



Software Tool for Interactive Design of Customized Hand Splints

Adam Gorski¹ , O. Remus Tutunea-Fatan² , Louis M. Ferreira³ 

¹Western University, agorski3@uwo.ca

²Western University, rtutunea@eng.uwo.ca

³Western University, lferreir@uwo.ca

Corresponding author: O. Remus Tutunea-Fatan, rtutunea@eng.uwo.ca

Abstract. Physio-therapeutic hand splints are commonly used to stabilize or immobilize certain parts of a hand and to ease discomfort and accelerate injury recovery. Hand splints are commonly made from heat-deformable thermoplastic materials that are custom fit by physiotherapists on patient's hand and that are fabricated via a largely manual and tedious process. To enhance patient's experience, one possibility would be to switch to a digitally-driven design and fabrication process to start with the virtual model of the patient's hand geometry to be acquired through techniques specific to reverse engineering. Nonetheless, the switch from traditional to digitally-driven design/fabrication of hand splints would imply that medical professionals are to learn modeling tools that are specific to reverse engineering and that in turn might become a large deterrent for the adoption of the new technology. To address this, the current study will present a custom-built software tool that is specifically targeting the process of interactive design for hand splints. The tool is capable of generating various types of hand splints with minimal user involvement in a sense that the user can select few basic input parameters to be then passed to the rest of the design process that will be carried out in a more or less automated/software-guided manner. The developed software tool runs standalone without the need of any external software or libraries. Finally, the tool was used to generate hand splints characterized by smaller mean absolute surface deviations when compared with their counterparts fabricated through traditional approaches.

Keywords: Customized Hand Splint, Interactive Design, Software Tool

DOI: <https://doi.org/10.14733/cadaps.2024.976-997>

1 INTRODUCTION

In physiotherapy, hand splints are generally used to immobilize and help support various parts of a hand. According to the traditional fabrication technology, hand splints are typically made from a material that softens at relatively low temperatures such that it can be slowly applied to patient's hand and then cut to a

specific shape. Unfortunately, this process is long and hence it might pose significant difficulties for patients with hand injuries who are also often required to travel from remote areas to places where the specialized healthcare centers are located. By contrast, a digital model of patient's hand can serve equally well as the primary input to the splint design process to be carried out in this case in a fully digital manner. This idea aligns well with the overarching theme of telehealth where in person patient visits become less mandatory [3]. In this scenario - admittedly still in an experimental phase at this time, primarily due to the lack of widespread, accurate, cheap and user-friendly geometry acquisition tools - patients would record a video of their hand - ideally with a personal mobile device - to be then sent to the team of medical professionals to reconstruct the hand geometry by means of specialized photogrammetry-based tools. In this approach, the digital model of the hand will become the primary input to the hand splint design and fabrication cycle.

Along these lines, it is relatively well known that non-contact data acquisition systems can be used to generate relatively accurate digital 3D models. This technology has evolved rapidly and - under certain conditions - even relatively inexpensive devices can be used to acquire data to constitute the basis of high quality digital models [19]. As such, it can be anticipated with a reasonable degree of accuracy that future developments of this technology would eventually eliminate the need for the patient to travel to the clinic where the data acquisition system is located. Nonetheless, at this time, regardless if laser scanning [16], transmissive [5] or photogrammetry-based methods are used for hand scanning purposes, patient trips at the clinic cannot be avoided.

Once the hand geometry is reverse engineered and available in a digital form, physiotherapists are often required to perform small adjustments in order to ensure a better functional fit of the splint to be generated. While this process can be automated to a certain extent by means of scripts/macros to be developed for commercial CAD packages, the versatility and applicability of this approach remain rather limited. Past attempts in this direction [10, 18] were neither user friendly, interactive or a integrated in a standalone package. Alternate approaches involving measurements were also proposed [13] but their accuracy remains constrained by the number of measurements performed such that numerous fitting checks were required to ensure the appropriate fit between the splint and the hand for which it was created.

To address all these issues, the present study was focused on the development of a *de novo* interactive software tool specialized on the customization of hand splints whose design would be initiated by a virtual model of patient's hand. Furthermore, no external libraries were used to ensure a minimal dependence of third party code that could also be subjected to various licensing restrictions. The fully automated standalone parametric hand splint generator was designed to be intuitive and user-friendly as well as to require a minimal amount of user input/interaction. Depending on the intent, hand splint models could be completed even by inexperienced users in less than half minute owed to the guided design steps. The upcoming sections will detail the modules of the software tool as well as the validation of its output.

2 SOFTWARE MODULES

The software tool was conceived in the form of linear sequence of user interfaces that guide the user through the splint design process. The linearity of the tool ensures that each subsequent stage is unlocked once the previous one is completed. Nonetheless, the user can move back and forth between the unlocked stages and any changes made in the upstream design phases will be automatically propagated to the downstream ones. This architecture of the software tool matches the typical iterative splint generation process used by physiotherapists. The typical phases of the splint design process - graphically presented in Fig.1 are:

1. Geometry loading: Import the tessellated model of the hand - typically acquired through scanning - and verify its dimensional accuracy.
2. Geometry alignment: Align the digital hand model with the principal planes of the world coordinate system either by means of the automatic algorithm available or by means of the manual tools.

3. Set rough splint boundaries: Position three cutoff planes for fingers, wrist and thumb locations; rotate delimiting planes until the desired rough splint geometry is obtained.
4. Refinement of boundary curves: Manually perform fine adjustments of the splint boundaries - converted to parametric curve representations - in order to obtain desired splint boundaries.
5. Final adjustments: Ensure the fit between the splint and patient's hand; add optional embossing as needed. After this phase, the splint geometry can be exported in a triangular mesh format to be fabricated through additive processes.

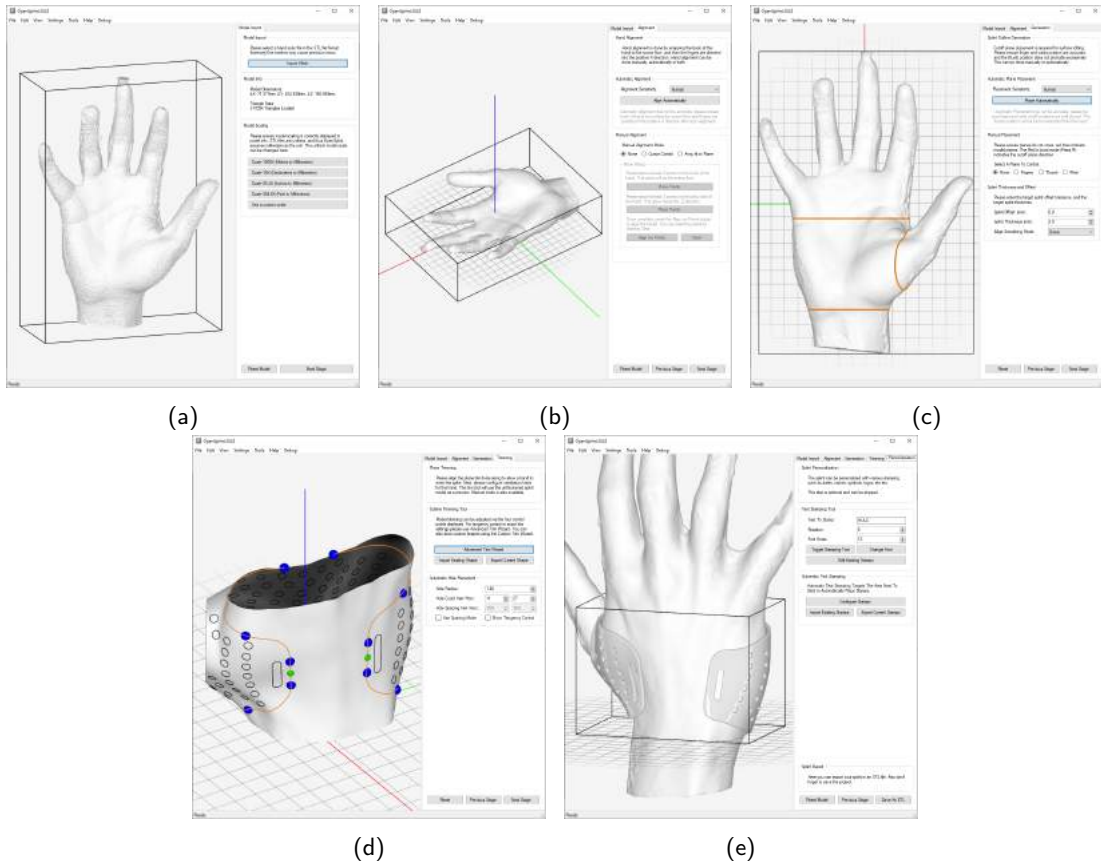


Figure 1: Splint generation phases: (a) geometry loading, (b) geometry alignment, (c) set rough splint boundaries, (d) refinement of boundary curves, and (e) final adjustments.

The software tool provides detailed guiding instructions on the right-hand side of the user interface. In addition to this, the user has the option to toggle axis indicators and maintain the display of the original model. Numerous animations were also incorporated in the tool in order to ensure smooth transitions between modules and to augment the overall experience of the user throughout the splint design process.

2.1 Geometry Loading

The core of this phase consists of a mesh importing tool capable to load a tessellated representation of the hand geometry, typically acquired through scanning. While various triangular mesh formats could be used, the

software tool was designed to load STL files whose dimensions were assumed to be in millimeters. Some basic checks are performed to ensure the accuracy of the model scale and user is prompted to verify the size of the models that appear to be inaccurate. After data import, model size and triangle count are displayed (Fig. 2). After that, the user is asked to verify once again the dimensions of the model and in case that adjustments are needed the model can be scaled via predefined formats or a custom scale.

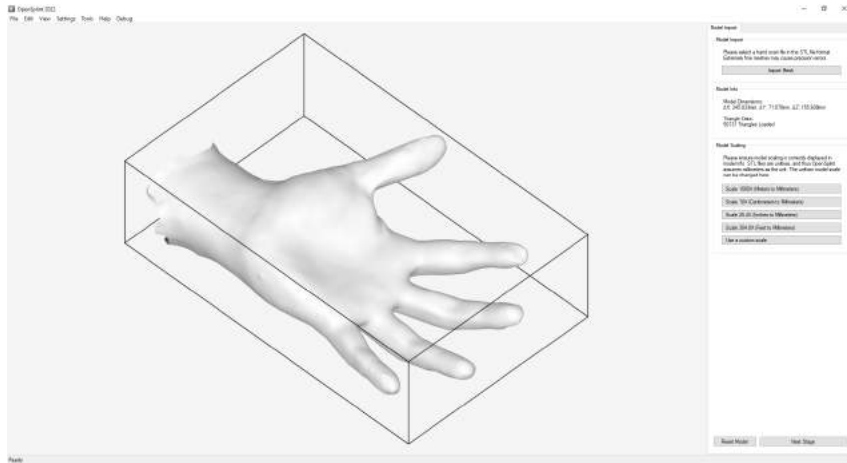


Figure 2: Geometry loading.

2.2 Geometry Alignment

Aligning the model involves adjusting the mesh such that the back of the hand is aligned with the horizontal plane of the coordinate system in such a way that fingers are pointing towards the positive X-axis. This can be achieved either automatically - by means of a curvature analysis algorithm - or manually, through several different methods. For manual alignment, the hand geometry can be rotated by means of a three-axis real or local Euler alignment tool. Alternately, marker points can be placed on specific areas of the hand. All three types of alignment can be observed in Fig. 3.

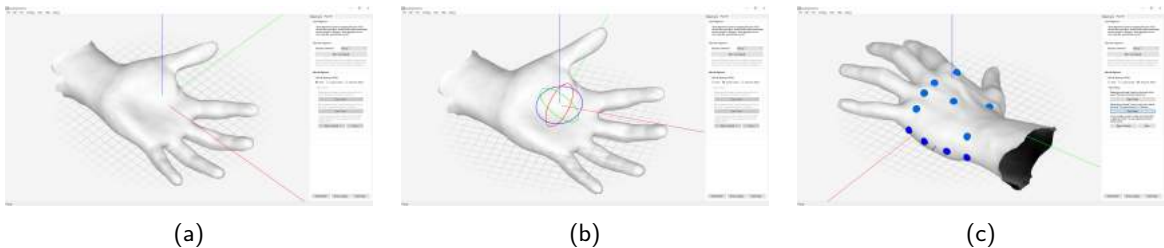


Figure 3: Model alignment options: (a) automatic, (b) Euler angle-based, and (c) best fit point-based.

2.3 Set Rough Splint Boundaries

Physiotherapists rely on the quick trimming stage to rapidly eliminate unimportant areas. This is achieved by defining three points of interest on the hand through the application of three cutoff planes. These points include the initial position of the fingers, the location of the wrist, and the position of the thumb. The repositioning of the fingers and wrist is visible in Fig. 4b, along with both local and global coordinate support provided by the interaction system. As a result, a region of interest is generated at the center of the hand/palm. To accomplish this, the user is to select a cutoff plane and move it to a designated location where Euler rotation controls can be used for fine adjustments of the plane orientation. An automatic positioning algorithm is also available (Fig. 4a). This approach analyzes the curvature of the mesh to determine regions of interest in order to position the trimming planes.

For this phase, the typical recommendation would be to commence with automatic positioning and subsequently fine-tune the planes manually. This technique is better because the software tool solely focuses on the identification of the curvature landmarks, rather than regions of injury that constitute the main target of the treatment plan.

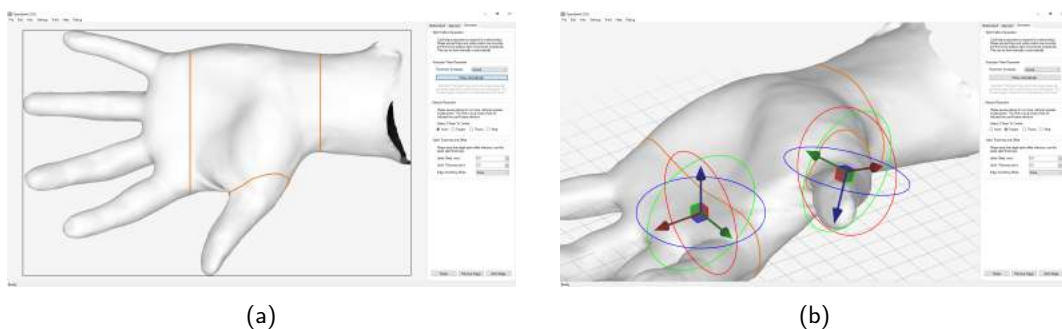


Figure 4: Trimming/cutoff plane positioning options: (a) automatic and (b) manual.

During this phase, the user is prompted to specify both the desired thickness of the splint and the acceptable air-gap tolerance between the hand and the splint. In addition, the software requests information regarding the preferred edge smoothing modes, which can be either no edge smoothing or domed edges, and the user is given the option to select between them.

2.4 Refinement of Boundary Curves

This phase requires the user to adjust the trim lines by means of points located on the previous boundaries (Fig. 5). By manipulating the control points on the splint cutout, the user can customize the trim curves to provide varying degrees of support to different areas of the hand according to patient's specific requirements. Additionally, the user can modify the tangency power and add or remove control points as necessary. The underlying mathematical representation of the trim lines is that of composite Bezier curves. This representation offered sufficient design flexibility such that the use of NURBS was not warranted.

In this phase, the user selects a method for hole placement and adjusts the slots accordingly. The software has the capability to vertically and horizontally place a specified number of holes, or place holes at a specified distance apart both horizontally and vertically. Additionally, the hole radii can be modified as per requirement. Another noteworthy aspect is the special slot adjustment, which allows users to add or remove slots in order to adjust the splint tightness at different hand positions. The slot length can also be adjusted to accommodate various types of straps, including pinned straps. The slots also offer directional, length, and position locking,

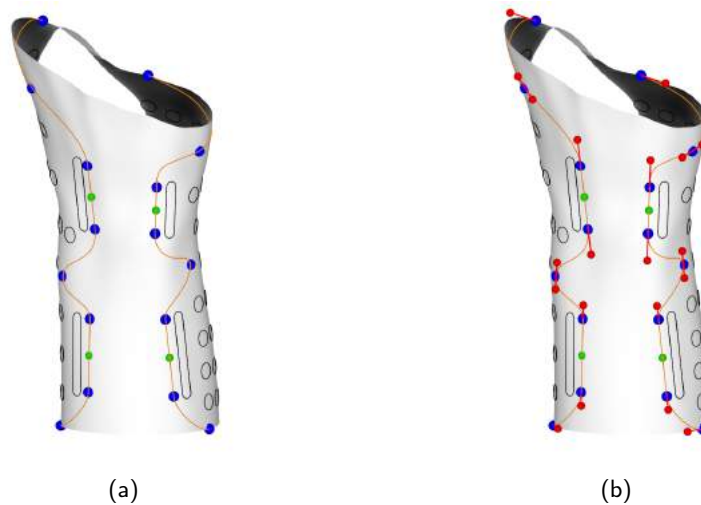


Figure 5: Fine tuning of splint boundary curves: (a) basic and (b) advanced trimming.

ensuring that the slot length follows the adjustment point distance and minimizes the directional difference between opposing slots. Users can also adjust the slot offsets from the trimming lines.

2.5 Final Adjustments

In the final stage, users are given the opportunity to inspect the model and ensure that the fit is correct for the hand. The software also allows the inclusion of personalization features by allowing the user to add/emboss patient-specific information. For this purpose, the user has the ability to modify the embossing font, rotation, sizing, and more. Additionally, it is possible to add shapes and symbols/decals, as long as they are provided in a vector-graphics format. In this final phase, users are provided with two options for the assessment of splint match with the original hand geometry: they can either use the manual fitment tool to analyze individual cross-sections or they can rely on the automated fitment test that calculates the mean surface tolerance deviation between the hand and the splint model. These features along with the text embossing tool are illustrated in Fig. 6.

3 COMPUTATIONAL FRAMEWORK

The codebase for the software is a combination of C# and C++, with C# handling user interaction and C++ managing more computationally intensive tasks. To facilitate the interoperability between the two languages, the application platform invokes a neighboring dynamic link library. Hardware-accelerated rendering is achieved using a custom OpenGL rendering wrapper, while the user interface is implemented using .NET Windows Forms. Interaction with the application is possible through either ray casting or a frame buffer, where each pixel corresponds to a specific object. To ensure optimal performance, the codebase employs advanced coding techniques for computationally intensive scenarios, with parallelization utilized throughout. To minimize the need for synchronization and maximize performance, special emphasis is placed on parallelization. More computationally intensive operations such as triangle neighbor detection and common vertex finding are executed using OpenCL, with built-in software fallbacks. C++ parallelization is implemented using OpenMP and the Parallel Patterns Library, while Advanced Vector Extension support is present but disabled in favor of OpenCL.

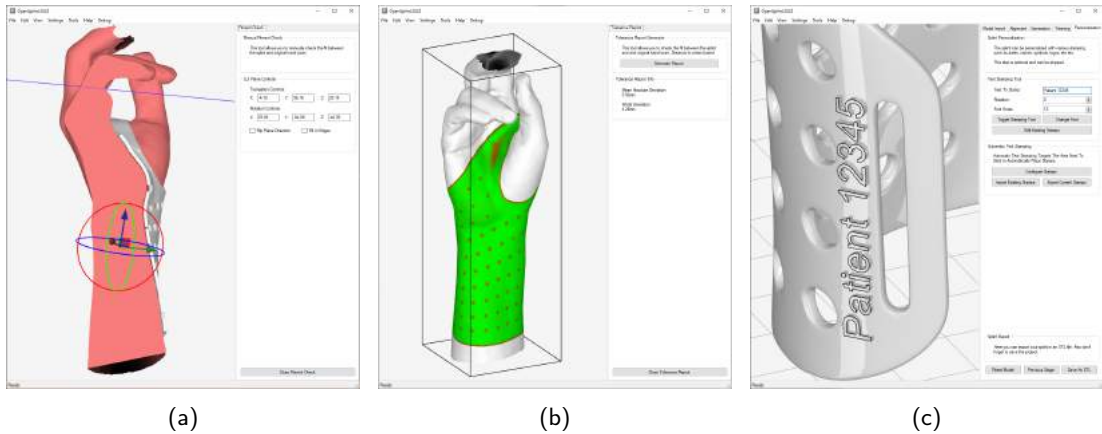


Figure 6: Features of the model preview stage: (a) manual cross section fitment check, (b) automatic fitment report, and (c) personalized embossing.

The software is responsible for all mesh operations, which includes hand geometry import, alignment, lofting, trimming, embossing, and splint geometry export. Linear algebra and vector operations are heavily utilized throughout the software. In cases where precision is critical, linear system comparisons are preferred over vector comparisons due to their lower false positive rates during repair stages. The software stores data in a per-triangle format but may convert to vertex data format depending on the specific mesh operation. The rendering visuals can be customized to be either flat shaded or vertex normal interpolated. Each of the modules presented in Section 2 can be linked to an internal computational subroutine (Fig. 7).

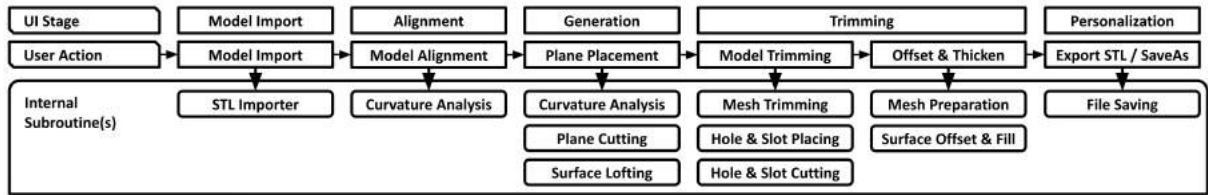


Figure 7: Correlation between software modules and computational engines.

With respect to user control and interaction, transformation matrices were used for positioning and rotation. However, there are some instances in which quaternions were employed for spherical interpolation in animations since interpolating rotation matrices directly did not yield spherical rotation. The plane control system enables the use of both local and global coordinate systems (Fig. 4b) and internally relies on matrices and their inverses to facilitate a better interaction experience. To position vectors, raycasting is extensively utilized throughout the system. A straightforward linear system setup focused on the edges of each triangle was employed to compute valid raycasts. In this regard, the software first calculates ray's position and direction based on the projection matrix and viewport coordinate. After an intersection is determined the software compares the intersection point against the three edge planes per triangle built via Eq. 1, 2, 3, and ensures that the point is either found *in front of* each plane or none at all for it to be considered a valid intersect. The *in front-of* check is performed by means of Eq. 4. The double check is performed to compensate for reversed triangle winding. The closest intersection is also calculated in a similar manner. Squared distance comparisons were implemented to prevent costly square root calls.

$$S_1 = (\vec{v}_2 - \vec{v}_1) \times (\vec{N}) \quad (1)$$

$$S_2 = (\vec{v}_3 - \vec{v}_2) \times (\vec{N}) \quad (2)$$

$$S_3 = (\vec{v}_1 - \vec{v}_3) \times (\vec{N}) \quad (3)$$

$$H \cdot S_1 < \vec{v}_1 \cdot S_1 \quad \&\& \quad H \cdot S_2 < \vec{v}_2 \cdot S_2 \quad \&\& \quad H \cdot S_3 < \vec{v}_3 \cdot S_3 \quad (4)$$

The geometry engine was optimized to run in any 3D planar coordinate system. The software tool makes extensive use of linear algebra to build linear systems based on normal, tangent and bitangent vectors in a sense that almost all comparisons are completed by assessing the side of a plane on which a 3D point is present (Fig. 8). This is accomplished via dot products and is also used to ensure the success of every cutting operation (polygon splitting, rewinding, cutting, etc.).

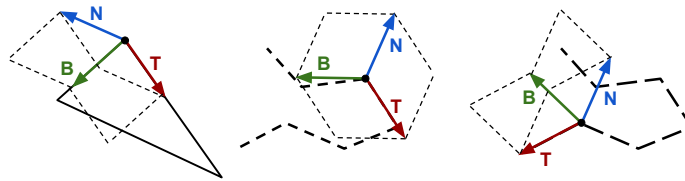


Figure 8: Tangent, bitangent and normal (TBN) vectors.

For example, the point in *poly* - polygon detector algorithm (Fig. 9) - was used by both cutting and text embossing subroutines. This algorithm produces both a cutoff plane and an intersect plane. The normal of the cutoff plane is based on the direction of the first two vertices of the cut polygon and the cut projection direction, while the intersect plane is obtained by calculating the cross product of the cutoff plane and the normal of the cut path. The intersect plane is then used to intersect the cut polygon, with any intersections behind the cutoff plane being disregarded. If the number of intersections is odd, the point is deemed "inside", while if the number is even, the point is considered "outside". In situations where points are located exactly on the cut polygon, special boundary conditions have to be applied since points cannot be automatically assumed as being "inside".

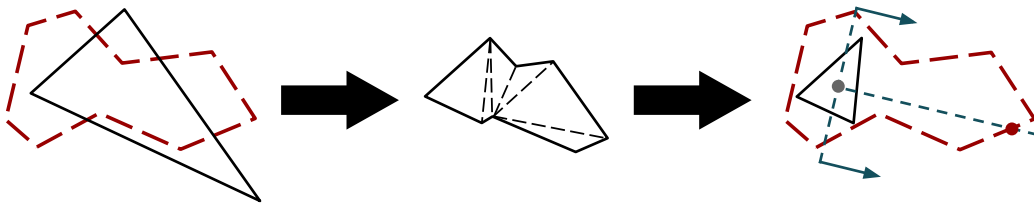


Figure 9: Functionality of the *poly* algorithm.

3.1 Curvature-Based Hand Feature Detection

Detection of hand scan features involves the identification of areas of high and low curvature to be subsequently categorized as fingers, palms, or wrists. The curvature search process is comprised of two distinct stages: detection and grouping.

During the detection stage, a curvature search algorithm is executed on every triangle in the mesh. The algorithm starts with a triangle and samples its neighboring triangles to determine if their angle difference falls within a specific range. If this happens, the algorithm flags these triangles in a local buffer before proceeding to run on the newly flagged triangles. The algorithm terminates once it has examined all neighboring triangles within the starting triangle's angle range. Angle comparisons are made between the starting triangle and its neighboring triangles to ensure consistent results regardless of mesh's aspect (fine or coarse). After the algorithm completes, the flagged triangles' total area is calculated, and if it exceeds a specified threshold, the flagged local buffer is merged additively with the global triangle buffer. Fig. 10 depicts the original recursive version of the algorithm. Its current form was rewritten into an iterative method that eliminates the need for a stack frame [11]. The currently-implemented version is characterized by a four time increase in performance compared with its original form and is also coupled with a reduced chance of stack overflow error on large models.

Additional modes are also present for the detection stage. In one mode only the starting triangle is flagged on the global buffer instead of the entire local buffer. Alternatively, instead of merging the local and global buffers, the local buffer can be added to improve detection of other areas of the hand. When selecting detection parameters, certain parameters such as angle are non-dimensional and can be used universally for hand scans. Other parameters, such as the minimum surface area, are either fixed, dependent on the total mesh surface area, or determined numerically. In the latter case, the curvature analysis algorithm is executed with surface area continuously increasing or decreasing until it detects five high-curvature regions representing each finger.

The algorithm for region grouping represents the second stage of the detection process. During this phase, the detected triangles are grouped into regions that are touching each other. This allows the generation of vertex and surface normal averages that can be used to identify features and create alignment data or cutoff planes. The region grouping algorithm is similar to the curvature search algorithm, except that it is not constrained by angle or distance limits, but by the flagged triangles.

The initial step in aligning hand geometry involves running the algorithm to extract the fingers and thumb. This process entails identifying regions of somewhat flat curvature, and subsequently inverting the detected triangles to pinpoint areas with high curvature (Fig. 11a). The data is then processed to calculate its edges and generate vectors that represent the starting and ending positions of the thumb. This facilitates the alignment process in subsequent stages. Next, the algorithm is run again, but it searches for areas with flatter curvatures. During this phase of detection, two areas of interest are identified: the back of the hand (Fig. 11b) and the palm. Typically, the triangles detected on the back of the hand have a greater combined surface area, but in certain cases, the palm may have a larger surface area and this disrupts the entire alignment process. To address this issue, the position and surface normal of the detected region are compared to the average position and surface normal of the thumb, thus ensuring that the identified area faces away from the thumb vector. Additionally, side alignment can be achieved by restricting the triangles on which the algorithm runs. For instance, by limiting the starting triangles to those with normals pointing sideways (contained in the principal horizontal plane), the resulting detection identifies large areas that correspond to the side of the hand (Fig. 11c). This process is performed following the alignment of the back of the hand since this enables the algorithm to restrict the Y component of the start triangle surface normal. To ensure that the fingers are oriented towards the positive X axis, the algorithm relies again on the data extracted from the finger and thumb regions in a sense that it confirms that the direction from the fingers to the center in the model points towards the positive X axis.

The curvature based feature detection can also assist in automatically placing planes for the surface loft

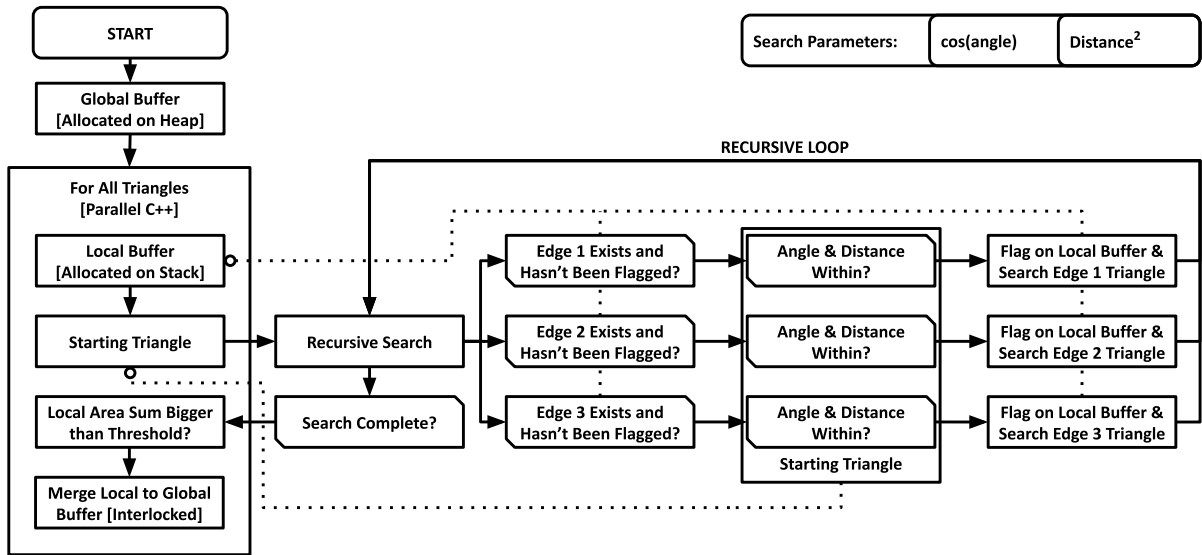


Figure 10: Recursive form of the curvature search algorithm

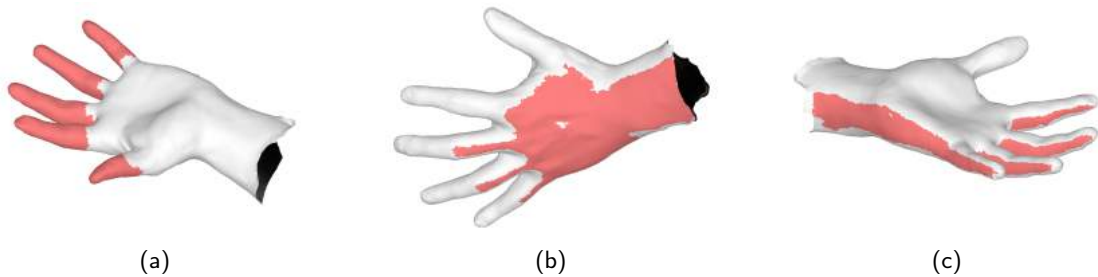


Figure 11: Detection examples involving the curvature search routine: (a) finger and thumb, (b) back of hand, and (c) side of hand.

data. This is done via the finger and thumb data, where the finger start plane is placed between the finger and thumb detected positions. The wrist plane is set 1.5X the distance between the finger and thumb detection region past the thumb. Cut paths are generated from the finger and wrist planes, which are then used to calculate the nearest vertex to the thumb position on both paths. Then a line is drawn the between these two points, and the closest distance to the thumb is found, and the direction to the thumb from that point. This data is then used to generate the thumb cutoff plane.

3.2 Surface Lofting

As surface offsets might be restricted by geometry, a surface loft is employed to construct a mesh. This enables smoother meshing, which helps to prevent non-manifold surface offsets. Moreover, it facilitates the generation of grid data for automatic hole placement, as well as the wrapping and unwrapping of boundary curves onto and out of surface.

The surface lofting phase commences by making three incisions for wrist, fingers and thumb to establish a foundational mesh. However, the thumb cut is offset by a few millimeters to avoid floating point errors when

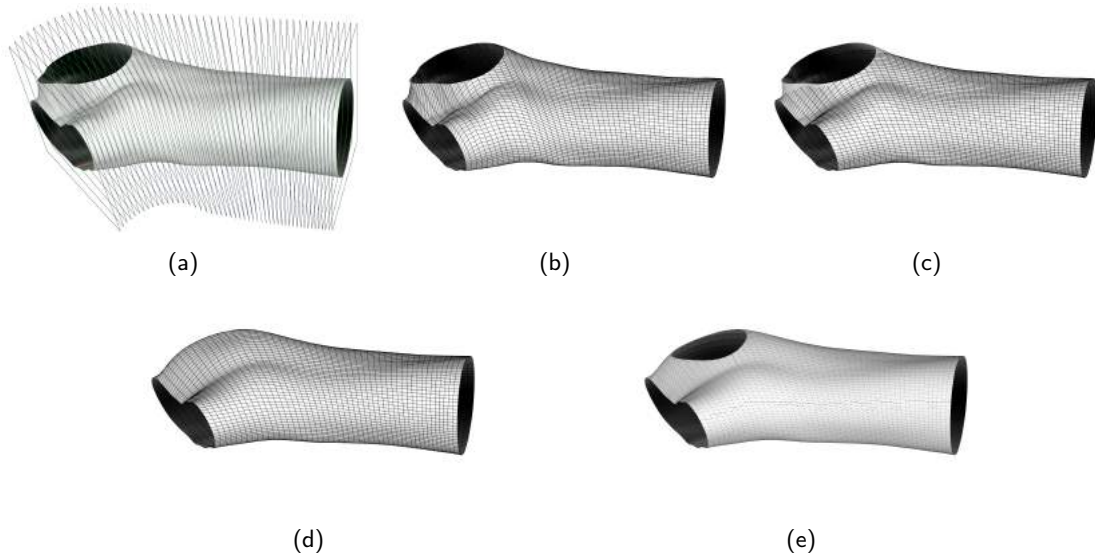


Figure 12: Surface lofting phases: (a) interpolated slicing planes, (b) raw grid data, (c) smoothing, (d) inserting triangles, and (e) raw triangular mesh data.

re-slicing the mesh during a later stage. The entire surface lofting process is depicted in Fig. 12. By using the base mesh data, the linear systems of the two finger and wrist planes are interpolated with each other based on the distance between the centroids of their plane intersection paths. These interpolated systems are then employed to slice the base mesh, and any gaps are filled in directly. The resulting paths are divided into 50 segments each, with segment cross products calculated to ensure that the winding of each path is correct.

Next, a slice line is created on the back of the hand to align all of the paths against each other. This is accomplished by intersecting the alignment slice line with each individual path and inserting a vertex at that location. The vertex order of each path is then shifted so that the inserted vertex is the first item in the array. The dividing algorithm is then re-run to create 50 equal segments per path. The resulting paths are stored in a double array to represent the underlying grid data, with triangles inserted between the points to generate the trimming preview mesh.

Finally, the trimming preview mesh is cut with the thumb plane, without any offset applied. Additionally, the grid data used by the trimming algorithm is prepared. It is tiled and offset so that the center of the doubled grid data corresponds to the back of the hand. This eliminates the need for more expensive tiling checks.

3.3 Mesh Cutting

To perform mesh cutting, there are three modes available: planar, projected, and quad. The planar method involves slicing the entire model based on a linear system. After that, the algorithm determines the side of a plane on which a triangle vertex is located and stores the results in a buffer. This buffer is then used to generate two vertices for line-plane intersections based on an XOR pattern. If triangle cutting is required, AND patterns are used to build the new polygon. If line projection is required, the two line-plane intersections from the valid XOR patterns are used (Fig. 13).

To enable projected cuts, a more intricate algorithm is used. A straightforward approach would involve the use of a planar algorithm and only cut triangles where segment projections are feasible. However, this approach would lead to triangle vertices being placed on other triangle edges and this in turn would cause

issues for other edge-finding algorithms that require two triangle vertices to match in order to recognize them as a valid triangle neighbors. To avoid these issues, a more sophisticated technique was employed. The same cutting algorithm was integrated in other subroutines - such as hole and slot cutting, trimming and stamping tools - that also rely on cutting for their own functionality. Nonetheless, the rejection checks were slightly modified in these cases.

The principal steps of the plane cutting subroutine are:

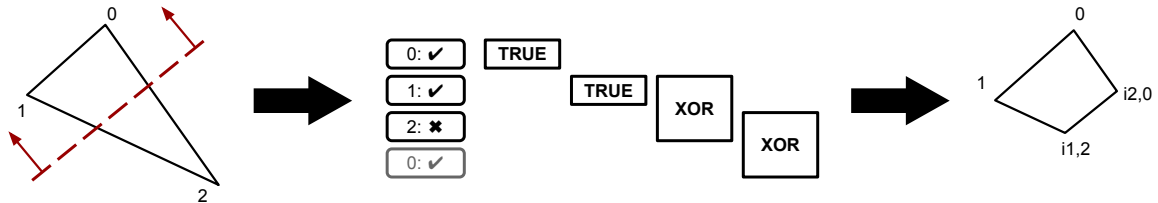


Figure 13: Overview of the plane cutting technique.

1. Verify if triangle intersects plane path segment at any point.
2. Ensure that the triangle intersection is found between the cut path segment.
3. Add the resulting projected line segment into the internal path buffer.
4. Analyze the path buffer segments to build paths.
5. Run the looping algorithm that extracts polygons from the triangle data and projected cut paths intersections.
6. Check for any polygons inside other polygons. If this occurs, then find a valid path between the two polygons such that they can be merged via two common edges.
7. Check winding of polygons to ensure they are valid.
8. Split polygons into triangles.
9. Check if resulting triangle centroid is located inside the main projected cut polygon. This check is only performed for the very first triangle split from the polygon, since if the first triangle in a polygon is to be removed then the remaining triangles from that polygon will also be removed.
10. Save all valid triangles into the main triangle buffer.

The initial step of the algorithm involves projecting (in the case of holes, slots, and text) or intersecting (when trimming quads) the cut lines or quads onto each triangle. To achieve this, a linear system is constructed by means of two points and a vector that denotes the projection direction. Subsequently, this linear system is used to determine the intersection points of the triangle, using only the XOR patterns previously outlined (Fig. 13). Next, the algorithm proceeds by constructing two new linear systems, which are based on the two vertices of a segment of the cut path. These cut planes are oriented towards each other and are trimmed against the XOR path discovered in the previous step. In the event that the original XOR path no longer exists, the triangle is deemed to be unaffected by the cut paths and is therefore skipped from the cutting stage, but not from the rejection stage. The clipping performed via the two linear systems guarantees that only the triangle where the cut line is projected on is cut. During this stage, some tools such as text stamping may gather additional data for edge filling. Once all the segments have been processed for each triangle, the path analyzer algorithm reconstructs the projected triangle segments into paths (Fig. 14a).

The subsequent step involves taking the projected paths and comparing them against the edges of the triangle in order to construct polygons. During this stage, multiple edge case checks are performed to prevent issues caused by floating-point approximations. Additionally, special checks are carried out to avoid special polygon tangency conditions according to which the cut path touches a triangle edge without crossing it or does not touch it at all. Following this, the polygon building stage can commence. This process operates akin to certain board games - such as "Snakes and Ladders" - where the algorithm begins at zero, moves to the first path, which directs it to a different location, and then proceeds through any branches until it reaches its original starting vertex. All vertices traversed by the algorithm are then saved as an extracted polygon. During this phase, the algorithm flags those vertices for skipping on the outer loop. The algorithm then moves on to the next valid outer loop vertex and begins the branch loop anew (Fig. 14b).

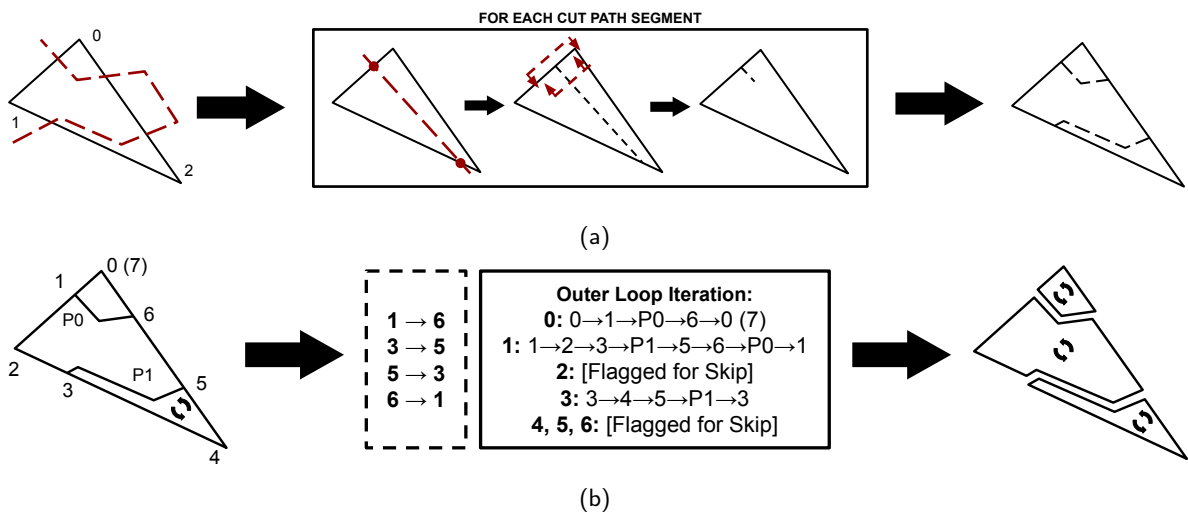


Figure 14: Shape cutting algorithm: (a) cut path projection onto triangle and (b) polygon building stage.

The subsequent step involves handling any polygon-in-polygon scenarios. The polygon-in-polygon algorithm was developed with the ability to reconstruct any type of polygon conditions where multiple polygons are found within other polygons. This is achieved by initially determining which polygon contains other polygons. Subsequently, the vertices of the polygons are compared to identify a valid path between two polygons (*i.e.*, one that does not intersect any other polygons), enabling them to be merged. The polygon merger can handle multiple polygon issues of any kind, as it operates recursively. An example of a two polygon merging operation is shown in Fig. 15. The red circle in the figure depicts a larger gap than what would occur in reality during the merging of vertices forming the merge path. The polygon-to-triangle splitter utilizes small tolerances to account for these conditions.

The penultimate stage of the algorithm involves the polygon winding checker. Due to the complexity of ensuring the correct winding direction in the polygon-in-polygon algorithm, the winding check is conducted prior to the triangles being sent to the polygon-to-triangle splitter. The polygon winding checker determines the polygon winding direction by evaluating if the area of the polygon is positive or negative. In this phase, all polygons should have the same winding direction such that if the area is not of the correct sign, the polygon is reversed. Since this polygon is situated on an arbitrary 3D plane, two planes are constructed based on the direction from the second to the first vertex of the original target triangle. These planes act like transformed X' and Y' axes, and the distance from each polygon vertex to these planes is taken into account for area calculation.

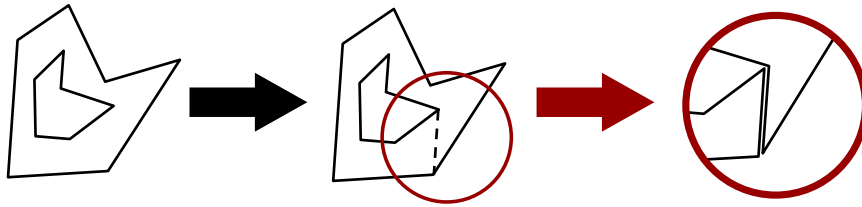


Figure 15: Nested and outer polygon merging.

The final stage of the algorithm involves the polygon-to-triangle splitter. This algorithm begins at the first polygon vertex and evaluates the next two vertices to form a valid triangle. To qualify as valid, two conditions must be met: i) the second vertex must be in front of the plane formed by the first and third vertices and the triangle normal, and ii) there must be no other polygon vertices inside the potential new triangle, which is checked using three planes. Since this algorithm also operates on an arbitrary 3D plane, it employs linear planes once again to determine the location of vertices relative to a plane, and requires correct winding for successful execution. One failing and one passing example are presented in Fig. 16. The polygon splitter performs early checks to determine if the first resulting triangle from a polygon is inside the cut path region, depending on the type of cutting. If it is, the entire polygon being worked on is skipped since all of its triangles will be inside the cut rejection region. For triangles that do not intersect the projected cut lines, checks are performed to determine if the triangle is inside or outside of the cut line.

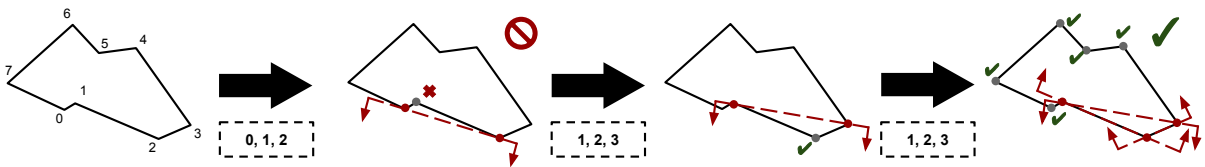


Figure 16: Polygon to triangle splitter.

In case of standard cutting, this step would mark the end of the process. Nonetheless, for tools like the stamper, no triangles will be rejected but they will be stored in a separate buffer. The triangles in the rejection buffer are then offset and the edges are filled based on the stamping edge data gathered during the path building stage.

3.4 Mesh Trimming

The mesh trimming process involves three stages: Bezier curve wrapping, slot placement, and hole placement. In the Bezier curve wrapping stage, the software uses the surface loft to wrap Bezier curves onto the 3D mesh. Composite Bezier curves are used to ensure that the curves pass exactly through the designated points. The user places these points on the surface loft but limited by the resolution of the underlying loft data. The points are then transformed into UV space, and composite Bezier curves are constructed based on the point and slot configuration. The next step involves three iterations of intersecting the data present in the UV space. These intersections split the curve on each U and V coordinate corresponding to the loft resolution. The final intersect iteration is performed diagonally, dividing the Bezier curve at each triangle edge as it passes from one edge to another. After the Bezier curve is split, each Bezier vertex is bilinearly interpolated back onto the surface loft. The resulting wrapped data is used to generate cut quads for mesh trimming. The tangents of the cut quads are built using the two normal values computed from the cross products of neighboring grid

data directions for each individual grid data position. This technique generates splint boundaries of a superior quality (Fig. 17).

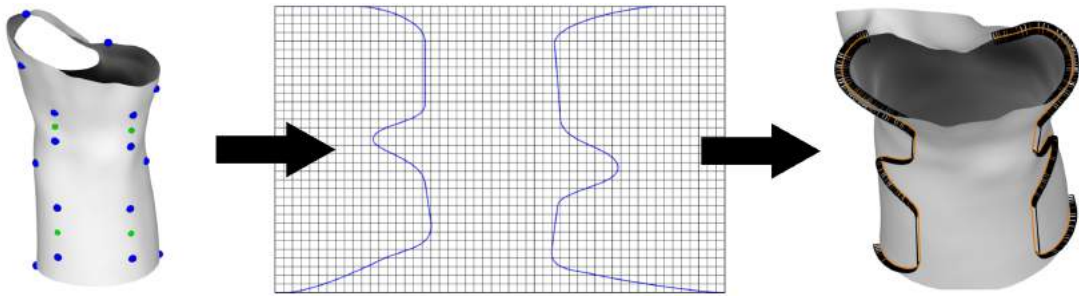


Figure 17: Wrapping and quad cutting approaches.

Following that, the holes are positioned by using the grid wrap data to interpolate their positions onto the mesh. To prevent any distortion on areas with high curvature, distance spacing is also applied. The process of hole positioning based on grid data is demonstrated in Fig. 18b. A rotational matrix is created from the cut direction to transform each cutting hole to be parallel with the mesh using cross products to compute normals for each grid point. Slots, on the other hand, are calculated differently. Rather than performing any type of interpolation, which can lead to slots not being offset perpendicularly to the trim line due to curvature distortion, slot positions are calculated by slicing the model via a plane (Fig. 18a). The resulting slice line is used to create an exact offset distance between the trim line and the slot. Slots are transformed using rotation matrices, and the tangent and bitangent data is taken into account to ensure the slots are parallel to the trim lines they are linked to. The resolution of the Bezier curve was set to 32 segments.

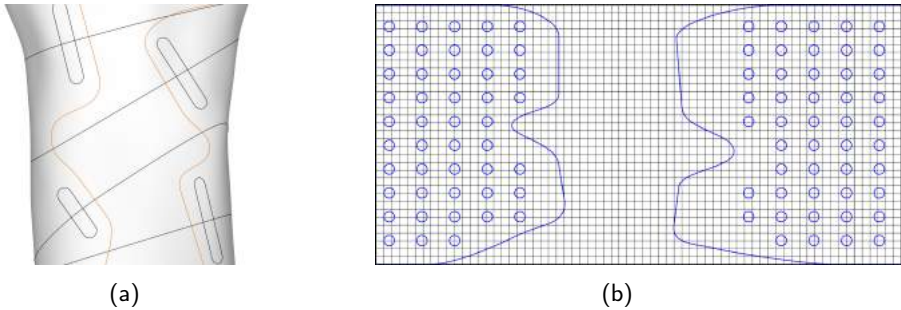


Figure 18: Feature positioning techniques: (a) slice lines underlying the equidistant offset slots and (b) grid used to control the location of the breathing holes.

3.5 Mesh Preparation

To ensure accuracy of the final splint, various mesh repair and smoothing procedures are carried out on the model prior to adding thickness to the splint geometry. The first step involves the merging of adjacent vertices based on their proximity to neighboring triangles' vertices. Vertices within a 0.01 mm cube size are combined, and any triangles that have had at least two vertices merged together are eliminated. Next, triangles containing vertices situated on the edges of other triangles are identified and separated to prevent any disruptions of the edge finding algorithm (Fig. 19).

The last repair operation requires the removal of the triangles characterized by excessively sharp angles via Eq. 5. The associated plane building method is depicted in Fig. 8. The mesh edge smoothing process is applied to the mesh edges, the mesh itself, and the mesh edge normals in order to obtain cleaner and smoother looking edge triangles for the thickening stage. To achieve this, the edges found by the edge finder are assembled into paths, and then subjected to standard Laplacian smoothing. This process involves performing eight smoothing passes with a weight of 0.25 and a range of 3. For the mesh smoothing stage, the common vertex finder is used to collect vertex data, so that vertices of triangles that share a common vertex can be used for mesh Laplacian smoothing. The algorithm is optimized to support both regular and length weight Laplacian smoothing to mitigate mesh volume loss when multiple smoothing passes are performed. While numerous techniques were proposed for feature conservation [7, 8], the length-based approach allowed the most facile implementation. The final stage of smoothing involves the edge normals. More specifically, rather than smoothing the vertices themselves, the normals associated with the triangles that share the vertices are smoothed. This step is important to ensure that the thickening operation does not result in self-intersection or non-manifold geometry. Of note, self-intersection checks are not performed at this time due to their complexity as well as the time required for their implementation.

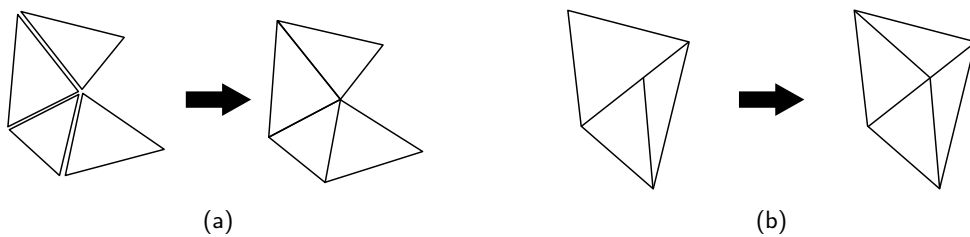


Figure 19: Polygon repair methods: (a) vertex welding and (b) false edge correction.

$$|\hat{D} \cdot \vec{v}_3 - \hat{D} \cdot \vec{v}_1| < 2 \cdot 10^{-3} \quad \text{where} \quad \hat{D} = \frac{(\vec{v}_1 - \vec{v}_2) \times \vec{N}}{\|(\vec{v}_1 - \vec{v}_2) \times \vec{N}\|} \quad (5)$$

3.6 Thickening

The process of thickening the splint involves two offset operations. The initial offset is executed to establish a gap of tolerance between the hand and the splint, while the second offset is conducted for thickening purposes. To thicken splint geometry, mesh triangles were duplicated, offset by a specified amount, and then filled in with edges. The early versions of the software relied on linear systems to determine a 3D point by considering three offset planes. However, this approach was not effective when dealing with sudden sharp offsets due to the presence of a divisor in the plane calculation system and more-than-three plane issues. Therefore, the alternate method utilized a re-normalized average normal that was based on the average normal of the triangles sharing a vertex. Although this technique may not be suitable for models with sharp edges [14], it is appropriate for the smooth geometry of the splint. Consequently, the offset vertices approximated using this approach will accurately represent a real offset operation.

Just prior to the thickening stage, the software identifies and stores all mesh edges in a buffer and then performs two offset operations on the trimmed mesh in order to generate the inner and outer surfaces of the splint. The next step involves filling in the triangles with either a sharp or a domed fill. For the sharp fill, the software searches for triangles with edges in the edge buffer and inserts triangles between the corresponding first and second offset triangles. On the other hand, for the domed fill, the path analyzer is first executed to blend the dome edges correctly. The direction of the dome is determined using the cross product of the

triangle tangent edge and the triangle normal. After a dome path is generated for each path segment, another algorithm fills in the triangles between each dome path. However, the maximum dome distance is limited for slots and holes to prevent alterations in their dimensions. To ensure that slots and pinned slots can still fit correctly, the user can only enable doming for the trim edges and breathability holes. The implementation of doming was chosen over filleting because it is much simpler and can be implemented in a relatively short amount of time (Fig. 20).

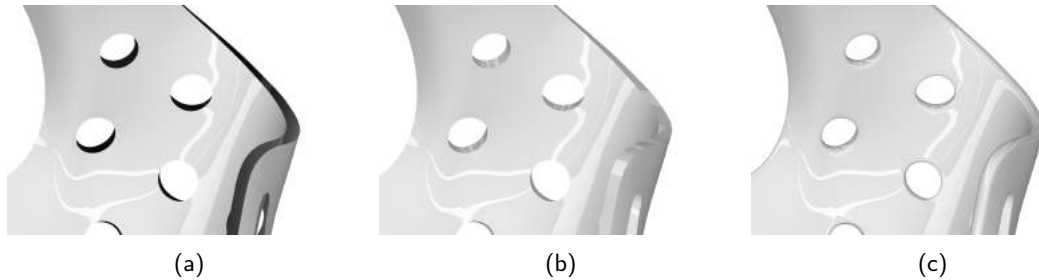


Figure 20: Edge filling modes: (a) unfilled, (b) direct fill, and (c) dome.

3.7 Performance Optimization

The capability to respond quickly to parameter changes represents one of the most important traits of any software tool used for interactive design purposes. To achieve this important objective, parallelization was implemented throughout the entire code. Significant performance gains were realized by relying on OpenCL for repeated tasks such as edge detection, common vertex detection, and surface deviation analysis. Nonetheless, data had to be manually padded to convert from 12-byte to 16-byte aligned 3D vectors for both the SIMD and OpenCL approaches before being copied to GPU memory. This was necessary to avoid significant performance losses that occur when attempting to load unaligned data during runtime [4]. Depending on the hardware setup, the GPU compute approach may also reduce noise, heat, and power consumption. As Table 1 suggests, OpenCL has significantly outperformed the other parallelization methods that were tested. The tests were performed on a system characterized by the following hardware specifications: i9-10900KF (CPU), GeForce RTX 3060 Ti (8GB VRAM), 32GB-3200 CL16 (system RAM).

	C#	C++	C++ SIMD	OpenCL
50K Triangles	3963	555	564	224
100K Triangles	18015	2346	2265	594
167K Triangles	52299	6832	6469	1607

Table 1: Runtime (ms) for various edge detection parallelizations (three-run average.)

The software includes a fallback mechanism, whereby if OpenCL support is absent or if an error arises, the C++ version of the algorithm is used instead. Although this version is also optimized, it ended up being significantly slower than the one involving OpenCL. While CUDA was also considered for its faster performance compared to OpenCL [2, 15, 9], supporting a broad range of hardware for maximum user accessibility was prioritized higher and therefore OpenCL was used for the initial implementation of GPU-accelerated code.

4 MODEL VALIDATION

Although it can be expected that the use of specialized splint design software will inherently simplify the splint generation process, it is rather unclear at this time how a splint generated through the developed fully digital method will compare with a counterpart produced through traditional techniques that are implemented in the current medical practice.

To assess the accuracy of the splints, two traditional hand splints were created and scanned by means of a CT scanner. Furthermore, the geometry of the hand with and without wearing the splints generated through the traditional approach was scanned by means of a custom-built 3D photogrammetry scanner. The two scans were performed to account for changes in hand position after splint was removed as well as to ensure the proper alignment of the CT-scanned splint to the hand. To minimize human error, the hand model was aligned to its splint wearing counterpart by means of singular value decomposition and covariance matrices for least squares best fit alignment based on two sets of points from one model to another [1]. The alignment of both traditional splints is depicted in Fig. 21. Subsequently, the model of the hand wearing the splint was removed and surface deviation analysis was performed between hand and splint models. The target skin to splint tolerance was set to 0.8 mm for all splints. The steps of this process are:

1. Register the model of the hand with the splint on with its counterpart without the splint.
2. Register the model of the splint obtained through CT scanning with its counterpart on the hand. The model of the hand without the splint can be used to improve the registration accuracy.
3. Eliminate the splint wearing model and determine the deviation between the model of the splint and that of the hand without the splint.

The model of the splint generated by means of the interactive software tool did not require any additional registration since it was generated directly from a photogrammetry-scanned model of the hand. Furthermore, the software-generated splints were created to closely resemble their real-life counterparts even though any differences in their outline will not affect the surface deviation amount.



Figure 21: Traditionally-fabricated splint to hand alignment: (a) sample 1 (T-splint 1) and (b) sample 2 (T-splint 2).

A qualitative graphical representation of the surface deviation for four splint models generated by means of the traditional approach (T-splint 1 and T-splint 2) and interactive software tool (S-splint 1 and S-Splint 2) is presented in Fig. 22 whereas Table 2 summarizes the quantitative results for the same four splint models. The splints generated using the software had on average 4 times lower mean absolute surface deviations, and 1.4 times lower maximum surface deviations. The surface deviation maps also suggest a superior deviation consistency to translate into a better patient experience.

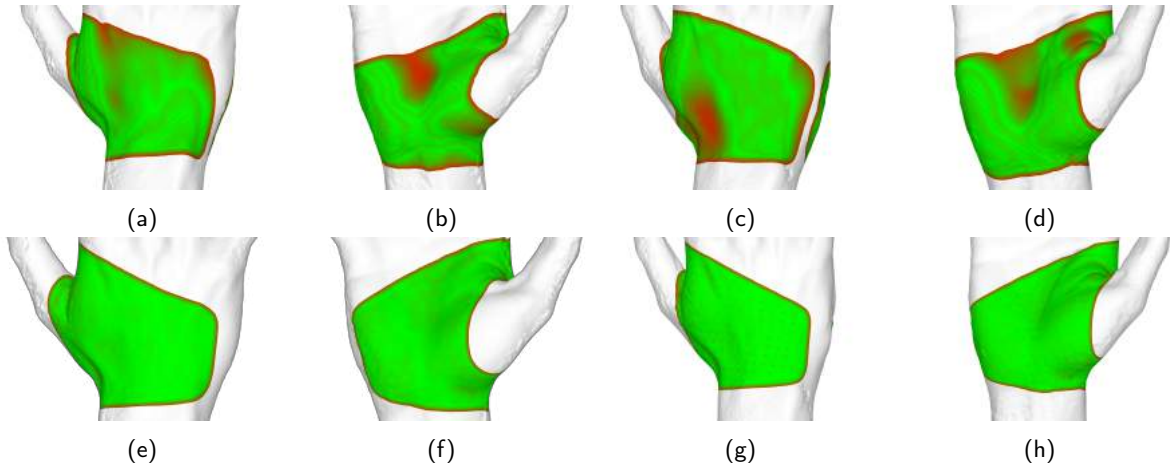


Figure 22: Qualitative comparison of surface deviation for splints generated through traditional (T-splint) and software tool (S-splint) techniques: (a) T-splint 1 dorsal, (b) T-splint 1 palmar, (c) T-splint 2 dorsal, (d) T-splint 2 palmar, (e) S-splint 1 dorsal, (f) S-splint 1 palmar, (g) S-splint 2 dorsal, and (h) S-splint 2 palmar

	Mean abs. dev.	Max. abs. dev.
T-splint 1	1.04	4.95
T-splint 2	1.07	4.66
S-splint 1	0.28	3.07
S-splint 2	0.24	2.85

Table 2: Quantitative comparison of surface deviation (mm) for splints generated through traditional (T-splint) and software tool (S-splint) techniques.

5 LIMITATIONS AND FUTURE IMPROVEMENTS

Similar to any other solutions to a given problem, the proposed approach has its own limitations and/or drawbacks. For instance, the 3D data acquired through photogrammetry sometimes yields as non-manifold. This is a consequence of various errors and/or artifacts caused by the fact that raw hand scan data is typically decimated to a triangle count in the low hundreds of thousands and then smoothed by means of a Laplacian algorithm. However, these operations can sometimes leave behind small artifacts that could interfere with various software routines, even if previous mesh repair attempts are performed. To counteract this, error handling has been implemented and it allows users to revert and adjust the software parameters prior to reattempting to continue the splint generation process. On top of this, faulty geometry can be remeshed, repaired or healed such that the splint design process can succeed.

Despite the significant splint design progress achieved by the current version of the software tool, additional software improvements can still be conceived. For instance, one important enhancement could involve the redesign of the lofting technique to enable the construction of splints to incorporate fingers. This improvement would provide users with the ability to create various types of splints, such as regular hand splints, hand and finger splints, or finger splints alone. Nonetheless, this addition would require extensive changes to both lofting and trimming subroutines. Other issues, such as resolution or extreme curvature lines could also be addressed with a variable lofting system, whereby the resolution of the loft data varies based on its vertical or horizontal

UV coordinate.

Another issue pertains to the incomplete geometry submitted to the software. Although remeshing and mesh refinement are relatively simple features that could be added, they would not be able to reconstruct geometry with missing sections. While filling in smaller holes in the mesh with tangent-aware filling might be a viable solution, larger sections of missing features, such as fingers, may pose a significant challenge. Numerous approaches for filling in missing mesh sections exist [20], but these methods may not be able to accurately handle large portions of complex missing features, such as fingers. Present efforts are underway to address this issue by means of statistical shape modeling [12].

Another significant issue in physiotherapy pertains to repositioning of patient's fingers and thumbs for improved healing. In traditional thermoplastic splints, a physiotherapist adjusts patient's hand while bending the splint to fit their hand. This process is necessary when patients are unable to hold their hand in the target position required for splint fabrication purposes. By contrast, 3D scanners do not allow the acquisition of hand geometry in target positions. To address this, the software would need to incorporate finger and thumb position adjustment capabilities. While simple mesh shearing and bending could achieve this, the accuracy of the geometry, especially in areas characterized by significant bulk deformation such as the thumb, would be questionable. One solution to this problem would be to implement a custom finite element solver in the software to accurately simulate skin [17] and muscle deformation. This approach would enable a physiotherapist to reposition fingers and thumbs with reasonable confidence in the accuracy of the software-generated mesh [6].

6 CONCLUSIONS

The developed software tool has demonstrated with a reasonable degree of confidence its capability to efficiently and accurately generate patient-specific hand splints. By eliminating the requirement for engineering-specific software proficiency, medical professionals might be able to design and fabricate hand splints in a more facile manner. Furthermore, even in cases where CAD expertise is present, the use of the interactive design tool would inevitably be faster compared than using a generic commercial CAD package.

As such, the elimination of stringent needs for advanced 3D modeling capabilities would translate in a superior patient experience since the digitally-generated splints would end up fulfilling better their intended functional goals - owed to their more precise fit with patient's hand - and would require less visits at the clinic, an aspect that is particularly important for those living in distant areas. Finally, its innate modular structure allows the interactive software tool can be repurposed relatively easily for splints addressing other upper or lower limb pathologies including but without being limited to those of fingers, combined hand and fingers, legs, ankles or knees.

ACKNOWLEDGEMENTS

This research was partially funded by the *Natural Science and Engineering Research Council of Canada* (NSERC) and Mitacs Canada.

Adam Gorski, <http://orcid.org/0009-0004-4429-8256>

O. Remus Tutunea-Fatan, <http://orcid.org/0000-0002-1016-5103>

Louis M. Ferreira, <http://orcid.org/0000-0001-9881-9177>

REFERENCES

- [1] Arun, K.S.; Huang, T.S.; Blostein, S.D.: Least-squares fitting of two 3-d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1987, 698–700. <http://doi.org/10.1109/TPAMI.1987.4767965>.

- [2] Asaduzzaman, A.; Trent, A.; Osborne, S.; Aldershof, C.; Sibai, F.N.: Impact of cuda and opencl on parallel and distributed computing. In 2021 8th International Conference on Electrical and Electronics Engineering (ICEEE). IEEE, 2021, 238–242. <http://doi.org/10.1109/ICEEE52452.2021.9415927>.
- [3] Bashshur, R.; Doarn, C.R.; Frenk, J.M.; Kvedar, J.C.; Woolliscroft, J.O.: Telemedicine and the covid-19 pandemic, lessons for the future. *Telemedicine and e-Health*, 2020, 26(5), 571–573. <http://doi.org/10.1089/tmj.2020.29040.rb>.
- [4] Eichenberger, A.E.; Wu, P.; O'brien, K.: Vectorization for simd architectures with alignment constraints. *Acm sigplan notices*, 2004, 39(6), 82–93. <http://doi.org/10.1145/996893.996853>.
- [5] Haleem, A.; Javaid, M.: 3d scanning applications in the medical field: a literature-based review. *Clinical Epidemiology and Global Health*, 2019, 7(2), 199–210. <http://doi.org/https://doi.org/10.1016/j.cegh.2018.05.006>.
- [6] Harih, G.; Kalc, M.; Vogrin, M.; Fodor-Mühldorfer, M.: Finite element human hand model: Validation and ergonomic considerations. *International Journal of Industrial Ergonomics*, 2021, 85, 103186. <http://doi.org/10.1016/j.ergon.2021.103186>.
- [7] Ji, Z.; Liu, L.; Wang, G.: A global laplacian smoothing approach with feature preservation. In Ninth International Conference on Computer-Aided Design and Computer Graphics (CAD-CG'05), 2005, 6–pp. <http://doi.org/10.1109/CAD-CG.2005.4>.
- [8] Jie, T.; Fuyan, Z.: Anisotropic feature-preserving smoothing of 3d mesh. In International Conference on Computer Graphics, Imaging and Visualization (CGIV'05), 2005, 373–378. <http://doi.org/10.1109/CGIV.2005.19>.
- [9] Karimi, K.; Dickson, N.G.; Hamze, F.: A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010. <http://doi.org/10.48550/arXiv.1005.2581>.
- [10] Li, J.; Tanaka, H.: Rapid customization system for 3d-printed splint using programmable modeling technique a practical approach. *3D Print Med*, 2018, 4. <http://doi.org/10.1186/s41205-018-0027-6>.
- [11] Liu, Y.A.; Stoller, S.D.: From recursion to iteration: what are the optimizations? In Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, 1999, 73–82. <http://doi.org/10.1145/328690.328700>.
- [12] Lorenz, C.; Krahnstover, N.: 3d statistical shape models for medical image segmentation. In Second International Conference on 3-D Digital Imaging and Modeling (Cat. No.PR00062), 1999, 414–423. <http://doi.org/10.1109/IM.1999.805372>.
- [13] Popescu, D.; Zapciu, A.; Tarba, C.; Laptoiu, D.: Fast production of customized three-dimensional-printed hand splints. *Rapid Prototyping Journal*, 2020, 26(1), 134–144. <http://doi.org/10.1108/RPJ-01-2019-0009>.
- [14] Qu, X.; Stucker, B.: A 3d surface offset method for stlformat models. *Rapid Prototyping Journal*, 2003, 9, 133–141. <http://doi.org/10.1108/13552540310477436>.
- [15] Su, C.L.; Chen, P.Y.; Lan, C.C.; Huang, L.S.; Wu, K.H.: Overview and comparison of opencl and cuda technology for gpgpu. In 2012 IEEE Asia Pacific Conference on Circuits and Systems. IEEE, 2012, 448–451. <http://doi.org/10.1109/APCCAS.2012.6419068>.
- [16] Treleaven, P.; Wells, J.: 3d body scanning and healthcare applications. *Computer*, 2007, 40(7), 28–34. <http://doi.org/10.1109/MC.2007.225>.
- [17] Tsap, L.; Goldgof, D.; Sarkar, S.: Human skin and hand motion analysis from range image sequences using nonlinear fem. In Proceedings IEEE Nonrigid and Articulated Motion Workshop, 1997, 80–88. <http://doi.org/10.1109/NAMW.1997.609857>.
- [18] Wang, Z.; Dubrowski, A.: A semi-automatic method to create an affordable three-dimensional printed

- splint using open-source and free software. *Cureus*, 2021, 13(3). <http://doi.org/10.7759/cureus.13934>.
- [19] Yang, Y.; Xu, J.; Elkhuisen, W.S.; Song, Y.: The development of a low-cost photogrammetry-based 3d hand scanner. *HardwareX*, 2021, 10. <http://doi.org/10.1016/j.ohx.2021.e00212>.
- [20] Zhao, W.; Gao, S.; Lin, H.: A robust hole-filling algorithm for triangular mesh. *The Visual Computer*, 2007, 23, 987–997. <http://doi.org/10.1007/s00371-007-0167-y>.