A Method for Automatically Animating the Reassembly of Arbitrary Voronoi-Fractured Objects

Stefan Sietzen*
Supervised by: Michael Wimmer†

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna / Austria

Abstract

In this paper I present a method to procedurally generate the necessary data to animate 2 robots assembling an arbitrary Voronoi-fractured object without intersections. The main problem of finding a possible assembly sequence is tackled using various methods of geometrical calculation and linear programming. The goal of this work is to automatically generate visually interesting animation which can then be used for various scenarios, for example decorative visuals or game scenery.

Keywords: Procedural Animation, Robotics, Part Assembly

1 Introduction

Animation of complex behaviours for games or visual effects is mostly done by manually setting keyframes (or using motion capture if applicable). This is a relatively tedious process and, while walk cycles for example are a method of generating repetitive motion automatically, keyframes are not efficient for repeated motion that can be defined algorithmically.

In this paper I take the real world example of automatic robot assembly, which is in reality a very complex problem necessitating not only geometric computations, but also computer vision and other disciplines to accurately map the real world to a processable data model, to the virtual realm where all geometric and physical data is already known to develop a setup that can automatically and efficiently generate complex animation.

In Section 2 I introduce the reader to a few necessary concepts to make this paper self-contained and describe the physical simplifications I adopted to reduce the complexity of the problem. In Section 3 I explain the algorithms used for the generation of the animation and the adaptations I made compared to [10] to better fit them to my requirements. In Section 4 I briefly show how the implementation in Houdini was set up and evaluate the benefits and constraints of that environment. In Section 6 I evaluate my setup and show some areas where improvement is desirable as well as problematic scenarios where the system is prone to failure.

*stefan.sietzen@gmx.at
†wimmer@cg.tuwien.ac.at

2 Background

In the following paragraphs I briefly introduce a few key concepts for the better understanding of the subsequent sections.

2.1 Voronoi Fracturing

Voronoi diagrams are used in different scientific fields, for example, as Fisher writes in [5], for modelling forest dynamics [7], animating lava flows [12] and neural network design [9]. Their aesthetic potential also makes them useful for architectural application.

Let P be a set of n distinct points (known as sites for the purpose of this definition) in the plane. The Voronoi diagram of P is the subdivision of the plane in n Voronoi cells, one for each site in P, with the property that a point q lies in the cell of site p_i if and only if $dist(q, p_i) < dist(q, p_j)$ for each $p_j \in P$ with $j \neq i$ [3, 6, 11].

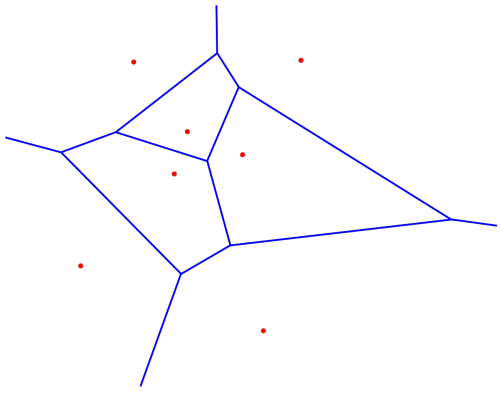

Figure 1: Voronoi Diagram (Markus Matern, 2009, public domain)

In Computer Graphics, Voronoi fracturing is a widely

used method to prepare objects for rigid body destruction simulation. Its property of generating convex shapes is friendly to simulation algorithms that can usually perform much faster with convex bodies. The process of fracturing an object with this method can be controlled by the distribution of the source points. When simulating an impact of one object hitting another, points are often distributed more densely in the impact areas.

In our case, Voronoi fracturing is chosen as the method of splitting the source object into parts because it provides a good balance between visual complexity (as opposed for example to just uniform cubes) on one side and computational complexity on the other side.

2.2 Reconstructing Voronoi fractured objects

Given the inner convexity of Voronoi fractured pieces, it is generally impossible for two pieces to be geometrically interlocked with each other, so when deconstructing the object piece by piece it is always possible to remove at least one piece without intersecting another piece. Therefore there is always a sequence to take the object apart, which in reverse implicates that there exists always a sequence to assemble the pieces back together from a separated state.

To achieve realism, it is usually necessary to start with the pieces that have contact with the ground the object is placed upon. Once this constraint is introduced we lose the ability to state that there is generally always an assembly sequence. Object features that decrease the probability of having a valid assembly sequence are discussed later in this paper.

To decrease the complexity for the calculations, apart from above prerequisite and the trivial necessity to always attach the next piece to an existing piece, no physical constraints are taken into account, so, although for better visual appearance the distance to the local center of gravity is taken into account, it is always assumed that attaching one piece to the existing structure yields a rigid connection without static instability.

2.3 Linear Programming

Linear programming is a method for optimizing a function within a set of linear constraints, defined by inequalities. Linear programming is widely used in economic optimization but has a large field of usages in modelling properties of various systems. One of the most popular algorithms to solve a linear problem is the Simplex algorithm, which although in the worst case can have exponential complexity, has proven in practise to be very fast. As the application in this paper has a very limited number of constraints, the complexity of this component will not be taken into consideration.

3 Program Description

I'll first describe the overall structure of the system and then explain the most important tasks in detail.

3.1 General Structure

The system consists of several modules that encapsulate major steps of the procedure.

Fracturing Module

Inputs:

- Polygonal Object
- Piece Count
- Template Points for original Piece Position

Outputs:

- Fractured Pieces
- Template Points for original Piece Position including lying orientation

This Module fractures the input object by evenly distributing a defined number of points in it and taking those as cell points for a Voronoi fracturing operation. The resulting pieces are then placed on the template points for the original position and a rigid body simulation of the pieces falling to the ground from a minimal height is performed to get valid orientations for the pieces lying on the ground.

Piece Assembly Module

Inputs:

- Fractured Pieces
- Template Points for original Piece Position and orientation
- various parameters to control animation details

Outputs:

- Animated piece geometry assembling from original Position
- Transformation Data for 2 Robot End Effectors assembling the Object

This Module contains the main calculations. It generates the assembly sequence, calculates assembly directions and subsequently animation paths for the pieces and the robots.

Robot Module
Inputs:

- Transformation Data for end-effector
- Base position
- Retracted effector position

Output:

- animated robot geometry

This Module is essentially just a robot rig that attaches onto the end effector transformation data generated by the second module and generates transformations for various parts of the robot via inverse kinematics and physical simulation (for the hydraulic hoses).

3.2 Generating an assembly sequence

Existing work on assembly sequence generation often constructs an exhaustive graph of all possible assembly sequences, and relies on user input to determine if a part can be placed before another to do so ([2, 4]). These liaisons can explode exponentially with the part number, so it is not feasible to let a human define them when exceeding a certain number of parts. [10] states:

> ”From a given set of parts P there can be many different assembly configurations which can be formed. Thus if one is using a forward search algorithm to find feasible subassemblies in the process of generating a final known assembly, the branching factor becomes high enough to make the algorithm inefficient. The problem of finding how to assemble a given product can thus be converted to an equivalent problem of finding how the same product can be disassembled.”

For these reasons, the assembly algorithm which I used is also based on disassembly. It is in fact quite similar to the one proposed by [10], although Nnajis algorithm constructs an exhaustive and/or graph of possible disassemblies to further evaluate their quality / cost. As the cost of an assembly sequence does not really influence its visual quality, I used a simplified algorithm that just finds **one** disassembly sequence if one can be found.

Another simplification is, that while Nnajis algorithm uses all disassembly directions (directions in which a piece can be removed without collision in a given disassembly state) as a base and reduces them by heuristics to make the computation more efficient, I just regard one vector (the one with the biggest minimal directional difference to all directional constraints). In short form, my algorithm is as follows:

Pre-calculate data before disassembly simulation:

- An adjacency graph consisting of nodes representing pieces and edges for each pair of pieces that have mating faces, see Figure 2
- Pieces that are in contact with the ground, their connected clusters and their centroids

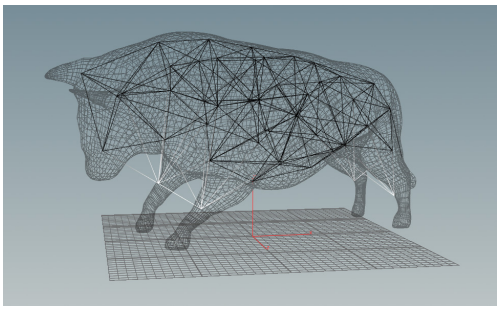

Figure 2: Piece adjacency structure, pieces with ground contact are colored white

Then:

1. Take pieces within certain y-coordinate threshold and sort them by their distance to the nearest ground centroid
2. Take piece with largest distance and perform a breadth first graph traversal originating from each ground component and check whether each remaining piece would be still connected to at least one ground component if the candidate was removed.
3. Take piece with largest distance and calculate the removal direction. This is a direction in which the piece can be removed without violating the directional constraints caused by adjacent pieces.
4. If there exists such a direction, intersect rays originating at each vertex of the candidate and going in this direction, with the remaining pieces to check whether the piece can be removed all the way.
5. If any of those criteria fail, take the next candidate from the sorted array.
6. If all candidates within a given y-threshold have failed, increase the threshold. This leads to an assembly that is less layered, but we have more potential candidates to remove.

Ground-pieces are ignored while there are still other pieces left.

3.2.1 Calculating a disassembly direction

Nnaji [10] describes the set of directions, in which a piece can be disassembled as an open convex polyhedral cone

defined by spanning vectors, that are created by intersecting all halfspaces defined by the geometric planes of mating faces.

As mentioned earlier, [10] uses heuristics to reduce the set of possible directions. In my algorithm, I further reduced the directions, only regarding one vector which is calculated by maximizing the minimal distance to any mating face. This results in a visually centered placement of the piece when assembling. To find this direction, I used a linear programming method taken from [1, 8] for finding the interior point in a convex polyhedron. The mating faces each define a halfspace that can be written as

$$aj*x1+bj*x2+cj*x3-dj<=0, j=1..n$$

We can extend them by one dimension to

$$aj*x1+bj*x2+cj*x3-dj+x4<=0, j=1..n$$

with d being always 0 as we construct the polyhedral cone with its top located on the origin. As the cone is by definition open if the piece can be disassembled, we have to close the polyhedron to find a point inside, where the position coordinates can be interpreted as our wanted directional vector.

This is done by adding the equations.

$$xj<1, xj>-1, j=1...3$$

Now we maximize x4 with a linear program. If we get a non-zero-length solution we have (x1, x2, x3) as our ideal direction. If, on the other hand, we get (0, 0, 0) as a solution, we know that the polyhedron defined by the normal vectors of the mating faces is not a convex open cone but a closed space. This means that there is no valid disassembly direction for the current piece in the current assembly state so we have to look for another piece that can be removed. Not getting a valid solution from the linear program also means that the normal vectors of the mating faces span the *R3* with positive linear combinations, while a valid solution means this is not the case. This is shown in Figure 3, where the scenario is shown simplified in 2 dimensional space.

3.3 The robot setup

When designing a robot configuration for our purpose, we have to consider mainly two key aspects:

- How many robots do we need to achieve the task?
- What kind of gripper do we need?

As the piece has to be picked up from an arbitrary starting position lying on the ground, it cannot be assumed that the faces that can be used to pick it up can be also used to place the piece into the assembly. Therefore for many pieces a regripping operation has to occur at some point which necessitates a second robot arm.

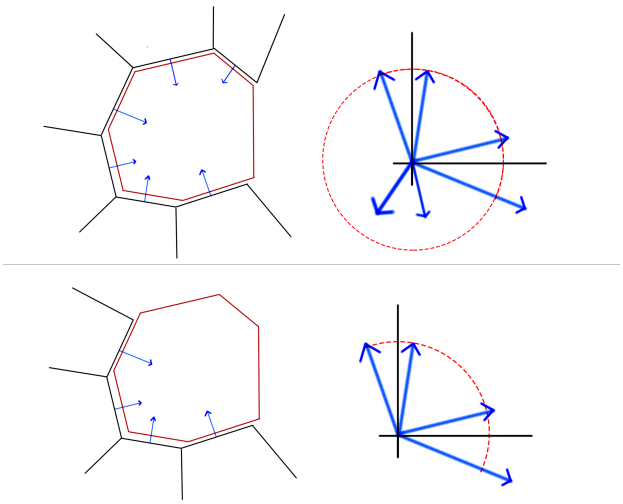

Figure 3: Lower section: disassembly direction exists, normal vectors don not span *R2*. Upper section: normal vectors span *R2*, disassembly direction does not exist.)

The gripper choice is a bit more complicated. The mechanics of grasping are rather complex, but simplified, a grasp that achieves force closure needs 4 fingers with friction, which have to attach to faces which normal vectors together positively span the 3 dimensional space (among other criteria) [13]. Unluckily it is impossible to have on the one hand a set of mating faces (Here defined as F_{m}) for a piece that have normal vectors together spanning an open polyhedral cone and on the other hand a set of four faces not containing any of F_{m} but together spanning the *R3*.

It is also intuitively obvious, that for example placing a last missing piece to a sphere is not possible when the piece is grasped by four fingers. See Figure 4. As a friction-based grasp can not be relied on to work for every piece, we'll take the physically less realistic approach of using a one finger gripper with a universal suction cup.

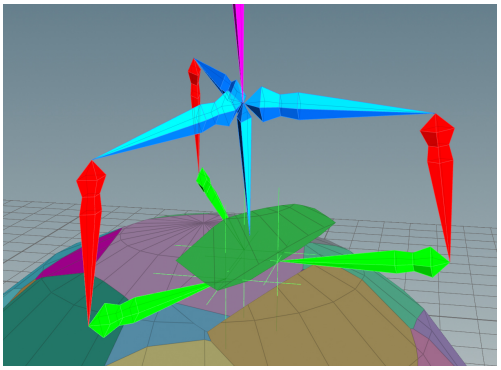

Figure 4: Impossible insertion with four finger gripper

3.4 Finding attachment faces

In general we need two faces per piece where the robot finger can attach onto. One for the first robot picking up the piece, one for the second robot placing it into the assembly (I will call this docking in the following descriptions). The two faces have to meet certain criteria. If one face meets the criteria for both, it is not necessary to regrasp and the piece is only manipulated by one robot.

The criteria are

- accessibility: a face that, when extruded infinitely, intersects the piece, will be discarded as a candidate
- feasible direction: the normal has to be within a certain angular distance to the direction from which the robot finger should attach. This angular constraint can be set by the user to fit better the specific robot geometry.

After discarding any faces that don not meet these criteria, a heuristic is used to determine the final choice. To achieve mechanical stability, we search for the face that is closest in distance to the docking respectively pickup direction extended from the centroid of the piece. I will call these two faces docking and pickup face, see Figure 5 for visualization.

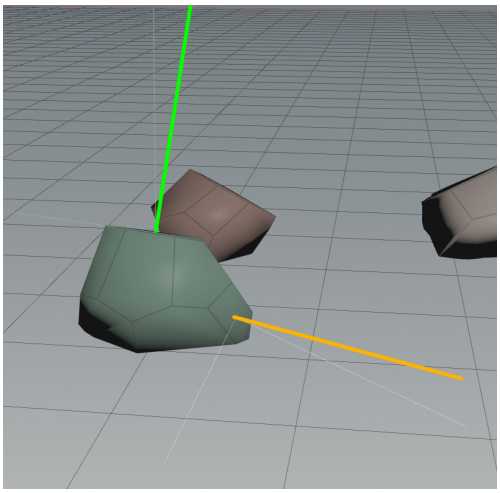

Figure 5: The normal of the pickup face in green, the normal of the dock face in orange. In case the two are within a certain threshold of directional similarity, the dock vector becomes obsolete as the first robot completes the entire operation.

3.5 Generating the motion paths

When the assembly sequence has been generated, we not only have an order in which the pieces can be assembled, but also a vector for each piece that defines the assembly direction. As we already know the rest transformation (position and orientation) and the final transformation in the assembly, we only need a few more keyframes for each piece that can then be used to interpolate a continuous motion.

The final assembly position of a piece is approached in the assembly direction, so one obvious keyframe can be defined by translating the final position backwards along the assembly direction. As it is guaranteed by the assembly sequence algorithm that this direction is collision free for the to-be assembled piece, we can move the piece in this direction as far as we want (or as far as the robot is capable of moving it).

Another important keyframe is right after the start position. Picking up the piece should happen in a vertical direction, as only this will guarantee that no other piece is intersected in the process. So we define the second position to be at some height above the start position. This point is also a convenient choice for the regrasping step, so we orient the piece along the y axis with the docking face towards the second robot arm, to guarantee collision free accessibility when the regrasping occurs. Additionally, to avoid potential collision with the assembled geometry when moving from the regrasp to the approach point, we introduce one more point that is located vertically above the approach point, so the piece is guaranteed to move above the object geometry. This leads us to our main steps of the per piece motion, also shown in Figures 6 and 7:

1. rest
2. picked up/regrasp
3. intermediate step for collision avoidance
4. approach
5. assembled

3.6 Generating Robot Motion

As we want to keep our system modular and decoupled, we want to define the least amount of necessary data to define the robot motion. The defining data is therefore limited to the transformation frame of the of the gripper directly attaching to the pieces. All other transformations for various robot parts have to be adapted to follow this transformation, requiring inverse kinematics.

Obviously when attached to a piece, a robot has to follow its motion exactly. Other than that, we just have to plan the motion in a way that there will not be any collision with the object geometry or with the other robot. To achieve the latter, we introduce a retracted position, to which the gripper of each robot moves when not manipulating the piece to allow the other robot to move freely.

Following on these prerequesites, a high level interface has been created to construct the exact per piece animation based on those key steps. The user can set the exact timing of each step and can create additional steps using the main

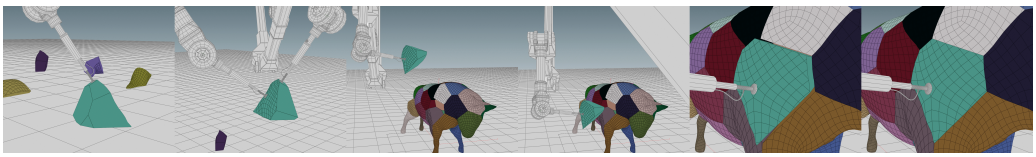

Figure 6: The key stages of the animation

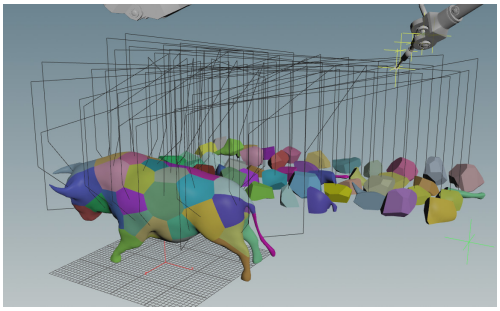

Figure 7: The motion paths for each piece

steps as reference. The interface for the piece animation allows the user to choose for each keyframe:

- The reference position (Start, End and From Previous keyframe)
- The offset direction to the reference position (along dock face normal, pickup face normal, disassembly vector) and the offset length
- An additional absolute offset vector
- The orientation chosen from start, regrasping, end or previous keyframe.
- A relative time from 0 to one, representing a normalized interval of the time required to move one piece from its start- to its end position including robot approach/retraction.

The keyframe system for the robot, as the robots do, attaches to the animation of the pieces. So a keyframe is either referencing the retracted position without further options, or one of the piece keyframes. In the latter case, an offset along the normal of the attachment face or along the assembly vector can be specified in addition to an absolute offset vector. This enables the user to specify additional approach/retraction points relative to the piece position.

With this setup it is then possible to interpolate a transformation for the piece and the two robotic grippers for any intermediate state between rest position and end position. Positional interpolation is trivial, orientation interpolation is done using quaternions to ensure a smooth SLERP motion. By sequencing each 0 to 1 animation for the individual pieces according to the pre-calculated assembly sequence we get the necessary data for the complete assembly animation.

3.7 Constructing the robots

The robots have to meet a set of criteria to be able to realistically assemble the object:

- They need to be, in their most extended state, long enough reach every piece from its start to its end position
- They need to be positioned so that they don not intersect each others retracted state in any animation state
- They need to be able to transform the end effector (single finger gripper) with 6 degrees of freedom in relation to the next segment of the arm.
- The end effector needs to be long enough for approaching even piece end positions inside a narrow corridor with no guarantee that the next arm segment has an aligning orientation.

This has to be done manually, but a robot design can be chosen in a way that optimizes for a wide range of assembled objects. For example a long arm (compared to the object size) with a long and thin end effector can be constructed so that it fits every object size, although the size relations would lead to an unrealistic appearance. Figure 8 shows an example robot configuration, meeting the necessary properties.

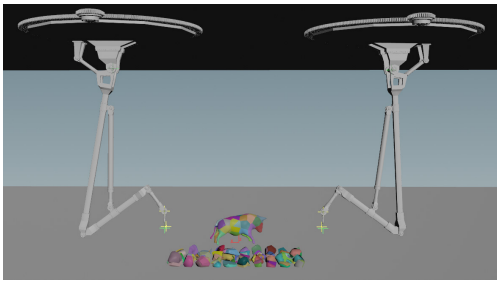

Figure 8: Example robot configuration

4 Implementation in Houdini

Houdini is a commercial 3D application that has a great focus on proceduralism and data transparency. The heart of it is its nodegraph paradigm, where each operation is represented non-destructively. It also allows the user to very naturally work with custom geometry attributes and

custom geometry manipulation and was therefore chosen as the framework in which to implement and develop the program described in this paper.

The advantage of using this application was clearly its vast amount of existing features and procedures, which are implemented in a very robust way and allow for rapid experimentation and the ability to quickly visualize and manipulate each state and its data interactively.

On the other hand, although programming custom code is easy for simple operations, there is no advanced debugging that goes beyond logging or inspecting resulting data in a spreadsheet. Never the less the ability to quickly inspect various stages or branches in the complex nodegraph representing the data flow of the program is very helpful.

4.1 Digital Assets

"Digital Assets" in Houdini are subcomponents compiled into a separate file and therefore reusable in various scenes. They usually have custom interfaces exposing important parameters and references to other scene objects as input. Output is, as it is generally in Houdini, realized by referencing either the asset itself or nodes in its subnetwork.

4.2 Implementation of Modules as Digital Assets

The Modules of the program have already been described in 3.1, here I will explain some implementation specifics.

Fracturing Module

This asset takes a geometry reference for the object to fracture and for the geometry to take its reference points from. There is also an option to specify the duration of the presimulation calculating the stable rest transform and if the pieces should be repositioned to their original x and z coordinates after the y coordinate and the orientation for the rest transform have been calculated.

Piece assembly Module

This asset, in addition to the inputs described in 3.1, exposes the parameters to control details of the assembly motion paths as described in 3.5 and 3.6 and parameters to control the timing of the animation and directional thresholds for choosing dock- and pickup faces.

A lot of the logic in this modules is not realized using a pre-existing node, but the "VEX-Wrangle" node, which lets you write code in Houdini's own VEX language, that is then compiled and runs, depending on the context, highly parallelized. For solving the linear programs described in 3.2.1, the lp_solve was wrapped with Houdini's API, called HDK (Houdini Development Kit).

Object	Pieces	Fracture Time	Sequence Time
bull	45	4.7	7.3
bull	100	5.3	27.3
bull	150	6	65.4
bridge	45	5.5	3.7
bridge	100	6.0	37.7
bridge	150	6.7	259
box	45	11.5	3
box	100	13.7	67.3
box	150	13.8	644.9

Table 1: Performance test results, in seconds

5 Performance

The Performance was tested on the test objects in Figure 9 using a piece count of 45, 100 and 150. The processor on the test system was an Intel i7-5820K Hexacore (The performance relevant parts of my implementation are single threaded). The bull was chosen because it is a fairly complex real-world object with a lot of concave areas and four independent ground-contacting components. The "bridge" was used early on to test concavity and multiple ground contacts. The box is was used because of its geometric simplicity. The results are shown in Table 1. The "Fracture Time" is the time taken to initially prepare the object by the first module, "Sequence Time" is the time spent calculating the assembly sequence. The "Sequence Time" is also plotted in Figure 5, as it is much more linked with the work described in this paper.

Interestingly the bull is by far the fastest for higher piece numbers and the box the slowest. As Houdini's VEX language does not offer access to the system time and therefore does not allow measuring time inside of nodes, I can only guess that this is caused by a higher degree of interconnectivity. It is planned to circumvent this restriction by writing a custom function with the API and then doing further measurements.

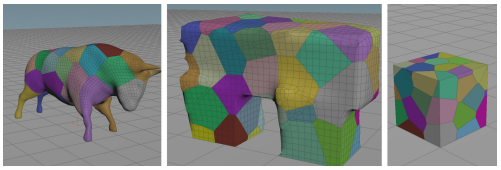

Figure 9: Test objects: bull, bridge, box

6 Conclusions and Outlook

The presented work achieves good results for simple objects but is not perfect by any means. In this section I describe scenarios that are likely to break the algorithm and future improvements to tackle these problems are proposed.

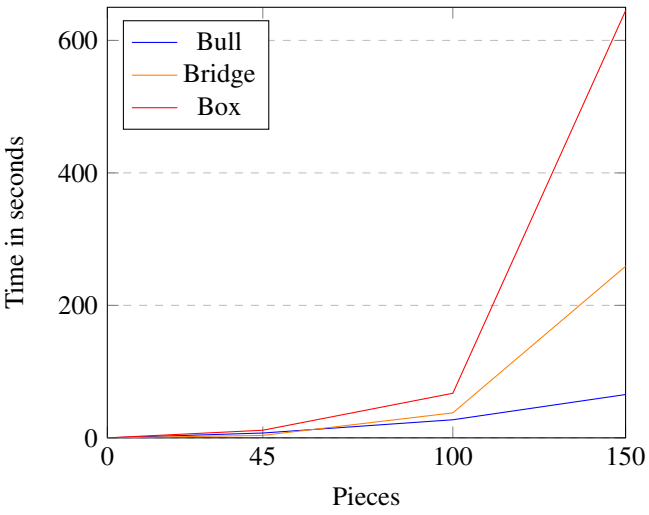

Figure 10: Performance test results diagram

6.1 Problematic object characteristics

As the assembly direction for each piece is calculated only once per piece and then checked for intersections in its direction, objects with a high concavity are prone to failure. See Figure 11 for two examples where the algorithm fails to generate a valid sequence, although there exists one. The left object has a lot of pieces that could be approached from the side, e.g. the camera direction, but as the normals of the mating faces between the pieces don not face either side, but lie on the plane of the object, the assembly vectors are generated in a way that they intersect.

The torus is similar, the assembly vector for some of the inner pieces intersect with other pieces. (Note that a higher number of pieces lead in this case to the successful generation of an assembly sequence)

6.2 Potential improvement

The assembly algorithm should be adapted to include more disassembly directions in its search for a removable piece. This would greatly reduce the probability of running into a situation like explained in 6.1. Also the heuristic of choosing the next piece to disassemble described by [10], which suggests choosing the piece with the lowest count of mating neighbours, should be explored and compared to the current approach.

References

[1] Various authors. qhull. `http://www.qhull.org/html/qhalf.htm`, 1995.

[2] L. S. Homem de Mello and A. C. Sanderson. And/or graph representation of assembly plans. *IEEE Transactions on Robotics and Automation*, 6(2):188–199, 1990.

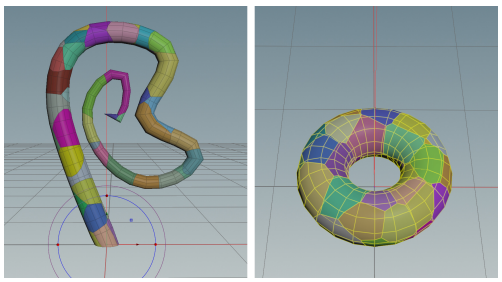

Figure 11: Problematic objects

[3] M. I. Shamos F. P. Preparata. *Computational Geometry: An Introduction*. Springer, 1985.

[4] T. De Fazio and D. Whitney. Simplified generation of all mechanical assembly sequences. *IEEE Journal on Robotics and Automation*, 3(6):640–658, 1987.

[5] J. Fisher. Visualizing the connection among convex hull, voronoi diagram and delaunay triangulation. In *37th Midwest Instruction and Computing Symposium*, 2004.

[6] M. Overmars M. de Berg, M. van Kreveld and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000.

[7] F. Mercier. and O. Baujard. Voronoi diagrams to model forest dynamics in french guiana. In *Proceedings of GeoComputation*, volume 97, pages 161–171, 1997.

[8] J. Meyron and P. Alliez. Cgal. `http://bit.ly/2nefC2u`, 2014.

[9] A.K. Garga N.K. Bose. Neural network design using voronoi diagrams. *IEEE Transactions on Neural Networks*, 4(5):778–787, 1993.

[10] B. Nnaji. *Theory of Automatic Robot Assembly and Programming*. Chapman and Halll, 1993.

[11] J. O'Rourke and J. E. Goodman. *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.

[12] D. Stora, P.-O. Agliati, M.-P. Cani, F. Neyret, and J.-D. Gascuel. Animating lava flows. In *Graphics Interface (GI'99) Proceedings*, pages 203–210. Kingston, (1999).

[13] A. Sudsang and J. Ponce. New techniques for computing four-finger force-closure grasps of polyhedral objects. In *Proc. IEEE International Conference on Robotics and Automation*, volume 1, pages 1355–1360, 1992.