangle has been tagged with the status `remove`, it is ignored during almost all testing and calculation processes, as it has already been replaced with new ones, which take up the same area. The only test for which it is still needed is the in-out-test that is conducted for the triangle's centroids to find out if a triangle, which has three vertices with location `on`, lies inside, outside or on the object it is not part of.

5.2 Determination of Vertex Locations

To determine the location of a vertex from one object, another object is needed, since the location is specified in relation to a geometric object. Let us call the process of this determination in-out-test.

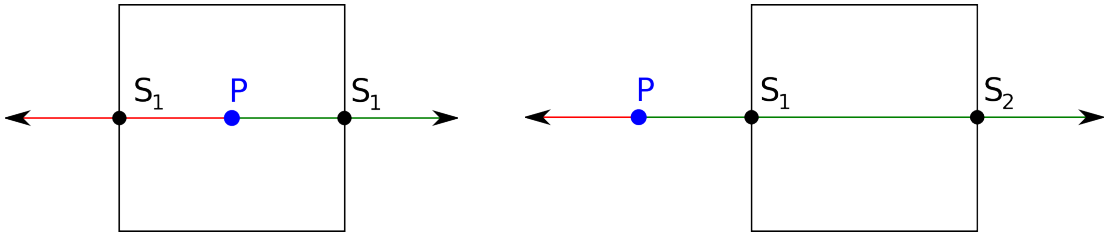
If the vertex lies on the surface of a complete, hollow object, its location is set to `on`. If it lies inside an object, the number of hits of a line in any direction with the object surface has to be odd ($n \equiv 1 \mod 2$), whereas if it lies outside, the number of hits has to be even ($n \equiv 0 \mod 2$).

A small cross-section example visualises this:

Figure 5.3a displays a point P, which is located inside a cube. If we go left, we find one intersection of the line with the cube; if we go right, we also find one intersection. As both directions give us an odd number of intersections, P's location is `in`.

In Figure 5.3b, P lies outside the cube. If we go left, we find no intersection at all. If we go right, we find two intersections with the cube. Since $2 \mod 2 \equiv 0 \mod 2$, we get an even number of intersection points for both vectors, so P's location is `out`.

The in-out-test function works as follows: First, we have to choose a vector v_1 with an



(a) P is inside the object (b) P is outside the object

Figure 5.3: v_1 is the green vector, v_2 the red one.

arbitrary but fixed direction. In addition, we will also use a parallel vector v_2 with the opposite orientation, as seen in Figure 5.3, so let us take

$$\vec{v}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \text{with} \quad \vec{v}_2 = -\vec{v}_1 = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}.$$

Then we can test every vertex against every triangle.

The easiest case is the one in which the vertex lies on a triangle surface (identified with barycentric coordinates as described in Section 4.4): The vertex location is set to `on` and the next vertex will be checked.

For a vertex with a still unknown location the processing continues if the vector v_1 (or v_2 respectively) does not run through the triangle's face[1]. If it did, either two hits would need to be counted (in on one side and out again on the other) or no hit at all (ignoring the face or not hitting it). Then, we shoot rays from the vertex which has to be tested in the direction v_1, as depicted in more detail in Section 5.3. Additionally, we also do this for v_2 as to verify the results. If the rays hit a triangle face, a hit for their direction is counted and the point is registered for every hit that is in a unique location. Multiple hit points with the same coordinates are only counted as one; due to the *uniqueness* property, there can only be one point in this location.

Finally, the number of hits is evaluated. If it is even, the vertex has to lie outside the object and the location is set to `out`. If it is odd, the vertex lies inside the object and its location is set to `in`. If the number of hits is even for one direction and odd for the other, there have to be self-penetrations in the hull. In this case, we set the location to `on`.

This counting method is only reliable if the properties stated in Section 2.3 are fulfilled: If there are holes in the object, the *coherence* property is violated, so vertices are misclassified as `in` instead of `out` (and vice versa) if the hole lies in the direction v_1 or

[1] More precisely, we can test whether the vector is orthogonal to the triangle's normal, since it would then run parallel to or in the plane the triangle lies in. The check for orthogonality can be done with the dot product. If it is $\neq 0$, we continue checking.

v_2 from them. Objects which have triangles inside violate the *hollowness* property and cause similar problems. All vertices, for which the ray hits these triangles are classified incorrectly.

This part of the algorithm is used for several tests:

- If at least one corner of a triangle has the vertex location `in`, the triangle has to either lie inside the object or it has to intersect the object's hull. After running the algorithm, all triangles with this condition lie inside the triangle, whereas in the beginning, only those triangles which have all vertices located inside the object are inside the triangle and all others intersect the object and will be subdivided.
- If the triangle centroid's location is `in`, the triangle lies partially or completely inside the object. If the former is true, the triangle intersects the object and will be subdivided; inside triangles will be removed.
- If the triangle centroid's location is `on`, the triangle overlaps with a part of the object or its centroid lies on the intersection line of the two objects. In the former case, the triangle will be kept, in the latter case it will be subdivided.

The last two points will be elaborated on in Section 5.5.

5.3 Detection of Intersection Points

Since the triangles need to be subdivided along intersection lines, the intersection points of two triangles have to be determined. This is done for every pair of triangles in three steps:

First of all, we check, which intersection test we need for computing the intersection points. For triangles in the same plane, we use the intersection test for two lines (cf. Section 4.2) for each pair of edges from different triangles. Otherwise we use the Hesse normal form (cf. Section 4.3) to compute the intersection point of a straight line with a plane. The line runs along one of the first triangle's edges and intersects the plane, the second triangle lies in. This is done for every edge of the triangle.

Finally, the barycentric coordinates (cf. Section 4.4) for the calculated point are computed. If each coordinate lies in $[0, 1]$ and they sum up to one, the point lies inside the triangle and, provided it is novel, the intersection point can be added to the list of intersection points for this pair of triangles.

After these steps, the triangulation can be conducted.

5.4 Triangulation

In Section 4.1, we already discussed the different possibilities of intersections between two triangles. There are a few actions which have to be taken for all triangulations:

If an intersection point is in the same place as a vertex of a triangle we want to subdivide, it will be ignored for this calculation, because no subdivision is needed. However, the point is used for the subdivision of the triangle that has been met if it lies on its edge or face.

After the triangulation, every new triangle's normal is compared to the old one's. If they point into opposite directions, the new triangle has a definition of inside and outside that contradicts the original triangle's. To correct this, two of the new triangle's vertices are swapped and thus is the normal. Otherwise, nothing is changed.

Furthermore, the new triangles are added to their vertices lists of using triangles and all new triangles are added to the old triangle's replacement triangles list.

In the figures for the following sections, all subdivisions marked in green can be optimised in the way described in Chapter 4.

5.4.1 One Intersection Point

If there is only one intersection point, it can either lie on one of the triangle's edges or on the triangle's face.

One Point on an Edge

For one point on an edge, the triangle is then subdivided into two parts (cf. Figure 5.4) which share one vertex and the intersection point.

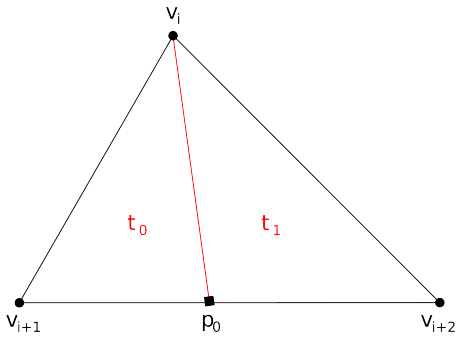


Figure 5.4: Triangulation for one intersection point on an edge.

One Point on the Face

One point on the face results in a very simple triangulation. As can be seen in Figure 5.5, every subtriangle consists of two vertices from the original triangle and the intersection point.

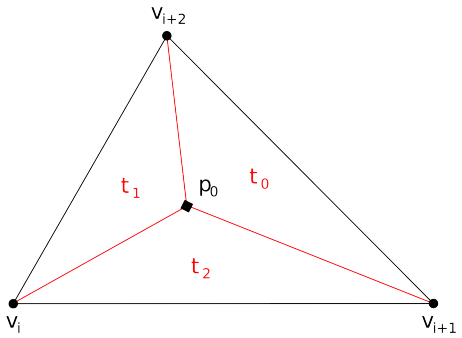


Figure 5.5: Triangulation for one intersection point on a face.

5.4.2 Two Intersection Points

With two intersection points, which both lie on edges, there are two possible triangulations. Either both points lie on the same edge, or they lie on different edges.

Two Points on Edges

If the two points lie on the same edge, the triangulation is done as depicted in Figure 5.6. The vertex v_i which is opposite to the edge with the intersection points is part of all subtriangles. The intersection point closer to v_{i+1} and the two vertices v_i, v_{i+1} form the triangle t_0. t_2 is defined in parallel with v_i, v_{i+2} and the other intersection point as corners and t_1 is formed by v_i and the two intersection points.

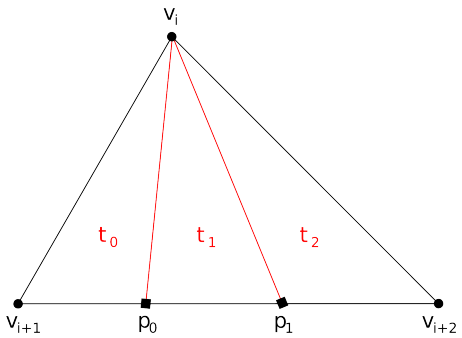


Figure 5.6: Triangulation for two intersection points on the same edge.

If the two points lie on different edges, the subtriangule t_0 is fixed and the other two triangles can be optimised. The result can be seen in Figure 5.7

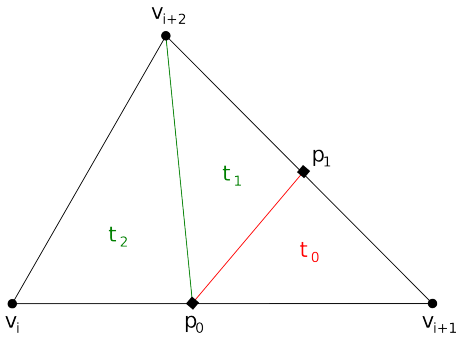


Figure 5.7: Triangulation for two intersection points on different edges.

One Point on an Edge and one on the Face

Triangulating a triangle for one intersection point on an edge and one on the face is rather simple. The two intersection points are connected by an edge and they form triangles with the v_{i+1}, v_{i+2}. The other two triangles contain two vertices contained in the original triangle each and the intersection point on the face, as depicted in Figure 5.8.

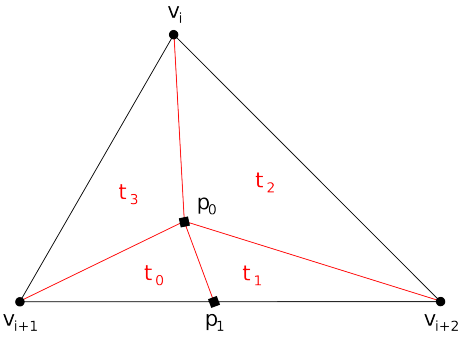


Figure 5.8: Triangulation for one intersection point on an edge and one on the face.

Two Points on the Face

The triangulation for two intersection points on a face is more difficult as we do not get fixed reference points such as points on edges. While all previous cases can be handled with modulus calculation, this is not enough to determine how to best triangulate this case. Instead, we need to check the angles between the vectors from each vertex to both intersection points. Let vertex v_i be the vertex with the smallest angle between the vectors is connected to the intersection point that is closer to it by an edge. As displayed in Figure 5.9, the triangles t_0 and t_4 share both v_i, the intersection point, and one different point $v_j \neq v_i$ as their third corner. Triangle t_2 uses the other two

vertices and the other intersection point. The two remaining triangles, t_1 and t_3 use both intersection points and one vertex v_j with $v_j \neq v_i$.

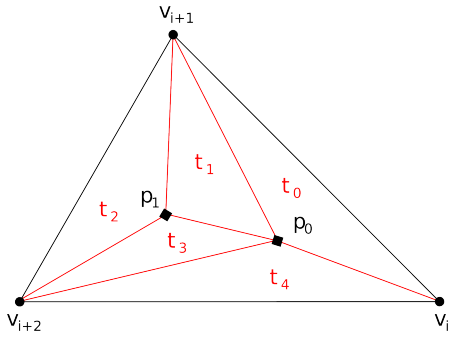


Figure 5.9: Triangulation for two intersection points on the face.

Theoretically, we could take a random vertex as the one with only one neighbouring intersection point. However, we choose the one with the smallest angle between the vectors to the intersection points, since we want to avoid getting slivers and this is a neat possibility to include a bit of local mesh optimisation.

This idea could be extended, by creating all three possible triangulations and then choosing the locally optimal one. Let us nevertheless choose the solution pointed out above, since the angles that are most likely to be the smallest ones are those between the vectors from the corner points to the intersection points.

5.4.3 Three Intersection Points

Three or more intersection points are only possible if the two triangles being tested lie in the same plane.

For three intersection points, there are six possibilities how the triangle can be subdivided. The blue triangles are examples of how the combination of intersection points in each category can occur.

1. If all intersection points lie on the face then the triangle, which has these points as corners lies completely inside the other triangle, which can be triangulated as seen in Figure 5.10a. The vertex-point pair with the smallest distance is connected first. Then the second and third vertex are connected with the second and third point. The remaining subquadrangles can be triangulated in two ways each, as mentioned above.

2. A triangle with one intersection point on an edge and two on the face can be triangulated like the one in Figure 5.10b; the whole smaller triangle lies on its face. For the triangulation, the two points on the face are connected to the vertex opposite the point on the corner and to the closer of the two remaining vertices.

3. If two intersection points lie on edges and one point lies on the face, there are two possibilities: either both points lie on the same edge or they lie on different edges.

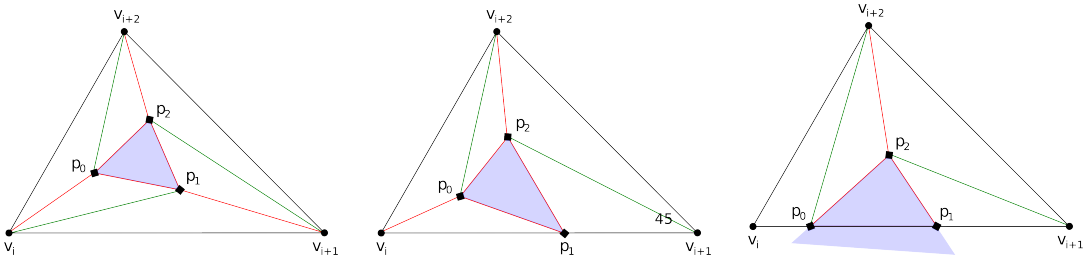
   a) For two points on the same edge and one on the face, the one on the face is connected to the vertex opposite to that edge and the blue subtriangle spanned by the intersection points is fixed. The remaining four subtriangles can be computed as in Section 5.4.2 (cf. Figure 5.10c).

   b) For two points on different edges, the triangulation is entirely fixed as depicted in Figure 5.10d.

4. Triangles with all three intersection points on edges can be triangulated in two different ways:

   a) In one way, the three points lie on three different edges, so they naturally subdivide the triangle into four subtriangles as seen in Figure 5.10e.

   b) Alternatively, the two points share an edge and the third is on a different edge. Then two triangles are fixed and two more can be calculated as in the section above. This can be seen in Figure 5.10f.



(a) All points on the face (b) One point on edge, two on the face (c) Two points points on same edge, one on the face

(d) Two points on different edges, one on the face (e) Three points on three edges (f) Three points on two edges

Figure 5.10: Triangulations for three intersection points.

5.4.4 Four Intersection Points

With four intersection points, there are five ways to subdivide the triangle.

1. There are two intersection points on edges and two more on the face. The points on the edges can either share an edge or lie on different edges.

   a) The first case can be seen in Figure 5.11a: except for the quadrangle, which is spanned by the points, all subtriangles are fixed.

   b) Figure 5.11b displays the second case. Here, we have two quadrangles that can be optimised.

2. If there are three intersection points on edges and one on the face, two of those three have to share an edge, whereas the third lies on a different one. Except for one missing triangle, this case is very similar to the first. The result can be seen in Figure 5.11c.

3. For the last category, all four points lie on edges. Again, we have two possibilities.

   a) Figure 5.11d displays the case in which the four points lie on two edges. We get one fixed subtriangle and there are two optimisable quadrangles.

   b) In the other case, two points share an edge and the other two use one of the other edges each, as shown in Figure 5.11e. Then, three subtriangles are created naturally and the remaining two can be optimised from the quadrangle.

5.4.5 Five Intersection Points

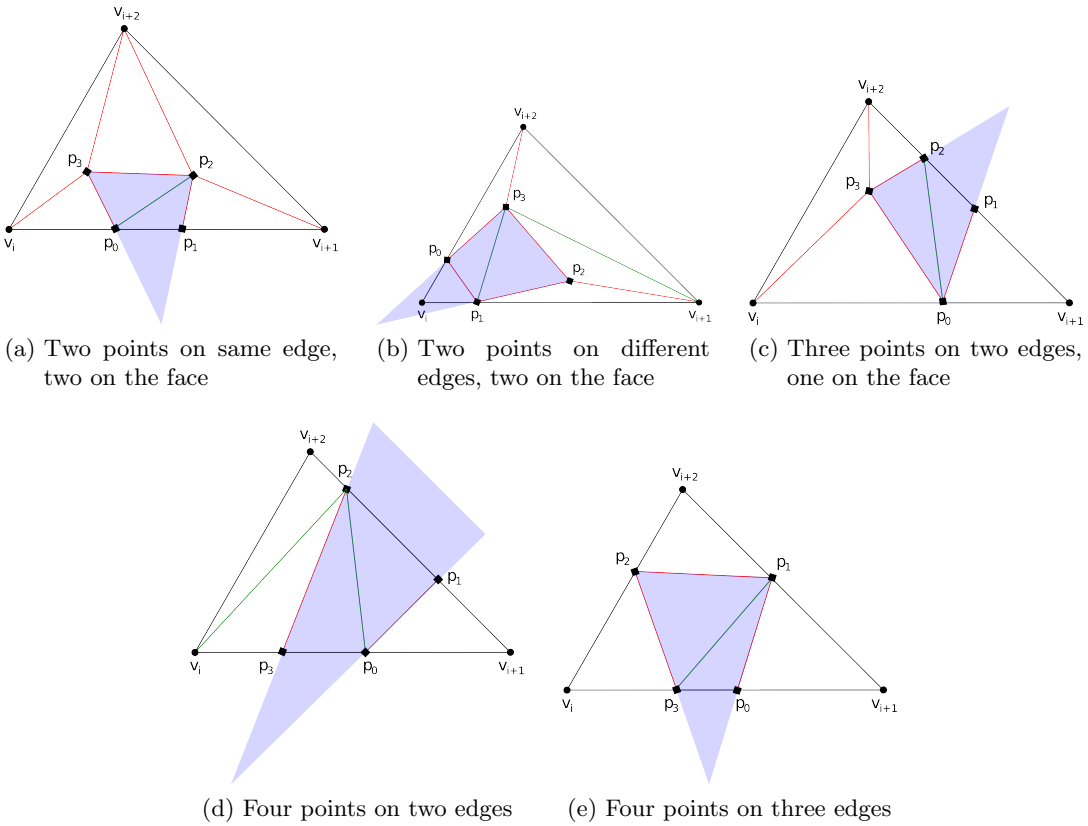
If we have five intersection points, we can subdivide the triangle in two ways:

1. For four points on edges and one point on the face, the points occupy two edges. As we can see in Figure 5.12a, most triangles are fixed and there is one quadrangle we can optimise.

2. If all five points lie on edges, there are two pairs of points sharing one edge each and one point which lies on the third edge. Except for the part that is spanned by the points on the edges, all subtriangles are fixed, as can be seen in Figure 5.12b

5.4.6 Six Intersection Points

There is exactly one way to triangulate for six intersection points, which is depicted in Figure 5.13: the hexagon in the middle can be divided into two quadrangles which an be optimised. The three outer subtriangles are fixed.



(a) Two points on same edge, two on the face

(b) Two points on different edges, two on the face

(c) Three points on two edges, one on the face

(d) Four points on two edges

(e) Four points on three edges

Figure 5.11: Triangulations for four intersection points.

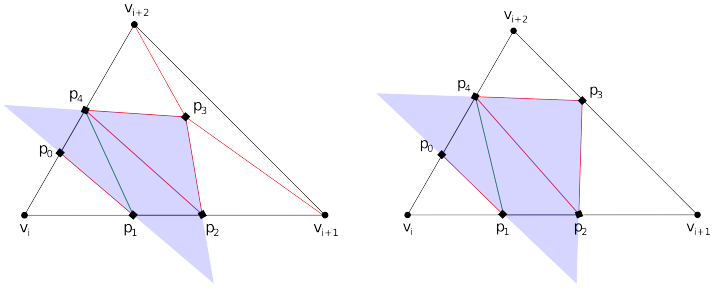
5.5 Classification of Triangles

Immediately after the triangulation, the original triangle is tagged `remove`, since it has been replaced with subtriangles.

As soon as all triangles have been processed, all those triangles which have not been tagged `remove` are evaluated. These triangles add up to the same hull as the original objects and they constitute a subdivision which is exactly as refined as necessary — no further subdivisions are needed. There are several possibilities how the triangle's vertex locations can be combined. They lead to different **classifications**:

All in If all vertices are inside the object, the triangle is removed, since it lies completely inside the final object. All intersections have been performed, thus there cannot be any more intersections.

Mixed in/on This happens for triangles, which have one or two corners on the surface of the object and at least one corner inside. For the same reasons as above, they are removed.



(a) Four points on two edges, one on the face

(b) Five points on five edges

Figure 5.12: Triangulation for five intersection points.

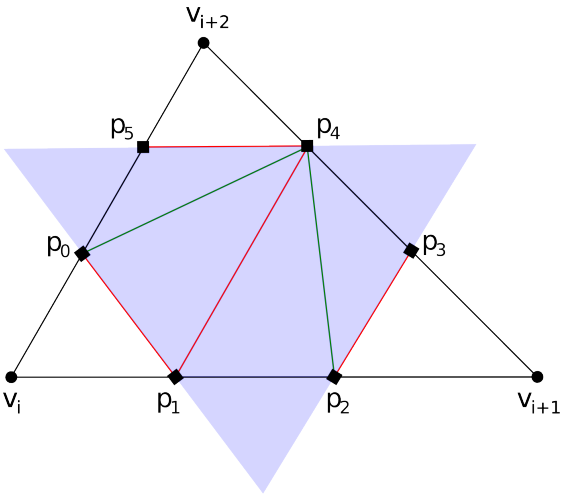


Figure 5.13: Triangulation for six intersection points.

All out If all vertices lie outside the other object, the triangle is kept (i.e. not tagged `remove`). As already pointed out for the first case, there cannot be any additional intersections, so the triangle face cannot lie partly inside an object part that is between its vertices.

Mixed out/on This happens for triangles, which have one or two corners on the surface and at least one corner outside the other object. As in the previous case, we keep the triangles.

All on This is a critical case. Triangles which have all their vertices on the surface of the final object can either lie completely on its surface or run through the inside of the object. We want to keep those triangles which are on the surface and remove those that are inside the object.

To deal with the last case, the location of the triangle's **centroid** is tested. If it is `in`, the triangle lies inside the object and it will be tagged `remove`. However, if it is `on`, we are faced with another choice to take: The triangles can either lie on the surface, or the

two objects could share a face that is inside the object. The distinction can be made by using the normals: If they point into the same direction, the triangles lie on the surface and we want to keep one of them. And if they point in opposite directions, the triangles share a face inside the object and we want to remove both.

5.6 Saving data

In the final step the data can be saved.

Triangles are added to the `finalTriangles` list if *a*) the triangle is not tagged with `remove`, *b*) none of its vertices is tagged `in`, and *c*) there is no triangle in the `finalTriangles` list that is equal (with respect to vertex positions) to the triangle.

Vertices are added to the `finalVertices` list if *a*) they are part of a triangle that is saved, and *b*) there is no vertex in the `finalVertices` list that is equal (with respect to coordinates) to the vertex.

During this process, duplicate vertices between groups are removed to preserve our *uniqueness* property.

When this is finished, all vertices and triangles from the `finalVertices` list and the `finalTriangles` list are written to a new `obj` file.

6 Implementation

The program implementing our merging algorithm has been implemented in C++. Since every algorithm needs data to work with, we need some classes for the geometric primitives `Vertex`, `Triangle` and `Vector`. Furthermore, several classes are needed to execute the algorithm. For our algorithm, these are an `OBJHandler`, a `Triangulator` and the main class, the `MagicMeshMerger`.

In the following paragraphs, an overview over these classes will be given.

6.1 Vertex

Vertices are 3D points which belong to an object. In the data currently provided by the RAMSIS system, there are several pairs of vertices with identical coordinates, lying on the border between two successive subgroups. These doubles are removed automatically during the saving process. In order to do this, we need to provide the class member function `equals()`, which determines whether two vertices have identical sets of coordinates.

During the algorithm, we have exactly two objects and the location is specified in relation to one which the vertex is not part of. The relation to that object is saved as a location tag, which is an `enum` that can take values out of

out meaning it lies outside this object. Vertices with this value will be part of the merged object, because they are not involved in any intersections.

on meaning it lies on the surface of both objects. Vertices with this value will be part of the merged object.

in meaning it lies inside this object. Vertices with this value will not be part of the merged object.

unknown which is the initial value.

For more convenient debugging, the `Vertex` class also contains printing methods for both the coordinates and the location.

r0.4

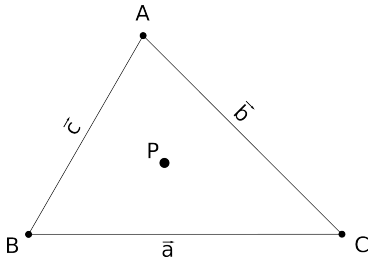


Figure 6.1: Labelled triangle.

6.2 Triangle

There are some pieces of information, which triangles should store:

The pointers to the **corner vertices** A, B, C are stored in a counter-clockwise order.
The **normal** is computed as the cross product of $\vec{b}$ and $\vec{c}$ and kept as a pointer as well.
The **area** is calculated as half the length of the normal.
The **centroid** P is the barycenter of the triangle. It is computed as $\frac{A+B+C}{3}$.

Additionally, each triangle also keeps an initially empty list of triangles, which are going to replace it. This is done to distinguish triangles that are tagged `remove` because they have been replaced by new, smaller triangles, and those which are tagged `remove` because they lie either inside one of the objects or directly on the surface of the object they are not part of.

Triangles receive a tag which contains status information about their prospective handling. This status tag, like the location tag for vertices, is an `enum` which can take values out of

check if the triangle is still eligible for further processing like intersections or saving.

remove if the triangle has been replaced with newer triangles if it lies inside the other object, or if it lies on the surface of both objects.

The algorithm requires meshes without overlapping or identical triangles. Hence, triangles should only appear once. To identify duplicates, we need an `equals()` function, which checks if two triangles have their vertices in identical positions.

During the calculation of intersection points, we have to get from the starting point to the intersection point, so we have to determine a scaling factor t, with which the edge has to be scaled. This factor is calculated by `calculateT()`.

In several parts of the program, barycentric coordinates are needed. They are computed by the `calculateBarycentricCoordinates()` function, which is described in section 4.4.

There are two `intersects()` functions: One implements the intersection of two edges, as described in Section 4.2, whereas the other implements the intersection of a line with a plane, using the Hesse normal form, as explained in Section 4.3. Both use `calculateT()` and, if $0 \leq t \leq 1$, it also uses `calculateBarycentricCoordinates()`.

If needed, `swapSides()` can swap the inside and outside side of a triangle. This is achieved by swapping two of the triangle's vertices. Now, the orientation of the vertices has changed and, if the normal was computed as $\vec{b} \times \vec{c}$ before, it can now be computed as $\vec{c} \times \vec{b} = -\vec{b} \times \vec{c}$

For more efficient debugging, the triangle coordinates and status can be printed.

6.3 Vector

The vectors used by the algorithm are 3-dimensional vectors, which keep their own direction, orientation and length. Furthermore, the `Vector` class offers functions for computing angles between two vectors as well as the dot and cross products. Like `Vertex` and `Triangle`, this class also provides an `equals()` function for checking whether the coordinates of two vectors are identical.

All the functionality of this class could have been provided by an existent library, but there were a few reasons why I decided against the use of libraries exceeding the Standard Template Library (STL).

First of all I wanted to facilitate the integration of the algorithm into the RAMSIS project, so any used libraries had to be open and available for both Windows and Linux. Secondly, it is not worthwile to include external libraries for just some functions that can be implemented rather easily or that are already provided by the RAMSIS system. And finally, it might have made sense to use a library for the vectors if I had also used one for triangles and vertices, but I could not find any library, which fully satisfied my needs for the algorithm without including a lot of unnecessary functionality as well.

6.4 OBJHandler

The `OBJHandler` class is the interface for the `obj` file format. It has functions for reading and parsing files as well as writing to files.

Moreover, it also provides the in-out-test, which tags every vertex according to its location in relation to an object, as described in Section 5.2.

6.5 Triangulator

The `Triangulator` class takes a triangle and intersection points. It checks, if any of the given intersection points is identical to one of the triangle's vertices. If this is the case, it removes them and then creates a number of new triangles dependent on the number and location in the triangle of intersection points that have been delivered. In Section 5.4, the different possibilities are described and illustrated. If there are several possibilities to triangulate a triangle, the locally optimal solution is selected. In the end, a list containing all new triangles is returned.

Due to time constraints, this class only implements triangulations for one or two intersection points. However, the cases with more intersection points can be simulated easily by recursively subdividing triangles with one or two of the intersection points that have been found.

6.6 MagicMeshMerger

The `MagicMeshMerger` (`MMM`) is the main class of the program. It carries out large parts of the central calculations and delegates the remaining tasks to the responsible components.

At the start of the program, it uses the `OBJHandler` to read the `obj` files and to get lists containing all vertices and triangles of each object, as well as lists containing a list of every group for each object. Then the `MMM` calls the `OBJHandler` again, giving it one object's vertices and the other object's triangles to carry out the in-out-test.

Then, it reduces the search space by restricting the evaluations to groups, which have overlapping bounding boxes. Only those triangles, which have at least one vertex inside this overlap, are checked, as we have seen in Section 5.1. All pairs of triangles from different groups which fulfil these criteria then undergo an intersection test, examining, which triangles intersect each other by using the methods described in Sections 4.3–4.4.

In case of an intersection, the intersection points are calculated and passed on to the `Triangulator` along with each triangle that has not already been tagged `remove`. Details about this process have been covered in Section 5.3.

After receiving all new triangles, the `MMM` decides, which triangles to keep and which to discard, as described in Sections 5.5 and 5.6. Then it gives the final set of triangles and vertices to the `OBJHandler` which writes the merged object to a file.

There are two functions in the `MagicMeshMerger` class, that are only used as helper functions and that do not require their own section in Chapter 5:
The bounding box of a set of points is determined by `getMinMax()`. This function finds

the smallest and largest value for each coordinate and returns them.
The test, whether or not two bounding boxes overlap is conducted by `calculateOverlap()`.
If the bounding boxes overlap, this function calculates the overlapping area, saves it to a local variable and returns true. Otherwise it returns false.

7 Conclusion

Now we have seen how the algorithm works, let us have a look at the results it produces. Afterwards we will see how it could be improved and at the end, there will be a summary of the results obtained with the merging algorithm.

7.1 Examples

In order to evaluate the performance of the algorithm, we need a number of test cases representing problems which can occur.

7.1.1 No Intersection

The constellation by far easiest to handle is that of two objects with bounding boxes which do not intersect at all. Such objects are immediately written to the file, without any further checks. The main difference between input and output is, that the input consists of two objects in two different files, whereas the output contains one object and writes them to one file. Thus, the `obj` file for the example in Figure 7.1 contains 16 vertices and 24 triangles.

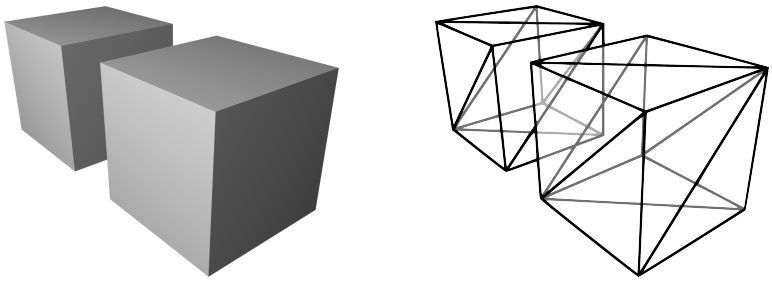


Figure 7.1: Merged object of two cubes which do not intersect.

7.1.2 Simple Intersection

This is an easy, common case. There are neither overlaps nor any shared elements between the objects. The only slightly special characteristic of this example is, that all

intersection points lie on edges. However, this does not need any special treatment. The intersection points are calculated and the faces are subdivided correctly, so we get the result seen in Figure 7.2.

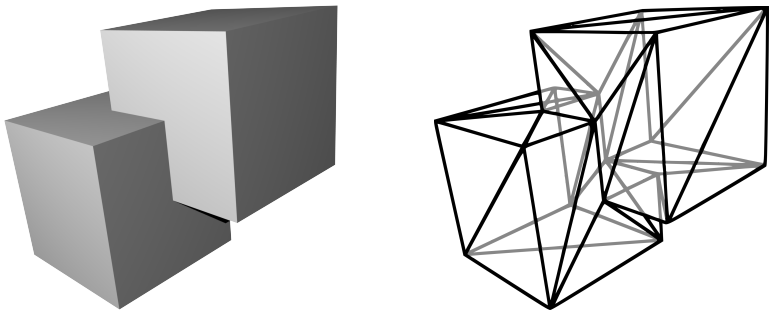


Figure 7.2: Merged object of two cubes intersecting in a simple way.

7.1.3 Shared Vertex

In this special case, the two objects touch in one point. The input has two different vertices for this point, since the points are read from different files. However, this case can be solved by checking the vertices for equality during the saving process. Since we work with `doubles`, which are imprecise, we have to allow a certain (small) difference between the coordinates. Otherwise they will not be identified as identical points. The result of this merging process can be seen in Figure 7.3. The `obj` file contains 15 vertices instead of 16, since two vertices have been merged, and 24 triangles.

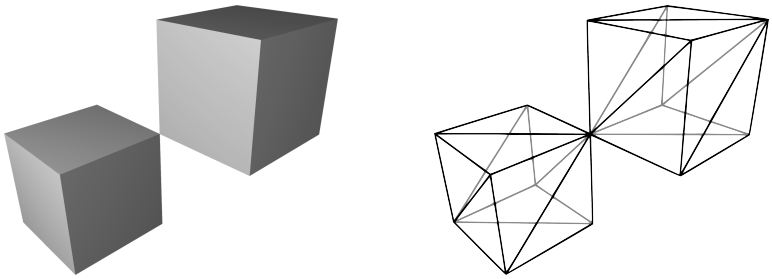


Figure 7.3: Merged object of two cubes which touch at one vertex.

7.1.4 Shared Edge

This case introduces a shared edge. The solution is quite similar to that of 7.1.3, except for the fact that we have two common points rather than one. Figure 7.4 displays the merged object. There are 14 rather than 16 vertices, as two pairs of vertices have been reduced to one vertex each, and 24 triangles in the `obj` file of the merged object.

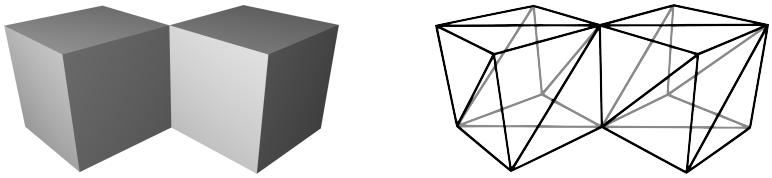


Figure 7.4: Merged object of two cubes sharing an edge.

7.1.5 Identity

As some movements, which are represented by RAMSIS manikins, do not involve the whole body, but only e.g. an arm, there can be a lot of groups being identical to each other. In this case, we only want to keep one version of the group. Figure 7.5 displays the result we get for merging two identical cubes. The `obj` file contains 8 vertices and 12 triangles.

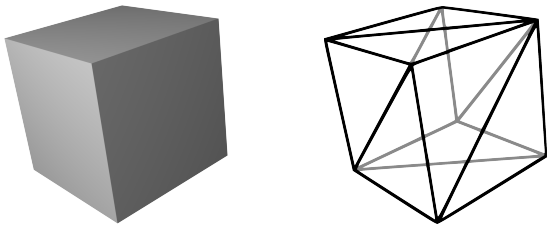


Figure 7.5: Merged object of two identical cubes.

7.1.6 Overlapping Face

Another possibility is that of overlapping faces. This can happen if objects are positioned like those in Figure 7.6. Then, we have to identify the intersection points and subdivide the corresponding triangles. During saving, the duplicate triangles are removed. In Figure 7.6, there are overlapping triangles on the top and bottom of the objects.

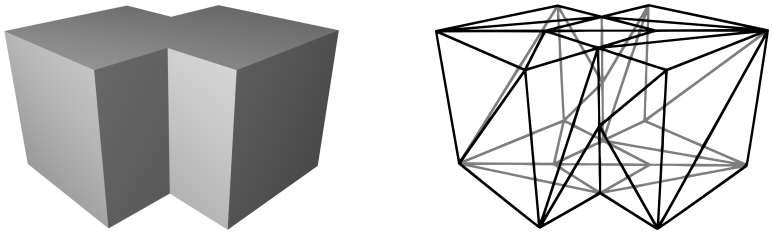


Figure 7.6: Merged object of two cubes which have overlapping faces

7.1.7 RAMSIS Mesh

The attentive reader might have noticed that there are no examples of merged RAMSIS manikins. This is, because unfortunately, the RAMSIS meshes have self-penetrations in several regions. These lead to triangles lying inside the original objects, so the manikins do neither satisfy the *hollowness* nor the *uniqueness* property. As already pointed out in Section 5.2, this leads to a wrong classification of vertices, so we produce invalid meshes. However, this problem can be solved by a modification of the algorithm, as we will see in a moment.

7.2 Further Work

All in all, the results the merging algorithm produces are quite good. However, there are several possibilities, how it could be modified or enhanced.

7.2.1 Reduction of Search Space

As a different method for reduction of search space to the one described in Section 5.1, we could use octrees rather than simply comparing the bounding boxes of groups.

Octrees are a tree-based structure. They allow us to recursively subdivide space into eight octants, until the desired level of refinement has been reached. Watt [9] uses a good visualisation of this subdivision process: Figure 7.7a shows how the subdivision is generally conducted and what the corresponding tree structure looks like. In Figure 7.7b an object is set into the initial cube. The cube is then subdivided as shown in Figure 7.7c, to match this object. The unlabelled nodes are the unused sub-cubes, the labelled ones are those we need for the object.

For this algorithm, the most detailed level of refinement that makes sense would include at least a few triangles. An optimal subspace would contain a managable yet meaningfully large number of triangles[1].

This method could be better for the reduction of the amount of data, which has to be processed. However, it would require bigger and more complicated data structures, which require more time for handling. Moreover, the results obtained by exploiting the group structure of the RAMSIS data are sufficiently performant.

Another problem that octrees could have is that the sets of triangles in their subspaces do not necessarily represent meaningful entities. E.g. a subspace could contain a few triangles from a manikin's hand as well as some from its thigh. Such random assignments

[1] An unvalidated estimate of a good number of triangles would lie between 20 and 50.

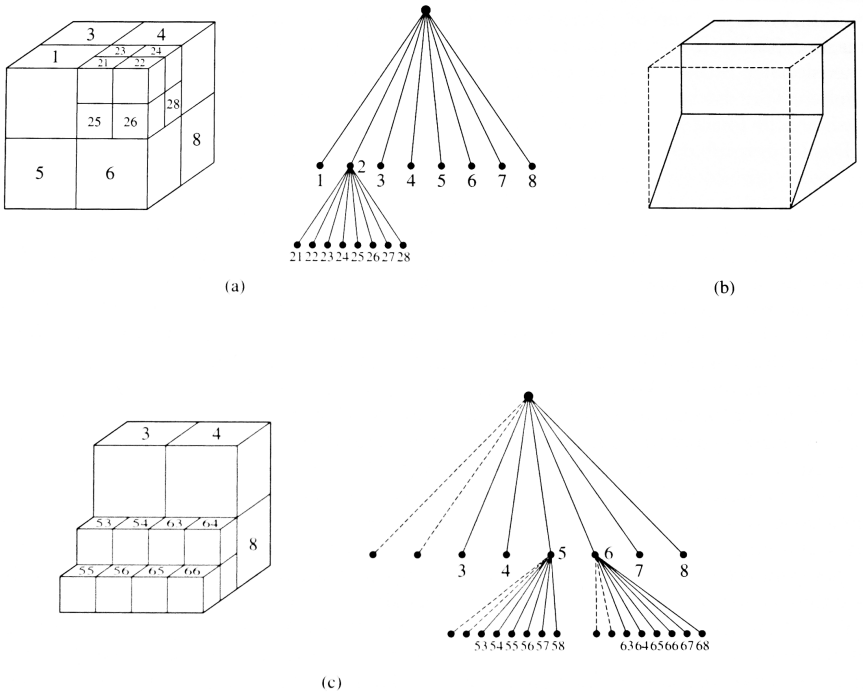


Figure 7.7: Recursive reduction of search space using octrees. (*Image source:* [9, p. 68])

lead to unconnected sets of triangles which do not form a closed hull of any part of the manikin's body. While this does not matter as long as the complete object is coherent, it is better to have semantically meaningful subdivisions.

For these reasons, I decided against the aid of octrees although this solution was the first idea I had.

Of course, if the RAMSIS data was not structured in groups, it would have been very helpful to use octrees, but the groups provide a very simple way to achieve a similar result without any additional computations.

7.2.2 Triangulation

Let us call a triangulation **optimal** if the smallest angles are maximal[2].
It is **globally optimal** if the optimality criterion is fulfilled for the whole mesh, and it is **locally optimal** if it there is a subset of the mesh for which the criterion is fulfilled.

Usually, it makes sense to avoid ending up with large numbers of narrow slivers, as they can lead to a number of numerical problems, e.g. rounding errors when trying to

[2] Alternatively, we could use different criteria, e.g. the smallest areas of the triangulation.

determine whether a point lies inside or outside the triangle. Since this classification is a core part of the algorithm, the errors would cause major problems.

The computed triangulation of the final object is not always optimal. As we can see in Figure 7.8, the triangulations of the side faces are not always the same, although the faces and the intersection points are very similar. The outcome depends on the order in which the intersection points are found.

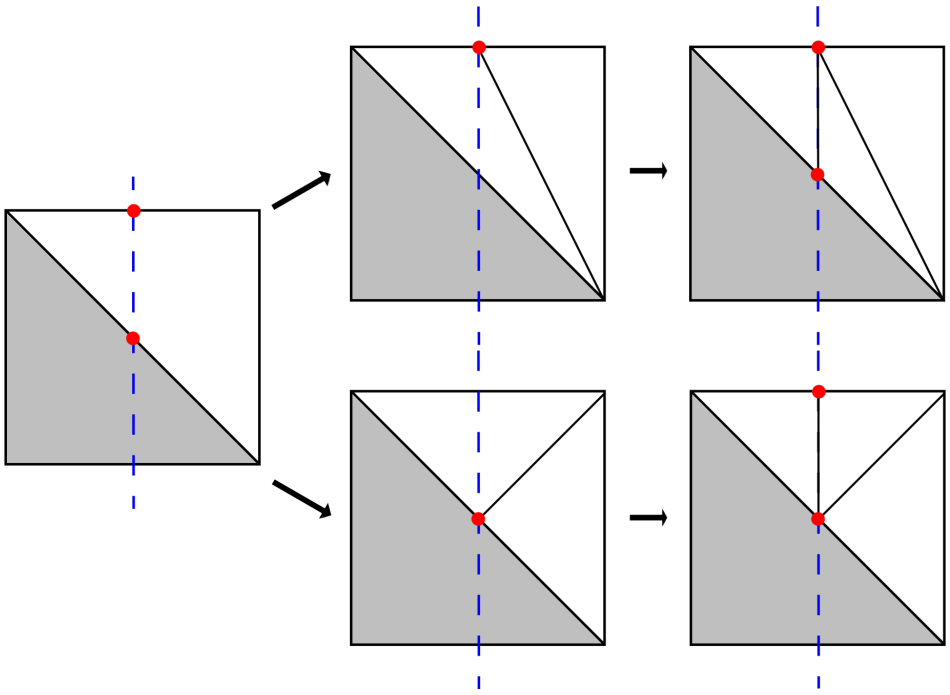
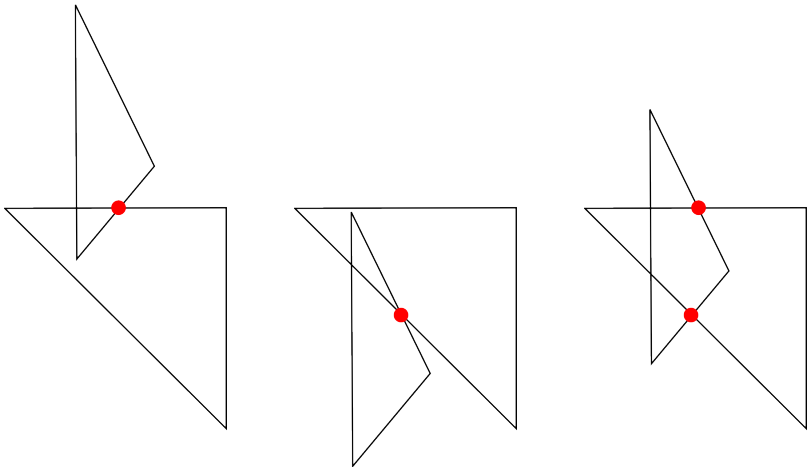


Figure 7.8: The triangulation depends on the intersection points' order of appearance. These are the original faces of the cubes in Figure 7.6. The lower, grey triangle is only added to resemble the original square faces.

In a situation like Figure 7.9a, the upper intersection point is found first because the triangle intersecting the affected one happens to appear before a different one, which would provide us with the lower intersection point. So, the affected triangle will be triangulated as shown in the upper part of Figure 7.8. As we can see, this triangulation is suboptimal with respect to the smallest angles.

In a situation like Figure 7.9b, which provides the lower intersection point first because the order of triangles is suiting, the triangulation will turn out like the lower one of Figure 7.8.

If, however, both intersection points are found simultaneously, the better triangulation is chosen automatically (Section 5.4.2). One example how this can happen, is depicted in Figure 7.9c.



(a) Situation providing the lower point first.

(b) Situation providing the upper point first.

(c) Situation providing both points at the same time.

Figure 7.9: Situations in which the different combinations of intersection points can occur.

Dealing with this problem on a more global scale is beyond the scope of this thesis. However, I have made an effort to keep the number of suboptimal triangulations low by choosing the locally better triangulation where this was possible (Sections 5.4.2 and 5.4.2).

One approach to obtaining a globally optimal solution would be running a mesh optimisation (by simplification) algorithm on the data after the merging process.

7.2.3 Preprocessing to Avoid Self-Penetrations

Self-penetrations seem to be a common problem in the area of mesh merging. We excluded meshes with self-penetrations and, as already mentioned in Chapter 3, Smith and Dodgson [6] exclude them as well. As already mentioned in 7.1.7, the merging algorithm cannot handle RAMSIS meshes due to self-penetrations. However, if we applied some preprocessing removing these self-penetrations, we would be able to apply our algorithm to manikins as well.

An easy way to conduct this preprocessing would be to adapt our merging algorithm. In general, the problem of self-penetration is, that there are triangles inside the object. So, the task is actually quite similar to that of merging, with the small but vital difference, that we want to preserve both the objects and their group structure.

If we conduct the modified merging algorithm for every pair of groups from an object, we can apply the main part of the original algorithm by pretending the groups belong to different objects. We would have to intersect triangles and classify them. In the

last step, we would deviate from the merging algorithm: Instead of writing all triangles which have to be kept into one list, we would write a list for every group. These lists then replace the old lists of triangles for these groups, and the old ones can be discarded. At the end of the modified algorithm, we can write each group to its object's file.

The resulting object should now contain all triangles which were on the surface of the original object as well as some new triangles which result from old ones that caused a self-penetration. All triangles and subtriangles which were inside the original object would not be part of the algorithm.

7.2.4 Context Preservation

Another useful addition to the algorithm would be a way of context preservation. The original RAMSIS meshes are divided into groups which have their own materials. These materials allocate semantics and colours to the group. E.g. skin is coloured beige, the upper part of the body is coloured orange, where a pullover would be worn, and the lower part of the body is coloured blue.

Preserving the allocation of material would require some additional information for each group or triangle. This could be the name of the material, so we would know which material to assign. Then, the file written at the end could contain groups, which use these materials.

This could be used for assigning priorities when designing the car interior. If, for example, a region was occupied by some part of the manikin, we could check which part it was. A region used by the head should definitely be large enough, as the head position should be a comfortable one for the vehicle occupant. In contrast, if it is used by a hand, but not close to the steering wheel, it could be less important (e.g. operating the radio), so the vehicle interior designers could assign a smaller priority to providing enough space for the hand than for the head.

Another way to apply this would be to assign one material to every manikin. Manikins with average proportions could be assigned a different colour than those, that are bigger or smaller than average. The designers would be able to see which spaces are occupied by what kind of passenger. Then, they would be able to adapt the vehicle interior to the target group they want to address with a specific type of vehicle. A car targeting females can, on average, be smaller inside than a car that is targeted at men.

7.3 Summary

As we have seen, the merging algorithm successfully merges meshes which fulfil the coherence, hollowness and uniqueness criteria.

First, it effectively reduces the search space, taking advantage of the structure of the input data.

Then, the algorithm determines, which triangles intersect each other and calculated the corresponding intersection points. There are two ways to calculate them, depending on whether or not the triangle lie in the same plane.

At the end, the vertex locations (in relation to the respective other object) for both the triangles' vertices and their centroids are utilised to classify the triangles. If the vertices or centroids lie inside the hull of the merged object, the triangles are discarded, and all triangles that are part of the hull, are saved.

The results obtained with the algorithm look very promising, however, as for most applications, there still are some aspects which require further improvement. A particularly interesting enhancement for our application can be added using a preprocessing algorithm for removal of self-penetrations. This would allow us to process the manikins provided by the RAMSIS system. For other applications with massive amounts of data, implementing the octrees would improve the performance considerably.

Beyond the application in vehicle occupant simulation, the algorithm can be applied to any problem that requires to merge triangulated meshes, as the implementation is very general. One such example is the area of 3D modelling. Here, we can obtain quicker results during rendering if the number of triangles is reduced. Thus, applying our merging algorithm to objects which are used a lot and have to be rendered could indeed produce a better (i.e. quicker) result.

Index

Bibliography

[1] Standard Template Library Programmer's Guide, 1994. [Online; accessed 18-May-2009].

[2] RAMSIS - Benefits and Advantages. Technical report, Human Solutions GmbH, August 2004.

[3] Michael Bender and Manfred Brill. *Computergrafik*. Hanser, second edition, 2006.

[4] Gerald Farin. *Curves and Surfaces for CAGD — A Practical Guide*. Academic Press, fifth edition, 2002.

[5] Jan-Holger Gründler, Lutz Kasper, Christina Schwalm, Dr. Claudia Seidel, and Grit Weber. *Das große Tafelwerk interaktiv*. Cornelsen Verlag, first edition, 2003.

[6] J. M. Smith and N. A. Dodgson. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. *Computer-Aided Design*, 39:149–163, 2007.

[7] Dr. Hartmut J. Speyer. RAMSIS - Anthropometric Types. Technical report, Human Solutions GmbH, May 2006.

[8] Wavefront Technologies. *Wavefront .obj file format specification for the Advanced Visualizer software.*

[9] Alan Watt. *3D-Computergrafik*. Pearson Studium, third edition, 2002.