UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 1534

THREE-DIMENSIONAL MESH CUTTING IN VIRTUAL SCENE

Bruno Pregun

Zagreb, June 2017

SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1534

REZANJE TRODIMENZIONALNIH MODELA U VIRTUALNOJ SCENI

Bruno Pregun

Zagreb, lipanj 2017.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 3 March 2017

MASTER THESIS ASSIGNMENT No. 1534

Student:	**Bruno Pregun (0036459671)**
Study:	Computing
Profile:	Computer Science

Title: **Three-Dimensional Mesh Cutting in Virtual Scene**

Description:

Investigate techniques for cutting three-dimensional objects in a virtual scene. Peruse and describe methods for mesh cutting especially in the context of generation of cutting surfaces with appropriate texture generation. Implement the investigated models. Implement a framework for analysis and comparison of the investigated models. Show examples of results. Evaluate the results and the implemented algorithms.
Implement the appropriate software solution. Use the appropriate graphics tools, APIs and libraries. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

Issue date:	10 March 2017
Submission date:	29 June 2017

Mentor:

[illegible]

Full Professor Željka Mihajlović, PhD

Committee Secretary:

[illegible]

Assistant Professor Tomislav Hrkać, PhD

Committee Chair:

[illegible]

Full Professor Siniša Srbljić, PhD

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 3. ožujka 2017.

DIPLOMSKI ZADATAK br. 1534

Pristupnik:	**Bruno Progun (0036459671)**
Studij:	Računarstvo
Profil:	Računarska znanost

Zadatak: **Rezanje trodimenzionalnih modela u virtualnoj sceni**

Opis zadatka:

Proučiti metode rezanja trodimenzionalnih modela u virtualnoj sceni. Opisati algoritam za konstrukciju i teksturiranje novih ploha dobivenih rezanjem. Ostvariti programsku implementaciju opisanog algoritma te izraditi aplikaciju koja vizualno demonstrira njegovo korištenje. Na različitim primjerima prikazati ostvarene rezultate. Načiniti ocjenu rezultata i implementiranih algoritama.
Izraditi odgovarajući programski proizvod. Po potrebi koristiti grafičke programske alate. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku:	10. ožujka 2017.
Rok za predaju rada:	29. lipnja 2017.

Mentor:

[illegible]

Prof. dr. sc. Željka Mihajlović

Djelovođa:

[illegible]

Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za diplomski rad profila:

[illegible]

Prof. dr. sc. Siniša Srbljić

CONTENTS

1. Introduction

One of the most extensively used methods for displaying three-dimensional objects in a virtual scene is using a polygon mesh. That is why the study of polygon meshes is a large part of computer graphics and related fields.

There are different ways to represent polygon meshes and each of them is used for different applications and goals. In most cases, polygonal meshes define the surfaces of a solid virtual object, and therefore do not explicitly represent it’s volumetric structure.

Polygonal meshes can be constructed from multiple types of elements. Such meshes contain vertices which are three-dimensional points in a virtual space, edges which represent connection between two vertices, and faces which are a closed set of connected edges. Faces usually contain three edges making them a triangle face, but most model formats also support faces with four edges (called a quad face). Some systems allow faces to have even more sides, but those are often represented as multiple triangle or quad faces, since this is supported by most rendering hardware. An example of a mesh that is constructed out of triangle and quad faces can be seen in Figure 1.1 below.

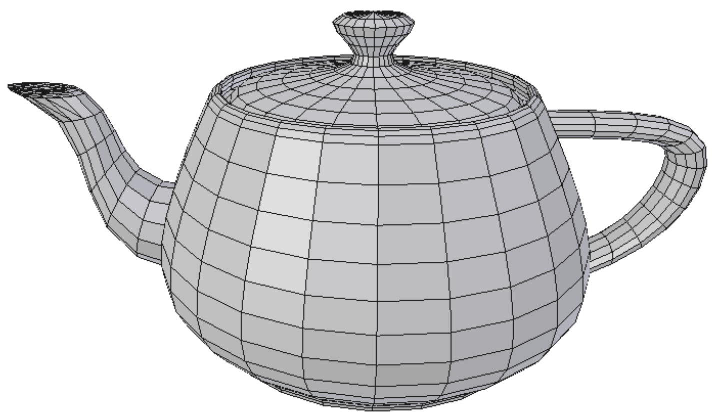

Figure 1.1: Example of a polygon mesh representing a teapot

Except for rendering such polygon meshes on screen, their application usually requires various operations to be made on the mesh structure. Operations include boolean logic (often called Constructive Solid Geometry), smoothing (surface subdivision), simplification (level of detail), and many others. One such operation is mesh cutting. This operation consists of cutting a polygon mesh into two separate parts using a plane and has various applications [1]. It can be used as a visual aid to convey three-dimensional information that would otherwise be difficult to display (for example slicing a volumetric graph or showing an interior of a complex object), or in physics simulations for breaking objects into smaller parts. It can be applied multiple times on the same object and can also be a part of other, more complex operations.

This thesis describes the steps and algorithms required to apply the mesh cutting operation on a virtual object and presents a program solution implementing the described methods.

2. Mesh cutting algorithm

Cutting a mesh requires multiple sub-operations. The first step is to define a polygon mesh and the plain along which the mesh will be cut. The second step is to actually cut the faces that intersect the plane and split the mesh into two separate parts. The third step consists of constructing an interior geometry that was revealed by the cut. The first two steps will be presented in detail in this chapter, while the third step will be discussed in the subsequent chapter.

When cutting a mesh, it is split into two parts, which means we have to remove the original mesh from the scene and replace it with two new meshes that represent each of the parts created by the cut. Since the cuts most often do not happen along the edges of the mesh, it is highly likely that we need to create additional geometry near the area that was cut, adding more triangles to the mesh. Once the new geometry is constructed we have to split it into two separate meshes. This is done by assigning each triangle a side it belongs to, depending on which side of the plane it was on before the cut.

2.1. Mesh specification

While there are many ways to represent three-dimensional virtual objects in the field of computer graphics, the most often used approach is to use polygons to represent the surfaces of the object. While some other approaches, e.g. point clouds, make it easier to perform operations like cutting, widespread use of polygon meshes in various applications forces us to come up with ways to implement these operations for such cases. Therefore, all of the techniques described in this thesis will assume we are working with polygon meshes. For this purpose, we will assume that each mesh is defined by two elements – vertices and triangle faces. Each mesh is constructed from a set of vertices which hold information about the shape of the object. As per usual, each vertex holds three vectors that store data.

One vector describes the position of the vertex in three-dimensional model space. Another vector describes the three-dimensional direction of the surface normal at that point. The last vector is a two-dimensional vector that holds the uv values required for texture mapping which will be discussed later. For our purposes, one vertex is, therefore, a set of eight floating point numbers. Of course, algorithms described in this paper work on any type of vertex as long as it has at least the position information, but the two other vectors are used to demonstrate how the algorithm can be applied onto a mesh that also contains some non-positional data.

To define the surfaces of our mesh we also need faces. These faces connect vertices and form a surface. Since every polygon face can be separated into multiple triangle faces and triangle faces are simple to work with, we will assume that every face on a mesh is a triangle. Triangle faces are simple to work with because they are supported by most, if not all, graphics hardware. This approach also guarantees that every face will be planar, which will prove useful for the task at hand. Converting a mesh with multiple-sided polygon faces (usually quads) into one with triangle faces is possible and is an operation supported by any 3D modelling software. More details about this operation will be given in the section describing polygon triangulation in the forth chapter.

For our purposes each face of a mesh will be a set of three vertices that make up the tips of a triangle. Vertex data will be interpolated according to the barycentric coordinate system defined by the triangle vertices in order to get the information about the shape of an object at any point in 3D space.

When loading a model from file, vertices are stored and indexed, assuring that we do not store the same vertex twice. That gives us a way to reuse the same vertex for multiple faces by just using its index multiple times, saving on memory. Therefore, in our application, faces are defined by three indices (integers). Those indices point to the vertices that the face uses. Model structure is described in more detail at the beginning of the forth chapter.

To recap, each vertex is three vectors (8 floating point numbers), each face is three indices (3 integers). This is all we need to define our polygon mesh – all other elements of the mesh that we might need, such as edges, can be inferred from this data.

2.2. Triangle Cutting

The process of cutting a single triangle with a plane is relatively simple. Before any intersection is made, we have to define each triangle edge. That is accomplished by acquiring the vertex data and connecting each of the three vertices together in a triangle. Once the edges are constructed and we have the data about each vertex, we proceed to intersect them with a cutting plane. We do that by using the following formula:

$$d=\frac{(\vec{p}-\vec{a})\vec{n}}{(\vec{b}-\vec{a})\vec{n}} \qquad \begin{aligned}&\vec{n}-\textit{cutting plane normal}\\&\vec{p}-\textit{point on a cutting plane}\\&\vec{a},\vec{b}-\textit{edge vertices}\end{aligned} \tag{2.1}$$

This formula not only tells us if the edge is intersected by a plane but also at what point it is intersected at. This is extremely helpful because it allows us to easily construct a new vertex at that exact point. We apply the formula to each edge of the triangle.

Since the plane splits the space into two parts, a part below the plane and the part above it, there are three possible outcomes for each of the triangles. Either the entire triangle is above the plane, below the plane, or intersected by a plane. The situation in which the triangle isn't being intersected by a plane are resolved trivially, by simply assigning the entire triangle to its appropriate side of the plane. However, if the triangle is intersected, additional steps will be required. Every intersected triangle always has one of its vertices on one side of the plane, while the other two vertices are on the opposite side. Another way to say this is that, assuming that the plane is infinite, every intersected triangle always has two edges that are intersected, and one edge that is completely on one side of the plane.

This means that we can only have two of the edges intersected, never just one or all three. Once these two edges are found, we have to construct two new vertices at the place the edges were intersected.

The intersection formula (2.1) gives us a value between 0 and 1 if the edge is intersected. If the result is outside that range, the edge is not intersected by the plane. A value closer to 0 gives us the intersection near one end of the edge, and a value closer to 1 gives us the intersection near the other end. This means the resulting number can be used to determine at which part of the edge the intersection occurred. This allows us to construct the new vertices using a simple linear interpolation formula:

$$x = b\,t + a\,(1 - t) \tag{2.2}$$

The position of the new vertices are interpolated according to the vertices at the each end of the intersected edge. Other vertex data is filled in depending on the purpose the algorithm is used for. In our case, normal vector and uv values are also interpolated. In some other cases there might be other types of data that we might want to copy from the closest vertex instead of interpolating, but we do not have such data in our example.

After the new vertices are interpolated using (2.2) and added to the mesh, we have to reconstruct the faces that we got from cutting the triangle. The original triangle face is removed from the mesh and discarded, while three new are added. Cutting a triangle always results in a smaller triangle and a quad. The quad is split into two triangles; therefore, each intersected triangle gives us three new triangles. These triangles are then split into two groups, one triangle on one side, and the two triangles that make a quad on the other side of the plane.

Triangle cutting operation is done for every triangle in the mesh. As each triangle is evaluated, or as a new triangle is constructed, it is assigned to the appropriate side of the mesh. Since we are using indexed vertices, this is done by having the operation result in two arrays of vertex indices, one for each side of the cut. The implementation details, as well as the benefits of this approach, will be discussed in the fourth chapter.

What follows is an example of cutting a mesh consisting of three triangles.

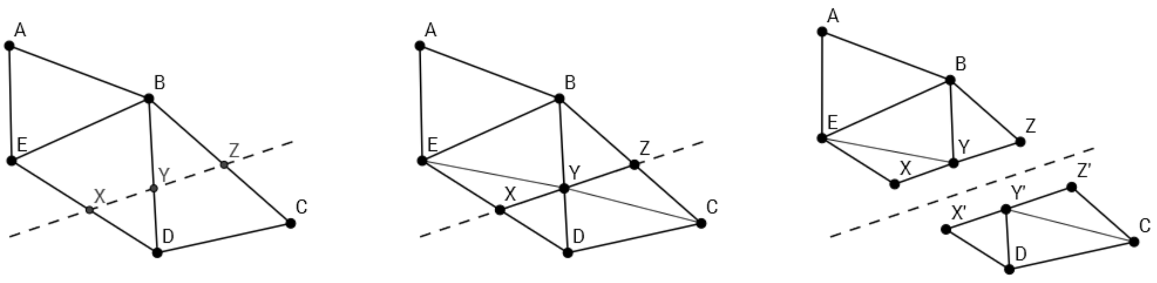

Figure 2.1: Process of cutting a simple triangle mesh

The first step, as shown on the left side of figure 2.1, is finding the points where the triangle edges intersect with the plane, which is represented as a dashed line. Triangle ABE is not intersected with the plane, which means it belongs completely to one side, while the other two triangles (BCD and BDE) have to be cut. The middle step shows how new edges can be constructed for each triangle. Edges XY and YZ are added as a result of cutting the triangles. Since we only want triangle faces in our mesh, we have to convert quads BYXE and CDYZ that appeared into corresponding triangles. Lastly, shown on the right, is a separation of two meshes that are made by the cut.

3. Interior face construction

Just cutting the existing triangles isn't enough in most applications. Since closed polygon meshes have an implied volume, it is necessary to maintain the illusion of that volume after the cut; otherwise the virtual object would appear hollow. This is done by examining the polygon along which the mesh is cut and filling the gap with new additional surfaces. As seen in Figure 3.1, the mesh without the interior face appears hollow after a cut.

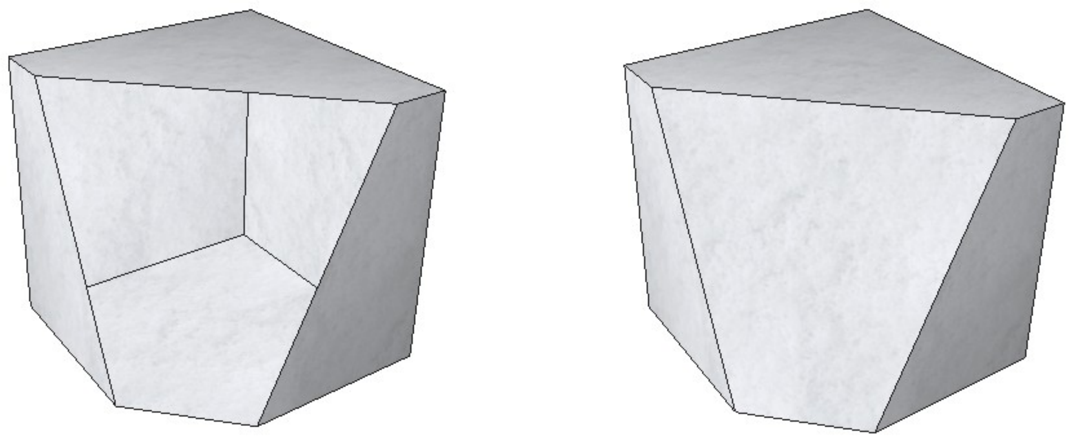

Figure 3.1: A mesh without (left) and a mesh with an interior face (right) after a cut

3.1. Polygon construction

Intersecting a triangle with a plane results in an edge. Our implementation of the mesh cutting operation remembers all edges that appear and joins them up in one or more closed loops by joining up the edges that share the same vertex. These loops define planar polygons that lay on the cutting plane. Once we have the polygons, we need to convert them into triangles so we can display them and cover up the gap that appears after the mesh is cut.

3.2. Polygon triangulation

Decomposing a simple polygon into a set of triangles is a classic problem in computer graphics, and many different solutions exist for it. We define a simple polygon as an ordered sequence of planar vertices, where consecutive vertices are connected by an edge and an edge connects the first and last vertex. A polygon is simple if the only place where edges intersect is at the vertices and each vertex shares exactly two edges. In our examples the polygons we deal with will in most cases be simple, unless the meshes are needlessly complex.

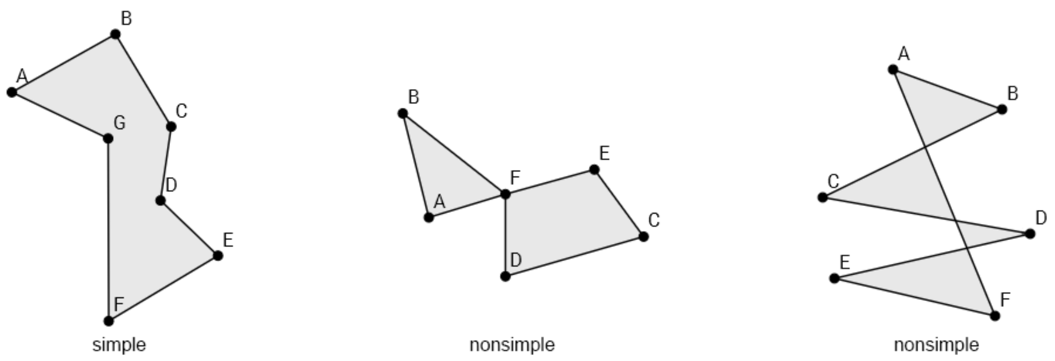

Figure 3.2: Simple and nonsimple polygons

Figure 3.2 shows three polygons. The one on the left is a simple polygon. The polygon in the middle is not simple because one of its vertices shares more than two edges. The polygon on the right is also not simple because its edges intersect at points that are not its vertices.

One solution for triangulation of a simple polygon could be to connect one of its vertices to every other vertex it is not already connected to, thereby creating triangles. The result of such a triangulation is shown in Figure 3.3.

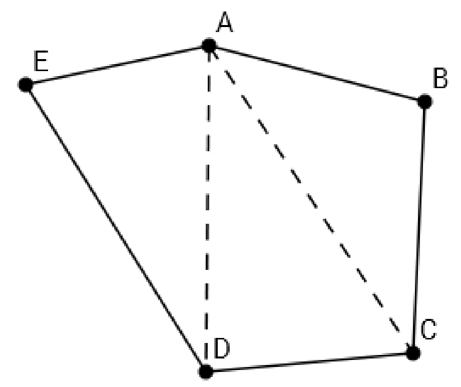

Figure 3.3: Naive triangulation

Unfortunately, this method only works for convex polygons. Since we can often get polygons the vertices of which are not all convex, we cannot use this method reliably and need a better solution. The solution comes in the form of a so-called ear clipping algorithm [2].

3.3. Ear clipping algorithm

Ear clipping is a recursive approach to the triangulation. It consists of finding an ear of a polygon and removing it, until no more ears are left. An ear of a polygon is a triangle that is formed by three consecutive vertices, for which the second one (called the ear tip) is convex, and no other polygon vertices are contained in that triangle. A polygon vertex is convex if its interior angle is smaller than 180 degrees. A triangle consists of one ear and we can place the ear tip at any of its three vertices.

It is proven that a polygon with four or more sides always has at least two non-overlapping ears. The ear clipping algorithm suggests that we can triangulate any simple polygon by recursively removing its ears. If we can locate an ear in a polygon with $n \geq 4$ vertices and remove it, we are left with a polygon with $n - 1$ vertices and can repeat the process. The removed ears become the triangles that we replace the polygon with.

The ear clipping process used in the application will be demonstrated with a following example:

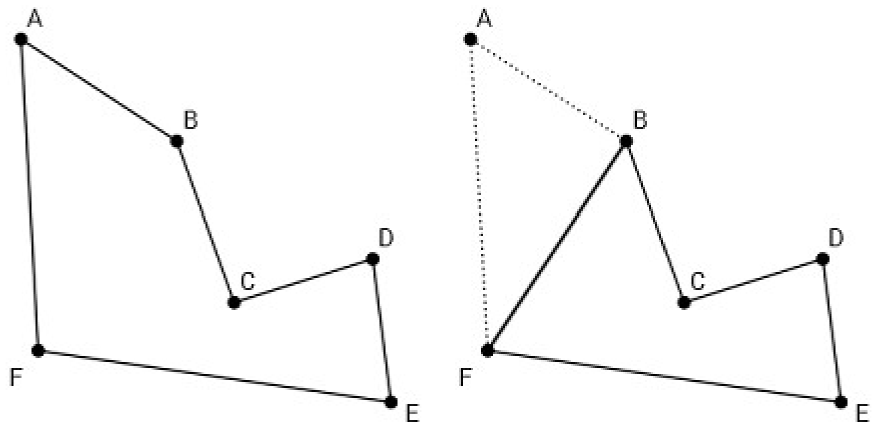

Figure 3.4: Removing the ear FAB from the polygon

The left side of Figure 3.4 shows the polygon we start with. This polygon is searched for ears. Vertex A is convex and no other vertices are contained within its triangle FAB. When considering a triangle for an ear, the triangle label is written with the potential ear tip as the middle letter. For example, vertex A is the tip of an ear designated as FAB. We have found the ear and we remove it.

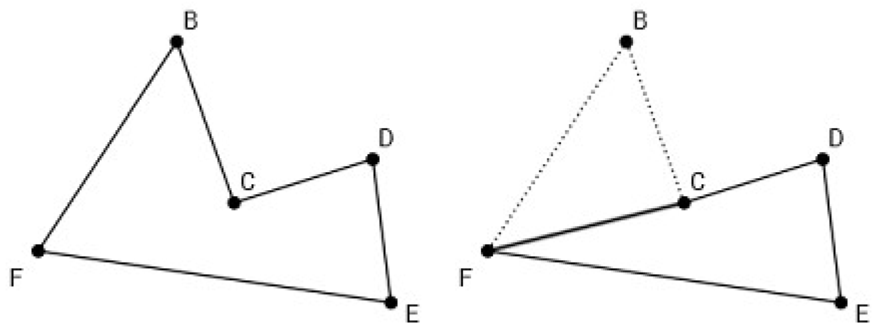

Figure 3.5: Removing the ear FBC from the polygon

Figure 3.5 shows the next iteration. The ear FBC is found and removed, leaving us with a quad.

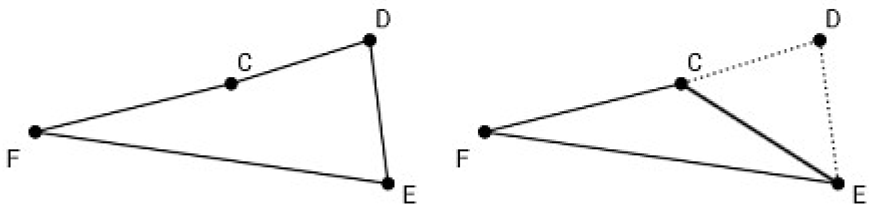

Figure 3.6: Removing the ear CDE from the polygon

The last iteration is shown in Figure 3.6. Vertex C cannot be an ear tip, because it is not convex; therefore, triangle FCD is not an ear. We test the next vertex. Vertex D is convex and no other vertices are contained within its triangle CDE. We have found the last ear and are left with a polygon with three vertices, which is our last triangle. The result of our triangulation is shown below in Figure 3.7.

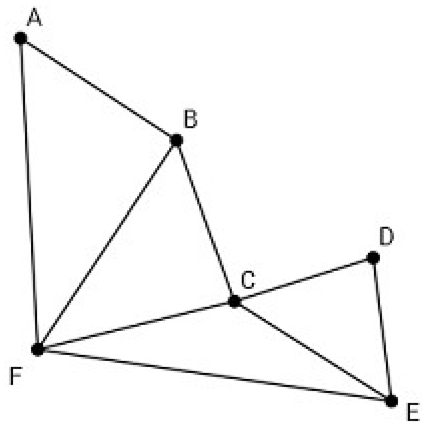

Figure 3.7: Complete triangulation of a polygon.

The ear clipping algorithm is not the fastest way to triangulate a polygon, but is simple to implement in code. Since our application does not require the triangulation of a large number of polygons for each cut, it is an acceptable compromise. We could speed up the algorithm by only considering reflex vertices (ones for which the interior angle is larger than 180 degrees) in the triangle containment test, but that would require having additional arrays that would have to be updated after every ear is removed, and would needlessly complicate the algorithm. The application code is written in such a way that it allows for easy replacement of the triangulation algorithm if the need for it arises in the future.

Even though the ear clipping algorithm does solve our problem with triangulation, some problems can still arise because we do not consider the possibility that our polygons might have holes in them. It is possible to update the algorithm so that it takes into account multiple polygons and treats the ones that are completely contained inside the others as holes. That would require keeping more information about the polygons and constructing a polygon hierarchy, because each polygon can have multiple holes and each of those holes can have multiple polygons with multiple holes within them. These hierarchies can easily get out of hand. Another fact that complicates this further is that, when cut, regular models can result in interior polygons for which we cannot tell if they are supposed to be holes or normal polygons. Geometries can connect and overlap in unexpected ways and there is nothing that would help the algorithm decide how to construct the interior faces. The solution we propose is to keep the models clean and simple. If they have to contain hollow elements, the vertices should be connected in a way that guarantees that any cross-section of the model results in simple polygons without any holes.

3.4. Geometry construction

Once the interior polygon is constructed and triangulated, what is left is to create the actual mesh geometry from it. This is done by first adding each vertex of a polygon to the mesh. We cannot use the vertices already in the mesh since the interior polygons will have different normal direction and uv values – this is why we must create new ones. The position of the new vertices is determined by the polygon vertices. Their normals are made to face perpendicular to the cutting plane; that way we get an appearance of a clean sharp cut. How we threat the uv values of the new vertices depends on what we are trying to accomplish. We can either try and unwrap the polygon along some predefined texture that defines the look of the meshes' interior or we can use three-dimensional volumetric textures. More details on texturing will be presented in the following chapter.

4. Implementation details

One of the goals of this thesis was to provide an implementation of the algorithms described in this paper. As such, the example application was developed as part of this thesis. The application was written in C++ and uses OpenGL for graphical rendering. Several third-party libraries were also used. GLFW [3] provides a window, OpenGL context and the ability to receive user input. Its code is Open Source, multi-platform and licensed under the zlip/libpng license. Another library is gl3w which is a public domain API that offers simple OpenGL core profile loading.

4.1. Mesh loading

There are many different formats for storing mesh data. Since mesh storing and loading is not the focus of this paper, we opted for the OBJ format. Wavefront OBJ file format is a geometry definition file that allows simple and open way to store and load mesh data. It is widely supported and easy to implement without the need for additional third-party libraries. The most common elements of an OBJ file are geometric vertices, texture coordinates, vertex normals and polygonal faces, which is all that we need to construct a mesh. Usually the model first lists the vertices and their position, then texture coordinates, and then follows with the vertex normals. The faces are then defined by specifying which one of these values each polygon vertex contains using indexing. That means that each data vector can be shared across the polygon vertices. Since OpenGL can only index on per vertex basis, this data needs to be properly stored and aligned in memory before being rendered on screen.

First, all vertex data is read into separate arrays and face indices are stored. Only then do we proceed with actual indexing of vertices and joining each necessary data vector together. With that we go from having an index for every data vector (position, uv, normal) to having an index for every vertex instead.

Indexing assures that each vertex is unique and is not listed twice in our data; instead, it is reused by using its index. This saves up on memory, but it requires two buffers, instead of one. We do not need just one buffer for the vertices, but we also need one for the indices, as well. Splitting the model into two buffers actually helps us even more, because when we cut an object, we can have each part of the model share the same vertex buffer, but have a different index buffer. This saves us from unnecessarily copying the data between parts during the cut, and gives us better performance. In short, objects can share the starting mesh and its vertex buffer, but each object has its own index buffer. Once they are cut, they still share the same vertex buffer, but their index buffer changes, since they are now indexing different vertices. New vertices created by the mesh cutting algorithm are added to the vertex buffer shared by all the objects with the same mesh; this ensures that all parts have access to the same vertices if needed.

4.2. Textures

Along with its mesh, another important element of a virtual object is its texture. Textures give more definition to the object’s surfaces and have a huge impact on the object’s appearance. In the example application models are unwrapped as usual in the 3D modelling tool and the texture coordinates are stored inside the vertex parameters.

Problems arise when trying to texture the interior parts of the mesh that are made visible by the cutting operation. We would like to somehow define the interior structure of the model. There are many ways this could be done and the best approach depends on the intended application. If the cutting algorithm is used for special effects, for example, in video games or simulations where complete accuracy isn’t required, just defining an interior texture that is projected onto the interior polygons could be enough. In other cases, where the application requires more accurate representation of the interior structure, other methods should be used, for example a three-dimensional volumetric texture.

Projecting a two-dimensional interior texture onto an interior polygon is trivial, but requires that we set up the uv values of the interior polygons vertices correctly. We get the right uv values by projecting the three-dimensional vertices of the polygon onto a two-dimensional cutting plane. The only thing we have to pay attention to is the scaling of the texture, since we probably do not want smaller polygons to have smaller texture detail. In our implementation we use negative uv values to designate that the triangle is using the interior texture, thereby avoiding the need for additional vertex data or multiple draw calls.

The problem with the two-dimensional texture approach is that, except for the texture image, we have no control over how the interior of the mesh will appear. Every cross-section of the mesh will have the same texture. One of the solutions for this is having multiple interior textures distributed along the mesh, and then picking the one closest to the cutting plane. This would give more control over the interior structure, while still maintaining the simplicity of two-dimensional textures. If the mesh has a skeleton for animations, a reasonable approach would be to define an interior texture along each bone, thereby giving each limb a unique interior structure.

There are situations where we simply cannot get the desired result with the described interior texture method. For example, if we want to have complex interior structure or if we want to precisely align everything with the outer textures. In these cases, it would be best to use three-dimensional textures [4] instead. Three-dimensional or volumetric textures work like regular two-dimensional textures, except with an added dimension. Instead of two texture coordinate values (u, v), they use three (u, v, w). Their biggest advantage is that they can define a volume of colour data and are, therefore, ideal for defining the interior structure of objects. This flexibility, however, comes with a cost in the form of higher memory requirements, which is why they are usually limited to lower resolutions. Another downside of using volumetric textures is the fact that they are difficult to create from an image or by a texture artist. As a result of this, they are not as easily available as regular textures are.

4.3. Virtual camera

To display a virtual scene we need a camera. In the example application, the camera features two modes. Orbiting mode tethers the camera to the target object and makes it orbit around, allowing the user to easily position the camera at the desired angle and cut the mesh from any side. Flying mode releases the camera from the target and allows the user to freely position it in space. The camera is controlled via a combination of mouse and keyboard. Holding the right mouse button and moving the mouse changes the camera’s orientation orientation. Keys **W** and **S** move the camera forwards and backwards. Additionally **A** and **D** keys are used in flying mode to move the camera sideways. Camera modes are switched using the **F** key.

4.4. Mesh Cutting

A scene is composed of virtual objects. Each object contains information about its mesh, textures, and a collection of sub-objects called fragments. An object starts out as having a single fragment that represents the entire mesh. Once the object is cut, its fragment is split and replaced with two fragments, each containing a different set of indices, representing each partition. Since objects can be cut more than once, each additional cuts splits the affected fragments further, increasing the number of fragments in the scene. An object contains a mesh, which means it has a vertex buffer that all its fragments can share. Each fragment contains the index buffer that indexes vertices from the vertex buffer. The structure of objects in the scene and their sub-objects is shown in Figure 4.1.

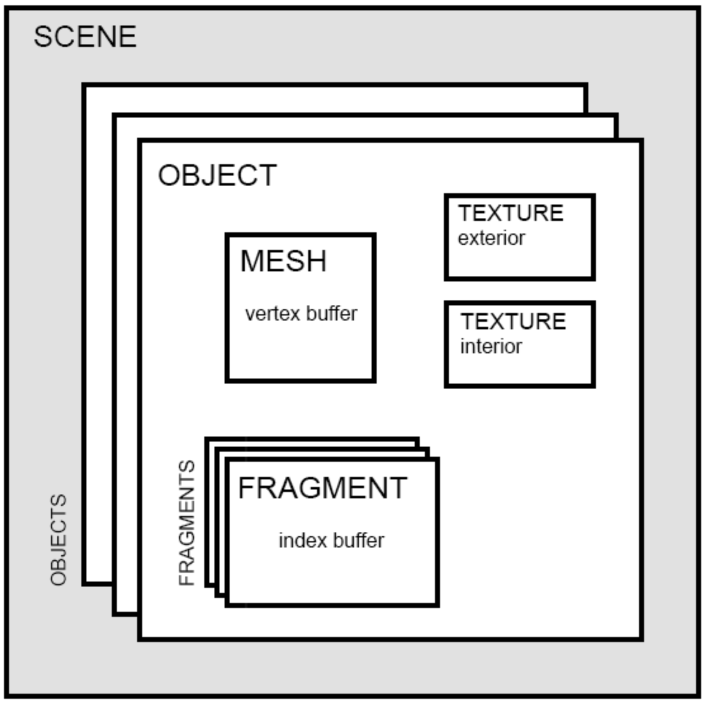

Figure 4.1: Structure of objects and fragments in a scene

When an object is cut, it delegates that operation to its fragments. Each fragment processes the cut and splits accordingly. The cutting operation follows certain steps, which are presented in the pseudocode below.

```
1.  For every object in scene
2.      If affected by the cut
3.          For every fragment of the object
4.              If affected by the cut
5.                  For every triangle in fragment
6.                      Cut triangles
7.                      Split triangles into new fragments
8.                  Construct interior faces
9.                  Replace cut fragment with new ones
```

In the example application, cutting is controlled by dragging and releasing the left mouse button across the scene. A cutting surface is constructed according to the direction of the dragging motion and the position of the camera. To better demonstrate the mesh cutting operation, several special effects were implemented. First, while the dragging is being performed, a line appears on the screen, indicating where the cut will be executed. Second, after the cut occurs, affected fragments are separated with a basic physics simulation, giving the impression of breaking up the object and allowing a better view of the object's interior.

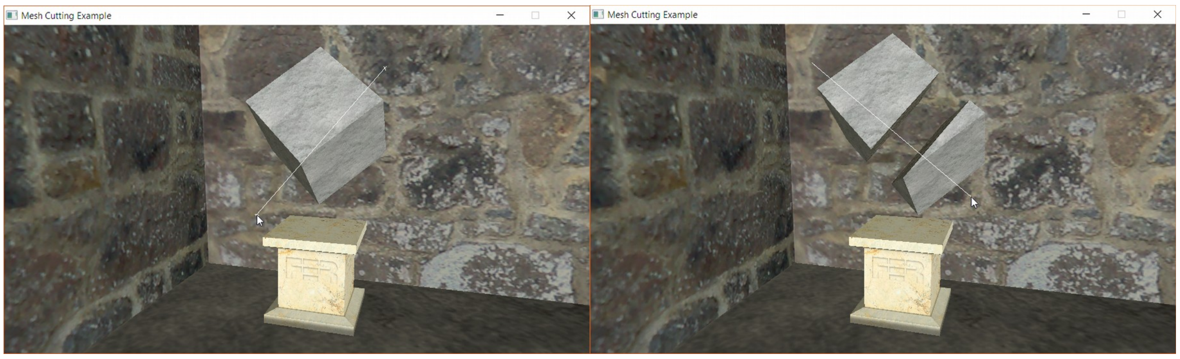

Figure 4.2: Two cuts on a simple cube mesh

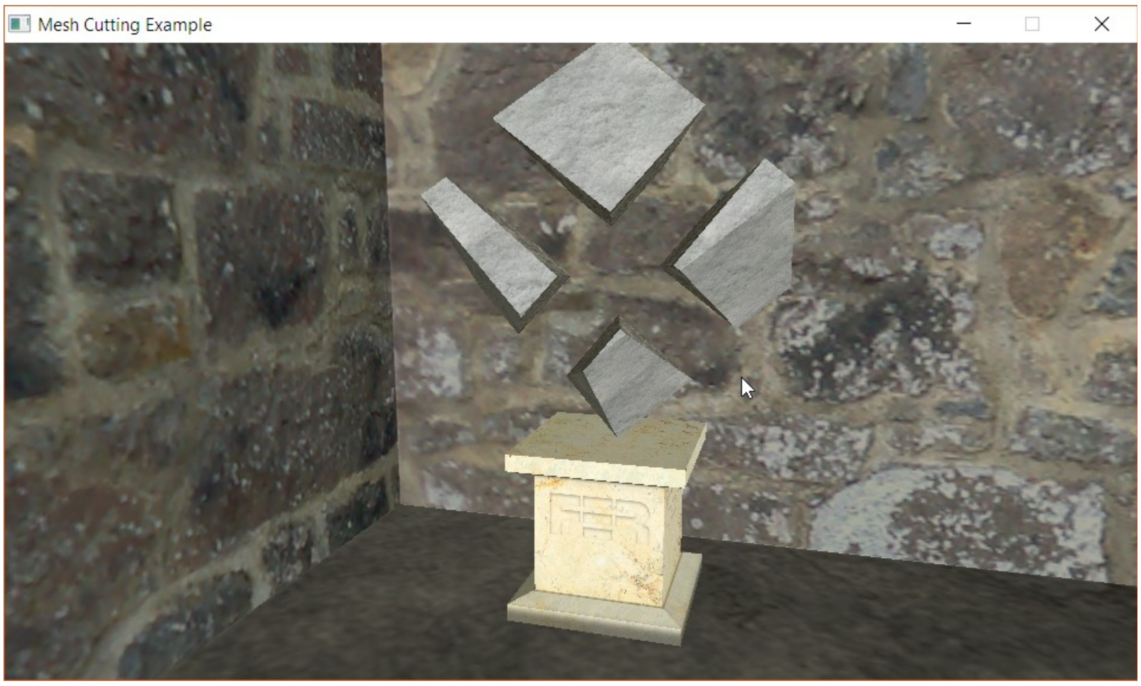

Figure 4.3: Final state after two cuts

Two example cuts can be seen in Figure 4.2. The image on the left shows the state of the object before the first cut is made. The image on the right shows the state right before the second cut. The Figure 4.3 shows the state after both cuts.

One side effect of mesh cutting is that, with each cut, there are more and more objects that have to be rendered on screen. And since we did not implement any cleanup algorithms that might simplify the geometry after multiple cuts, object geometries get more and more complex. This is not an issue with simple meshes or small number of cuts, but is certainly something that might pose a problem for big complex meshes and a large number of cutting operations. The proposed solution would be to cull smaller fragments and do a periodical cleanup of fragment geometries.

4.5. Scene Rendering

When rendering the scene, we iterate over every object in it and tell it to render itself on the screen. Each object renders itself by rendering each of its fragments. Fragments have transformation information (position, orientation, scale) and the vertex indices. Objects that the fragments belong to have the vertex data (actual mesh) and they also have their assigned textures. The camera object contains all the necessary information about the view.

Pressing **F3** on the keyboard while the application is running toggles between wireframe and solid view modes, as shown in Figure 4.4.

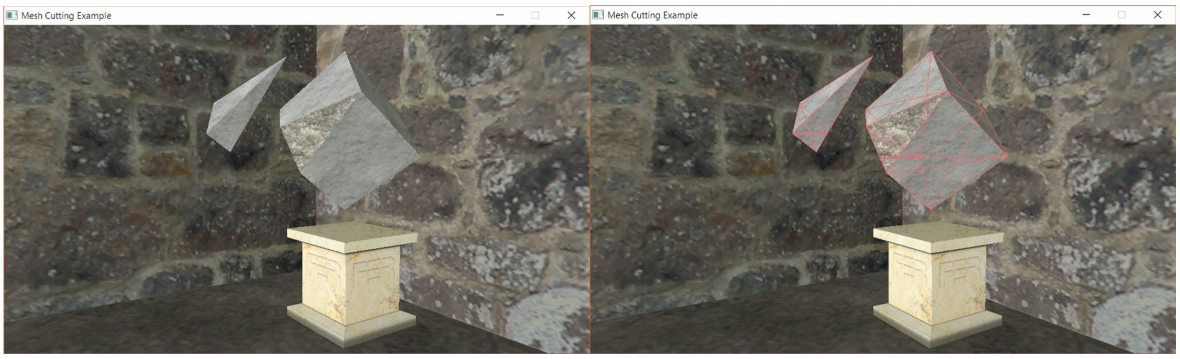

Figure 4.4: Two supported view modes: solid mode (left), wireframe mode (left).

5. Conclusion

Cutting a polygon mesh is a useful feature and is used for various applications, from special effects to scientific visualisation. Cutting all triangles in a mesh is relatively easy, as shown in this paper, but this by itself does not give us proper results for solid meshes. Therefore, other algorithms must be used to construct the interior of objects. Which versions of algorithms to use, whether to sacrifice mesh precision for faster cutting operation, and many other choices along the way are determined by the purpose of the application, but clever memory management and scene structure is always beneficial.

The methods described and implemented in this thesis work very well and cutting can be done in real time. While a few problems were noticed during the development, the current implementation has proved to be sufficient for simpler meshes. One of the problems is the fact that with each consecutive cut, the mesh becomes more and more complex. A more complex mesh results in slower execution speed, because it contains more triangles to process. Doing proper mesh cleanup after each cut would solve the rapid mesh complexity increase, but might also slow down cutting operations. Using more optimised triangulation methods would definitely help.

Except for the optimisation enhancements, another possible line of future work would be too investigate how a mesh might be cut by different cutting shapes. This thesis presents what could be a first step in developing a collection of tools to cut meshes, not just with planes but with cubes, spheres and other 3D shapes. Doing so would require the algorithms to generate nonplanar interior surfaces, which might prove challenging, but is definitely possible.

6. Bibliography

1. Blender, Bisect Tool Documentation
   https://docs.blender.org/manual/en/dev/modeling/meshes/editing/subdividing/bisect.html

2. David Eberly, Triangulation by Ear Clipping
   https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf

3. GLFW documentation
   http://www.glfw.org/documentation.html

4. Tom McReynolds, David Blythe
   Advanced Graphics Programming Using OpenGL, Elsevier, 2005

THREE-DIMENSIONAL MESH CUTTING IN VIRTUAL SCENE

ABSTRACT

This paper describes the process of cutting a three-dimensional polygon mesh that represents a solid object in a virtual scene. Mesh cutting is an operation that separates the mesh into two parts along a plane in 3D space. Triangulation techniques for filling the holes that appear during the cut are discussed and approaches to texturing newly revealed surfaces are presented. Scene composition and memory layout is covered and several compromises between cutting speed and visual precision and clarity are commented on.
This paper also contains the implementation details of an example application that was developed alongside. The application supports the cutting of multiple different models within a virtual scene and presents just one of many uses for the mesh cutting algorithm.

KEYWORDS:
computer graphics, polygon mesh, mesh cutting, triangulation, ear clipping

REZANJE TRODIMENZIONALNIH MODELA U VIRTUALNOJ SCENI

SAŽETAK

Ovaj rad opisuje postupak rezanja trodimenzionalne poligone mreže koja predstavlja popunjeni objekt u virtualnoj sceni. Rezanje modela je operacija koja dijeli model na dva dijela pomoću plohe u 3D prostoru. Opisane su tehnike triangulacije za popunjavanje rupa koje nastaju tijekom reza i predstavljeni su pristupi teksturiranja novonastalih površina. Rad sadrži opis strukture virtualne scene, raspodijele memorije i ukazuje na neke kompromise između brzine operacije rezanja i preciznog i čistog prikaza.
U ovom radu sadržani su i implementacijski detalji aplikacije koja je razvijena u sklopu rada. Aplikacija podržava rezanje više različitih modela u virtualnoj sceni i demonstrira jednu od mnogih primjena opisanih algoritama.

KLJUČNE RIJEČI:
računalna grafika, mreža poligona, rezanje modela, triangulacija, ear clipping