Variable-Radius Offset Surface Approximation on the GPU

Ann-Christin Woerl[1]
anwoerl@students.uni-mainz.de

Elmar Schoemer[1]
schoemer@uni-mainz.de

Ulrich Schwanecke[2]
ulrich.schwanecke@hs-rm.de

[1]Institute of Computer Science, Johannes Gutenberg University Mainz, Germany
[2] Computer Vision and Mixed Reality Group, RheinMain University of Applied Sciences Wiesbaden Rüsselsheim, Germany

ABSTRACT

Variable-radius offset surfaces find applications in various fields, such as variable brush strokes in 2D and 3D sketching and geometric modeling tools. In forensic facial reconstruction the skin surface can be inferred from a given skull by computing a variable-radius offset surface of the skull surface. Thereby, the skull is represented as a two-manifold triangle mesh and the facial soft tissue thickness is specified for each vertex of the mesh. We present a method to interactively visualize the wanted skin surface by rendering the variable-radius offset surfaces of all triangles of the skull mesh. We have also developed a special shader program which is able to generate a discretized volumetric representation that can be transformed into a skin mesh. In addition, we show the usefulness of our method to calculate classical Minkowski sums and demonstrate its use for packing problems.

Keywords
Variable-radius offsetting, Minkowski sum, Shader based shape approximation

1 INTRODUCTION

Offset surfaces are an important tool in CAD/CAM applications. Here they are used for various tasks, such as the construction of blend surfaces or the consideration of the radius of a milling cutter. They can also be used to account for the shrinkage of thermoplastics during 3D printing.

While simple variable-radius blend surfaces such as rolling ball surfaces are a widely used tool in geometric modeling, offset surfaces with freely varying radii are less common. One typical use case for these general variable-radius offset surfaces is the modeling of brush-strokes in 2D and 3D painting and modeling applications. Another rather niche application of variable-radius offset surfaces is forensic facial reconstruction. This is about reconstructing the face of a person on the basis of a skull.

Usually forensic facial reconstruction is performed by experts who, based on a lot of experience and statistical knowledge, apply the skin thickness to various points on a particular skull. In the next step, the gaps between these points are filled, which leads to a three-dimensional reconstruction of the face. In [1] the authors present an automatic method for forensic facial reconstruction. Their approach is based on a parametric template skull and dense statistics of the facial soft tissue thickness (FSTT). In the first step of their forensic facial reconstruction method, the skull template is fitted to the given skull. Then, based on the FSTT, a variable-radius offset surface is constructed. Finally, a head-template is fitted to the variable-radius offset surface. The most time-consuming part of this process is the creation of the variable-radius offset surface. In this paper we present a new method for the efficient generation of such a surface.

An explicit exact formula for a variable-radius offset surface can be complicated and time-consuming to evaluate. However, in many applications there is no need for an exact representation. Thus, we opted for an approximate representation, which leads to a better runtime. Our approach is motivated by the following simple observation: The intersection of a straight line with a solid defined by its boundary surface can be described by a set of disjoint intervals. Thus, we compute the intersection of three sets of parallel lines of a Cartesian 3D grid with a given variable-radius offset surface to approximate the surface. This also makes it possible to generate a triangle mesh of the offset (and simultaneously the inset) surface using an adapted Marching Cubes algorithm. Our algorithm works incrementally by dividing the given triangular mesh into single triangles and calculating the intersection intervals of the straight lines and the variable-radius offset surface of each individual triangle. Due to this decomposition we obtain a set of intervals per line. Each interval is directly joined with previously calculated intervals. We use a GPU to perform these computations in parallel. Therefore, we must pay particular attention to avoid incoherent memory access while computing the union of a new interval with an existing sorted list of disjoint intervals. Furthermore, our method can be used to calculate classical Minkowski sums. Another advantage of the proposed representation is that it can be used to quickly check whether a point is part of the Minkowski sum, which can be used for collision detections.

The paper is structured as follows: In the next section, we give a brief overview of work dealing with variable-radius offset surfaces and the related topic of Minkowski sums. Next, we describe our proposed data structure for representing variable-radius offset surfaces such that they can be efficiently evaluated on a GPU and avoid incoherent memory access. In section four we present a detailed description of our shader-based algorithm for calculating variable-radius offset surfaces and Minkowski sums on a GPU. In the penultimate chapter we present two application examples. The paper finally ends with a short summary and some thoughts on further improvements of the presented algorithm.

2 RELATED WORK

Fixed-radius offset surfaces are well studied since they are often used in CAD/CAM modeling. In contrast, variable-radius offset surfaces are less well known. Qun and Rokne [16] developed explicit formulas for variable-radius offset surfaces of parametric surfaces. In [18], Thiery et al. describe a shape representation algorithm using variable-radius offset surfaces. They introduce so called sphere-meshes, which are a connected set of spheres that are linearly interpolated over a triangular area. Wang and Manocha [21] describe an algorithm for fixed-radius offset computation on the GPU. Based on layered depth images, they calculate structured points located on a 3D grid, for which they compute the union of many spheres centered at these points. They use spatial hashing to find intersections. Gietzen et al. [10] use dense statistics of facial soft tissue thickness to reconstruct facial structures by a union of spheres. Achenbach et al. [1] further improved this approach by replacing the discontinuous union of spheres with a global optimized sphere-mesh, which is defined as the zero-set of a signed distance function. From this implicit representation, an explicit representation of the surface mesh can be obtained by using the marching cubes algorithm [15]. We present an algorithm that is able to interactively render a variable-radius offset surface based on per pixel calculations. Our method provides a higher accuracy as the one proposed in [1], while being much less time-consuming. In contrast to [21], our algorithm is able to work directly with 3D meshes without making an intermediate step via layered depth images and using hash functions for finding intersecting rays. Furthermore it can be easily generalized to calculate the Minkowski sum of polyhedra.

In general, there are two basic approaches to determine the Minkowski sum of two polyhedra. The first approach divides the polyhedra into convex components. Then the Minkowski sum is calculated for all pairwise convex components. Afterwards, the union of the individual results must be formed. Hachenberger [11] uses this approach to develop an algorithm for calculating the Minkowski sum of two non-convex polyhedra with exact geometric predicates, which, however, leads to high computation times. Varadhan and Manocha [20] also determine the Minkowski sum by convex decomposition of the polyhedra. They were limited to an approximate representation using a voxel grid representing a signed distance field, resulting in a much faster algorithm.

The second approach for calculating Minkowski sums for non-convex polyhedra is to determine the convolution of two polyhedra, which is a superset of the Minkowski sum. Afterwards, redundant surfaces must be removed by clever trimming. Lien [14] uses this approach and performs trimming in 2D using collision detection. This can be done efficiently by accepting a loss of accuracy. Campen and Kobbelt [3] use different criteria for trimming but are limited to the outer boundary of the Minkowski sum. This leads to a restricted field of applications due to the fact that inner boundaries are common. Li and McMains [13] describe an approach to represent the outer boundary of the Minkowski sum using voxelization on the GPU. The accuracy of their approximative approach is limited by the resolution of the voxel grid. In addition, no inner surfaces can be represented here either. Kyung et al. [12] describe a robust convolutional algorithm to calculate the Minkowski sum of two polyhedra, that finds and removes intersecting facets using kd-trees and achieves high accuracy.

3 SHADER BASED COMPUTATION

In our approach, the variable-radius offset surface is represented by three sets of parallel lines of a Cartesian 3D grid. This results in three independent calculations, each of the three orthogonal surfaces of the Cartesian 3D grid being considered a 2D grid. Thus, the image plane is discretized with a fixed grid width, with the grid points corresponding to the pixels. At each pixel our algorithm determines the intersection of a straight line with the given variable-radius offset surface. This results in a set of disjoint intervals for each pixel. Knowing the world coordinates of the underlying grid, it is possible to check in quasi constant time whether any position is part of the variable-radius offset surface or not. Based on the distance between the origin of each line and the variable-radius offset surface, the corresponding index in the array can be determined and the associated intervals can be found.

We use graphics hardware to compute the variable-radius offset surface for arbitrary triangle meshes. Since each object can be represented approximately by a 3D triangle mesh, this is not a limitation of our algorithm. Furthermore, this approach enables our algorithm to calculate not only the offset surface but also the inset surface simultaneously. A triangle mesh consists of three-dimensional vertex data with additional connectivity information. The vertices may also contain some additional properties such as normals or colors. As the triangle mesh passes through the rendering pipeline, the shader receives variable-radius data for each vertex of the mesh. All triangles of the mesh are considered independently of each other. In the following, we start by describing the general procedure for calculating the variable-radius offset surface for a single triangle which we will extend to multiple triangles afterwards. Implementations of the essential parts of the geometry and fragment shader can be found in Appendix A.

Vertex processing takes place first. Here, each individual vertex is processed by the vertex shader. The vertices are transformed into the two-dimensional image plane by multiplying them with the modelview and projection matrix.

The vertex shader is followed by a geometry shader. As an input it receives a single primitive and generates zero or more primitives as an output. We use one triangle as input and create two triangles as output. These two triangles are constructed as follows. Let $a,b,c \in \mathbb{R}^3$ be the vertices of the input triangle T_{abc} projected along the z-axis and

$$\begin{aligned}\begin{pmatrix} x_{min} \\ y_{min} \end{pmatrix} &= \begin{pmatrix} \min\{a_x,b_x,c_x\} \\ \min\{a_y,b_y,c_y\} \end{pmatrix}, \\ \begin{pmatrix} x_{max} \\ y_{max} \end{pmatrix} &= \begin{pmatrix} \max\{a_x,b_x,c_x\} \\ \max\{a_y,b_y,c_y\} \end{pmatrix}\end{aligned}$$

the lower left and upper right point of its axis-aligned bounding box. Furthermore, let $r_a, r_b, r_c \in \mathbb{R}$ be the variable-radii at the vertices and $r = \max\{r_a, r_b, r_c\}$ the maximum of these. A superset of the projection of the variable-radius offset surface is given by $[x_{min} - r, x_{max} + r] \times [y_{min} - r, y_{max} + r]$, i.e. the projected axis-aligned bounding box extended by the largest of the three radii. The geometry shader generates this rectangle consisting of two triangles and sends it to the fragment shader. In addition, the original coordinates of the triangle vertices are passed such that the link between the generated rectangle and the original triangle is known (see listing 1).

Finally, all objects on the image plane are rasterized. The resulting fragments are processed in the fragment shader. We generate a straight line whose reference points depends on the fragment position. The direction of the line corresponds to the view direction which is equal to the z-direction due to the parallel projection. Afterwards, we determine the intersection of this line with the variable-radius offset surface. If there is an intersection, the corresponding interval is saved, otherwise the fragment will be discarded (see listing 2). If we consider not only the triangular surface of a model, but the solid as such, then the front intersection of the ray with the variable-offset surface provides the offset surface, while the rear intersection defines the inset surface.

Since the triangle mesh consists of more than one triangle, we have to form the union of the resulting intersection intervals. As a data structure we propose a set of arrays with a predefined length, one for each pixel p_i, $i = 0,\dots,m-1$. The first entry contains a mutex variable $m_i \in \{0,1\}$ which is used for concurrency control. In the second entry we store the total amount of intervals associated with this pixel. The following elements describe depth intervals $[z_{2i}, z_{2i+1}]$, $i = 0,\dots,n-1$, where the number of intervals n depends on the depth complexity of the object. Our algorithm works incrementally by inserting new intersection intervals into this array. Thereby, inserting a new interval $[t_1,t_2]$ means to calculate the union of this interval with the given set of ordered intervals. We perform a kind of insertion sort algorithm to find the correct positions of t_1 and t_2. Based on these positions we form the union by using shifts and deletions. This layout of the data is suitable for the GPU. In our implementation, the arrays for all pixels are combined into a single one-dimensional array, which is connected to a shader storage buffer object (SSBO). Since SSBOs are not only readable but also writable, they are well suited for our application. Unfortunately, SSBOs use incoherent memory access. As the calculation on the GPU is performed in parallel for many triangles, this must be taken into account. To avoid errors while inserting a new interval, we implemented a mutex function, which utilizes the mutexes m_i to block parts of the SSBO which are already in use. We use the GLSL function `atomicCompSwap` which exchanges a value of an array iff two given values are equal. Our mutex function uses `atomicCompSwap` to repeatedly check whether the mutex entry m_i of the current pixel array is zero. If $m_i = 0$ we set $m_i = 1$ and the insertion of a new interval can start. We take the union of the new interval with the already existing set of intervals saved in the part of the array which belongs to the actual straight line. After finishing the insertion, the intervals are disjoint again. We have to update the number of intervals and check whether it still fits the maximum number of intervals. Finally, the mutex entry must be reset, i.e. $m_i = 0$. Instead of a fixed array for each pixel, one could also use linked lists, as it is done in modern A-Buffer implementations (see e.g. [22]).

In the next section we discuss the details of determining the intersection of the straight line with the variable-radius offset surface.

3.1 Variable-radius offset surface

A straight line with reference point $p \in \mathbb{R}^3$ describing the position of a fragment and direction $u \in \mathbb{R}^3$ corresponding to the view direction is defined by

$$s(t) := p + t \cdot u,\ t \in \mathbb{R}.$$

The variable-radius offset surface of a triangle T_{abc} can be easily visualized, as shown in Figure 1 on the top and in the center. The triangle can be understood as a skeleton of the surface (see [2]). Tkach et al. [19] use this representation for hand modeling and tracking. In contrast to our work, they use an iso-level-function to describe the variable-radius offset surface which leads to complex computations. We use ray-tracing to approximate the surface instead. The final goal is to calculate the intersection of the ray $s(t)$ with the variable-radius offset surface. Therefore, we can decompose the variable-radius offset surface into simple geometric figures like spheres located at the corner points of the triangle, truncated tangential cones at the edges and triangles shifted to the tangent planes of the spheres. These simple geometric figures are displayed in the center of Figure 1 in different colors. Intersecting a ray with these geometric figures is much easier than finding an explicit form of the variable-radius offset surface. Later, the union of these different intersections is formed to obtain the correct intersection interval.

To calculate the intersection of the ray with the upper and lower triangles, we embed the triangles into the tangential planes of the spheres at the triangle vertices. The upper tangential plane can be described by

$$E_{r_+} := \{x \in \mathbb{R}^3 \mid n_+^\top (x - a) = r_a\}$$

with $n_+ \in \mathbb{R}^3$ being the normal vector of the plane. The Hesse normal form allows to determine the intersection between $s(t)$ and E_{r_+} directly by insertion. The same calculation holds to the lower triangular surface. After the intersection with the plane has been calculated, it must be checked whether it is inside the triangle. The normal vectors of the tangential planes can be determined based on the following system of equations

$$\begin{array}{rcl} n^\top (b - a) & = & r_b - r_a \\ n^\top (c - a) & = & r_c - r_a \\ n^\top n & = & 1 \end{array} \qquad (1)$$

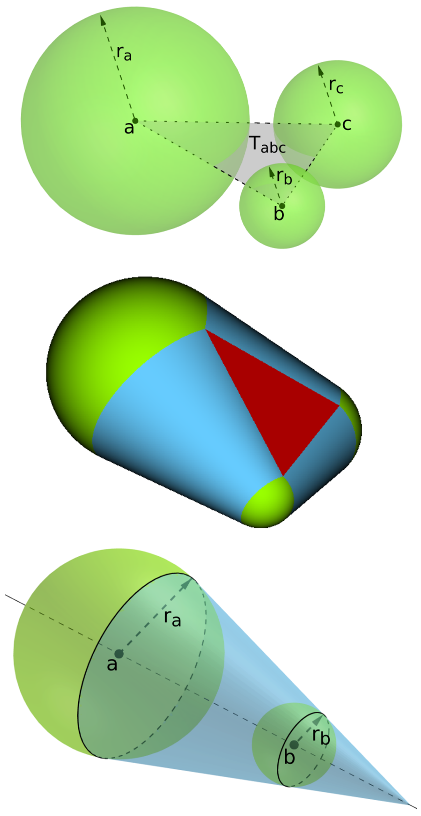

Figure 1: Top: Triangle with variable radii at the vertices. Center: Variable-radius offset surface of a triangle decomposed into spheres (green), tangential cones (blue) and tangent planes (red). Bottom: Tangential cone of the spheres at vertices a and b

by assuming that the distance of each corner point a, b, c to the tangential plane has to equal the radii r_a, r_b, r_c. System (1) leads to a quadratic equation with the two solutions n_-, n_+ for the lower and upper tangential plane.

In order to determine the intersection points of the ray $s(t)$ with the spheres placed at the corners of T_{abc}, we have to find the points on the ray whose distance from the center corresponds to the radius of the spheres.

The intersections of the ray $s(t)$ with the truncated cones at the edges of the triangle can be calculated by using the implicit form of the conical surfaces, which are tangential to the two spheres placed at the endpoints of the edge (see bottom of Figure 1):

$$(x \times (b - a) - a \times b)^2 = (r_a(x - b) - r_b(x - a))^2, \quad (2)$$

with $x \in \mathbb{R}^3$ being the points of the conical surface. The intersections of the ray $s(t)$ with the tangential cone can then be calculated by inserting $s(t)$ into (2). A solution of this quadratic equation is valid, i.e. it lies within the truncated cone, if

$$r_a(r_a - r_b) + d^\top a < d^\top x < r_b(r_a - r_b) + d^\top b.$$

with $d = b - a$.

In summary, using our method for calculating the variable-radius offset surface, only linear and quadratic equations have to be solved, which is easy to implement in a shader program. For each geometric figure we obtain an intersection interval of which we form the union.

3.2 The Minkowski Sum of a 3D mesh and a cuboid

As already mentioned, our method can be generalized to also determine the classical Minkowski sum. In the following we demonstrate this by calculating the Minkowski sum of a 3D mesh and an axis-oriented cuboid.

The variable-radius offset surface of a triangle can be easily visualized, such that a direct calculation of the intersection of a ray with the variable-radius offset surface is possible. In contrast, the decomposition of the Minkowski sum of a triangle and a cuboid into simple geometric objects is not as easy and clear. That is why we choose a different approach in this case. We use the fact that the Minkowski sum describes the subspace in which two objects collide. We assume that the triangle is fixed in space, while the cuboid can be moved along the ray. Then we determine the interval of the ray in which the cuboid center can be placed such that cuboid and triangle collide. This interval corresponds to the intersection of the ray with the Minkowski sum. Therefore we need a dynamic intersection test. In consequence of choosing the direction of the ray as the negative z-direction, the collision always occurs on the front side of the cuboid. So it is sufficient to use a rectangle instead of a cuboid, which simplifies the needed intersection test.

We use the separating axis theorem (SAT, [9]) for collision detection. It says that two convex polyhedra do not collide if it is possible to find a plane that separates them. Or in other words: Two convex polyhedra do not collide if there is an axis for which the projection of the two polyhedra do not overlap. The separating axis is perpendicular to the separating plane. It is sufficient to check only a small set of directions D which consists of the two normal vectors of the triangle and rectangle and the six pairwise cross products of the edges. Furthermore, it is also necessary to check the five normals of the edges which lie in the same plane as the triangle respectively the rectangle. This is due to the fact that when triangle and rectangle are located within the same plane, the cross product of the edges results in the zero vector.

First we assume that both the triangle and the rectangle are fixed in space and set a condition to check whether they collide. Let $o \in \mathbb{R}^3$ be the barycenter of the rectangle and $h_x, h_y \in \mathbb{R}^3$ its half-axes. We get a collision iff for all directions $d \in D$ it holds

$$d_{\min} - r \leq d^\top o \leq d_{\max} + r \tag{3}$$

with

$$\begin{aligned} r &= |d^\top h_x| + |d^\top h_y|, \\ d_{\min} &= \min\{d^\top a,\, d^\top b,\, d^\top c\} \text{ and} \\ d_{\max} &= \max\{d^\top a,\, d^\top b,\, d^\top c\}. \end{aligned}$$

In the last paragraph we assumed that both the triangle and the rectangle are fixed in space. Now, the triangle stays fixed while the rectangle can be moved along the ray $s(t) = p + t \cdot u$. Consequently, the center of the rectangle changes to $o = p + t \cdot u$, which leads to an adjusted inequality (3), i.e. for all $d \in D$ it must hold

$$d_{\min} - r \leq d^\top (p + t \cdot u) \leq d_{\max} + r. \tag{4}$$

An intersection interval is obtained for each direction to be checked. Concerning our former simplification to use a rectangle rather than a cuboid, we have to increase the intersection intervals by the length of the cuboid in z-direction. By forming the intersection of all these intervals, we obtain the interval in which no separating plane can be found. This interval is equivalent to the intersection of the ray with the Minkowski sum.

3.3 The Minkowski Sum of two 3D meshes

Finally, we can calculate the Minkowski sum of two arbitrary 3D meshes. Our approach divides one of the 3D meshes into single triangles and calculates the Minkowski sum of a 3D mesh and an individual triangle incrementally. We form the union of one resulting Minkowski sum with the ones previously calculated.

Since we use a GPU, the triangles of the 3D mesh are treated separately. Thus, we can adjust inequality (4) introduced in the last section to perform a dynamic intersection test between two triangles $T_1 = T_{a_1 b_1 c_1}$ and $T_2 = T_{a_2 b_2 c_2}$. This leads to

$$d_{\min}^{(1)} - d_{\max}^{(2)} \leq d^\top (p + t \cdot u) \leq d_{\min}^{(2)} - d_{\max}^{(1)} \tag{5}$$

with

$$\begin{aligned} d_{\min}^{(i)} &= \min\{d^\top a_i,\, d^\top b_i,\, d^\top c_i\} \quad \text{and} \\ d_{\max}^{(i)} &= \max\{d^\top a_i,\, d^\top b_i,\, d^\top c_i\},\ i = 1, 2. \end{aligned}$$

The directions $d \in D$ we have to check are the two normals of the triangles, the nine pairwise cross products of the edges and the six normals of the edges, which are lying in the same plane as the triangles to cover the degenerated case. After intersecting all these resulting intervals, we obtain the intersection interval of the ray with the Minkowski sum.

4 APPLICATIONS

After explaining the theoretical background, we now focus on some practical applications. Furthermore we will evaluate the runtime of our algorithm. All our tests were conducted on a system with an Intel Core i7 processor with 2.5 GHz, 16 GB RAM and an NVIDIA GeForce GTX 1070 GPU. The runtime of the algorithm is affected by the orientation of the individual surfaces with respect to the image plane. The reason for this is that a rectangle is generated in the geometry shader whose size corresponds to the sum of the bounding boxes of the two objects projected into the image plane. To eliminate this factor from the comparisons, the runtime is considered in relation to the number of fragments processed. For the runtime analysis of the different applications, a window size of 500 x 500 pixels is used as default. In general, doubling the window size in both the *x*- and *y*-directions leads to a factor of four of the processed fragments. The execution of our algorithm with doubled window sizes confirmed that the runtime is almost quadrupled.

4.1 Fixed radius inset/offset computation

The algorithm described in 3.1 calculates the variable-radius offset surface. Of course this approach can also be used to compute the special case of a fixed radius offset. For evaluating the approximation error of our method, we use a 3D mesh of a wooden puzzle. Figure 2 displays the original puzzle (top left) and the exact offset surface (top right) with an offset radius of 0.2. To compare our approximation error with the exact offset surface, we calculate the Hausdorff distance with a low resolution approximation respectively a higher resolution approximation (Figure 2 bottom). Therefore, the Hausdorff distances are linear interpolated between dark blue and red (see Figure 2 lower right), while dark blue parts correspond to a Hausdorff value of 0 and red parts to a Hausdorff value of at most one percent of the offset radius ($2e^{-3}$). By comparing the lower left and middle part of Figure 2, one can see, that the approximation error of our method can become arbitrary small by refining the approximation resolution.

In addition to the offset calculation, our algorithm is also suited to simultaneously calculate an inset surface. Figure 3 displays these calculations for the already mentioned wooden puzzle.

4.2 Forensics

The main application for our algorithm is facial reconstruction. Therefore, we calculate the variable-radius offset surface of a given model skull. Gietzen et al. [10] developed an automatic approach for facial reconstruction based on dense statistics of FSTT. Computer tomographic data is used to generate a 3D model of a skull. Afterwards, landmark points are mapped to a

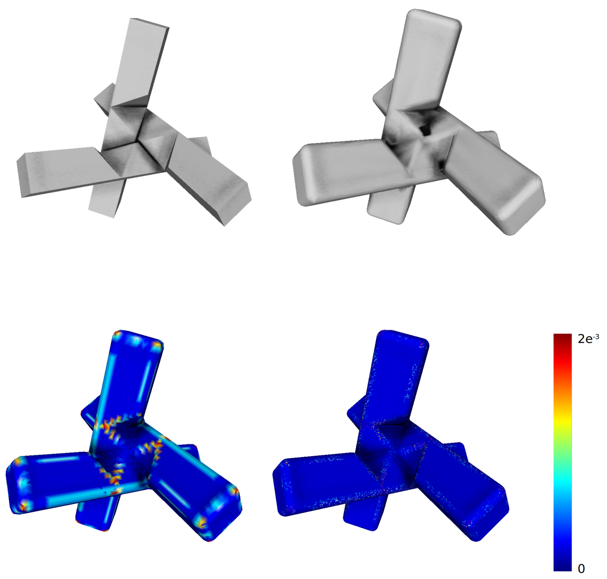

Figure 2: Original puzzle (top left), exact offset surface (top right), low poly approximation (lower left), high poly approximation (lower middle) with color bar of error measurement (lower right).

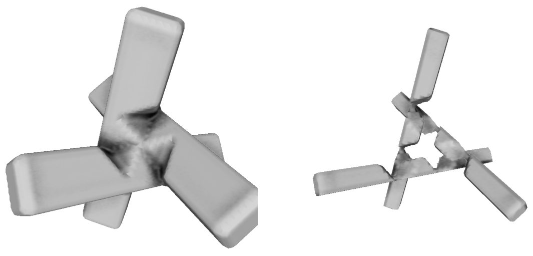

Figure 3: Offset (left) and inset (right) approximation

given model skull for which the average distance between the bones and the skin is known for each vertex. With respect to these vertex-radius pairs, [10] generate a sphere mesh (see [18]) which was further improved by Achenbach et al. [1].

If FSTT data is understood as radii at the vertex points, then our approach can be used to calculate a facial reconstruction in an efficient manner by using a 3D mesh representation of the skull. Figure 4 shows the result of our algorithm. On the top the model skull is shown, while on the bottom the result of the variable-radius offset surface reconstruction using the FSTT is displayed.

To compare our approach to [1], we need to generate a new 3D mesh of the facial reconstruction. Therefore, we use a modified version of the marching cubes algorithm (see [15]). Marching cubes extracts a triangular mesh of an isosurface from a three-dimensional discrete scalar field by calculating the intersection points of this isosurface with the edges of a voxel. We adapted this

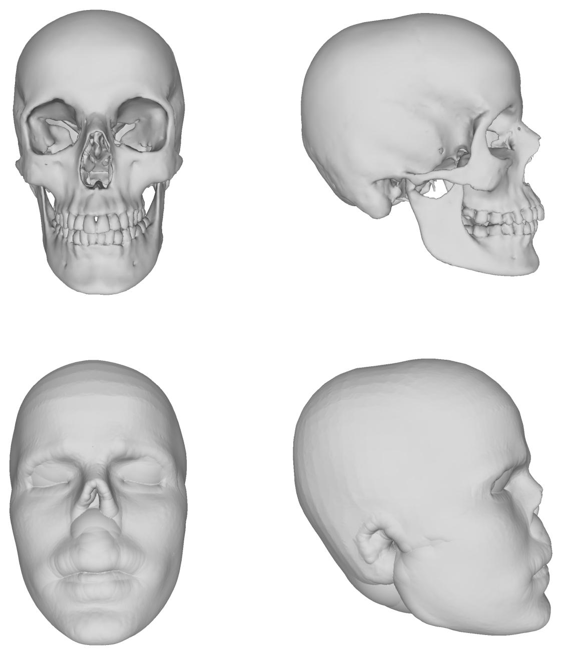
Figure 4: Facial reconstruction (bottom) of a skull (top).

algorithm so that it works with our representation. The intersection points can be read directly from our representation of the variable-radius offset surface.

The three calculations of the variable-radius offset surface need 6 seconds, while marching cubes takes an additional 18 seconds. So we need 24 seconds to perform a facial reconstruction with our approach, while the approach presented in [1] takes 67 seconds on an Intel Xeon CPU ($4 \times 3.6Ghz$).

The runtime of the facial reconstruction depends on the chosen resolution. We choose a resolution of 1 mm per voxel, while [1] use a resolution of 2 mm per voxel. So our approach is almost 3 times faster, while at the same time allowing twice the resolution.

The upper graph in Figure 5 depicts the runtime of the fixed-radius offset surface calculation of a skull with approximately 138,000 triangular surfaces depending on the relative offset radius compared to the diagonal L_d of the bounding box. Taking a closer look at the curve, one can assume that it matches with a power function of the kind $y_i = c \cdot x_i^p$. More detailed investigations show that the exponent corresponds nearly to the value 1.5. This means that a doubling of the radius does not lead to a complete quadrature of the runtime. The lower graph in Figure 5 shows the runtime of the calculation of a variable-radius offset surface of a skull. The runtime is shown in dependency of the number of processed fragments. The curve is almost linear from about 200 fragments with a slope of 140 ms per 100 million fragments.

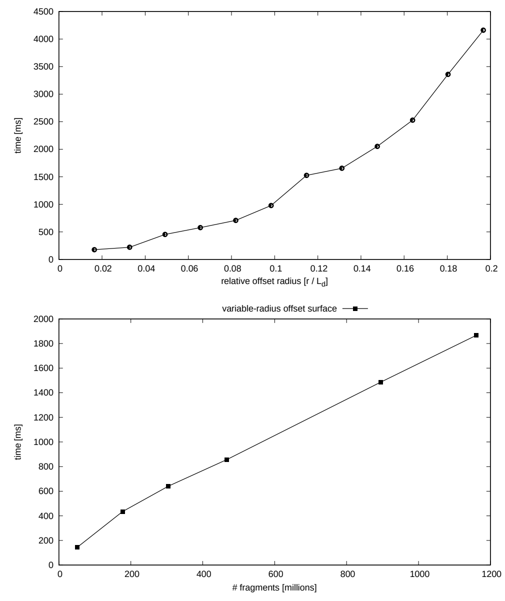

Figure 5: Top: Runtime of the fixed-radius offset surface depending on its relative radius. Bottom: Runtime of the variable-radius offset surface

4.3 Packing problems

Packing problems are a kind of optimization in which a number of objects are packed collision-free into a container. As an real world example we choose the packing of standardized cuboids into a trunk to determine its volume. There are two different standards to calculate this volume - the European and the American standard. The American standard is described in the SAE J1100 standard [17]. It requires to place a set of different sized cuboids in the trunk to determine its volume. Ding and Cagan [8] used extended pattern search to solve this problem with respect to the SAE standards.

In our work, we choose the European standard, which is described in the DIN standard 70020-1 [4] and ISO 3832 [5], but our approach can be used with the American standard as well. To determine the volume of a trunk we have to calculate the maximum amount of standardized cuboids measuring 50 x 100 x 200 mm which can be placed collision-free inside the trunk. This can be done by physically packing these cuboids or by an algorithmic approach. Eisenbrand et al. [6] developed an algorithm to solve this problem by discretizing the trunk geometry using a three-dimensional cubic grid. They used different greedy algorithms and heuristics to determine a maximum amount of axis-oriented cuboids which can be placed collision-free inside the cubic grid. In [7], they extended this approach by allowing continuous positions and orientations while using a simulated annealing algorithm to solve this problem. This randomized algorithm selects a lot of random positions inside the trunk where it tries to place the cuboids. Each time, it has to decide whether this position leads to a collision with the trunk boundary which is the bottleneck of this algorithm. This is equivalent to decide whether the given position is part of the Minkowski sum of the cuboid and the trunk. We only allow axis-oriented cuboids. Therefore, we have to calculate six different Minkowski sums, one for each different orientation. The calculation of each Minkowski sum can be done in real-time.

Using our data structure to decide whether a given position is part of the Minkowski sum corresponds to a simple look up in a list of sorted intervals. This can be done in a quasi constant time. Figure 6 shows the result of the packing algorithm using our representation of the Minkowski sum of a trunk and a standardized cuboid. Our approach can be used in all algorithms that pack axis-oriented DIN- or SAE-cuboids into a trunk. These can benefit from our method of collision detection between cuboid and trunk geometry.

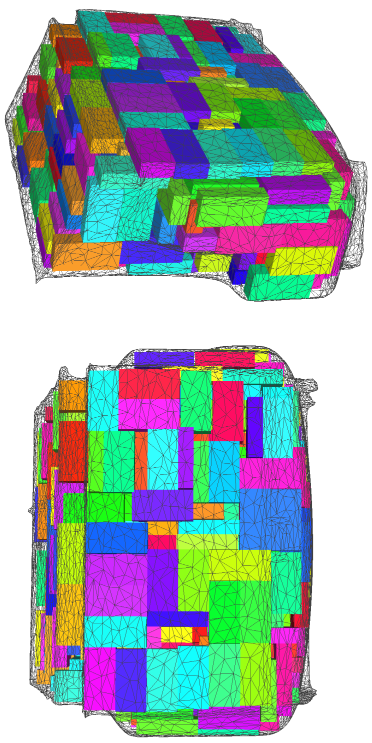
Figure 6: Result of a packing algorithm based on a trunk geometry

5 CONCLUSION AND FUTURE WORK

In this paper we presented a new method to approximate the variable-radius offset surface. Our implementation utilizes the GPU, which, due to its parallel functionality, is well-suited to perform similar calculations simultaneously. We use a set of straight lines over a two-dimensional grid to represent the variable-radius offset surface. Applying our algorithm to determine three orthogonal representations, we are able to generate a new triangular mesh of the variable-radius offset surface by adapting the marching cubes algorithm. Because the focus of our work was on developing an algorithm for calculating the variable-radius offset surface, the marching cubes algorithm is not optimized yet. Optimizing the marching cubes algorithm contains a large potential of speed improvements and could be done in further research. Our algorithm also allows the fast calculation of classical Minkowski sums between a triangular mesh and another object with a relatively small description complexity. Further improvements could focus on optimizing the implementation for general triangle meshes.

We demonstrated the value of our algorithm on three different applications. First, we applied our algorithm to offset and inset computation and showed, that the approximation error of our method can become arbitrarily small. Second, we demonstrated how to use our method for facial reconstructions in a forensic context. We compared our method with [1] and demonstrated that it provides a clear speed advantage while allowing a higher resolution. Third, we apply our algorithm to solve packing problems. As an example we determine the trunk volume by packing axis-oriented standardized cuboids. Therefore, we use the the Minkowski sum representation presented here for collision detection in quasi constant time. With the approach presented here we can calculate the Minkowski sum of the trunk geometry and axis-oriented cuboids in a few milliseconds and use them to pack standardized cuboids into a trunk without collision with the boundary.

6 REFERENCES

[1] Jascha Achenbach, Robert Brylka, Thomas Gietzen, Katja zum Hebel, Elmar Schömer, Ralf Schulze, Mario Botsch, and Ulrich Schwanecke. A Multilinear Model for Bidirectional Craniofacial Reconstruction. In Puig Puig et. al., editor, *Eurographics Workshop on Visual Computing for Biology and Medicine*, pages 67–76. The Eurographics Association, 2018.

[2] Jules Bloomenthal and Ken Shoemake. Convolution surfaces. *SIGGRAPH Comput. Graph.*, 25(4):251–256, July 1991.

[3] Marcel Campen and Leif Kobbelt. Polygonal boundary evaluation of minkowski sums and swept volumes. *Computer Graphics Forum*, 29(5):1613–1622, 2010.

[4] Deutsches Institut fuer Normung e.V. DIN 70020 part 1, road vehicles; automotive engineering; dimensional terms, 02 1993.

[5] Deutsches Institut fuer Normung e.V. ISO 3832, passenger cars - luggage compartments - method of measuring reference volume, 06 2002.

[6] Quan Ding and Jonathan Cagan. Automated trunk packing with extended pattern search. *SAE Technical Papers*, 03 2003.

[7] Friedrich Eisenbrand, Stefan Funke, Andreas Karrenbauer, Joachim Reichel, and Elmar Schömer. Packing a trunk - now with a twist! *International Journal of Computational Geometry & Applications*, 17(05):505–527, 2007.

[8] Friedrich Eisenbrand, Stefan Funke, Joachim Reichel, and Elmar Schömer. Packing a trunk. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 618–629, 2003.

[9] Christer Ericson. *Real-time collision detection.* Morgan Kaufmann, 2007.

[10] Thomas Gietzen, Robert Brylka, Jascha Achenbach, Katja zum Hebel, Elmar Schömer, Mario Botsch, Ulrich Schwanecke, and Ralf Schulze. A method for automatic forensic facial reconstruction based on dense statistics of soft tissue thickness. *PLOS ONE*, 14(1):1–19, 01 2019.

[11] Peter Hachenberger. Exact minkowksi sums of polyhedra and exact and efficient decomposition of polyhedra into convex pieces. *Algorithmica*, 55(2):329–345, 2009.

[12] Min-Ho Kyung, Elisha Sacks, and Victor Milenkovic. Robust polyhedral minkowski sums with gpu implementation. *Computer-Aided Design*, 67:48–57, 2015.

[13] Wei Li and Sara McMains. Voxelized minkowski sum computation on the gpu with robust culling. *Computer-Aided Design*, 43(10):1270–1283, 2011.

[14] Jyh-Ming Lien. A simple method for computing minkowski sum boundary in 3d using collision detection. In *Algorithmic foundation of robotics VIII*, pages 401–415. Springer, 2009.

[15] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[16] Lin Qun and J. G. Rokne. Variable-radius offset curves and surfaces. *Mathematical and Computer Modelling*, 26(7):97 – 108, 1997.

[17] Society of Automotive Engineers. SAE J1100, motor vehicle dimensions, 11 2009.

[18] Jean-Marc Thiery, Émilie Guy, and Tamy Boubekeur. Sphere-meshes: Shape approximation using spherical quadric error metrics. *ACM Transactions on Graphics*, 32(6):1 – 12, 2013.

[19] Anastasia Tkach, Mark Pauly, and Andrea Tagliasacchi. Sphere-meshes for real-time hand modeling and tracking. *ACM Trans. Graph.*, 35(6), November 2016.

[20] Gokul Varadhan and Dinesh Manocha. Accurate minkowski sum approximation of polyhedral models. In *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.*, pages 392–401. IEEE, 2004.

[21] Charlie CL Wang and Dinesh Manocha. Gpu-based offset surface computation using point samples. *Computer-Aided Design*, 45(2):321–330, 2013.

[22] Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. In *Computer Graphics Forum*, volume 29, pages 1297–1304. Wiley Online Library, 2010.

A SHADER IMPLEMENTATION

```
#version 450 core
layout (triangles) in;
layout (triangle_strip, max_vertices = 6) out;
layout (std430, binding=3) buffer data_buffer {
    float radians[]; };

in VS_OUT {
    int index;
} vdata[];

out GS_OUT {
    vec3 vertices[3];
    float id[3];
} data;

// uniform variables: zoom, uMMat, uVMat, uPMat

void main() {
    vec4 hp[3];
    vec3 p[3];
    for(int i=0; i<3; i++) {
        data.vertices[i] = (uMMat * gl_in[i].gl_Position).xyz;
        data.id[i] = vdata[i].index;
        hp[i] = uPMat * uVMat * uMMat * gl_in[i].gl_Position;
        p[i] = hp[i].xyz/hp[i].w; }

    float radius_ = max(radians[vdata[0].index], max(radians[vdata[1].index], radians[vdata[2].index]));

    vec4 radius = uPMat * vec4(zoom*radius_, zoom*radius_, 0, 1);

    float min_x = min(p[0].x, min(p[1].x, p[2].x));
    float max_x = max(p[0].x, max(p[1].x, p[2].x));
    float min_y = min(p[0].y, min(p[1].y, p[2].y));
    float max_y = max(p[0].y, max(p[1].y, p[2].y));

    vec4 s = vec4(min_x - radius.x, min_y - radius.y, 0, 1);
    vec4 t = vec4(max_x + radius.x, min_y - radius.y, 0, 1);
    vec4 u = vec4(min_x - radius.x, max_y + radius.y, 0, 1);
    vec4 v = vec4(max_x + radius.x, max_y + radius.y, 0, 1);

    gl_Position = s; EmitVertex();
    gl_Position = t; EmitVertex();
    gl_Position = u; EmitVertex();
    EndPrimitive();

    gl_Position = t; EmitVertex();
    gl_Position = v; EmitVertex();
    gl_Position = u; EmitVertex();
    EndPrimitive();
}
```

Listing 1: Essential parts of the geometry shader

```
#version 450 core
layout (std430, binding=1) buffer offset_data {
    int intervals[]; };
layout (std430, binding=3) buffer data_buffer {
    float radians[]; };
layout (std430, binding=5) buffer shader_error {
    int error[4]; };

in GS_OUT {
    vec3 vertices[3];
    float id[3];
} data;

// uniform variables: uMMat, uVMat, uPMat, uInvVMat, uInvPMat, width, height, lod, decimalPrecision
vec3 intercept_points[7], normals[7]; // ray surface intersections

void mutex(int t1, int t2, uint i) {
    int max_value = 5000;
    bool done = false;

    while (!done && max_value-- > 0) {
        if (atomicCompSwap(intervals[i], 0, 1) == 0) {
            // insert interval [t1, t2] into interval list at index i, see section 3
            save_interval(t1, t2, i);
            atomicExchange(intervals[i], 0);
            done = true; }
    }
    if (max_value == 0)
        atomicExchange(error[0], 1);
}

int check_t(vec3 t[7], uint i) {
    bool intersection = false;
    int idx = -1;

    for (int i = 0; i < t.length(); i++) {
        if (t[i].z != -1) {
            intersection = true;
            break; }
    }
    if (intersection == false)
        discard;

    float t_small = infinity;
    float t_big = -infinity;

    for (int j = 0; j < t.length(); j++) {
        if (t[j].z == -1)
            continue;
        t_small = min(t_small, t[j].x);
        if (t[j].y > t_big) {
            t_big = t[j].y;
            idx = j; }
    }

    if (t_small > t_big)
        discard;

    mutex(int(floor(t_small * decimalPrecision)), int(ceil(t_big * decimalPrecision)), i);

    return idx;
}

void main() {
    // Build triangle normal ---------------------
    vec3 e0 = data.vertices[1] - data.vertices[0];
    vec3 e1 = data.vertices[2] - data.vertices[0];
    vec3 e2 = data.vertices[1] - data.vertices[2];
    vec3 n_triangle = normalize(cross(e0, e1));

    vec4 u1 = uInvVMat * uInvPMat * vec4(0, 0, -1, 0);
    vec3 u = normalize(u1.xyz); // view direction

    uint posX = uint(gl_FragCoord.x);
    uint posY = uint(gl_FragCoord.y);
    uint idx = (posY * width + posX) * lod * 2;

    vec4 p1 = uInvVMat * uInvPMat * vec4((2.0f * posX)/width - 1, (2.0f * posY)/height - 1, 0, 1);
    vec3 p = p1.xyz / p1.w;

    // Intersections -------------------------
    vec3 tmp[7]; // store depth values of intersections
    for (int i = 0; i < tmp.length(); i++) {
        tmp[i] = vec3(0, 0, -1);
        intercept_points[i] = vec3(0, 0, 0);
    }

    // see section 3.1 for detailed intersection tests
    intersect_triangle(p, u, n_triangle, tmp[0]);

    intersect_sphere(p, u, 0, tmp[1]);
    intersect_sphere(p, u, 1, tmp[2]);
    intersect_sphere(p, u, 2, tmp[3]);

    intersect_cone(p, u, 0, 1, tmp[4]);
    intersect_cone(p, u, 2, 0, tmp[5]);
    intersect_cone(p, u, 1, 2, tmp[6]);

    // union of intervals; find index of nearest point in view direction
    int min_idx = check_t(tmp, idx);

    vec4 ndc = uPMat * uVMat * vec4(intercept_points[min_idx], 1);
    gl_FragDepth = (1 + ndc.z)/2;

    vec3 normal_ = normalize(normals[min_idx]);
    vec3 lDir = normalize(u); // light direction
    FragColor = vec4(max(dot(lDir, normal_),0.0)* vec3(0.94), 1.0);
}
```

Listing 2: Essential parts of the fragment shader