Crowd & Light Simulation for SNF 2018

50.017 Graphics and Visualisation Project Report

Koh Kai Wei, Kwok Shun Git and Chan Wei Ren

Abstract

We present a tool for helping installation artists with the creation of interactive and appealing night light installations. We targeted the problem of visualising animated lighting sequences and placing sensors on the installation. Our tool allows the artist to load in their preliminary physical designs and simulate crowd movements around the installation. Following that, they can see the installation from different viewpoints and also load in the animated lighting sequences. As large-scale light installations have a long construction time, the tool will enable many iterations of testing before the physical structures are even built, which reduces wastage of resources.

We chose `three.js` as it is cross-platform, which reduces adoption friction. Also, many tools and plug-in libraries are also available for it.

1. Crowd Simulation

Problem Scoping

To aid the artist with visualising their installation in the context of their site, we simulate crowds navigating around the installation. The simulation serves multiple purposes:

- Highlight potential choke points around the structure,
- Visualise the lighting sequence reacting to the crowd,
- Identify suitable positions to place sensors or cameras on the structure

To simulate humans walking around the structure and being affected by it, we looked up research papers on crowd simulation and also behavioural models of pedestrian behaviour. Most papers describe agent models, which concern agents getting to their destinations while meeting certain constraints, like avoiding collisions with others. However, many were rather complicated to implement and none specifically mention looking at artwork. Thus we have attempted to solve it in a novel manner.

As the site given was linear, we assumed that our crowd agents want to cross over to the other side in general (1). We first generate a navigation mesh (2), and process it to become a graph used for path search (3). Random secondary target locations are chosen for the crowd agents, from which paths to and from the target were generated (4). The agents then spawn (5) and move along the path, being affected by other pedestrians and the structure using a force-based model (6). The agents will randomly decide to stop along their path to observe the structure, and will randomly start moving again. After they reach their destination they disappear, and wait to spawn again.

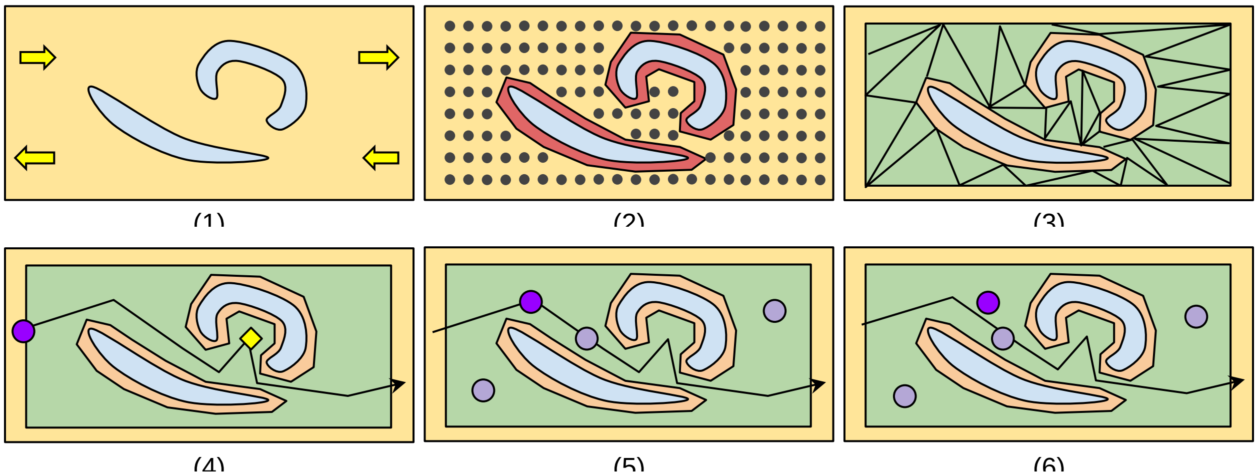


Figure 1.1: Crowd agent path finding process

Navigation Meshes

After loading in their physical design, a navigation mesh is generated based on the structure. Navigation meshes are used for pathfinding around the structure, as shortest path or least effort algorithms rarely generate paths that approach closed convex spaces of the installation.

The navigation mesh indicates where the traversable area around the structure are. To define the traversable area, first we define the *collision volume*, which is the set of all points along the installation that are below the average human height (1.6 metres). The traversable area is then defined to be the overhead projection of the collision volume, offset outwards by a certain radius (Minkowski sum). This definition allows crowd agents to walk under parts of the installation that are above human height.

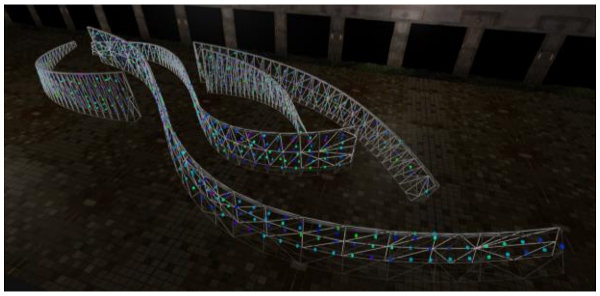
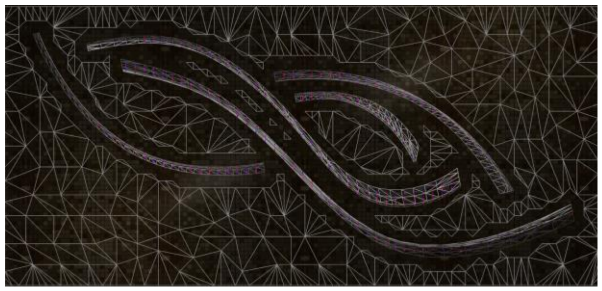
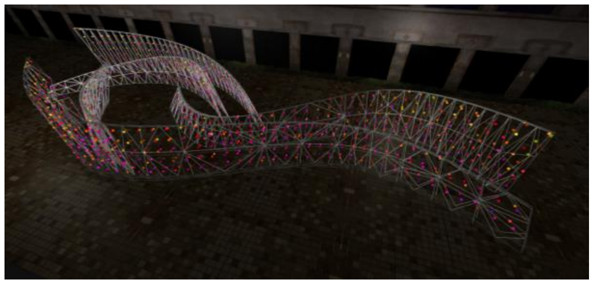
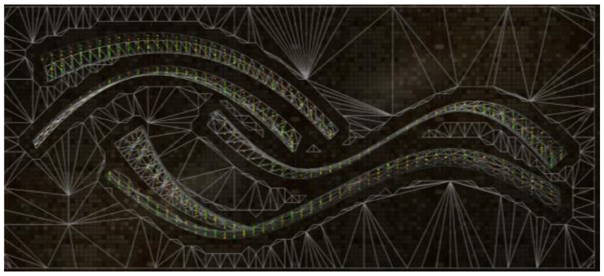
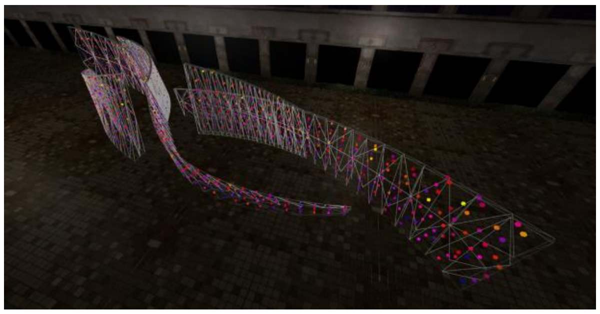
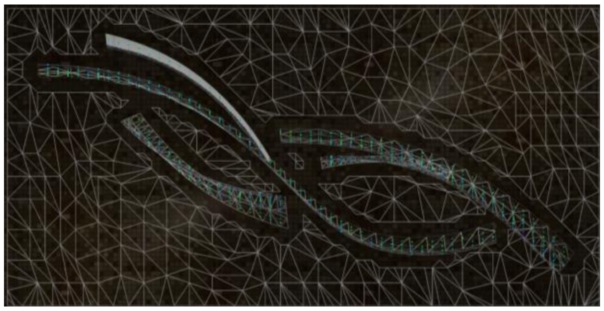
The traversable area is built out of a square grid with certain points removed. A graph representation of the mesh is then built, indicating which nodes (faces in the mesh) are adjacent. To actually generate a path from point A to point B, the closest nodes in the graph to the points are found. An A* search is then performed on the graph from the source node to the destination node.

The kd-tree and mesh simplification are already implemented in `three.js`. The navigation mesh is generated by three-pathfinding, which also implements the A* search and path polyline generation ("funnel algorithm"). We implemented the algorithm to generate the raw mesh fed into three-pathfinding.

Navigation Mesh Algorithm

```
1. Generate Kd-tree of the structure
2. Initialize 2D square grid
3. for p(x, y) in grid do
     for z in heights
       get the distance to the structure from p(x,y,z)
       if distance < radius then
         remove p(x, y) from grid
4. for each square (a,b,c,d) in grid
     add face to mesh if more than 2 points in square remain
5. Simplify mesh using Delaunay Triangulation and use three-pathfinding to build graph
```

The algorithm performed well on the different iterations of models loaded in.

Model	Navigation Mesh
	
	
	

2. Light Placement

Problem Scoping

In order to simulate light on our structure for use of visualization, we must find a way to place points of lights along the surface of the structure at a fixed interval. The issue is that the mesh that is provided to us by the architects has around 95000 vertices (and thus around 33000 faces). Also, all the vertices are not evenly spaced along the entire stretch of the structure. The structure itself is a rather complex curve that was modeled in Rhino and not by a mathematical equation that would allow us to calculate fixed points along the surface. This problem can be observed in figure 1 below.

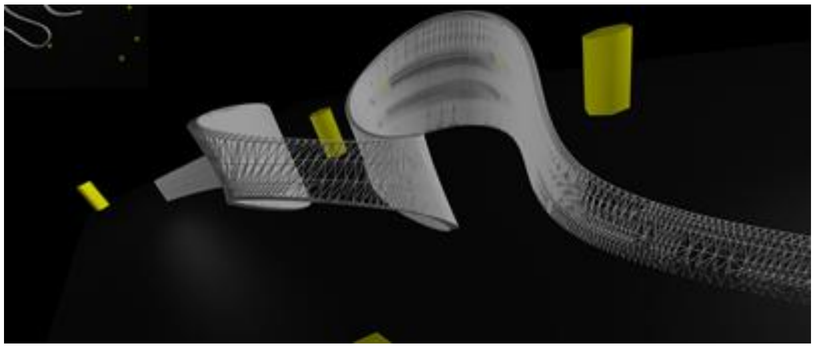


Figure 2.1: The Architectural Mesh Structure

What we had to simulate in the structure was a set of light points that are uniformly placed in a grid like format, as the LEDs would be placed in our lattice structure as shown in figure 2 below.

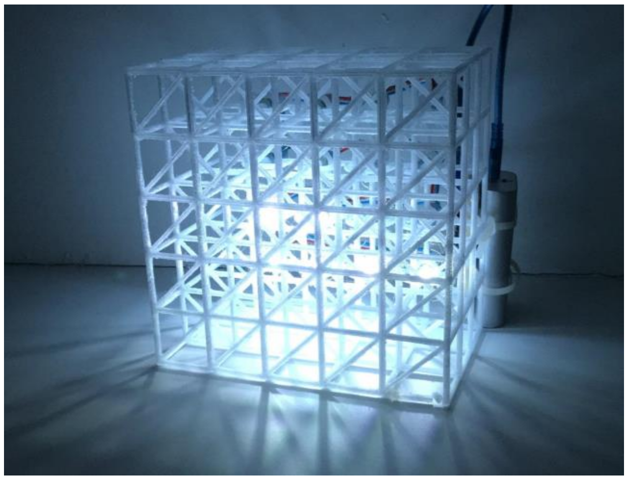


Figure 2.2 : A small segment of our light installation with grids of LED behind

Approach 1: Stratified Sampling

Stratified sampling works by breaking down the faces in the mesh into smaller sub faces using the Octree algorithm, then it uses an exponential distribution to sample faces that are as close to the center of each octree node. We implemented this in our visualization application and it resulted in some success in generating points along our structure. How uniformly spaced the points will be is determined by adjusting parameters of the exponential distribution used.

Stratified Sampling Algorithm

1. Voxelize the model using an Octree algorithm
2. Pick a point on the surface of the object according to a probability distribution (In the paper, an exponential distribution ($\lambda e^{-\lambda d}$) was chosen whereby d is the distance relative to the edge length of a voxel and the center of the originating voxel.)
    a. Each triangle in a voxel is subdivided until the probability density function can be considered to be constant throughout its area.
    b. Sub-triangles that cross boundaries can be subdivided further until no edge is bigger than a fixed length, relative to a voxel edge length.
    c. The function value at the centroid of a terminal sub-triangle is multiplied by its area and is defined as the sub-triangle's priority.
    d. A roulette scheme is used to select a sub-triangle according to these priorities.
    e. Once a terminal sub-triangle is chosen, uniform sampling is used to produce a sample from it.
3. If samples generated are too close to each other
    a. Enforce a minimum distance between samples relative to the size of the voxel by eliminating samples one by one until the minimum distance constraint is satisfied.

The results generated can be varied by changing the variable λ as noticed below.

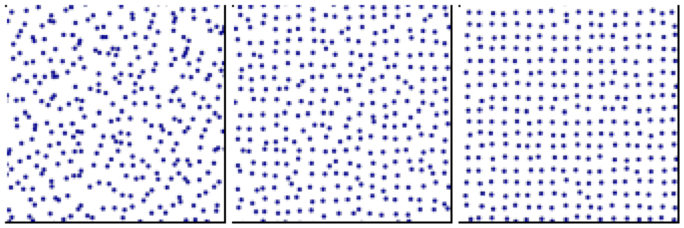


Figure 2.3: Varying λ. From left to right, values 0, 10, and 30

However when performing this algorithm, the load time of the Octree (as seen in Figure 2.4) was extremely long on javascript (given that it had to group 95000 vertices) and when we tried to simplify the sampling by using an exponential distribution and the center of each triangle face to select a point inside each Octree subdivision we ended up with a highly clustered point cloud (Figure 2.5).

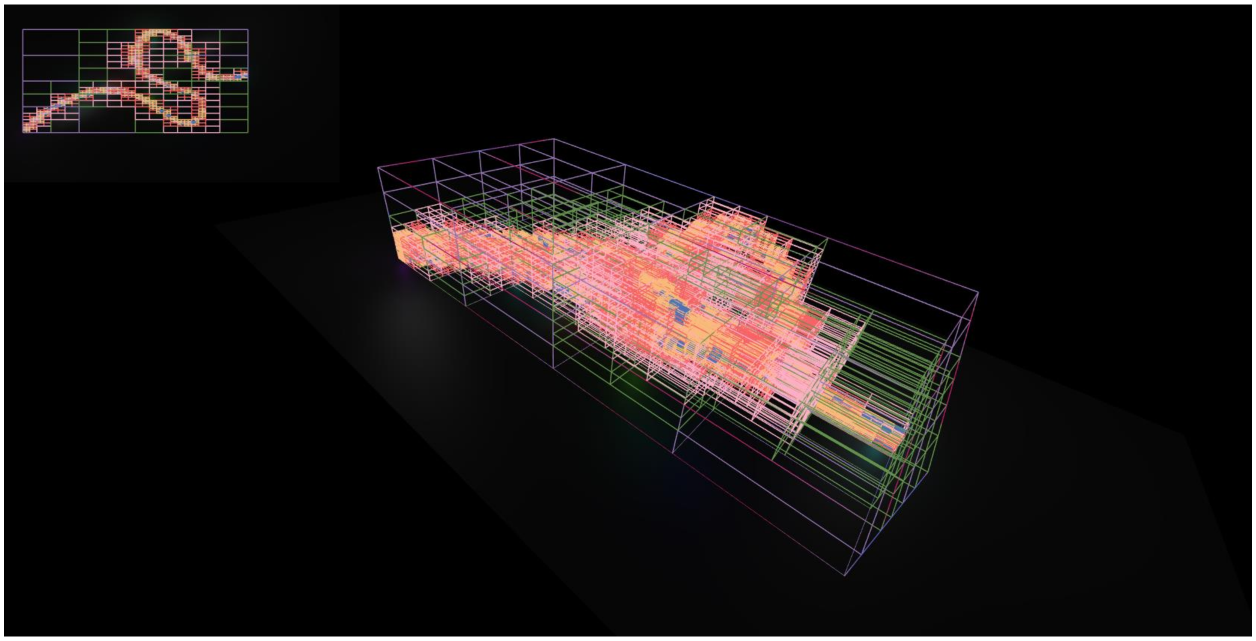


Figure 2.4 :Our generated Octree represented by bounding boxes

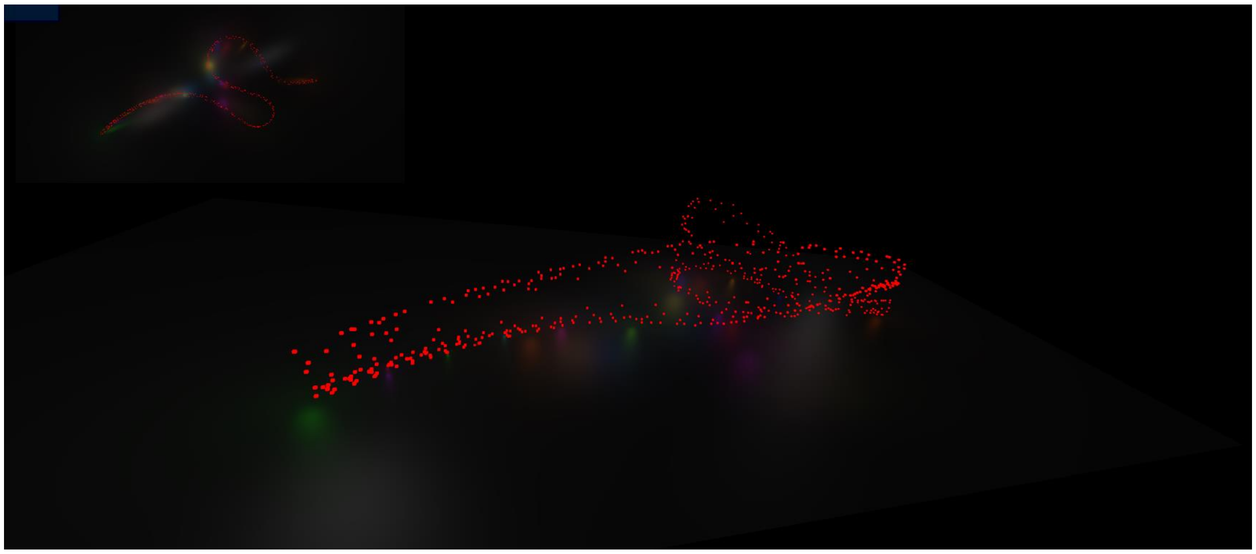
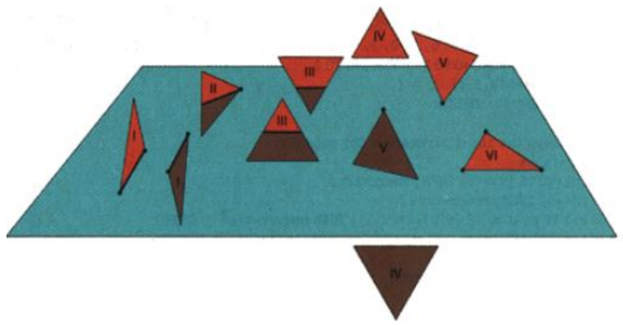


Figure 2.5 : Sampled point cloud from our algorithm implementation

Approach 2 : CONREC Contouring Algorithm (By Paul Bourke)

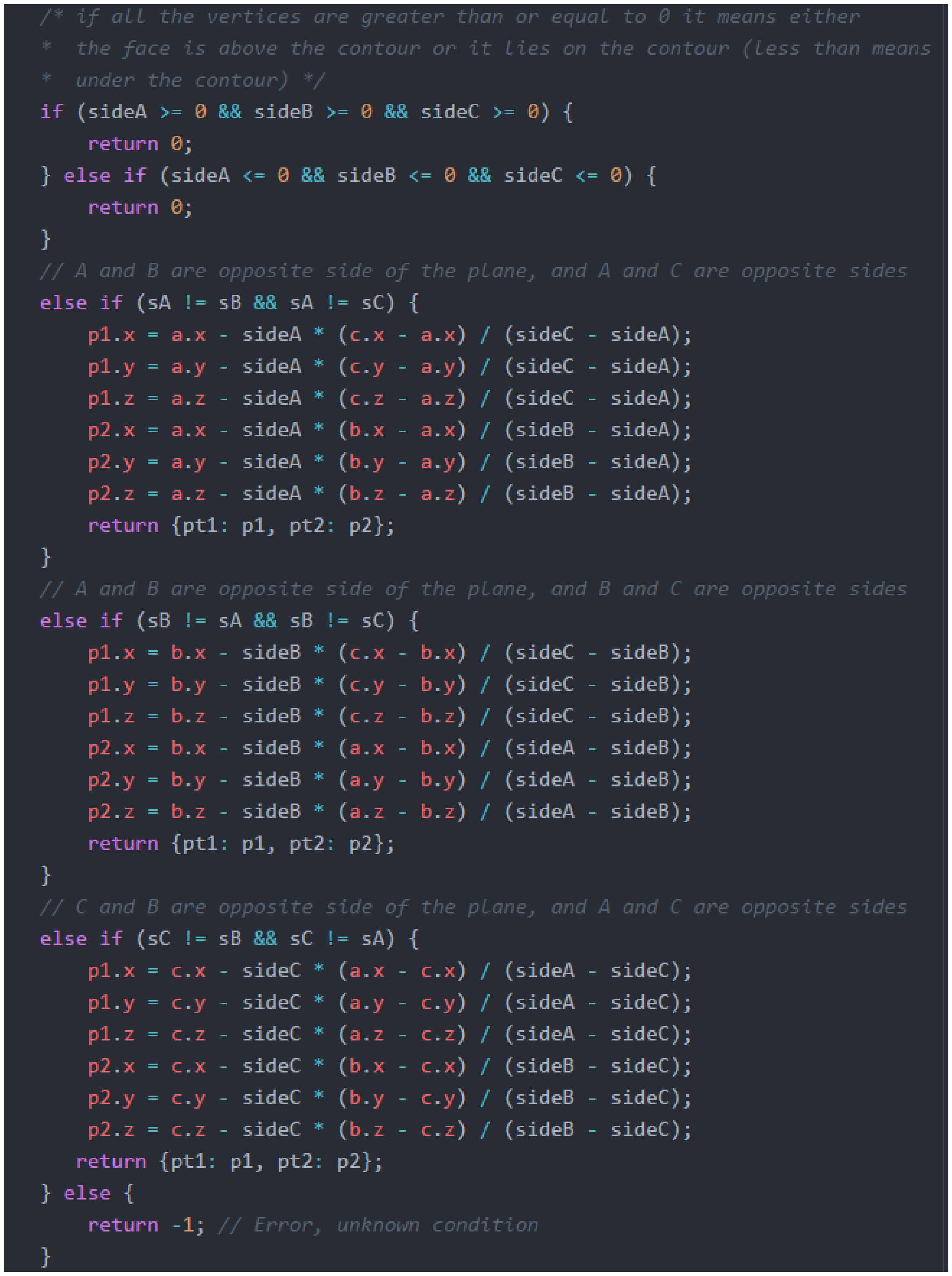
After deciding to try other approaches to solving our problem, we landed on a paper written by Paul Bourke which describes a contouring algorithm that allows us to draw contours of any mesh. The algorithm uses simple plane triangle intersections to generate a contour. We took it one step further by doing line-plane intersections on the contours to create point lights along our structure.

There are 10 possible cases that happen when a plane intersects a triangle, listed below:

#	Vertices in relation to contour line			
1	Below	Below	Below	 I I II II III III IV IV V V VI
2	Below	Below	On	
3	Below	Below	Above	
4	Below	On	On	
5	Below	On	Above	
6	Below	Above	Above	
7	On	On	On	
8	On	On	Above	
9	On	Above	Above	
10	Above	Above	Above	

We only need to concern ourselves with 3, 4, 5, 6 and 8 as those are the only cases where we need to create a line between the edges joining the vertices of the triangle. We implemented the algorithm in Javascript, shown in Figure 2.6.

And for our use case, the algorithm performs fast enough to place points along any complex mesh. It also allows us to control the point density of the light points we wish to simulate, thus allowing us to get the number of lights as close to what our actual structure will have as well.



```
/* if all the vertices are greater than or equal to 0 it means either
*  the face is above the contour or it lies on the contour (less than means
*  under the contour) */
if (sideA >= 0 && sideB >= 0 && sideC >= 0) {
    return 0;
} else if (sideA <= 0 && sideB <= 0 && sideC <= 0) {
    return 0;
}
// A and B are opposite side of the plane, and A and C are opposite sides
else if (sA != sB && sA != sC) {
    p1.x = a.x - sideA * (c.x - a.x) / (sideC - sideA);
    p1.y = a.y - sideA * (c.y - a.y) / (sideC - sideA);
    p1.z = a.z - sideA * (c.z - a.z) / (sideC - sideA);
    p2.x = a.x - sideA * (b.x - a.x) / (sideB - sideA);
    p2.y = a.y - sideA * (b.y - a.y) / (sideB - sideA);
    p2.z = a.z - sideA * (b.z - a.z) / (sideB - sideA);
    return {pt1: p1, pt2: p2};
}
// A and B are opposite side of the plane, and B and C are opposite sides
else if (sB != sA && sB != sC) {
    p1.x = b.x - sideB * (c.x - b.x) / (sideC - sideB);
    p1.y = b.y - sideB * (c.y - b.y) / (sideC - sideB);
    p1.z = b.z - sideB * (c.z - b.z) / (sideC - sideB);
    p2.x = b.x - sideB * (a.x - b.x) / (sideA - sideB);
    p2.y = b.y - sideB * (a.y - b.y) / (sideA - sideB);
    p2.z = b.z - sideB * (a.z - b.z) / (sideA - sideB);
    return {pt1: p1, pt2: p2};
}
// C and B are opposite side of the plane, and A and C are opposite sides
else if (sC != sB && sC != sA) {
    p1.x = c.x - sideC * (a.x - c.x) / (sideA - sideC);
    p1.y = c.y - sideC * (a.y - c.y) / (sideA - sideC);
    p1.z = c.z - sideC * (a.z - c.z) / (sideA - sideC);
    p2.x = c.x - sideC * (b.x - c.x) / (sideB - sideC);
    p2.y = c.y - sideC * (b.y - c.y) / (sideB - sideC);
    p2.z = c.z - sideC * (b.z - c.z) / (sideB - sideC);
   return {pt1: p1, pt2: p2};
} else {
    return -1; // Error, unknown condition
}
```

Figure 2.6: Handling Plane-Face intersections

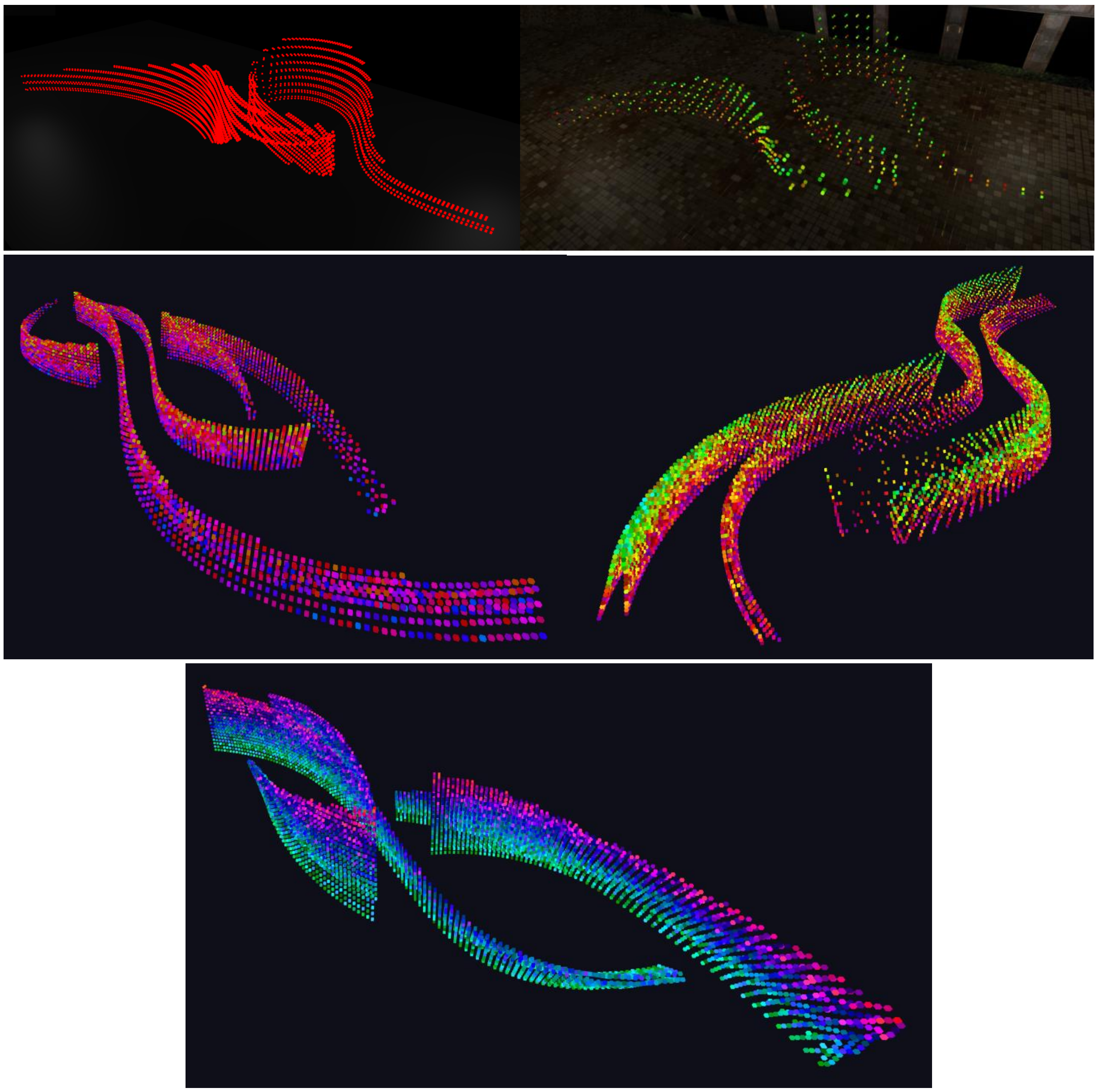


Figure 2.7: Our light placement algorithm working on different models

3. Light Pattern Mapping and Generation

To visualize light patterns along the installation, we needed a way to programmatically change the colours of the lights according to their position. To do this, we mapped the lights along an xy-plane, with each light having its own x and y index. This allows us to program light patterns spatially, which means that we can create “waves” of light across the installation. This was done in line with the vision we had for the exhibit.

For simplicity, the lights were mapped to the xy-plane according to the order in which they were generated. The (integer) y-index indicates the vertical position of the lights for each vertical column of lights, with index 0 being the light closest to the ground. Since each column of lights was generated by slicing the form at regular intervals along an axis, the x-index is defined in that same order; when the light generation algorithm slices the form and adds a column of lights to the installation, it also adds the same column of lights to an incrementing (integer) x-index.

At this point, the architectural side had not given us any cues as to how the lights would flow through the installation. As such, we generated a basic light pattern to simulate the installation in its “passive” state, with seemingly random but smooth and consistent (locally) colour patterns.

To do this, we made use of the perlin noise generator, with the xy-plane of the perlin noise space being spatially mapped to the lights in the manner described above. The z-axis of the perlin noise space is mapped to time, with the z-value increasing as time moves forward. This can be imagined as the xy-plane moving along the z axis over time, and those values on the xy-plane at that specific point of time being mapped to the lights on the installation at the same point of time.

We opted to code the colours of the lights in terms of hue, saturation and luminance (HSL). As the values in the perlin noise space range from 0 to 1, we opted to have these values change the hue of each light, mapping the range of [0, 1] to [0, 360] for hue (which is the full range of hue). Saturation and luminance were set to constant values of 80% each.

4. Camera Controls

Being a simulation that is meant to help us visualise the exhibit from different points of view, including visitors' points of view and what sensors may see, it was natural for us to include a multitude of different views, each with their own customizable camera angles, to help us observe the different behaviours and patterns of the installation or crowd. Each of these views serves a specific purpose, and may have their own intuitive controls to control the camera.

Top View

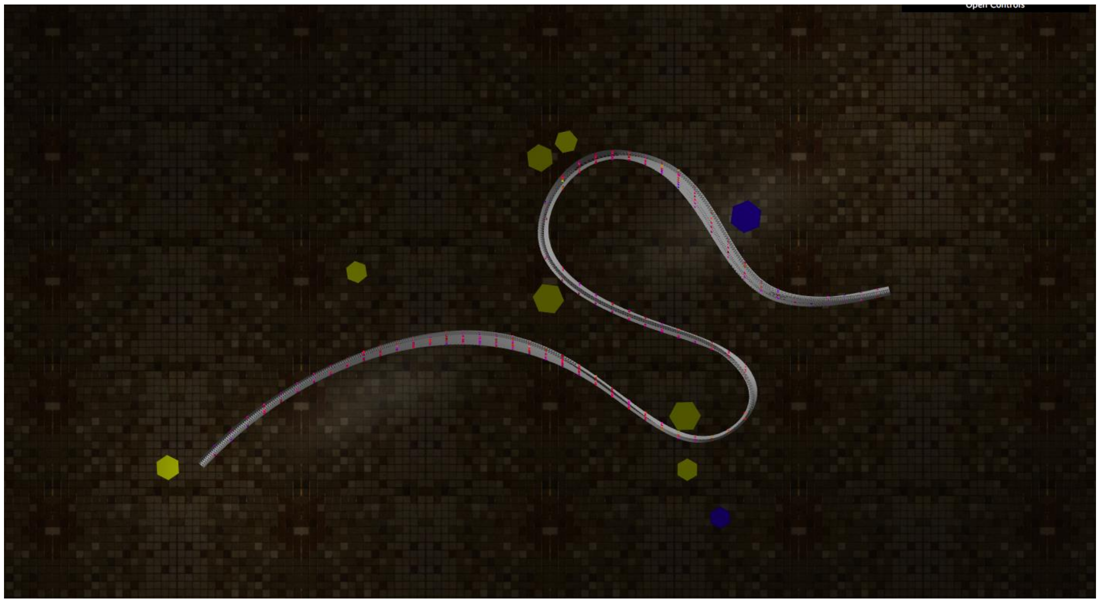


Figure 4.1: Top View

This view provides a top-down view of the installation, providing a convenient overview of the positions and movement of the crowd around the installation. An orthographic camera was chosen to minimize distortion of the structure and people with height.

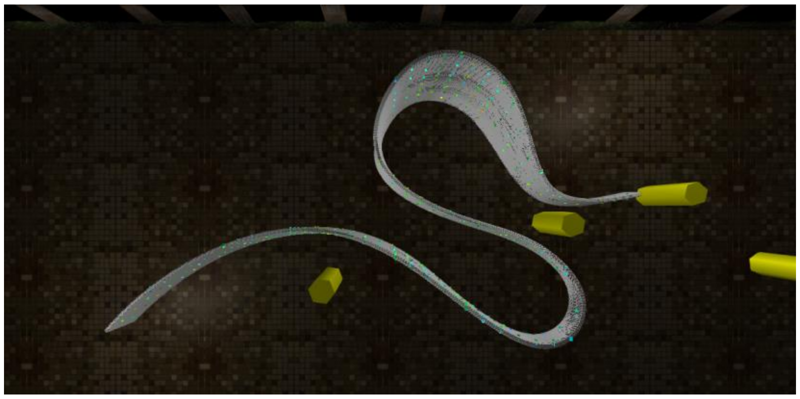


Figure 4.2: Distortions from a perspective camera

Free View

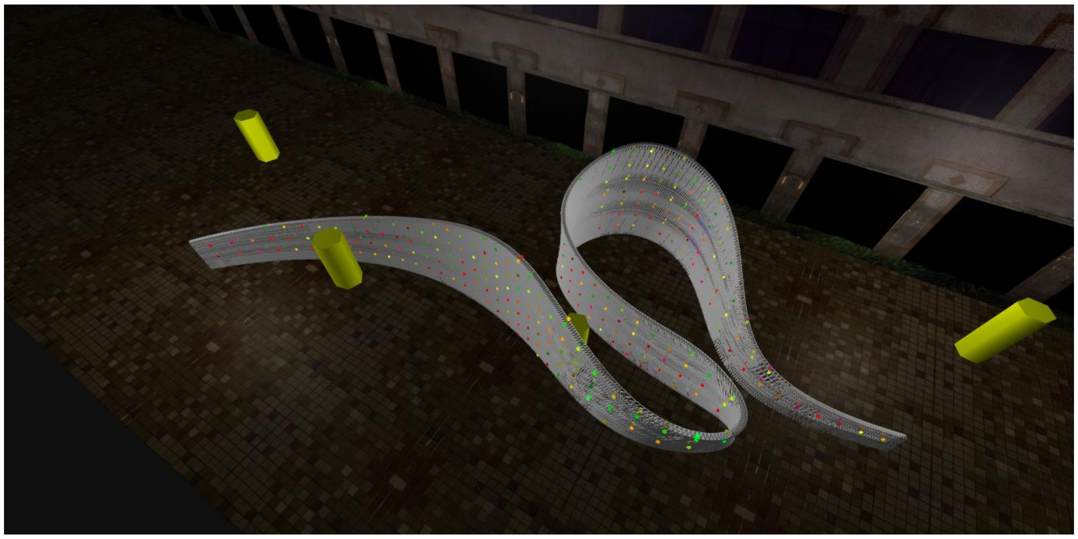


Figure 4.3: Free View

This view provides a free camera that the user can control, allowing them to view the installation from any angle. Clicking and dragging using the left mouse button revolves the camera around the installation, while clicking and dragging using the right mouse button translates the camera around the space. Scrolling zooms in and out, and these 3 controls give the user the ability to place the camera wherever desired.

Human View

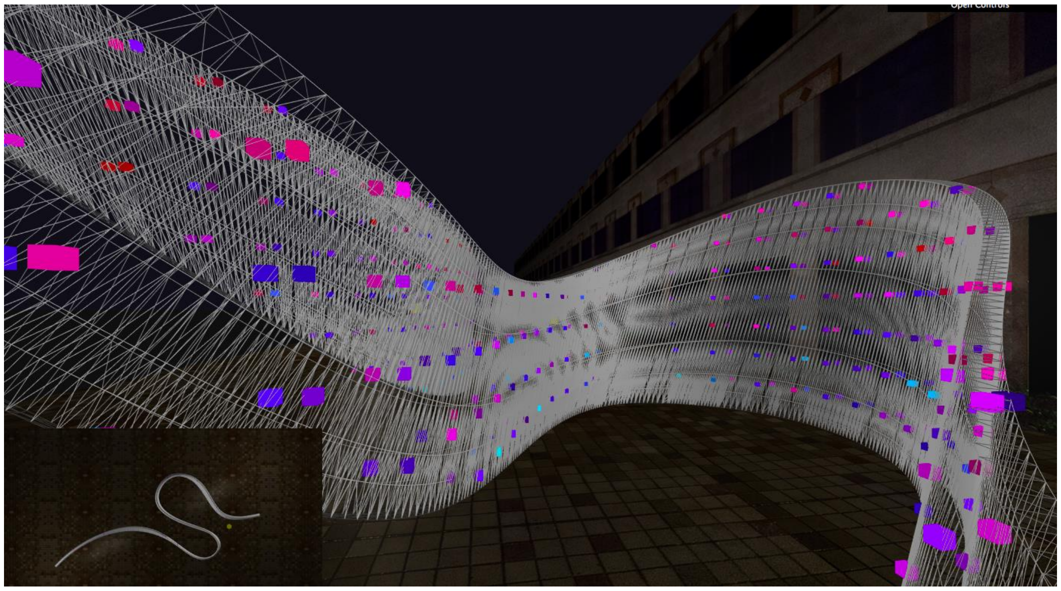


Figure 4.4: Human View

Clicking on a human from any view places the camera where their eyes would be, and facing in the same direction as them. This view allows us to visualize what the installation looks like from a visitor’s perspective as they walk around and look at it. This view also includes a smaller version of the top-down view at the bottom left as a “mini-map”, which helps the user figure out where they are.

Form View

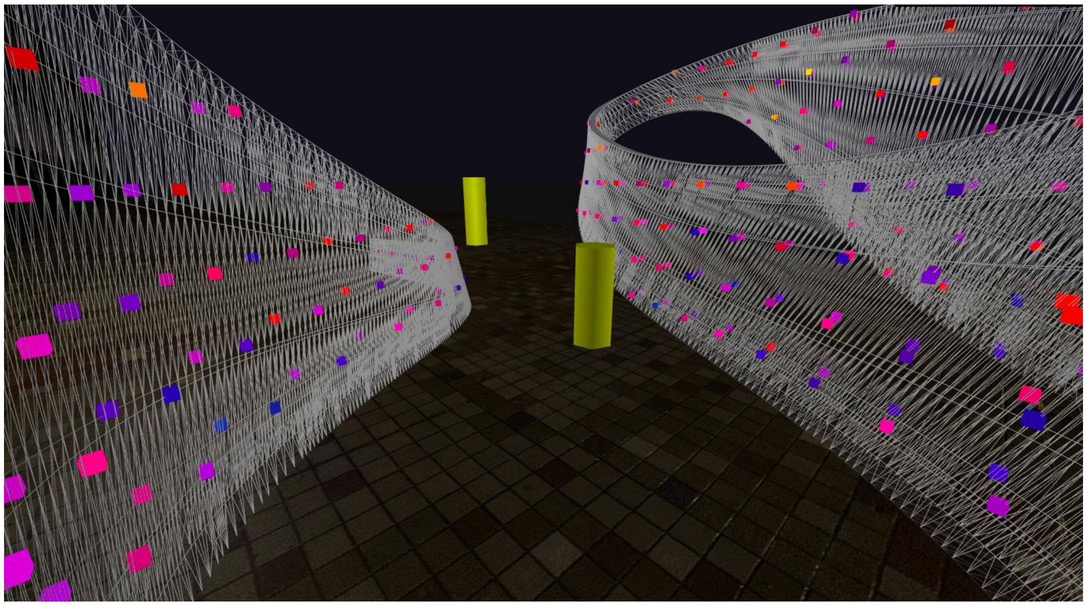


Figure 4.5: Form View

Clicking on any part of the installation places a camera on that point, allowing the user to view what a camera (possibly used for computer vision to determine the number of people in the area) placed at the point would see. The user is also able to click and drag on the left mouse button to rotate the camera around that point, allowing them to determine the best camera placement and angle for such a sensor.

Default View

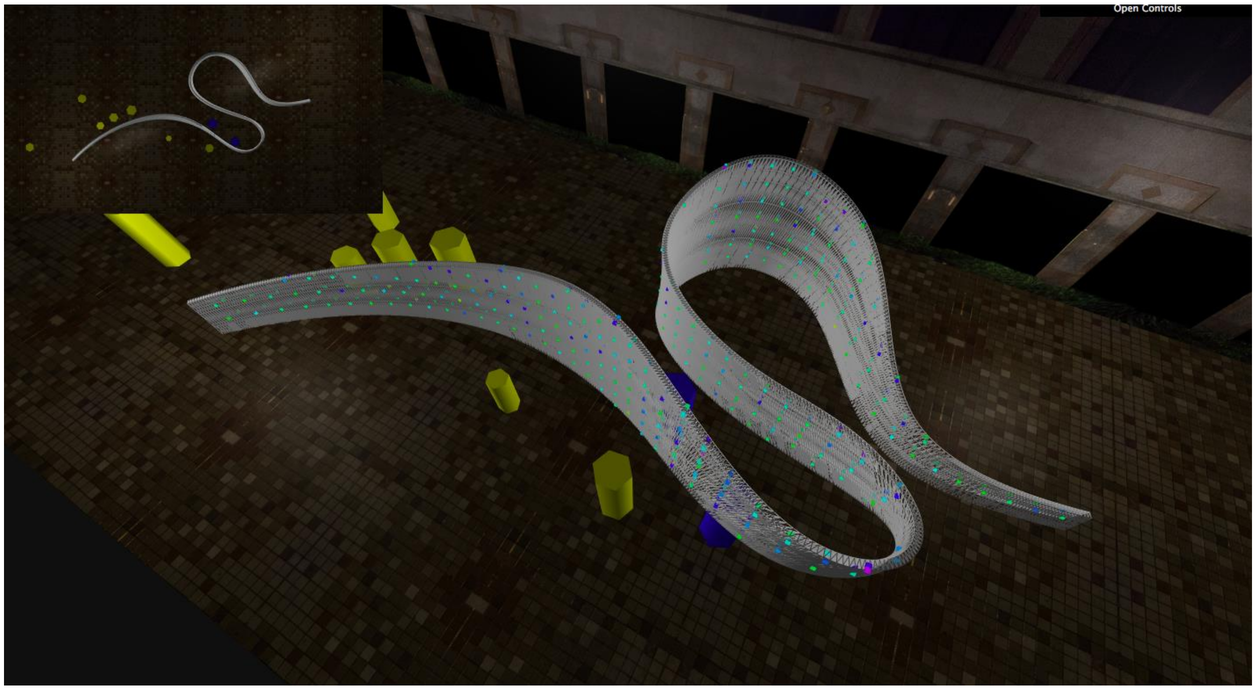


Figure 4.6: Default View

Right clicking from any view resets to this view, where there is a smaller top-down view on the upper left portion of the screen, and a free view with a default camera angle (this default angle is similar to the top-down view) as the main view. This is also the view that the application defaults to when it is loaded.

View Selection

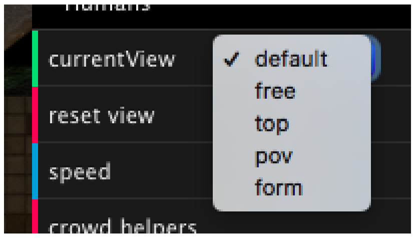


Figure 4.7: View Selection

Users are easily able to select which view they want via the control panel on the top right of the screen. Since right clicking is unavailable on most mobile devices, a “reset view” control was also added to the control panel for mobile users to reset the view to the default view.

Important Links and folders

Our visualization App - https://randna.me/50.017-GnV-Project/
One Page Link - http://randna.me/50.017-GnV-Project/about/
./data - contains some STL files that we used to test with.

References

Bourke, P. (1987, July). *CONREC : A Contouring Subroutine*. From Paul Bourke Website: http://paulbourke.net/papers/conrec/

McCurdy, D. (n.d.). *Donmccurdy - Three Pathfinding*. From Github: https://github.com/donmccurdy/three-pathfinding

Nehab, D., & Philip, S. (2004). *Stratified Point Sampling of 3D Models.* Princeton University, Computer Science Department.