1 Einleitung

Der größere Projekt-Rahmen, in den diese Arbeit eingegliedert wird, ist der Bau eines Replikators. Objekte sollen von einem 3D-Scanner eingescannt und von einem 3D-Drucker ausgedruckt werden. Die gescannten Modelle sind jedoch nur selten direkt für den Druck bereit und müssen daher zuvor noch bearbeitet werden. Eine Voraussetzung für den Druck ist, dass Modelle wasserdicht sein müssen – sich in ihrer Oberfläche also keine Löcher befinden. Ein solches Loch befindet sich i.d.R. immer in der Stehfläche des Modells, da der 3D-Scanner diese nicht erfassen kann.

Ziel dieses Projektes wird es sein, Löcher in 3D-Modellen zu finden und diese zu füllen. Das so bearbeitete Modell muss im Anschluss wieder in ein für 3D-Drucker gebräuchliches Format exportiert werden – typischerweise handelt es sich dabei um das STL-Format. Das Ganze soll plattformunabhängig als Web-Anwendung zur Verfügung gestellt werden.

Es existieren bereits verschiedene Verfahren, um Löcher zu füllen und die erzeugten Flächen weiter an ihre Umgebung anzupassen, damit sie natürlicher erscheinen. So wird in [1] das Loch erst mit dem Advancing Front-Algorithmus gefüllt und anschließend noch auf Basis von Poisson-Gleichungen an die umliegende Struktur des Modells angeglichen. In [2] wird ebenfalls über die Geometrie gearbeitet, jedoch werden Radiale Basisfunktionen eingesetzt. Ein Volumen-basiertes Verfahren wird in [3] eingesetzt, wobei eine Octree-Datenstruktur sowohl für das Finden von Löchern, als auch das Angleichen an die Umgebung herangezogen wird.

2 3D im Browser

WebGL ist eine Spezifikation für die Darstellung von 3D-Inhalten im Browser. Entwickler können WebGL über eine JavaScript-API nutzen, vorausgesetzt, der verwendete Browser unterstützt WebGL. PlugIns oder ähnliche Erweiterungen werden hierfür nicht benötigt.

Mit Firefox 4 und Chromium 8 hielt eine erste WebGL-Unterstützung in diesen Browsern Einzug [4]. Die aktuellen Versionen Firefox 23 und Chromium 28 sind sowohl in den verfügbaren WebGL-Features, als auch der erbrachten Leistung uneingeschränkt als 3D-Umgebung nutzbar.

Der Umgang mit der WebGL-API ähnelt dem mit OpenGL. Um die Entwicklung zu erleichtern, empfiehlt es sich daher auf eine WebGL-Bibliothek zurückzugreifen. In diesem Projekt wurde sich für *three.js* [5] entschieden, da es eine der weitverbreiteren Bibliotheken ist und über eine ausreichende Dokumentation verfügt.

3 Erkennung von Löchern

Die definierende Eigenschaft von Löchern ist, dass jede Kante, die zu einem Loch gehört, nur zu einem Face gehört. Um diese Zugehörigkeiten zu entdecken, bietet sich als Datenstruktur ein HalfEdge-Mesh an. Ein HalfEdge-Objekt hält als Information einen Ausgangs-Vertex; ein Face, zu dem es gehört; das nächste HalfEdge, also die nächste Kante; und das gegenläufige HalfEdge. Somit liegen einem Informationen über alle Kanten und welche Faces sie bilden vor. Mit dem Wissen, dass eine Kante zu einem Loch gehört, wenn sie nur zu einem einzigen Face gehört, lassen sich dann die Löcher im Modell finden (siehe Abbildung 1).

Hierbei wurde auf die Java-Implementierung eines Kollegen aus der Arbeitsgruppe „Computer Vision und Mixed Reality“ zurückgegriffen, die nur noch nach JavaScript portiert werden musste.

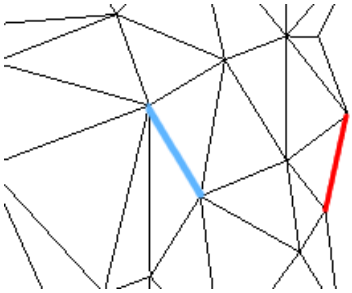

Abbildung 1: Blau markiert *(links)*: Kante, die zu zwei Faces gehört. Rot markiert *(rechts)*: Kante, die zu nur einem Face gehört und daher Teil eines Loches ist.

4 Füllen von Löchern

Algorithmen zum Füllen von Löchern lassen sich größtenteils in eine von zwei Kategorien einsortieren: Geometrie-basierte Verfahren und Volumen-basierte Verfahren.

Geometrie-basierte Verfahren berechnen neue Punkte für die Füllung anhand der Oberfläche des Models. Ein typisches Verfahren dieser Art ist die *Advancing Front*, bei der das Loch iterativ verkleinert wird (vgl. [1, 6]).

Volumen-basierte Verfahren arbeiten hingegen mit dem Volumen des Modells. Über das Volumen lässt sich die Form des Modells ableiten und Löcher gemäß der errechneten Form abdecken (vgl. [3]).

Für dieses Projekt wurde das Geometrie-basierte **Advancing Front**-Verfahren gewählt. Ausgangspunkt ist dabei die Form des gefundenen Loches – also alle Kanten, die das Loch begrenzen. Dies ist die initiale Front, die im Anschluss iterativ verkleinert wird, bis das Loch schließlich geschlossen ist. Das Verkleinern wird erreicht, indem an jedem Vertex der Front eine entsprechende Regel angewendet wird, die sich nach den Winkeln benachbarter Vertices richtet.

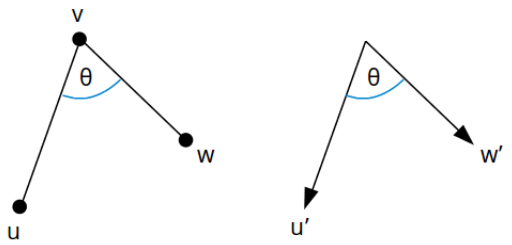

Abbildung 2: Winkel θ zwischen den Vektoren u' und w'.

Nachdem die initiale Front eines Loches feststeht, werden zunächst die Winkel aller Vertices berechnet. Für den Winkel eines Vertex werden neben dem Vertex selbst noch die beiden Nachbarn benötigt. Seien u, v und w Vektoren, wobei u der Nachbar von v und v der Nachbar von w ist (siehe Abbildung 2). Der Winkel θ von v errechnet sich dann wie in Gleichung 1 beschrieben.

$$\begin{aligned} u' &= u - v \\ w' &= w - v \\ \theta &= \cos^{-1}\left(\frac{u' \cdot w'}{|u'| \cdot |w'|}\right) \end{aligned} \tag{1}$$

Von den erhaltenen Winkeln wählt man den kleinsten und beginnt an dessen Vertex mit dem Füllen des Loches. Dafür wird abhängig von der Winkelgröße eine von drei Regeln angewandt:

1. Regel für $\theta \leq 75°$
2. Regel für $75° < \theta \leq 135°$
3. Regel für $135° < \theta$

Im Fall von **Regel 1** muss lediglich der Winkel geschlossen werden, indem eine neue Verbindung von u nach w teil der neuen Front wird. Der Vertex v wird dadurch von der Front ausgeschlossen (siehe Abbildung 3).

Bei **Regel 2** wird ein neuer Vertex benötigt. Hierfür bietet es sich an, diesen Winkelhalbierend zu setzen und dann auf einen ähnlichen Abstand wie u zu v oder w zu v zu versetzen (siehe Abbildung 3).

Für **Regel 3** bieten sich verschiedene Ansätze, wobei jedoch immer mindestens ein neuer Vertex entsteht. In [6] wird z.B. angedeutet, dass zwei neue Vertices hinzugefügt werden, die den Winkel dritteln. In diesem Projekt wird ein neues Face an die Kante von v nach w gesetzt (siehe Abbildung 3).

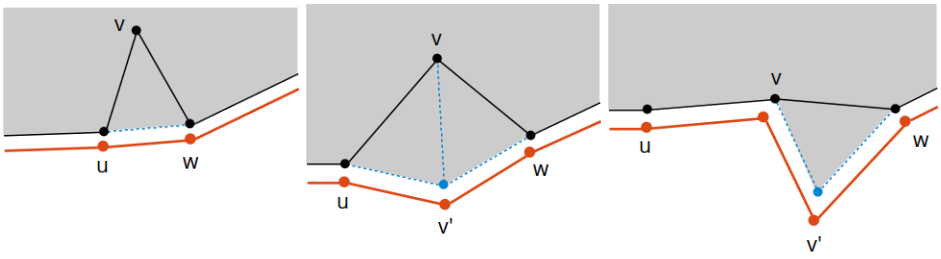

Abbildung 3: Änderungen an der Front bei Regel 1, 2 und 3 (v.l.n.r.). In Schwarz ist der alte Verlauf zu sehen, während rot die neue Front darstellt (leicht nach unten verschoben, damit besser zu erkennen).

5 Implementierung

Aufgrund der verwendeten JavaScript-Features wird ein halbwegs aktueller Browser benötigt. Während der Entwicklung wurden Firefox 23 und Chromium 28 verwendet. WebGL und Web Workers erfordern mindestens Firefox 4 oder Chromium 8. Aus Performance-Gründen sollte jedoch eine aktuelle Version gewählt werden.

Als JavaScript-Framework für den Umgang mit WebGL wurde *three.js* (r60) eingesetzt. Datenobjekte wie `THREE.Vector3` wurden dann auch für Berechnungen während des Füllprozesses weiterverwendet.

5.1 Finden von Löcher

Unter den Testmodellen befanden sich auch zwei mit problematischer Struktur. Das Modell *puppe.obj* hatte Vertices, die nicht Teil eines Faces waren, und sowohl das Modell *puppe.obj* als auch *adonis.obj* hatten Vertices, die unter anderem mit sich selbst Faces gebildet haben. In der 3D-Darstellung waren diese fehlerhaften Bereiche nicht zu sehen, bei der Löcher-Suche führten sie jedoch mitunter zu Schwierigkeiten. In einem ersten Schritt werden Modelle nach dem Laden in die Anwendung daher erst einmal auf solche fehlerhaften Vertices hin untersucht und entsprechend korrigiert. Dies bedeutet, dass fehlerhafte Vertices und dadurch ebenfalls betroffene Faces entfernt werden. In der Darstellung der Modelle verändert sich dadurch nichts und die Löchersuche ergibt korrekte Ergebnisse.

Zur Löchersuche wird zuerst ein HalfEdge-Mesh aufgebaut. Dadurch liegen alle benötigten Informationen vor, wie die Verbindungen der Vertices untereinander. Von Interesse sind dabei diejenigen Vertices, die Teil eines Loches sind. Zwischen zwei solchen Randpunkten liegt eine Randkante. Randkanten erkennt man daran, dass sie nur Teil eines einzigen Faces sind, während alle anderen Kanten zu zwei Faces gehören.

Aus den vorhandenen Randpunkten wird ein beliebiger als Startpunkt ausgewählt. Dieser Randpunkt ist der Ausgangspunkt für das erste Loch. Von diesem Randpunkt aus wird zum nächsten benachbarten Randpunkt gegangen – die HalfEdge-Struktur gibt uns diese Information. Es wird solange von Randpunkt-Nachbar zu Randpunkt-Nachbar gesprungen, bis man wieder am Ausgangspunkt angelangt ist. Damit ist das erste Loch gefunden. Die bisher besuchten Punkte werden markiert, damit sie im weiteren Verlauf nicht mehr beachtet werden. Danach wird der nächste unmarkierte Randpunkt gewählt und das nächste Loch abgelaufen. Dies wird solange wiederholt, bis keine unmarkierten Randpunkte mehr vorhanden sind.

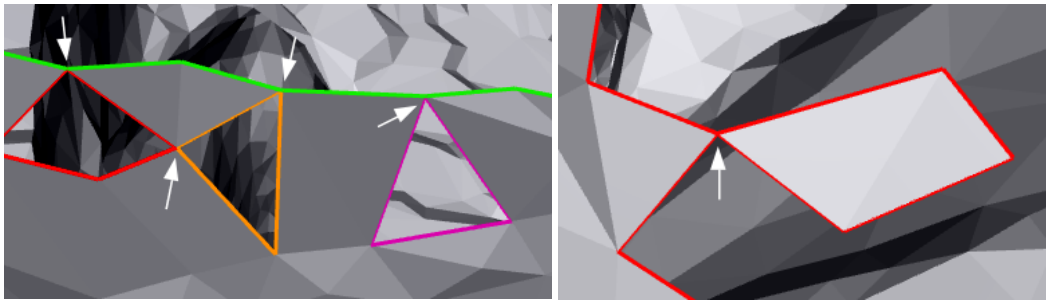

Abbildung 4: Mit weißen Pfeilen markiert: Randpunkte, die zu mehreren Löchern (farbig hervorgehoben) oder mehrfach zum selben gehören.

Bisher wird jedoch nicht ein Spezialfall beachtet, der für Probleme in der Löchererkennung sorgen kann. Dies kann passieren, wenn ein Randpunkt zu mehr als einem Loch gehört oder mehrfach zum selben (siehe Abbildung 4). Diese Punkte werden im Weiteren als „Multirandpunkte“ bezeichnet. Um diese Fälle angemessen zu behandeln, muss das bisherige Vorgehen angepasst werden. Multirandpunkte dürfen nach einem Besuch nicht gleich als besucht markiert werden, da sie für ein anderes Loch erneut besucht werden müssen. Ein Multirandpunkt darf erst markiert werden, nachdem auch alle seine Nachbar-Randpunkte markiert wurden. Trifft man während dem Ablaufen der Randpunkte eines Loches auf einen Multirandpunkt, findet man den nächsten Randpunkt des aktuellen Loches, indem man die Winkel zwischen dem zuvor besuchten Punkt, dem aktuellen Multirandpunkt und allen potentiellen nächsten Randpunkten berechnet. Daraufhin wählt man den Randpunkt, mit dem der kleinste Winkel gebildet wird (siehe Abbildung 5).

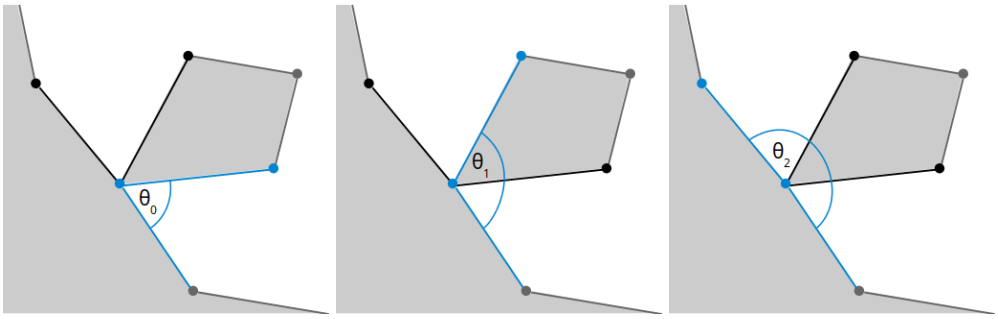

Abbildung 5: Bestimmung des nächsten Randpunktes nach einem Multirandpunkt über die Winkel.

5.2 Advancing Front

Während der Ausführung des Advancing Front-Algorithmus wird immer nur ein einziges Loch betrachtet und gefüllt. Es werden drei Listen geführt: `front`, die die aktuell in der Front vorhanden Vertices enthält; `filling.vertices`, die die aktuell in der Füllung vorhanden Vertices enthält; und `filling.faces`, die die aktuell in der Füllung vorhandenen Faces enthält. Ein Face beinhaltet dabei lediglich die Indizes seiner Vertices in der Liste `filling.vertices`. Wird später die Füllung mit dem restlichen Modell vereinigt, müssen diese Indizes entsprechend angepasst werden.

Initiale Winkel. Zuerst müssen die Winkel der Vertices berechnet werden. Um den Winkel zu einem Vertex zu berechnen, benötigt man den Vektor `v` an dieser Stelle, den vorherigen Vektor `vp`, sowie den darauffolgenden Vektor `vn` innerhalb der Front des Loches. Zunächst müssen die Vektoren relativ zu `v` in den Ursprung verschoben werden. Falls nicht anders erwähnt, handelt es sich bei den Vektoren im Code um `THREE.Vector3`-Instanzen.

```
1 vp.sub( v );
2 vn.sub( v );
```

Für die Winkelberechnung und Umrechnung von Radianten in Grad kann auf Funktionen von three.js zurückgegriffen werden.

```
3 var angle = vp.angleTo( vn );
4 angle = THREE.Math.radToDeg( angle );
```

Für den Advancing Front-Algorithmus brauchen wir den zum Loch gerichteten Winkel. Bisher erhält man über diese Berechnung jedoch nur den kleineren Winkel, der auch in die andere Richtung gerichtet sein kann. Um dies nachträglich zu korrigieren, berechnet man zunächst das Kreuzprodukt von `vp` und `vn` und verschiebt den resultierenden Vektor zurück in das Modell.

```
5 var cross = new THREE.Vector3().crossVectors( vp, vn );
6 cross.add( v );
```

Ist der Kreuzvektor kürzer – und somit näher am Ursprung – als der Winkelvektor `v`, wird der gegenüberliegende Winkel genommen:

```
7 if( cross.length() < v.length() ) {
8     angle = 360.0 - angle;
9 }
```

Leider ist dieses Vorgehen nicht hundertprozentig verlässlich und so kann es besonders bei Löchern, deren Vertices kaum auf einer Ebene liegen, zu falschen Winkel-Werten kommen, die zu fehlerhaften Füll-Ergebnissen führen.

Die Winkel und zugehörige Informationen werden in Instanzen der Klasse `Angle` gespeichert. Ein `Angle`-Objekt hat Attribute für den Winkel in Grad, die drei zugehörigen Vektoren, die vorherige `Angle` in der Front, sowie die nächste `Angle`. Die an eine doppel-verkettete Liste erinnernde Struktur ist später notwendig, um durch den Füllprozess veränderte Winkel zu aktualisieren.

Heap-Vorgehen. Die im vorherigen Schritt erzeugten Winkel werden in einem Heap gesammelt. `Angle`-Objekte werden dabei mit ihrer Winkelgröße als Schlüssel eingefügt. Anschließend werden die Schlüssel aufsteigend sortiert, wodurch beim kleinsten Winkel in der Front begonnen wird. Nach jeder angewandten Regel wird der Heap erneut sortiert, so dass immer zuerst der kleinste Winkel abgearbeitet wird. Dies ist wünschenswert, da die Regeln 1 und 2 stabilere Ergebnisse liefern als Regel 3. Durch das Vorziehen der kleineren Winkel entsteht somit eine bessere Gesamtfüllung.

Nachdem anhand des Winkels eine der Advancing Front-Regeln angewendet wurde, müssen die Nachbarn des Winkels aktualisiert werden, da sich durch den Füllvorgang die Front verändert. Die Vertices der vorherigen und nachfolgenden `Angle` werden aktualisiert, der zugehörige Winkel neu berechnet und die `Angle` neu im Heap einsortiert.

Regel 1 wird auf Winkel angewendet mit $\theta \leq 75°$. In diesem Fall wird für ein neues Face lediglich eine Verbindung zwischen `vp` und `vn` hinzugefügt, so dass das neue Face in der Füllung aus den Vertices `vp`, `v` und `vn` besteht (siehe Abbildung 6). Aus der Front muss der Vertex `v` entfernt werden.

Regel 2 gilt für Winkel mit $75° < \theta \leq 135°$. In diesem Fall wird ein neuer Vertex angelegt. Dafür werden die Vektoren relativ zu `v` in den Ursprung verschoben und eine Ebene aufgespannt. Winkelhalbierend wird dann innerhalb der Ebene ein

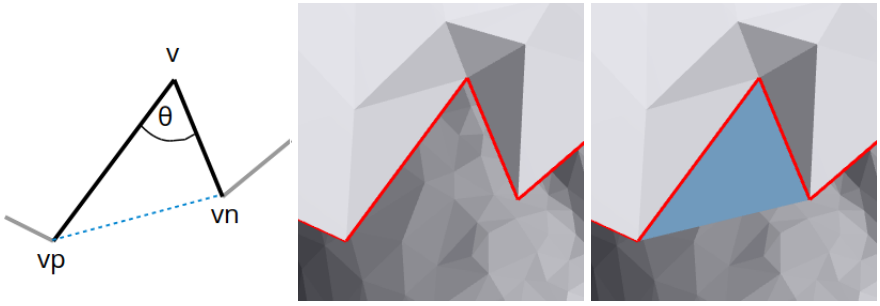

Abbildung 6: Ein Winkel vor und nach der Anwendung von Regel 1.

neuer Punkt gesetzt, wobei der Abstand des neuen Punktes von v der Mittelwert aus den Abständen von vp und vn nach v ist (siehe Abbildung 7).

```
1  var origin = new THREE.Vector3( 0, 0, 0 );
2  vp.sub( v );
3  vn.sub( v );
4
5  var avLen = Utils.getAverageLength( vp, vn );
6  var plane = new Plane( origin, vp, vn );
7
8  var vNew = plane.getPoint( 1, 1 );
9  vNew.setLength( avLen );
10 vNew.add( v );
```

Der neue Vertex vNew ersetzt in der Front dann den Vertex v. In der Füllung kommen zwei neue Faces hinzu: (v, vp, vNew) und (v, vNew, vn).

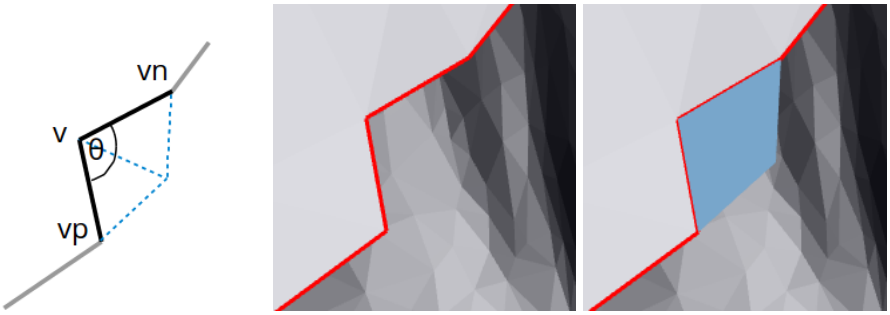

Abbildung 7: Ein Winkel vor und nach der Anwendung von Regel 2.

Regel 3 gilt für Winkel mit $135° < \theta$. Für solche Winkel wird ein neuer Vertex vNew zwischen v und vn eingefügt, der ein neues Face (vn, v, vNew) bildet. Der neue Vertex wird bestimmt, indem zunächst das Kreuzprodukt c1 von vn und vp berechnet wird. In der Regel zeigt dieses Kreuzprodukt vom Loch aus nach innen in das Modell, wenn man von außen auf das Loch schaut.

```
1  var vnClone = vn.clone().sub( v ),
2      vpClone = vp.clone().sub( v );
3  var c1 = vnClone.clone().cross( vpClone ).normalize();
```

Als nächstes wird das Kreuzprodukt `c2` von `c1` und `v` berechnet, welches ungefähr in das Loch hineinzeigt. Ausgehend von `vn` und `c2` kann nun eine Ebene aufgespannt werden, auf der ein Punkt `vNew` gesetzt wird mit der selben Länge wie `vn` (siehe Abbildung 8). In diesem letzten Schritt ähnelt die Regel 3 der Regel 2.

```
4  var origin = new THREE.Vector3( 0, 0, 0 );
5  var c2 = c1.cross( vnClone ).normalize().add( v );
6  var plane = new Plane( origin, vnClone, c2.clone().sub( v ) );
7
8  var vNew = plane.getPoint( 1, 1 );
9  vNew.setLength( vnClone.length() );
10 vNew.add( v );
11 vNew = Utils.keepNearPlane( vNew, [vp, v, vn] );
```

Mit der Funktion `Utils.keepNearPlane` in Zeile 11 wird versucht, zu starke Wölbungen in eigentlich ebenen Löchern zu verhindern. Anhand der Varianz – gewonnen aus den Vektoren `vp`, `v` und `vn` – wird die Position des neuen Vertex angeglichen.

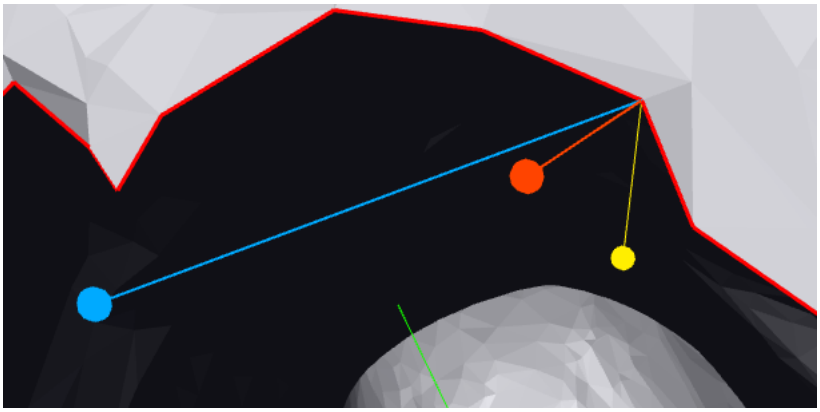

Abbildung 8: Während der Anwendung von Regel 3: `c1` (gelb), `c2` (blau), `vNew` (rot).

Kollisionstest. Für jedes neues Dreieck muss geprüft werden, ob es sich nicht mit einem anderen Bereich der bisherigen Füllung oder des Modells überschneidet. Dafür prüfen wir für jede Seite des neuen Dreiecks, ob es sich mit der Fläche eines der existierenden Dreiecke schneidet. Hierbei folgt die Implementierung der mathematischen Beschreibung aus [7]. Das neue Dreieck bestehe aus den Vektoren v_0, v_1 und v_2 und wird mit einem Dreieck mit den Vektoren w_0, w_1 und w_2 getestet.

In einem ersten Schritt wird anhand Formel 2 geprüft, ob die Gerade zwischen v_0 und v_1 die Ebene schneidet, in der das andere Dreieck liegt. Hierbei ist n die Normale der Ebene. Für einen Wert $0 < r < 1, \quad r \in \mathbb{R}$ existiert ein Schnittpunkt an der Stelle $v_0 + r \cdot (v_1 - v_0)$.

$$r = \frac{n \cdot (w_0 - v_0)}{n \cdot (v_1 - v_0)} \tag{2}$$

Existiert kein Schnittpunkt mit der Ebene, liegt auch keine Kollision mit diesem Dreieck vor. Andernfalls muss geprüft werden, ob die Gerade nicht nur die Ebene, sondern auch genau die Fläche des Dreiecks schneidet. Dies geschieht in Formel 3.

$$\begin{aligned} r' &= r - w_0 \\ w_1' &= w_1 - w_0 \\ w_2' &= w_2 - w_0 \\ s &= \frac{(w_1' \cdot w_2')(r' \cdot w_2') - (w_2' \cdot w_2')(r' \cdot w_1')}{(w_1' \cdot w_2')^2 - (w_1' \cdot w_1')(w_2' \cdot w_2')} \\ t &= \frac{(w_1' \cdot w_2')(r' \cdot w_1') - (w_1' \cdot w_1')(r' \cdot w_2')}{(w_1' \cdot w_2')^2 - (w_1' \cdot w_1')(w_2' \cdot w_2')} \end{aligned} \tag{3}$$

Gilt anschließend, dass $s, t \geq 0$ und $s + t = 1$, dann existiert ein Schnittpunkt der Geraden mit dem Dreieck. Vorausgesetzt es wurde noch kein Schnittpunkt gefunden, muss der Test noch einmal mit der Geraden von v_1 nach v_2 wiederholt werden – außer bei Regel 1, da hier nur eine Seite des Dreiecks neu ist.

Aufgrund einer gefundenen Kollision muss ein neues Face abgelehnt werden. Der Winkel in der Front besteht jedoch unverändert weiter, weshalb das `Angle`-Objekt wieder in den Heap zurück muss. Um zu vermeiden, dass nach dem Sortieren die selbe `Angle`, deren Ergebnis gerade abgelehnt wurde, gleich wieder bearbeitet wird, wird ein Flag `Angle.waitForUpdate` auf `true` gesetzt. Bis sich der Winkel der `Angle` ändert – durch Veränderung der Nachbar-Vertices –, bleibt dieses Flag bestehen.

Standardmäßig wird aus Performance-Gründen der Kollisionstest nur mit den Faces der Füllung durchgeführt, da hier das Risiko von Kollisionen auch am höchsten ist. Optional lässt sich im Benutzer-Interface aber auch auf einen vollständigen Test mit dem gesamten Modell umstellen.

Merging. Zwei Vertices, die nahe aneinander liegen, müssen gemergt – vereinigt – werden. Wie groß der Abstand zwischen den zwei Punkten ist, bevor sie vereinigt

werden, wird individuell für jedes Loch berechnet. Dabei wird der durchschnittliche Abstand aller benachbarten Vertices des Loches als Grenzwert vorgeschlagen. Ein niedriger Grenzwert für den Abstand resultiert dabei in einer feineren Füllung (mehr Faces), während ein hoher Grenzwert zu einer groberen Füllung führt. Dieser Unterschied wird in Abbildung 9 veranschaulicht.

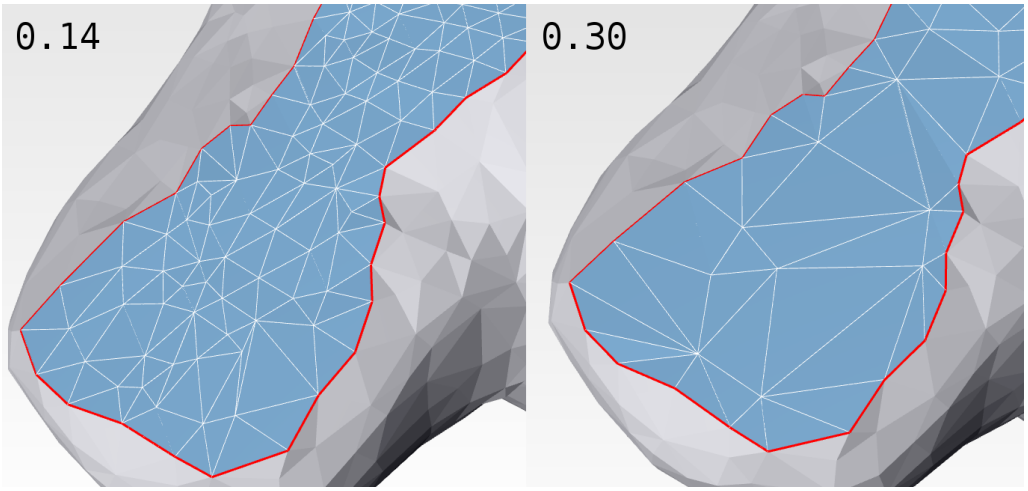

Abbildung 9: Vergleich von Merging-Grenzwerten: 0,14 *(links)* und 0,30 *(rechts)*. Von der Anwendung für das Loch empfohlen waren 0,289.

Eine Prüfung, ob Vertices vereinigt werden können, findet jedes Mal nach Anwendung einer Regel statt. Verglichen wird dabei lediglich der neue Vertex mit seinen beiden Nachbarn in der Front – für Regel 1 entfällt das Merging, da kein neuer Vertex entstand. Liegt der Abstand unter einem vorgegebenen Grenzwert, wird der Nachbarpunkt `t` mit dem neuen Vertex `v` vereint, was konkret bedeutet, dass Vertex `t` entfernt wird (siehe Abbildung 10).

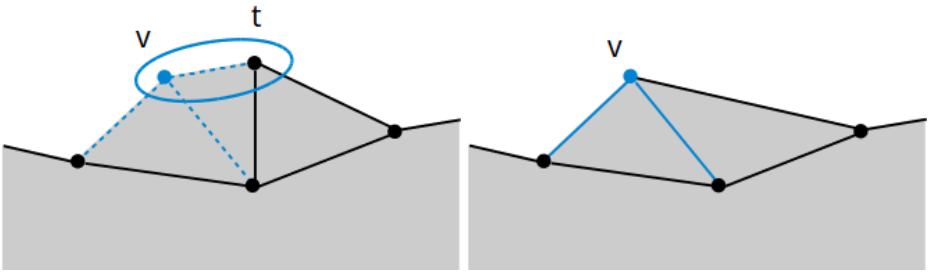

Abbildung 10: Merging: Der Abstand der eingekreisten Punkte im linken Bild liegt unterhalb des Grenzwertes und `t` wird daher mit `v` vereinigt. Das Resultat ist im rechten Bild zu sehen.

Faces in `filling`, in denen der Vertex `t` ein Bestandteil war, müssen daraufhin aktualisiert werden. An die Stelle tritt nun der neue Vertex `v`. Da die Faces als An-

gaben nur die Indizes ihrer Vertices speichern, müssen ebenfalls all jene Indizes um den Wert 1 verringert werden, die einen Index größer als der entfernte Vertex aufweisen. Ebenso muss `t` aus aus der `front`-Liste entfernt werden und alle `Angle`-Objekte im Heap, die u.a. durch den Vertex `t` gebildet werden, müsse aktualisiert werden.

5.3 Web Workers

Web Workers sind ein relativ neues Feature, mit denen sich JavaScript-Code als Hintergrundprozesse im Browser ausführen lassen. Auf Systemen mit mehr als einem CPU-Kern lässt sich so Arbeit parallelisieren. Für dieses Projekt wurde der Kollisionstest zur Parallelisierung gewählt. Jeder Worker-Prozess erhält dabei eine andere Untermenge der zu testenden Faces.

Workers lassen sich auf zwei Weisen anlegen: Zum Einen als eigene JS-Datei, deren Pfad dann angegeben wird, und zum anderen als Inline-Worker, wenn der Code mit in die `index.html`-Datei geschrieben wird. Da das Projekt jedoch auch ohne Server lauffähig sein soll, können nur Inline-Workers eingesetzt werden. Der Code für den Worker wird wie üblich innerhalb eines `<script>`-Tags in der HTML-Datei eingefügt:

```
1 <script id="worker-collision" type="javascript/worker">
2     // ... JavaScript-Code ...
3 </script>
```

Ein neuer Worker wird dann erzeugt, indem der Inline-Code ausgelesen, in einem `Blob`-Objekt verpackt und als Objekt an den Worker übergeben wird.

```
1 var code = document.getElementById( "worker-collision" ).textContent;
2 var blob = new Blob( [code], { type: "application/javascript" } );
3 var workerBlobURL = window.URL.createObjectURL( blob );
4 var worker = new Worker( workerBlobURL );
```

Dann benötigt der Worker allerdings immer noch Zugriff auf andere Klassen, wie z.B. `Utils` oder `THREE`. Hierfür existiert speziell für die Worker-Prozesse die Funktion `importScripts`, die jedoch einen absoluten Pfad zur JavaScript-Datei erfordert. Um die Anwendung flexibel zu halten, kann dieser nicht direkt festgelegt werden. Zunächst wird der Pfad außerhalb bestimmt und dann an den Worker geschickt.

```
1 var theURL = document.location.href;
2 var index = theURL.indexOf( "index.html" );
3 if( index != -1 ) {
4     theURL = theURL.substring( 0, index );
```

```
5 }
6
7 worker.postMessage( { cmd: "url", url: theURL } );
```

Innerhalb des Workers werden Nachrichten wie folgt behandelt und die Skripte eingebunden:

```
1  function handleMessages( event ) {
2      switch( event.data.cmd ) {
3          case "url":
4              var url = event.data.url + "js/";
5              importScripts(
6                  url + "threeJS/three.min.js",
7                  url + "Plane.js",
8                  url + "Utils.js"
9              );
10             break;
11         // ... handle other commands ...
12     }
13 }
14
15 self.addEventListener( "message", handleMessages, false );
```

Eine sehr direkte Implementierung, in der für jeden Test ein neuer Worker-Prozess gestartet wurde, war jedoch viel zu langsam. Eine erste Optimierung war daher, einmalig einen Pool fester Größe an Worker-Prozessen anzulegen. Für den Kollisionstest werden die zu testenden Daten dann lediglich in den Pool gegeben und freie Workers bearbeiten sie. Dies wird über die Klasse `WorkerManager` realisiert. Als Pool-Größe bzw. Worker-Anzahl hat sich eine Größe entsprechend der Anzahl an CPU-Kernen bewährt. Auf einem Rechner mit 4 Kernen (Intel i5) werden also 4 Workers gestartet.

Der Kollisionstest (zu diesem Zeitpunkt nur mit der Füllung) war jedoch immer noch 8× langsamer als die nicht parallelisierte Implementierung. Ursache ist der Aufwand für die Kommunikation, denn Nachrichten-Objekte müssen erst serialisiert und auf Worker-Seite wieder de-serialisiert werden. Dies geschieht beim Aufruf von `postMessage` automatisch. Je größer die zu testende Füllung, desto größer ist auch die Nachricht. Eine unerwartete Optimierung bestand darin, die Daten zuvor selbst in JSON umzuwandeln.

```
1 data.faces = JSON.stringify( data.faces );
```

Im Worker muss das JSON auch wieder entpackt werden:

```
1 var faces = JSON.parse( event.data.faces );
```

Mit dieser kleinen Änderung ist die parallele Implementierung nur noch ca. 5× langsamer als die iterative, was den Kollisionstest für die Füllung betrifft. Wie später in der Evaluierung zu sehen sein wird, nähert sich die parallele Version jedoch an die iterative an, wenn mit dem gesamten Modell geprüft wird. Von Vorteil ist dabei, dass kaum zusätzlicher Kommunikationsaufwand entsteht, denn das Modell muss nicht für jeden Test erneut an den Worker geschickt werden. Da sich das Modell während dem Füll-Prozess nicht verändert, reicht es, es zu Beginn einmal an die Workers zu senden und später nur noch das zu testende Face zu übermitteln.

5.4 Verschiedene Modi

Der Advancing Front-Algorithmus lässt sich in drei verschiedenen Modi ausführen: *iterative*, *responsive* und *parallel*. Die Zwischenschritte und Ergebnisse des Füll-Prozesses sind immer die selben, jedoch unterscheiden sich die Modi im internen Ablauf.

Mit **iterative** wird der Algorithmus in einer `while`-Schleife so lange ausgeführt, bis er terminiert. Dieser Modus ist der schnellste, dafür reagiert das Browser-Fenster während dieser Zeit auf keine Eingaben oder durch JavaScript ausgelöste DOM-Änderungen (z.B. aktualisieren des Fortschrittsbalkens).

In **responsive** bleibt das Fenster weiterhin gegenüber Eingaben offen – das Modell kann z.B. gedreht werden und der Fortschrittsbalken verändert sich. Dies ist möglich, da dieses Vorgehen nicht auf einer großen Schleife beruht, sondern auf Funktionsaufrufen. Es gibt eine Hauptfunktion, die am Ende immer wieder aufgerufen wird – was bei einer Schleife einem Durchlauf entspräche. Diese Funktionsaufrufe werden vom Browser intern auf einem Stack verwaltet und abgearbeitet. Dadurch, dass die eine große Schleife aus dem iterativen Modus in viele Funktionsaufrufe aufgebrochen wurde, ist es nun möglich, dass Ereignisse der GUI dazwischen abgearbeitet werden können (siehe Abbildung 11). Das geht allerdings auf Kosten des Füllprozesses, weshalb dieser Modus deutlich langsamer ist.

Der **parallele** Modus ist erst einmal ähnlich zu *responsiv*. Allerdings wird der Kollisionstest über Web Workers parallelisiert, was einen Geschwindigkeitsschub in der Ausführung liefert. Wie viel Zeit dadurch gewonnen wird, hängt aber auch von Faktoren ab wie der Anzahl an CPU-Kernen und der Anzahl an Faces, die getestet werden müssen.

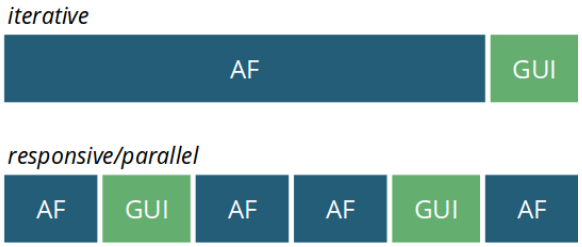

Abbildung 11: Vereinfacht dargestellter interner Ablauf im Browser. Der iterative Modus blockiert das Abarbeiten von GUI-Ereignissen.

5.5 Modell-Export

Der Export unterstützt die beiden Formate OBJ und STL – letzteres wird üblicherweise im 3D-Druck verwendet. Da es mit JavaScript nicht möglich ist, direkt Dateien auf ein Speichermedium zu schreiben, kann der erzeugte Export lediglich zum Download angeboten bzw. – da die Anwendung lokal ausgeführt wird – ein Speichern-Dialog aufgerufen werden. In JavaScript ist dies wie folgt umgesetzt:

```
1  var exportData = SceneManager.exportModel( "obj" );
2  var content = new Blob( [exportData], { type: "text/plain" } );
3  var download = document.createElement( "a" );
4
5  download.download = "myFile.obj";
6  download.href = window.URL.createObjectURL( content );
7  download.setAttribute( "hidden", "hidden" );
8  document.body.appendChild( download );
9
10 download.click();
```

In Zeile 1 wird das Modell zu OBJ umgewandelt. Die Variable `exportData` enthält den gesamten OBJ-Inhalt als String. In Zeile 2 wird der OBJ-Text in einen Blob gepackt. In Zeile 3 ff. wird ein HTML-Link erzeugt und erhält das Attribut `download`. Der Wert für das Attribut wird der Dateiname. Im `href`-Attribut wird der Dateiinhalt angegeben. Um den Speichern-Dialog schließlich auszulösen, wird in Zeile 10 das Click-Event per `download.click()` gefeuert.

Für den STL-Export müssen zudem alle Koordinaten von `float`-Werten in die wissenschaftliche Notation umgeschrieben werden. Anstatt `-0.02648` muss `-2.648e-2` geschrieben werden. Hierfür wurde die Funktion `Utils.floatToScientific` geschrieben, die einen float-Wert als String behandelt und entsprechend umschreibt. Da dies für jeden einzelnen Wert erfolgen muss, ist der STL-Export vergleichsweise langsamer als der Export für OBJ.

6 Evaluation

Die hier verwendeten Modelle haben eine ungefähre Anzahl von 10.300 Vertices und 20.400 Faces. Größere Modelle mit einer Face-Anzahl von 100.000 und mehr überlasten leider die Anwendung und machen die Bearbeitung unmöglich.

Der in diesem Projekt implementierte Advancing Front-Algorithmus funktioniert besonders gut mit Löchern, die ungefähr auf einer Ebene liegen, wie z.B. in Test-Modell *squirrel_bottom_hole_plane.obj* (Abbildung 14). Durch die in Kapitel 5.2 erwähnte Methode zum Verhindern von starken Wölbungen bei Regel 3 ergeben sich zudem relativ flache Füllungen, wie in Abbildung 12 zu sehen.

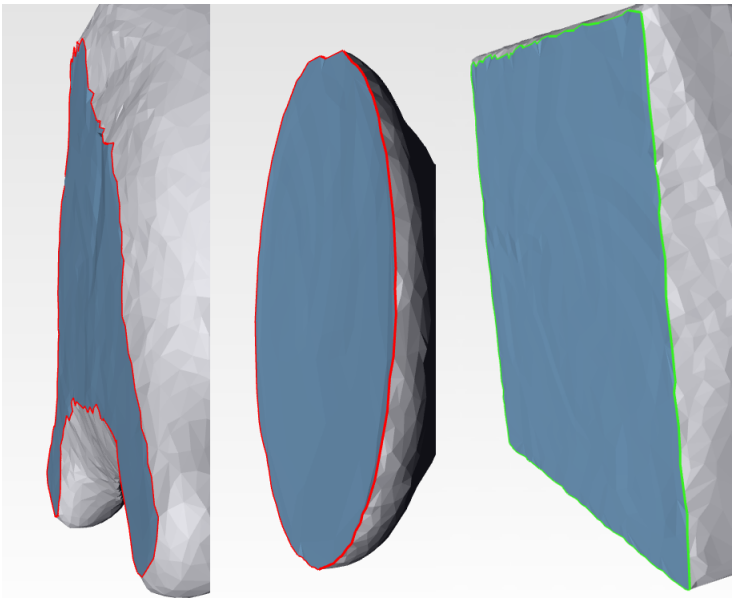

Abbildung 12: Die Modelle *squirrel_hole_bottom_plane.obj*, *diana.obj* und *adonis.obj* (v.l.n.r.) mit gefüllter Unterfläche.

Löcher, die sich z.B. über eine rundliche Oberfläche erstrecken, lassen sich ebenfalls füllen. Jedoch erscheint die Füllung oft unebener als der umliegende Bereich, wie links in der Abbildung 13 zu sehen. Hier wäre noch Raum für an den Füll-Prozess anschließende Optimierungsverfahren.

Problematischer sind Löcher, bei denen größere, unebene Abschnitte fehlen. Diese Löcher sind besonders anfällig für das in Kapitel 5.2 beschriebene Winkelkorrekturproblem. Auch wenn der Füll-Prozess gelingt, ist es keine gelungene Rekonstruktion, sondern lediglich eine provisorische Füllung. Wie rechts in der Abbildung 13 zu sehen ist, konnte die Rundung der fehlenden Spitze des Eichhörnchenschwänzchens durch den Füll-Vorgang nicht wiederhergestellt werden und das Modell wirkt an der Stelle eingedellt.

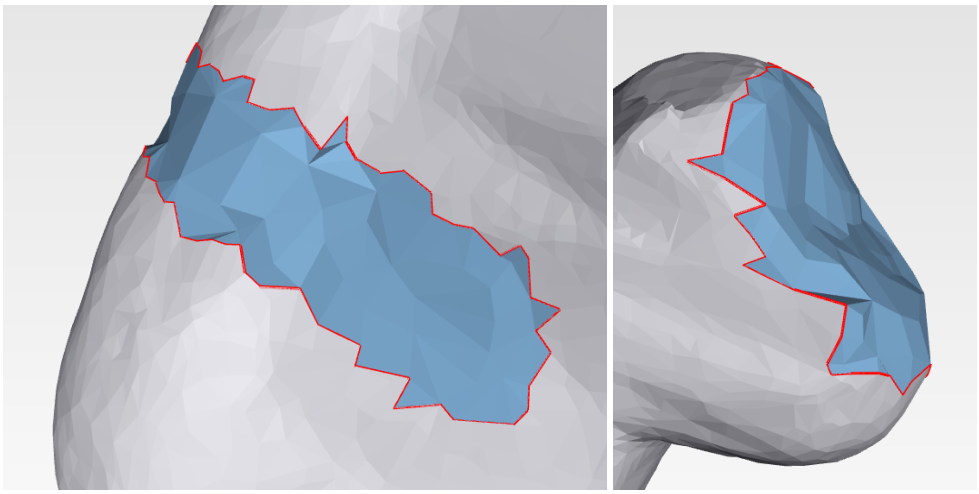

Abbildung 13: Links das Modell *squirrel_hole_back_round.obj* und rechts *squirrel_missing_tip.obj*.

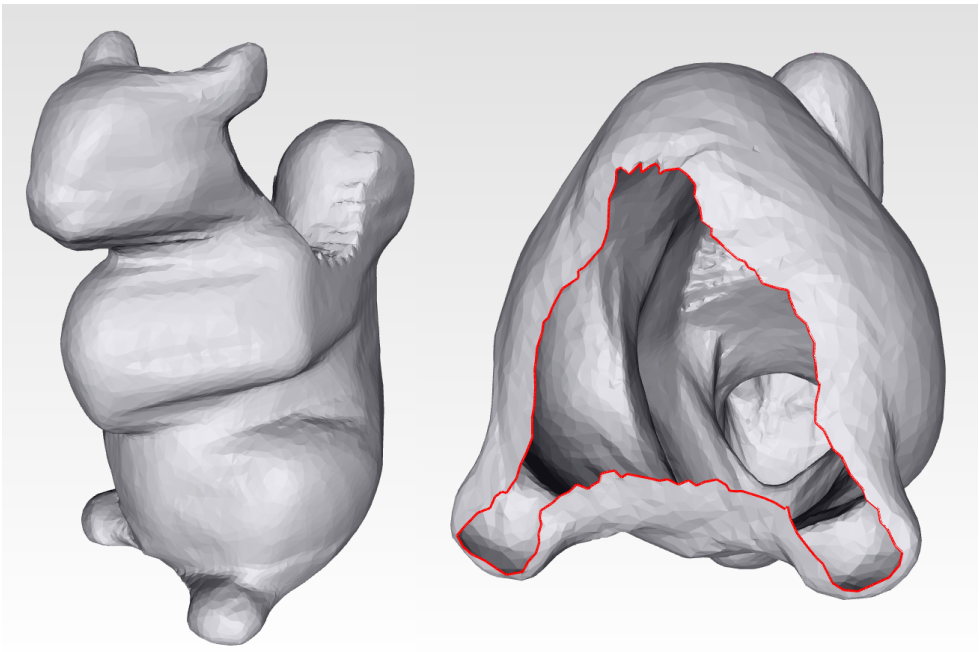

Abbildung 14: Modell *squirrel_hole_bottom_plane.obj*. In rot markiert ist das Loch in der Unterseite.

Die folgenden Zeitmessungen wurden auf diesen zwei Test-Systeme ausgeführt:

- **Desktop**: Intel® Core™ i5-2400 CPU @ 3.10GHz × 4 mit GeForce GTX 560 Ti
- **Laptop**: Intel® Core™ Duo T2390 CPU @ 1.86GHz × 2

In einer ersten Testreihe (Tabelle 1) wurde das Modell *squirrel_hole_bottom_plane.obj* (10.070 Vertices, 19.999 Faces) mit Chromium 28 sowohl auf dem Desktop als auch dem Laptop mit den verschiedenen Füll-Modi getestet. Als Merging-Grenzwert wurde der von der Anwendung vorgeschlagene Wert von 0,289 genommen und der Kollisionstest nur mit der Füllung durchgeführt. Das Finden der Löcher dauerte dabei auf dem Desktop durchschnittlich 102 ms und auf dem schwächeren Laptop 255 ms. In den Tests hat sich gezeigt, dass der iterative Modus der schnellste ist, während der responsive Modus am langsamsten ist. Der parallele Modus liegt zwischen den beiden, wobei die Geschwindigkeit letztlich auch sehr von der gewählten Anzahl an Worker-Prozessen abhängt. So hatte die Ausführungsgeschwindigkeit auf dem Laptop mit nur zwei Kernen nur einen schwachen Vorteil gegenüber der responsiven Variante, während auf dem Desktop mit vier Kernen ein deutlicherer Vorteil beobachtet werden konnte.

System	**Finden [ms]**	**Füll-Modus**	**Füllen [ms]**
Desktop	96	iterative	265
Laptop	275	iterative	584
Desktop	111	responsive	2.124
Laptop	289	responsive	2.855
Desktop	106	parallel (4 Workers)	934
Desktop	95	parallel (2 Workers)	1.212
Laptop	201	parallel (2 Workers)	2.457

Tabelle 1: Testreihe mit dem Modell *squirrel_hole_bottom_plane.obj* und Chromium 28 als Browser. Merging-Grenzwert: 0,289. Kollisionstest: Füllung.

In der nächsten Testreihe (Tabelle 2) wurde der Kollisionstest auf das gesamte Modell inklusive dem jeweiligen Zustand der Füllung ausgeweitet. Hierbei sollte geprüft werden, wie weit der parallele Modus an den schnellen iterativen angenähert werden kann. Die Überlegung dabei war, dass durch den ausgeweiteten Kollisionstest die parallelisierte Überprüfung auf Kollisionen einen Vorteil erhalten müsste. Zwar konnte der parallele Modus in diesem Test nicht den iterativen schlagen, jedoch ist ein Vorteil für mehr Kerne zu erkennen. Während die Ausführung auf dem Laptop (zwei Kerne) fast doppelt so lange dauerte, liegen die Zeiten auf dem Desktop (vier Kerne) nur wenige hundert Millisekunden auseinander. Dies ist zudem ein kleinerer Zeitabstand als im vorherigen Test, in dem nur mit der Füllung geprüft wurde.

System	Füll-Modus	Füllen [ms]
Laptop	iterative	13.825
Laptop	parallel (2 Workers)	25.957
Desktop	iterative	6.849
Desktop	parallel (4 Workers)	7.160

Tabelle 2: Testreihe mit dem Modell *squirrel_hole_bottom_plane.obj* und Chromium 28 als Browser. Merging-Grenzwert: 0,289. Kollisionstest: Gesamtes Modell inklusive Füllung.

In der Testreihe zu Tabelle 3 wurde der Merging-Grenzwert von den empfohlenen 0,289 auf 0,12 gesenkt, was zu einer feineren Füllung führt. Dies bedeutet auch mehr Aufwand für den Kollisionstest und sollte einen Nachteil für den parallen Modus darstellen, da gegen Ende immer größere Datenmengen an die Worker verschickt werden. Getestet wurde nur auf dem Desktop-System mit vier Kernen. Wie erwartet ist der iterative Ansatz wieder schneller mit rund 27 Sekunden, jedoch bleibt die parallele Variante mit optimalen vier Worker-Prozessen nur knapp 2 Sekunden dahinter. Wählt man mehr oder weniger Worker-Prozesse als Kerne zur Verfügung stehen, wirkt sich dies negativ aus. Ein Prozess weniger hatte einen Zeit-Verlust von 10 Sekunden zur Folge, während ein Prozess mehr zu einer Verlängerung um 5 Sekunden führte.

Füll-Modus	Merging	Füllen [ms]
iterative	0,12	27.151
parallel (3 Workers)	0,12	39.745
parallel (4 Workers)	0,12	29.360
parallel (5 Workers)	0,12	34.565

Tabelle 3: Testreihe mit dem Modell *squirrel_hole_bottom_plane.obj* und Chromium 28 als Browser. System: Desktop. Kollisionstest: Gesamtes Modell inklusive Füllung.

In einem Vergleich von Chromium 28 mit Firefox 23 (siehe Tabelle 4) zeigt sich für den iterativen und responsiven Modus nur ein minimaler Unterschied. Im parallelen Modus hängt Chromium jedoch eindeutig Firefox ab (vgl. Tabelle 1, 2).

Browser	Finden [ms]	Füll-Modus	Kollisionstest	Füllen [ms]
Firefox	168	iterative	Füllung	360
Firefox	172	responsive	Füllung	2.500
Firefox	169	parallel (4 Workers)	Füllung	2.044
Firefox	165	parallel (4 Workers)	Modell + Füllung	12.258

Tabelle 4: Testreihe mit dem Modell *squirrel_hole_bottom_plane.obj* und Firefox 23 als Browser. System: Desktop. Merging: 0,289.

7 Zusammenfassung und Ausblick

In diesem Projekt wurde eine Web-Anwendung entwickelt, mit der Löcher in 3D-Modellen gefunden und gefüllt werden können. Das gefüllte 3D-Modell kann anschließend wieder in ein für 3D-Drucker übliches Format – namentlich STL – exportieren werden. Für das Finden der Löcher wurde eine HalfEdge-Datenstruktur gewählt. Für das Füllen wurde ein Advancing Front-Algorithmus implementiert und zusätzlich der Kollisionstest im Verfahren mit Web Workern parallelisiert. Wie sich in der Evaluation gezeigt hat, ist die Advancing Front-Implementierung gut geeignet für Löcher, die ungefähr auf einer Ebene liegen, weist jedoch Schwächen auf, wenn die Löcher Teil einer stark unebenen Oberfläche sind.

Die Advancing Front-Regel 3 in diesem Projekt ist mehr eine Notlösung, da die neu erzeugten Vertices nicht immer optimal gesetzt werden. Wie in [8] oder [9] behandelt, könnte ein besseres Verfahren zur Bestimmung des neuen Vertexes implementiert werden, wie z.B. *Moving Least Squares*. Für entstandende Füllungen könnten zudem Optimierungsverfahren eingesetzt werden, wie es z.B. in [1] geschieht auf Basis von *Poisson*-Gleichungen.

In diesem Projekt werden bereits Web Workers eingesetzt, jedoch entsteht für kleine Modelle kein Vorteil. Es könnte überprüft werden, welche Aufgaben sich noch parallelisieren lassen könnten, oder wie sich die existierende Parallelisierung noch verbessern lassen könnte, z.B. indem der Kommunikations- und/oder Serialisierungsaufwand stärker minimalisiert wird.