UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Diana Algma

Edge Chamfering Algorithm

Master's Thesis (30 ECTS)

Supervisor: Jaanus Jaggo, MSc

Tartu 2018

Edge Chamfering Algorithm

Abstract:

When comparing real-life objects and 3D models, one telltale sign that an object in question is a model, is the sharpness of its edges. This is more apparent for hard surface models like models depicting furniture and machinery, which look more real if their edges are a bit rounder. The goal of this thesis is to create an algorithm that uses the edge chamfering technique to automatically smooth out all hard edges of a mesh. The thesis describes 3D models and how they are stored in a game engine, different ways to soften hard edges, and the implementation of the created algorithm.

Key words:

Chamfering, beveling, models, edges, edge chamfering, Unity

CERCS: P170 Computer science, numerical analysis, systems, control

Servade tahumise algoritm

Lühikokkuvõte:

Võrreldes päris esemeid ja 3D mudeleid, tunneb 3D mudeli enamasti ära selle teravate servade tõttu. Eriti torkavad need silma kõvakatteliste mudelite puhul, nagu mööbli ja masinate mudelid, mis näevad kumeramate servadega välja reaalsemad. Selle lõputöö eesmärk on luua algoritm, mis kasutab servade tahumise tehnikat, et automaatselt mudeli teravaid servi kumerdada. Lõputöö kirjeldab 3D mudeleid ja kuidas neid mängumootoris salvestatakse, erinevaid viise servade tahumiseks ja loodud algoritmi implementatsiooni.

Võtmesõnad:

Tahumine, mudelid, servad, servade tahumine, Unity

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

3D objects often have hard edges that do not look natural. This can be changed by smoothing out the hard edges. To do that, modelers often use subsurface modifiers (Subdivision Surface Modifiers), which adds many faces and vertices to the mesh. This increases the vertex count drastically, making this method unsuitable for video games due to performance issues. An alternative way to render softer edges is to use normal maps, which circumvents the performance problem by using the low poly model together with the normal map for rendering. Creating normal maps is not only time-consuming in itself but using them requires more memory, and increases the time it takes to render faces.

The edge chamfering algorithm proposed in this thesis uses already existing vertices to create chamfers in place of hard edges. Hence, the vertex count will not go up a lot. This makes edge chamfering more suitable for real-time applications, such as video games with complex scenes, or mobile games. Games like Alien:Isolation [1] and Star Citizen [2] use this method to create more natural-looking models [3], [4]. This method increases the triangle count of the model but not as drastically as when using subsurface modifiers. Edge chamfering can turn low poly (low-polygon count) models into what are sometimes called medium poly (medium polygon count) models. Medium poly models are more complex than low poly models but less complex than the high poly models that can be created using subsurface modifiers. Low poly models were used in earlier 3D video games since the hardware could not render more complex models at that time. In recent years, the performance of graphics rendering hardware has increased a lot, allowing newer games to use medium poly models, and still run in real-time. However, high poly models created using subsurface modifiers are still mostly too complex for real-time rendering, and therefore usually used for precalculated applications like 3D movies.

Currently, edge chamfering is done manually by an artist, using some sort of modeling software. The only available automatic edge chamferer is a purchasable plugin called Quad Chamfer [5] for programs 3ds Max and 3ds Max Design. Having an automatic edge chamferer in a game engine like Unity would make edge chamfering much more accessible for developers, and speed up the process of game development.

The goal of this thesis is to create an algorithm for the Unity game engine that uses edge chamfering to automatically smooth out all hard edges of a given mesh. The result of the thesis is a Unity asset that can chamfer a given model using the edge chamfering algorithm. The algorithm identifies hard edges, pulls them apart, and creates faces between them. It also modifies the UV coordinates to correspond to the changes. The algorithm has to handle cases, where many hard edges are connected to a single point, or where UV seams and hard edges meet or overlap. In some cases, it creates some new vertices.

Chapter 2 describes the Unity editor. Chapter 3 describes where are 3D models used, how 3D meshes are stored in Unity and what are hard edges. Chapter 4 explains what is edge chamfering, and what are its advantages and disadvantages. Chapter 5 describes other methods that are used to smooth edges of models. Chapter 6 introduces the algorithm for automatic edge chamfering. Chapter 7 demonstrates how the created asset is used, and how the vertex and triangle counts change when chamfering a model. Chapter 8 describes some improvements that can be done on the created asset in the future.

2 Unity editor

The edge chamfering asset created in this thesis is a tool used in Unity Editor during game development. It speeds up the development process by allowing a quick way to chamfer existing models inside the Unity editor. For creating the asset and Unity illustrations, Unity version 2017.1.1f1 was used.

Unity is a multiplatform game engine developed by Unity Technologies [6]. It is used to develop 3D and 2D games for personal computers, consoles, and mobile devices. First announced to target OS X, it has since targeted 27 platforms.

Unity is a good entry-level game engine, and very capable for large-scale developments. Some notable games released on Unity are Ori and the Blind Forest [7], [8], Superhot [9], [10], and Cuphead [11], [12]. Also, in the University of Tartu, a few games have been made on Unity: Tribocalypse VR [13], and Reality B [14]. Unity is considered easy to learn and can get many people into the world of game making.

Unity engine has an editor for development, that simplifies and accelerates the process of game-making. The editor has many tools that provide the means to create and customize different aspects of the game that is being created, and the development process itself. Tools are more convenient to use if there is some sort of UI in the editor to use them. There are different means to do it: using the inspector, separate windows, or custom menu buttons.

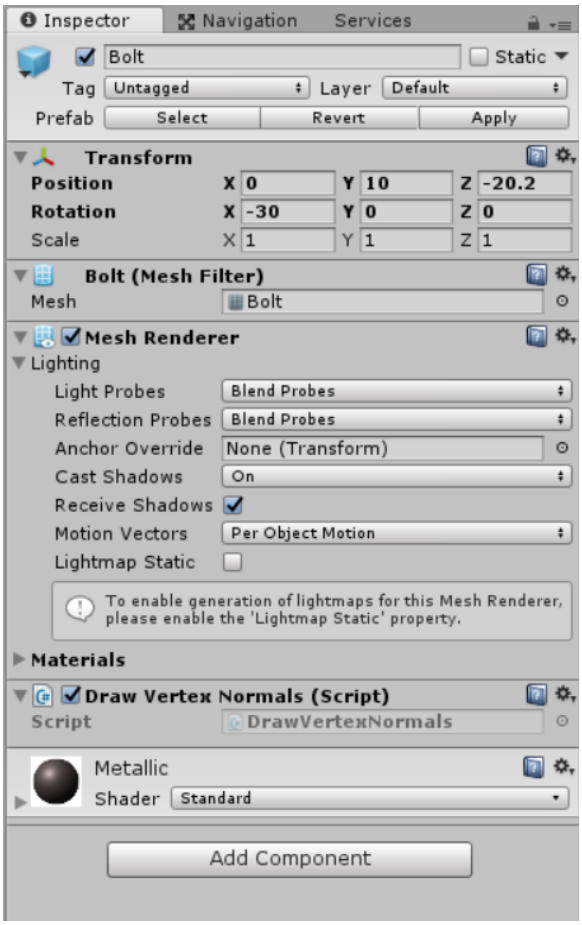

Figure 1. The Inspector view of a bolt model.

The inspector window displays different components of the selected object, and depending on the component it might enable changing different variables. Figure 1 shows the inspector view of a model. The components of the model are Transform, Mesh Filter, Mesh Renderer, a script called Draw Vertex Normals, and a material with the name Metallic.

Custom windows are useful when the info to be displayed is not attached to a specific component or object, or if it would take up more space than is available in the inspector view. The Unity asset UVee [15] uses a custom window to show and enable changing the UV map of an object (Figure 2).

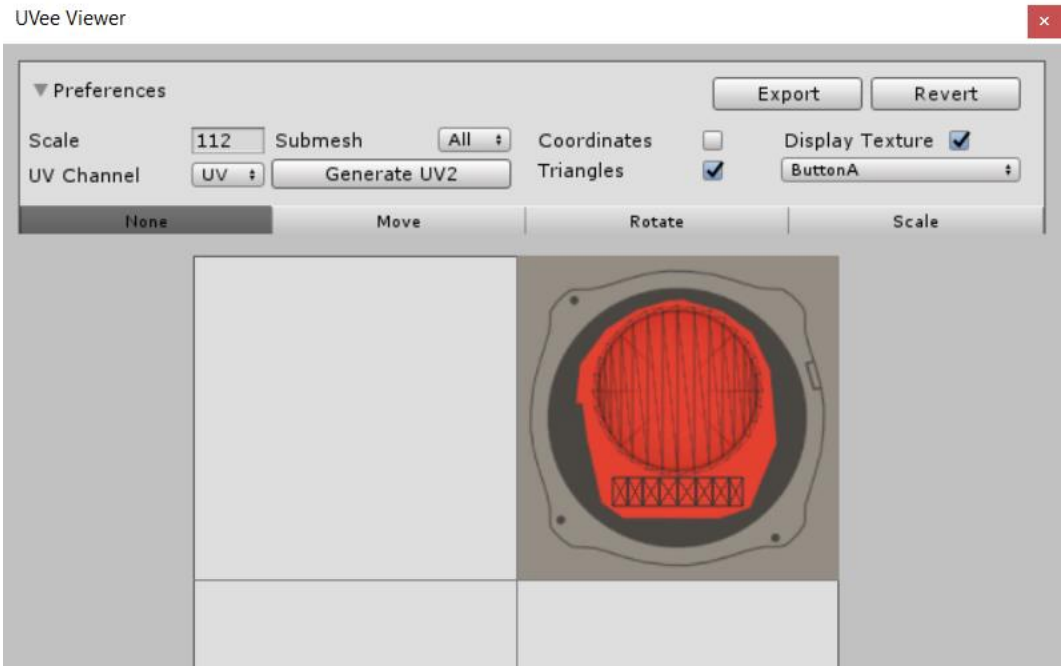

Figure 2. Custom window UVee Viewer.

It is possible to add custom options to different menus. As an illustration, to open the window of the Unity asset UVee [15], the user can choose the custom option “UVee Window” under “Window” in the menu bar (Figure 3). New options can be added to many different menus. For instance, menus that open when right-clicking on different objects.

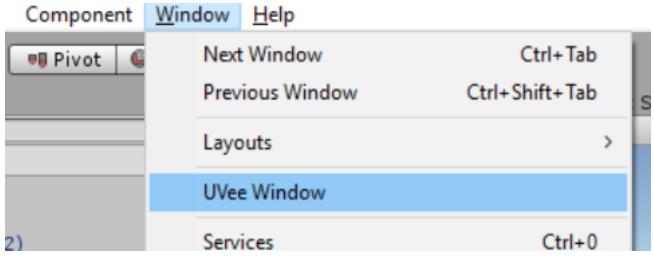

Figure 3. Custom option "UVee Window" under "Window" in the menu bar.

The edge chamfering asset created in this thesis is an extension for the Unity editor. A new menu option is added to start the process, but using the edge chamfering asset is mainly done through the inspector via a script component.

Inspector customizing is more effective if the script is attached to the object. It enables access to private variables, values can be changed and given without having to hard-code them into the scripts. It gives a good control over fine tuning on how the scripts should perform in the games since the modification can be done during run-time. For the changes to take effect during runtime, special care is needed while writing the script to make sure that the new values are used.

Unity has supported scripting in three languages: C#, Boo, and a Unity version of Javascript called UnityScript. In 2014, Unity announced that they are dropping Boo support starting from Unity version 5.0 [16]. In 2017, it was announced that UnityScript support will also be gradually dropped, with version 2017.2 beta not having the option to create ’JavaScript’ files [17]. This leaves C# to be the only supported scripting language for Unity. Scripts written in this thesis are in C#.

3 3D models

3D models are usually constructed as 3D polygon meshes but they can also be constructed using other data, for instance, NURBS [18], [19]. When exported, all models, either based on polygon meshes or other data, are usually converted into triangle meshes. This chapter describes 3D polygon and triangle meshes, how they are stored in a game engine, what are hard edges, and what are 3D models used for.

3.1 3D polygon mesh

3D polygon mesh consists of vertices, edges, and faces defining a polyhedral object [20]. The polygons (faces) in a mesh are typically simple, mostly triangles and quadrilaterals, to simplify rendering. Figure 4 shows an example of a triangle mesh. A triangle mesh is a polygon mesh where all the polygons are triangles. When creating models using modeling software like Blender, modelers can create faces with more than three or four edges. When a model is imported into Unity, all its polygons are triangulated which is a process that divides all other polygons into triangles [21]. Meshes in Unity are therefore stored as triangle meshes [22].

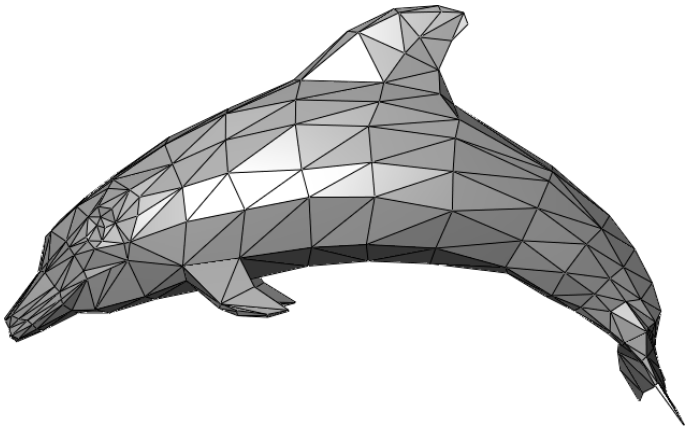

Figure 4. Triangle mesh of a dolphin [23].

This thesis is about modifying meshes in Unity. Therefore, the geometry analyzed in this thesis consists of only triangles. In the Mesh class in Unity, vertices are stored in an array, and triangles are defined by groups of indices of its vertices in the vertex array. The triangles are stored in an integer array so every triangle consists of three elements in the array. The first triangle is the first three elements, the second triangle is the 4th, 5th, and 6th element and so on. The mesh in Figure 2 could be stored as vertices = [(0,0,0), (1,0,0), (0,1,0), (1,0,0)], and triangles = [0,2,1,1,2,3]. Unity uses clockwise winding order, which means that triangles formed with vertices in clockwise order, define a front-facing triangle. Only the front faces of triangles are rendered onto the screen. Winding order is depicted with a black arrow in Figure 5.

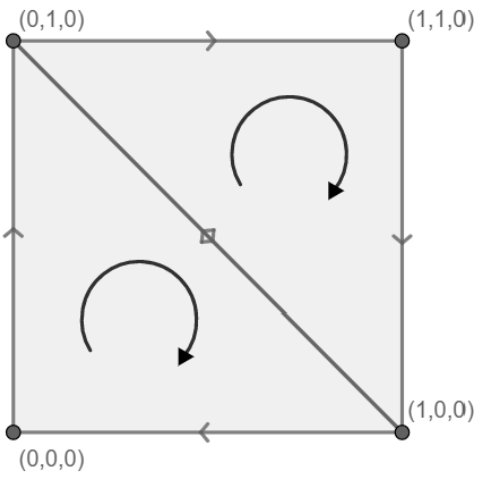

Figure 5. Mesh of two triangles. Winding order is shown with black arrows.

3.2 Information stored in vertices

A vertex can hold different information, in addition to its position, it can also store color, normal vector, tangent vector, and texture coordinates. This information is saved in additional arrays where the *n*th element corresponds to the *n*th vertex in the vertices array. The first element in the normals list is the normal vector of the first vertex in the vertices array. Table 1 demonstrates the information of five vertices of a cube. The cube does not have any vertex colors and has UV coordinates for only the first channel.

Index	Vertices	Normals	UV (first channel)
0	(-1.0, 1.0, -1.0)	(0.0, 0.0, -1.0)	(0.5, 0.3)
1	(1.0, 1.0, -1.0)	(0.0, 0.0, -1.0)	(0.3, 0.3)
2	(1.0, -1.0, -1.0)	(0.0, 0.0, -1.0)	(0.3, 0.5)
3	(-1.0, -1.0, -1.0)	(0.0, 0.0, -1.0)	(0.5, 0.5)
4	(-1.0, 1.0, 1.0)	(0.0, 0.0, 1.0)	(0.5, 1.0)
…	…	…	…

Table 1. Information of the first five vertices of a textured and seamed cube as it is stored in the lists.

3.2.1 Vertex position

Vertex position determines where the vertex is located in the mesh. For a 3D mesh, the vertex position is a 3D vector (Vector3 [24] in Unity). The position vector is the vector from the mesh origin point to the vertex position.

3.2.2 Vertex color

Vertex color which consists of an RGB color and/or an alpha value is typically used in the shader to modify the output color of the mesh. It can also be used to control the transparency or apply multiple textures. Figure 6 shows a mesh of a sphere that is colored using vertex painting. The shader colors the mesh using the vertex colors saved.

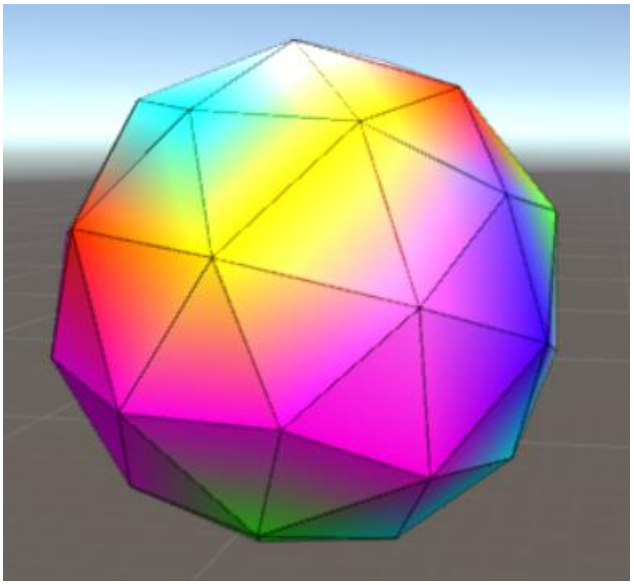

Figure 6. Sphere colored using vertex painting.

3.2.3 Vertex normal

Every vertex has a normal. Normals are used mainly for shading. The shading of a face (a triangle) is determined by the normals of its vertices. Surface normals are linearly interpolated between the vertex normals in the graphics card.

If all the vertices of a face have same directional normals, the surface of the face will appear flat. If the vertex normals are pointing in different directions, the surface will appear to be curved. Figure 7 illustrates how the grey surface normals are interpolated using the blue vertex normals. The left image in Figure 7 illustrates a flat surface with codirectional vertex normals. The right image shows a surface with different vertex normals. The green dotted line shows how the surface appears when shaded.

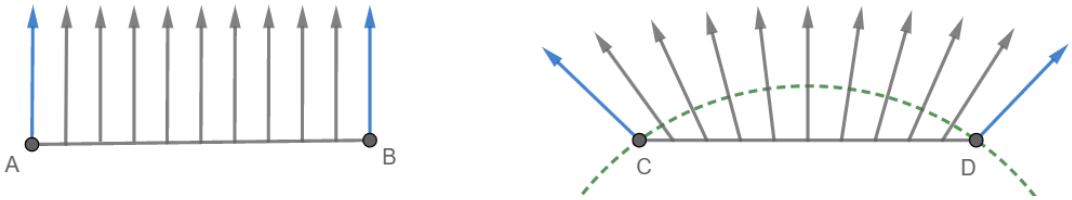

Figure 7. Linearly interpolated surface normals for flat (left) and curved surface (right).

In addition to vertex normals, the normals info can also be stored in world space normal maps. With normal maps, the graphics card can render details like wrinkles or defects on surfaces without the need to add these details to the mesh geometry. There are two kinds of normal maps: world space and tangent space. World space normal maps are not as popular as tangent space normal maps, mainly because tangent space normal maps are not dependent on the orientation of the surface it is applied on [25]. World space normal maps work only if the object does not rotate or deform. Tangent space normal maps are also referred to as tangent normal maps, or simply normal maps. World space and tangent normal maps can be easily distinguished by the color range represented on the map. Tangent normal maps usually have mostly blueish colors while world space normal maps also feature red, green, and yellow tones.

3.2.4 Vertex tangent

Tangent in Unity is a four-dimensional vector. The first three parameters define the vector, and the fourth parameter is 1 or -1. Tangent vectors are perpendicular to the normal vectors. On a flat surface, the tangent vector exists on the surface plane. On a curved surface, it lies tangent to a reference point (the vertex normal).

Tangent vectors are used for applying normal maps. Unlike world space normal maps, tangent normal maps use the surface as the reference instead of the world space [26]. This makes tangent normal maps reusable on models with different shape or geometry. Figure 8 shows a world space normal map (left), and a tangent space normal map (right) of a boot.

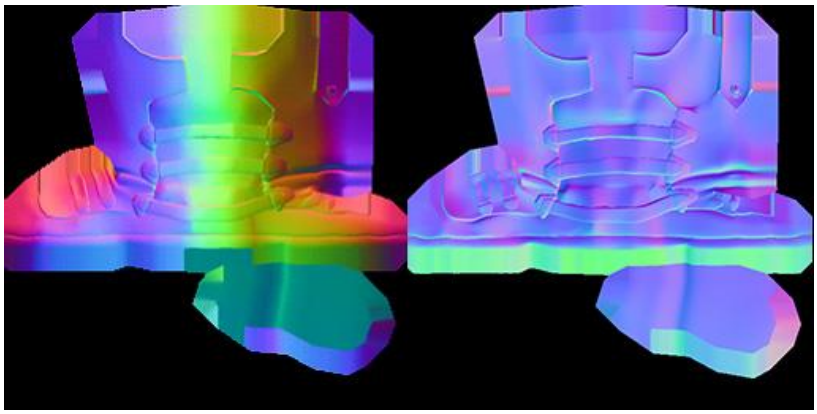

Figure 8. World space normal map (left), and tangent space normal map (right) of a boot [27].

3.2.5 Texture coordinates

Texture coordinates are pairs of numbers usually used to apply a 2D texture on a 3D mesh. Texture coordinates, also called UV coordinates, or UVs, range from 0 to 1 and refer to a point on the 2D texture that the vertex should be "pinned" on. The texture is then stretched and fitted on the mesh triangle. Figure 9 displays a model of a button without texture with and without wireframe (drawn edges). Figure 11 shows the model with texture from Figure 10 applied. The author of the button model and texture is to Jaanus Jaggo.

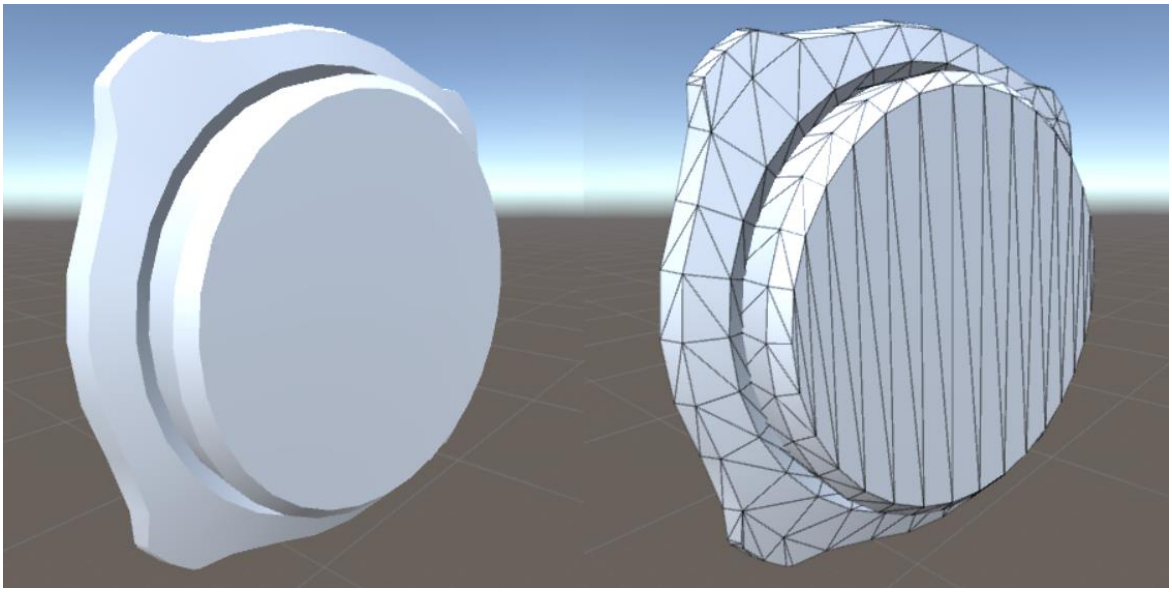

Figure 9. Button model without wireframe (left), and with wireframe (right).

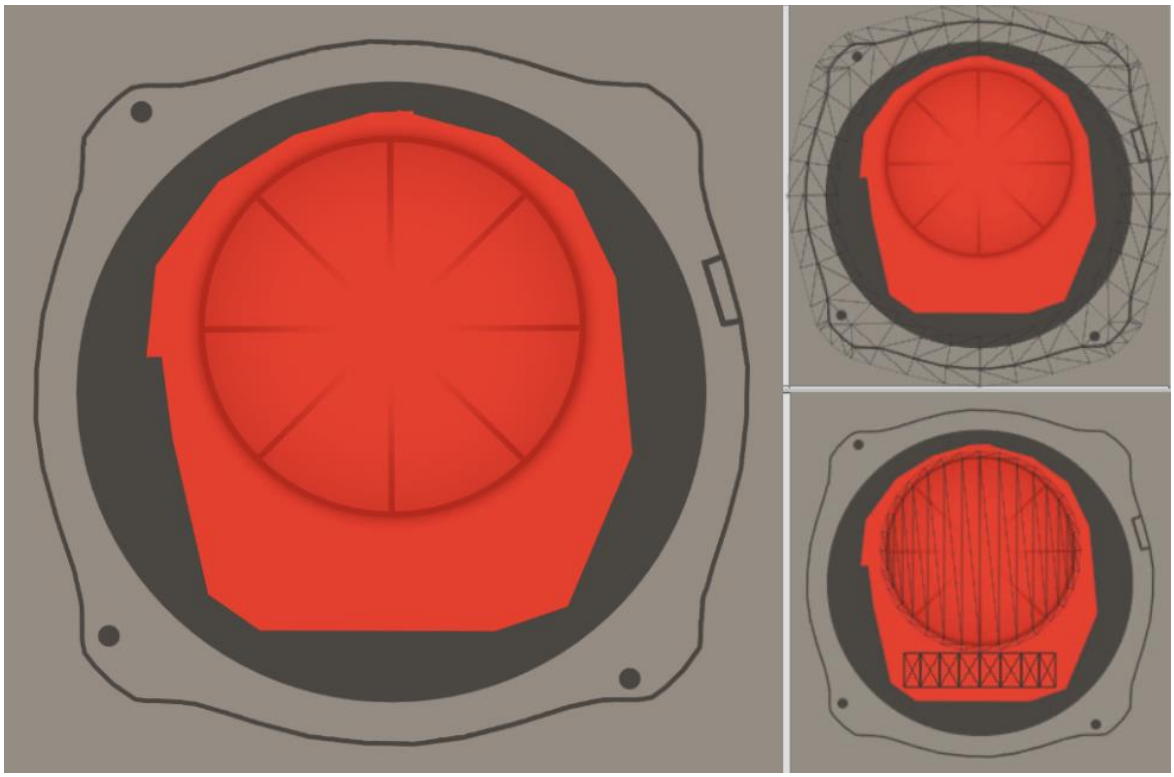

Figure 10. UV map of the button model from Figure 9. The images on the right show the two sub-meshes mapped on the UV map.

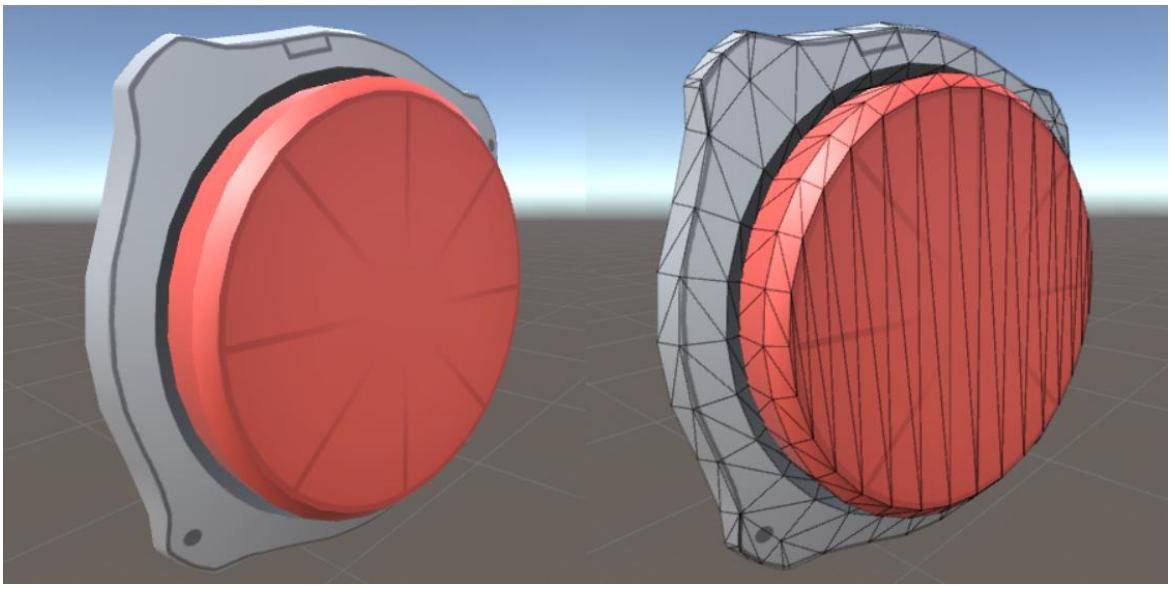

Figure 11. Button model without wireframe (left), and with wireframe (right) from Figure 9 with texture from Figure 10.

Like with normals, a vertex can hold only one pair of texture coordinates (in one set), meaning that if a vertex has a place on different textures or different positions on one texture, there need to be multiple vertices in place of one. The edges that are positioned in two different places on the UV map are called seams. These edges have doubled vertices with different UV coordinates.

In Unity, a mesh can have four texture coordinate sets, or channels, which all have their own uses [28]. The first one is used for diffuse, metal/spec etc. The second one is used for baked lightmaps. The third channel is used for real-time GI (Global Illumination) data. The fourth channel can be used for whatever the user wants.

3.3 Hard edges

There are two ways to shade a mesh (or one face of a mesh): flat shading and smooth shading. In the case of flat shading, the faces appear flat and have codirectional surface normals. The edges of a flat-shaded mesh look sharp. Vertices are doubled where faces on different planes are connected. These edges that have at least one vertex doubled with different normals are called hard edges. The left image in Figure 12 illustrates a flat-shaded mesh. The grey lines are the faces, and blue vectors are the vertex normals.

In the case of smooth shading, vertices are not doubled, and vertex normals are an average of the normals of the faces. These normals are called shared vertex normals. This way, the faces will look curved, and the edges will look smooth and not pronounced. The right image in Figure 12 illustrates a smooth-shaded mesh with shared vertex normals shown as blue arrows. The surface of the mesh will appear to look curved and is represented by the dotted green line.

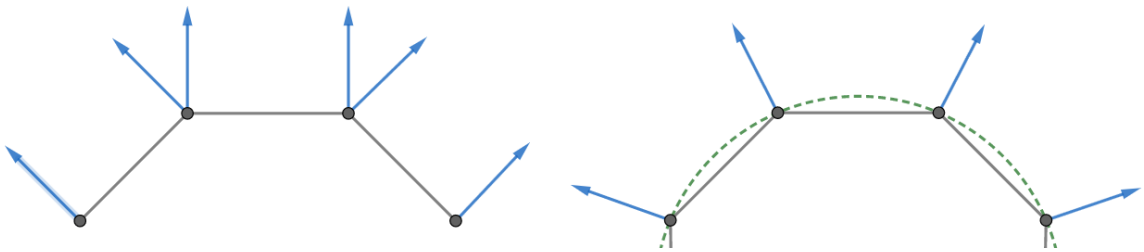

Figure 12. Vertex normals on a flat-shaded mesh (left), and vertex normals on a smooth-shaded mesh (right).

Figure 13 compares two meshes of a not very rounded sphere with edges and vertex normals drawn. The left sphere is flat-shaded and has multiple vertices at one position with different normals. All the edges of the left sphere are hard edges. The right sphere is smooth-shaded and has shared vertex normals. Figure 14 shows the same two meshes but without normals or edges drawn. The edges of the left sphere are very sharp and visible. The edges of the right sphere are soft and not distinguishable. The outline remains not smooth for both spheres because shading does not change the geometry of the mesh, and the illusion of a curved surface comes from interpolated surface normals.

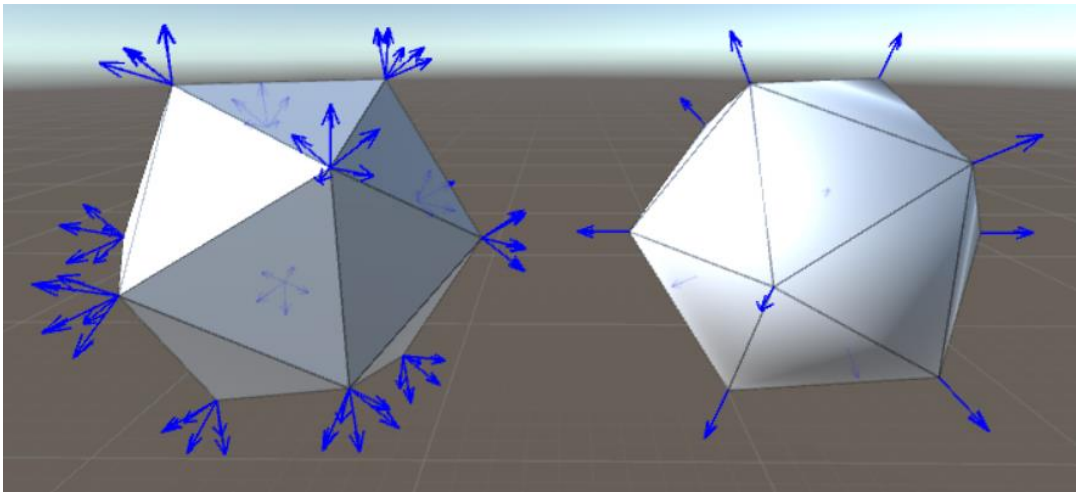

Figure 13. Sphere with flat shading and hard edges (left), and sphere with soft edges and shared vertices (right).

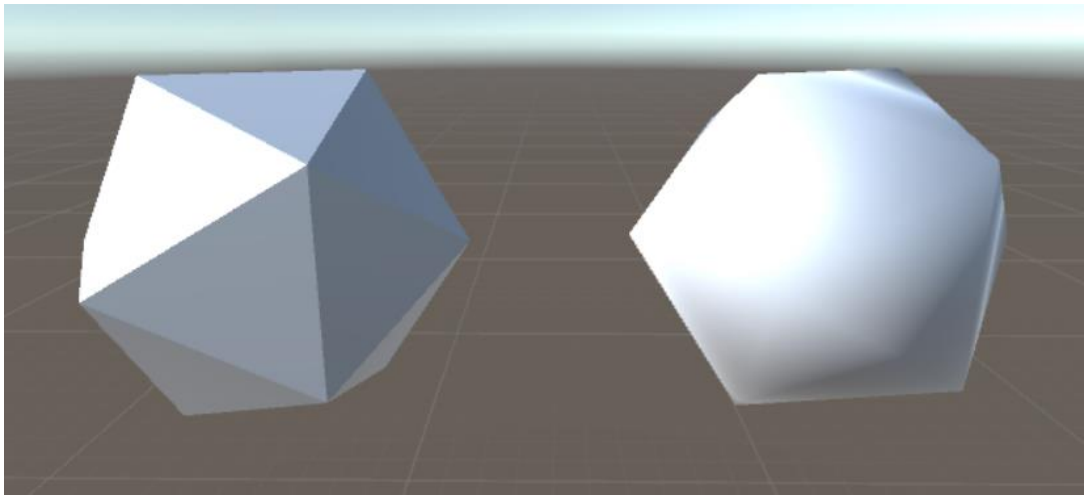

Figure 14. Spheres from Figure 13 with only shading, no edges or normals shown.

A mesh can have both soft and hard edges. For example, a mesh of a cylinder should look round for the most part, but the edges of the top and bottom faces should look sharp. Figure 15 shows a cylinder mesh with both soft and hard edges.

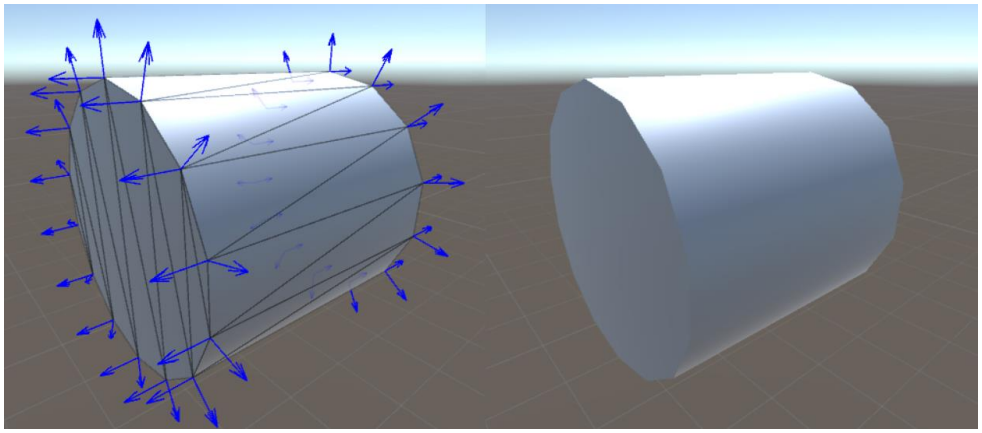

Figure 15. Mesh of a cylinder that has hard edges only on the top and bottom faces; mesh with normals and edges shown (left), and mesh with only shading (right).

3.4 Uses of 3D models

3D models are used in many different fields. For instance, 3D models and other CGI are used in countless films such as Jurassic World (2015) [29], and the Transformers film series [30]. 3D models are also used in the production of animations, such as Toy Story (1995) [31], and games like Star Citizen [2], [4]. 3D models are not only used for entertainment purposes but also in education, architecture, and product manufacturing and design. In education, one use for models is studying anatomy. Figure 16 displays a screenshot of one such software – Anatomy & Psychology by Visible Body [32].

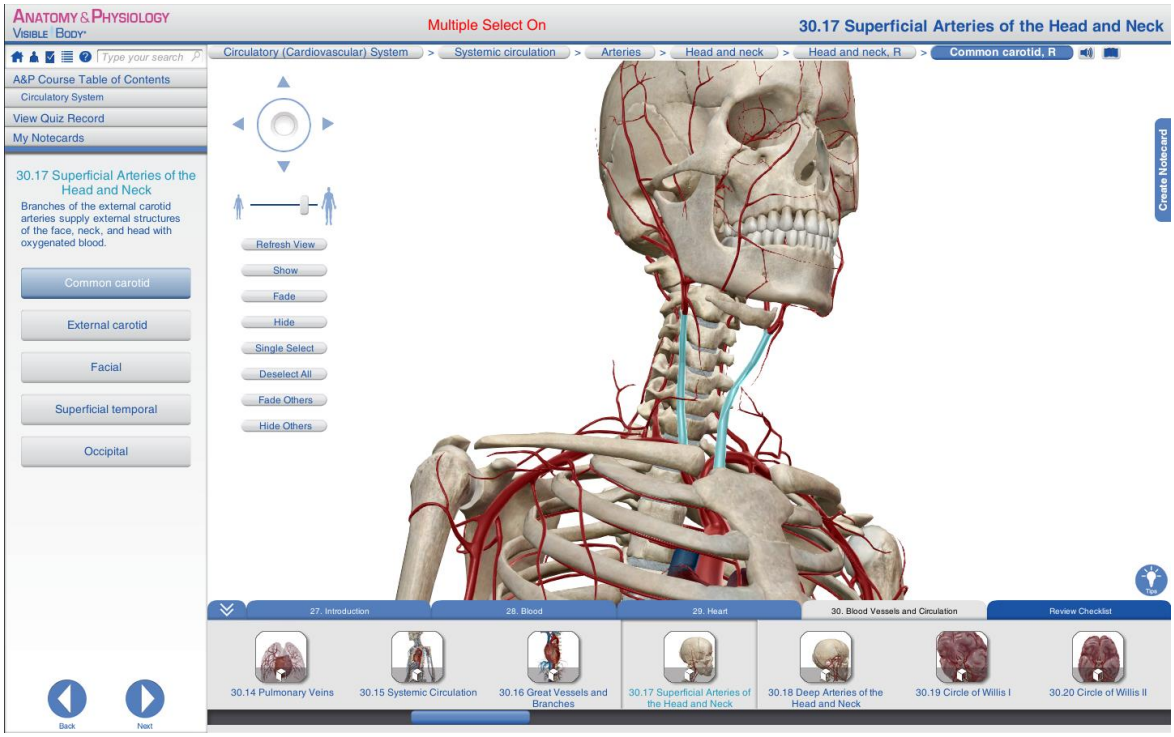

Figure 16. Screenshot of Visible Body Anatomy & Psychology software [33].

The complexity of the models varies by field. In films and animations, the rendering of the models is not done in real-time, and the mesh can be more complex in order to show more detail and realism. Real-time is often considered to be seemingly instantaneous, or almost instantaneous [34]. In games, the rendering is done in real-time, and performance is a priority which limits the complexity of the models making realism harder to achieve. For fields like education, architecture, and design, the models have to most likely be complex in order to be useful since a lot of detail has to be visible. For example, if an educational model of a skeleton does not feature enough detail for different bones to be distinguishable and recognizable, the model will not be usable for learning about different bones in the skeleton. Performance is secondary in that case.

4 Edge chamfering

In this thesis, an algorithm for Unity game engine is created that uses edge chamfering [35], [36] to automatically smooth out all hard edges of a given mesh. Chamfering a hard edge makes the edge look rounder and smoother. Most of the time, the terms “chamfer” and “bevel” are used interchangeably. In technical usage, they can sometimes be differentiated. In this thesis, “bevel” is used to refer to a flat-shaded face in place of a hard edge. “Chamfer” is used to indicate a smooth bevel with face-weighted normals.

Edge chamfering modifies a hard edge so it looks rounder. A new face is created between the faces in place of the hard edge. The vertices used to create the new face (the chamfer), are the already existing vertices. Normals of the vertices are not changed, thus retaining face-weighted normals. This leaves the illusion, that the faces are flat but the chamfer is rounded. Figure 17 illustrates the normals on a hard edge before and after chamfering. The green dotted line on the right image demonstrates how the chamfer is shaded, giving an illusion of a round edge. The solid green line depicts the actual chamfer.

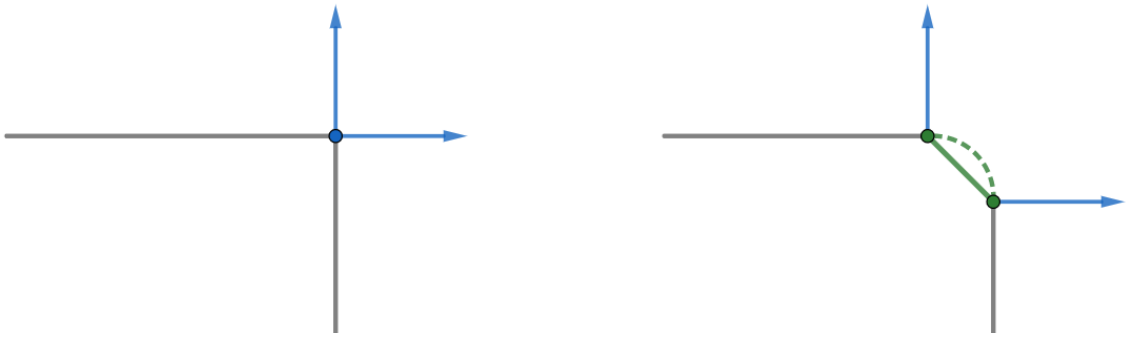

Figure 17. Normals on a hard edge before (left) and after chamfering (right).

In Figure 18, there is a cube with normals shown as blue arrows. The chamfered version of the cube is shown in Figure 19; the right side has the chamfer highlighted in green.

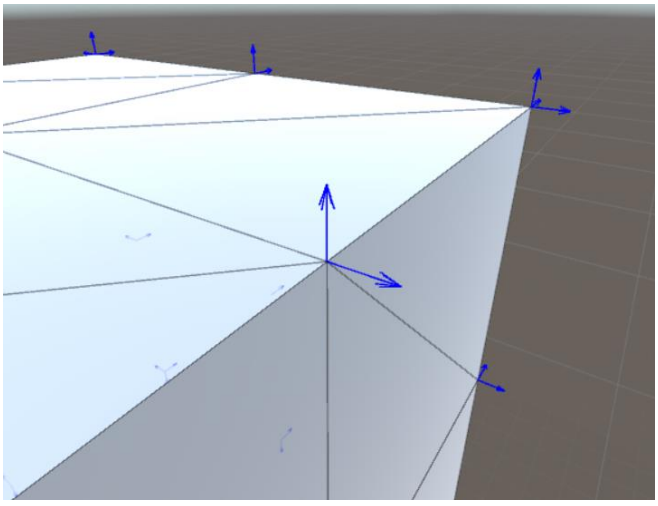

Figure 18. Cube with hard edges. Normals are shown with blue arrows.

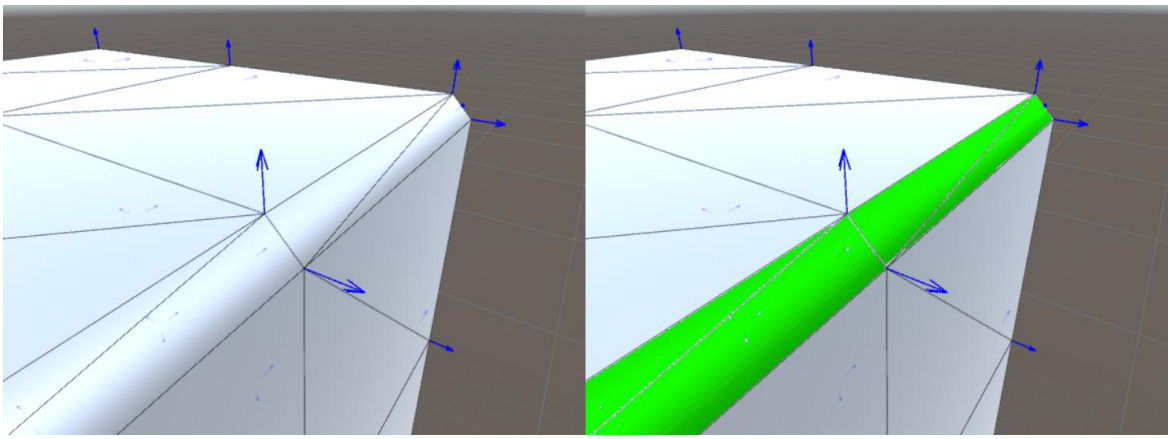

Figure 19. Cube from Figure 18 after chamfering. The chamfer is highlighted in green on the right side.

Chamfering makes the edges of a model appear softer and more natural. Figure 20 shows two models of bolts with hard edges. Figure 21 shows the same models after hard edges are chamfered.

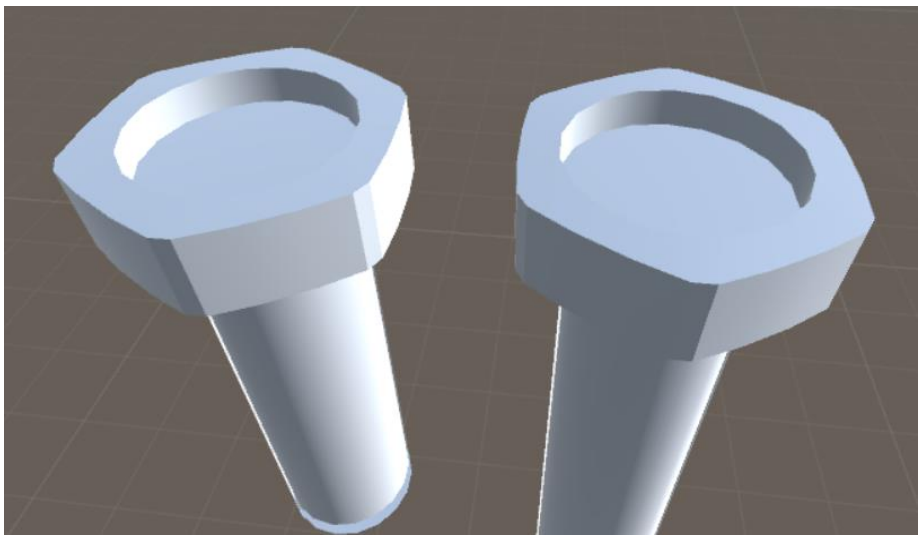

Figure 20. Two models of bolts before chamfering (models created by Jaanus Jaggo).

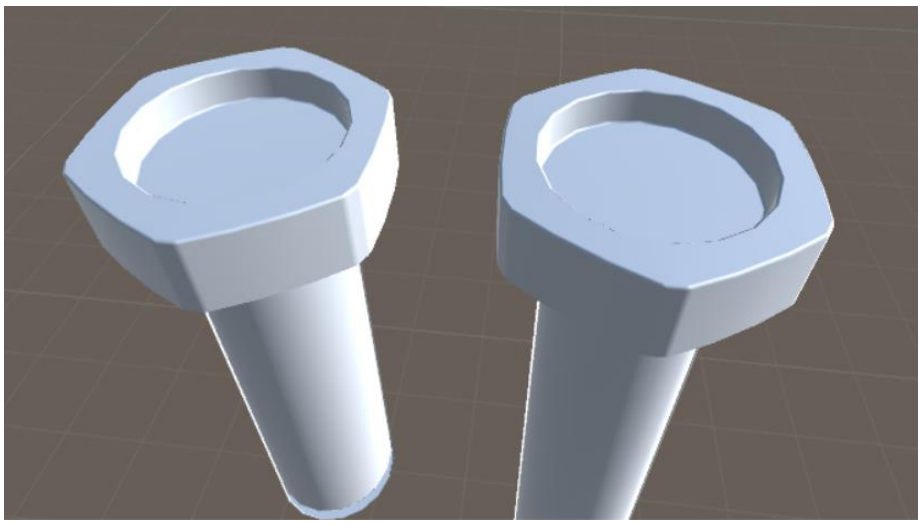

Figure 21. Bolt models from Figure 20 after chamfering.

4.1 Uses of edge chamfering

There are two kinds of models – hard surface models and organic models. The exact definition of both is debatable [37], so in the context of this thesis, a hard surface model is defined as something manmade. Some examples are models depicting buildings, machines, furniture, or smaller items like boxes, pencils, or bolts. The asset created in this thesis can be applied to any kind of model, but the effect will be better on hard surface models.

Figure 22. A corner of a conference table [38].

Man-made objects like furniture and machinery often have chamfered or round edges to make the objects safer to use, or due to the limitations of the manufacturing process. Figure 22 shows an example of chamfered edges of a table. Edges of 3D models usually have hard edges that are sharp and unnatural since no edge is completely sharp in the real world. A small chamfer can make the model look a lot more real. In the game Star Citizen, edge chamfering is used in creating large-scale models, for example, the bomber starship Gladiator (Figure 23) [4].

Figure 23. Exterior shot from below of the Gladiator used in Star Citizen [39].

Edge chamfering is an easy way to improve existing models. For instance, provided a finished scene of a game with low poly models, the user can use edge chamfering to improve the visuals of all or some of the models. With an automated edge chamfering tool, this does not require much effort or time.

4.2 Advantages and disadvantages

The method of edge chamfering creates smooth-shaded chamfers where hard edges used to be and leaves the faces with the original shading. This leaves a visually pleasing and realistic model with vertices on chamfers having face-weighted normals.

Chamfering a model might cause specular aliasing when rendered. If the material is very reflective, or the object is looked at from afar, the specular highlight on the chamfered object looks jagged (Figure 24). Aliasing can be minimized by using a wider chamfer on more reflective or glossy materials like shiny metals [40]. It can be also be solved by using the original model when the object is far from the viewer or using some other antialiasing methods like temporal AA [41], or Valve Geometric Specular Aliasing [42]. Aliasing can also be a problem with other edge smoothing techniques.

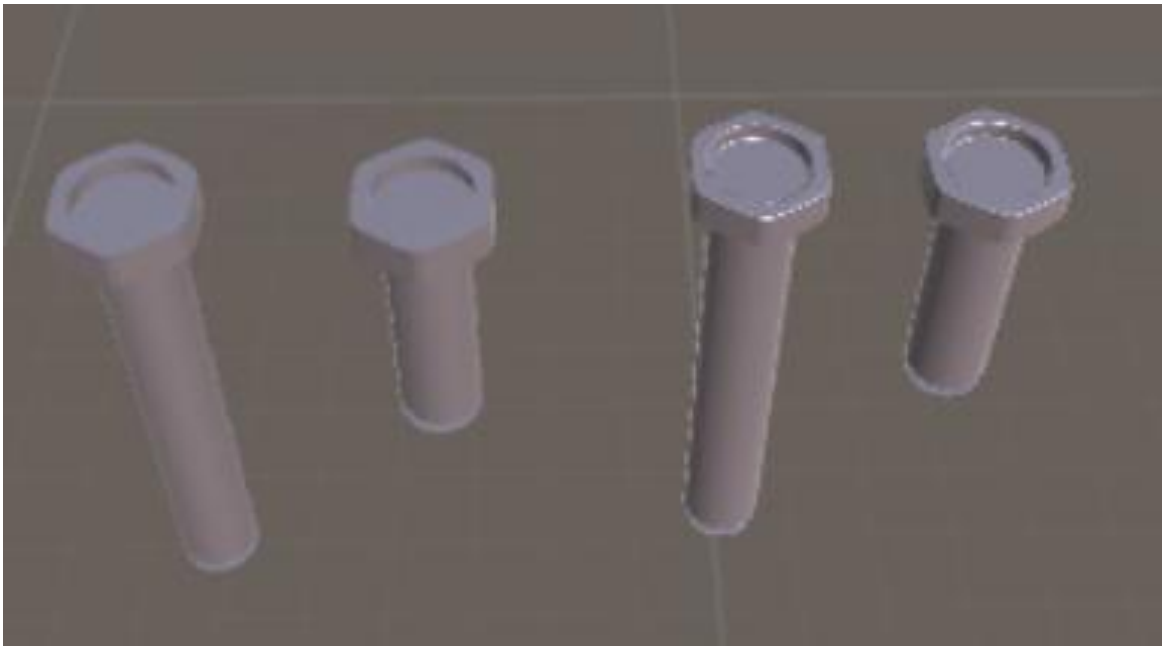

Figure 24. Bolt models from Figure 20 with a metallic material when looked at from afar (zoomed in for clarity). Two bolts on the left are not chamfered, the two on the right are chamfered. The white bits on the right bolts are the aliased highlights.

Edge chamfering is suitable for real-time rendering because of the minimal amount of vertices created. Chamfers will be created between the existing vertices without creating new vertices in most cases.

New vertices are created in following occasions:

- separating corner-connected vertices (see chapter 6.2),
- recreating seams (see chapter 6.4),
- filling holes where five or more hard edges met (see chapter 6.6).

Since the number of vertices created by edge chamfering is minimal, it does not make the model too complex for it to affect performance or memory usage. If the model has not been unwrapped to create a UV map, then there are no seams to be recreated, and even less new vertices are added.

For the graphics card, higher vertex count means more work and time to render the scene [43], [44], [45], [46]. If the vertex count is too high, then it will visibly hinder performance when rendering is done in real-time. How many vertices is too many, is very debatable, and

depends on many variables like the hardware, how much of the scene is visible, the size of the model, and much more [43], [45]. Adding chamfers to a mesh creates many new triangles, but triangle count is not as impactful on performance or memory as vertex count is [44], [46], [47], [48], [49], [50], [51], [52]. When a normal map is added for a mesh, the fragment shader has to use it for calculations, increasing the workload on the fragment shader [43], [44]. Since no normal maps are created during chamfering, the cost of rendering individual triangles will not increase. Measurements of how many vertices and triangles are created are detailed in chapter 7.2.

Creating a Unity asset using this method makes it easier to use without having to have knowledge about modeling. Different modeling software can be used to chamfer edges, but they are difficult to use for a beginner and the process of edge chamfering consists of difficult steps. The Unity asset makes edge chamfering more accessible for programmers and beginner modelers.

5 Other methods and their limitations

Edge chamfering is not the only method used to make hard edges of a model smoother or to give a model visually more complexity without creating a geometry too complex. This chapter explores some of the common methods and their limitations. The tools and methods described in this chapter are tested in the modeling software Blender v2.78 and might differ in older and newer Blender versions.

5.1 Subdivision surface modifiers

One technique for smoothing the edges of a mesh is using subdivision surface modifiers [53]. Blender offers the use of two subdivision algorithms – Catmull-Clark [54] (Figure 25) and Simple [53] (Figure 27).

5.1.1 Catmull-Clark subdivision

The Catmull-Clark algorithm was created by Edwin Catmull and Jim Clark in 1978[1]. It defines surfaces recursively using these steps [54]:

1. Add a *new face point* for every face. A *new face point* is the average of every point of the face.
2. Add a *new edge point* for every edge. A *new edge point* is the average of the two endpoints of the edge and two new face points of the faces connected to the edge.
3. Add an edge from every new face point to neighboring new edge points.
4. For every original point P, take F as the average of the *n* new face points of the faces connected to P, take R as the average of all *n* edge midpoints for edges connected to P, where edge midpoints are the average of their two endpoint vertices, and not *new edge points*. Create a new vertex point for every vertex P at $\frac{F+2R+(n-3)P}{n}$.
5. Connect new vertex points to new edge points of the edges that were connected to P.
6. Define new faces using all created edges.

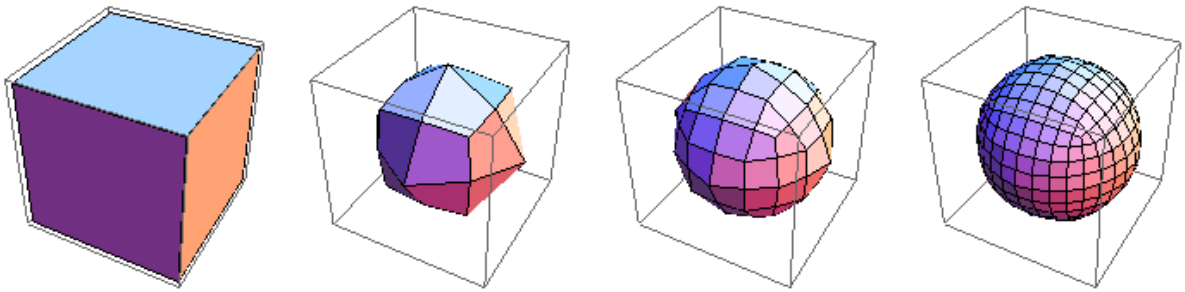

Figure 25. Subdivision on a cube using Catmull-Clark subdivision [55].

Catmull-Clark subdivision is used to create smooth surfaces. Figure 26 shows a beveled cube with different number of iterations (subdivisions) of Catmull-Clark subdivision applied.

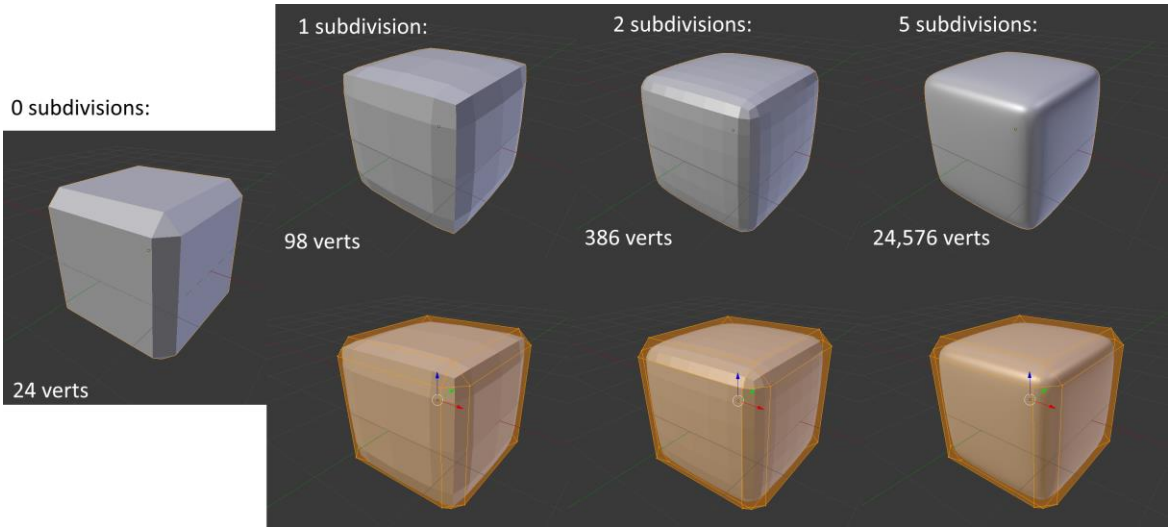

Figure 26. A beveled cube in Blender subdivided with Catmull-Clark subdivision algorithm after 0, 1, 2, and 5 subdivisions from left to right. The vertex counts displayed are of the smooth-shaded model.

This method might not be suited for creating smooth edges if the rest of the mesh should remain the same since subdivision changes all the vertices in the mesh. This might make the mesh smaller (like with the cube in Figure 25), or change the overall shape of the model. If subdivision is used together with the Blender Bevel tool (see chapter 5.2), the result will be better (Figure 26), but many bevel segments and subdivisions are needed to get a model that looks chamfered.

5.1.2 Simple subdivision

Simple subdivision algorithm [53] only subdivides the surface but does not do any smoothing. After using simple subdivision, the mesh can be modified using modeling software to look more detailed which can include smoother edges. It is also used to apply displacement maps in more detail. Displacement maps change the vertex positions of the existing geometry, and it will have a more detailed effect if the mesh has more vertices.

Figure 27 demonstrates the beveled cube from Figure 26 after applying simple subdivision twice. Figure 27 a) and b) show the cube with no other modifications after the subdivision, Figure 27 c) shows it after the vertices created by subdivision have been manually moved to make the cube rounder.

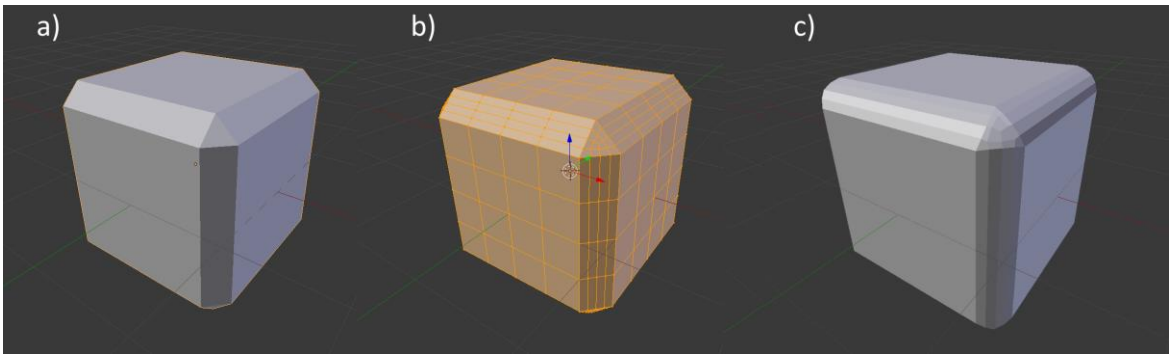

Figure 27. A beveled cube in Blender subdivided with Simple subdivision algorithm 2 times just after subdivision (a and b), and after manually moving some vertices created by the subdivision (c).

5.1.3 Subdivision effect on performance

Using subdivision surface modifiers increases the vertex count (Figure 26), and heavy use of it will decrease performance making it unsuitable for real-time rendering in computer games with complex scenes. The method is suitable for and used in graphical art like animation films. For example, subdivision surfaces were used for simulating clothing in the short film Geri's game[2].

5.2 Bevel tool in Blender

In the modeling stage after the model is created, tools like Blender's Bevel tool [56] allow beveling edges and corners. Additional steps have to be taken to make the bevels smooth i.e. to create a chamfer.

Selecting edges with two adjacent faces, or selecting vertices in 'vertex only' mode, and using the Bevel tool will smooth out the edges and/or corners by creating new faces. Figure 28 depicts the result of beveling one edge in Blender using the Bevel tool.

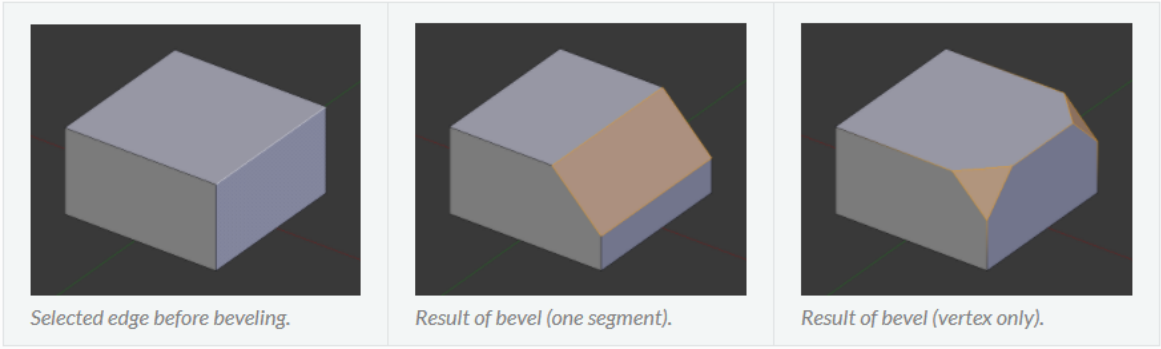

Figure 28. Beveling using Bender's Bevel tool [57].

When using the Bevel tool, the created face (the bevel) will be flat-shaded. The hard edge is not really smoothed out, but instead, it is replaced with two hard edges. For the bevel to be smooth-shaded so it would look like the edge is rounder, custom vertex normals have to be used. The shading looks best with face-weighted normals, which means that the vertex normals are the normals of the original faces, leaving a chamfer between these faces smooth-shaded. Figure 29 compares three cubes with beveled edges and different shadings.

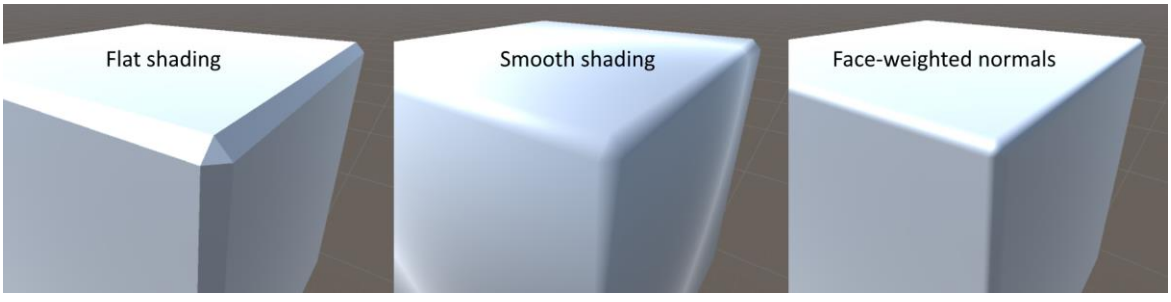

Figure 29. A cube with beveled edges with flat shading (left), smooth shading (middle), and face-weighted normals (right).

5.2.1 Changing vertex normals of a face created with the Bevel tool

For the normals of the beveled models to be face-weighted, the normals would have to be manually set so the right faces would be flat-shaded. One way to do this is to transfer the normals from the original model. This is done with the following steps:

1. Have low poly object open in Blender
2. Duplicate it (Shift+D in Object Mode)
3. Add a Bevel modifier to the new object (Figure 30)
4. Set ‘Width Method’ to ‘Width’
5. Set the position as the same as the low poly object
6. Add a new modifier Data Transfer
7. Check ‘Face Corner Data’
8. Choose ‘Projected Face Interpolated’
9. Choose ‘Custom Normals’
10. Set ‘Source Object’ as the low poly object
11. Hide low poly object
12. Adjust Width and Segments until satisfied
13. Apply both modifiers (Bevel, then Data Transfer)
14. Select ‘Auto Smooth’ (Figure 31)
15. (Optional) Delete the low poly object

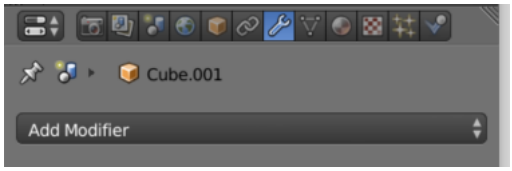

Figure 30. Adding modifiers in Blender.

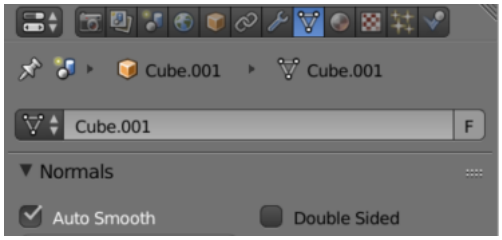

Figure 31. Auto Smooth option in Blender.

The result will be a chamfered model.

5.3 High and low poly baking

Another method is to create both a high and a low poly model and bake the details from the high poly model to the normal map of the low poly model. This chapter describes the work-flow of creating normal maps based on the tutorial Sculpt, Model and Texture a Low-Poly Skull in Blender [58], and gives an example of normal maps used for creating chamfers.

Figure 32 illustrates how using a low poly model, a texture, and a normal map can result in a visually complex model. The brownish map in the middle is the UV map, the blueish map on the right is the normal map.

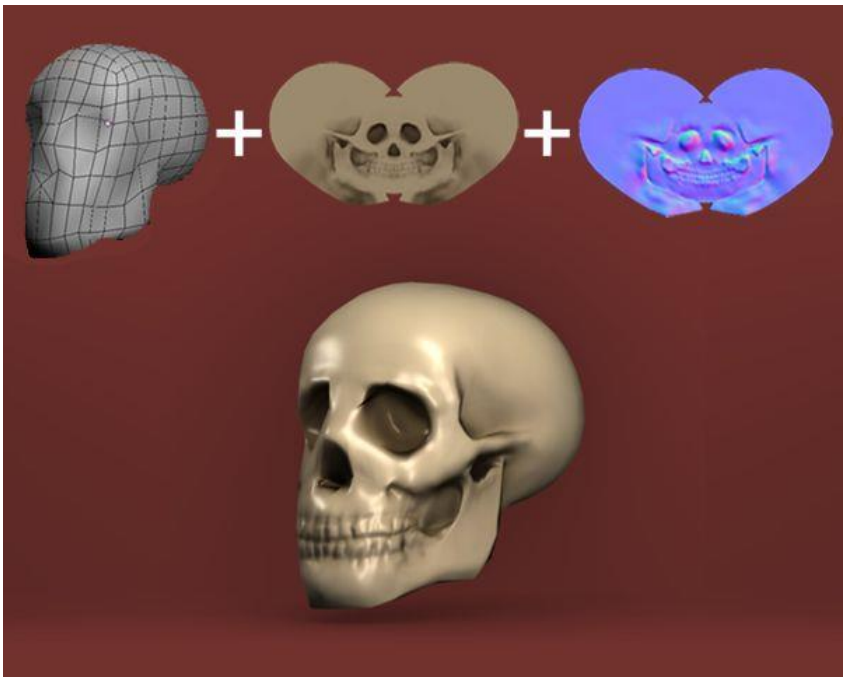

Figure 32. Low poly mesh with a texture and a normal map appears as a detailed rendered object [59].

The standard workflow starts with creating a high poly model. High poly models are usually created by sculpting - a method of modeling that imitates real life sculpting by modifying the position of all vertices under the "brush". A low poly model is then created by duplicating the high poly model and reducing the level of complexity of one of them. Figure 33 demonstrates a high poly (right), and a low poly (left) model of a skull created in this way.

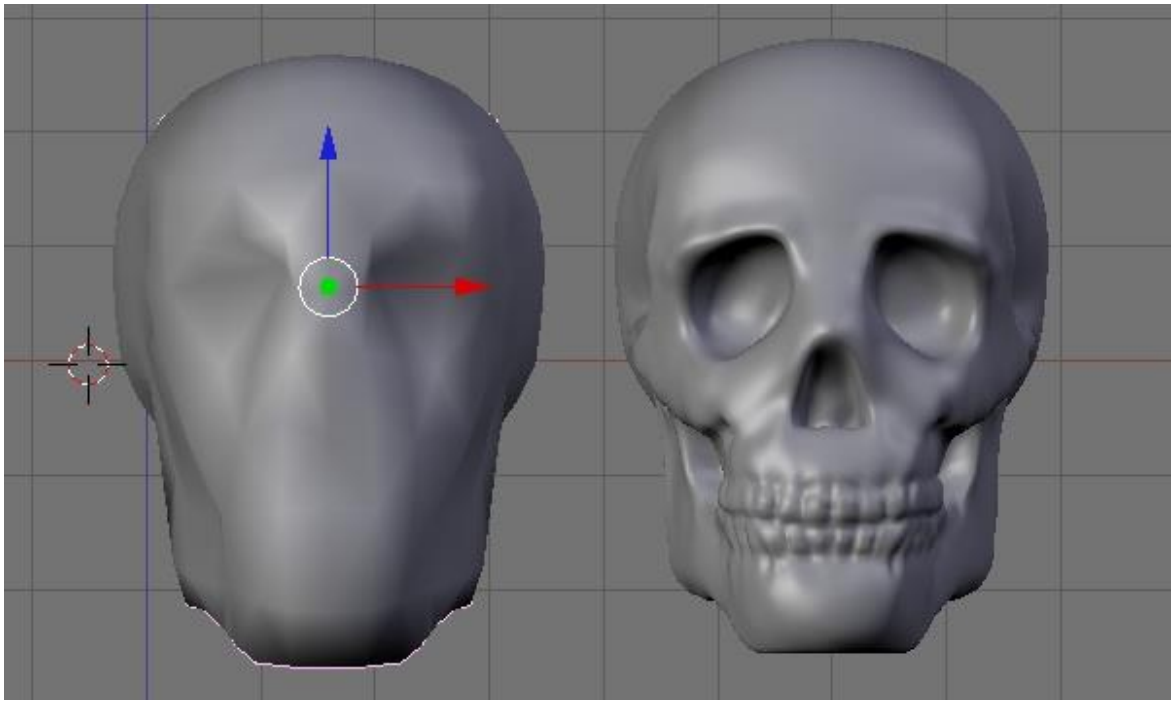

Figure 33. Low poly (left) and high poly (right) model of a skull [60].

The low poly model is unwrapped to get a UV layout. The low poly is later used in the game engine, and the maps are applied to it. For baking details onto a normal map, both models have to be located in the same position. A new texture image is created for the low poly model. A normal map is then baked from the high poly model onto the new texture on the low poly model. The low poly model can then be used for rendering with the created maps, and it will appear to be as detailed as the high poly model.

To create the effect of chamfered edges using normal maps, both a low poly model with hard edges and a high poly model with chamfered edges are needed. Using these models, the above-mentioned workflow will create a normal map for the low poly model to look chamfered. An example of the result would be a normal map (Figure 34) that makes the edges appear chamfered, a low poly chair model (Figure 35), and the rendered chair model (Figure 36) with texture and normal map applied.

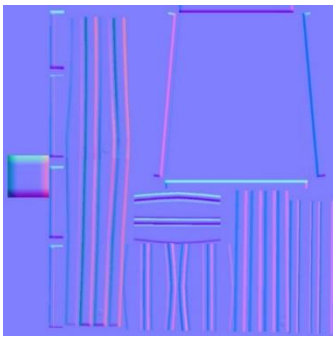

Figure 34. Normal map for the chair in Figure 35 [61].

Figure 35. Low poly chair model [62].

Figure 36. Chair model from Figure 35 with texture and normal map applied [63].

Rendering the low poly model with the normal map is easier on performance than using the high poly model itself. After the normal map is created, the level of detail will not be modifiable afterward without creating another normal map. More work will be needed to create this model initially since two models would have to be created. The process of creating normal maps for many models is time-consuming and not efficient if the normal maps are used only to chamfer edges. For very big and complex models, like the aforementioned Gladiator (Figure 23), the normal maps will be very big and could cause a memory bottleneck. In addition to requiring memory to be stored, normal maps will make rendering the triangles in the fragment shader more complex since the normal maps are used in the calculations. However, using a normal map for adding detail will still be easier to render than using the high poly model.

6 Edge chamfering algorithm

The input of the edge chamfering algorithm is a model that can consist of multiple meshes, and a chamfer scale which is a float. The chamfer scale determines how wide the chamfered edges will be for the whole mesh, or in other words, it is the radius of the chamfer. The algorithm modifies the geometry of the input model to chamfer any hard edges of the mesh(es). The output is a model that is identical to the input model except the edges of the output model are chamfered according to the chamfer scale.

The edge chamfering algorithm consists of 7 main steps.

1. Populating data structures.
2. Separating corner-connected vertices.
3. Pulling vertices on hard edges apart.
4. Recreating UV seams.
5. Creating faces between hard edges.
6. Filling the holes where multiple hard edges met.

This chapter will describe each of these steps. Figure 37 compares cubes before chamfering (left), after step 3 (middle), and after step 6 (right).

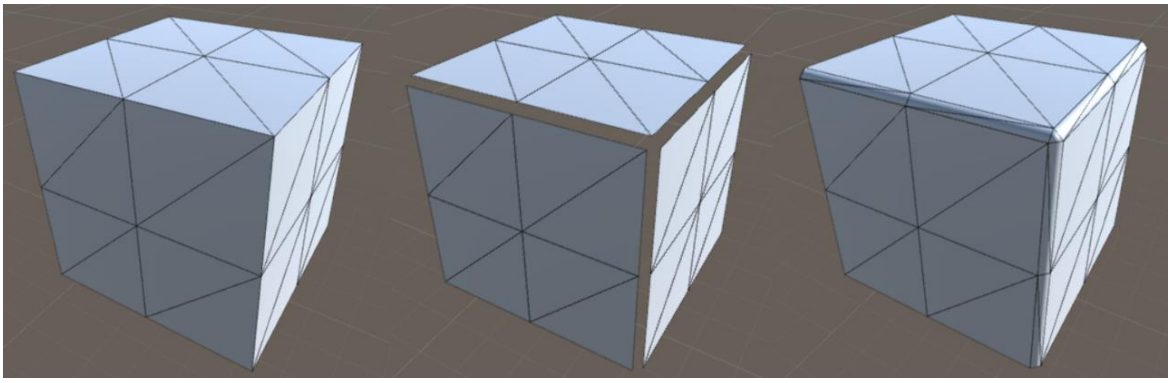

Figure 37. Cube before chamfering (left), after step 3 of chamfering (middle), and after step 6 of chamfering (right).

After chamfering is done, a new mesh is created using the modified lists. New parameters set for the new mesh are vertices, triangles, normals, tangents, colors, and UV coordinates for all four channels. Then, the bounds of the mesh are recalculated, and the mesh is returned.

Using the transforms of the new and old version of the models, the hierarchy of the original model is applied to the new chamfered models. Positions, rotations, and other components of the old mesh are also applied to the new objects.

6.1 Populating data structures

The input of the algorithm is a Unity Mesh [64] object (referred to as mesh in this thesis), and chamfer scale (float). The algorithm will modify the properties of the mesh. The existing properties are saved and used to populate various data structures to speed up and simplify the algorithm. Figure 38 illustrates the workflow of how the data structures are populated.

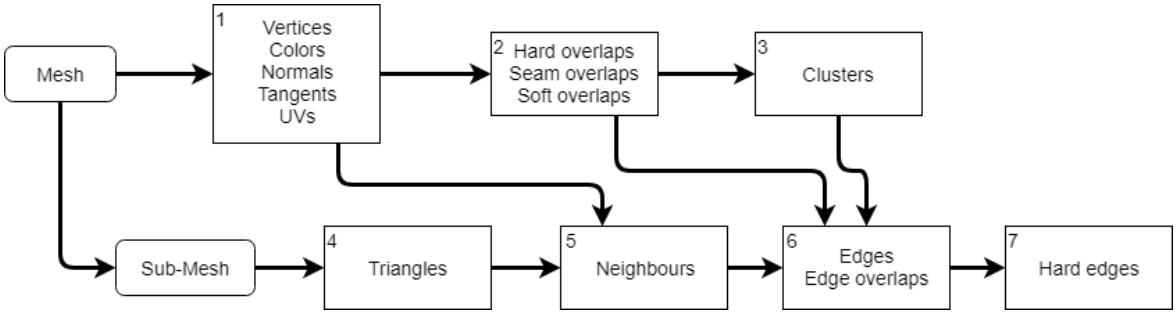

Figure 38. Workflow of populating the data structures.

First four (box 1 in Figure 38): vertices, colors, normals, tangents, and UVs, are fetched straight from the mesh and saved to Lists [65].

Using vertices, normals, and UVs, different overlaps are detected and saved (box 2). Overlaps are vertices that have the same position. Multiple vertices in one position are needed when the vertex in that position has to have different properties, for example different normals. Since one vertex can only have one set of properties, multiple vertices have to exist instead of only one. The algorithm saves three types of overlaps for each vertex:

- hard overlaps,
- seam overlaps,
- soft overlaps.

Table 2 shows the overlap types, and if the parameters are same or different. Hard overlaps are overlaps that have different normals. These overlaps will be used to detect hard edges. Seam overlaps are overlaps that have the same normals but different UV coordinates. First UV channel is used for finding seam overlaps since seams are the same for all UV channels. Soft overlaps have the same position, same normals, and same UV coordinates.

	position	**normal**	**UV coordinates**
Hard overlap	Same	Different	*Not checked*
Seam overlap	Same	Same	Different
Soft overlap	Same	Same	Same

Table 2. Different types of overlaps.

Based on overlaps, the vertices are mapped into clusters (box 3). One cluster consists of vertices that have the same position. Clusters are saved as a List of integers. The *n*th element in the cluster list is the cluster index of the *n*th vertex. Vertices in the same cluster have the same cluster index.

Following data will be sub-Mesh specific (second row in Figure 38). For every material used in the mesh, there exists a sub-Mesh with its own separate triangles. The triangle list is fetched from the sub-Mesh, and saved (box 4). From the triangles, neighbors are also identified for each vertex and saved (box 5). Neighbors are vertices, that are connected with an edge.

Edges are mapped using overlaps and clusters (box 6). Overlapping edges are also saved as either hard-, seam-, or soft overlaps. Using the edge map, a list of all hard edges is created (box 7).

6.2 Separating corner-connected vertices

Usually, one vertex is connected to one set of edge-connected triangles. Modeling software or modelers can create cases where a vertex connects to more than one set of edge-connected triangles [66]. Figure 39 shows vertex A, on the left, with one edge-connected set of triangles. On the right in Figure 39 is vertex B with two edge-connected sets of triangles. Vertex B is a corner-connected vertex since it connects the corners of multiple edge-connected triangle sets.

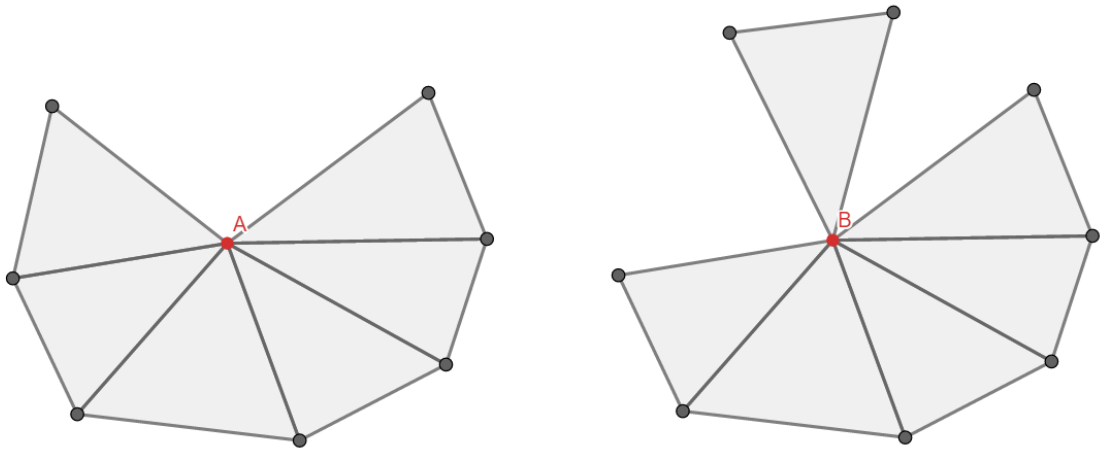

Figure 39. Vertex A (left) with one edge-connected triangle set, and vertex B (right) with two edge-connected triangle sets.

An example where corner connected vertices can create problems for chamfering is a model with three extruding cubes (Figure 40). There are three corner-connected vertices where the three cubes meet. The corners are connected because the normal vector of these faces is the same. On the right in Figure 40, faces belonging to one connected group have the same color. These corner-connected vertices are illustrated in Figure 41 with the faces in the connected group.

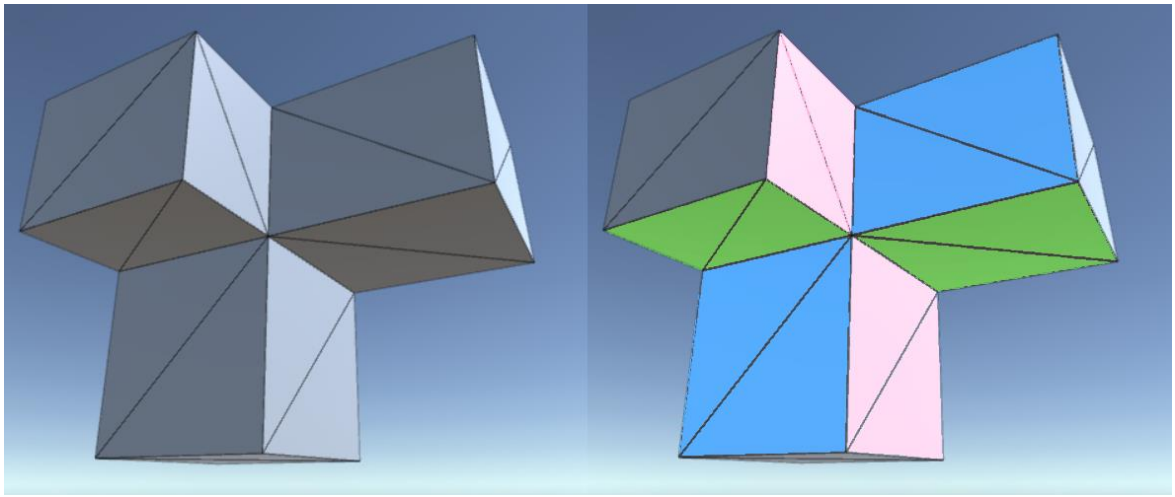

Figure 40. Model with three corner-connected vertices in the middle.

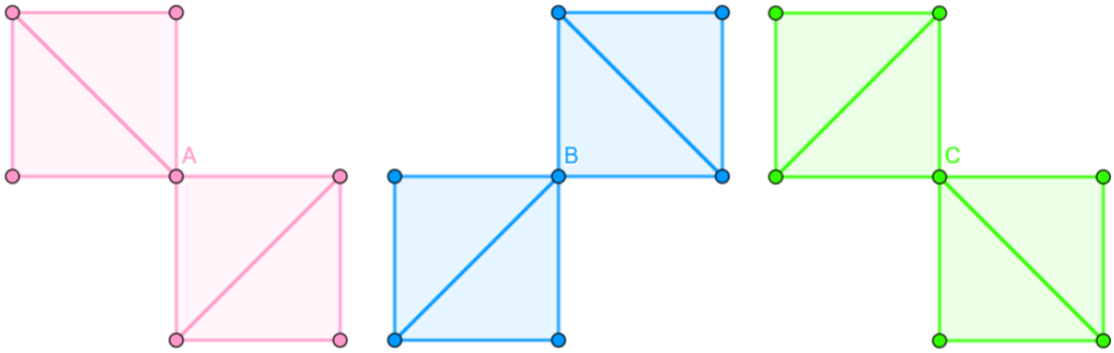

Figure 41. Colored faces from Figure 40, and their vertices. A, B, and C are corner-connected vertices.

If the corner-connected vertices were not separated, they cannot be pulled apart to make room for the chamfer. Without separating corner-connected vertices, the middle part of this model after chamfering could look like the left side of Figure 42. With separation, the result looks like the right side of Figure 42.

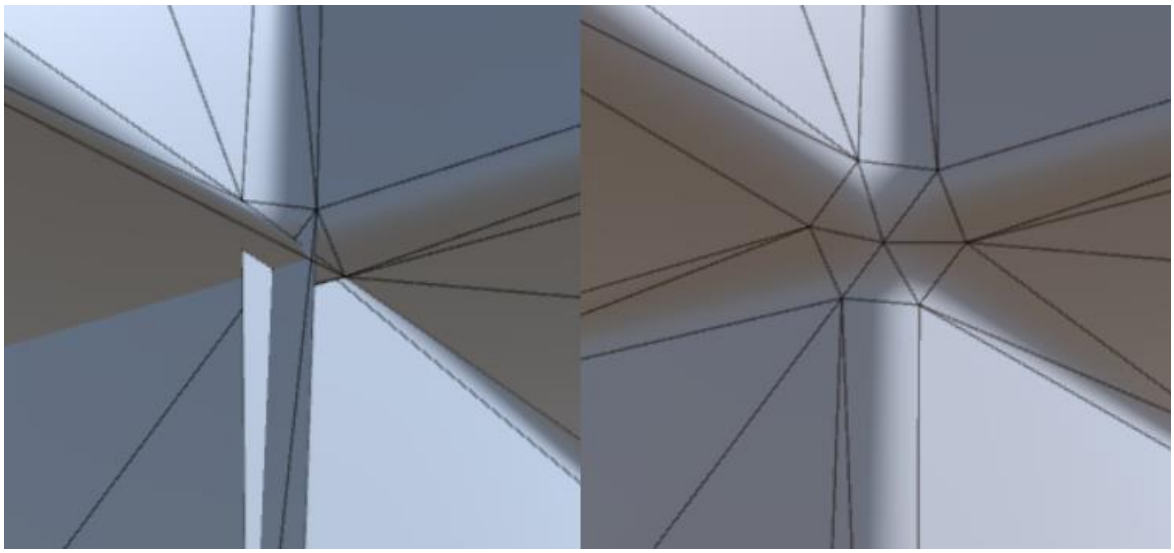

Figure 42. The middle part of the model in Figure 18 after chamfering without separating corner-connected vertices (left), and with separating corner-connected vertices (right).

Corner-connected vertices are usually on hard edges, and they would have to be pulled into two while chamfering. For this, a new vertex has to be created, and all the edges (and therefore triangles) have to be connected to the right vertices. Figure 43 shows a corner-connected vertex before and after separation.

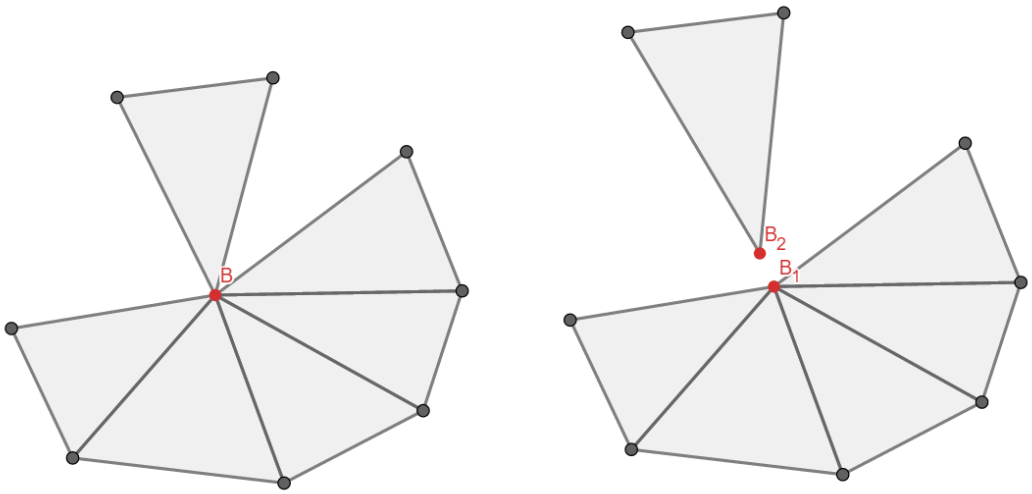

Figure 43. Corner-connected vertex before (left) and after (right) separation.

The process of separating corner-connected vertices consists of three sub-steps:

1. Grouping the neighboring vertices of the middle vertex.
2. Creating a new vertex for every group after the first one.
3. Connecting all edges and triangles to the new vertex if they are not in the first group.

Sub-step 1. First, all neighboring vertices that are connected to the middle vertex via a hard edge are found. If there are more than two, then there is a corner-connected vertex. Otherwise, nothing has to be done in this part of the algorithm, and this vertex is ready for pulling. Then, the neighbors are ordered by their position around the middle vertex and its normal vector.

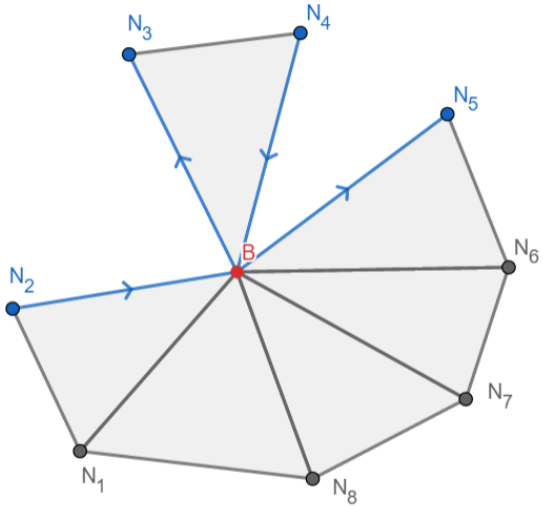

Figure 44. Neighbors ordered around vertex B and sorted by type. Blue – hard edge with its direction shown; grey - soft edge.

In Figure 44, ordered vertices are labeled N_1 to N_8. The first vertex N_1 is chosen randomly. The others are ordered using the signed angle between the vectors from the center to the neighbor (vectors $\overrightarrow{BN_1}$, $\overrightarrow{BN_2}$, etc.). Next, all the neighboring vertices are categorized according to what sort of edge they are connected with. In reference to the middle vertex, the categories are as follows: hard edge in, hard edge out, and soft edge. Soft edges are doubled,

with different directions, one for each triangle on either side of the soft edge. In Figure 44, hard edges are shown as blue with an arrowed line, and soft edges are colored grey. Unity uses clockwise winding order, meaning that edges BN_3, N_3N_4, and N_4B create a front-facing triangle with edge BN_3 going out of the middle vertex B, and N_4B going in. Then, groups are formed using the sorted neighbors so that a group starts with a hard edge going out, and ends with a hard edge going in. For the vertex B in Figure 44, the groups will be [N_3, N_4] and [N_5, N_6, N_7, N_8, N_1, N_2].

Sub-step 2. For each group after the first, a new vertex is created. If there are two groups, one vertex is created; if there are three groups, two new vertices are created, and so on. New vertices will have the same information (position, normal, UVs etc.) as the original middle vertex.

Sub-step 3. The first group will stay connected to the middle vertex A, and other groups will be connected to a new vertex. Triangles and edges are changed to reflect the new vertices. For all groups, neighbors and overlaps have to be changed so the old middle vertex, and the added new vertices would be in the right lists. Neighbors of neighbors are also updated to avoid conflicts.

After the separation of all corner-connected vertices is complete, neighbor lists are modified to also include seam- and soft overlaps of the vertices. For pulling vertices apart, two hard edges have to be found for a vertex on a hard edge. It can happen, that one hard edge is connected to the vertex, and the other is connected to its the seam-, or soft overlap. For example, in Figure 45, where a hard edge (blue) intersects a seam (red). At the intersection point, the edges would be connected to vertices as shown in Figure 46. If the neighbors of the soft and seam overlaps are added to the neighbor lists, the edges can be found in the next part of the algorithm.

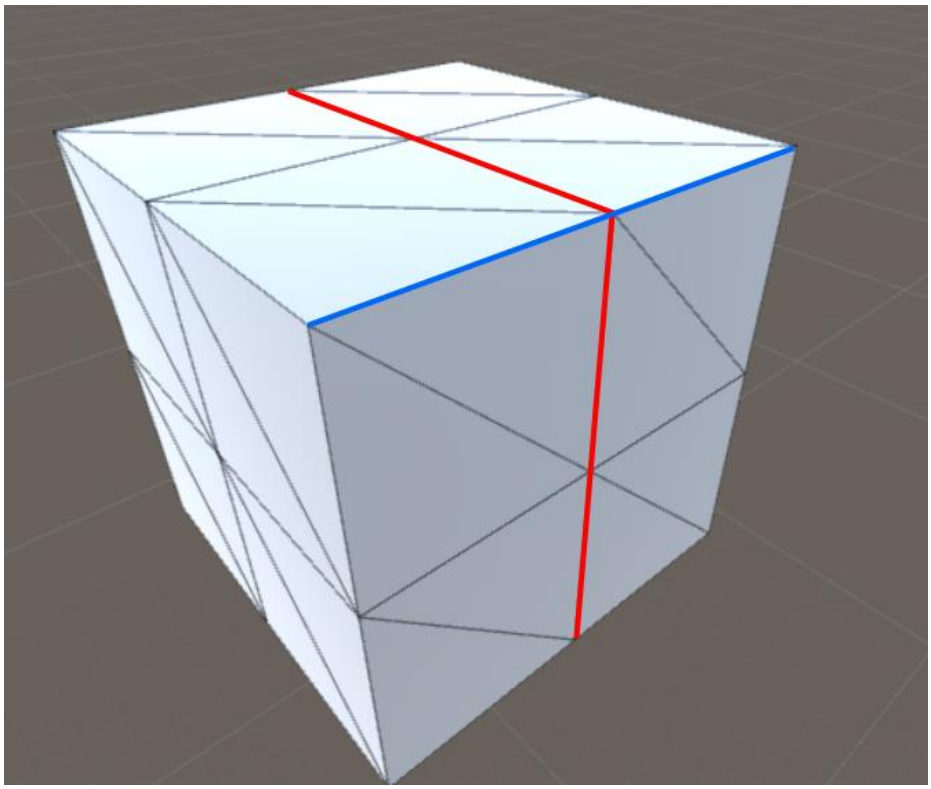

Figure 45. A cube where a hard edge (blue) intersects a seam (red).

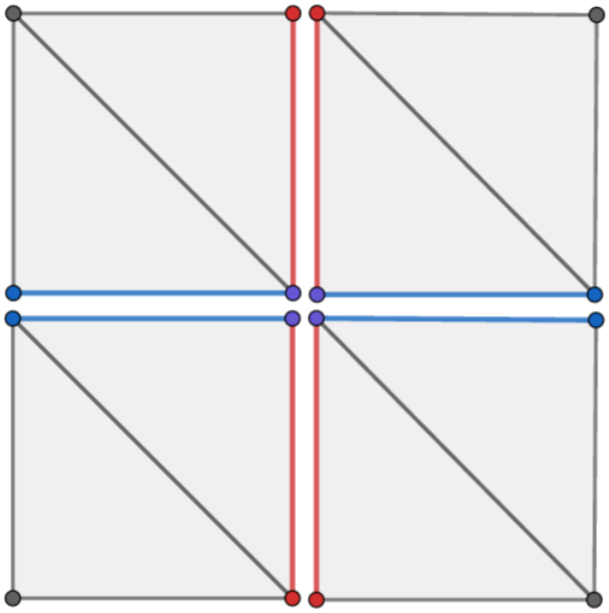

Figure 46. Illustration of how the vertices are connected at the intersection of a hard edge (blue), and a seam (red) in Figure 45.

6.3 Pulling vertices on hard edges apart

To make room for the chamfer, all duplicate vertices on hard edges would have to be pulled apart. Pulling apart vertices means changing their position, perpendicular to their normal vector, to be farther from the others.

Figure 47 displays a cube with normals shown as blue arrows. Figure 48 shows the cube after pulling the vertices on hard edges apart. Note, that the mesh does not get bigger by pulling the vertices apart. The faces remain at the same positions but get smaller if they have a hard edge.

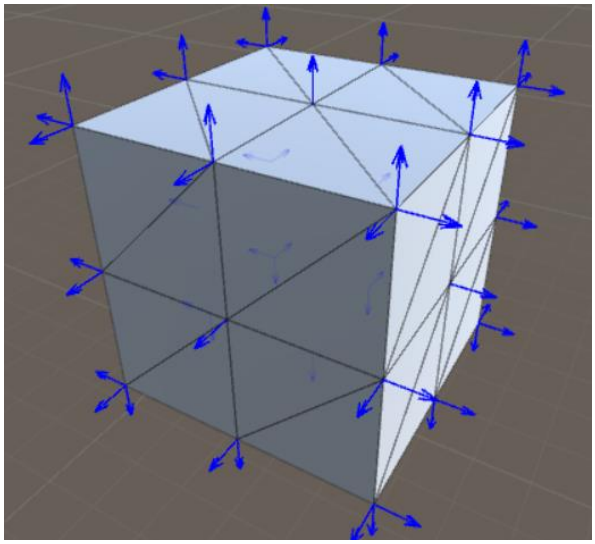

Figure 47. Cube with normals shown as blue arrows.

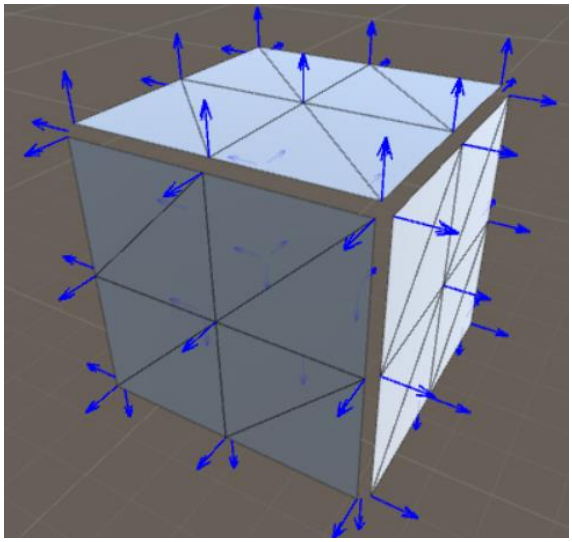

Figure 48. Cube from Figure 47 after pulling the vertices apart.

Figure 49 depicts a side-view of the cube where vertices B_1 and B_2 are pulled to positions B_1' and B_2' respectively. The green dotted line is the position of the chamfer created between them in step 5 of edge chamfering. Pulling B_1 and B_2 to positions B_1' and B_2' on the model is shown in Figure 50.

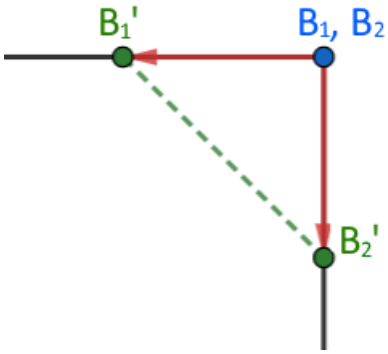

Figure 49. Side view of pulling two vertices A and B on the edge of a cube. The new positions are A' and B' respectively, and faces are created at the dotted green line in step 5.

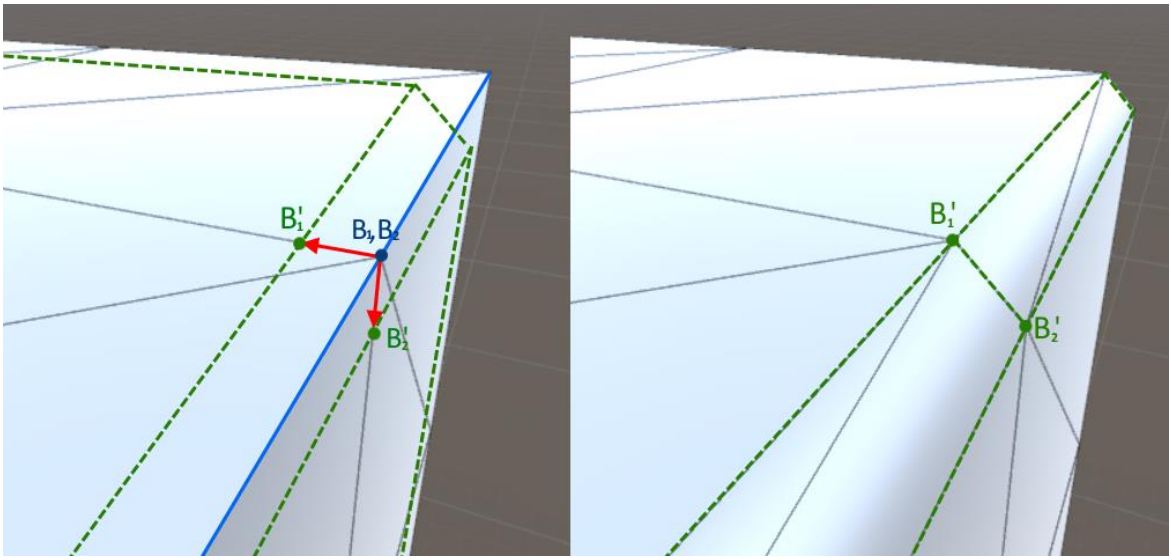

Figure 50. Pulling vertices B_1 and B_2 to positions B_1' and B_2' (left), and the resulting chamfer between B_1' and B_2' (right).

Vertices are pulled one at a time without looking at its hard overlap(s). In Figure 49 and Figure 50, vertex B_1 is pulled to the position B_1' without looking at B_2 and vice versa. For pulling a vertex, the hard edges (one incoming and one outgoing), and the vertex normal are needed. The following assumes the vertex to be pulled is denoted by B, and hard edges are AB and BC (Figure 51).

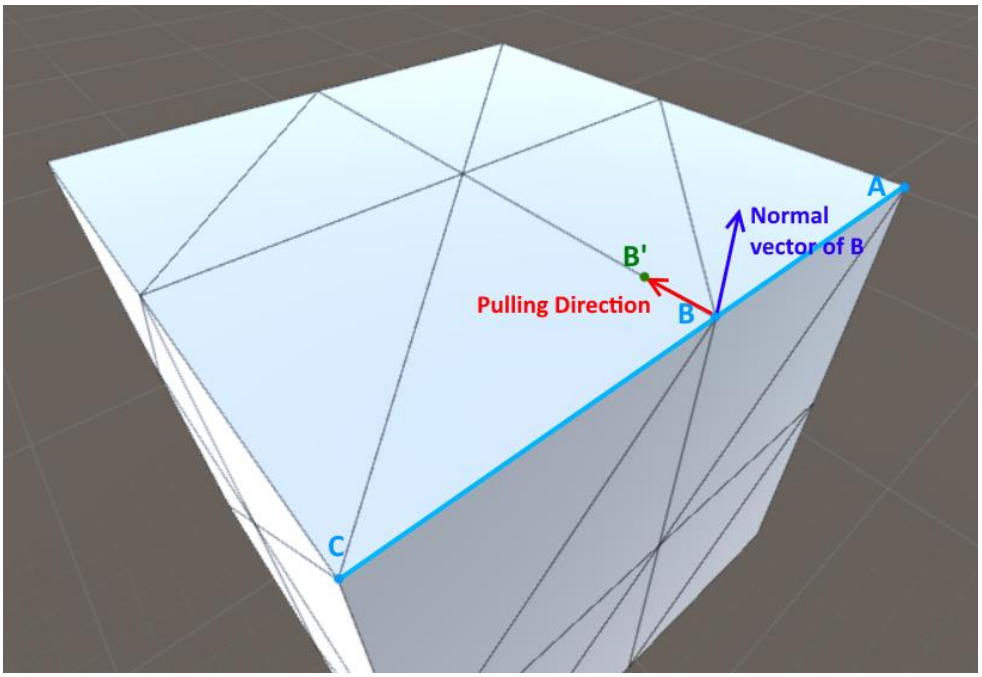

Figure 51. Pulling vertex B to position B'. AB and BC are hard edges, the dark blue arrow is the normal vector of B, the red arrow is the pulling direction.

Vertices are pulled using following sub-steps:

1. Finding hard edges AB and BC, and vectors $\overrightarrow{BA}$ and $\overrightarrow{BC}$.
2. Checking whether hard edges AB and BC are aligned.
    a. Calculating pulling direction for aligned hard edges.
    b. Calculating pulling direction for not aligned hard edges.
3. Calculating the distance modifier.
4. Calculating the new vertex position.
5. Calculating the new UV coordinates.

Sub-step 1. First, two vectors are needed, $\overrightarrow{BA}$ and $\overrightarrow{BC}$. B is the vertex we are pulling, and hard edges go from A to B, and from B to C. Figure 52 depicts the top-down view from the vertex normal direction of a vertex in the middle of the hard edge on the cube (vertex B in Figure 51). The direction of the edges AB and BC are shown with arrows.

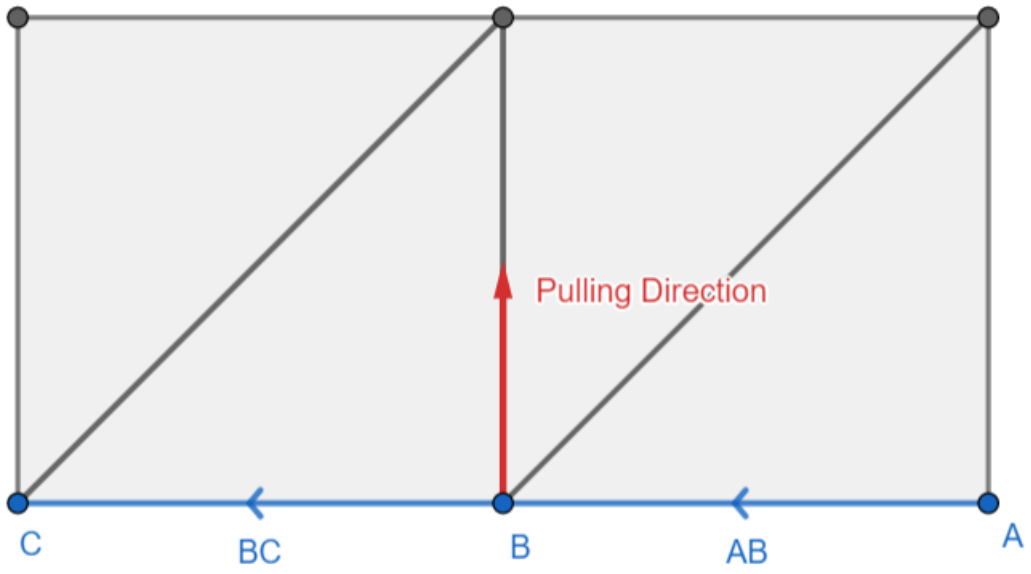

Figure 52. Vertex B is in the middle of a hard edge on a cube. Hard edges are AB and BC.

First, A and C must be found from the neighbors of B. A and C must be connected to B via an outgoing hard edge, and incoming hard edge respectively. Vectors $\overrightarrow{BA}$ and $\overrightarrow{BC}$ are found by calculating $B - A$, and $C - B$ respectively.

To make sure that the pulling is done on the plane perpendicular to the normal vector of the vertex, $\overrightarrow{BA}$ and $\overrightarrow{BC}$ are projected onto that plane. This way, the vertex is always pulled along that plane even if the adjacent faces are curved, for example on a cylinder with some soft edges (Figure 53).

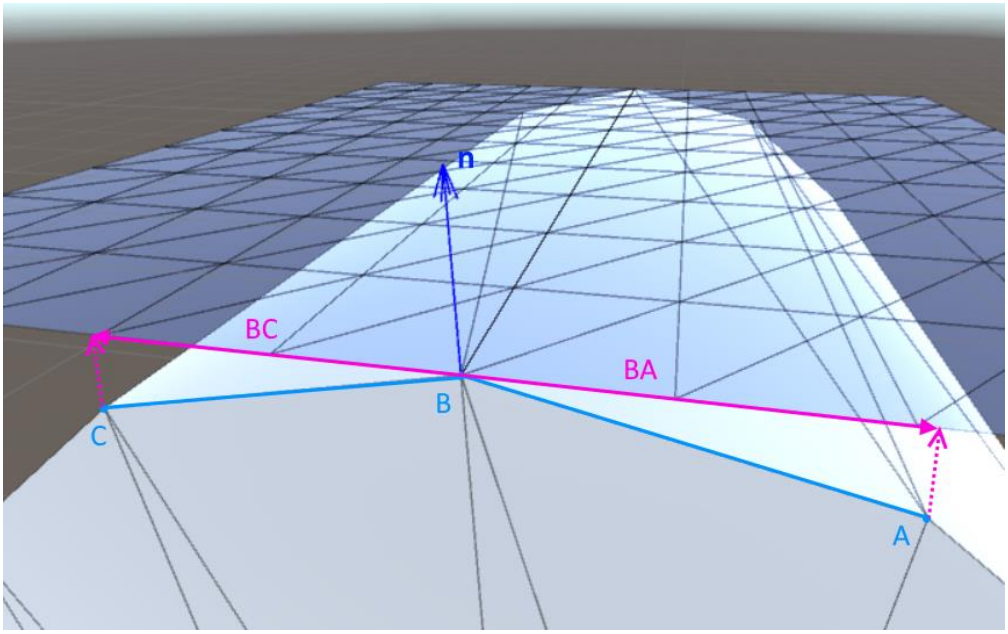

Figure 53. Projecting $\overrightarrow{BA}$ and $\overrightarrow{BC}$ onto the plane (transparent blue) perpendicular to the vertex normal on the curved part of a cylinder. Pink dotted vectors show where A and C are projected. Pink solid vectors are the projected $\overrightarrow{BA}$ and $\overrightarrow{BC}$.

Sub-step 2. The direction in which the vertex is pulled is calculated in one of two methods. The first method (sub-step 2.a) is used when the hard edges are aligned. Otherwise, the second method (sub-step 2.b) is used. Hard edges are aligned if they lie on the same straight line in 3D space. Figure 54 illustrates both cases. Hard edges AB and BC are aligned while BC and CD are not.

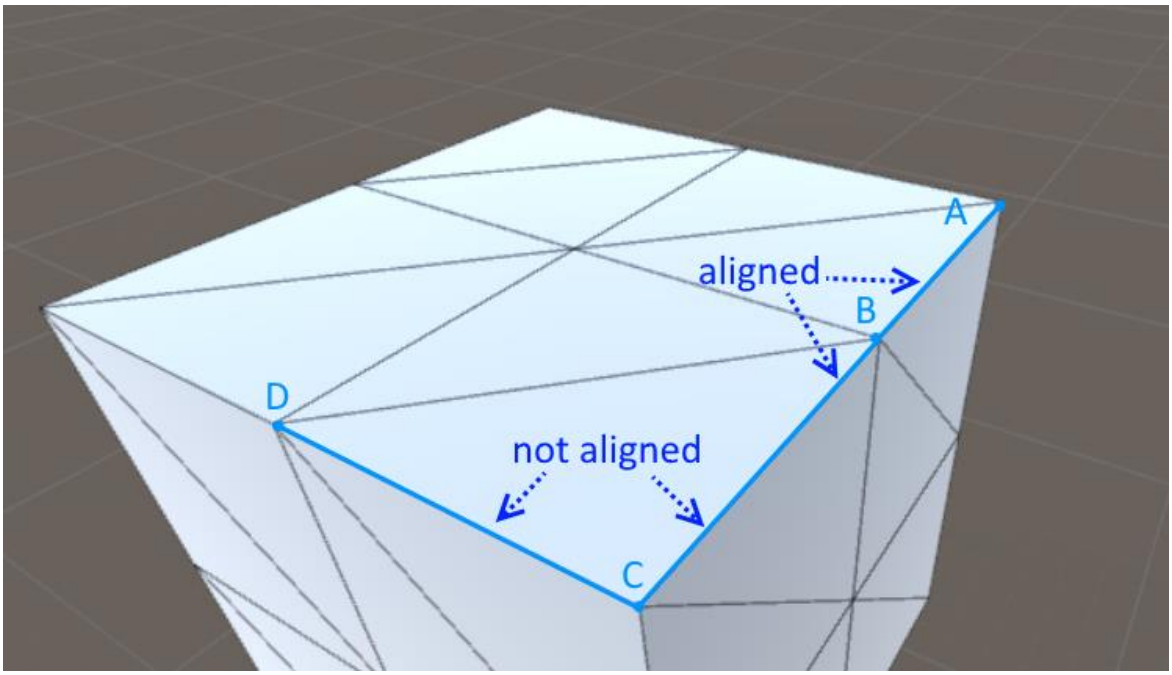

Figure 54. Example of aligned hard edges, and not aligned hard edges. Hard edges connected to B are aligned (AB and BC) while hard edges connected to C are not aligned (BC and CD).

To check, if the hard edges are aligned, the cross product of $\overrightarrow{BA}$ and $\overrightarrow{BC}$, $\overrightarrow{BA} \times \overrightarrow{BC}$, is used. Cross product of two vectors is a vector that is perpendicular to both of the vectors [67]. If the vectors are aligned, every vector on the plane perpendicular to one of those vectors is perpendicular to both vectors, and the cross product is a zero vector. If $||\overrightarrow{BA} \times \overrightarrow{BC}|| = 0$, the edges are aligned, and the first method is used.

Sub-step 2.a. In the case of aligned hard edges, the pulling direction is simply the cross product of the normal vector of vertex B ($\vec{n}$), and $\overrightarrow{BC}$.

$$pullingDirection = \vec{n} \times \overrightarrow{BC}$$

Figure 52 shows the pulling direction for aligned edges with a red arrow. The vertex will be pulled perpendicularly to the normal vector and the hard edge in the direction of the face.

Sub-step 2.b. Second method, used when hard edges are not aligned, is to first find the angle bisector of $\overrightarrow{BA}$ and $\overrightarrow{BC}$. The vertex should be pulled along this vector. The angle bisector is where the angle between $\overrightarrow{BA}$ and $\overrightarrow{BC}$ is smaller, but the face could be in the opposite direction (Figure 55).

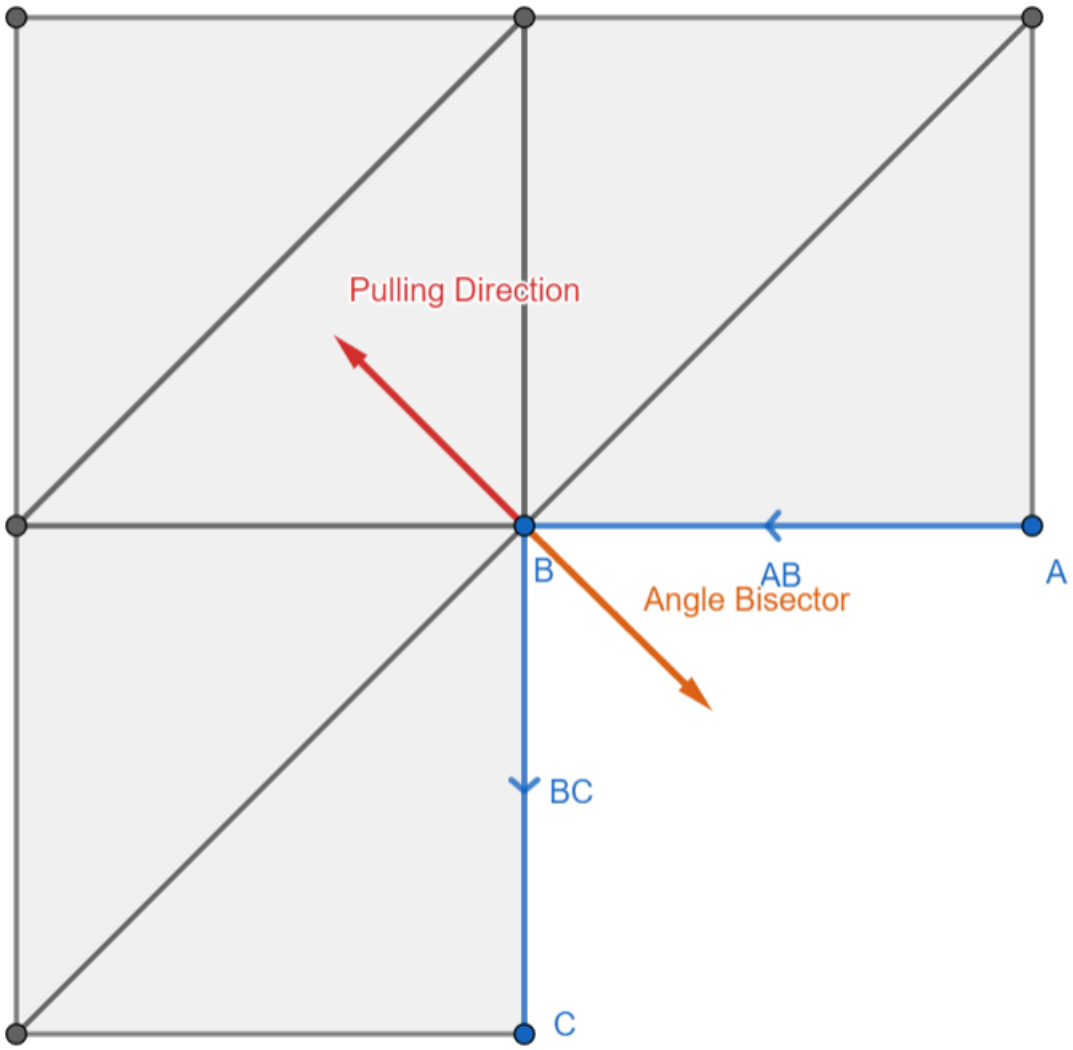

Figure 55. Pulling direction for vertex B when the hard edges do not align.

If the angle between $\overrightarrow{BA}$ and $\overrightarrow{BC}$ is larger than 180° (counterclockwise when viewed top-down from the vertex normal direction), then the angle bisector and the pulling direction are contradirectional (have opposite directions). To determine if the pulling direction should be the opposite of the angle bisector, the scalar triple product [68] ($(\overrightarrow{BA} \times \overrightarrow{BC}) \cdot \vec{n}$) can be used, which is a combination of cross product and dot product. The cross product and the normal vector are compared to see if they are pointing in the same direction or the opposite which tells us the same about the angle bisector and the pulling direction.

The cross product $\overrightarrow{BA} \times \overrightarrow{BC}$ will always be either a zero vector or parallel to the normal vector of B since $\overrightarrow{BA}$ and $\overrightarrow{BC}$ were projected onto the plane of the normal vector in sub-step 1. In the current sub-step, the cross product cannot be a zero vector since then the hard edges are aligned, and this case is already covered in sub-step 2.a. Therefore, the cross product is a vector parallel to the normal vector.

If the angle between $\overrightarrow{BA}$ and $\overrightarrow{BC}$ is less than 180°, then $\overrightarrow{BA} \times \overrightarrow{BC}$ will point outward from the faces, and it will be codirectional to the normal vector of B. If the angle between $\overrightarrow{BA}$ and $\overrightarrow{BC}$ is more than 180°, the cross product will point inward in reference to the mesh, and in the

opposite direction of the normal vector. Now, it needs to be determined if the cross product and the normal vector are codirectional or contradirectional.

From the geometric definition of dot product [69], $a \cdot b = \| a \| \| b \| \cos(\theta)$, where θ is the angle between vectors a and b. If the angle between a and b is more than 90°, then $\cos(\theta) < 0$, and the dot product will also be negative since vector lengths are non-negative. If the cross product and normal vector are pointing towards a similar direction, then the angle between these vectors is less than 90°, and their dot product is larger than 0.

Therefore, if $(\overrightarrow{BA} \times \overrightarrow{BC}) \cdot \vec{n} < 0$, then it means the cross product is contradirectional to the normal vector, and the pulling direction is the inverted angle bisector. Otherwise, the angle bisector is also the pulling direction. Figure 55 shows a vertex B that has non-aligned hard edges. The angle between the hard edges is larger on the side where the faces are, and the pulling direction is the inverted angle bisector.

An alternative way to determine if the angle between $\overrightarrow{BA}$ and $\overrightarrow{BC}$ is bigger than 180°, would be to use the Vector3.SignedAngle [70] method in the Unity game engine, and compare it to 0. However, the actual code of SignedAngle also uses the scalar triple product in addition to calculating the angle between the vectors [71]. Hence, it is more efficient to use the scalar triple product in place of Vector3.SignedAngle.

Sub-step 3. Pulled edges should be parallel to the old edges. The shortest distance between the old edge and the new for all hard edges in the mesh will be denoted by h. The value of h will be calculated by dividing the chamfer scale with a constant (see sub-step 4). For the old and new edge to be parallel, pulling distances for vertices have to differ when the angles between hard edges are different. Figure 56 illustrates the pulling distances of vertices A, B, and C so the new edge is parallel the old edge. In Figure 56, $h = 2\ units$, and pulling distances for vertices A, B, and C have the magnitude of ~2.8, 2, and ~2.2 units respectively. To get these pulling distances, a distance modifier is calculated using the angle between $\overrightarrow{BA}$ and $\overrightarrow{BC}$. Pulling distance is the product of h, and the distance modifier.

$$pullingDistance = h * distanceModifier$$

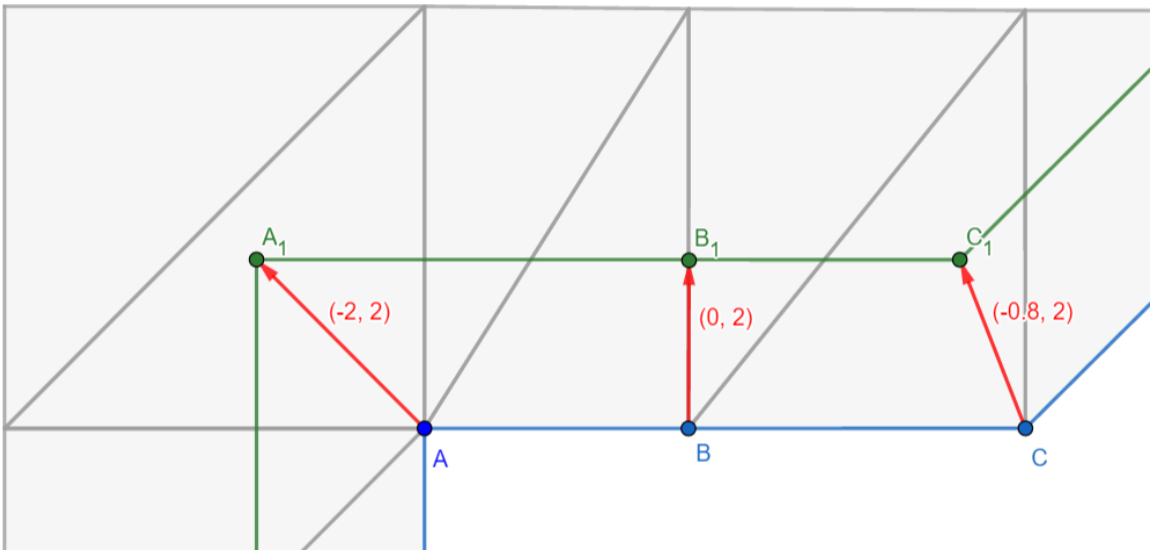

Figure 56. Pulling vertices with different angles between hard edges. Hard edges and original vertices are colored blue, pulling vectors are colored red, and new vertices and edges are colored green.

If the angle between the pulling direction and one hard edge is 90°, then the pulling distance is equal to *h*, and the distance modifier equals 1. For instance, in Figure 57, the pulling direction of B is perpendicular to its hard edges, and the pulling distance is equal to the distance between the edge BC, and the edge B_1C_1. BC and B_1C_1 are parallel. To calculate the distance modifier for pulling vectors non-perpendicular to the hard edge, a right triangle is constructed using the pulling direction, hard edge (or its extension), and *h*. The pulling vector is the hypotenuse of the right triangle.

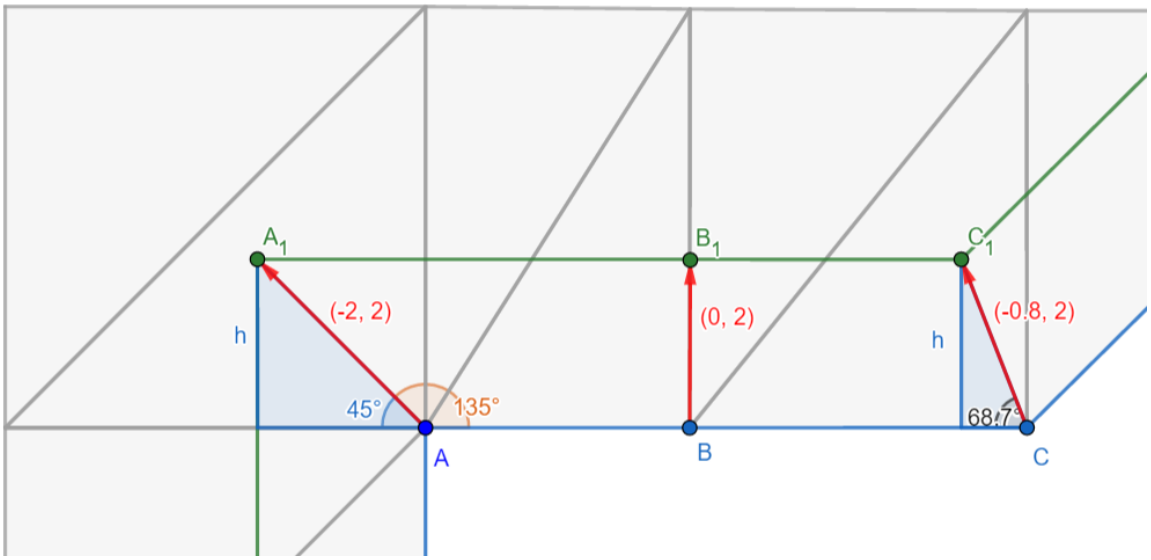

Figure 57. Calculating the distance modifier using right triangles and sine.

Knowing the length of *h*, and the angle opposite to *h*, we can use sine to calculate the length of the pulling vector. The angle opposite to *h* is half of the angle between the hard edges because the pulling vector is also the angle bisector. The angle between the hard edges $\overrightarrow{BA}$ and $\overrightarrow{BC}$ is denoted by α. The angle opposite to *h* is therefore α/2. From the definition of sine [72]: $\sin\frac{\alpha}{2} = \frac{h}{pullingDistance}$. A formula to calculate the distance modifier can be formed by setting $h = 1$. Then, the pulling distance is also the distance modifier.

$$\sin\frac{\alpha}{2} = \frac{h}{pullingDistance} \overset{h=1}{\Longrightarrow}$$

$$\sin\frac{\alpha}{2} = \frac{1}{pullingDistance} \Longrightarrow$$

$$distanceModifier = pullingDistance = \frac{1}{\sin\frac{\alpha}{2}}$$

Because $\sin(x) = \sin(180° - x)$, the formula works also when the angle between the hard edges is $> 180°$. An example would be vertex A in Figure 57, where $\frac{\alpha}{2} = 135°$ but the angle opposite to *h* is $45° = 180° - 135°$. To eliminate abnormally large pulling distances, modifiers bigger than 10 are changed to 0 so that the vertex would not be moved at all.

Sub-step 4. The distance between the old and new edge (*h*) is calculated by dividing the chamfer scale (given as a parameter for the edge chamfering algorithm) with a constant.

This way, changing the chamfer scale will linearly change the length or radius of the chamfer.

The new vertex position is calculated using this equation:

$$newPosition = oldPosition + pullingDirection * \frac{chamferScale}{20} * distanceModifier$$

Sub-step 5. The part of the algorithm that moves the UV coordinates was created during the course Computer Graphics Project [73]. If the mesh has been unwrapped, the UV coordinates of the vertices have to be moved during chamfering. To calculate the new UV coordinate, barycentric coordinates [74] are used (Appendix I).

To use barycentric coordinates, the triangle on the original mesh where the pulled vertex is closest to has to be found. The triangle is used for calculating the barycentric coordinates since it has both position and UV coordinates for all its three vertices. Ideally, the new vertex will be positioned inside this triangle, but it is not necessary for calculating the barycentric coordinates. To find the triangle, the neighbors of the vertex are sorted around the old position so that the first neighbor in the list is the first neighbor to the right of the new vertex position. Figure 58 shows how the neighbor N_1 is the first neighbor to the right of the new vertex position if sorted clockwise around the original vertex, and N_7 is the last. The sorting is done the same way as in step 2, sub-step 1. The triangle that is used for calculating UV coordinates is formed by the last neighbor, the first neighbor, and the original vertex. In Figure 58, the triangle is N_7N_1A.

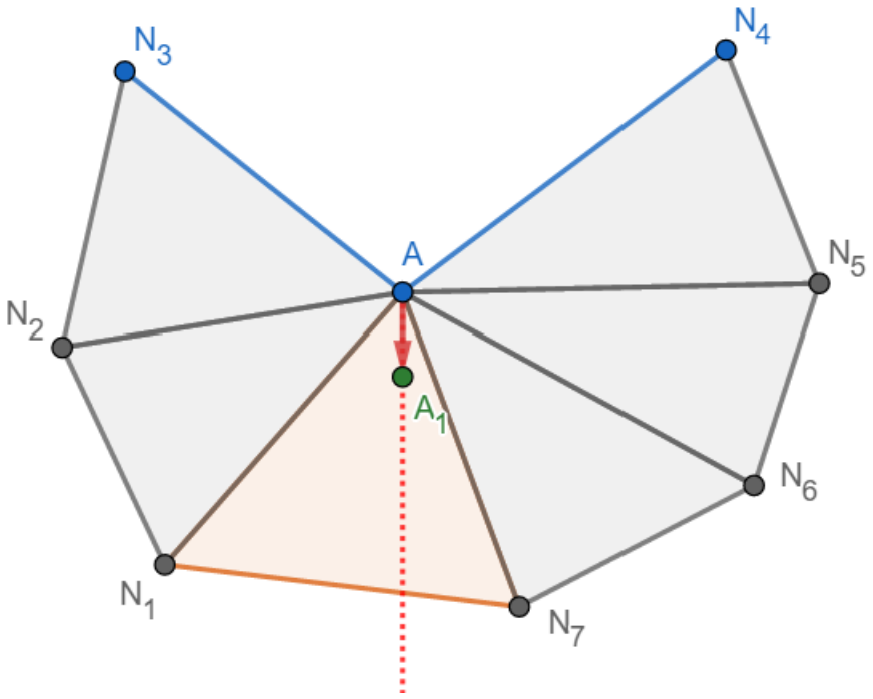

Figure 58. Neighbors of vertex A sorted from N_1 to N_7 around vertex A starting from the position of the new position A_1. The triangle used for calculating UV coordinates is N_7N_1A (orange). The red arrow is the pulling direction of A, and the dotted red ray is its extension.

Once this triangle has been located, the barycentric coordinates of the new vertex position (A_1 in Figure 58) are calculated in reference to the triangle. The UV coordinate of the new vertex is calculated using the barycentric coordinates, and the UV coordinates of the triangle vertices (N_7, N_1, and A).

6.3.1 An alternative way to calculate pulling vector

Another way to calculate the vertex pulling vector is to use the average of the edges connected to the vertex. For each vertex, a direction vector is calculated to all adjacent vertices. The average of these vectors is the pulling direction vector. The pulling distance is determined by the chamfer scale and the average length of the vectors to the adjacent vertices. In Figure 59, the pulling direction for vertex V is the average of vectors $\vec{a}, \vec{b}, \vec{c}$, and $\vec{d}$. Pulling direction for vertex V using this method is calculated as follows:

$$pullingDirection = \frac{norm(-1,0) + norm(0,1) + norm(1,1) + norm(1,0)}{4} = \\ = \left(\frac{1}{4\sqrt{2}}, \frac{1}{4} + \frac{1}{4\sqrt{2}}\right)$$

All direction vectors should be normalized ($norm(vector)$) so they would have an equal impact on the pulling direction.

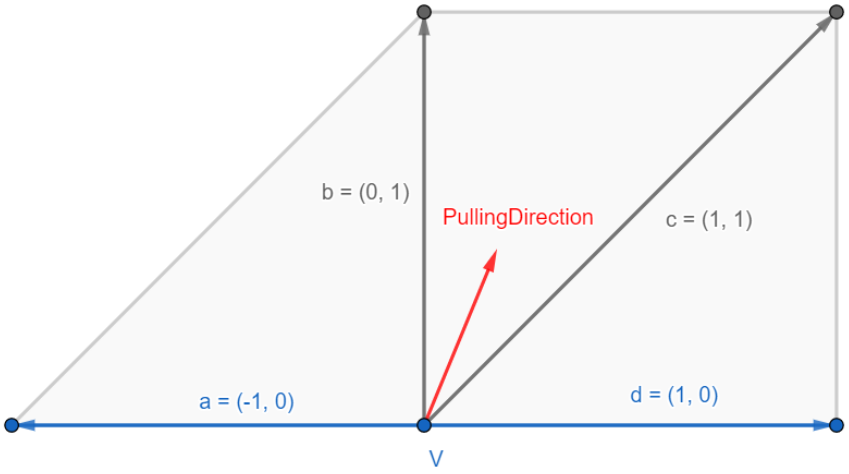

Figure 59. Pulling direction on a triangle strip calculated using the average of the vectors to all the neighbors.

This method of calculating the pulling direction is faster and less complex than the previously described method, but the result is not correct for every vertex. In some cases, the resulting average does not accurately represent the correct displacement direction. One such case is on the edge of a triangle strip with right-angled triangles. Figure 59 illustrates such case where the vertex V is to be pulled and vectors $\vec{a}, \vec{b}, \vec{c}$, and $\vec{d}$ are direction vectors to its adjacent vertices. This results in the vectors on the edge of a triangle strip to be twisted towards the direction of the diagonal edges when pulled.

6.4 Recreating UV seams

This step of the algorithm was also created during the course Computer Graphics Project. It is only needed for the UV map. If the mesh is not unwrapped and thus has no UV coordinates for its vertices, this step is skipped. This chapter explains the necessity of this step and in-

troduces the workflow of recreating the seams. The sub-chapter “Cases to consider” describes the special cases where the workflow has to account for additional factors. A more detailed description of how this part of the algorithm was created, and how it works, is available on the project page of the Computer Graphics Project course [73].

A UV seam has to be recreated if a hard edge and a seam overlapped. The problem becomes apparent when a chamfer is added between these hard edges. Since the edges have a different location on the UV map, the new faces will stretch over the UV map to connect them (Figure 60). This results in the chamfer having a weird texture that is usually a big part of the texture squeezed onto the small face of the chamfer (Figure 61).

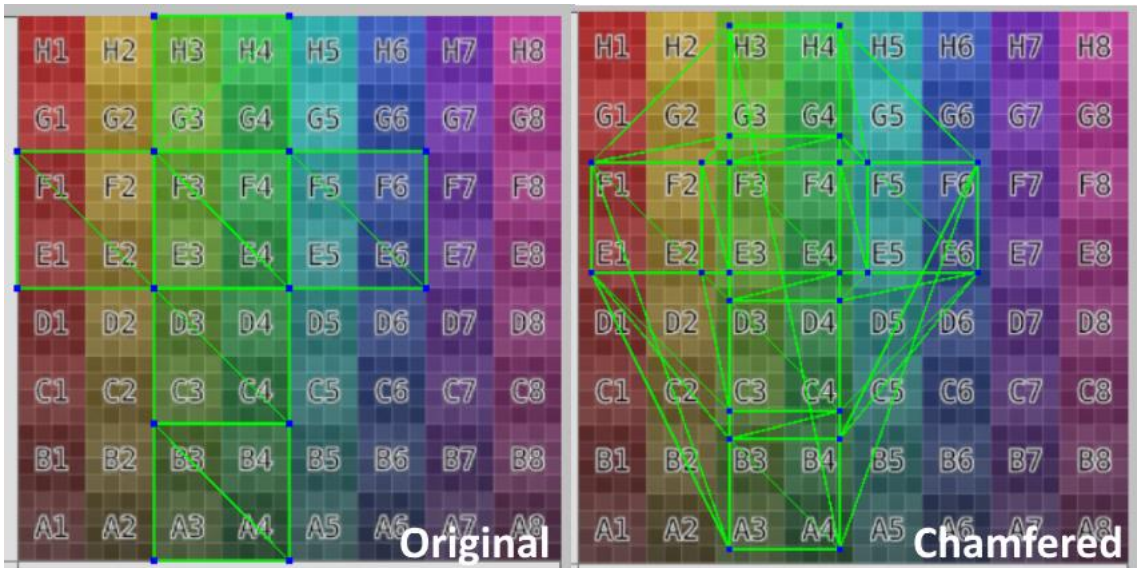

Figure 60. UV map of a seamed cube before chamfering (left) after chamfering if seams are not recreated (right). Edges, and therefore triangles stretch over other parts of the UV map.

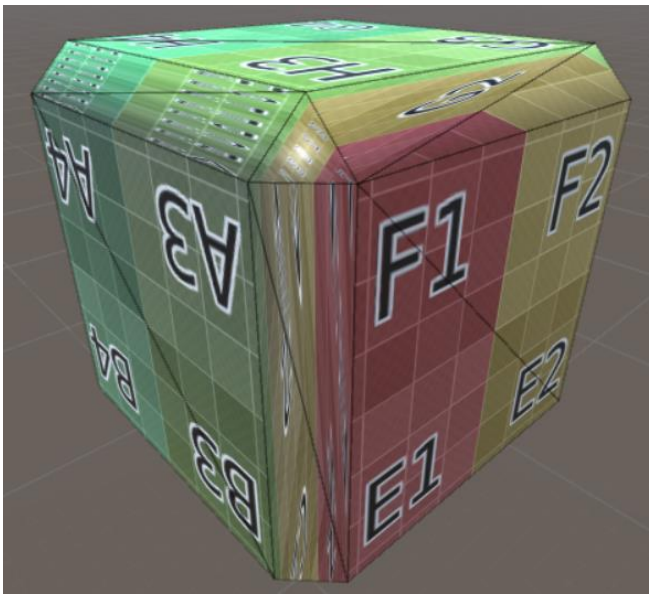

Figure 61. Seamed cube after chamfering without recreating seams corresponding to the UV map in Figure 60. Three chamfered edges are visible where the texture is mapped from a big part of the UV map.

Figure 62 illustrates a hard edge and seam A_1A_2, and its duplicate B_2B_1 being pulled apart. Seams are recreated one hard edge (A_1A_2 in Figure 62) with its duplicate (B_2B_1) at a time. The vertices of the edges are in turn handled one at a time with its duplicate vertex (A_1 and B_1, then A_2 and B_2).

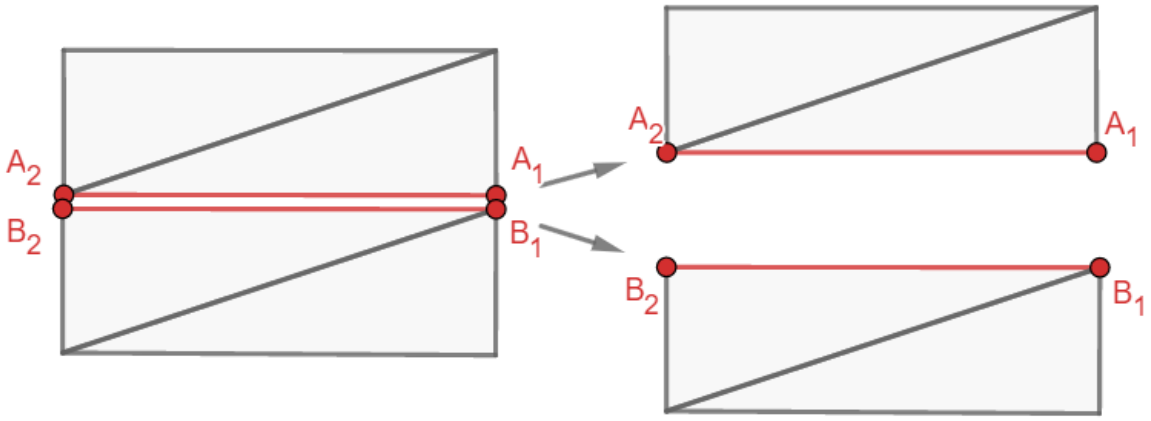

Figure 62. Hard edge that is also a seam being pulled apart.

The way the UV map will look is inspired by the Blender Bevel tool when used on an already unwrapped model. For every pair of vertices A_1 and B_1, the following sub-steps are taken:

1. Duplicate B_1, the new vertex is C_1.
2. Find the barycentric coordinates of C_1 for triangle $A_1oldA_1A_2$ (Figure 63), where $oldA_1$ is the position of A_1 before pulling.
3. Calculate the UV coordinates of C_1 using its barycentric coordinates, and the triangle constructed with the UV coordinates of A_1, $oldA_1$, and A_2.
4. If both A_1 and A_2 are dealt with, change the overlap from B_2B_1 to C_2C_1 so the new faces would be created between them, and update appropriate overlaps.

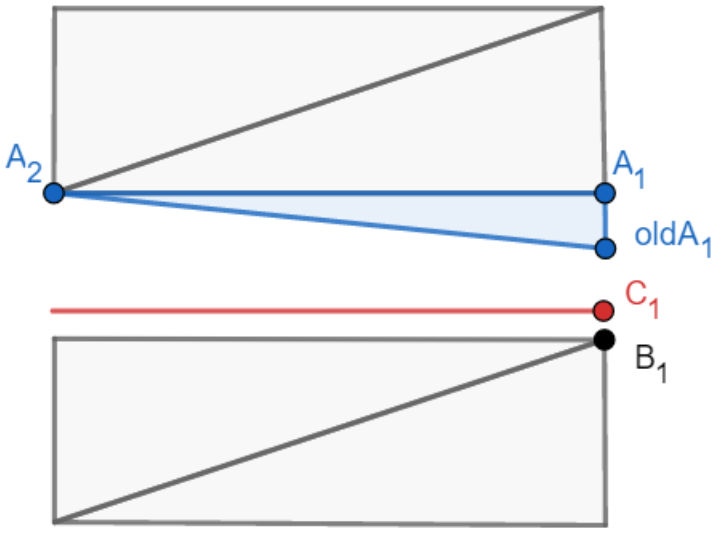

Figure 63. Calculating the UV coordinates of C_1. Barycentric coordinates of B_1 are calculated in reference to the triangle $A_1oldA_1A_2$ [75].

When these steps are done, the model looks visually the same, but new vertices are added. The faces created in the next step will be created using the vertices added in this step, which are close together on the UV map. The resulting chamfers will have textures that are mapped from the same place that was next to the hard edge previously, and will not feature any parts of the texture, that should not be on this part of the model. Figure 64 demonstrates the cube from Figure 61 after chamfering is done with recreating seams during chamfering. Figure 65 shows the corresponding UV map. Note, that the UV coordinates of the holes (in this case, the triangular face in the corner) are corrected in step 6 (see chapter 6.6, sub-step 2).

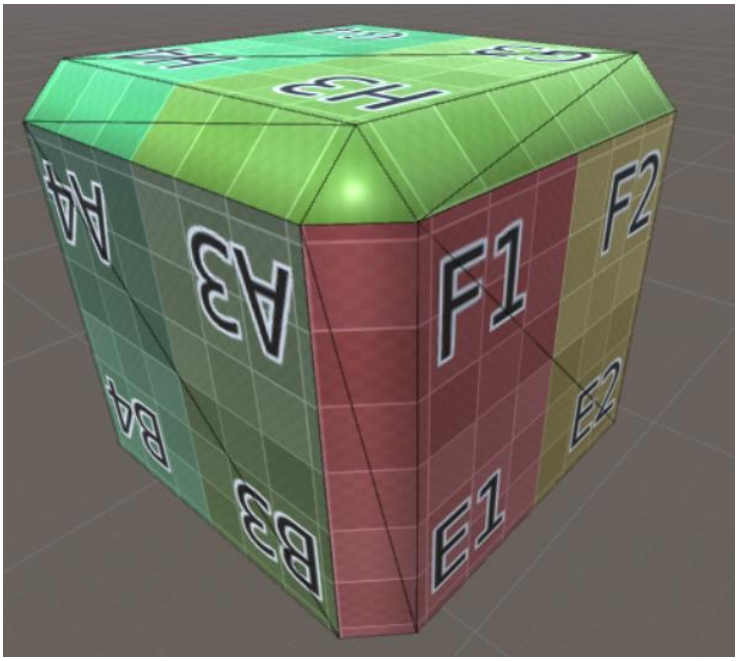

Figure 64. Seamed cube from Figure 61 after chamfering with seams recreated.

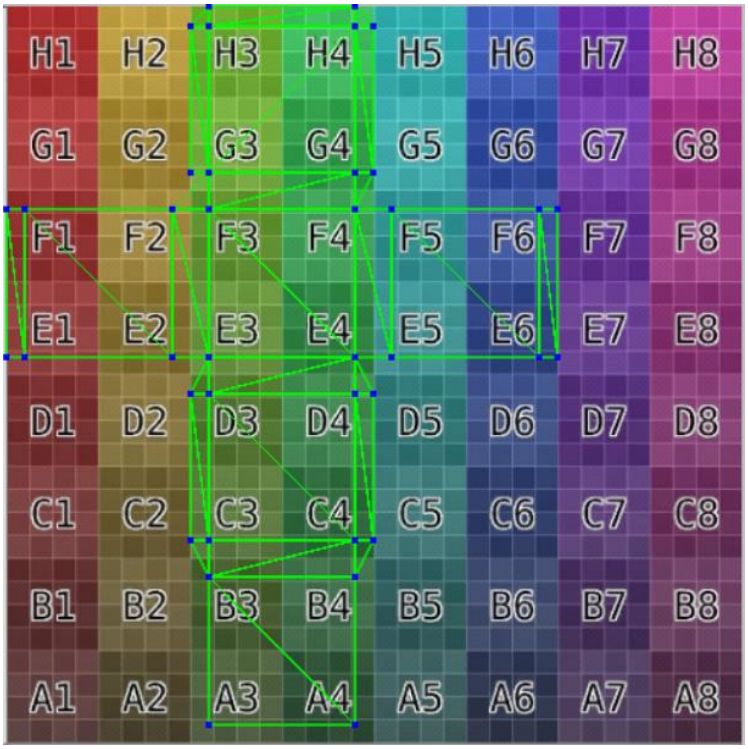

Figure 65. UV map of the cube in Figure 64.

6.4.1 Cases to consider

Not every time can the seam be recreated using exactly the workflow presented above. Since vertices are shared by at least two edges, the vertex can already be processed, and it might change if and which vertex should be duplicated. There are three main cases with different solutions for edges with previously processed vertices if there are no intersections with other hard edges that are seams.

The first case would be where either one or both vertices are already duplicated in the correct way. For example, if A_2 and B_2 have already been processed by duplicating B_2 (Figure 66 a), then sub-steps 1-3 will be skipped for A_2, and the previously created C_2 will be used together with the newly created C_1 for sub-step 4.

The second case would be if either one or both vertices are duplicated using the other vertices. For example, if instead of B_2 like in the first case, A_2 was duplicated (Figure 66 b), then also A_1 will have to be duplicated so the whole edge would be duplicated instead of one vertex from the first, and the other vertex from the second vertex.

The third case is when both vertices are processed, but with one duplicating A, and the other duplicating B (Figure 66 c), then one new vertex C is needed to duplicate the whole edge.

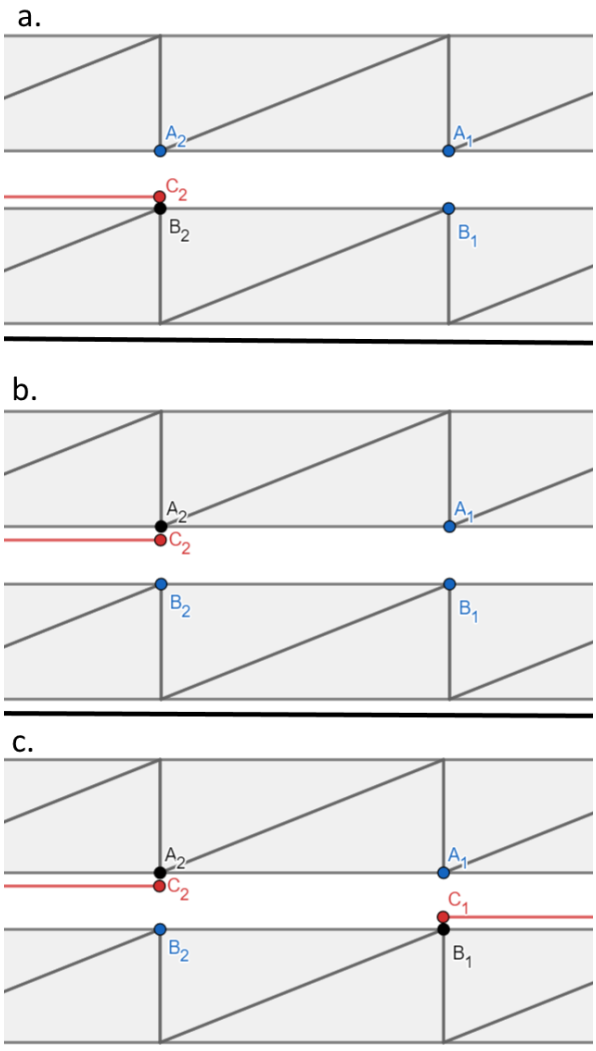

Figure 66. Three cases where one or more vertices are already duplicated. Vertex B_2 is duplicated (case a), vertex A_2 is duplicated (case b), or vertices A_2 and B_1 are duplicated (case c).

In places where multiple hard edges meet, that are also seams, it can happen, that one vertex is already duplicated, but the UV coordinates are gotten from some other vertex belonging to some other edge. For example, B_1 in Figure 67 is already duplicated, but the UV coordinates are not calculated using A_1 but using the vertex A_x that belongs to another edge. In this case, B_1 will have to be duplicated again without changing the existing duplicate.

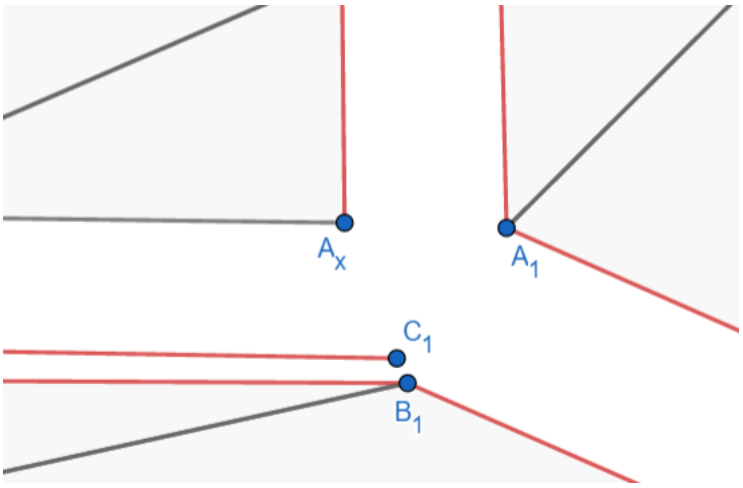

Figure 67. The intersection of three hard edges that are seams, where B_1 is already duplicated but with using the UV coordinates of A_x. For the pair B_1 and A_1, a new vertex C has to be created.

6.5 Creating faces between hard edges

When all duplicate vertices are pulled apart, the spaces created have to be turned into faces so the resulting mesh would have no holes in it. First, faces are created between hard edges. Figure 68 shows a cube just before faces are created between hard edges on the left, and after on the right.

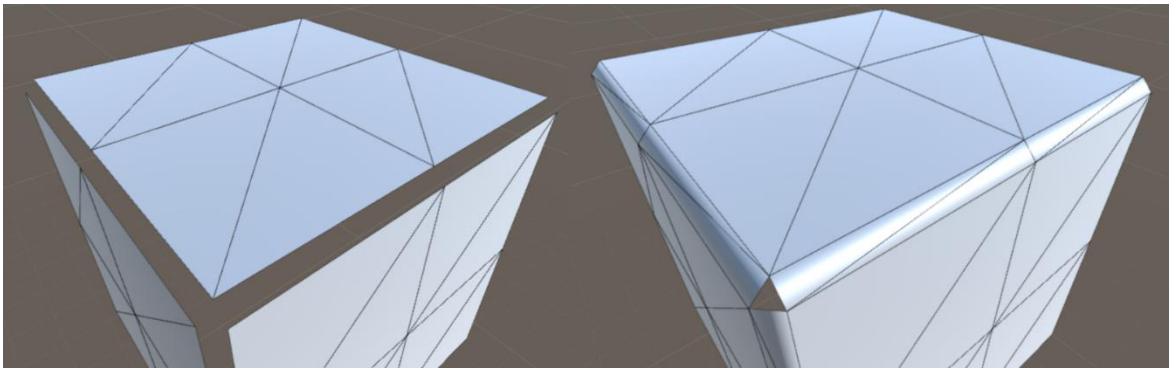

Figure 68. Cube before creating faces between hard edges (left), and after (right).

This step consists of three sub-steps:

1. Finding pairs of overlapping hard edges.
2. Creating new triangles.
3. Saving new edges for clusters.

Sub-step 1. Using the list of hard edges, and the edge map to find the overlapping edge, pairs of edges are found that have to be connected with a face. Overlapping edges can be codirectional, or they can have different directions. In well-made 3D models, the overlapping edges always have different directions. Overlapping hard edges could be codirectional if, for example, one face was flipped to be facing the other way or more than two faces are

connected to the same edge. Figure 69 compares hard edges with same and different directions. A hard edge and its overlapping edge have the same direction if the starting vertices of both are from the same cluster. If two overlapping edges have the same direction, they are not addressed, and will not get a chamfer between them. When hard edges have the same direction, they cannot be connected with a new face in such a way that the new face and the existing faces connected to the edges would be facing the same way.

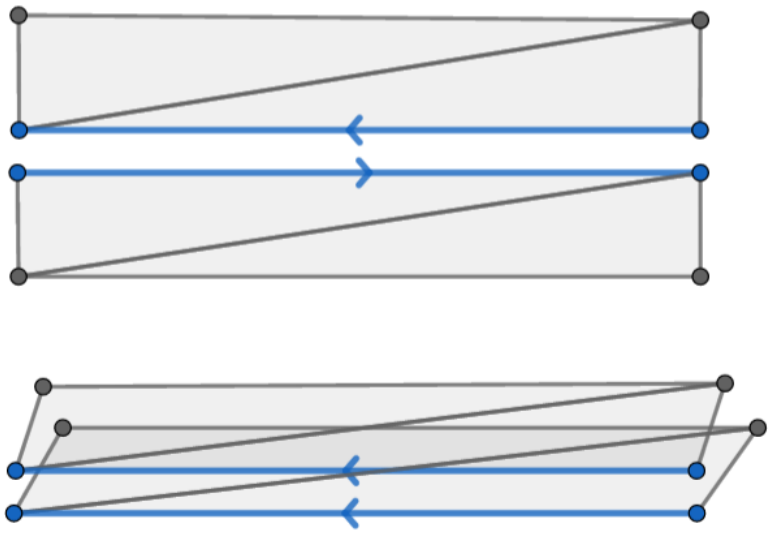

Figure 69. Hard edges (colored blue) with different directions (top), and same direction (bottom).

Sub-step 2. Two triangles are added to the triangle list to form a rectangular face between the edges. For two previously overlapping edges, such as AB and A'B' (Figure 70), the triangles created are AA'B and AB'A'.

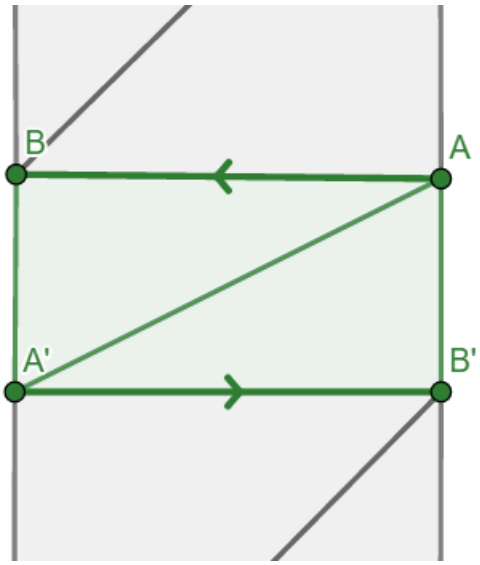

Figure 70. Adding two triangles between two edges. The previously overlapping hard edges are AB and A'B'. New triangles are colored green.

Sub-step 3. At vertices that had more than one overlap, like the corner of a cube (Figure 71), holes have remained. Edges of the holes are saved for every cluster. Clusters are defined in chapter 6.1 as groups of vertices in the same position. If vertices in a cluster have hard edges connected to them, they will be pulled apart. If there are more than two vertices in the

cluster with hard edges connected to them, then pulling them apart will leave a hole at the position of the cluster. In Figure 71, vertices A, B, and C belong to one cluster.

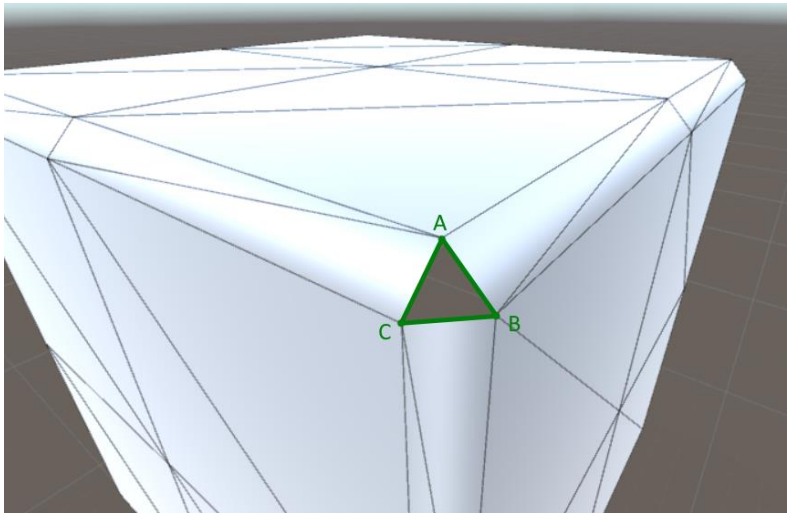

Figure 71. Hole remaining at the corner of a cube after faces have been created between hard edges. A, B, and C are the vertices that the hole is in-between.

The edges that border the hole (BA, CB, and AC in Figure 71) are created during sub-step 2 when the triangles are added between hard edges. Then the points they are created with (B and A', and A and B' in Figure 70) are checked for additional overlaps. If an edge of a created triangle not overlapping with a hard edge (for example AC in Figure 71) has other overlapping vertices besides each other (B in this example), there will be a hole, and edge CA is saved for this cluster. Note, that this will be the orientation of the edge of the triangle filling the hole, not the triangle between hard edges.

6.6 Filling holes where multiple hard edges met

When filling the holes, each cluster is checked. If there are more than two overlapping vertices, a hole might be present. It could also happen, that the additional vertices are caused by broken geometry, or there could be multiple holes instead of one. New triangles will be created based on how many overlapping vertices are in the cluster that belong to one hole.

The process of filling the holes consists of the following sub-steps:

1. Finding edge cycles for every cluster.
2. Creating new triangles
    a. in case of 3 edges,
    b. in case of 4 edges,
    c. in case of more than 4 edges.

Sub-step 1. The beforehand saved edges are now sorted so they form a cycle. For the cluster in Figure 71 the cycle could be CA→AB→BC. For every edge in the cycle, the first vertex is the same as the last vertex of the previous edge, and the last vertex is the same as the first vertex of the succeeding edge.

If the edges of the cluster do not form a cycle, then there might be some faults in the mesh topology. This means that there are some triangles, edges or vertices, that do not seem correct. For instance, some triangles might not even be connected to the rest of the mesh, or they could be connected by only one edge (Figure 72). Broken topology can be caused by an error made by the modeler, or as a result of exporting the mesh. This might cause the

edges saved for a cluster not to form a cycle. Cycles with at least three edges are connected with faces. Other edges not part of a cycle are ignored.

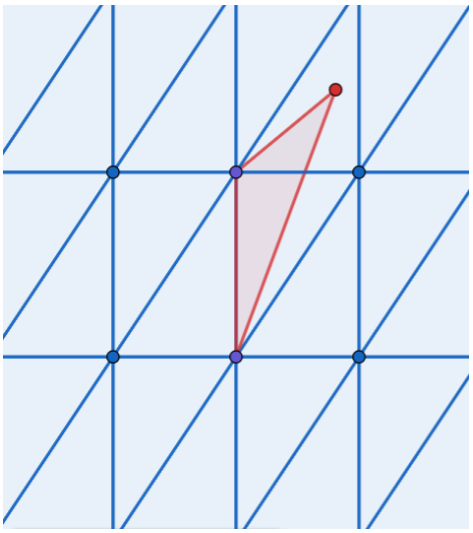

Figure 72. A red triangle connected to the rest of the mesh (blue) by only one edge.

Sub-step 2. Next, the cycles are used to create new faces depending on how many edges are in the cycle. The number of edges in a cycle is equal to the number of vertices in the cycle.

If the vertices of the mesh have UV coordinates, then the new triangles should be mapped using vertices that are close together on the UV map for the texture to be mapped from the appropriate place. If the hole is not on a seam, no additional changes are needed. If the hole lies on a seam, there is a chance that instead of the duplicated vertex, the original has to be used to create a triangle or triangles. A combination of the vertices and duplicates has to be found, that create the triangle(s) in the right place. If no such combination can be found, new vertices have to be created. The creation of triangles considering the UV coordinates is implemented for sub-step 2.a, which is the most common occurrence of holes on seams. The same for sub-step 2.b and 2.c will be implemented as future work. The current implementation of sub-step 2.b and 2.c assumes that the vertices do not have UV coordinates, and will not look for different combinations.

Sub-step 2.a. If three vertices are pulled apart, for example in the corner of a cube, one triangular face between these vertices covers the hole perfectly (Figure 73).

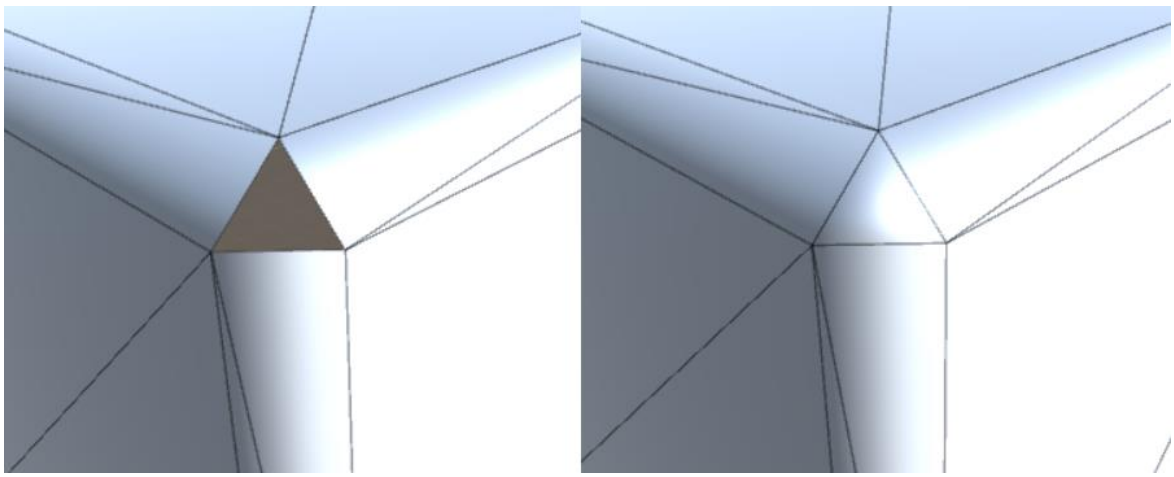

Figure 73. A triangular hole left after step 5 in the corner of a cube before (left) and after filling (right).

Sub-step 2.b. If there are four vertices pulled apart, two triangle faces are created (Figure 74). The new edge connects two of the four vertices that are closer together. Otherwise, a slope might form when the triangles are created (Figure 75).

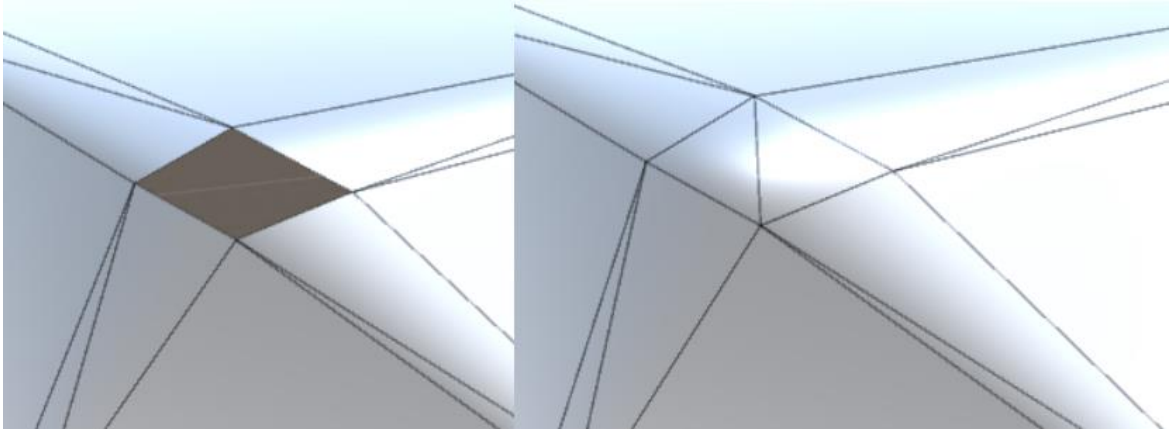

Figure 74. A hole with 4 vertices before (left) and after filling (right).

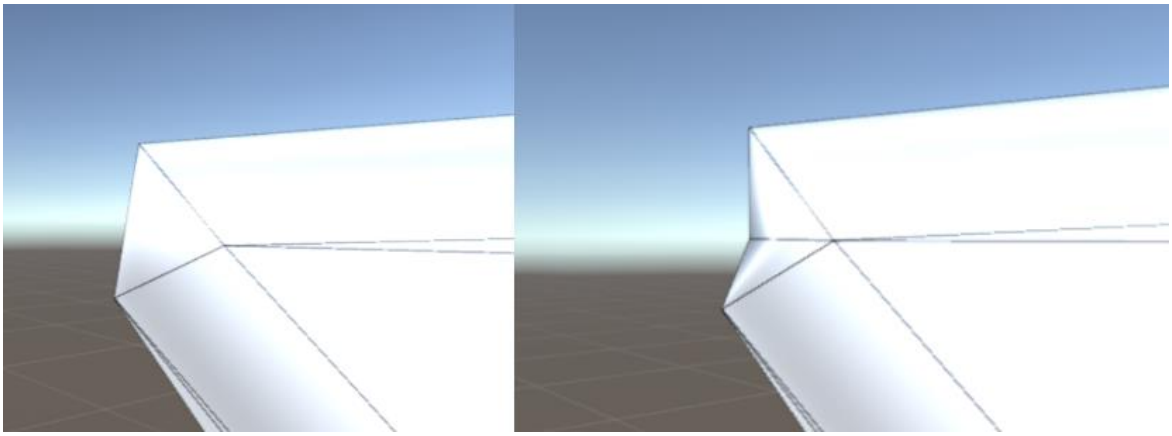

Figure 75. Side view of a hole with 4 vertices without a slope (left), and with a slope (right).

Sub-step 2.c. If there are more than 4 vertices that have been pulled apart, a new center point is created for the result to seem natural. The center point has the average position and normal of all overlapping vertices in that cycle. Using the edges of the hole, and the new center point, triangular faces will be added. Figure 76 compares holes with 6 vertices before and after filling.

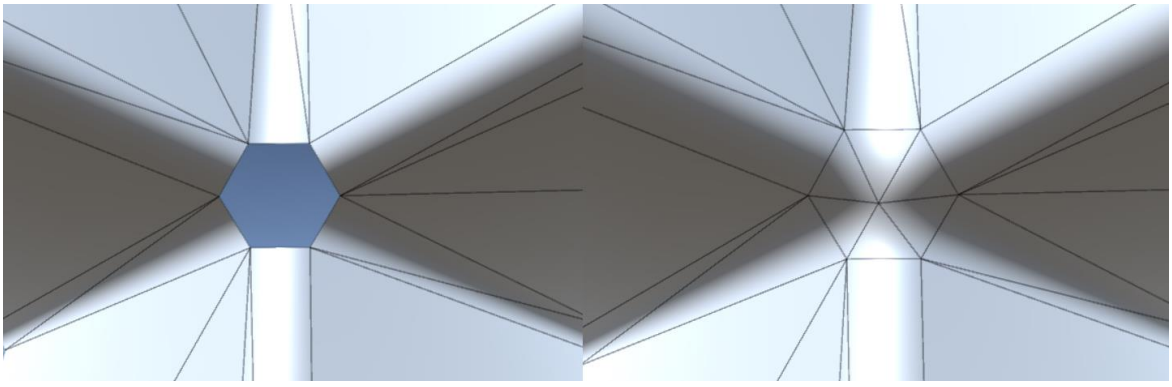

Figure 76. A hole with 6 vertices before (left) and after filling (right).

The new vertex in the center is needed for the shading to look nice. If the triangles were created using only the existing vertices, the new faces would look weird and not as uniformly smooth when shaded. Figure 77 shows a hole with 6 vertices that was filled without creating a new vertex in the center.

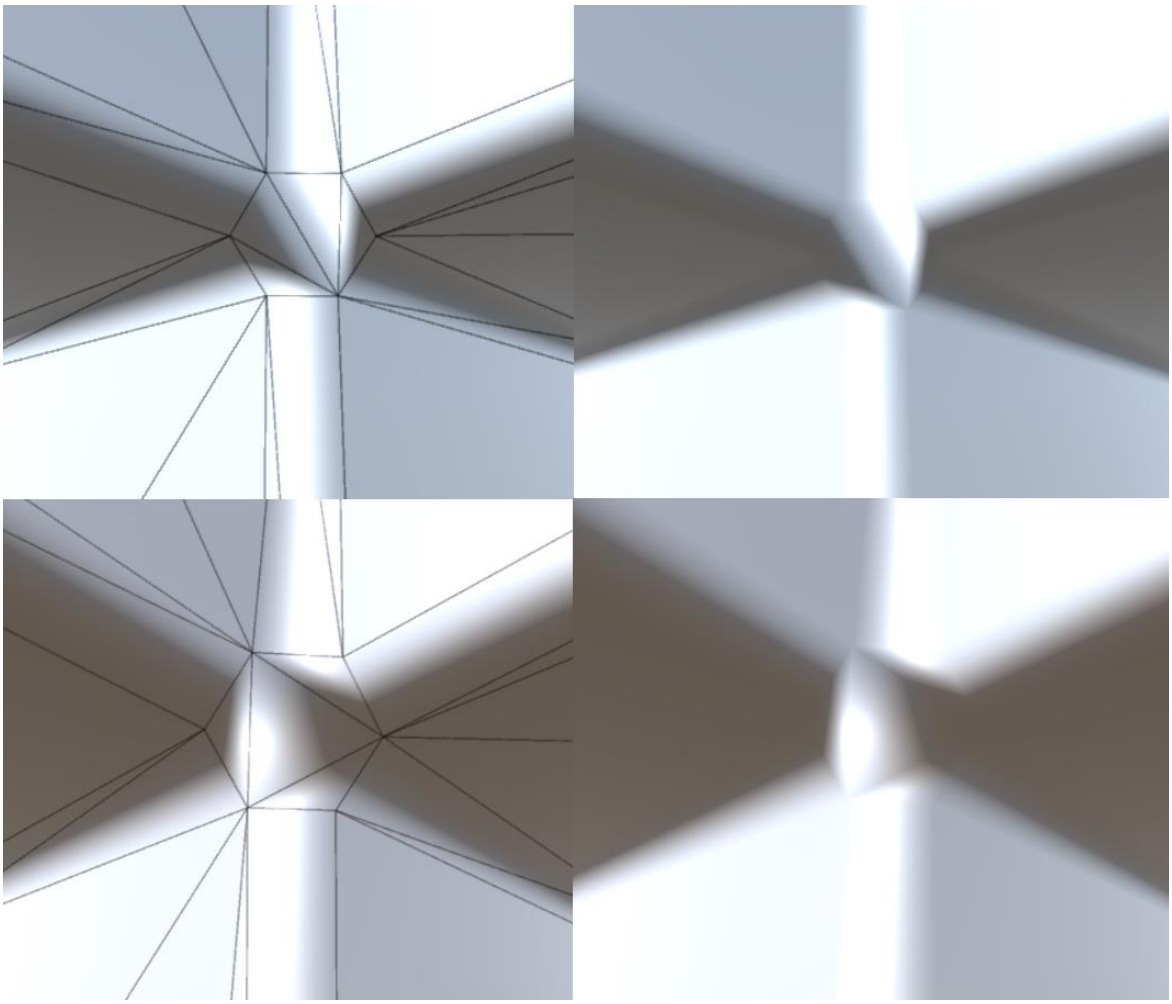

Figure 77. Two ways of filling a hole with 6 vertices without creating a center vertex. Edges are shown on the left but the right side shows only the shading.

7 Resulting product

The practical part of this thesis provides a new Unity asset (Appendix II) that can be used to chamfer any kind of model prefab. The asset will consist of two main parts. The first part is a script that is attached to a new object and it finds all the meshes of the model to be chamfered and enables chamfering of the model with the given chamfer scale. This script is shown in the inspector window as a component. The second part is a script that uses the chamfering algorithm described in the thesis. This takes care of the hard edge detection and chamfering. The asset also includes some additional scripts that support the two main scripts.

Some models shown and compared in this chapter, and chapter 7.2, namely Bolts, Block, Droid, Drone, and Ship, are created by Jaanus Jaggo. The Ship was modeled as a replica of the Scifi ship model by Tor Frick [76]. Below (Figure 78, Figure 79, and Figure 80) are some examples of models before and after chamfering.

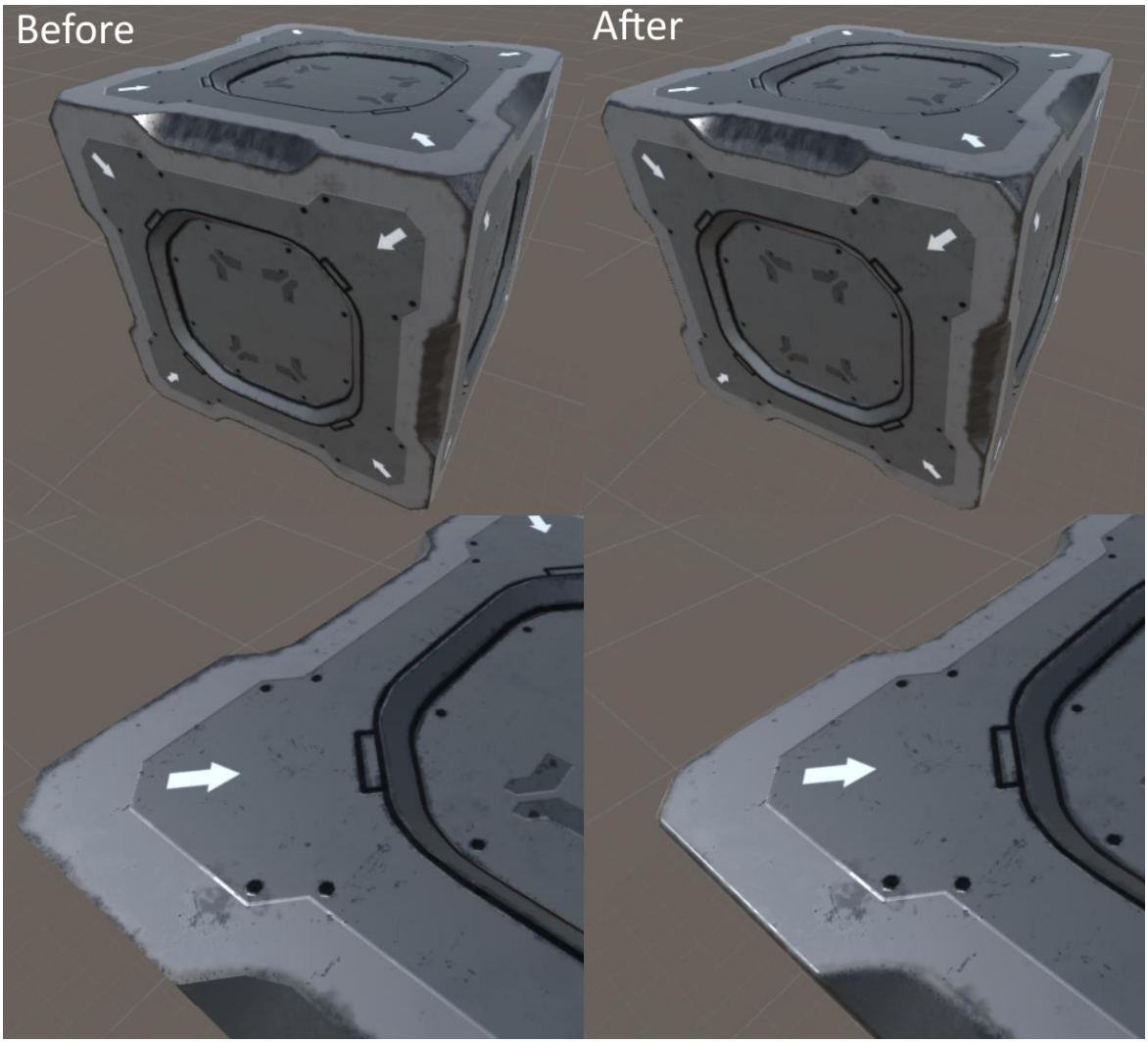

Figure 78. Block before (left) and after (right) chamfering.

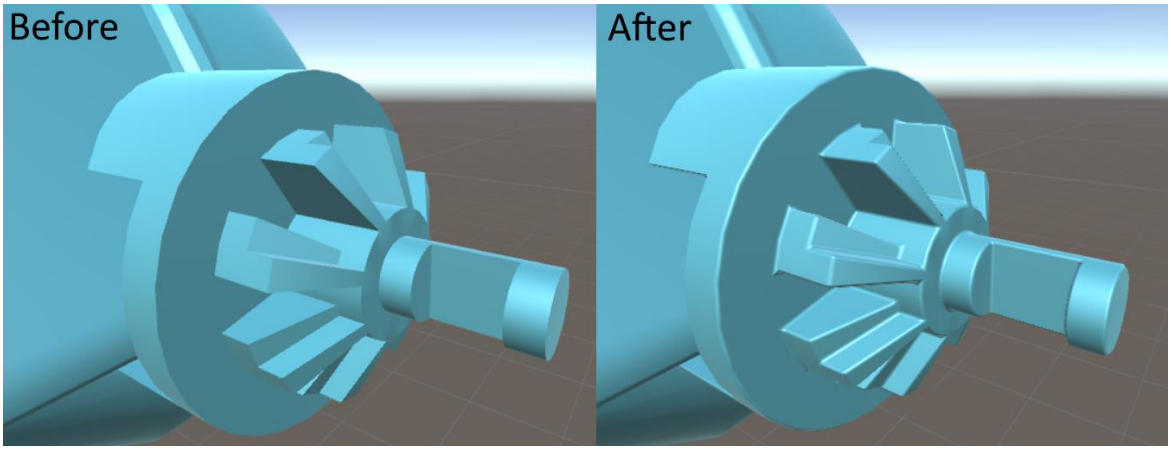

Figure 79. Part of the Ship model before (left) and after (right) chamfering.

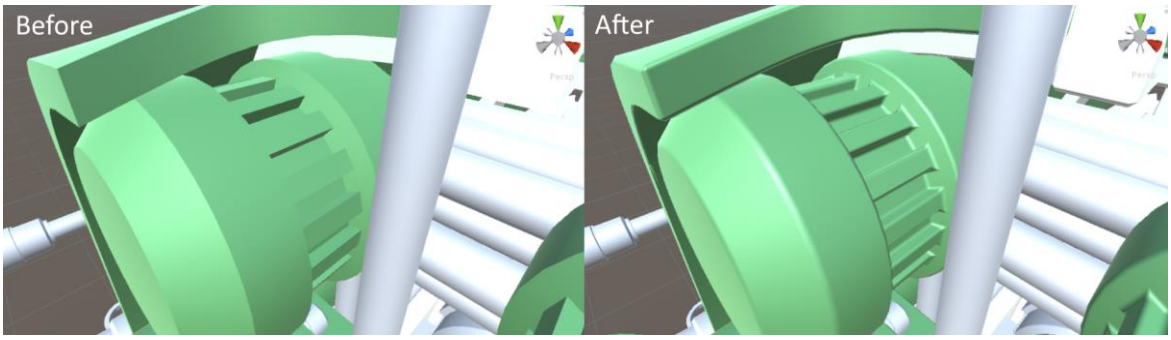

Figure 80. Part of the Drone model before (left) and after (right) chamfering.

Chapter 7.1 describes the process of using the created edge chamfering asset. Chapter 7.2 compares the vertex and triangle counts of different models before and after chamfering.

7.1 Chamfering models using the edge chamfering asset

To use the product, the user must have the Unity asset (Appendix II) installed, and a model prefab has to be present in the project that will be chamfered. The asset can be installed from the menu bar by choosing 'Assets' -> 'Import Package' -> 'Custom Package…', and selecting the Chamferer asset.

The following steps are used to chamfer a model:

1. In the Assets folder (or any subfolder) right click, and choose 'Create' -> 'Chamfered Object'.
2. In the opened window, choose the directory and name for the new model prefab.
3. Select the created prefab in the Asset folder.
4. In the component list, under 'Chamferer (Script)', select the 'Mesh' to be chamfered.
5. Specify 'Chamfer scale'.
6. Press 'Apply'.

After pressing 'Apply', new chamfered model prefab of the selected mesh is created in the folder that can be added into the scene. The original model that was chosen to be chamfered remains unchanged since the chamfered version is created as a new prefab.

Step 1. To create a new object that will become the chamfered object, a custom option 'Chamfered Object' is used that is shown when right-clicking in the project window in an empty space in some folder and choosing 'Create' (Figure 81).

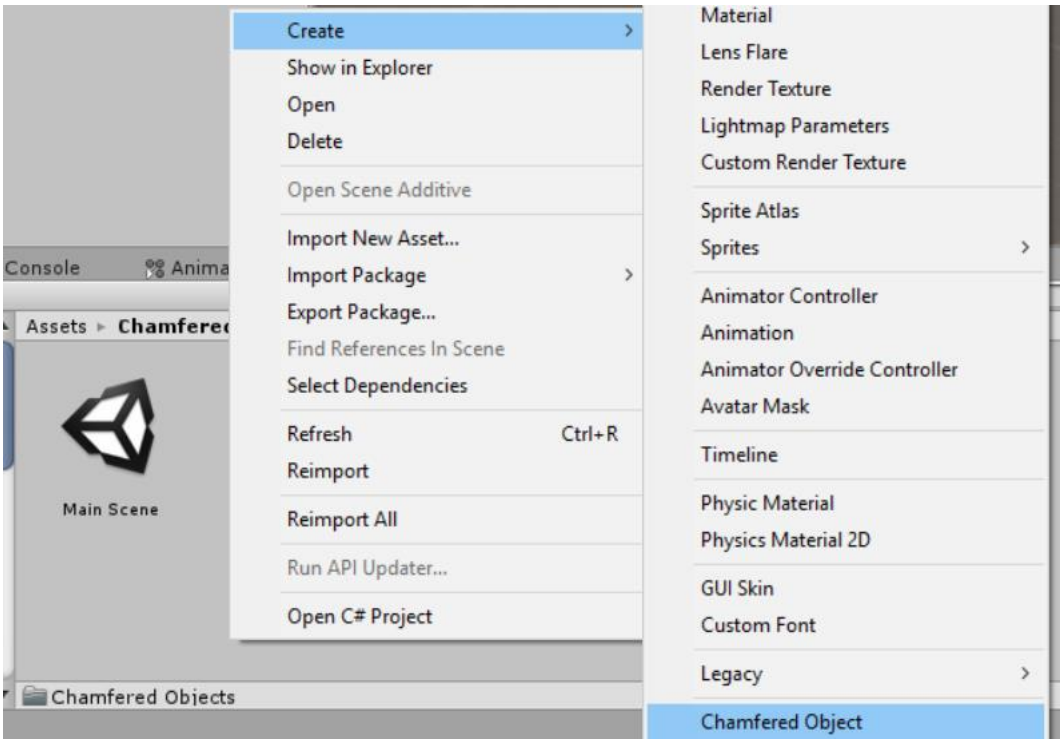

Figure 81. Creating a new chamfered object in the Unity Editor.

Step 2. When a new Chamfered Object is selected from the 'Create' menu, a window opens where the user can select the name and location of the new object (Figure 82). After clicking 'Save', the object is created.

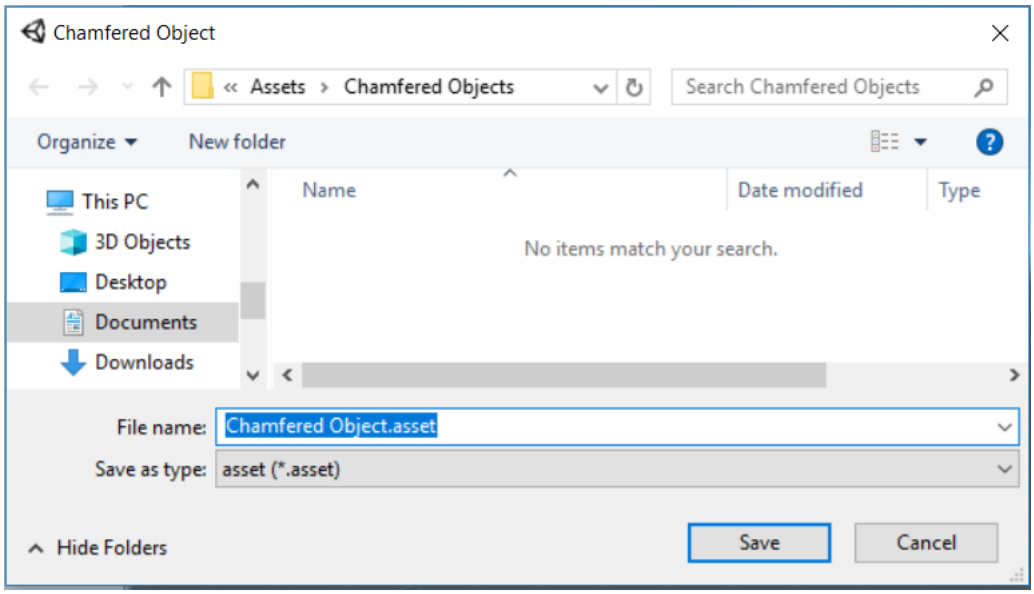

Figure 82. The window for creating the chamfered object.

Step 3. To see the Chamferer component, the created object has to be selected (Figure 83). It is located in the folder chosen in step 2.

Step 4. The inspector view will now show the components of the created object. Under the Chamferer (Script) component, there is a Mesh field (Figure 84). To add a model to the script, either the model should be dragged into the Mesh field, or it can be chosen by clicking on the circle next to the field, and selecting the model from the list that opens.

Figure 83. The created Chamfered Object when selected in the project folder.

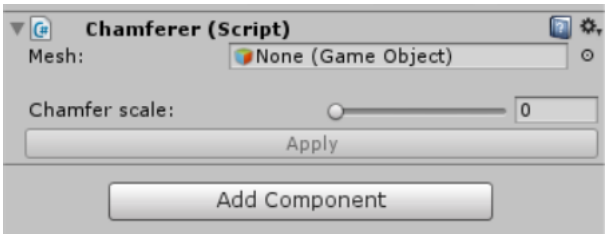

Figure 84. Chamferer script component before mesh or chamfer scale is selected.

If a model is selected, the component displays the list of its sub-meshes with their vertex counts (Figure 85).

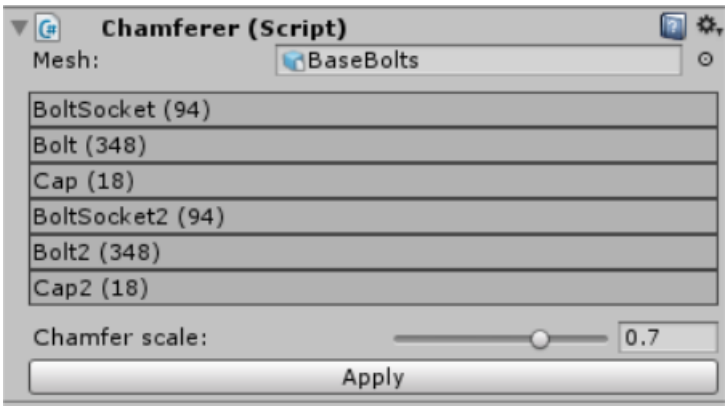

Figure 85. Chamferer script component with a model selected showing the list of its sub-meshes.

Step 5. The chamfer scale can be selected by either dragging the slider or entering the value in the field next to the slider.

Step 6. Pressing 'Apply' turns the object into the chamfered variant of the chosen model. A new object that stores the modified mesh is also created. Now, the chamfered object can be dragged into the scene. The Chamferer script remains on the object, and it can be used to chamfer the object again with a new chamfer scale, or even using a new mesh.

7.2 Vertex and triangle counts before and after chamfering

The edge chamfering algorithm created in this thesis tries to create a minimal amount of new vertices. If the model has not been unwrapped, then a new vertex is created only in places where more than three hard edges meet, which does not happen very often. If the model is unwrapped before seams are added, then the polygons might be tiled on top of each other on the UV map. This might default to almost every edge being a seam, i.e. having

multiple vertices in place of one for storing multiple UV coordinates. This could result in most or all of the hard edges being seams that have to be recreated during chamfering, meaning that more vertices are added. If a model has been seamed and then unwrapped, the seams are limited to those that were marked, and not that many vertices are doubled. Thus, the vertex count of the seamed and unwrapped model is usually smaller than of the same model that is unwrapped without marking seams. This applies to both not chamfered and chamfered models.

Since unwrapping, and adding seams affects only the vertices, the triangle count should remain the same when seams are created, or the model is unwrapped. Similarly, the amount of hard edges does not change, because creating seams duplicates the vertices using the same normal. Hence, the amount of triangles created for chamfers does not change with creating or changing the UV map of the original model.

When chamfering, the triangle count will increase by two for every hard edge, and additional triangles are created where more than two hard edges intersect. For an intersection of three edges, one triangle is created, for an intersection of four edges, two triangles are created, and for an intersection of *n* edges ($n > 4$), *n* triangles are created.

In the following tests, various models were chamfered, and their vertex and triangle counts were observed before and after chamfering. Changes in vertex and triangle counts are defined as $\frac{Count\ After}{Count\ Before}$. If the count remains the same, the change equals 1. If the count is doubled, the change equals 2. Models that are unwrapped with no seams are noted with 'UW'. Models that are unwrapped with seams are noted with 'S&UW'. Below are images of the models, and their UV maps where applicable (Figure 86, Figure 88, Figure 90), tables with their respective vertex and triangle counts with changes (Table 3, Table 4, Table 5), and bar charts of their respective vertex and triangle counts (Figure 87, Figure 89, Figure 91, Figure 92). For the Cylinder model (Figure 88), it might seem like it has nice seams by default (UW) without creating them manually since making only the rounded side smooth-shaded creates doubled vertices where the top and bottom part meet the smooth side. However, the rounded side is not cut in half, and thus has to be unwrapped so one edge that connects to either top or bottom part is in the center and significantly smaller than the other edge that is unwrapped as the outer circle.

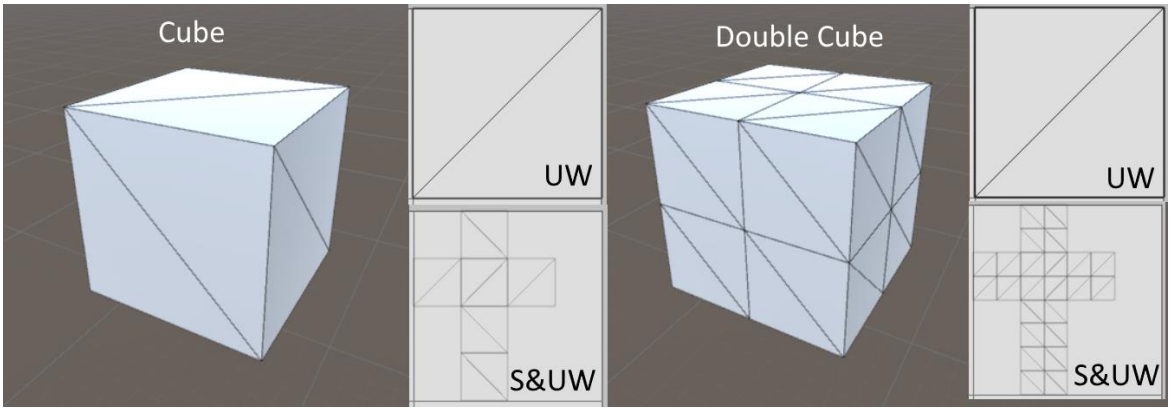

Figure 86. Models Cube (left) and Double Cube (right) with their respective UV maps.

	Model	*Cube*	*Cube UW*	*Cube S&UW*	*Double Cube*	*Double Cube UW*	*Double Cube S&UW*
Vertices	**Before**	24	24	24	54	96	54
	After	24	48	38	54	144	75
	Change	1	2	1.58	1	1.5	1.39
Triangles	**Before**	12	12	12	48	48	48
	After	44	44	44	104	104	104
	Change	3.67	3.67	3.67	2.17	2.17	2.17

Table 3. Changes in vertex and triangle counts before and after chamfering for models Cube and Double Cube.

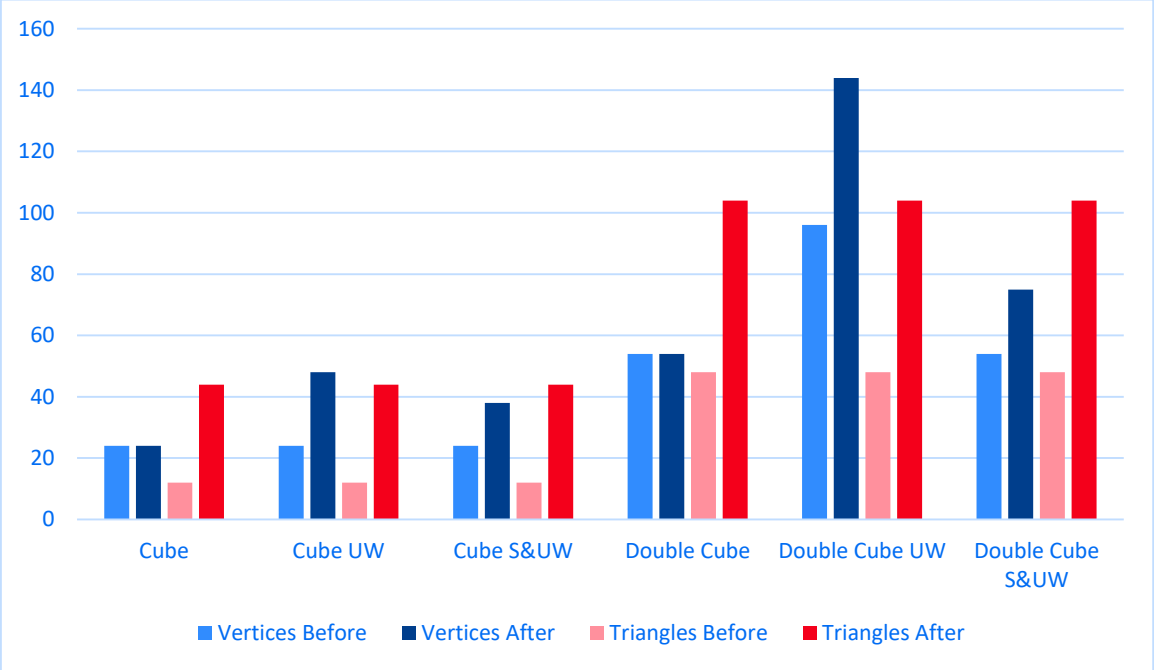

Figure 87. Vertex and triangle counts before and after chamfering for models Cube and Double Cube.

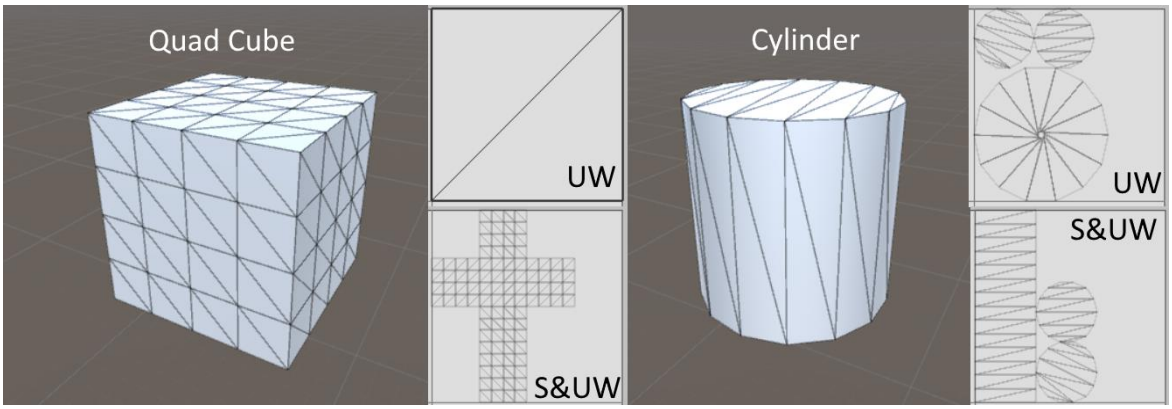

Figure 88. Models Quad Cube (left) and Cylinder (right) with their respective UV maps.

	Model	*Quad Cube*	*Quad Cube UW*	*Quad Cube S&UW*	*Cylinder*	*Cylinder UW*	*Cylinder S&UW*
Vertices	**Before**	150	384	150	56	56	58
	After	150	480	185	56	112	114
	Change	1	1.25	1.23	1	2	1.97
Triangles	**Before**	192	192	192	52	52	52
	After	296	296	296	108	108	108
	Change	1.54	1.54	1.54	2.08	2.08	2.08

Table 4. Changes in vertex and triangle counts before and after chamfering for models Quad Cube and Cylinder.

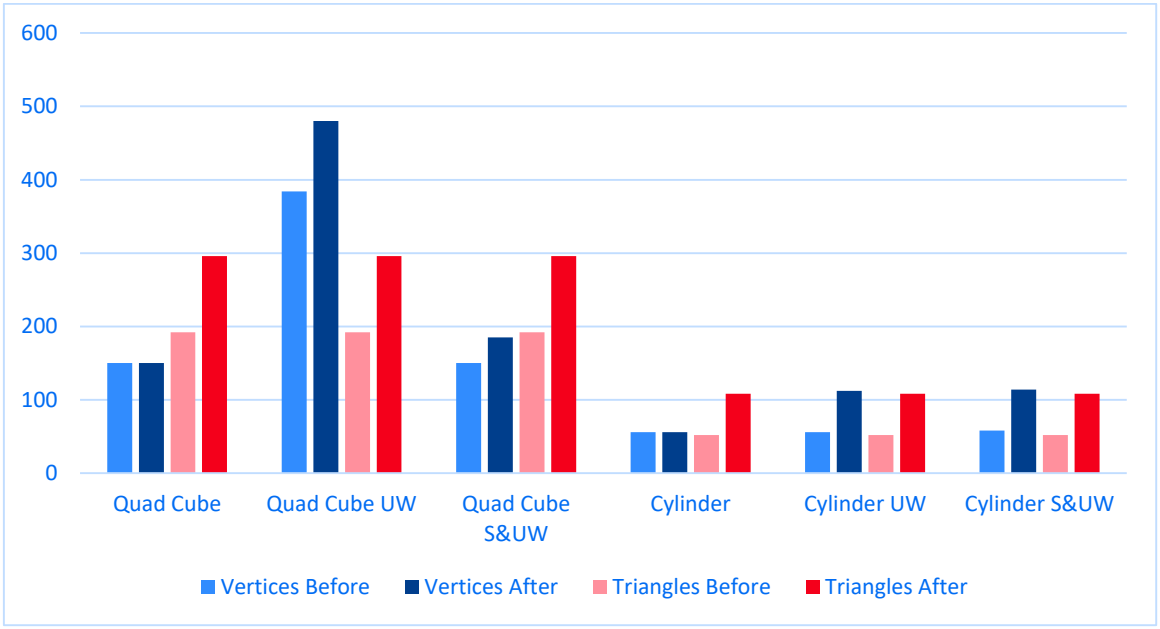

Figure 89. Vertex and triangle counts before and after chamfering for models Quad Cube and Cylinder.

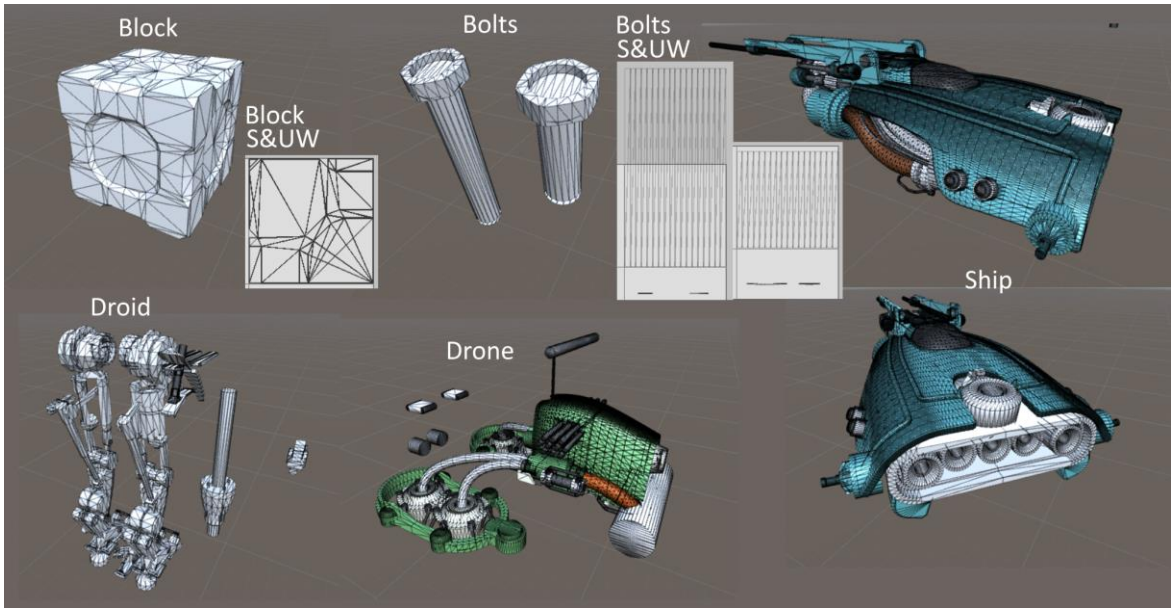

Figure 90. Models Block (top left), Bolts (top middle), Droid (Bottom left), Drone (bottom middle), and Ship (right). UV maps of Block and Bolts are next to their respective models.

	Model	***Block S&UW***	***Bolts S&UW***	***Droid***	***Drone***	***Ship***
Vertices	**Before**	1080	920	18525	73308	56668
	After	1352	1030	18676	75080	60616
	Change	1.25	1.12	1.01	1.02	1.07
Triangles	**Before**	824	872	11593	84738	71070
	After	1736	1736	36052	141786	114418
	Change	2.11	1.99	3.11	1.67	1.61

Table 5. Changes in vertex and triangle counts before and after chamfering for models Block, Bolts, Droid, Drone, and Ship.

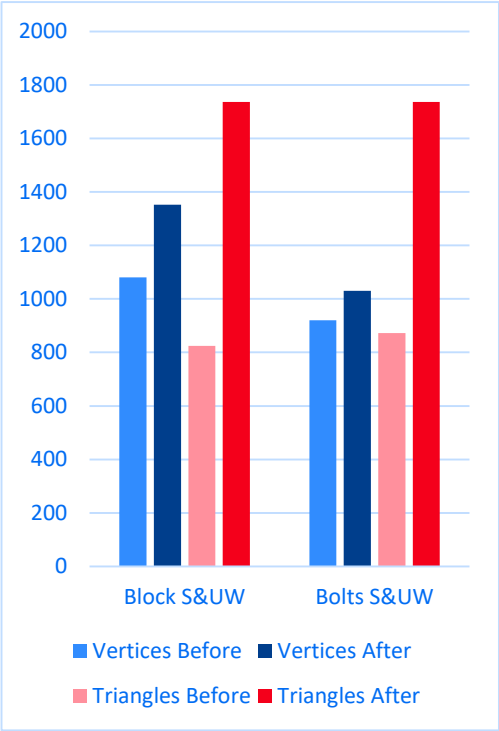

Figure 91. Vertex and triangle counts before and after chamfering for models Block and Bolts.

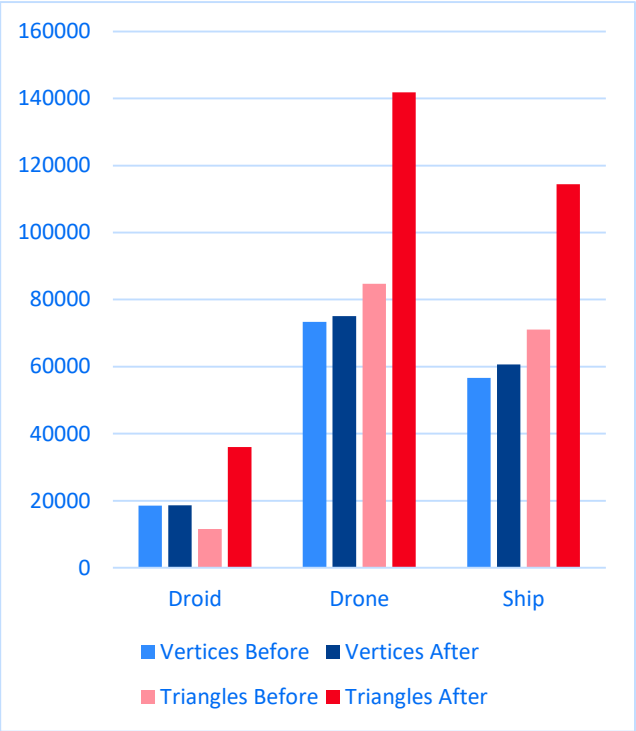

Figure 92. Vertex and triangle counts before and after chamfering for models Droid, Drone, and Ship.

When chamfered, the vertex count of models with no UV map does not change, or changes very little (1.07 for the Ship model). For models with UV maps, depending on the number of seams that overlap with hard edges, the vertex count could remain similar to the original (1.12 change for Bolts), or it could even double (change 2 for Cube UW, and Cylinder UW). No vertex count changes over 2 were observed, but it is theoretically possible if all or most of the edges are hard edges and also seams, and multiple intersections with more than two of these edges occur.

Triangle count changes varied from 1.54 for Quad Cube to 3.67 for Cube. The number of triangles created for the Cube was 32: two triangles for every hard edge, and one triangle for every corner; $12 * 2 + 8 * 1 = 32$. The change in triangle count depends mainly on the ratio of hard edges to all edges, the number of intersections, and the number of edges that intersect in each intersection. If all edges of the Cube were hard edges, the change would be 5.33, because there would be six more hard edges, and the eight intersections would turn into four intersections with four edges, and four intersections with five edges.

$$\frac{(12 + 6) * 2 + 4 * 2 + 4 * 5}{12} = 5.33$$

In conclusion, the change of the triangle count for most models should usually not exceed 4 and would exceed 5 only in extreme cases.

8 Future work

While the algorithm created in this thesis works well, the created asset can be improved further. Particularly, the ease of use of the asset, increased control over chamfering, and the part of the algorithm that deals with UV seams at holes. For further improvement, a separate tool independent of Unity can be created for chamfering models.

8.1 Ease of use

Although the asset remarkably accelerates the process of chamfering models compared to the process of chamfering them manually using some modeling software, it can be made even more user-friendly. The current flow of chamfering a model using the asset is not very intuitive, or particularly convenient. Steps that have to be taken are very specific (see chapter 7.1), and can hardly be figured out without something to disclose the steps.

Additionally, when a chamfered object is dragged into the scene, it cannot be chamfered again from the Chamferer component of the object in the scene. Instead, the prefab has to be selected and chamfered. The object that was in the scene will lose its mesh filter, and chamfer attributes. It essentially becomes a broken, or an empty object. It then has to be replaced. A better approach would be allowing the chamfering of an object in the scene and automatically creating a new prefab for it. The new prefab could be created by opening a window for the user to choose the location and name while giving an option to replace an existing prefab with the new one.

8.2 Control over chamfering

Current implementation enables assigning only one chamfer scale for the whole model. To chamfer only one sub-mesh, or assign different chamfer scales for different sub-meshes, the sub-meshes would have to be made into separate prefabs. Then, they can be chamfered with a different chamfer scale. It would be much more convenient to be able to assign different chamfer scales for sub-meshes, while also being able to set one chamfer scale for the whole model.

The current chamfer scale could be made into a master chamfer scale, which would be the default for all sub-meshes. Individual sub-meshes would have a chamfer scale of their own, which would override the master chamfer scale when changed. Furthermore, the current listing of sub-meshes under the Chamferer script component would also have to reflect the hierarchy. This way, a chamfer scale set for a sub-mesh, instead of the master chamfer scale, could be the default for its children. If the component would continue to show the sub-meshes in a list with no hierarchy, it would not be clear to the user, which chamfer scales would affect others.

8.3 UV seams

A part of the chamfering algorithm that deals with UV coordinates is not implemented as of yet, namely the correct UV mapping of holes with more than three vertices that lie on seams (see chapter 6.6, sub-step 2).

For holes with four vertices, two triangles are added. They would either be both mapped as a single group in one place on the UV map, or one triangle would be in one group, and the second triangle in another group.

For holes with more than four vertices, the created middle vertex would be on a seam. The triangles created would have to be grouped according to where they should be placed on the UV map.

For the most part, the correct vertices would be found from the displacements. Occasionally, new vertices would have to be created. Using the logic of creating displacements (see chapter 6.4).

8.4 Chamfering tool independent of Unity

The Unity asset can be used only in Unity with a project that includes the models. For additional freedom and accessibility, a separate tool could be created that is independent of Unity. The tool can be created in Unity, but it would be used without it. Therefore, instead of an asset, it would be a separate program. The user could provide a model file, and chamfer scale(s) as input, and get the file of the chamfered model back.

9 Conclusions

Chamfering is the process of smoothing sharp edges, also known as hard edges. Very often it is done by modelers who do it in their modeling software using the built-in tools. This allows them to have good control over the process, and get the look they want. For the less experienced modelers, or for just enthusiasts who want to make a game, and to make the models look more natural, asset created in this thesis will be of help.

The main concentration of this thesis was creating the edge chamfering algorithm that can automatically chamfer any mesh while leaving the edges smoothly shaded. The algorithm creates a minimal amount of extra geometry so the chamfered models would not hinder performance at run-time while having edges that are a lot smoother.

The algorithm uses existing vertices on hard edges for creating most of the new faces. Hard edges have two pairs of vertices with different normals that create the hard edge when shaded. During edge chamfering, these vertices are moved, and chamfers are created between hard edges. The normals of the vertices on hard edges are already oriented in a way that the added chamfers are automatically smoothly shaded, so there is no need for modifying vertex normals to get a smooth edge.

For meshes that have no UV coordinates, hardly any new vertices are created. Meshes that have UV coordinates usually have either manually added seams or seams that are created automatically. During edge chamfering, these seams are essentially removed, because the edges are connected with chamfers. For the texture to look right, the seams have to be recreated which also creates new vertices. The total amount of vertices added is still a lot less than if subsurface modifiers were used.

A Unity asset was created, that uses the algorithm to automatically smooth the edges of the selected mesh. This asset takes a model, and a chamfer scale for input, and detects different sub-meshes present in the model. Then all the user needs to do is to press the ‘Apply’ button to get the chamfered version of the model. The created asset makes chamfering different models fast and easy to do inside a popular game engine Unity, making it more accessible to game developers. The asset can be further improved in the future to be even more user-friendly and give the user more control over chamfering.

10 References

[1] *Alien: Isolation*. SEGA. www.alienisolation.com/ (16.05.2018)

[2] *Star Citizen | Your First-Person Universe*. Roberts Space Industries, Cloud Imperium Games. robertsspaceindustries.com/star-citizen (16.05.2018)

[3] Sneddon, Mark. "A short explanation about custom vertex normals (tutorial)". *Polycount Forum*, 28 July 2015, polycount.com/discussion/comment/2330935#Comment_2330935 (16.05.2018)

[4] Johns, Matthew Trevelyan. "A short explanation about custom vertex normals (tutorial)". *Polycount Forum*, 28 July 2015, polycount.com/discussion/comment/2330817#Comment_2330817 (16.05.2018)

[5] Silaghi, Marius. "Chamfer the Right Way." *Marius Silaghi's Store. 3D Graphics Tools.* Silaghi, Marius. mariussilaghi.com/products/quad-chamfer (16.05.2018)

[6] *Unity – Multiplatform – Publish your game to over 25 platforms*. Unity Technologies. unity3d.com/unity/features/multiplatform (16.05.2018)

[7] *Ori and the Will of the Wisps*. Microsoft Studios. www.orithegame.com/ (16.05.2018)

[8] Batchelor, James. "Unity Focus: Making Ori and the Blind Forest." *MCV*. Newbay Media, 1 December 2014. www.mcvuk.com/development/unity-focus-making-ori-and-the-blind-forest (16.05.2018)

[9] *SUPERHOT – The FPS where time moves only when you move*. SUPERHOT Team. superhotgame.com/ (16.05.2018)

[10] SUPERHOT Team. "SUPERHOT by SUPERHOT Team – Kickstarter." *Kickstarter*. Kickstarter. www.kickstarter.com/projects/375798653/superhot (16.05.2018)

[11] *Cuphead: Don't Deal With The Devil*. STUDIOMDHR Entertainment. www.cupheadgame.com/ (16.05.2018)

[12] "They bet everything on a game." *Unity*. Unity Technologies. unity.com/madewith/cuphead (16.05.2018)

[13] Mikson, Jens-Stefan. *Virtual Reality Game Design Analysis Based on Tribocalypse VR.* 2017

[14] Ever, Kalle. *Keskkondade vaheliste portaalide algoritm ja selle kasutamine arvutimängus.* 2017.

[15] Parabox, LLC. UVee. *Unity Asset Store*. Unity Technologies. assetstore.unity.com/packages/tools/modeling/uvee-6933 (16.05.2018)

[16] aleksandr. "Documentation, Unity scripting languages and you." *Unity Blog*. Unity Technologies, 3 September 2014. blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/ (16.05.2018)

[17] Fine, Richard. "UnityScript's long ride off into the sunset." *Unity Blog*. Unity Technologies, 11 August 2017. blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/ (16.05.2018)

[18] Rouse, Margaret. "nonuniform rational B-spline (NURBS)." *WhatIs.com*. TechTarget, March 2011. whatis.techtarget.com/definition/nonuniform-rational-B-spline-NURBS (16.05.2018)

[19] Rouse, Margaret. "3D model." *WhatIs.com*. TechTarget, September 2016. whatis.techtarget.com/definition/3D-model (16.05.2018)

[20] "Introduction to Polygon Meshes." *Scratchapixel 2.0*. Scratchapixel. www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh (16.05.2018)

[21] Model Tab. *Unity Manual. Version 2018.1*. Unity Technologies, 30 April 2018. docs.unity3d.com/Manual/FBXImporter-Model.html (16.05.2018)
[22] Anatomy of a Mesh. *Unity Manual. Version 2018.1*. Unity Technologies, 30 April 2018. docs.unity3d.com/Manual/AnatomyofaMesh.html (16.05.2018)
[23] Chrschn. Dolphin Triangle Mesh. 27 March 2007. commons.wikimedia.org/wiki/File:Dolphin_triangle_mesh.png#/media/File:Dolphin_triangle_mesh.png (16.05.2018)
[24] Vector3. *Unity Manual Version 2018.1*. Unity Technologies, 30 April 2018. docs.unity3d.com/ScriptReference/Vector3.html (17.05.2018)
[25] Autodesk.Help. "Object space, world space and tangent space." *Autodesk knowledge network*. Maya. Autodesk Inc, 9 September 2014. knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2015/ENU/Maya/files/Asts-Object-space-world-space-and-tangent-space-htm.html (16.05.2018)
[26] Tangent Space Normal Mapping. *CRYENGINE Technical Documentation*. CRYTEK GmbH, 21 June 2013. docs.cryengine.com/display/SDKDOC4/Tangent+Space+Normal+Mapping (16.05.2018)
[27] "World space normal map and tangent space normal map of a boot." Tangent Space Normal Mapping. *CRYENGINE Technical Documentation*. CRYTEK GmbH, 21 June 2013. docs.cryengine.com/display/SDKDOC4/Tangent+Space+Normal+Mapping (16.05.2018)
[28] Pitt, Ben. "Important information regarding Mesh UV channels." *Unity Forum*, 30 November 2015. forum.unity.com/threads/important-information-regarding-mesh-uv-channels.370746/#post-2402221 (16.05.2018)
[29] "Jurassic World." *Imagine Engine*. Imagine Engine Design Inc. image-engine.com/film/jurassic-world/ (16.05.2018)
[30] Horn, Leslie. "Transformers VFX Guru Explains Why Building CGI Bots Is Getting Harder." *Gizmodo*. Gawker Media, 27 June 2014. gizmodo.com/transformers-vfx-guru-explains-why-building-cgi-bots-is-1597172174 (16.05.2018)
[31] Snider, Burr. "The Toy Story Story." *Wired*. Condé Nast Publications, 1 December 1995.
[32] Visible Body – Virtual Anatomy to See Inside the Human Body. *Visible Body*. www.visiblebody.com/ (16.05.2018)
[33] Skull Screenshot. *Anatomy warehouse*. AnatomyWarehouse.com. www.anatomywarehouse.com/visible-body-human-anatomy-physiology-software-pc-or-mac-download-a-104249 (16.05.2018)
[34] Burger, Thomas. "How Fast is Realtime? Human Perception and Technology." *PubNub*. PubNub Inc, 9 February 2015. www.pubnub.com/blog/how-fast-is-realtime-human-perception-and-technology/ (16.05.2018)
[35] Obscura. "A short explanation about custom vertex normals (tutorial)." *Polycount Forum*, 2 July 2015, polycount.com/discussion/154664/a-short-explanation-about-custom-vertex-normals-tutorial (17.05.2018)
[36] ahcookies. "I'm an tech artist in the industry and I'd love to clear up some misconceptions about rendering and game art workflows that I frequently see in comments from SC fans." *Star Citizen subreddit*, Reddit, 12 October 2015, www.reddit.com/r/starcitizen/comments/3ogi3o/im_an_tech_artist_in_the_industry_and_id_love_to/ (17.05.2018)

[37] Pluralsight. “What’s the difference between hard surface and organic modeling?” *Pluralsight*. Pluralsight LLC, 16 June 2015. www.pluralsight.com/blog/film-games/whats-the-difference-between-hard-surface-and-organic-models (16.05.2018)
[38] “Conference Collection Edge Detail Guide. A1.” *The Conference Collection*. DeskMakers. deskmakers.com/products/tables/conference-tables (17.05.2018)
[39] Johns, Matthew Trevelyan. Star Citizen… The Gladiator: Exterior shots. *Artstation*. Ballistiq Digital Inc. www.artstation.com/artwork/lDN2a (16.05.2018)
[40] Johns, Matthew Trevelyan. “A short explanation about custom vertex normals (tutorial).” *Polycount Forum*, 29 July 2015, polycount.com/discussion/comment/2331313#Comment_2331313 (17.05.2018)
[41] “Introducing Temporal Anti-Aliasing.” *Sketchfab*. Sketchfab, 20 June 2017. blog.sketchfab.com/introducing-temporal-anti-aliasing/ (21.05.2018)
[42] GDC, “Advanced VR Rendering.” *GDC Vault*, Speaker Alex Vlachos from Valve. UBM Tech. www.gdcvault.com/play/1021771/Advanced-VR (21.05.2018)
[43] Provost, Guillaume. “Beautiful, Yet Friendly Part 1: Stop Hitting the Bottleneck.” 2003.
[44] Provost, Guillaume. "Beautiful, Yet Friendly Part 2: Maximizing Efficiency." 2003.
[45] Optimizing Graphics Performance. *Unity Manual. Version 2018.*1. Unity Technologies, 30 April 2018. docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html (17.05.2018)
[46] torgo300. “Performance Art 3: Polygons don’t matter.” *Microsoft TechNet*. Microsoft, 19 June 2007. blogs.technet.microsoft.com/torgo3000/2007/06/19/performance-art-3-polygons-dont-matter/ (17.05.2018)
[47] equil. “3D model polygon count… How much is optimal?” *Unity Forum*, 19 July 2010, forum.unity.com/threads/3d-model-polygon-count-how-much-is-optimal.54832/#post-349293 (17.05.2018)
[48] MrDude. “3D model polygon count… How much is optimal?” *Unity Forum*, 20 July 2010, forum.unity.com/threads/3d-model-polygon-count-how-much-is-optimal.54832/#post-350119 (17.05.2018)
[49] Detocroix. “3D model polygon count… How much is optimal?” *Unity Forum*, 20 July 2010, forum.unity.com/threads/3d-model-polygon-count-how-much-is-optimal.54832/#post-350141 (17.05.2018)
[50] CrazyButcher. “[Technical Talk] – FAQ: Game art optimisation (do polygon counts really matter?)” *Polycount Forum*, 25 November 2007, polycount.com/discussion/50588/technical-talk-faq-game-art-optimisation-do-polygon-counts-really-matter (17.05.2018)
[51] Johnstone, Kevin. “[Technical Talk] – FAQ: Game art optimisation (do polygon counts really matter?)” *Polycount Forum*, 25 November 2007, polycount.com/discussion/comment/762412#Comment_762412 (17.05.2018)
[52] perna. “[Technical Talk] – FAQ: Game art optimisation (do polygon counts really matter?)” *Polycount Forum*, 25 November 2007, polycount.com/discussion/comment/762429#Comment_762429 (17.05.2018)
[53] Subdivision Surface Modifier. *Blender 2.79 Manual*. Blender. docs.blender.org/manual/en/dev/modeling/modifiers/generate/subsurf.html (17.05.2018)
[54] Catmull, Edwin, and James Clark. "Recursively generated B-spline surfaces on arbitrary topological meshes." Computer-aided design 10.6 (1978): 350-355.

[55] "Illustration of the iterative subdivision of a cube using the Catmull-Clark algorithm implemented in Mathematica." *Catmull-Clark subdivision surface*. RosettaCode, 5 October 2017. rosettacode.org/wiki/File:CAM_noholes_1.png (17.05.2018)
[56] Bevel. *Blender 2.79 Manual*. Blender. docs.blender.org/manual/en/dev/modeling/meshes/editing/subdividing/bevel.html (17.05.2018)
[57] "Selected edge before beveling, result of bevel (one segment), result of bevel (vertex only)". *Bevel*. Blender. docs.blender.org/manual/en/dev/modeling/meshes/editing/subdividing/bevel.html (17.05.2018)
[58] Shah, Karan. "Sculpt, Model and Texture Low-Poly Skull in Blender." *Envato Tuts+*. Envato Pty Ltd, 9 June 2009. cgi.tutsplus.com/articles/sculpt-model-and-texture-a-low-poly-skull-in-blender--cg-7 (17.05.2018)
[59] Shah, Karan. "Skull model created with a low-poly mesh, texture, and a normal map." Sculpt, Model and Texture Low-Poly Skull in Blender. *Envato Tuts+*. Envato Pty Ltd, 9 June 2009. cgi.tutsplus.com/articles/sculpt-model-and-texture-a-low-poly-skull-in-blender--cg-7 (17.05.2018)
[60] Shah, Karan. "Low-poly and high-poly skull mesh." Sculpt, Model and Texture Low-Poly Skull in Blender. *Envato Tuts+*. Envato Pty Ltd, 9 June 2009. cgi.tutsplus.com/articles/sculpt-model-and-texture-a-low-poly-skull-in-blender--cg-7 (17.05.2018)
[61] Austro. "Normal map." Wooden Chair. *TurboSquid*. TurboSquid. www.turbosquid.com/FullPreview/Index.cfm/ID/791045 (17.05.2018)
[62] Austro. "Base low poly model." Wooden Chair. *TurboSquid*. TurboSquid. www.turbosquid.com/FullPreview/Index.cfm/ID/791045 (17.05.2018)
[63] Austro. "Finished model." Wooden Chair. *TurboSquid*. TurboSquid. www.turbosquid.com/FullPreview/Index.cfm/ID/791045 (17.05.2018)
[64] Mesh. *Unity Manual. Version 2018.1*. Unity Technologies, 30 April 2018, docs.unity3d.com/ScriptReference/Mesh.html (17.05.2018)
[65] List <T> Class. *Microsoft Developer Network.* Microsoft, msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx (17.05.2018)
[66] Marschner, Steve, and Peter Shirley. Fundamentals of Computer Graphics. 3rd ed. CRC Press, 2010, pp. 263.
[67] Marschner, Steve, and Peter Shirley. Fundamentals of Computer Graphics. 3rd ed. CRC Press, 2010, pp. 24.
[68] Nykamp, Duane Q. "The scalar triple product." *Math Insight*. mathinsight.org/scalar_triple_product (17.05.2018)
[69] Marschner, Steve, and Peter Shirley. Fundamentals of Computer Graphics. 3rd ed. CRC Press, 2010, pp. 23.
[70] Vector3.SignedAngle. *Unity Manual. Version 2018.1*. Unity Technologies, 30 April 2018, docs.unity3d.com/ScriptReference/Vector3.SignedAngle.html (17.05.2018)
[71] Vector3.cs. *Unity-Technologies/UnityCsReference*. Github, 18 April 2018. github.com/Unity-Technologies/UnityCsReference/blob/master/Runtime/Export/Vector3.cs (17.05.2018)
[72] Marschner, Steve, and Peter Shirley. Fundamentals of Computer Graphics. 3rd ed. CRC Press, 2010, pp. 19.
[73] Algma, Diana. "Creating new UV Maps after Edge Chamfering." *Arvutigraafika projekt*. Tartu Ülikooli arvutiteaduse instituut, 27 April 2018. courses.cs.ut.ee/2018/cg-pro/spring/Main/ProjectUVMapsEdgeChamfering (17.05.2018)

[74] Marschner, Steve, and Peter Shirley. Fundamentals of Computer Graphics. 3rd ed. CRC Press, 2010, pp. 44-49.
[75] Algma, Diana. "Using barycentric coordinates to calculate the UV coordinates of point C1." Creating new UV Maps after Edge Chamfering. *Arvutigraafika projekt*. Tartu Ülikooli arvutiteaduse instituut, 27 April 2018. courses.cs.ut.ee/2018/cg-pro/spring/Main/ProjectUVMapsEdgeChamfering (17.05.2018)
[76] Frick, Tor. Scifi ship speedmodel. *Artstation*. Ballistiq Digital Inc. www.artstation.com/artwork/ZOmJ0 (18.06.2018)
[77] Algma, Diana. Edge Chamfering Algorithm. *Bitbucket*, Atlassian Inc, 21 May 2018. bitbucket.org/DAlgma/edgechamferingalgorithm/src/master/ (21.05.2018)

Appendix

I. Barycentric coordinates, and how they are used

This appendix describes what are barycentric coordinates [74] and how they are used in this thesis.

Barycentric coordinates enable defining the position of a point in two dimensions using three points. The position of point p can be defined through points a, b, and c as follows:

$$p = a + \beta(b - a) + \gamma(c - a) \Longrightarrow$$

$$p = (1 - \beta - \gamma)a + \beta b + \gamma c \xrightarrow{a \equiv 1 - \beta - \gamma}$$

$$p(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c;\ \ \alpha + \beta + \gamma = 1$$

With barycentric coordinates, the position of a point in one space can be calculated so it would be in the same position relative to three other points in another space. For this, first, α, β, and χ are calculated using p, a, b, and c in the first space. Then, p is calculated using α, β, and χ, and points a, b, and c in the second space.

Barycentric coordinates are mostly used for linearly interpolating the properties of a triangle using the properties of its vertices. In this thesis, they are used to calculate the UV coordinates of a point given a triangle on the model and the UV map, and the coordinates of the point on the model.

II. Edge Chamfering asset

The asset created in the thesis is accessible from a private repository [77]. The repository is private to enable the author to sell the asset in the Unity Asset Store in the future. To review the source code, either the access to the repository or the asset itself can be obtained by contacting the author, for example, by email (diana.algma@eesti.ee).

III. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Diana Algma**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

    1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

    1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Edge Chamfering Algorithm,

supervised by Jaanus Jaggo,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **21.05.2018**