Geometry of Hiding:

Generating Visibility Manifolds

Adrian Koretski, adrian.koretski@mail.mcgill.ca

Supervisor: Clark Verbrugge, clump@cs.mcgill.ca

School of Computer Science McGill University

Montréal, Québec, Canada

December 9, 2020

Abstract

Understanding what a character observes in a game as they move through the space is useful for game analysis and design. In this work we describe a tool that generates *visibility manifolds* given a game terrain and game agent path. Our tool accommodates multiple kinds of game agent motion and considers the impact of visual occlusion in order to efficiently generate a closed 3D mesh representation of the area seen over time. The resulting shape demonstrates unintuitive properties of game agent observations, and an efficient Unity implementation allows the constructed shape to be used in interactive game design.

Contents

1 Introduction

Agent visibility is a well known problem in the fields of video games and artificial intelligence. It is often studied by building models involving direct line of sight in predetermined configurations. However, these models are rarely portable and are not reusable in different configurations. They also provide little information which can be used for analysis of observer visibility, terrain structure or observer path. We are interested in producing accurate geometric data which can be reused (given a terrain and path) in calculating visibility and can be used to analyse paths and terrain. These models would exist in a three dimensional space where two axes represent space and the third represents time.

However, generating perfectly accurate geometry for this sort of case would be computationally expensive and requires curved faces. Instead, we produce a close approximation by generating visibility polygons at multiple steps and connecting them to one another.

We aim to not only represent what the observer can see but also represent how this visibility evolves over time as the observer moves in the environment. We do so by modeling three dominant movement types: forward motion, sideways motion and rotation. These movements are pieced together to form a path which is used in the final mesh generation. In addition to this path, the environment terrain is used to calculate occlusions which determine the shape of the mesh. The meshes are generated by calculating individual visibility polygons at discrete points in time throughout the path of the observer. These polygons are then connected to one another to form the overall mesh.

Connecting the individual polygons is done by assigning attributes and identifiers to the vertices. These are then used to connect the layers to one another.

The end result is a mesh in obj format (vertex data along with triangle data).

This means that the data generated is generic and can be used to perform multiple kinds of analysis and calculations. Moreover, for a given path and terrain, the data only needs to be calculated once, even if other parameters in the analysis change. For instance, if the tool user wants to determine whether a player taking a specified route would be seen by a guard, the user can try many different player routes without needing to recalculate the guards visibility manifold.

The entire system is implemented in the Unity game engine and can be easily added to a preexisting project. Generating a manifold is done by adding the "geometry of hiding" master gameobject and tagging obstacles with the "obstacle" tag. The path is generated via a gameobject which specifies the observer path itself. It needs to be tagged with the "path" tag.

The user has the option of enabling visual representations of the generated geometry (both the polygons and manifolds) and whether or not the generated mesh is saved to disk.

In the following chapters, we will discuss the specifics of how the tool works and will provide an analysis of the output geometry.

More specifically, we will go over what data is required for the construction of the mesh. This includes observer path and environment terrain data as well as the attributes and identifiers associated with individual visibility polygons. We then take the time to break down the algorithm used into two sections. We begin by going over how we generate the polygons and then proceed with connecting them to form the visibility manifold. These sections include properties associated to the geometry of the problems.

We then generate representative manifolds on which we perform our analysis. the shortcomings of our methods are observed by demonstrating parts of geometry which are incorrectly generated, causing certain volumes to be either

erroneously included or excluded from the manifold.

Our analysis takes into account the effects of modifying the properties used to construct the manifold as well as properties intrinsic to the terrain or path used. Multiple combinations of each are analysed, however we use specific test cases in the report for illustration purposes as it is difficult to adequately represent many different three dimensional geometric shapes in two dimensions.

We finally conclude with an analysis of the limitations of the tool as well as how they can be overcome. We also take the time to provide some roadmaps towards future improvements of the tool.

2 Background and Related work

The aim of our tool is to generate visibility manifolds representing the evolution of the observers visibility area. To do so, individual visibility polygons are generated and then connected to one another, forming the final mesh. This is what we will be focusing on. Indeed, the two dimensional visibility problem has been solved at length and thus we will be using a modified angular sweep approach (Asano 1985).

A small motivating example can be seen in figure 1. Here we see an observer moving forward towards a square obstacle which will start to obstruct the observers view. One common way to construct the manifold is to take individual slices and extrude them into slabs stacked onto one another. This can be seen in figure 2. In the first panel, we clearly see how this solution produces crude approximations which can lead to poor intersection calculations and other potential issues.

This can be improved by taking additional smaller steps, as can be seen in the second and third panels of figure 2. There is still potential for problems to

appear, however, we could further reduce the step size to reduce the inaccuracies. The issue with this solution is that the complexity of the shape, along with the data to store is, increases with a decrease in step size. Indeed, in our example, we use step sizes of 1, 0.5 and 0.1 whereas the number of vertices grows from 312 to 624 to 3384 respectively.

The manifold generated from the ideal connections (manually generated), seen in the last panel, uses 246 vertices, fewer than even the coarsest slab manifold we presented. This manifold does not only have the benefit of having fewer parts but will also cause fewer issues due to inaccuracies in its structure. We claim that our tool is capable to producing manifolds such as these. We will go over the details in the following chapters.

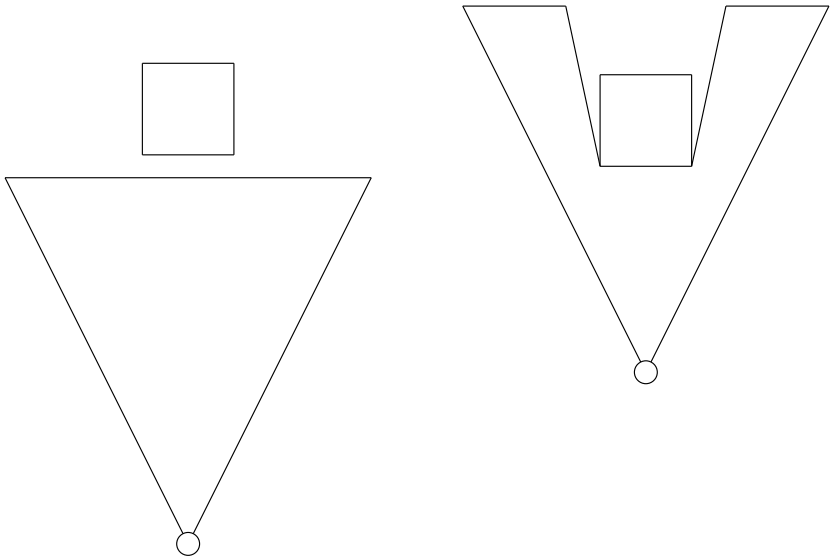

Figure 1: Starting and ending positions. The observer (white dot) moves upward towards a square obstacle.

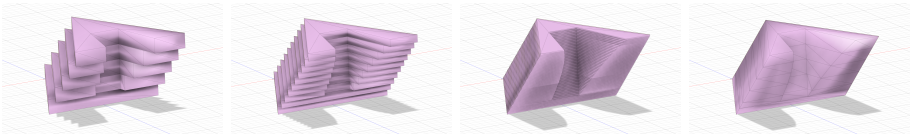

Figure 2: Outcomes of slab based system (panels 1 through 3) and an ideal connection set (panel 4).

Our approach to generating visibility polygons differs in that it includes identification of individual vertices (referred to as nodes) within the polygon. It also differs in that it is restricted to a particular area in order to preserve a degree of semblance to real world visibility.

The overall problem is commonly seen in medicine where various scans of the human body are made via MRI or CT as cross sectional slices and then reconstructed to provide a digital representation of organs and body parts. The desired result is to have an accurate and smooth three dimensional representation of the scan subjects. Keppel (Keppel 1975) began this surface approximation work back in 1975 and was followed by numerous researchers in subsequent years. The solution often involves minimising constraints and functions (Zou et al. 2015) and, in some cases, adding additional rules to ensure geometric stability (Bajaj, Coyle, and Lin 1996). To achieve the smoothness needed for organic tissue, curves and spline interpolation is commonly used. Although curves can be produced from revolving visibility polygons, we do not use such an approach. This is for three main reasons. Firstly, the majority of the connections are required to be linear rather than curved and would introduce inaccuracies otherwise. Secondly, storing and using curved surfaces requires a significant overhead in both space and complexity. Finally, calculations regarding curves (both to construct them and use them) are significantly slower than straight lines and triangles. Our aim is to develop a tool which quickly outputs a simple and generic mesh for further use capable to be worked with quickly.

Certain solutions use properties of the slices themselves (Barequet and Vaxman 2007). Indeed, this work extracts properties out of the vertices which make up the individual slices. Additionally, new vertices are created based on the properties of two consecutive slices. These are generated when contour overlays are being created based on the two slices. These contours are finally used to interpolate the surfaces. Our solution differs in that the properties we use are intrinsic to the creation of the vertices themselves. For instance, one vertex may be the result of terrain being seen, another may be caused by an occlusion. These properties are then leveraged to directly connect vertices from two separate layers.

We also share similarities with function minimisation. Indeed, when we cannot connect a vertex to an adjacent layer due to a lack of compatible properties, we fall back on minimising a distance function. This heuristic may not be correct in the general case, however we will see later how we can enforce conditions where this approach works well.

Another common problem addressed in (Meyers, Skinner, and Sloan 1992; Bermano, Vaxman, and Gotsman 2011) and (Boissonnat and Memari 2007) is branching as well as joining structures. This is prevalent in the medical industry as organic matter often branches out into multiple structures or different organs are connected to one another. Although interesting, these cases are of little use to us due to the nature of visibility. We focus our tool to generate meshes for only one observer. This means that only one visibility area is ever present in a slice thus no branching structures may be present. Finally, if we were to generate visibility manifolds for multiple observers at once, we could simply generate individual manifolds and add them to one another.

Branching structures are often accompanied with non parallel slices and fuzzy data. We do not need to worry about non parallel slices due to the fact

that we take data at discrete timesteps, resulting in parallel visibility polygons. We also have the advantage of being able to generate accurate visibility polygons without fuzzy data. As we mentioned previously, two dimensional visibility is a solved problem and is the basis of how we create our slices.

Many solutions generate additional data from the given slices. For instance, in (Barequet and Vaxman 2009), straight skeletons are generated and then projected onto the slices. Our method also generates additional data, however given that we can repeat the simulation, we can also generate slices at any given point in time by interpolating the observers position in the path. This means that we can create additional slices in cases where consecutive polygons lack enough similarities to be easily connected. Cases where our shortest distance heuristic would tend to fail are usually when a large number of connections rely on said heuristic across large gaps. Filling in these gaps with additional polygons both limits the distance between slices and increases the similarity of consecutive slices.

Prior work in game contexts is present and follows a technique where individual visibility polygons are stacked onto one another (Tremblay 2016). These polygons, however, do not connect to one another. Rather, they simply extrude directly in the vertical direction. This naive extrusion process causes inaccuracies in the resulting structure. To avoid this, many small slices and extrusions are generated. Although this may help in accuracy, it introduces artifacts dues to very small geometry. Additionally, such a large number of visibility polygons requires a large amount of storage and may slow down further calculations.

Our approach shares similarity with (Barequet and Sharir 1996) where, after two consecutive layers are connected, we stitch the remainder of the geometry by triangulating the areas between the layers such that we create a band of triangles from every two consecutive slices.

3 Methods

The visibility manifold construction algorithm, which is broken down into three main sections, will be discussed. To begin, section 3.1 will go over the ***parameters and data structures*** required. This includes the restrictions which may apply. This setup is then used in section 3.2 to explain how a ***visibility polygon*** is constructed. This will include the properties of the resulting polygons. Finally, these visibility polygon constructions are used in section 3.3 where they are connected to one another in order to form a ***visibility manifold***. Here, we go into detail on how the properties of the polygons and how their components are used to form the visibility manifold.

3.1 Parameters and Data Structures

In this section, the data structures and parameters used throughout the algorithm are introduced and explained. Section 3.1.1 begins by introducing ***vertex nodes*** which are the fundamental building blocks that define the geometry used to construct the geometry of interest. Afterwards, section 3.1.2 discusses the ***observer visibility*** by elaborating on the structure used in defining an observer path. The basic visibility triangle is introduced in this section as well. Following this, section 3.1.3 explains how ***obstacles*** are structured. This includes the introduction of obstacle vertices. Once this is done, the ***non similar polygon threshold*** will be briefly defined in section 3.1.4. Finally, section 3.1.5 contains ***constraints and guidelines*** on how to use the setup.

3.1.1 Vertex Nodes

Vertex nodes are a fundamental part of the system and are used in nearly every step leading up to the final construction. For the sake of simplicity and brevity,

we will simply refer to them as ***nodes***. We will now take some time to discuss their general properties.

Types: We begin by talking about node types. There are five different node types: obstacle, intercept, shadow, origin and leg. They are fleshed out in more detail throughout section 3.2 with the exception of obstacle nodes which are fleshed out in section 3.1.3. They do, however, all share certain general node properties.

Coordinates: Firstly, a node may be viewed as a class which can be instantiated with various properties. One such property is the ***node position***. This position is in the 2D plane and uses X and Y as its axes. In addition to a position in space, a node also has a position in time. This ***timestamp*** indicates when the node was created and observed. It also allows us to group it with other nodes in a group referred to as a time slice.

Identifiers: Each node is given an ***identifier*** upon instantiation. This identifier is typically a numerical value used to distinguish it from other nodes. It is important to note that the identifier itself it not necessarily unique. However, each node instance may be uniquely identified by using its identifier in conjunction with its timestamp and type. This allows a node to be traced as it evolves over different timeslices.

Neighbors: A node may also connect to other nodes by maintaining a list of ***neighbors***. These connections are always bidirectional and allow the construction of connected graphs. Given that the nodes are augmented with extra information (type, identifier), the resulting graphs have properties which allow us to construct the desired geometries. They also allow us to state and enforce graph properties which translate to geometric properties.

3.1.2 Observer Visibility

Path Pips: The visibility manifold is based on what the observer is able to see as he traverses the scene. We define said path as a discrete list of path ***pips***. Each pip consists of a position in 2D space, the angular orientation of the observer and a timestamp. The position indicates the placement of the observer in the scene, the angular orientation indicates where he is looking and the timestamp gives us the point in time.

Visibility Triangle: The visibility triangle is the area visible to the observer assuming no obstructions. It is in the shape of an isosceles triangle with the observer at the vertex defining the apex angle. The triangle is oriented such that the angular orientation of the observer faces the base of the triangle. The exact shape of isosceles triangle is defined by setting the size of the apex angle which sets the field of view angle.

In addition to this, we determine the scale of the triangle by setting its height. This sets the maximum distance at which the observer may see. These values, once set, are used throughout the entirety of the path the observer takes without being modified. They are set to be positive values and the apex angle must be less than 180 degrees in order for the visibility triangle to be well defined.

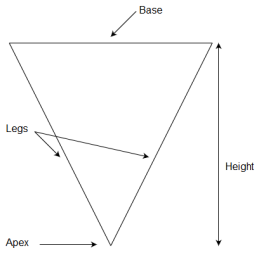

Figure 3: Terminology associated with isosceles triangles.

3.1.3 Obstacles

The scene which is navigated by the observer is built up of ***obstacle nodes*** connected to one another. Before proceeding further with obstacles, we will take a moment to introduce and explain obstacle nodes. Obstacle nodes come in two varieties: ***pinned*** and ***unpinned***.

Pinned and Unpinned Obstacle Nodes: The main difference between the two is that pinned obstacle nodes, once defined along with all of their properties, do not change at any point during the algorithm. They mainly serve as a reference graph which is used when constructing the manifold geometry. The reason for maintaining a separate graph of obstacle nodes is because the construction of the visibility geometry modifies graph structures.

Unpinned obstacle nodes are copies of pinned obstacle nodes and may be modified. They also hold a time coordinate whereas unpinned nodes do not since they do not associate with a particular point in time. As this suggests, the scene is static and has no moving objects or obstacles. This also means that no new pinned nodes may be introduced. This allows us to know in advance how many pinned nodes we will have and assign to each a unique number as an identifier.

Generating Unpinned Nodes: When copying a pinned node, the unpinned variant will inherit the identifier but maintains uniqueness in that it is unpinned and has a timestamp. This does, however, mean that an obstacle node may only be copied once per time frame. Throughout the remainder of our discussion, we will refer to unpinned obstacle nodes simply as obstacle nodes and will illustrate them as red dots while pinned obstacle nodes will be referred to explicitly and will be illustrated as grey dots.

Restrictions: The obstacles in the scene are defined as pinned obstacle nodes

connected to one another. It is important to note that edges connecting obstacle nodes must not cross. If they do, then the geometric graph connecting the points will be ill defined. This can be resolved by adding an extra pinned obstacle node at the intersection point. Another restriction is that a pinned obstacle node may not exist without neighbors. This is because such an obstacle would cast an infinitely small shadow, making a visibility polygon and manifold ill defined.

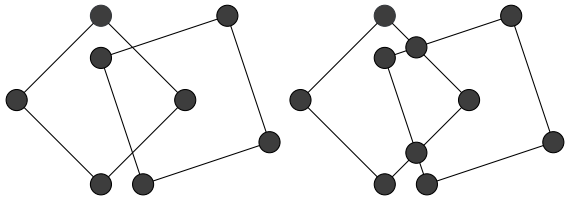

Figure 4: On the left we have obstacles with crossing edges. We fix this on the right by adding nodes to the points where the edges overlap.

Directionality: One may notice that we have not made any mention of directionality in terms of inside versus outside. This is because the obstacles are not considered to be "filled". Rather, the edges connecting obstacle nodes are walls which block visibility. This allows a box obstacle to either be a room an observer cannot see outside of, or a room which he cannot see into.

3.1.4 Non Similar Polygon Threshold

There is one additional parameter to be set which is used when connecting visibility polygons into the final visibility manifold. This constraint is the ***non similar polygon threshold*** which sets the maximum time difference between two consecutive non similar visibility polygons. This parameter is set heuristically and is typically relatively small. This setting is primarily used in generating extra intermediate layers.

3.1.5 Constraints and Guidelines

Finally, we have some constraints and guidelines when setting up the scene. When generating paths, it is advised for those paths to have a fairly smooth transition from one pip to the next. This include position, angular orientation and time position. For instance, if we wish to rotate by 180 degrees over 1 seconds, it is better to do so over several steps by subdividing both the direction and time frame into smaller pieces rather than go from 0 degrees at time 0 directly to 180 degrees at time 1 in a single step.

Additionally, it is suggested to keep the visibility angle and distance in mind when setting time frames. Setting a large value for either can lead to large changes unless the time steps are small. As stated previously, large changes from one step to another causes crude approximations in the output.

Providing specific values or thresholds for these parameters is difficult as they greatly depend on the terrain (both scale and geometry), path (how coarse it is and how close it comes to terrain) as well as one another. For instance, in a situation where the path does not come near any corner and is very fine grain, then larger values may be used for both the field of view angle and view distance. Parameters would need to be tuned according to the users needs and situation.

Finally, an observer cannot move through walls from one step to the next. We consider this to be ill defined movements and thus, causes ill defined outputs.

3.2 Visibility Polygon

In this section, we go over the step by step process on how to build the visibility polygon. The process begins in section 3.2.1 where the ***visibility triangle is created*** using a path pip. It is a triangular shape which helps guide the visibility

polygon graph construction. Once this is done the relevant ***obstacle nodes are gathered*** in section 3.2.2. Afterwards, this connected graph of obstacle nodes will be adapted to the visibility triangle. This is accomplished in section 3.2.3 by ***splitting node connections*** with newly defined intercept nodes. Section 3.2.4 shows how these intercept nodes are connected to one another to ***form the triangle base***. Afterwards, section 3.2.5 proceeds by ***casting shadows*** which involves the introduction of shadow nodes. The resulting graph is then augmented by creating the ***origin and leg nodes*** in section 3.2.6. These nodes are associated with the apex of the visibility triangle. Once this graph is built, section 3.2.7 moves on to ***defining the visibility polygon***. The generated polygon ***properties*** and representation are then discussed and elaborated in section 3.2.8.

3.2.1 Creating the Visibility Triangle

We begin by generating the ***visibility triangle*** associated with the current path pip. It is important to note that this area is distinct from the visibility polygon. The visibility triangle is purely a geometric shape with no nodes nor graph like properties. It's main use is to help guide the construction of the visibility polygon. To represent it, three vertices are defined which set the three corners.

To set the first corner, the pip location is directly used. This is where the observer is positioned. The two vertices defining the base of the triangle require some trigonometry. First, the length l of the legs must be determined.

$$l = \frac{h}{cos(v)} \tag{1}$$

where h is the height of the triangle and v is the apex angle. This value is then

used in calculating the position of the leg corners.

$$\begin{aligned} c_x &= pip_x + l * cos(pip_{orientation} \pm v) \\ c_y &= pip_y + l * sin(pip_{orientation} \pm v) \end{aligned} \tag{2}$$

The $\pm$ is used due to the symmetry of defining the two base vertices, where '-' defines the left vertex and '+' defines the right vertex.

3.2.2 Gathering Obstacle Nodes

We begin constructing the graph by identifying obstacle nodes which may affect visibility. This is done by copying pinned obstacle nodes into unpinned obstacle nodes that have either of the following two properties:

- The node position is contained within the visibility triangle.
- The node has an edge that crosses one of the visibility triangle edges.

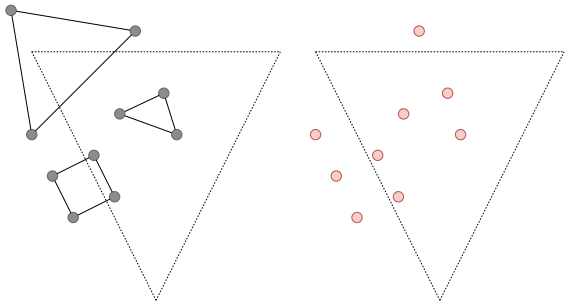

Figure 5: After identifying all pinned nodes (left) affecting visibility, they are copied into unpinned nodes (right).

Once the set of obstacle nodes needed is determined, they are copied, resulting in a set of disconnected obstacle nodes. All of these nodes are given the timestamp associated with the current path pip. These nodes are then connected to one another such that they mimic the original unpinned graph.

In order to do this for a specific node **n** in the new set, the pinned counterpart to this node is selected from the original set. Once this is done, for each node in the neighbor list, the unpinned counterpart is searched for in the new set of unpinned nodes. If such a node exists, we connect it to n. Otherwise, we move onto the next entry in the list.

```
1
2  void connectNodes(List<Node> unpinned_set, List<Node> pinned_set)
3  {
4    for (int i = 0; i < unpinned_set.size(); i++)
5    {
6      Node current_node = unpinned_set[i];
7      Node pinned_node = getCounterpart(current_node, pinned_set);
8      List<Node> pinned_neighbors = pinned_node.neighbors();
9      for (int j = 0; j < pinned_neighbors.size(); j++)
10     {
11       Node unpinned_neighbor = getCounterpart(pinned_neighbors[j], unpinned_set);
12       if (unpinned_neighbor != null)
13         current_node.addNeighbor(unpinned_neighbor);
14     }
15   }
16 }
17
18 Node getCounterpart(Node node, List<Node> node_set)
19 {
20   for (int i = 0; i < node_set.size(); i++)
21     if (node_set[i].identifier == node.identifier)
22       return node_set[i];
23   return null;
24 }
```

This process results in a copy of the subset of the pinned obstacle nodes graph impacting visibility which will be modified and augmented with additional nodes.

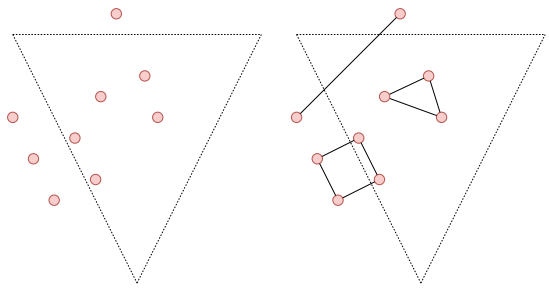

Figure 6: Using the original pinned nodes (found in figure 3), we connect the unpinned nodes.

3.2.3 Splitting Node Connections

Next the graph edges which cross the visibility triangle edges are subdivided. This prevents edges from crossing one another in the graph during future steps. This is done in two parts. The first part, which is handled in this subsection, addresses the edges crossing over the base of the triangle. The second part where the legs of the triangle are addressed is handled when casting shadows.

The first step is to identify which edges intersect the visibility triangle base as well as where this intersection occurs. Using one of these intersections points, a node of type ***intercept*** is created. The previous connection between the two obstacle nodes in this edge is removed. Each of the two nodes is then connected to the newly formed intercept node.

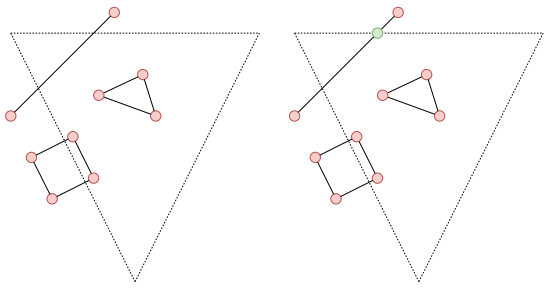

Figure 7: All edges intercepting the base are split into two new edges connected to an intercept node.

We will now take a moment to introduce the intercept node type. As has been seen, these nodes are the result of obstacle node edges intercepting the base of the visibility triangle. They inherit the timestamp from their parent obstacle nodes as well as their identifiers. The two intercept node identifiers are combined into a tuple, ordered from smallest to largest. This ordering is to ensure consistency among intercept nodes. Whether obstacle node A is behind the triangle or obstacle node B is behind the triangle, the resulting intercept

node has the same identifier.

It is also worth noting that all nodes of this type will be colinear to one another given that they all originate from points on the line defining the base of the visibility triangle. When illustrating these nodes, they will be presented as green points.

3.2.4 Forming the Triangle Base Graph

The current graph is now augmented by adding the base of the visibility triangle as a graph. The first step is to create two new dummy nodes holding no special properties. They are placed at each of the two corners of the triangle base.

Either of the two dummy nodes is now selected. A list of all the intercept nodes is then generated. This list needs to be ordered based on distance between the intercept node and the selected dummy node. The selected dummy node is prepended to this list while the other dummy node is appended to it. Each node in the list is then connected to the node before and after it.

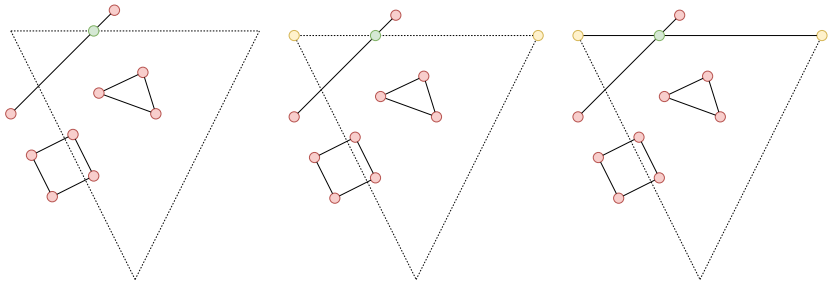

Figure 8: Using the visibility triangle, we add the two dummy nodes in yellow (middle figure). After this, we connect all the intercept nodes in a graph representing the base of the visibility triangle.

This forms a graph representing the base of the visibility triangle with the two dummy nodes at the ends. Had we chosen the other dummy node, this construction would have lead to the same result.

3.2.5 Casting Shadows

The process now involves casting shadows in the form of ***shadow nodes***. Shadow nodes, displayed in blue, are nodes representing the location behind an obstacle node which is hidden from the observer. Their identifiers are the same as their obstacle counterpart, and thus, are unique among shadow nodes.

To cast these shadow node, we iterate over all the obstacle nodes contained within the visibility triangle. For each such node, we determine the line which passes through both the obstacle node position and the observer position. This line is then used to find all edges intersecting it as well as their associated intersection points. Out of these points, the one closest to the observer position is selected. If this point is closer to the observer than the obstacle node itself, it is ignored and we move onto the next obstacle. Otherwise, this point becomes a shadow node and splits the edge in the same fashion as intercept nodes. In addition to splitting the edge, this shadow node also sets the obstacle node associated to it as its neighbor.

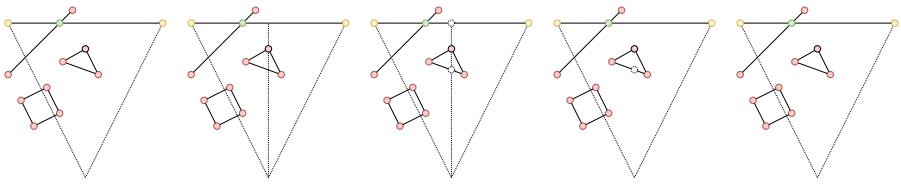

Figure 9: We begin by selecting a node in the first panel. In the second panel, a line passing through the selected node and the observer is drawn. In the following panel, points representing intersections are shown. In the fourth panel, the point closest to the observer is selected. This point is closer than the obstacle node so it is ignored, shown in the final panel.

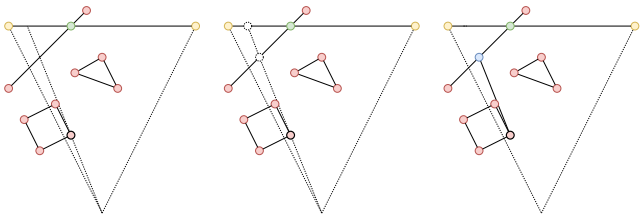

Figure 10: The same process is applied here as in figure 7, however, the point closest to the observer position is further than the obstacle node it originated from and thus, is turned into a shadow node.

3.2.6 Creating the Origin and Leg Nodes

In order to finish constructing the visibility graph, the ***origin*** and ***leg nodes*** must be added.

Origin Node Starting with the origin node, it is a unique node, displayed in pink and placed at the pip position. It is meant to represent the observer himself and the beginning of his visibility area. There is only one origin node per time frame which is why no identifier is associated with it. The only neighbors that the origin node may have is the two leg nodes.

Leg Node Leg nodes, represented in purple, are similar to shadow nodes which are cast in the direction of the legs of the visibility triangle. As mentioned previously, leg nodes are connected to the origin node.

When a leg node is cast onto the base of the visibility triangle, it replaces the dummy node. This includes taking on neighbors the dummy node had as well as its position. If this is not the case and there is an obstacle between the base and origin node. The leg node simply takes on the position it is cast upon in this situation.

Finally, the two leg nodes are given the identifiers "0" and "1" corresponding to left and right. Since there are always only two leg nodes, we know that they are unique. For both leg nodes and the origin node, they are assigned the same

timestamp as the rest of the nodes in the graph.

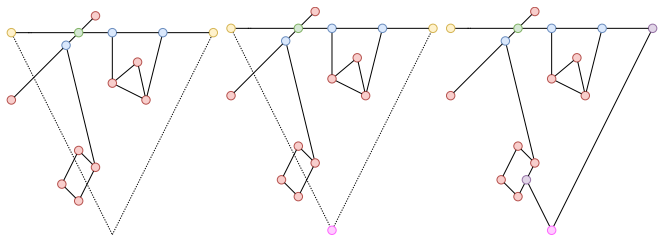

Figure 11: An origin node is added to the graph (middle panel). The leg nodes are then added in the right panel. The left leg node does not reach the base of the visibility triangle while the one on the right does, and so, replaces the right dummy node.

3.2.7 Defining the Visibility Polygon

Now the visibility polygon can finally be defined. This is done by traversing the graph starting at the origin node while always taking the "right most turn" until we return to the origin node.

To do so, we begin by creating two references. One is the current node reference, used as the current position during the graph traversal. The second is the previous node reference. This reference is used to determine which following node is the correct "right most turn".

To begin, the origin node is set as the previous node while the left leg node is set as the current node. In the following part, we loop until the current node reference is set to the origin node.

All the neighboring nodes of the current node are checked to find and select the one with the smallest angle with respect to the previous node. The current node is saved into a list of visited nodes which will be our visibility polygon. The current node is set as the previous node while the smallest angle node is set as the current node.

Once the current node is set to be the origin node, we break out of this loop.

The visited node list is then returned as the visibility polygon associated with the current path pip.

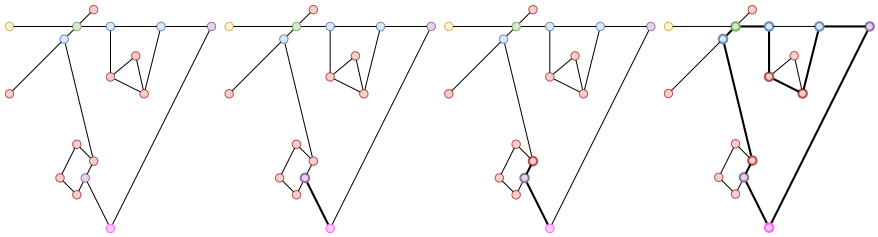

Figure 12: Visualisation of the visibility polygon graph construction.

```

List<Node> constructGraph(Node origin, Node left_leg)
{
    List<Node> polygon;
    Node* current = &left_leg;
    Node* previous = &origin;
    Node* next;

    while (current != &origin)
    {
        polygon.push_back(*current);
        next = getRightMostNeighbor(current, previous);
        previous = current;
        current = next;
    }
    return polygon;
}

Node* getRightMostNeighbor(Node* current, Node* previous)
{
    float min_angle = infinity;
    float min_node*;

    for (int i = 0; i < current->neighbors.size(); i++)
    {
        Node* current_node = &current->neighbors[i];
        float current_angle = getAngle(previous, current, current_node);
        if (current_angle > max_angle)
        {
            min_node = current_node;
            min_angle = current_angle;
        }
    }
    return min_node;
}
```

3.2.8 Polygon Properties

Now that a process for constructing visibility polygons has been established, their properties may be discussed.

Rather than being a simple set of points, the visibility polygon we construct is stored as a list of nodes. Given that these nodes all contain properties independent from the visibility polygon, said polygon is augmented with this additional node data. This also opens up the possibility to represent visibility polygons in a different fashion. Thus far, we have shown each node in its 2D position. Instead of this, the list can be displayed as a horizontal line with all the nodes in the order used to define the polygon. The ends both contain the same origin node to represent that this is a closed polygon.

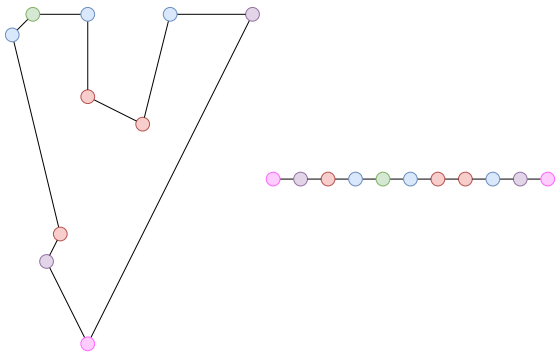

Figure 13: Taking the complete visibility polygon and representing it as a list of nodes with the same origin node at both ends.

Visibility polygons also have a similarity property between each other. Two polygons can either be similar or not. Note that this is distinct from an ordering. For two polygons to be similar, they must both have the same node types with the same identifiers in the same order. This essentially means that the lists from two polygons must be identical with the exception of the node positions (in 2D space) and their timestamps.

Figure 14: On the left, we can see two similar polygons (assuming all identifiers match) whereas on the right, the two polygons are non similar.

There are two ways of displaying the visibility polygon itself. We can use the reference frame of the terrain, allowing us to see how the observer changes position and orientation. This is useful to see how the visibility area changes when interacting with the terrain and how various movements impact the polygon itself. On the other hand, we can use the reference frame of the observer, leading the visibility polygons to always face the same direction and have the same position. This become practical when focusing on how individual polygons change with respect to one another as well as how they connect to one another.

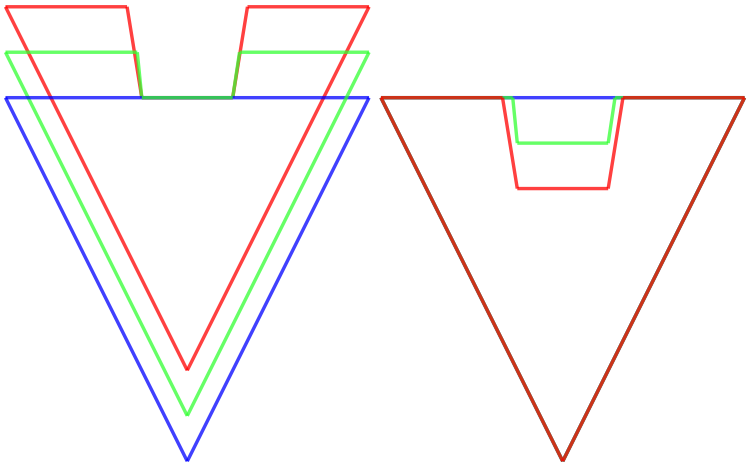

Figure 15: Illustration of different reference frames. On the left we show the polygons in the terrain frame and on the right we show the polygons in the observer frame.

3.3 3D Manifold

Now that the system for building augmented visibility polygons has been established, the system for connecting visibility polygons into a visibility manifold can be discussed. Firstly, consecutive ***similar visibility polygons*** will be connected to one another in section 3.3.1. Following this, consecutive ***non similar visibility polygons*** will be connected in section 3.3.2. This process is broken down into two separate parts. In the first part, ***additional polygons*** will

be generated in between non similar layers to help bridge the non similarities. In the second part, these ***newly generated polygons will be connected***. Finally, the connections between each consecutive set of layers will be ***triangulated*** in order to express the resulting graph as an orientable manifold. The process is explained in section 3.3.3 and includes the triangulation of the first and last layers to ensure that the generated manifold is closed.

3.3.1 Similar Visibility Polygons

The process for connecting two similar visibility polygons is straight forward. Since these two layers are similar, each node in one layer has a corresponding node in the other layer. These corresponding nodes have the same type and identifier, although have different timestamps. Given such a pair of nodes, they are connected to one another. This is done for all the nodes in either layer.

Figure 16: Demonstration of connecting similar polygons.

3.3.2 Non Similar Visibility Polygons

Before being able to directly connect non similar visibility polygons, additional intermediate polygons must first be generated. When a non similarity between two layers is found, two additional layers are generated for each detected discontinuity. A discontinuity is a point in time where visibility polygons immediately after and immediately before are non similar. These new layers are generated such that they are on either side of a discontinuity. Each pair is separated from one another by, at most, the ***non similar polygon threshold*** discussed in section 3.1.4.

Generating Additional Visibility Polygons: The first step in generating these polygons is to find the points in time where they are located. To do so, a modified binary search is used.

To begin, two references to visibility polygons are generated. These represent the desired polygons and are set to the two non similar layers that are being connected. These layer references will be referred to as layer A, associated with **after** the discontinuity, and layer B, associated with **before** the discontinuity.

The pip data from layers A and B is then averaged to create a new pip. This pip is used to generate a new visibility polygon called N. This new polygon is then compared to B. If N and B are similar, B is set to N. Otherwise, A is set to N. This process is repeated until the time difference between A and B is less than the non similar polygon threshold.

The result is a series of unconnected layers starting with the original Before layer, the newly generated layers B and A, and then the original After layer. It is important to note that layer B and the original before layer will be similar, but layer A and the original after layer might not. This would be caused by having multiple discontinuities in between two layers. In this case, the process needs to be repeated for layer A and the original after layer.

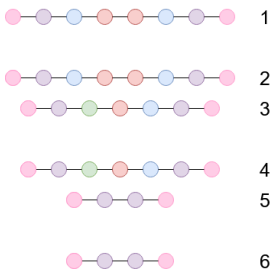

Figure 17: Possible outcome of generating intermediate layers. The top and bottom layers are the originals whilst layers 2, 3, 4 and 5 were added. Discontinuities are present between layer 2 to 3 and 4 to 5.

```
1
2  // Previously defined.
3  float non_similarity_threshold;
4  List<Polygon> polygon_list;     // List of polygons defining the manifold.
5
6  void buildIntermediate(Polygon poly_0, Polygon poly_1)
7  {
8    Polygon* B = &poly_0;
9    Polygon* A = &poly_1;
10
11   while(A->pip.timestamp - B->pip.timestamp > non_similarity_threshold)
12     buildNextInterval(B, A);
13
14   insertNewPolygons(B, A);
15
16   if (!Polygon.similar(A, poly_1))
17     connectNonSimilar(A, poly_1);
18 }
19
20 void buildNextInterval(Polygon* B, Polygon* A)
21 {
22   Polygon* N;
23   Pip intermediate_pip = (B->pip + A->pip) / 2;
24   N = buildPolygon(intermediate_pip);
25   if (Polygon.similar(B, N))
26     B = N;
27   else
28     A = N;
29 }
30
31 void insertNewPolygons(Polygon* B, Polygon* A)
32 {
33   Polygon poly_1 = polygon_list.pop();
34   Polygon poly_0 = polygon_list.back();
35
36   if (B != &poly_0)
37     polygon_list.push_back(*B);
38   if (A != &poly_1)
39     polygon_list.push_back(*A);
40
41   polygon_list.push_back(poly_1);
42 }
```

The result of the entire process is a stack of disconnected visibility polygons where the original before layer is at the bottom and the original after layer is at the top. Any two consecutive layers in this stack are either similar or are separated by at most the value of the non similar polygon threshold.

Connecting Additional Visibility Polygons: The generated stack can now be connected. Consecutive layers which are similar are connected as per usual. When two layers are non similar, the common nodes are connected to one another. After this, all the unconnected nodes are identified.

For each such node, the closest node (in space) on the opposing layer is

found. It is important to note that this node pair cannot cross another node pair (edges should not cross). It is possible for the closest node to already have a connection so long as the previous point still holds. Out of all the unconnected nodes, the one with the smallest distance to its nearest neighbor is chosen and connected.

This process is repeated for the remaining unconnected nodes. By the end of this process, all nodes in one layer will connect to at least one node in the other layer.

```

// Previously defined.
List<Polygon> polygon_list;

void connect(int current)
{
  while (current != polygon_list.size() - 1)
  {
    if (Polygon.similar(polygon_list[current], polygon_list[current + 1]))
      connectSimilar(polygon_list[current], polygon_list[current + 1]);
    else
      connectNonSimilar(polygon_list[current], polygon_list[current + 1]);
  }
}

void connectNonSimilar(Polygon B, Polygon A)
{
  connectSimilarNodes(B, A);

  while (B.hasSingleNodes() || A.hasSingleNodes())
    connectNonSimilarNodes(B, A);
}

void connectNonSimilarNodes(Polygon B, Polygon A)
{
  List<Node> single_B = getSingleNodes(B);
  List<Node> closest_B = getAllClosestNodes(single_B, A);
  List<float> distances_B = getDistances(single_B, closest_B);

  List<Node> single_A = getSingleNodes(A);
  List<Node> closest_A = getAllClosestNodes(single_A, B);
  List<float> distances_A = getDistances(single_A, closest_A);

  int smallest_B = getSmallestDistanceIndex(distances_B);
  int smallest_A = getSmallestDistanceIndex(distances_A);

  if (smallest_A == -1 || distances_B[smallest_B] < distances_A[smallest_A])
    connect(single_B[smallest_B], closest_B[smallest_B]);
  else
    connect(single_A[smallest_A], closest_A[smallest_A]);
}

void List<Node> getAllClosestNodes(List<Node> single_B, Polygon A)
{
  List<Node> closest_nodes;
  for (int i = 0; i < single_B.size(); i++)
    closest_nodes.push_back(A.getClosest(single_B[i]); // Factors in edge crossing.
  return closest_nodes;
}
```

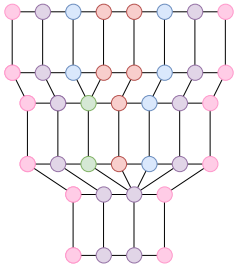

Figure 18: Possible outcome of connecting non similar polygons.

3.3.3 Triangulation

All that remains now is to triangulate the resulting graphs in order to have a closed orientable manifold. For this process, the connections between the layers must be triangulated. After these sides are triangulated, the top and bottom visibility polygons are triangulated as well to cap off the manifold.

Each consecutive layer pair is triangulated independently from the rest. For a given pair, a pair of node references is generated. The node references, labeled B and T, refer to the origin nodes of the bottom and top layers respectively.

At this point, two properties need to be checked. If the node following B (B.next) in the bottom visibility polygon is connected to T, then this node is set as the new B.

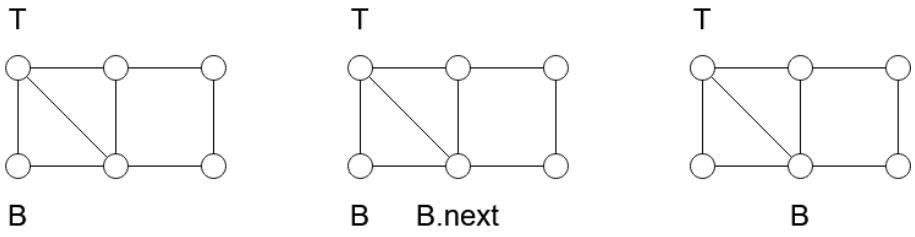

Figure 19: Visualisation of the case where B.next and T are connected. B.next becomes the new B.

If this is not the case, the node following T (T.next) is checked for connec-

tivity with B. If so, then this new node is set to T.

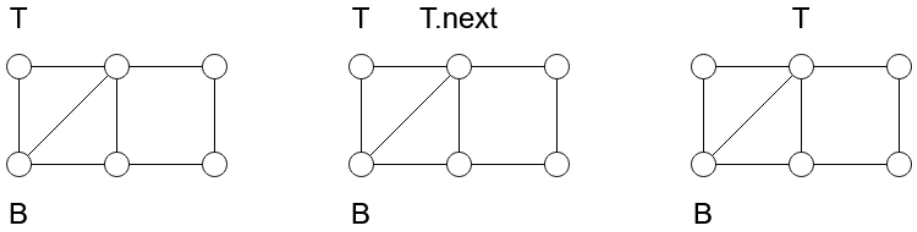

Figure 20: Visualisation of the case where T.next and B are connected. T.next becomes the new T.

These properties effectively check whether or not a triangle is already formed. If neither case holds, then no such triangle exists and a connection between B→next and T is formed.

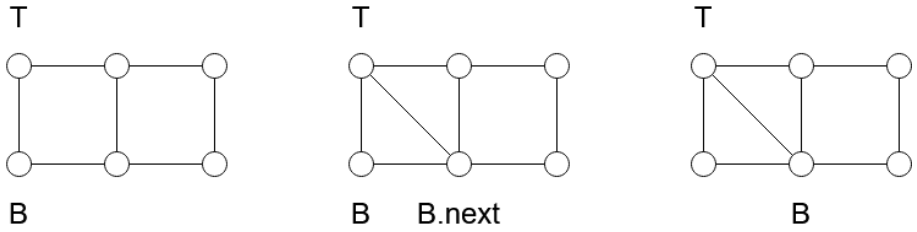

Figure 21: If neither B.next is connected to T nor T.next is connected to B, we connect T to B.next. On the next iteration of the algorithm, we fall into the first case where B.next becomes the new B.

```
void triangulate(Polygon p0, Polygon p1)
{
  int i0 = 0;
  int i1 = 0;

  List<Node> n0 = p0.nodes();
  List<Node> n1 = p1.nodes();

  while (i0 != n0.size() || i1 != n1.size())
  {
    if (i0 != n0.size() && isConnected(n0[i0 + 1], n1[i1]))
      i0++;
    else if (i1 != n1.size() && isConnected(n0[i0], n1[i1 + 1]))
      i1++;
    else if (i1 != n1.size())
      connect(n0[i0], n1[i1 + 1]);
    else
      connect(n0[i0 + 1], n1[i1]);
  }
}
```

This process is repeated until both B and T refer to the bottom and top origin vertices once again. To triangulate the top and bottom visibility polygons, the ear clipping method is used.

The top and bottoms are triangulated using (Meisters 1975) two ears theorem. We locate one ear of the polygon at a time and turn it into a triangle, remove it and repeat the process on the remaining polygon until only a triangle remains.

The result is a set of triangles which builds a closed orientable manifold. This manifold represents the visibility manifold visible by the observer throughout his path.

4 Experiments

The experiments are divided into two categories. In the first category, preliminary experiments are conducted to examine how varying parameters such as similarity threshold and step size affect the the resulting manifold.

Once the results from these experiments are gathered, additional experi-

ments are conducted to observe the output based on various terrains. These experiments provide insight into the behaviour of the system when used in real applications.

4.1 Visibility Parameters

Two main parameters are studied in the preliminary experiments. The first parameter is the similarity threshold where its size is changed from 1 to 0.25 and then to 0.125. The step size used is 1, which meaning that with a threshold of 1, no additional layers will be generated.

The effects are observed over a simple scenario where an observer strafes into a wall over the course of a single step. The point at which the wall comes into the observers visibility, causing a discontinuity in the visibility, is 0.375 time units.

The second observed parameter is the step size. For this, we conduct tests using three different movement types: parallel to where the observer faces (referred to as walking), perpendicular to where the observer faces (referred to as strafing) and rotation.

For all three movement types, a simple box is used as the terrain. This is because linear movements, such as walking and strafing, generates prisms with no curved faces in an empty terrain. These offer no interesting data with regards to step size.

Each movement type uses two paths. One is the coarse version which is described bellow and one is the fine version which follows the same trajectory but is subdivided to generate 4 times as many steps.

All three movement types use time step sizes of 1 for the coarse paths. Walking movement is 1 unit per step, strafing is 0.5 units per step and rotation is at 11.25 degrees per step. All of these are reduced to 25% for the finer path

variant.

4.1.1 Similarity threshold

We observe that for a threshold size of 1, no intermediate polygons are produced, as expected. This means that layer 0 and layer 1 are connected directly to one another. The resulting manifold is well formed but is not properly covering the volume when compared to the ideal manifold show in Figure 24 a). This shows the necessity of generating additional geometry to accurately connect non similar polygons.

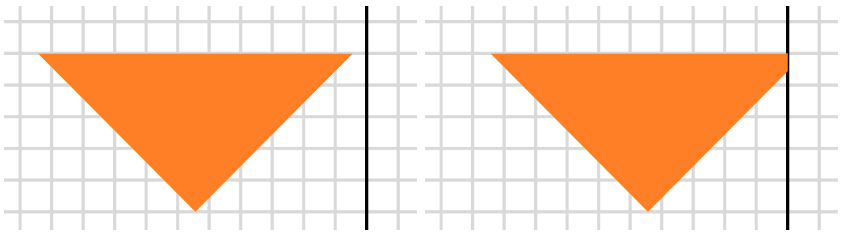

Figure 22: Figure showing the path used to study threshold size. On the left is the starting position and on the right is the end position.

Moving on to a threshold size of 0.25, we can see two new layers appearing around where the obstruction begins. These two layers are at t = 0.25 and t = 0.5 and are similar to layers 0 and 1 respectively. This similarity means that the connections between layers 0.25 and 0 as well as the connections between layers 1 and 0.5 are accurate to the ideal output. We still observe a noticeable lack of coverage between the two additional layers, however their closer proximity to the discontinuity as well as to one another leads to smaller inaccuracies.

Reducing the threshold further to 0.125, we can see that the intermediate layers have moved in even closer. They are now at t = 0.5 and t = 0.375 and are similar to layer 1 and layer 0 respectively. Just as in the previous test, the connections between the similar polygons are accurate to the ideal output.

We also observe that, in this test, the generated connections between the two intermediate layers matches the ideal manifold. This is due to the fact that one of the intermediate layers happens to fall exactly (or nearly) on the discontinuity.

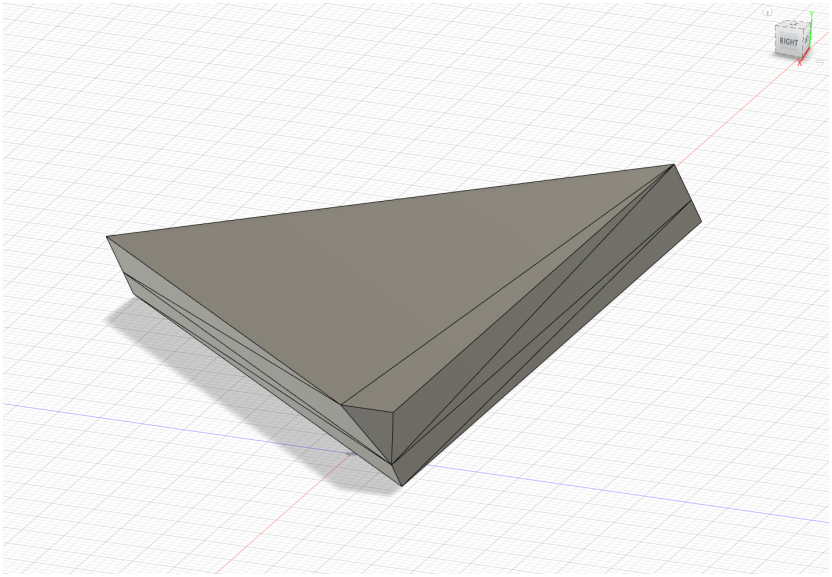

Figure 23: A zoomed out view to show what the ideal manifold looks like. We can see in the bottom of the image how the manifold seems cuts off because of loss of visibility due to the wall.

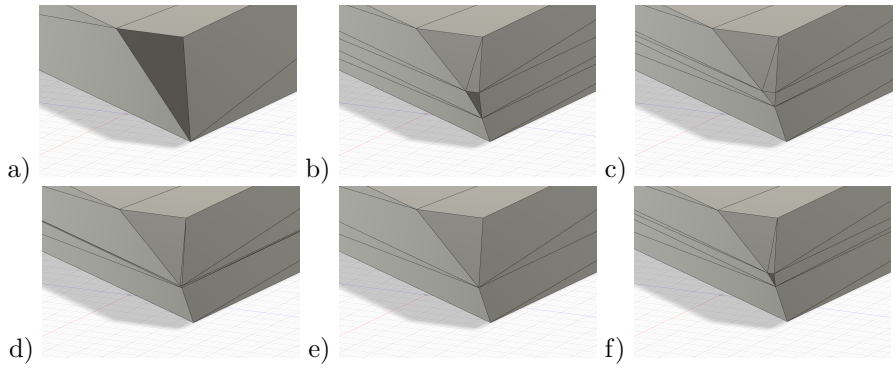

Figure 24: Visualisation of the connections. In the top row we have threshold sizes 1, 0.25 and 0.125 whereas the bottom contains threshold size of 0.02 and the ideal manifold. The last figure shows a manifold generated with threshold size 0.125 but with a different offset for when the discontinuity occurs.

This is a coincidence worth noting as it can falsely lead us to believe that a threshold size of 0.125 is ideal. It can be easily demonstrate that this is not the case by simply varying the point in time where the discontinuity occurs.

The worst possible situation is when the discontinuity is half way between two intermediate layers. This scenario is shown in Figure 24 f) where we took the same test but moved the observer such that obstruction begins at $t = 0.4375$ rather than $t = 0.375$. Here we observe that, once again, the space between the non similar polygons is missing volume which is present in the ideal manifold.

We then decide to set the threshold to 0.02 which results in Figure 24 d). This is a near ideal output, matching the ideal very closely. The main difference is that what appears to be a single layer at the discontinuity is, in fact, two layers very close to one another. This, in turn, means that the connections between these two layers cause extremely thin triangles which may be unsuitable for certain applications.

4.1.2 Step size

The first observation we make is that any differences that arise between the coarse and fine tests occur exclusively when trying to triangulate non planar quads. This makes sense as a finer mesh subdivision of a flat plane does not reveal additional details in the structure. This does, however, lead to flat faces with unnecessarily triangulated surfaces.

Non planar quads occur when some sort of rotational motion is present. The rotational motion when the observer rotates is self evident. The rotational motion in the strafing and walking test is less obvious. It occurs when a shadow is cast from an obstacle node and the observer moves tangentially with respect to said node. The tangential movement of the observer translates to tangential movement of the shadow which defines rotational motion.

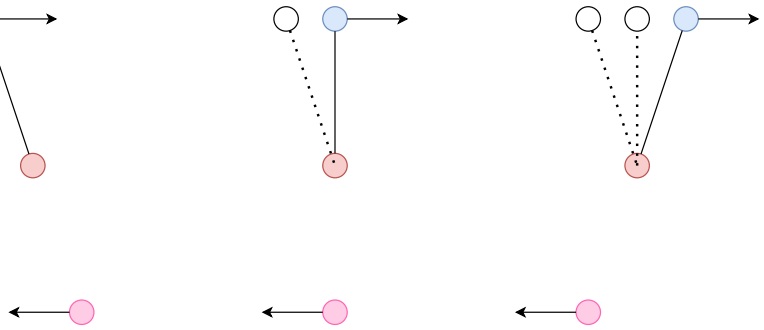

Figure 25: Illustration of how the observer moving in straight motion causes rotational motion of the shadow node with respect to its casting obstacle node. As the observer (pink) moves in a straight line towards the left, the shadow node (blue), moves towards the right. In the last panel, we can see how the result is a rotating line.

This means that the type of observer motion is not particularly important so long as non planar quads can be formed. This allows us to focus our analysis of step sizes onto a particular case which can be generalised to all cases.

For this reason, we use a single non planar quad to illustrate our observations as opposed to full manifolds.

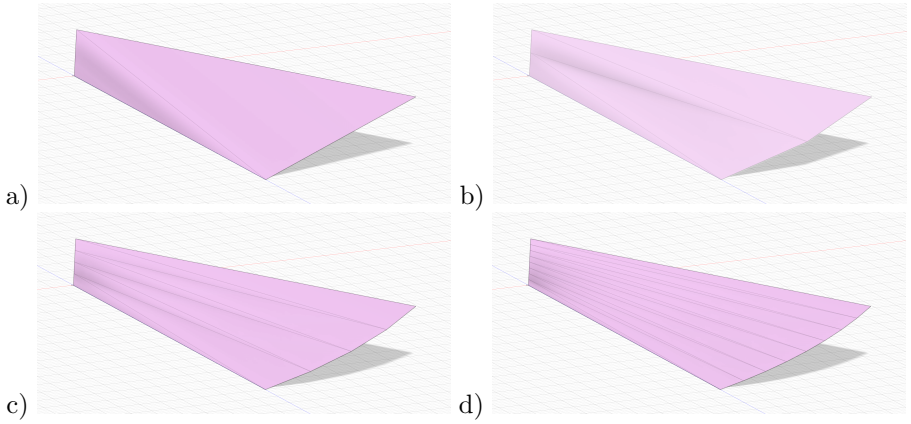

Figure 26: Visualisation of connecting a non planar quad. The step size in a) is 1 for time and 16 degree for the angle. This is halved in each subsequent figure.

Our model assumes that a naive triangulation of a quad gives a good approximation to a smooth surface when, in reality, it does not. This leaves certain volumes either incorrectly included or incorrectly excluded from the manifold.

This is mitigated by decreasing the step size within the path. Figure 26 shows the effects of using the same paths with smaller step sizes. As we can see, smaller subdivisions allow the overall manifold to more closely resemble an ideal curve.

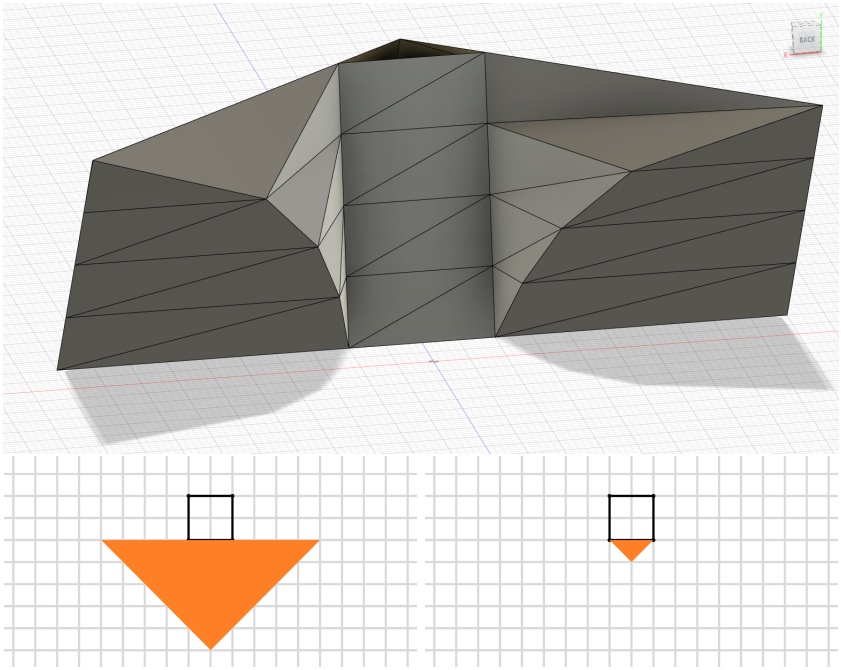

Figure 27: A manifold generated by moving towards a square obstacle. The path is such that no non similar polygons are generated.

Additionally, we see that less the points appear to be planar, the worse the approximation is. This is clear in the experiment in figure Figure 27 where the faces towards the bottom of the manifold are more accurate than the ones towards the top.

Large non planar quads are not only caused by large step sizes but also when the distance between a shadow and its casting node is far greater than the distance between the observer and said node. This is evident in Figure 27 as, towards the bottom, the observer is far from the obstacle and gets closer, causing the quads towards the top to become increasingly non planar.

We can mitigate this by avoiding close proximity to obstacles, reducing the visibility depth to prevent long shadow-caster distances, or avoiding paths that stray too close to obstacles. These solutions, unfortunately, rely on the user

having this knowledge and are not done automatically.

4.2 Terrain

The terrain experiments are conducted to observe how the limitations caused by the manifold parameters affect the output depending on terrain. More specifically, we observe the impact of the similarity threshold.

In the previous section, we established that step size impacts the output only when rotational motion is present. Having already analysed this impact, we forgo further analysis in this section.

When analysing threshold size impact, we ensure that all discontinuities fall in the center of two intermediate layers. This artificial restriction allows us to see and analyse the worst case scenario for real world situations.

Three terrain variants are used in these experiments.

Flat: The simplest and most straight forward terrain is a flat wall. No obstacle nodes are present within the visibility area at any point.

Internal: This terrain type involves obstacle nodes. However, these nodes form angles of more than or equal to 180 degrees when observed by the observer, leading to no possible shadows.

External: This terrain type involves obstacle nodes which form angles of less than 180 degrees when observed by the observer. This sort of terrain leads to shadow nodes.

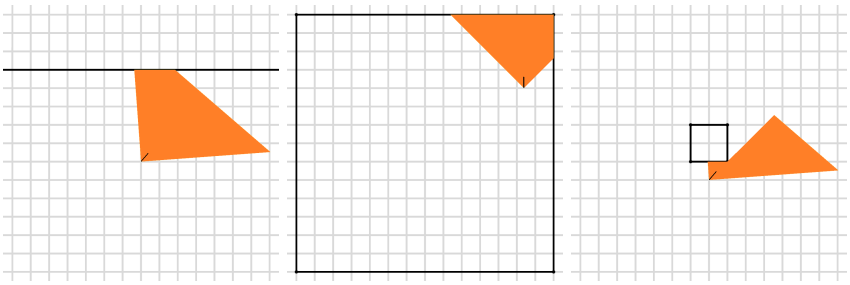

Figure 28: From left to right we see examples of flat, internal and external terrain.

It is important to note that we are interested in how the terrain is perceived by the observer rather than inherent terrain properties. We can use a square obstacle and place the observer within the box, resulting in an internal setup, place the observer outside, resulting in an external setup or ensuring that the observer only looks at walls (no nodes) resulting in a flat setup.

For all three terrain types, the observer undergoes the same three motions we saw in the step size tests in section 4.1.2. This results in a very large number of complex manifolds, each of which is both challenging to represent as a static image and discuss. For this reason, we select a subset of manifolds which concisely represent the resulting properties and limitations.

4.2.1 Missing Volume

Regardless of what kind of motion is used, when involving flat or internal terrain, the generated geometry around discontinuities excludes volumes which would otherwise be included in the ideal manifold.

This is due to the convex nature of the generated polygons as well as the restrictions we impose on naive connections. Flat and internal terrain means that the visibility polygons are exclusively convex. In addition to this, we use a smallest distance metric when connecting nodes in non similar polygons.

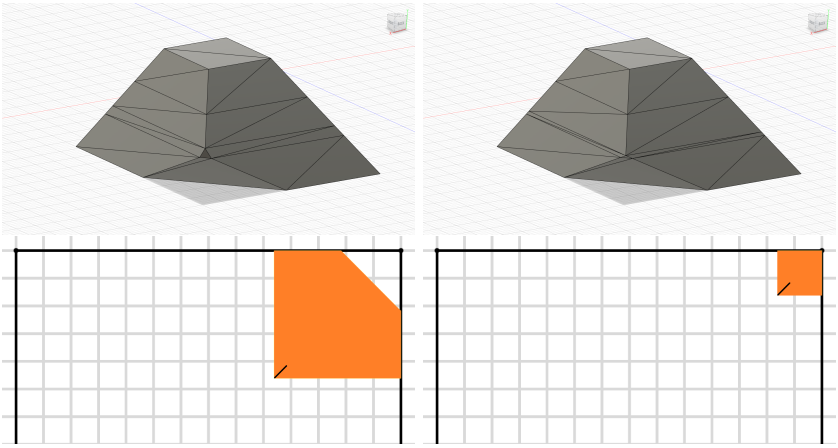

Figure 29: An experiment where the observer walks forward in an internal terrain. On the top left is the generated manifold whereas on the right is the ideal output. The bottom two images show the starting and ending positions respectively.

By viewing the intermediate polygons from the reference frame of the observer, we can see that one is a subset of the other. When the intermediate polygons straddle only a single discontinuity point, one we will always have one polygon be a subset of the other. This, coupled with the concave nature of the polygons, means that any connections between the two will always be contained within the larger polygon. Additionally, the naive approach causes the connections to be shorter or equal to the ideal leading to, in turn, generated faces to differ from the ideal. This, in combination with all the connections being contained within the polygon, leads to generated manifolds being smaller than the ideal. This leads to certain volumes being considered non visible when, in reality, they should be visible.

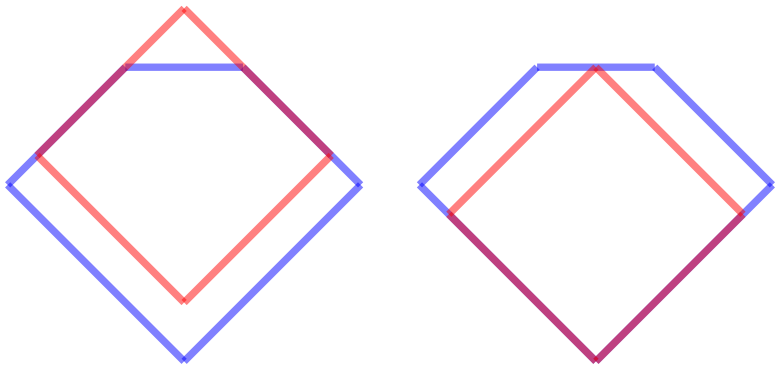

Figure 30: An example of how, in flat or internal terrain, one generated polygon will contain the other. On the left is the static room reference frame and on the right is the observer reference frame. This figure is based on the experiments shown in Figure 29

On the other hand, external terrain has missing volumes due to a different reason. As can be observed in Figure 31, the geometry does not extend high enough into the manifold. This occurs when a shadow node changes which edge it projects onto or alternatively is non existent in the top layer.

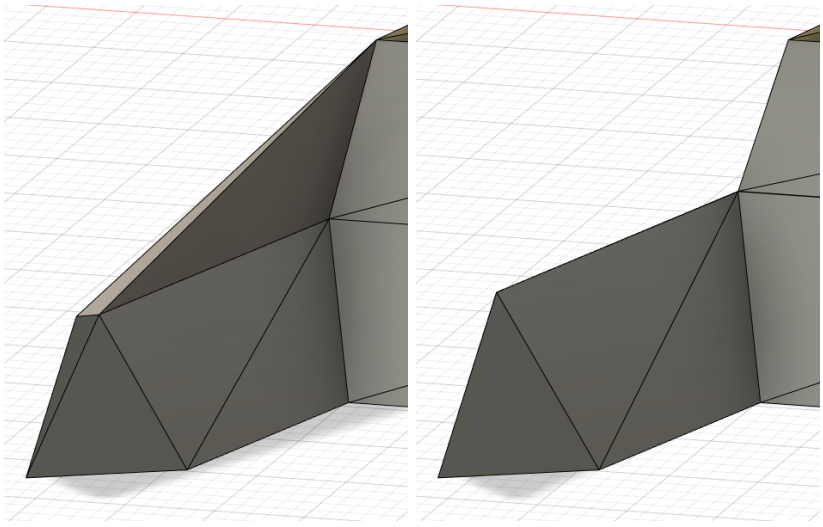

Figure 31: An experiment where the observer moves in an external terrain. On the left is the output manifold and on the right is the ideal manifold.

This sudden snap in either position or presence occurs between the two intermediate layers. This would ideally cause a step in the geometry. The system, however, connects the layers linearly which means that this sudden step is not properly captured, leaving a part of the volume outside of the manifold.

This process also applies if the node is missing from the bottom layer. It simply means that the volume is missing from the bottom rather than the top.

A more intuitive way to interpret this kind of artifact is to view is as approximating a step function using a straight line as in Figure 32. We can see that the line incorrectly excludes the area bellow the step function after x = 1.

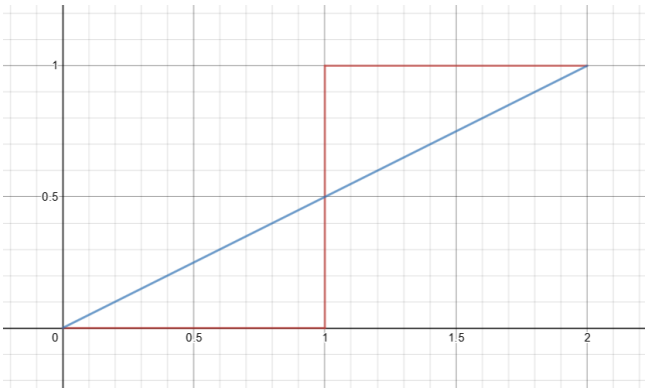

Figure 32: An example of approximating a step function using a straight line.

4.2.2 Additional Volume

When observing external terrain, we notice that parts of the generated geometry extend past the volume determined by the ideal case. This is evident in Figure 33 and is caused by much the same reasons as why volume is missing when working with flat and internal terrains.

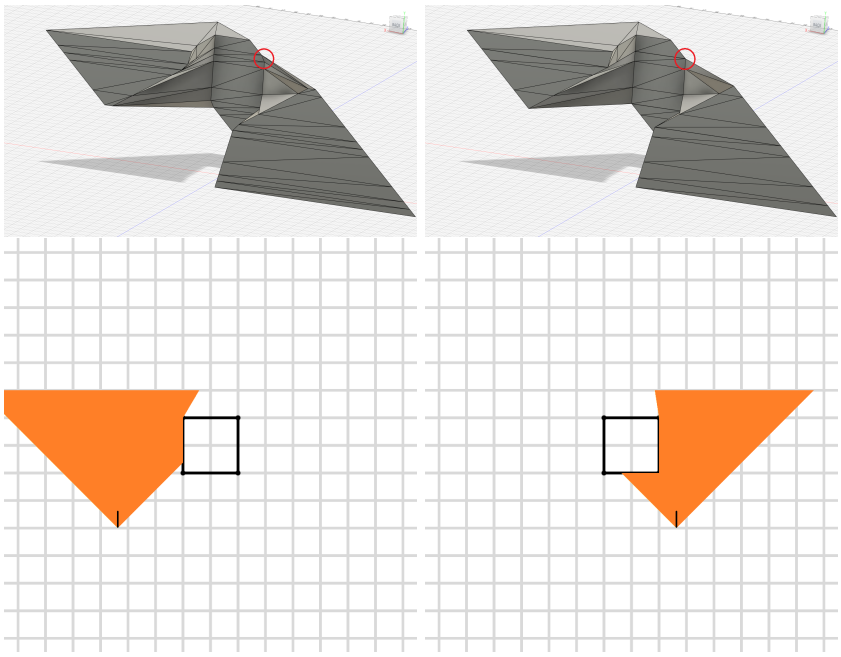

Figure 33: An example of how additional volume is generated from external terrain. On the top left is the generated manifold and on the top right is the idea. Circled in red we can see the extra volume from the approximation. The bottom two images show the beginning and end positions of the observer path respectively.

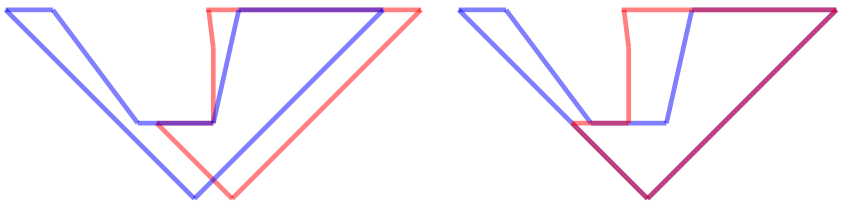

Figure 34: Two polygons generated about the disconitnuity show how neither is a subset of the other and that they are not convex. This can lead to certain connections crossing areas where there is no volume and create false volumes.

The naive connections between layers leads to different faces from the ideal scenario. The difference here, however, is that the visibility polygons are concave. This means that when nodes are removed or added around obstacle nodes, the closest connection may be outside of the polygons, causing volume outside

of the manifold to be included in what we generate.

Additionally, shadow nodes disappearing, appearing or suddenly projecting to a different edge also causes excess volume. Similarly to how the linear connection between intermediate volumes not capturing certain volumes when a step is present, it also captures volumes that are outside of the ideal manifold.

This is analogous to Figure 32 where the area below the line from x = 0 to x = 1 is not included by the step function.

5 Conclusion

Our tool generates visibility manifolds with a high degree of accuracy. When experiencing no discontinuities, the output manifold will match the ideal, neglecting rotational components. Any differences observed in these cases are due to the rotational motion not being captured by linear connections. Additional differences are caused by naive triangulations of non planar quads.

To remedy these issues, smaller step sizes may be used as well as avoiding proximity between the observer and terrain as well as not using overly deep fields of view. These solutions, although help limit the generation of artifacts, rely on user knowledge and caution when employing the tool.

When handling discontinuities, the tool does exhibit certain limitations. The naive closest point connection scheme may lead certain parts of the geometry to be truncated or, alternatively, extra geometry may be added to the resulting shape. This can, however, be limited by the use of appropriate threshold sizes.

Although smaller threshold sizes produce more accurate outputs, they may have a side effect of producing very small or thin triangle which may be unfavorable for certain applications.

6 Future Work

In future work, a system can be built to optimise the generated manifold topology. Indeed, certain flat faces of a generated manifold may be built up of many smaller coplanar triangles. These can be reduced to smaller number of larger triangles which would, in turn, reduce the complexity of the mesh.

Additionally, excessively thin or small triangles can be eliminated to aid in the stability of the mesh when performing calculations. This could potentially introduce certain artifacts or biases, however it could be left up to the user whether or not they wish to perform this kind of post processing.

An additional feature that may be added is the detection of problematic areas in a manifold due to large changes in the mesh over short periods of time. This would allow us to detect when we would need to break down the mesh into smaller units by using smaller step sizes. This could also highlight what areas of the terrain may be problematic which would aid the user in making better decisions when performing analysis.

Additionally, we can use a better strategy for triangulating non planar quads. Thus far we have simply taken these quads and cut them in half to generate two triangles. However, these triangles do not always accurately represent the original quad.

Up until now we have used smaller step sizes to better capture the contour of the changes. Although this does work reasonably well, it generates additional geometry in other parts of the manifold where it is not necessary. It also has the issue of requiring the system to perform additional calculations which could be better spent in other areas.

An alternative would be to triangulate non planar quads into a larger number of triangles. This would provide us with a more accurate representation of the

quad while reducing computation and complexity with regards to our current solution. Additionally, both methods may be combined if a highly detailed triangulation is required. Examples of these triangulations may be seen in Figure 35.

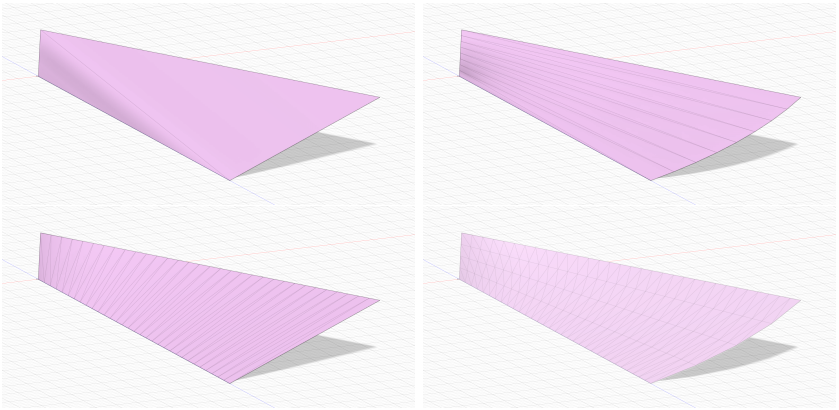

Figure 35: Top left is the original naively triangulated quad followed by the same naive triangulation scheme with smaller step sizes on the top right. On the bottom left we can see what a mesh with additional triangulation could potentially look like and the bottom right shows a possible combination of the two strategies.

References

[Asano 1985] Asano, T. 1985. Efficient algorithms for finding the visibility polygons for a polygonal region with holes. *Transactions of IECE of Japan* E-68:557–559.

[Bajaj, Coyle, and Lin 1996] Bajaj, C. L.; Coyle, E. J.; and Lin, K.-N. 1996. Arbitrary topology shape reconstruction from planar cross sections. *Graph. Models Image Process.* 58(6):524–543.

[Barequet and Sharir 1996] Barequet, G., and Sharir, M. 1996. Piecewise-linear interpolation between polygonal slices. *Computer Vision and Image Understanding* 63(2):251–272.

[Barequet and Vaxman 2007] Barequet, G., and Vaxman, A. 2007. Nonlinear interpolation between slices. *ACM Symposium on Solid and Physical Modeling 2007* 97–107.

[Barequet and Vaxman 2009] Barequet, G., and Vaxman, A. 2009. Reconstruction of multi-label domains from partial planar cross-sections. In *Proceedings of the Symposium on Geometry Processing*, SGP '09, 1327–1337. Goslar, DEU: Eurographics Association.

[Bermano, Vaxman, and Gotsman 2011] Bermano, A.; Vaxman, A.; and Gotsman, C. 2011. Online reconstruction of 3d objects from arbitrary cross-sections. *ACM Trans. Graph.* 30(5).

[Boissonnat and Memari 2007] Boissonnat, J.-D., and Memari, P. 2007. Shape reconstruction from unorganized cross-sections. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, SGP '07, 89–98. Goslar, DEU: Eurographics Association.

[Keppel 1975] Keppel, E. 1975. Approximating complex surfaces by triangulation of contour lines. *IBM J. Res. Dev.* 19(1):2–11.

[Meisters 1975] Meisters, G. H. 1975. Polygons have ears. *The American Mathematical Monthly* 82(6):648–651.

[Meyers, Skinner, and Sloan 1992] Meyers, D.; Skinner, S.; and Sloan, K. 1992. Surfaces from contours. *ACM Trans. Graph.* 11(3):228–258.

[Tremblay 2016] Tremblay, J. 2016. *Computing Techniques for Game Design.* Ph.D. Dissertation, McGill University.

[Zou et al. 2015] Zou, M.; Holloway, M.; Carr, N.; and Ju, T. 2015. Topology-constrained surface reconstruction from cross-sections. *ACM Trans. Graph.* 34(4).