**Web-based processing of**

**3D geometries for 3D printing**

Sebastian Dorn

August 26, 2013

In the summer semester of 2013, a web application was developed at RheinMain University of Applied Sciences as a Master's project to find and subsequently fill holes in 3D models. A HalfEdge data structure is used and for the filling process the Geometry-based Advancing Front algorithm is used for filling. As an optimization measure, an attempt was also made to parallelize part of the algorithm with Web Workers.

**Table of Contents**

1 Introduction 3

2 3D in the browser 3

3 Detection of holes 4

4 Filling holes 4

5 Implementation 6

5.1 Finding holes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

5.2 Advancing Front . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

5.3 Web Workers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

5.4 Different modes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

5.5 Model export . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

6 Evaluation 18

7 Summary and outlook 22

**1 Introduction**

The larger project framework into which this work will be integrated is the construction of a replicator. Objects are to be scanned by a 3D scanner and printed by a 3D printer.

However, the scanned models are rarely directly ready for printing and must therefore be processed beforehand. A prerequisite for printing is that models must be waterproof - i.e., there must be no holes in their surface.

Such a hole is generally located the 3D scanner is not able to detect holes in the standing surface of the model. The aim of this project is to find holes in 3D models and to fill them. The model processed in this way must then be exported back into a format that is common for 3D printers. format commonly used by 3D printers - typically the STL format.

The whole thing is to be made available as a platform-independent web application. Various methods already exist for filling holes and further adapting the generated surfaces to their surroundings to make them appear more natural. For example, in [1] the hole is first filled with the Advancing Front algorithm and then further adjusted to the surrounding structure of the model based on Poisson equations. In [2], geometry is also used, but radial basis functions are used. A volume-based method is used in [3], where an octree data structure is used for both finding of holes, as well as matching to the environment.

**2 3D in the browser**

WebGL is a specification for rendering 3D content in the browser. Developers can use WebGL via a JavaScript API, provided that the browser used supports WebGL. Plug-ins or similar extensions are not required for this. With Firefox 4 and Chromium 8, the first WebGL support was introduced in these browsers [4]. The current versions, Firefox 23 and Chromium 28, are both in the available WebGL features, as well as the performance provided unrestricted usable as a 3D environment. The handling of the WebGL API is similar with that of OpenGL. In order to facilitate the development it is therefore recommended to use a WebGL library. In this project, three.js [5] was chosen because it is one of the more widely used libraries and has sufficient documentation.

**3 Detection of holes**

The defining property of holes is that every edge that belongs to a hole belongs to only one face. In order to detect these affiliations, a HalfEdge mesh is a suitable data structure. A HalfEdge object holds as information an output vertex; a Face to which it belongs; the next HalfEdge, i.e. the next edge; and the opposite HalfEdge. Thus, one has information about all edges and which Faces they form. With the knowledge that an edge belongs to a hole, if it belongs to only one face, the holes can then be found in the model (see Figure 1). This was done using the Java implementation of a colleague in the Computer Vision and Mixed Reality group, which only needed to be ported to JavaScript.

라인, 도표, 종이접기, 디자인이(가) 표시된 사진

자동 생성된 설명

Figure 1: Marked in Blue (left): Edge belonging to two faces.

Red (right): Edge that belongs to only one Face and is therefore part of a hole.

**4 Filling holes**

Algorithms for filling holes can largely be sorted into one of two categories: Geometry-based methods and Volume-based methods.

**Geometry-based methods** calculate new points for filling based on the surface of the model. A typical method of this type is Advancing Front, where the hole is reduced iteratively (cf. [1, 6]).

**Volume-based methods,** on the other hand, work with the volume of the model. The shape of the model can be derived from the volume and holes can be covered according to the calculated shape (cf. according to the calculated shape (cf. [3]).

For this project, the geometry-based Advancing Front method was chosen. The starting point is the shape of the hole found - i.e. all edges that border the hole. This is the initial front, which is subsequently iteratively reduced until the hole is finally closed. The reduction is achieved by applying a corresponding rule to each vertex of the front, which is based on the angles of neighboring vertices.

라인, 도표이(가) 표시된 사진

자동 생성된 설명

Figure 2: Angle q between vectors u0 and w0

After the initial front of a hole is determined, the angles of all vertices are calculated first. For the angle of a vertex, in addition to the vertex itself the two neighbors are needed. Let u, v and w be vectors, where u is the neighbor of v and v is the neighbor of w (see Figure 2). The angle q of v is then calculated as described in equation 1.

텍스트, 폰트, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명

From the obtained angles, select the smallest one and start filling the hole at its vertex with the filling of the hole. For this purpose, depending on the size of the angle, one of three rules depending on the size of the angle:

1. Rule for q ≤ 75°
2. Rule for 75° *< q ≤* 135°
3. Rule for 135° *< q*

In the case of rule 1, only the angle needs to be closed by making a new connection from u to w becomes part of the new front. The vertex v is thus excluded from the front (see Figure 3).

In the case of rule 2, a new vertex is needed. For this it is useful to set this vertex bisecting and then to move it to a similar distance as u to v or w to v (see figure 3).

There are different approaches for rule 3, but there is always at least one new vertex is created. In [6], for example, it is implied that two new vertices are added, dividing the angle into thirds. In this project, a new face is added to the edge from v to w (see Figure 3).

라인, 도표, 디자인이(가) 표시된 사진

자동 생성된 설명

Figure 3: Changes at the front for rule 1, 2 and 3 (from left to right). In black is the old course is shown, while red represents the new front (slightly shifted downwards so that it is easier to see).

**5 Implementation**

Due to the JavaScript features used, a reasonably up-to-date browser is required. Firefox 23 and Chromium 28 were used during development. WebGL and Web Workers require at least Firefox 4 or Chromium 8, but for performance reasons a recent version should be chosen. The JavaScript framework used to handle WebGL was three.js (r60). Data objects like THREE.Vector3 were then further used for calculations during the filling process.

**5.1 Finding holes**

Among the test models were two with problematic structure. The model puppe.obj had vertices which were not part of a face, and both the model puppe.obj as well as adonis.obj had vertices which, among other things, formed faces with themselves.

In the 3D representation, these erroneous areas were not visible, but in the hole search they sometimes led to difficulties. In a first step, models are checked for such faulty vertices after loading into the application and then corrected accordingly. This means that faulty vertices and the faces affected by them are removed. In the representation of the models nothing changes thereby and the hole search yields correct results.

To search for holes, a HalfEdge mesh is first set up. Thereby all required information is available, such as the connections of the vertices to each other. Of interest are those vertices which are part of a hole. Between two such edge points lie an edge. Edge points can be recognized by the fact that they are only part of one face, while all other edges belong to two faces.

Any one of the existing edge points is selected as the starting point. This edge point is the starting point for the first hole. From this edge point we go to the next adjacent edge point - the HalfEdge Structure gives us this information. It is jumped as long as from edge point neighbor to the next neighboring edge point until the starting point is reached again. Thus the first hole is found. The points visited so far are marked, so that they are not observed in the further course. After that the next unmarked edge point is chosen and the next hole is walked. This is repeated until there are no more unmarked edge points.

스크린샷, 다채로움, 그래픽, 그래픽 디자인이(가) 표시된 사진

자동 생성된 설명

Figure 4: Marked with white arrows: Boundary points that belong to several holes (highlighted in color) or belong to the same one several times.

So far, however, no attention is paid to a special case that can cause problems in hole detection. This can happen, if a boundary point belongs to more than one hole or several times to the same one (see figure 4). These points will be referred to as "multi-edge points" in the following. In order to handle these cases appropriately, the previous procedure must be adapted. Multi-edge points must not be marked as visited immediately after a visit, because they must be visited again for another hole. A multi-edge point may only be marked after all its neighboring edge points have been marked.

If you encounter a multi-edge point while walking the edge points of a hole, you find the next edge point of the current hole by calculating the angles between the previously visited point, the current multi-edge point and all potential next edge points. Thereupon one selects the edge point with which the smallest angle is formed (see Figure 5).

도표, 라인, 디자인, 그래프이(가) 표시된 사진

자동 생성된 설명

Figure 5: Determination of the next edge point after a multi-edge point via the angles

**5.2 Advancing Front**

During the execution of the Advancing Front algorithm, only one hole at a time is only one hole is considered and filled. Three lists are maintained: front, which contains the currently present in the front; filling.vertices, which contains the vertices currently present in the filling; and filling.faces, which contains the faces currently present in the filling. A face contains only the indices of its vertices in the filling.vertices list. If the filling is later merged with the rest of the model, these indices must be adjusted accordingly.

**Initial angles.** First, the angles of the vertices must be calculated. To calculate the angle to a vertex, one needs the vector v at this position, the previous vector vp, as well as the following vector vn within the front of the hole. First, the vectors must be moved relative to v to the origin. If not mentioned otherwise, the vectors in the code are THREE.Vector3 instances.

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

For angle calculation and conversion of radians to degrees, you can use the functions of three.js can be used.

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

For the Advancing Front algorithm we need the angle directed to the hole. So far, however, this calculation only gives us the smaller angle, which can also be directed in the other direction. To correct this afterwards, we first calculate the cross product of vp and vn and shift the resulting vector back into the model.

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

If the cross vector is shorter - and thus closer to the origin - than the angle vector v, the opposite angle is taken:

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

Unfortunately, this approach is not one hundred percent reliable and so, especially in the case of holes whose vertices hardly lie on a plane, incorrect angle values can occur, leading to erroneous fill results.

The angles and related information are stored in instances of the Angle class. An Angle object has attributes for the angle in degrees, the three associated vectors, the previous Angle in the front, and the next Angle. The attributes attached to structure, reminiscent of a doubly-linked list, is later necessary to update angles to update angles modified by the filling process.

**Heap procedure.** The angles created in the previous step are collected in a heap. Angle objects are inserted as keys with their angle size. Then the keys are sorted in ascending order, starting from the smallest angle in the front. After each rule applied, the heap is sorted again, so that the smallest angle is always processed first. This is desirable because rules 1 and 2 produce more stable results than rule 3, so bringing forward the smaller angles produces a better overall fill.

After applying one of the Advancing Front rules based on the angle, the neighbors of the angle must be updated because the filling process changes the front. The vertices of the previous and subsequent Angle are updated, the corresponding angle is recalculated and the angle is re-sorted in the heap.

**Rule 1** is applied to angles with q ≤ 75°. In this case, for a new Face, only a connection between vp and vn is added, so that the new Face in the fill consists of the vertices vp, v and vn (see Figure 6). From the front, the vertex v must be removed.

**Rule 2** applies to angles with 75° < q ≤ 135°. In this case a new vertex is created. To do this, the vectors are moved relative to v to the origin and a plane is created. Bisecting the angle, a new point is then set within the plane, where the distance of the new point from v is the mean value of the distances from vp and vn to v (see Figure 7).

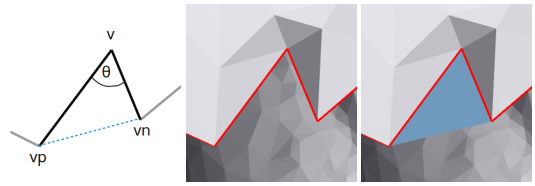


Figure 6: An angle before and after applying rule 1.

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 );

The new vertex vNew then replaces vertex v in the front. In the fill, two new faces are added: (v, vp, vNew) and (v, vNew, vn)



Figure 7: An angle before and after applying rule 2.

**Rule 3** applies to angles with 135° < q. For such angles a new vertex, vNew is inserted between v and vn forming a new face (vn, v, vNew). The new Vertex is determined by first computing the cross product c1 of vn and vp. In general, this cross product points inward into the Model from the hole when looking at the hole from the outside.

1 var vnClone = vn.clone().sub( v ),

2 vpClone = vp.clone().sub( v );

3 var c1 = vnClone.clone().cross( vpClone ).normalize();

Next, the cross product c2 of c1 and v is calculated, which points approximately into the hole. Starting from vn and c2, a plane can now be drawn on which a point vNew is set with the same length as vn(see Figure 8). In this last step, rule 3 is similar to rule 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] );

The function Utils.keepNearPlane in line 11 tries to prevent too much bulging in actually plane holes. Using the variance - obtained from the vectors vp, v and vn - the position of the new vertex is adjusted.

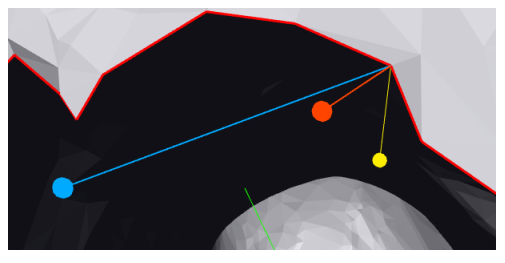


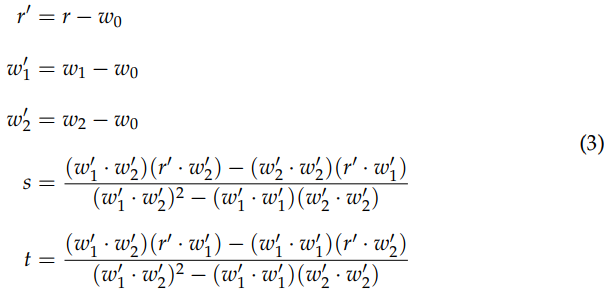
Figure 8: During the application of rule 3: c1 (yellow), c2 (blue), vNew (red).

**Collision test.** For each new triangle we have to check if it does not intersect with another area of the existing fill or model. For this purpose we check for each side of the new triangle if it intersects with the area of one of the existing triangles. Here the implementation follows the mathematical description from [7]. The new triangle consists of the vectors v0,v1 and v2 and is tested with a triangle with the vectors w0, w1 and w2.

In a first step, formula 2 is used to check whether the straight line between v0 and v1 intersects the plane in which the other triangle lies. Here n is the normal of the plane. For a value 0 < r < 1, r 2 R there is an intersection point at the position v0 + r - (v1 - v0).



If there is no intersection with the plane, there is no collision with the triangle. Otherwise, it must be checked whether the straight line intersects not only the plane, but also exactly the area of the triangle. This is done in formula 3.



If s, t ≥ 0 and s + t = 1, then there is an intersection of the line with the triangle. Provided that no intersection point has been found yet, the test must be repeated again with the straight line from v1 to v2 - except for rule 1, because here only one side of the triangle is new.

Because of a found collision a new face must be rejected. However, the angle in the front remains unchanged, so the Angle object must be returned to the heap. To avoid that after sorting the same Angle whose result was just rejected is processed again immediately, a flag Angle.waitForUpdate is set to true. Until the angle of the Angle changes - by changing the neighboring vertices - this flag remains.

By default, for performance reasons, the collision test is performed only with the faces of the fill, since the risk of collisions is also highest here. Optionally, however, the user interface can be switched to a complete test with the entire model.

**Merging**. Two vertices that are close to each other must be merged. The distance between the two vertices before they are merged is calculated individually for each hole. The average distance of all neighboring vertices of the hole is suggested as a limit value.

A lower limit value for the distance results in a finer filling (more faces), while a high threshold value results in a coarser fill. This difference is illustrated in Figure 9

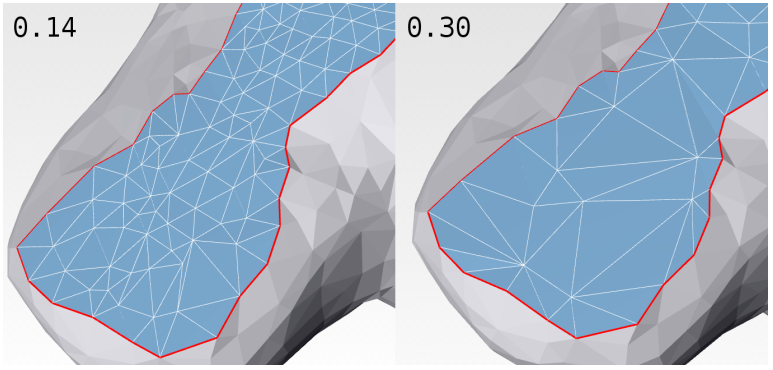


Figure 9: Comparison of merging limits: 0.14 (left) and 0.30 (right).Recommended by the application for the hole were 0.289.

A check whether vertices can be merged is performed each time a rule is applied. Only the new vertex is compared with its two neighbors in the front - for rule 1 the merging is omitted, since no new vertex was created. If the distance is below a given threshold, the neighboring vertex t is merged with the new vertex v, which means that vertex t is removed (see Figure 10).

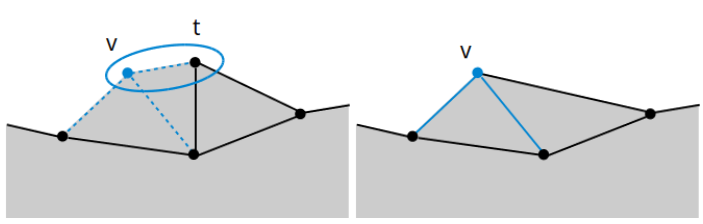


Figure 10: Merging: The distance between the circled points in the left image is below the threshold and t is therefore merged with v. The result is shown in the right image. The result can be seen in the right image.

Faces in filling, in which the vertex t was a component, must then be updated. The new vertex v now takes its place. Since the faces store only the indices of their vertices as data, all those indices must also be reduced by the value 1, which have an index larger than the removed vertex have. Likewise t must be removed from the front list and all AngleObjects in the heap, which are formed among other things by the Vertex t, must be updated.

**5.3 Web Workers**

Web Workers are a relatively new feature that allow JavaScript code to be executed as background processes in the browser. On systems with more than one CPU core, work can be parallelized in this way. For this project, the collision test was chosen for parallelization. Each worker process receives a different subset of the faces to be tested.

Workers can be created in two ways: Firstly, as a separate JS file whose path is then specified, and secondly, as an inline worker when the code is written to the index.html file. However, since the project should also be executable without a server, only inline workers can be used. The code for the worker is inserted as usual within a <script> tag in the HTML file:

1 <script id="worker-collision" type="javascript/worker">

2 // ... JavaScript-Code ...

3 </script>

A new worker is then created by reading the inline code, wrapping it in aBlob object, and passing it as an object to the worker.

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 );

Then, however, the worker still needs access to other classes, such as Utils or THREE. For this purpose, the importScripts function exists specifically for the worker processes, but it requires an absolute path to the JavaScript file.To keep the application flexible, this cannot be set directly.First, the path is determined outside and then sent to the worker.

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 } );

Within the worker, messages are handled and scripts are included as follows:

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 )

However, a very direct implementation in which a new worker process was started for each test was much too slow. An initial tweak was therefore to create a one-time pool of fixed-size worker processes. For the collision test, the data to be tested is then simply added to the pool and free workers process it. This is realized by the WorkerManager class. A pool size or number of workers corresponding to the number of CPU cores has proven to be useful. On a computer with 4 cores (Intel i5), 4 workers are started.

However, the collision test (at this point only with filling) was still 8× slower than the non-parallelized implementation. The reason is the communication overhead, because message objects must first be serialized and de-serialized again on the worker side. This happens automatically when postMessage is called. The larger the padding to be tested, the larger the message. An unexpected optimization was to convert the data itself to JSON beforehand.



In the worker, the JSON must also be unpacked again:

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

With this small change, the parallel implementation is only about 5×slower than the iterative one in terms of the fill collision test. However, as will be seen later in the evaluation, the parallel version approaches the iterative one when testing with the entire model. The advantage is that there is hardly any additional communication effort, because the model does not have to be sent to the worker again for each test. Since the model does not change during the filling process, it is sufficient to send it to the workers once at the beginning and later to transmit only the face to be tested.

**5.4 Different modes**

The Advancing Front algorithm can be executed in three different modes: iterative, responsive and parallel. The intermediate steps and results of the filling process are always the same, but the modes differ in the internal flow.

With iterative, the algorithm is executed in a while loop until it terminates. This mode is the fastest, but during this time the browser window does not react to any input or DOM changes triggered by JavaScript (e.g. updating the progress bar).

In **responsive**, the window remains open to input - the model can be rotated, for example, and the progress bar changes. This is possible because this approach is not based on a big loop, but on function calls. There is a main function that is called again and again at the end - which would correspond to a run in a loop. These function calls are managed and processed by the browser internally on a stack. By breaking the one large loop from iterative mode into many function calls, it is now possible to process GUI events in between (see Figure 11). However, this comes at the expense of the Fill process, which is why this mode is significantly slower.

The **parallel** mode is similar to responsive for now. However, the collision test is parallelized via Web Workers, which provides a speed boost in execution. However, how much time is gained also depends on factors such as the number of CPU cores and the number of faces that need to be tested.

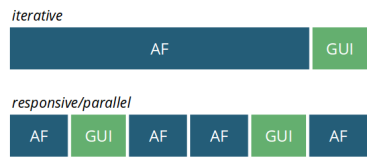


Figure 11: Simplified internal flow in the browser. The iterative mode blocks the processing of GUI events.

**5.5 Model export**

The export supports the two formats OBJ and STL - the latter is usually used in 3D printing. Since it is not possible with JavaScript to write files directly to a storage medium, the generated export can only be offered for download or - since the application is executed locally - aSave dialog can be called. In JavaScript, this is implemented as follows:

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 line 1 the model is converted to OBJ. The variable exportData contains the whole OBJ content as string. In line 2 the OBJ text is packed into a blob. In line 3 ff. a HTML link is created and gets the attribute download.The value for the attribute becomes the filename. In the href attribute the file content is specified. To finally trigger the save dialog, the click event is fired via download.click() in line 10.

For the STL export, all coordinates must also be rewritten from float values to scientific notation. Instead of -0.02648, -2.648e-2 must be written. For this the function Utils.floatToScientific was written, which treats a float value as a string and rewrites it accordingly.Since this must be done for each individual value, the STL export is comparatively slower than the export for OBJ.

**6 Evaluation**

The models used here have an approximate number of 10,300 verticesand 20,400 faces. Larger models with a face count of 100,000 and more unfortunately overload the application and make processing impossible.

The Advancing Front algorithm implemented in this project works particularly well with holes that are approximately on a plane, such as in TestModel squirrel\_bottom\_hole\_plane.obj (Figure 14). The method mentioned in chapter 5.2 for preventing strong curvatures in rule 3 also results in relatively flat fills, as shown in figure 12.

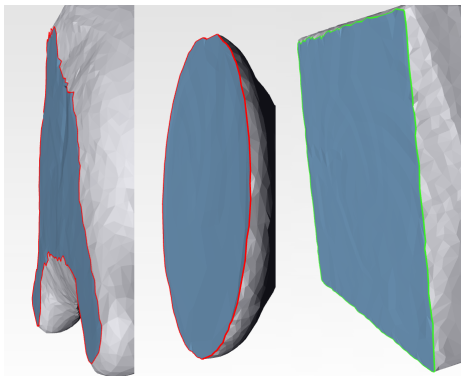


Figure 12: The models squirrel\_hole\_bottom\_plane.obj, diana.obj and adonis.obj(f.l.t.r.) with filled bottom surface.

Holes that extend over a rounded surface, for example, can also be filled. However, the filling often appears more uneven than the surrounding area, as can be seen on the left in Figure 13. Here, there would still be room for optimization methods following the filling process.

More problematic are holes where larger, uneven sections are missing. These holes are particularly susceptible to the angle correction problem described in chapter 5.2. Even if the filling process is successful, it is not a successful reconstruction but only a temporary filling. As can be seen on the right in Figure 13, the rounding of the missing tip of the squirrel's tail could not be restored by the filling process and the model appears dented at that point.

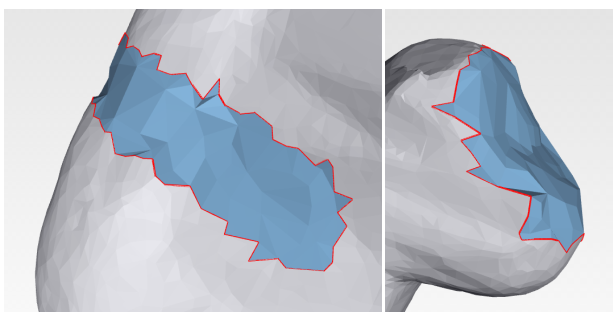
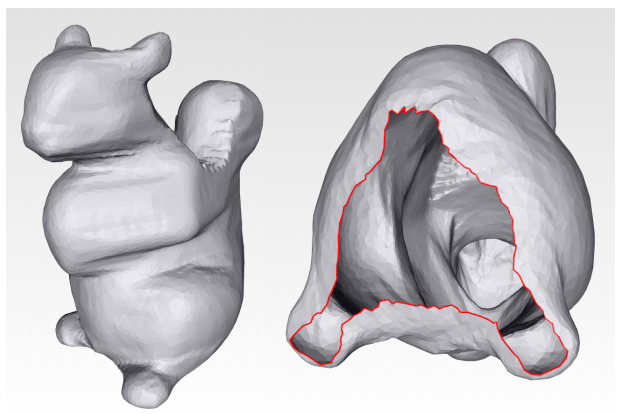


Figure 13: The model squirrel\_hole\_back\_round.obj on the left and squirrel\_missing\_tip.obj on the right.

  
Figure 14: Model squirrel\_hole\_bottom\_plane.obj. Marked in red is the hole in the bottom.

The following time measurements were performed on these two test systems:

• Desktop: Intel® Core™ i5-2400 CPU @ 3.10GHz × 4 mit GeForce GTX 560 Ti

• Laptop: Intel® Core™ Duo T2390 CPU @ 1.86GHz × 2

In a first series of tests (Table 1) the model squirrel\_hole\_bottom\_plane.obj(10,070 vertices, 19,999 faces) was tested with Chromium 28 both on the desktop and the laptop with the different fill modes. The application's suggested value of 0.289 was taken as the merging threshold and the collision test was performed with the fill only. Finding the holes took an average of 102 ms on the desktop and 255 ms on the weaker laptop. The tests showed that the iterative mode is the fastest, while the responsive mode is the slowest. The parallel mode lies between the two, although the speed ultimately also depends very much on the number of worker processes selected. Thus, the execution speed on the laptop with only two cores had only a weak advantage over the responsive variant, while a clearer advantage could be observed on the desktop with four cores.

|  |  |  |  |
| --- | --- | --- | --- |
| System | Find [ms] | Fill mode | Fill [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 |

Table 1: Test series with the model squirrel\_hole\_bottom\_plane.obj and Chromium 28 as browser. Merging threshold: 0.289. Collision test: Fill.

In the next test series (Table 2), the collision test was extended to the entire model including the respective state of the filling. This was to test how far the parallel mode can be approximated to the fast iterative one. The reasoning behind this was that the extended collision test should give the parallelized check for collisions an advantage. While the parallel mode could not beat the iterative one in this test, an advantage for more cores can be seen. While the execution on the laptop (two cores) took almost twice as long, the times on the desktop (four cores) are only a few hundred milliseconds apart. This is also a smaller time gap than in the previous test, which only tested with the fill.

|  |  |  |
| --- | --- | --- |
| System | Fill mode | Fill [ms] |
| Laptop | iterative | 13.825 |
| Laptop | parallel | 25.957 |
| Desktop | iterative | 6.849 |
| Desktop | parallel | 7.160 |

Table 2: Test series with the model squirrel\_hole\_bottom\_plane.obj and Chromium 28 as browser. Merging threshold: 0.289. Collision test: Total Model including fill.

In the test series for Table 3, the merging threshold was lowered from the recommended 0.289 to 0.12, resulting in a finer fill. This also means more effort for the collision test and should be a disadvantage for the parallel mode, since larger and larger amounts of data are sent to the workers towards the end. Only the desktop system with four cores was tested. As expected, the iterative approach is again faster with around 27 seconds, but the parallel variant with optimal four worker processes is only just under 2 seconds behind. Selecting more or fewer worker processes than cores available has a negative effect. One less process resulted in a time loss of 10 seconds, while one more process led to an increase of 5 seconds.

|  |  |  |
| --- | --- | --- |
| Fill mode | Merging | Fill [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 |

Table 3: Test series with the model squirrel\_hole\_bottom\_plane.obj and Chromium 28 as browser. System: Desktop. Collision test: Entire model including fill.

A comparison of Chromium 28 with Firefox 23 (see Table 4) shows only a minimal difference for iterative and responsive mode. In parallel mode, however, Chromium clearly outperforms Firefox (see Table 1, 2).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Browser | Find [ms] | Fill mode | Collision test | Fill [ms] |
| Firefox | 168 | iterative | fill | 360 |
| Firefox | 172 | responsive | fill | 2,500 |
| Firefox | 169 | parallel (4 workers) | fill | 2,044 |
| Firefox | 165 | parallel (4 workers) | model + fill | 12,258 |

Table 4: Test series with the model squirrel\_hole\_bottom\_plane.obj and Firefox 23 asBrowser. System: Desktop. Merging: 0.289

**7 Summary and Outlook**

In this project, a web application was developed that can be used to find and fill holes in 3D models. The filled 3D model can then be exported back into a format commonly used by 3D printers, namely STL. For finding the holes, a HalfEdge data structure was chosen. For filling, an Advancing Front algorithm was implemented and, in addition, the collision test was parallelized in the procedure with web workers. As shown in the evaluation, the Advancing Front implementation is well suited for holes that lie approximately on one plane, but has weaknesses when the holes are part of a strongly uneven surface.

Advancing Front rule 3 in this project is more of a stopgap, since the newly created vertices are not always optimally placed. As discussed in [8] or [9], a better method for determining the new vertex could be implemented, such as Moving Least Squares. For resulting fills, optimization methods could be used, e.g. as done in [1] based on Poisson equations.

In this project Web Workers are already used, but there is no advantage for small models. It could be examined, which tasks could still be parallelized, or how the existing parallelization could be improved, e.g., by minimizing the communication and/or serialization expenditure more strongly.

**Literatur**

[1] Zhao, Wei and Gao, Shuming and Lin, Hongwei, “A robust hole-filling algorithm for triangular mesh,” The Visual Computer, vol. 23, no. 12, pp. 987–997,2007.

[2] Wu, Xiao J and Wang, Michael Y and Han, B, “An automatic hole-filling algorithm for polygon meshes,” space, vol. 10, p. 2, 2008.

[3] Podolak, Joshua and Rusinkiewicz, Szymon, “Atomic Volumes for Mesh Completion.,” in Symposium on Geometry Processing, pp. 33–41, Citeseer, 2005.

[4] Alexis Deveria, “Can I use. . . WebGL.” http://caniuse.com/#search=WebGL.[2013-08-05].

[5] Ricardo Cabello, “three.js – JavaScript 3D library.” http://threejs.org/. [2013-08-05].

[6] George, Paul Louis and Seveno, Eric, “The advancing-front mesh generation method revisited,” International Journal for Numerical Methods in Engineering,vol. 37, no. 21, pp. 3605–3619, 1994.

[7] Dan Sunday, “Intersections of Rays and Triangles (3D).” http://geomalgorithms.com/a06-\\_intersect-2.html. [2013-08-18].

[8] Tekumalla, Lavanya Sita and Cohen, Elaine, “A hole-filling algorithm for triangular meshes,” School of Computing, University of Utah, UUCS-04-019, UT, USA,2004.

[9] Kalbe, Thomas and Fuhrmann, Simon and Uhrig, Stefan and Zeilfelder, Frankand Kuijper, Arjan, “A new projection method for point set surfaces,” in Eurographics 2009-Short Papers, pp. 77–80, The Eurographics Association, 2009.