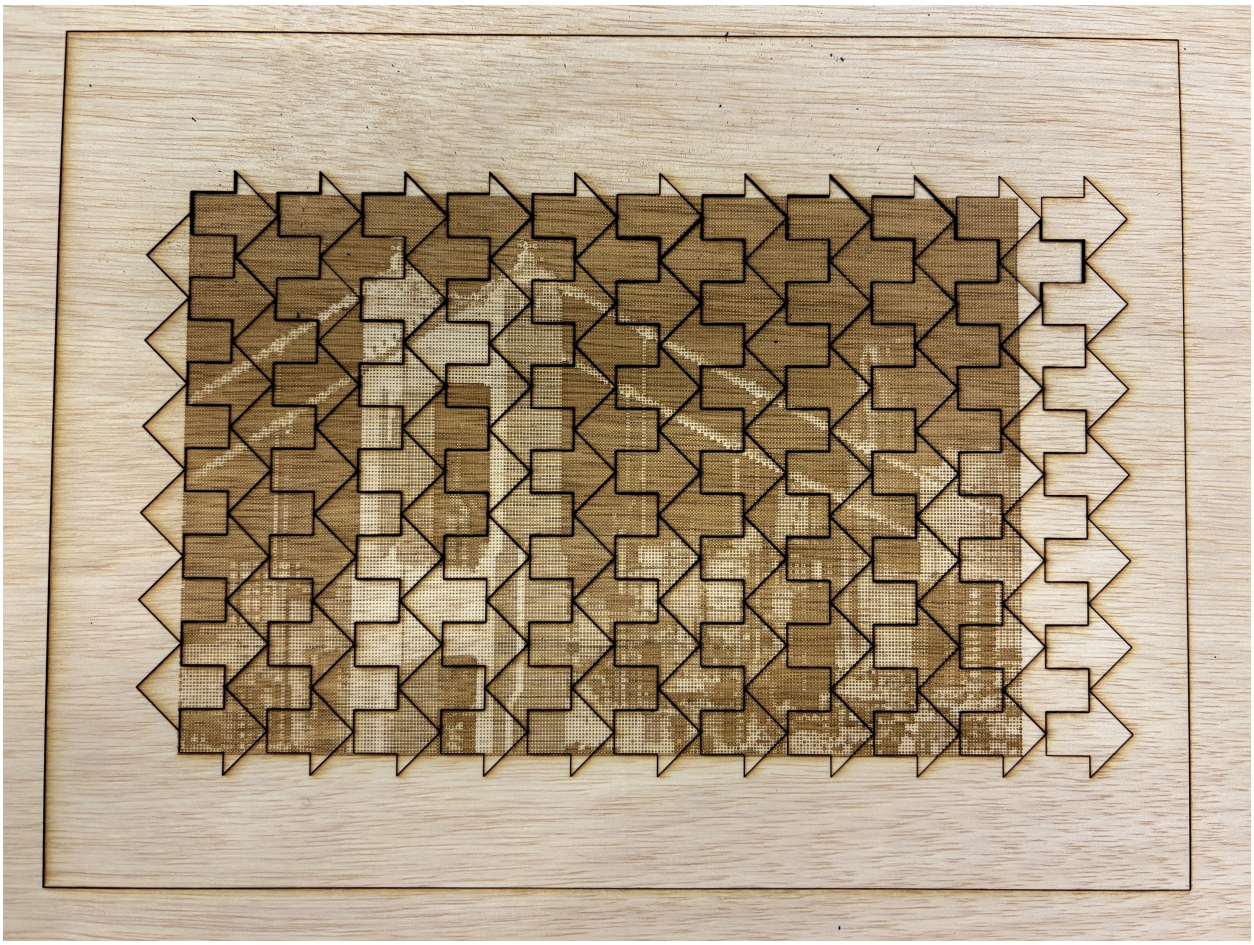
2D Dot Plotter Puzzles

Final Project - Computational Fabrication - ATLS 4519

By: Brett Rabbiner and Zeke Thoreson



For our final project, 2D Dot Plotter Puzzles, in Computational Fabrication (ATLS 4519), we decided to design a parametric puzzle generator. To give a bit more breadth, our project begins by converting an image into a 2D-dot-plot of circles

whose size is based on the brightness value of each pixel. Next, users select which style of puzzle they would prefer.

Inspired by jigsaw puzzles, Infinity Puzzles, tessellations, and a number of other geometric patterns, users can select between hexagon, square, triangle, and arrow shaped pieces while increasing/decreasing the number of pieces based on the size of them. One can even opt to go for more traditional style puzzle, selecting Voronoi based jigsaw pieces whose piece count and connecting pieces can both be adjusted based on the size preference of the user. All puzzles are generated using Grasshopper, a Rhino based plug-in, with incorporated python script. Each is 'baked' within Rhino to allow for easy upload to any laser cutter available. Though, while they are baked within a border, as every good gingerbread man knows you are meant to, "run, run, run, as fast as you can, you can't catch me," I'm an Einstein Tile...

Google Drive Containing All Final Files:

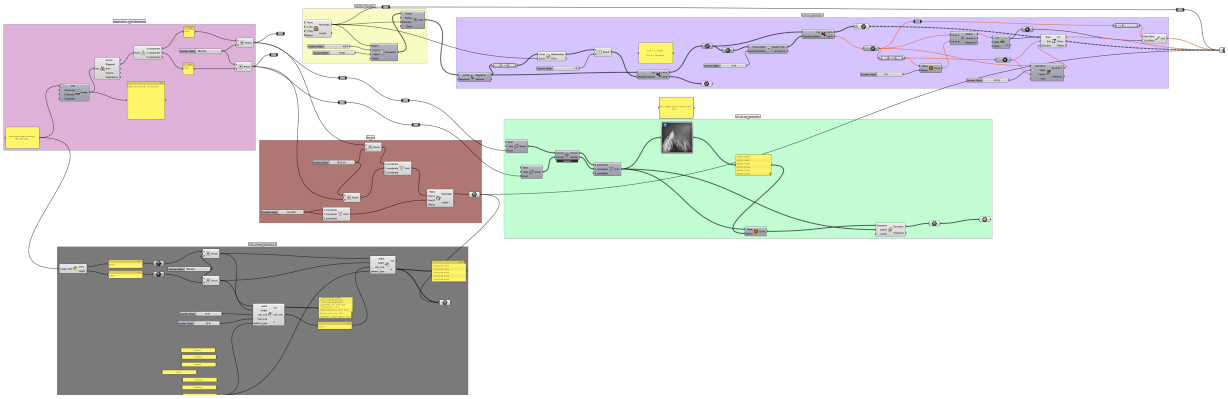
https://drive.google.com/drive/folders/1nc8aVaFNZ5jhZHBadOAcVHcrEYtu0Ovx?usp=sharing

Materials Used:

- 1/4" Sande Plywood Sheet

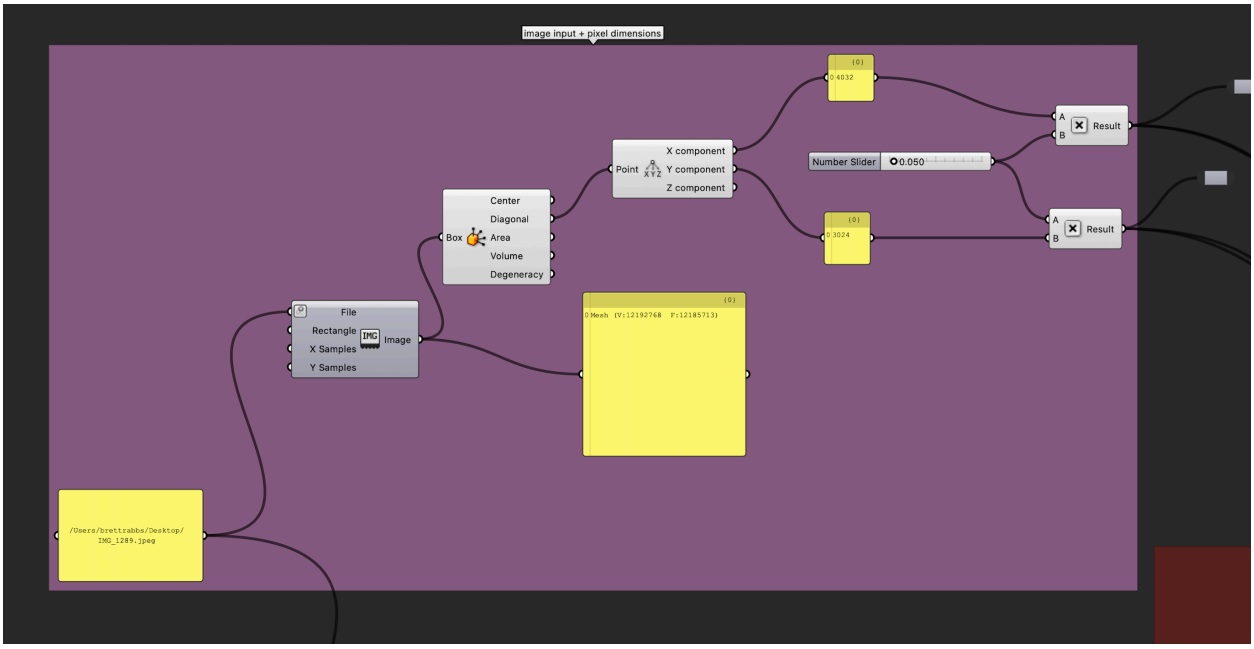
The Code and Grasshopper Files

Final Grasshopper File

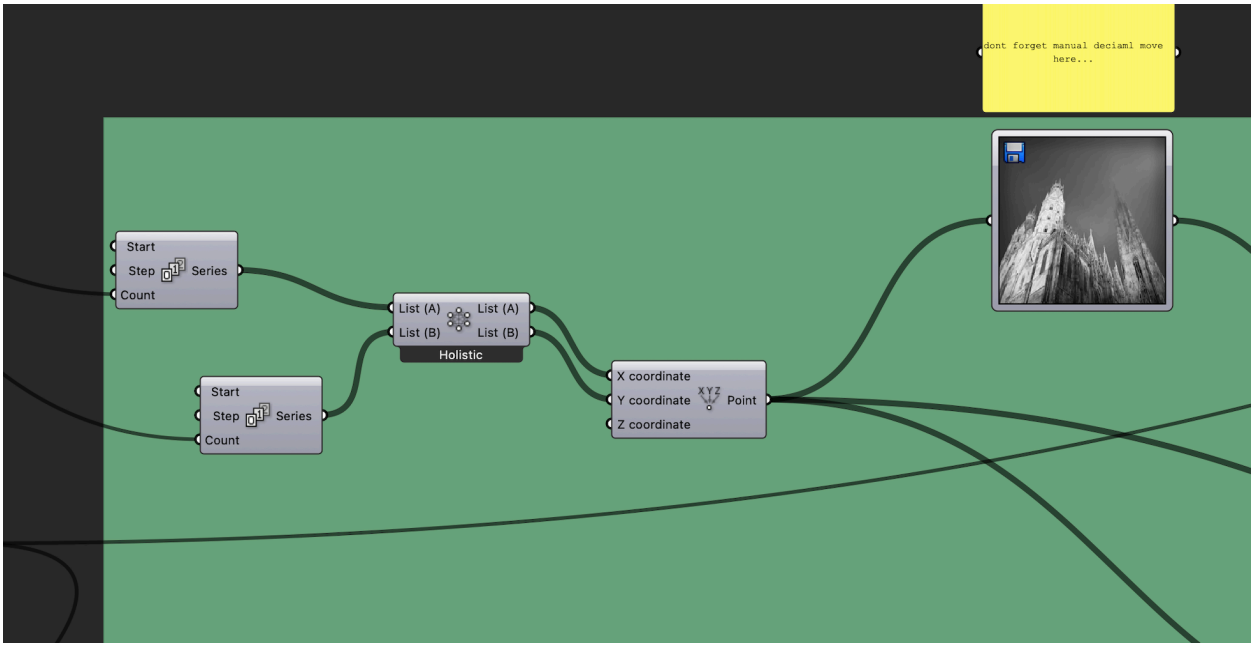


Full high definition photo of our Grasshopper file

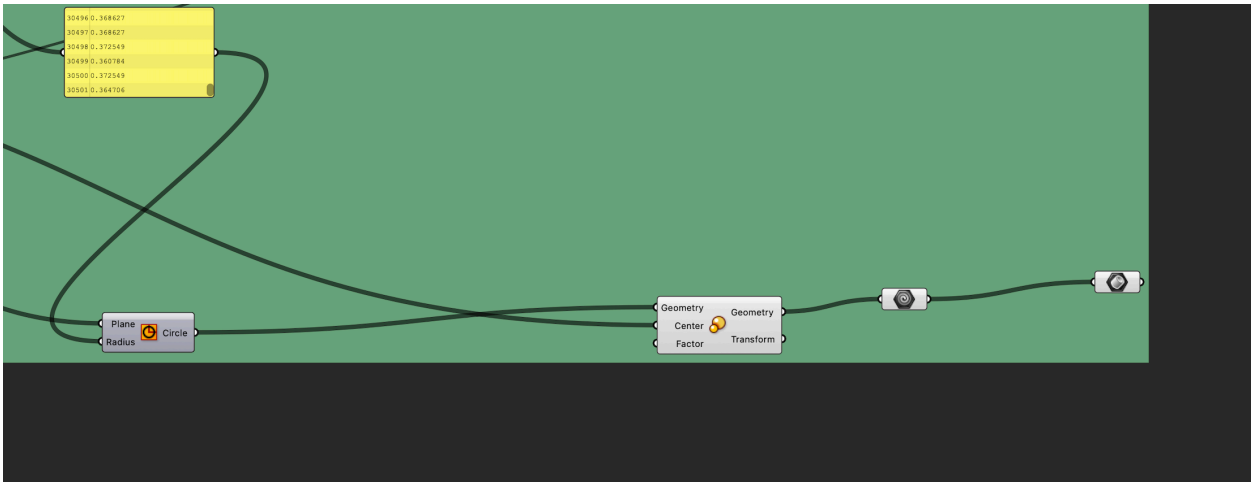
Walkthrough of Functionality:



This section deals with importing an image and getting the height and width of the image. The multiplication deals with the sizing of the image.



The two series components are populated by the multiplication components in the above image. From this we construct points spanning the image. The image sampler component returns values based off of the pixel brightness.

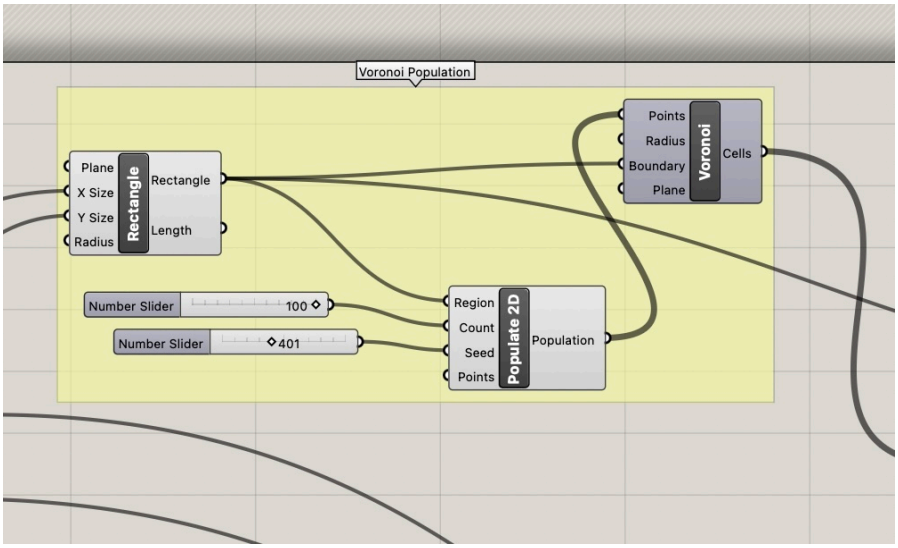


The circles are constructed with a radius of the returned values from the image sampler and baked.

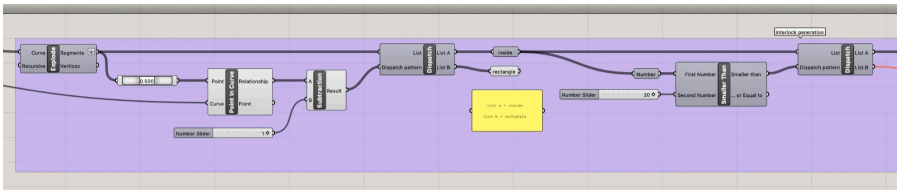
Below shows the given image and the Grasshopper output of that image.

Original image

Generated output

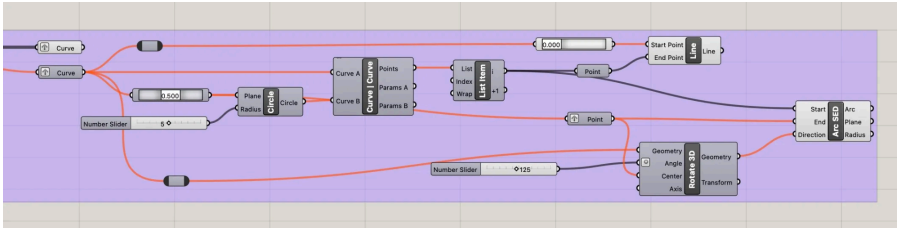


This section is how the Voronoi pattern is generated. Using the image border rectangle we bound the area that the pattern can be created within. The first number slider is how many random points are generated and the second deals with the randomization of point location.

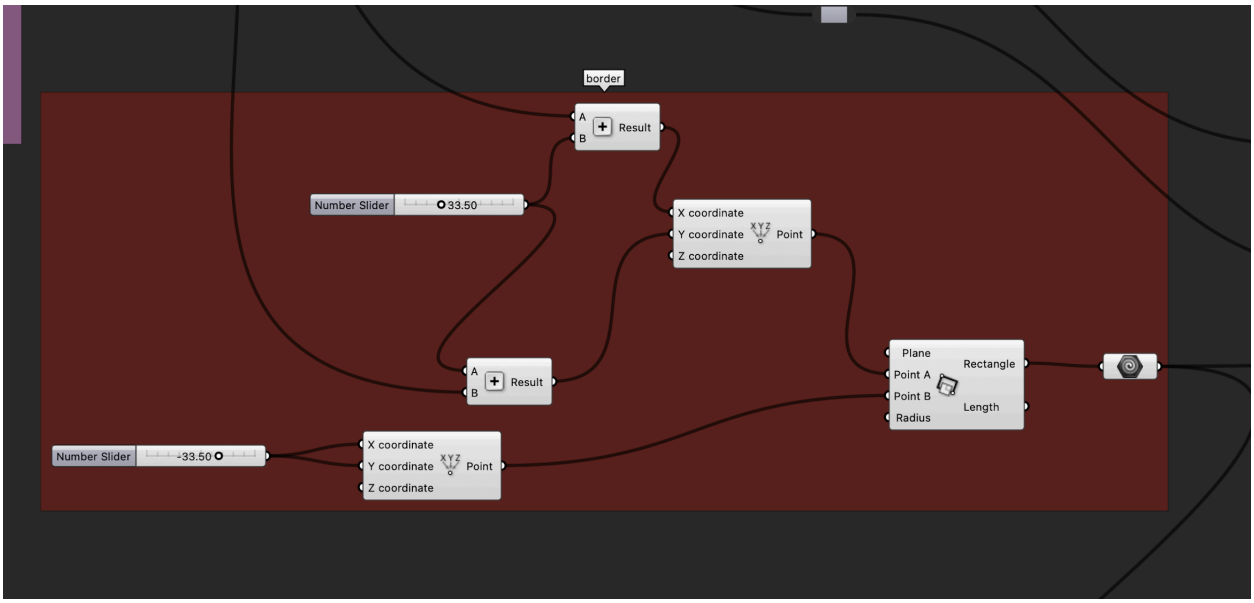


This section is where decide the placement of the interlocking components of the puzzle pieces. The Voronoi curves are passed in where we find the center point of

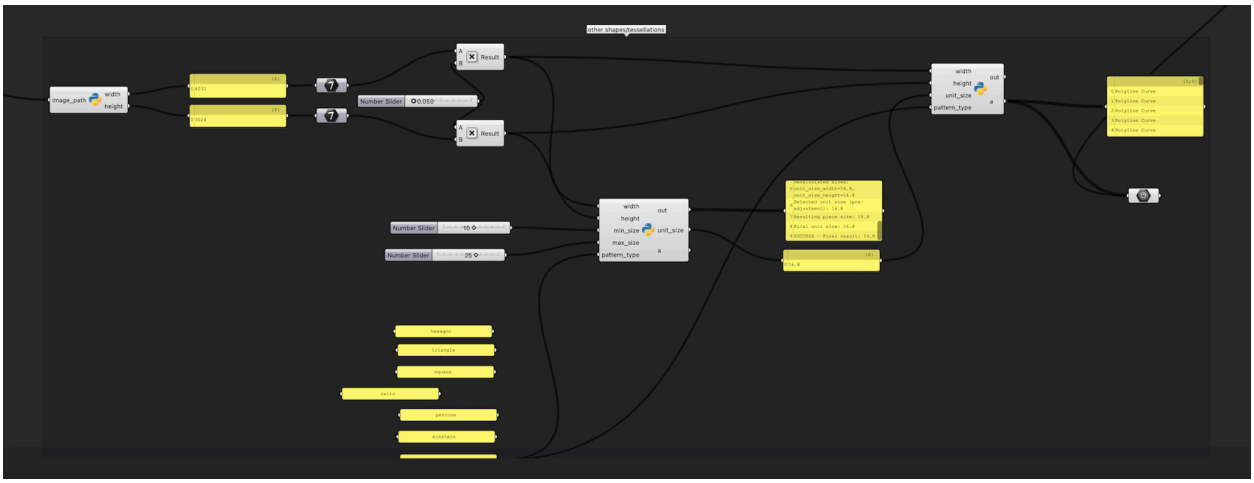
either the rectangle or one of the line segments. We only want the center points of the line segments and we only want the line segments that are long enough to place an interlocking joint on.



Here is where we create the circles that we use to interlock the pieces. We create a circle on the middle of the line segment. Using the two intersection points of the line and the circle, we have the start of one circle, the middle intersection, and the end of the second circle. These two circles are what we use for our interlocking joints. See image below.

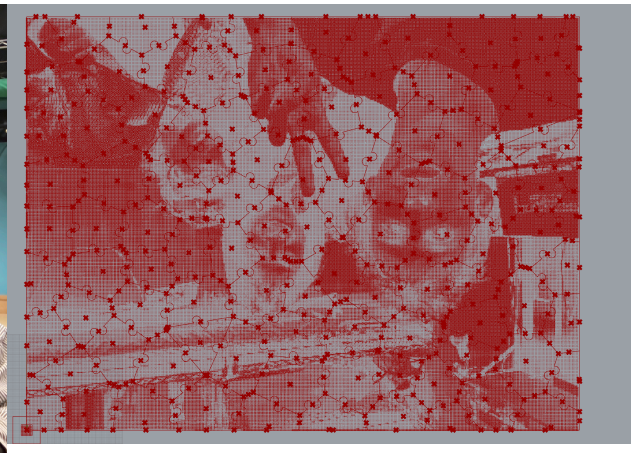


Here is the puzzle border generation. A rectangle is constructed from two points defined with respect to the image size, parameterized by their start position, allowing for even sizing as the image, then the user, each adjust its size.



Using the image path, we are able to extract the height and width of the user uploaded image. We multiply each dimension by 0.05 to scale the image a size that a size that is able to be fabricated. The first python script calculates a piece size that is reasonable for the given image dimensions. It takes the image dimensions, a user specified minimum and maximum piece size, and a user specified pattern shape. The second python script creates the puzzle pattern based off of the image dimensions, the puzzle piece size, and the pattern shape.

Original image



Generated output

Two YouTube videos we referenced to obtain our output:

https://www.youtube.com/watch?v=PGbYo4FTkOQ	https://www.youtube.com/watch?v=wSAeDY_9eQ0

Python File 1 - Calculating the size of each piece:

Our Grasshopper file uses two Python components. The first is for calculating the size of each piece and the second is for generating the puzzle pattern.

Unit Size Gist

```
def calculate_unit_size(_width, _height, min_size, max_size, pattern_type):
```

This function takes in five arguments:

- _width and _height are the dimensions of the photo that the user input
- min_size and max_size are the smallest and largest sizes that the user can choose from
- pattern_type is the type of tiling pattern that we want for our puzzle output. You can choose from hexagon, square, triangle, arrow, and voronoi.

The goal here is to find a unit size that will tile the area nicely without making shapes that are too big or too small.

```
# Convert to float vals
width = float(_width)
height = float(_height)
min_size = float(min_size)
max_size = float(max_size)
```

Ensuring all of the number values that we input are float. Had some weird bugs early on with inputting integers.

```
if pattern_type == "hexagon":
        size_factor = 2.0
    elif ...
```

size_factor is a way of translating unit_size into “how big does this piece really feel.”

```
target_piece_size = (min_size + max_size) / 2
initial_unit_size = target_piece_size / size_factor
```

This is just gives us a middle ground to work off of.

```
if pattern_type == "hexagon":
        width_units = width / (initial_unit_size * 1.5)
        height_units = height / (initial_unit_size * math.sqrt(3))
elif ...
```

Now we use some geometry knowledge to figure out how many shapes can we fit along each axis. For hexagons, the 1.5x and sqrt(3) come from the way hex tiles overlap in a honeycomb structure. Triangles and squares are simpler.

```
unit_size = min(unit_size_width, unit_size_height)
```

Choosing a safer piece size

```
piece_size = unit_size * size_factor
if piece_size < min_size:
    ...
```

```
elif piece_size > max_size:
    ...
```

This is the final check before returning the unit_size.

```
return unit_size
```

Returning the final value.

Python File 2 - Generating the puzzle pattern:

Puzzle Generation Gist

```
def create_tessellation(width, height, unit_size, pattern_type="hexagon"):
```

```
created_curves = []

width = float(width)
height = float(height)
unit_size = float(unit_size)
```

```
if pattern_type == "hexagon":
    created_curves = create_hexagon_tessellation(width, height, unit_size)
elif pattern_type == "triangle":
    created_curves = create_triangle_tessellation(width, height, unit_size)
elif pattern_type == "square":
    created_curves = create_square_tessellation(width, height, unit_size)
elif pattern_type == "arrow":
    created_curves = create_arrow_tessellation(width, height, unit_size)
else:
    print("Invalid pattern type. Choose from: hexagon, triangle, square, or arro
w")
```

```
return created_curves
```

Square Tessellation

```
def create_square_tessellation(width, height, square_size):
```

```
    cols = int(width / square_size) + 1
    rows = int(height / square_size) + 1
```

```
    for row in range(rows):
        for col in range(cols):
```

```
x = col * square_size
            y = row * square_size

            if x > width or y > height:
                continue

            pts = []
            pts.append(rg.Point3d(x, y, 0))
            pts.append(rg.Point3d(x + square_size, y, 0))
            pts.append(rg.Point3d(x + square_size, y + square_size, 0))
            pts.append(rg.Point3d(x, y + square_size, 0))
            pts.append(pts[0])  # Close the square

            polyline = rg.Polyline(pts)
            created_curves.append(polyline.ToNurbsCurve())
```

Triangle Tessellation

```
def create_triangle_tessellation(width, height, tri_size):
```

```
created_curves = []
tri_height = tri_size * math.sqrt(3) / 2

cols = int(width / tri_size) + 2
rows = int(height / tri_height) + 2
```

```
for row in range(rows):
    for col in range(cols):
```

```
x = col * tri_size / 2 - (tri_size / 2)
y = row * tri_height
```

```
if (col + row) % 2 == 0:
    pts.append(rg.Point3d(x, y, 0))
    pts.append(rg.Point3d(x + tri_size, y, 0))
    pts.append(rg.Point3d(x + tri_size / 2, y + tri_height, 0))
else:
    pts.append(rg.Point3d(x, y + tri_height, 0))
    pts.append(rg.Point3d(x + tri_size, y + tri_height, 0))
    pts.append(rg.Point3d(x + tri_size / 2, y, 0)
```

```
pts.append(pts[0])  # Close the triangle
polyline = rg.Polyline(pts)
created_curves.append(polyline.ToNurbsCurve())
```

Hexagon Tessellation (Honeycomb)

```
def create_hexagon_tessellation(width, height, hex_size):
```

```
created_curves = []
hex_width = hex_size * 2
```

```
hex_height = math.sqrt(3) * hex_size

cols = int(math.ceil(width / (1.5 * hex_size))) + 1
rows = int(math.ceil(height / hex_height)) + 1
```

```
for col in range(cols):
    for row in range(rows):
```

```
            x = col * (1.5 * hex_size)
        y = row * hex_height

        if col % 2 == 1:
            y += hex_height / 2

        hex_pts = []
        for i in range(6):
            angle = math.pi / 3 * i
            px = x + hex_size * math.cos(angle)
            py = y + hex_size * math.sin(angle)
            hex_pts.append(rg.Point3d(px, py, 0))

        hex_pts.append(hex_pts[0])  # Close the hexagon

        polyline = rg.Polyline(hex_pts)
        created_curves.append(polyline.ToNurbsCurve())
```

Arrow Tessellation

```
def create_arrow_tessellation(width, height, arrow_size):
```

```
created_curves = []
arrow_width = arrow_size
arrow_height = arrow_size
```

```
row_spacing = arrow_height
cols = int(width / (arrow_width * 2)) + 2
y_offset = 0
row_count = 0
```

```
while y_offset + arrow_height <= height + 1:
    base_point = rg.Point3d(0, y_offset, 0)
```

```
if row_count % 2 == 0:
    row_arrows = create_right_arrow_row(...)
else:
    row_arrows = create_left_arrow_row(...)
```

```
created_curves.extend(row_arrows)
y_offset += row_spacing
row_count += 1
```

```
return created_curves
```

Right Arrow

```
def create_right_arrow(arrow_width, arrow_height):
```

```
pts = [
    rg.Point3d(0, 0, 0),
    rg.Point3d(0, arrow_height, 0),
    rg.Point3d(arrow_width, arrow_height, 0),
    rg.Point3d(arrow_width, arrow_height * 1.5, 0),
    rg.Point3d(arrow_width * 2, arrow_height / 2, 0),
    rg.Point3d(arrow_width, -arrow_height / 2, 0),
    rg.Point3d(arrow_width, 0, 0),
```

```
    rg.Point3d(0, 0, 0)  # Closing the polyline
]
```

Left Arrow

```
def create_left_arrow(arrow_width, arrow_height):
```

```
pts = [
    rg.Point3d(0, 0, 0),
    rg.Point3d(0, arrow_height, 0),
    rg.Point3d(-arrow_width, arrow_height, 0),
    rg.Point3d(-arrow_width, arrow_height * 1.5, 0),
    rg.Point3d(-arrow_width * 2, arrow_height / 2, 0),
    rg.Point3d(-arrow_width, -arrow_height / 2, 0),
    rg.Point3d(-arrow_width, 0, 0),
    rg.Point3d(0, 0, 0)
]
```

Right Arrow Row

```
def create_right_arrow_row(arrow_width, arrow_height, num_arrows, base_poi
nt):
```

```
arrows = []
for i in range(num_arrows):
    arrow = create_right_arrow(arrow_width, arrow_height)
    move_vec = rg.Vector3d(base_point.X + i * arrow_width * 2, base_point.Y,
0)
    rs.MoveObject(arrow, move_vec)
    arrows.append(arrow)
```

Left Arrow Row

```
arrows = []
for i in range(num_arrows):
    arrow = create_left_arrow(arrow_width, arrow_height)
    move_vec = rg.Vector3d(base_point.X + arrow_width + i * arrow_width * 2,
base_point.Y, 0)
    rs.MoveObject(arrow, move_vec)
    arrows.append(arrow)
```

Code Gists and Grasshopper File

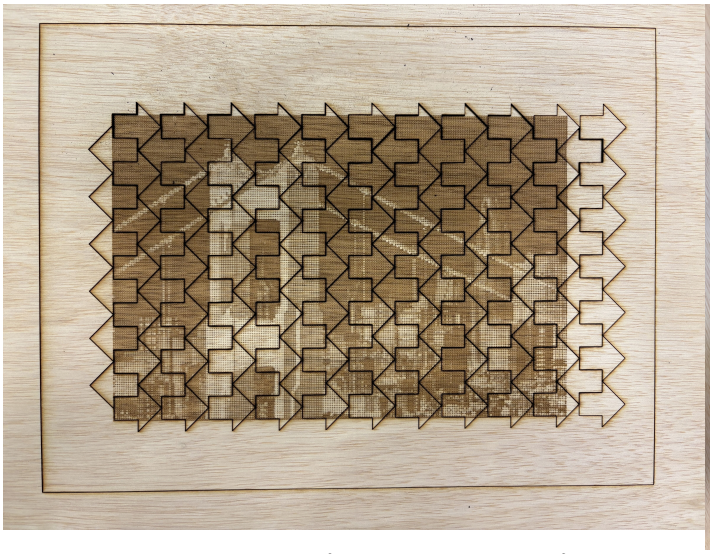
Final Grasshopper File

Unit Size Gist

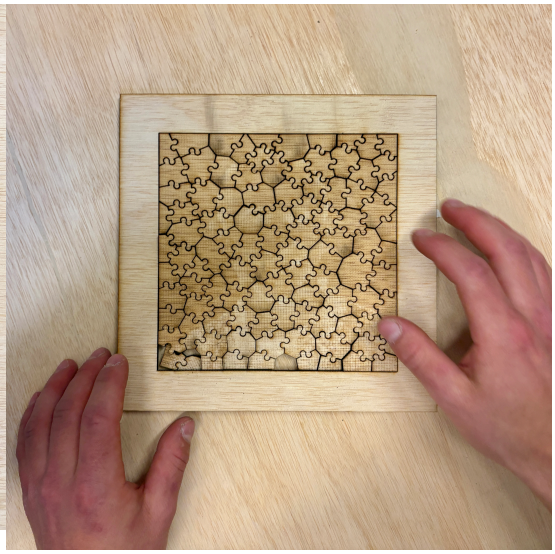
Puzzle Generation Gist

Process Documentation/Final Output

(videos + more photos in drive linked above)



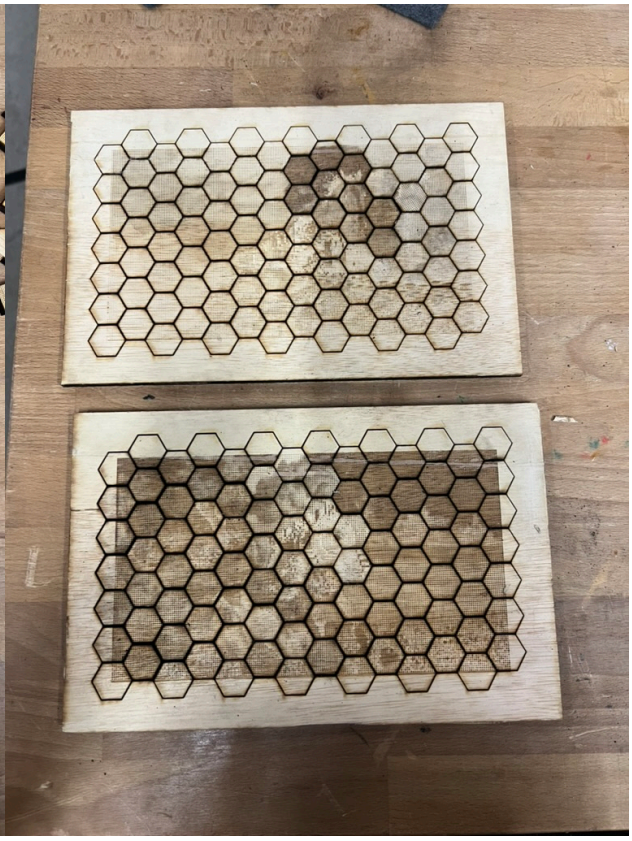
golden_sate_skyline arrow tessellation



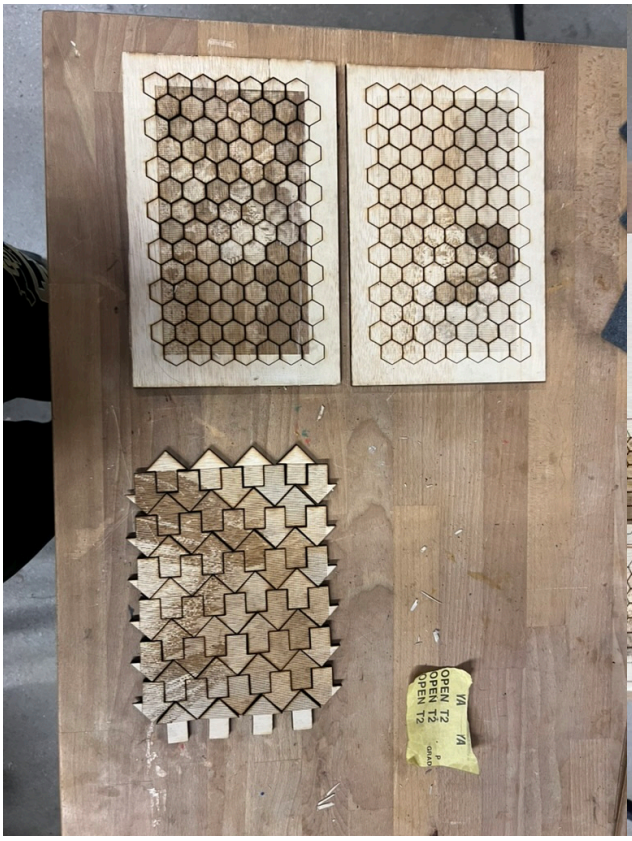
michael_rivera_face Voronoi pattern generation.

jigsaw_swap (same cut pattern)

horse_and_train images generated with hexagonal tessellation pattern.



horse_and_train face swap.



overall_shot of horse, train, and cathedral

broken_border :(

https://vimeo.com/1080019662?share=copy

video_of_laser cutting the cathedral image.

Reflection:

If we take a step back and look at the whole puzzle, it consists of a 2D-dot-plot image and pieces cut into our desired tessellation/jigsaw generated from our Grasshopper code. So, all in all, I would say our project pieced together how we had imagined and intended it would. Of course, however, there are a few minor details which we either missed on, did not have time to get to, or simply spent too much time on, ignoring the former two in the process.

As far as the code was concerned, the biggest time waster came in the form of grid generation. While it appeared simple to begin with, it caused many hours to be wasted attempting to generate square pieces within Grasshopper itself. Thankfully, after a quick conversation with Professor Rivera, and the inevitable revelation our initial plan was correct, we switched over a python script and alleviated ourselves of a nasty headache. As far as opportunities we missed, we feel this partially overlaps with what we simply did not leave time to accomplish. This came in the shape of Einstein Tile aperiodic tessellations, as well as the opportunity to provide an actual, cleaned up, user interface for our code. Though, the latter was seemingly a project in and of itself. Aside from these small minor setbacks and misses, the coding went smoothly.

Regarding the fabrication, our files generated, uploaded, and presented the desired cuts, just as we had hoped and expected. However, our material choice failed us quite a bit in the end. Especially for an intricate design such as our puzzles, laser cutting takes quite some time, and therefore quite obnoxious to hog during finals season. Moreover, we were hesitant to over-burn the wood at first, realizing after a few puzzles we needed to cut less pieces for smaller puzzles, and darken the rastered dots quite drastically. And so, while we may have missed on our material selection, we were not disappointed given our code was functioning well.

If we had the opportunity to redo any part of this project, it would most likely boil down to three potential improvements. First, going back and generating a functional Voronoi jigsaw, as well as one or two basic tessellations, to allow ourselves time to focus on one, much more complex tessellation, specifically, the Einstein Tiles. Further, the second key point of improvement would be making sure to fabricate on better material. While store-bought 1/4″ plywood is cheap, coming in large sheets, it often warps and has layers of veneer and glue, all of which cause the laser cutters to struggle, even with laser settings adjusted in an effort to assist it. To that point, the final improvement would be really slowing the laser down, making sure the puzzles were extremely visible and not hidden thereafter by the burns from the piece cuts.

To sum, we are both extremely pleased with the results—aside from the struggle with our material cutting fully. As a team, we planned well and functioned efficiently as a team, keeping the workload balanced, the timeline for deliverables

manageable, and the schedule planned in advance. It was a wonderful challenge, and an exciting way for us to wrap up the class!