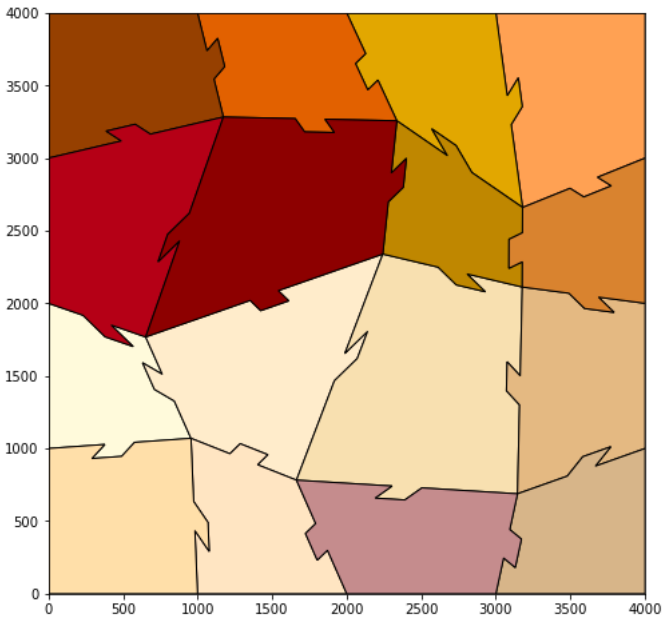
Ecole Polytechnique Fédérale de Lausanne
Computer Graphics and Geometry Laboratory

Interlocking Puzzle Assemblies

Maxime Fellrath

Doctoral Assistant:
Ziqi Wang

Professor:
Mark Pauly



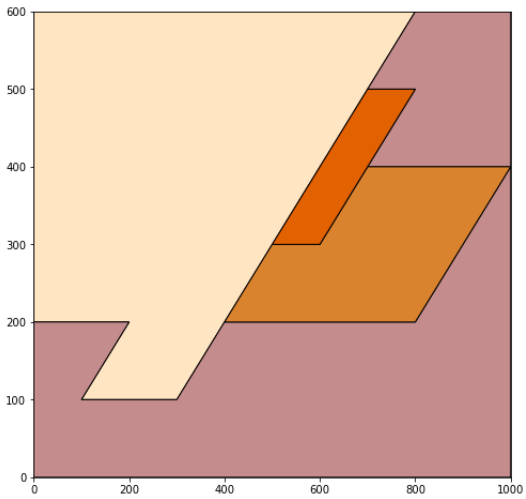
Lausanne, 2019-2020

Table of Contents

1. Introduction

1.1 Interlocking assemblies

Classic Interlocking assemblies are complex geometric structures made to be assembled piece after piece in a way that once assembled, only the last appended piece can separate from the structure. Figure 1.1[1] is an example of such an assembly:



We can see that the only way the puzzle disassembles is by removing the light yellow piece.

1.2 Multiple moving parts assemblies

The main goal of this bachelor project was to study and design joints for interlocking assemblies where no part can move on it's own independently from the other parts. A group of the pieces or all pieces have to move simultaneously to disassemble the assembly. In figure 1.1 and 1.2[2] we can see that the triangle can disassemble only if at least 2 pieces move at the same time:

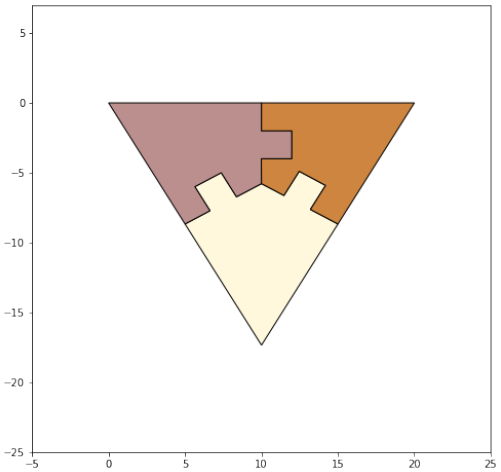


Figure 1.1: Interlocking triangle

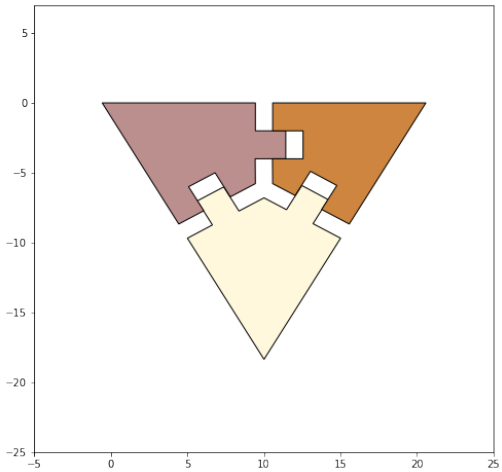


Figure 1.2: Opened interlocking triangle

[1] Assembly structure taken from [1]
[2] Assembly structure taken from [1]

The objective of this project at first was to make a testing algorithm that would tell whether a structure could be disassembled and if it needed more than one piece to move to do so. Then, once the testing algorithm was implemented, the second part was to make an algorithm that would design joints between pieces in a given structure in a way to give a disassembling possibility if multiple parts move simultaneously. The original follow-up of this project was to expand to the third dimension, however after some reflection we decided to stay with the second dimension and, using these joints, create a grid puzzle that would be possible to assemble only by sliding pieces together on the same plane.

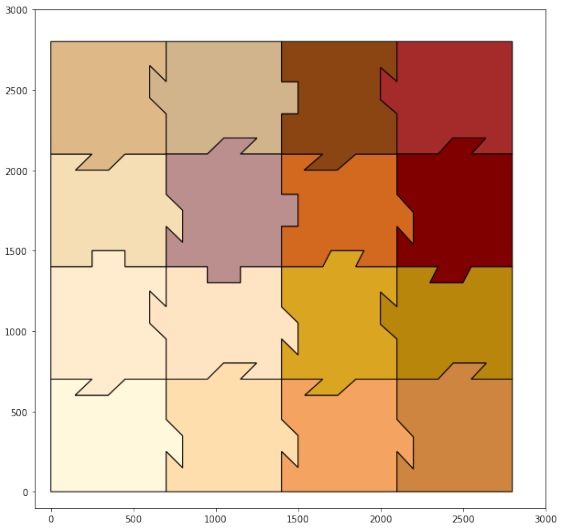


Figure 1.3: Puzzle grid

1.3 Disassembling logic

The wanted disassembling in this 4 by 4 grid is to first separate the entire grid in 4 pieces containing each 4 smaller pieces like shown down below:

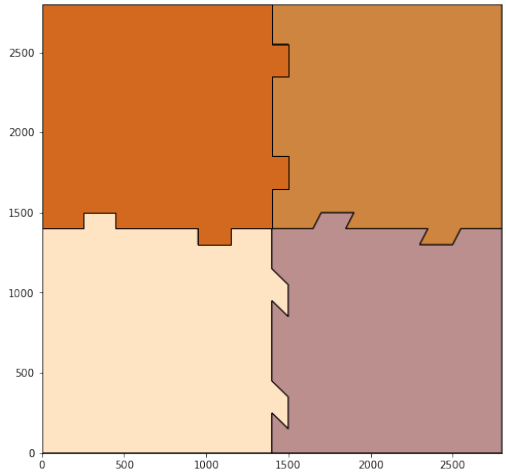


Figure 1.4: Same puzzle grid as Figure 1.3, but all groups of 4 pieces merged

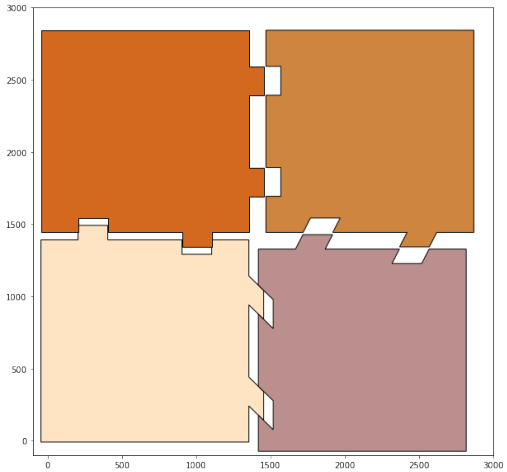


Figure 1.5: Disassembly of the grid

And finally separate all 4 smaller pieces:

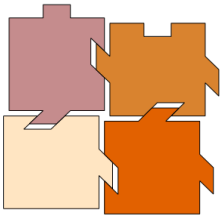


Figure 1.6: Disassembly of Figure's 1.4 bottom left corner

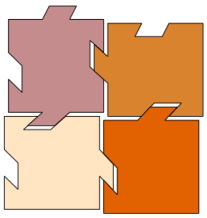


Figure 1.7: Disassembly of Figure's 1.4 bottom right corner

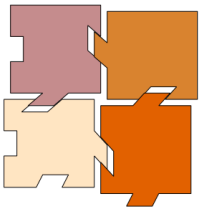


Figure 1.8: Disassembly of Figure's 1.4 top right corner

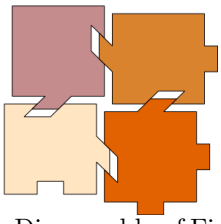


Figure 1.9: Disassembly of Figure's 1.4 top left corner

In theory by adding all moving vectors of each piece to their corresponding "4-pieces group" vector it is possible to separate the whole puzzle at once like shown below. In practice this should almost be impossible because you would require to move every piece in different direction at the same time.

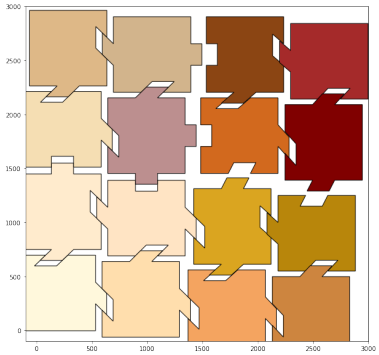


Figure 1.10: Disassembling at once the Figure 1.4 puzzle

In the end, we randomized the grid points to be able to create more puzzles, and maybe change their corresponding complexity, even though the assemble/disassemble logic stays exactly the same for all puzzles.

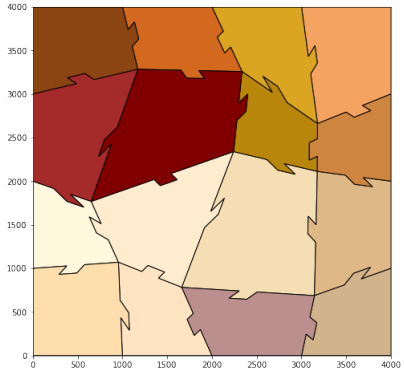


Figure 1.11: Random Puzzle with random seed 2777

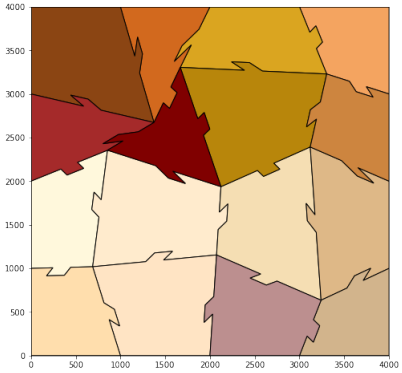


Figure 1.12: Random Puzzle with random seed 97

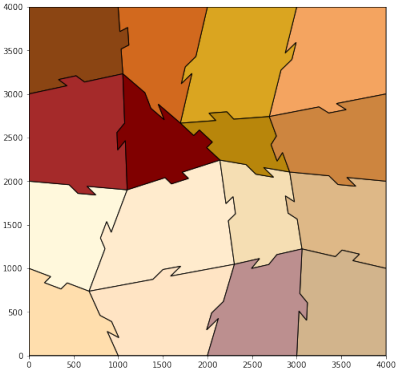


Figure 1.13: Random Puzzle with random seed 8163

2. Analysis

2.1 Theory

For the analysis we used this inequality based test[Ziqi Wang, Peng Song and Mark Pauly 2018][1]:

$$\max(\sum t_{ij})$$

$$\begin{gathered} s.t.(v_j - v_i) \cdot n_{ij} \geq 0, \forall interfaces(i,j) \\ 0 \leq t_{ij} \leq 1, \\ v_r = 0 \end{gathered} \tag{2.1}$$

where v_n is the direction vector of piece n, t_{ij} an auxiliary variable, n_{ij} the normal between the pieces i and j and v_r the velocity of a random piece which is fixed to avoid the whole puzzle to be able to move in any direction.

2.2 Implementation

Even if it took some time to figure out how to store the pieces, draw them and make the interactive display, I spent most of the time trying to understand how to solve the linear inequality and making sense of the result.
Using the cvxpy library to solve the inequation was then pretty straightforward and quickly gave satisfying results.

Algorithm 1 Testing algorithm

1: Matrix A ← Constructs_A()
2: x ← Variable_vector
3: tij ← Variable_ vector
4: constraints ← [(A*x-tij) ≥ [0,0...0,0], tij[i] ≥ 0, tij[i] ≤ 1 ∀ i]
5: objective ← Maximize(sum(tij))
6: result ← solve(constraint, objective)

The constructs_A() function construct the A matrix using the normals between all touching edges of neighbouring pieces. As the vector x in the algorithm represent the velocities v of the pieces, we can translate the $(v_j - v_i)\cdot \mathrm{n}_{ij}$ in the equation 2.1 by linearity of the dot product to $v_j \cdot \mathrm{n}_{ij}$ - $v_i \cdot \mathrm{n}_{ij}$. Then we are multiplying matrix A and the velocities vector v, so we need to store in matrix A for all touching pieces and for all of their touching edges $[-normal_{ij}, normal_{ij}]$, where $normal_{ij}$ represents the normal going from piece i to piece j.

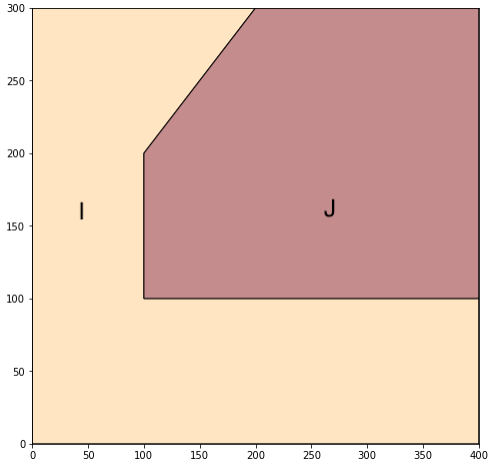


Figure 2.1: Simple piece with 3 touching edges

Resulting A matrix

$$\begin{bmatrix} -1 & 0 & 1 & 0 \\ -0.7 & 0.7 & 0.7 & -0.7 \\ 0 & -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} v_{ix} \\ v_{iy} \\ v_{jx} \\ v_{jy} \end{bmatrix}$$

2.3 Results

Once implemented it showed us not only if the piece was deadlocking, but also, in case it was not, the directions in which each piece should move to disassemble.

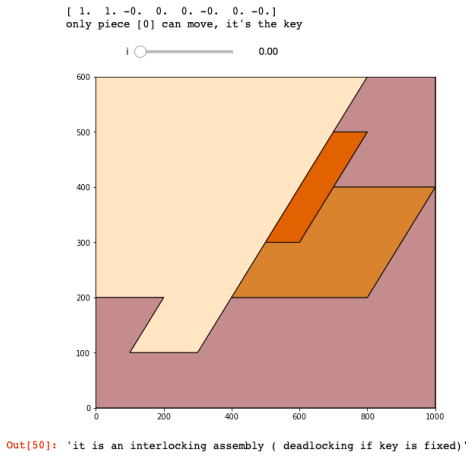


Figure 2.2: Interlocking assembly; only the key, i.e. the white piece is allowed to move.

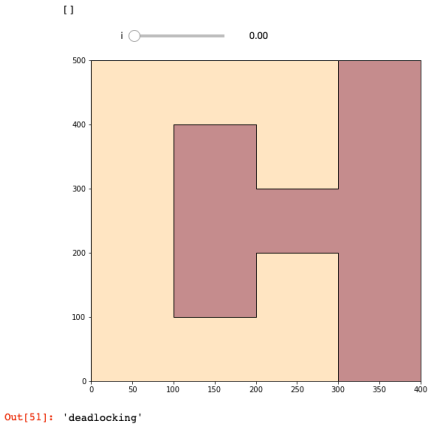


Figure 2.3: Deadlocking assembly.

The given vector v, here in Figure 2.2 [1. 1. 0. 0. 0. 0. 0. 0.] represent the velocities of the pieces. Hence, piece i moves in direction v[2*i]on the x axis and v[2*i+1] on the y axis. For those puzzles we can differentiate the pieces by their color for example, the light yellow is the first piece.

3. Design

3.1 Theory

The goal of the design step originally was to split a square surface in a voronoi, and design the joints afterwards. However the shapes were too irregular, and often the joints between two different pieces would intersect. This is why I decided to go for a more classic 4 by 4 square grid at first, and then I randomized some points in order to make more interesting puzzles.

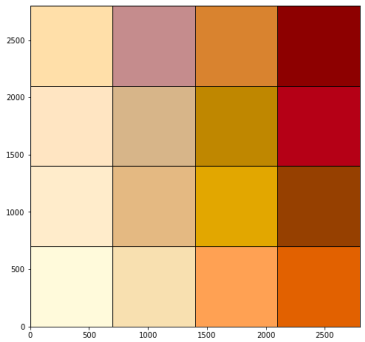


Figure 3.1: Basic square grid

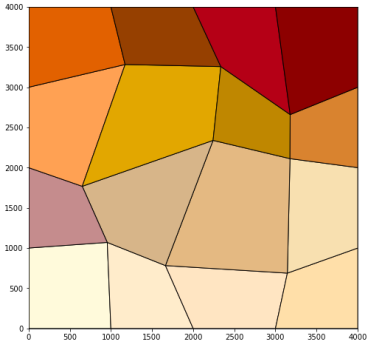


Figure 3.2: Randomized grid

After designing in this way the grids, I used a graph representation of it to find cycles of 4 pieces, and only kept the cycles on the extremities of the grid. Because I originally wanted to create more complex grid with different structures and shapes, I wanted my algorithm to be as general as possible. This is why I chose this cycle approach instead of simply putting the pieces on each quarter of the grid together.

Once the cycles were found I created joints for all neighbouring pieces. To do so, I first decided some basic velocities in which I wanted the pieces to disassemble. Then, for any touching pieces in all of the 4 parts of the first decomposition I added the the velocity of the two touching pieces to create the joint pointing toward that new direction. This process of joint creation is represented down below in figure 3.3 and 3.4. The orange arrow representing the orientation of the joint is created by adding the 2 black arrow, that represents the direction of the dark and light piece respectively. I finally created the joints between the 4 groups of four pieces using the same method.

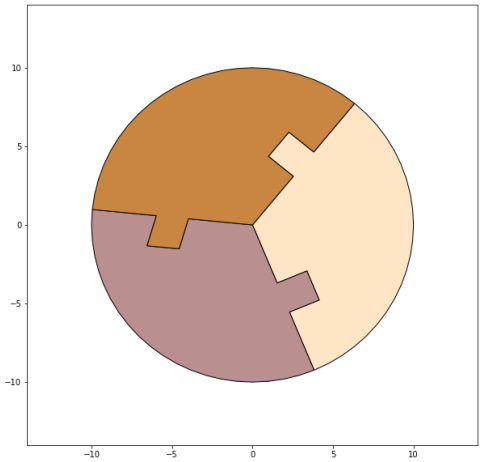


Figure 3.3: Joint creation process.

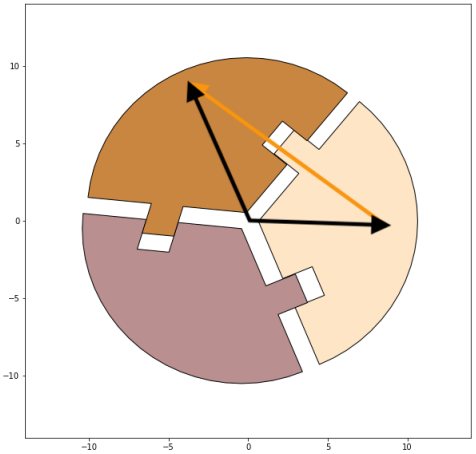


Figure 3.4: Joint creation process.

3.2 Implementation

Algorithm 2 Designing algorithm

```
1: Graph ←create_Graph_from_grid(Grid)
2: cycles ← get_cycles(Graph)
3: for cycle in cycles do
4:     for piece_i and piece_j in cycle do
5:         designJoint(piece_i, piece_j)
6:     end for
7: end for
8: for cycle_i, cycle_j in cycles do
9:     for piece_k in cycle_i and piece_l in cycle_j do
10:        if neighbour(piece_i, piece_j) then
11:            designJoint(piece_i, piece_j)
12:        end if
13:    end for
14: end for
```

3.3 Randomized joints angles

This method had the flaw that in the case of the regular Grid shown in figure 3.5 down below, the whole top row of the puzzle can move the same direction which is not the wanted disassembling way of this puzzle. This got fixed by subtly randomizing angles of joints which in theory does not allow anymore to disassemble the puzzle another way than the explained in the introduction.

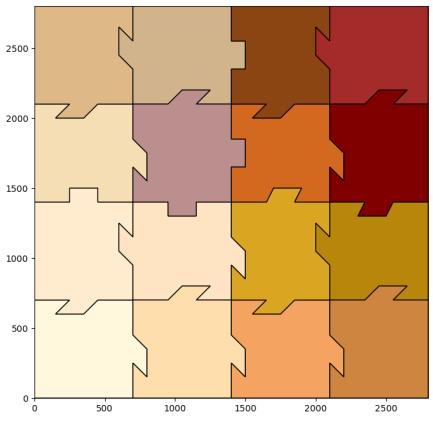


Figure 3.5: Puzzle grid with basic joints, right & left column, like top & bottom row can easily disassemble.

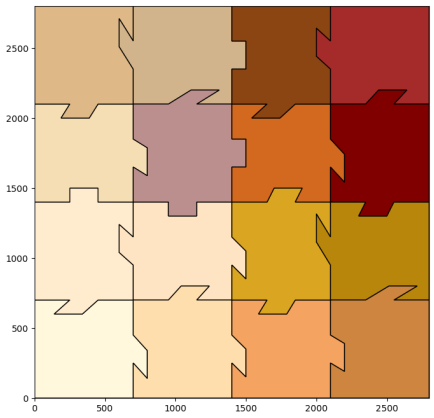


Figure 3.6: Puzzle grid with randomized Joints, preventing unwanted disassemble.

4. Program

4.1 Designing

To design a grid puzzle, and then make the laser cut possible, my doctorant Mr Wang wrote a code allowing me to launch the creation of a puzzle in a shell. By running the following command: "multipuzzle" a puzzle grid will be created and stocked in a rhino file, ready to be lasercut.

I then added arguments in order to precise the execution:

Table 4.1: multipuzzle command's arguments

Command	Default value	Description
-d	0	display the result grid, 0 for no, 1 for yes
-l	50	length of joints
-n	"none"	name of the piece stocked in a npy file
-o	"rhino/puz.txt"	output rhino file for lasercut
-r	-1	seed number for a random grid. if -1, the grid is not randomized
-s	2800	size of the square grid
-w	50	width of the joints

An example command:

```
Maxime$ multipuzzle -d 1 -l 80 -n example_test -o rhino/output_example.txt -r 6322 -s 4000 -w 100
```

Output Result:

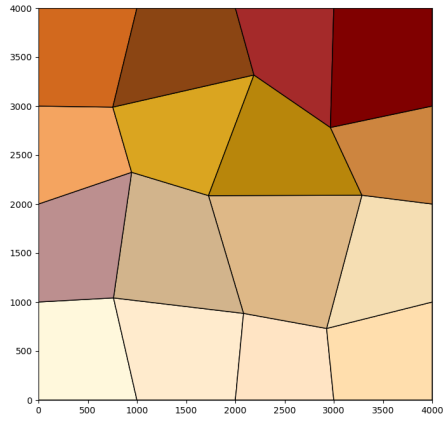


Figure 4.1: Output grid before the joint creation

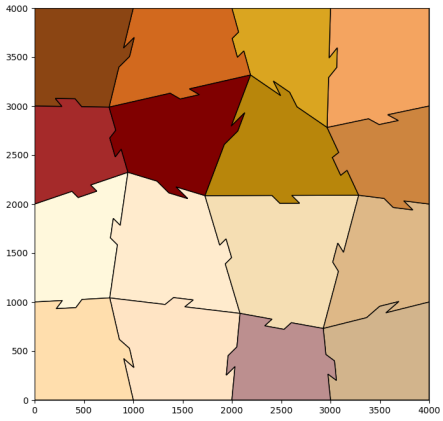


Figure 4.2: Output grid after the joint creation

```
piece created succesfully and saved as example_test.npy
```

4.2 Testing

To test the disassembling of the puzzle I have written a small python script to make sure the puzzle was not deadlocking: By running: `Maxime$ python test.py example_test.npy` it will test if firstly the 4 blocks of 4 pieces can disassemble, then if every of these blocks can also disassemble. The output generated down below shows us the resulting direction vectors as well:

```
[ 0.615  2.96  -2.96  -0.615 -0.615 -2.96   2.96   0.615]
Four blocks disassembling: OK

[-0.842  2.12   2.12  -0.842  0.659 -1.937 -1.937  0.659]
first Block disassembling: OK

[ 1.281  2.715  2.383  0.732 -1.074 -2.292 -2.59  -1.155]
second Block disassembling: OK

[-0.541  2.714 -2.557  0.026 -0.731 -4.082  4.555 -0.117]
third Block disassembling: OK

[ 0.391  2.313 -2.396 -0.474 -0.529 -2.963  2.534  1.123]
fourth Block disassembling: OK
```

Figure 4.3: Python test script output

For interactive assemble and disassemble plots, run the Testing.ipynb program in jupyter notebook, make sure to load in the "to_test" variable the wanted numpy array and then run the StudyMultipleMoving function:

```
StudyMultipleMoving(to_test, l = 4000)
```

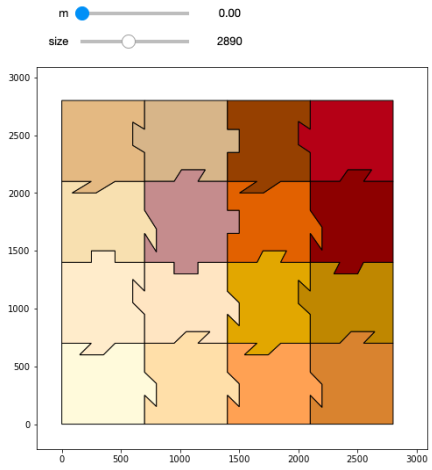


Figure 4.4: Jupyter Notebook grid visualisation

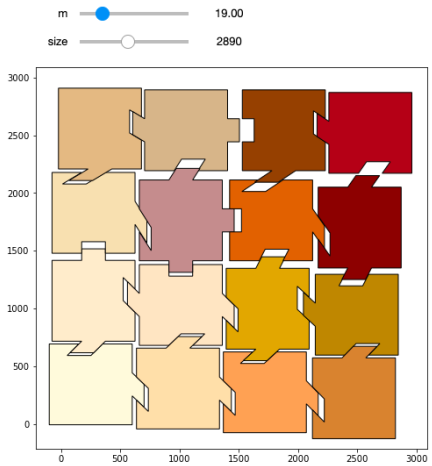


Figure 4.5: Jupyter Notebook grid visualisation

4.3 Images

To add Images over the puzzles and plot them in the testing.ipynb file, put an image in the ressources folder and run the following command:

```
StudyMultipleMoving(piece, image = "robert-bahn.jpg")
```

Have in mind that as all grids are square, a rectangle image will flatten or expand and may give a less satisfying result.

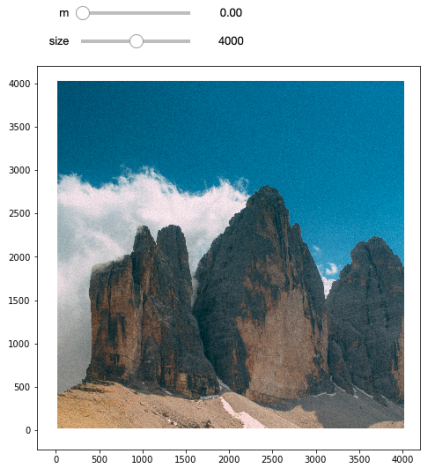


Figure 4.6: Jupyter Notebook grid visualisation with image; Photo by Robert Bahn

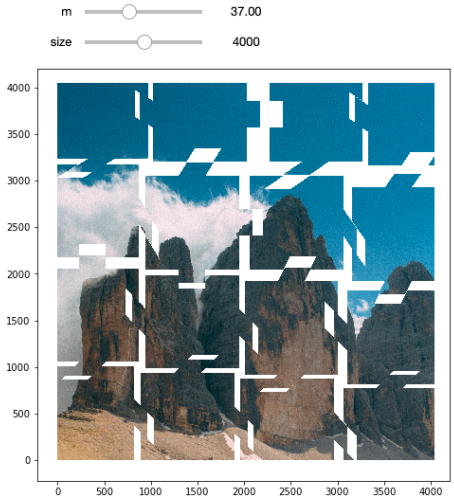


Figure 4.7: Jupyter Notebook grid visualisation with image; Photo by Robert Bahn

5. Conclusion

5.1 Project timeline

In the beginning of the project it wasn't very clear where it was headed. After some introduction by the first week understanding how the wildmeshing, polyfem, meshplot and Igl libraries work, we then decided to implement this interlocking test program. Only once the testing algorithm was implemented, we began to study multi-parts joints and how they could be designed. Afterwards we got a couple ideas for the project. The original idea was to upgrade the testing and design algorithm to the third dimension. However the implementation seemed too challenging. Then we thought of studying the forces of friction that would apply on the puzzle once printed. Even if this might have been a good idea, using the above mentioned libraries, we decided to go for the more playful goal of developing puzzles using the multi-parts joints. It was this idea that really motivated me and I would have liked to continue looking for other possibilities or a way to continue this project.

5.2 Final results

Although this project wasn't clearly oriented at these puzzles, I am very satisfied with the way it turned out. Not only I improved my programming skills, but I learned a lot of many python libraries and how to use them. It sometimes was difficult to understand where this project was headed and to make sense of the bigger picture, but once the puzzle objective was clear, it was really interesting to find this playful use of the testing and designing algorithm implemented, and this motivated me a lot. I am satisfied with the resulting grid but as I have not yet built them for real, I can only hope that playing at assembling and disassembling this kind of puzzle can be fun.

5.3 Continuation

Even if I only covered square shapes in this project, this method of puzzle creation could apply to many more kind of shapes containing different pieces. For example it could be interesting to build puzzles containing round pieces. There is a handful of possibilities.
However as I mostly covered the theoretic aspect of these puzzles, before making these puzzles for real we should cover the physical aspects. By that I mean for example finding the material that would hold the pieces together but not create too much friction, making the disassembling difficult. The optimal size of joints should also be looked into, as joints too thin and too short may not hold the pieces properly together but on the other hand making them too long and too thick may result in weak pieces.

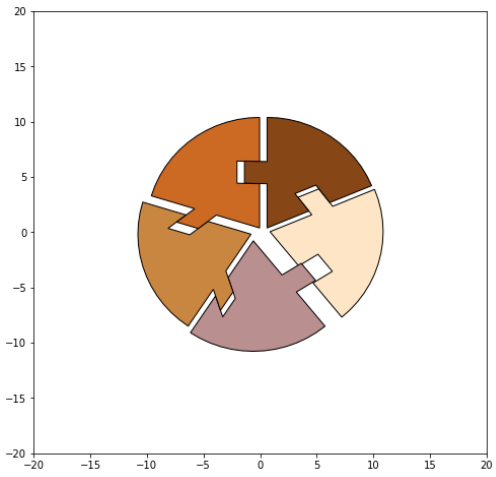


Figure 5.1: 5 parts disc assembly

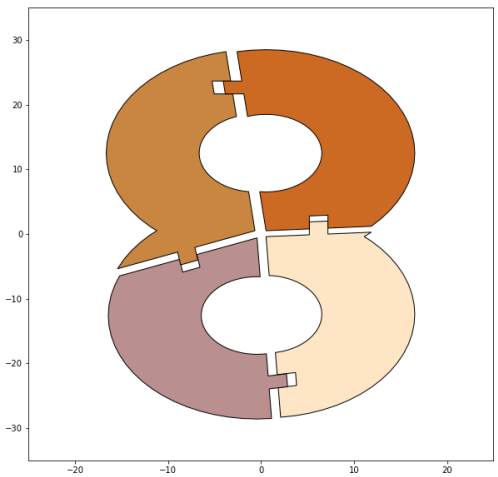


Figure 5.2: 4 parts eight assembly

Bibliography

[1] Ziqi Wang, Peng Song, and Mark Pauly. 2018. DESIA: A General Framework for Designing Interlocking Assemblies. ACM Trans. Graph. 37, 6, Article 191 (November 2018), 14 pages. 1, 5