Shahid Beheshti University
Faculty of Computer Science and Engineering

# TECHNICAL REPORT

| | |
|---|---|
| Title: | JAHAN: A Framework for Procedural Map Generation |
| Student Name: | Saeed Amiri-Chimeh |
| Student ID: | 96543003 |
| Supervisor: | Dr. Hassan Haghighi - Dr. Mojtaba Vahidi-Asl |
| Report ID: | CSE-GLB-0264-1433 |
| Last Update: | February 13, 2022 |

## 1 Introduction

This document reports on a framework for procedural map generation called `JAHAN`. This framework organizes map generation tasks in a pipeline of processes. Each process handles an aspect of map generation and relies on several modules, each implementing a procedural technique. Map designers can configure these modules and specify desired properties to generate game maps.

Moreover, this framework facilitates automatic map evaluation by allowing designers to declare assertions about the maps. `JAHAN` is capable of checking these assertions against generated maps automatically.

In the next section, we present the architecture of `JAHAN` and provide a running example. Next, Section 3 presents two case studies to demonstrate capabilities of `JAHAN` in generating different types of maps.

# 2 Architecture

This section presents processes and modules that constitute the JAHAN framework. In addition, we introduce essential algorithms and data structures used in the design of this framework. We first provide an overview of the whole framework and then elaborate its processes in the following subsections. For every process, we explain its flow and involved modules. Lastly, we talk about the assertion checking features of JAHAN at the end of this section.

## 2.1 Overview

The workflow of JAHAN follows a pipeline architecture made of seven processes. These processes are Canvas Generation, Area Layout Processing, Area Polygon Generation, Influence Map Generation, Heightmap Generation, Landscape Generation, and Marker Placement, where each process relies on the implementation of particular modules.

The canvas generation process provides the primary structure of the map. Its principal function is to partition an empty slate into a set of cells called canvas. Next, JAHAN engages with area layout processing. To this end, it takes a graph as input, where each vertex represents an area in the map and each edge connects two adjacent areas. In JAHAN, an area can be any significant part of a map. For example, jungles, deserts, mountains, and seas can be considered areas of a map in an open-world RPG game. For FPS games, corridors and fight arenas are the main parts of the map. Also, construction and resource sites are among the central areas of an RTS map.

The input graph of the area layout processing must be planar so we can embed it on a plane. After finding a planar embedding of the given graph on the slate, we create a skeleton for every area. An area skeleton is a spatial structure that abstracts the geometry of an area on the slate.

Next, the polygon generation process takes the canvas and skeletons as inputs and assigns no or one area to every cell in the canvas. At this stage, the overall geometry of all areas and their positioning on the slate emerges. We must find the nearest area skeleton for every canvas cell to generate area polygons. Here, the concept of distance is fundamental. In other words, using different distance functions leads to different geometries in the map.

After generating area polygons, we rasterize the map and determine the influence of every area on each pixel. For every area, we store this data in an area influence map. It is worth noting that the influence of an area polygon gradually fades as we move away from its borders, which creates a smooth transition between area geometries. The designer should provide a height profile for every area, given the influence maps. Height profiles determine the overall shape of the elevation and its detail pattern. The heightmap generation process uses the provided data to assign a height value to every map pixel.

Then, the landscape generation process creates sets of data for the surface and items of

the map. Designers can define different landscape profiles with different surface types and item settings. Also, they can determine how the landscape generation process should use these settings for every area. For example, they can assign a single landscape profile to a whole area or ask for a blend between specific profiles based on the height differences in an area.

Finally, the designers specify their desired properties of user-defined markers, and the marker placement process automatically finds the most fitting location for every marker specification. Figure 1 depicts the mentioned pipeline and its dependencies on different modules. In the next subsections, we talk about the presented modules in details.

## 2.2 Canvas Generation

This process generates the initial structure of the map in the form of a canvas. In short, a canvas is a partitioning of an empty slate into a set of cells. We later use these cells when generating different polygons for map areas. Canvas generation depends on the seed generator module, which generates coordinates called seeds on the slate. We use the generated seeds to create a Voronoi diagram [1, 9] that gives us the cells.

Different relative positioning of these seeds leads to various shapes of cells. Therefore, different implementations of the seed generator module can create canvases with distinct visual characteristics. For example, a canvas can be a grid of squares, hexagons, a set of radially positioned cells, or a set of cells with different degrees of randomness. Accordingly, providing designers with means to determine their desired seed pattern increases their control over the visual characteristics of the generated maps.

The final step of the canvas generation process is finding adjacencies between the cells. We use Delaunay triangulation [2] to find the neighbors of every cell.

Figure 2 shows the modules and sub-processes of canvas generation.

### 2.2.1 Seed Generator Module

This module outputs coordinates in a $[0, 1] \times [0, 1]$ plane. There are many possible implementations for this module. Currently, `JAHAN` provides the following five implementations:

- **Square Seed Generator:** This implementation generates seeds so that all cells will be squares of similar sizes.
- **Hexagon Seed Generator:** This implementation generates seeds so that all cells will be regular hexagons of similar sizes.
- **Radial Seed Generator:** This implementation generates seeds so that the resulting cells make circular patterns.

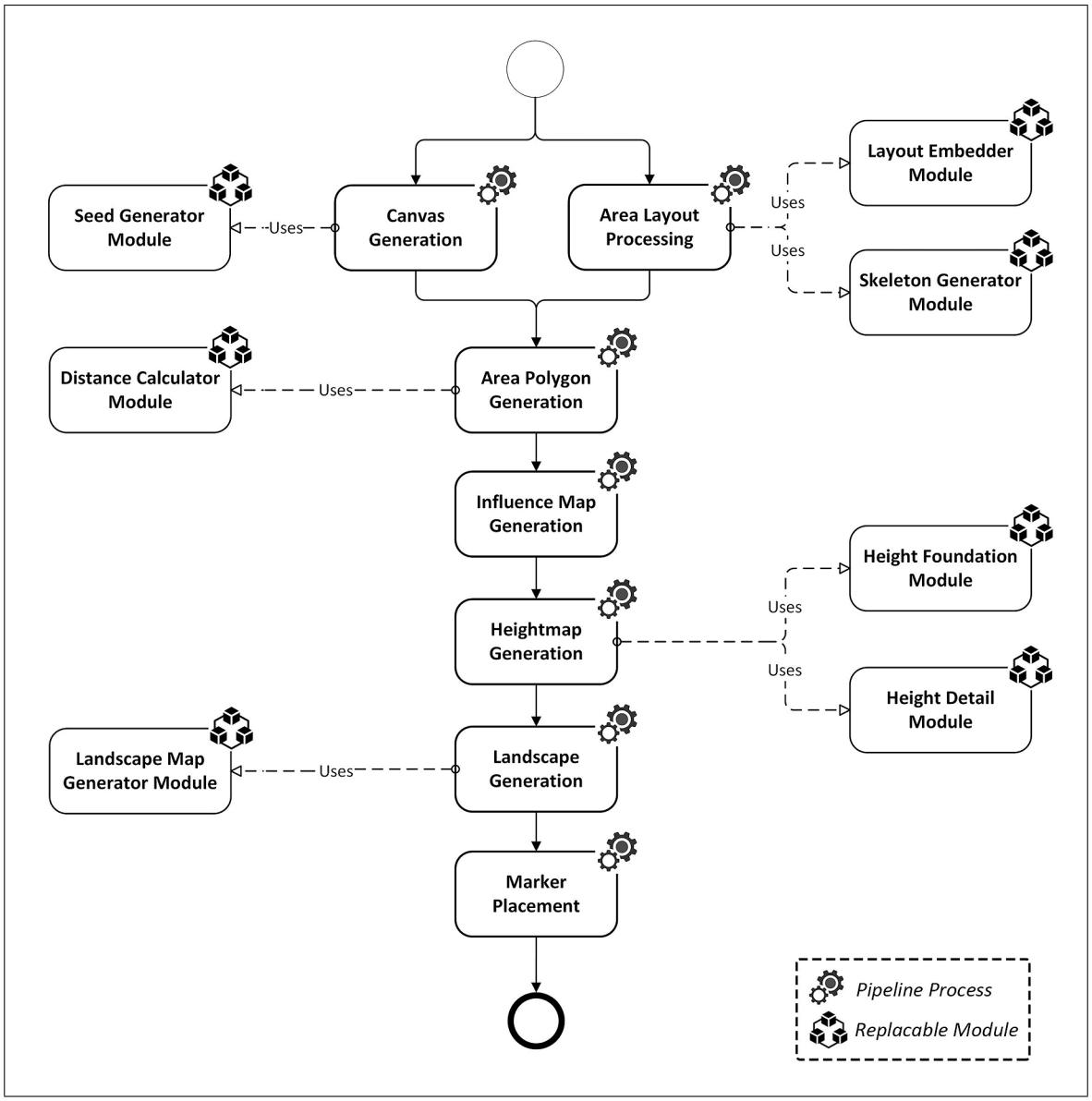



Figure 1: Processes and modules of the JAHAN's pipeline

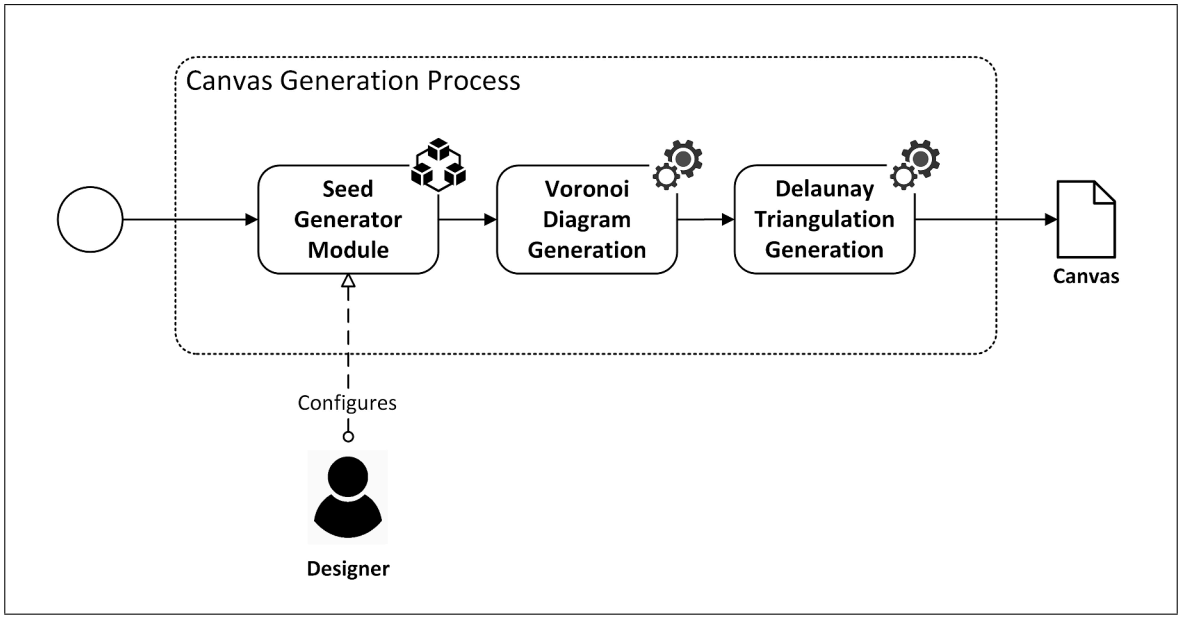



Figure 2: Sub-processes and modules of the canvas generation process

- **Loosed Square Seed Generator:** This implementation is similar to the square generator, but the seeds are randomly offset a little to break the strict symmetry of square cells.
- **Uniform Random Seed Generator:** This implementation randomly positions seeds on the slate. The result of this generator is a set of irregular hexagonal cells.

Figure 3 shows five different canvases generated by different seed generators.

## 2.3 Area Layout Processing

We define areas as single planar polygons on the slate to achieve generality and abstraction. Moreover, we say two areas are neighbors if they share a segment. These polygons create a planar graph on the slate with areas as its faces. We can model areas and neighborhoods as the dual of such a graph. In other words, we model the map layout as a graph where every node specifies an area and edges determine if any two areas share a border. Hereafter, we call such a graph the Area Layout Graph of the map. Area layout graphs are always planar since they are duals of planar graphs by definition.

Accordingly, area layout processing aims to take an area layout graph from the designer and provide essential spatial data for generating actual area polygons further in the pipeline. We start by checking the planarity of the given graph. Then, the layout embedder module generates a planar embedding of this graph. A planar embedding of a graph is an assignment

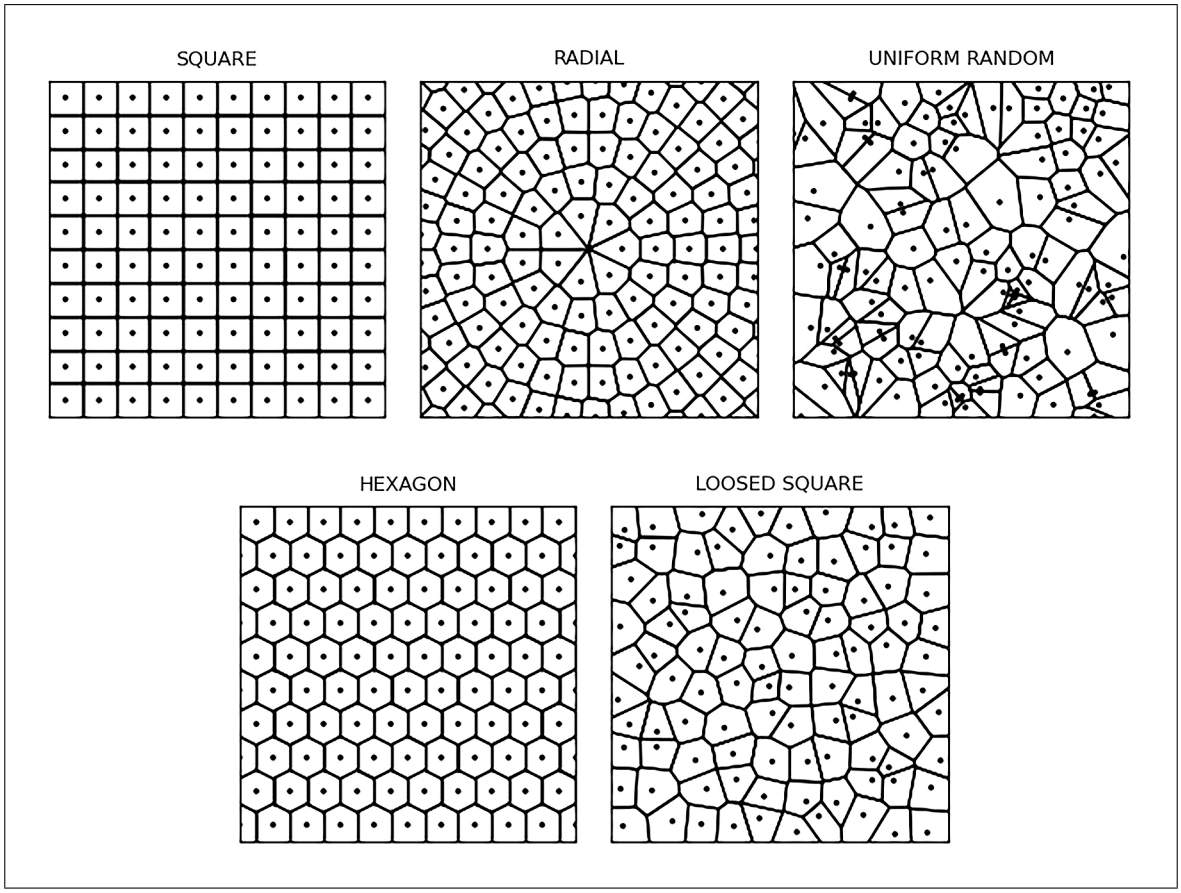



Figure 3: Results of canvas generation process using different seed generator modules

of 2D coordinates to its nodes and drawing edges so that the edges only intersect at their endpoints [13].

Finally, the generated planar embedding is fed into the skeleton generator module in order to create an area skeleton for every area. An area skeleton is a set of connected line segments on the slate, which abstract the extent of area polygons. We use these area skeletons in the following pipeline stages to generate actual polygons for areas. Figure 4 summarizes the flow of area layout processing.

### 2.3.1 Layout Embedder Module

`JAHAN` currently provides one implementation for the layout embedder module, mixing two graph embedding algorithms to obtain eye-pleasing planar embeddings for any given area layout graph. This implementation initially applies the linear-time embedding algorithm presented by Chrobak and Payne [3]. This algorithm assigns a coordinate to every node of the input graph while assuming edges to be straight, non-intersecting line segments. This

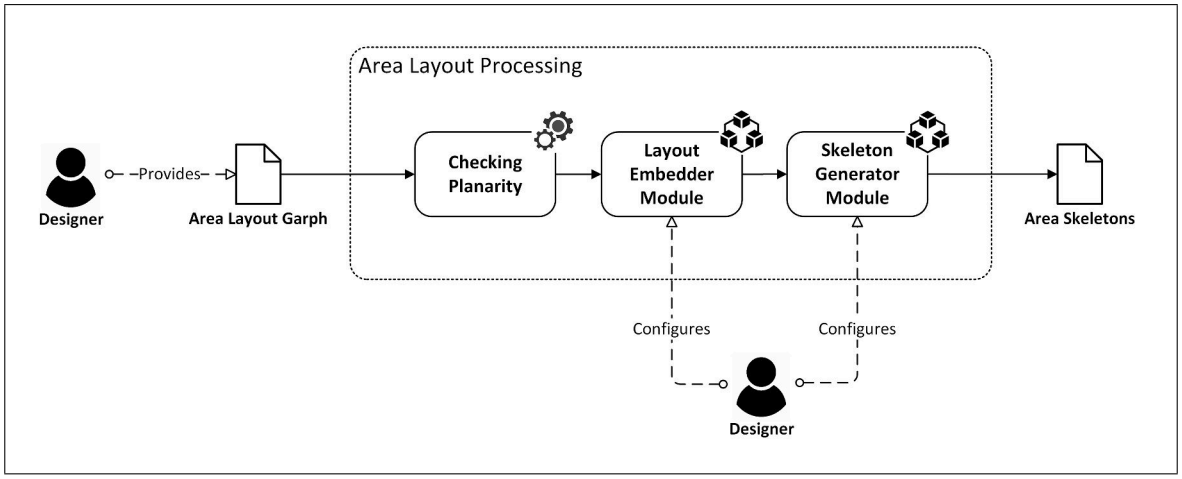



Figure 4: Sub-processes and modules of area layout processing

algorithm rapidly provides a planar embedding. However, it produces drawings resembling triangular shapes with a high variance in edge length which are not visually attractive.

To resolve the above issue, we take the resulting embedding and feed it as the starting embedding to the Fruchterman–Reingold algorithm [6], a force-directed graph drawing technique. Such algorithms draw graphs in a visually pleasing way. They are physical simulations and try to iteratively improve the drawing so that all the edges are of more or less equal length with as few crossing edges as possible [5].

Starting a force-directed graph drawing algorithm with an already planar embedding significantly increases the probability of preserving planarity; however, this algorithm does not formally guarantee the planarity in every iteration because of its simulative design. Therefore, we check the planarity of the final embedding and restart the process with a slightly different initial embedding if the current result is not a planar drawing of the given area layout graph.

Figure 5 shows how the described implementation works for an example area layout graph.

Throughout the rest of this section, we use the layout graph depicted in Figure 5 to generate different aspects of a map that resembles the open-world environment of RPG games. We chose this type of map to explain the capabilities of `JAHAN` because such maps involve considering different aspects of map generation, including the map's geometry, height data, landscape, and markers.

### 2.3.2 Skeleton Generator Module

This module takes an embedding and processes it to create a skeleton for every given layout area. Implementing this module depends on the characteristics of the provided embedding. We previously assumed that edges would be straight-line segments when presenting our

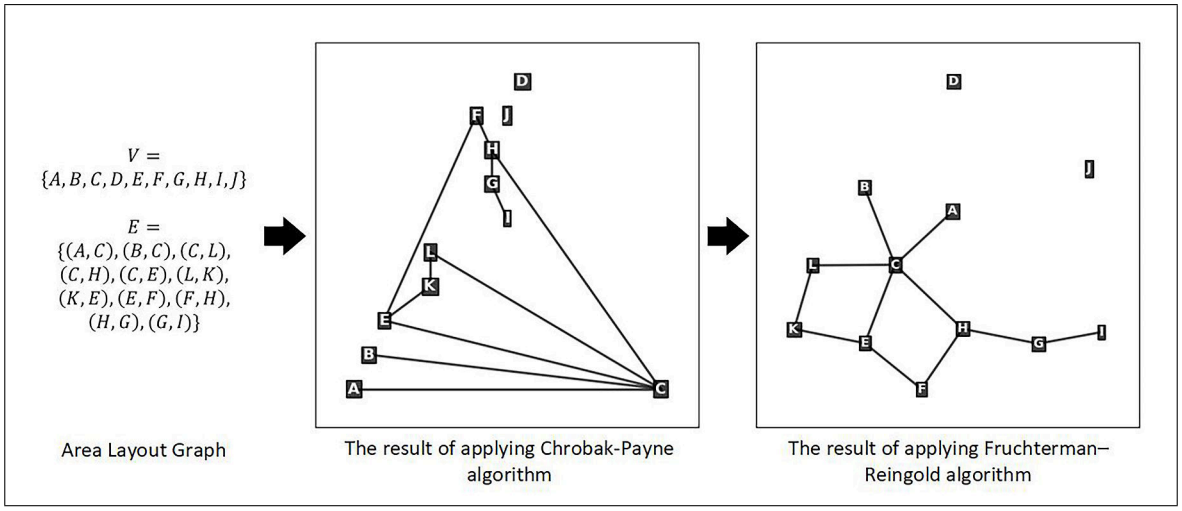



Figure 5: Planar embedding of an example area layout graph

implementation of the layout embedder module. Based on this assumption, JAHAN currently provides just one implementation for the skeleton generator module.

This implementation pivots around breaking every edge into two segments we call arms. Then, we return the skeleton of every area as the set of all arms connected to its corresponding node. The implementation also takes a stretch weight for every area from the designer to determine the breaking point of edges. For example, if two area nodes share an edge and the stretch weight of the first area is twice as big as the stretch weight of the second one, the algorithm breaks the edge precisely at two-thirds of the edge length from the first area node. Subsequently, stretch weights provide designers with the means to control the overall extension of the areas.

Algorithm 1 presents a pseudo-code for this implementation. In this algorithm, we iterate over the edges of the given layout graph (line 2). In every iteration, we acquire the coordinates of the edge vertices and their stretch weight (lines 3 to 6). Then, we calculate the breaking coordinates by applying a weighted average on the coordinates of the edge vertices (line 7). Finally, we create a line segment from every vertex to the breaking point and add it to its corresponding skeleton (lines 8 and 9).

Figure 6 shows the result of this algorithm if we feed it with the final embedding shown in Figure 5 and set all of the stretch weights to identical values.

## 2.4 Area Polygon Generation

This process takes a canvas and a set of area skeletons as inputs. It first labels canvas cells with area names while preserving the required adjacency between the areas. Next, a polygon is created for every area by merging cells that have similar area labels.

**Algorithm 1** Generating area skeletons for an area layout graph based on an embedding that assumes edges are drawn as straight-line segments.

**Inputs:**

- $G$: An area layout graph
- $Embedding$: Assigns a 2D coordinate to every area
- $StretchWeights$: A mapping that assigns a scalar to every area

**Result:**

- $S$: A mapping that assigns a set of line segments to every area

$S \leftarrow \emptyset$
**for** each edge $(v_1, v_2)$ in $G$ **do**
    $pos_1 \leftarrow Embedding[v_1]$
    $pos_2 \leftarrow Embedding[v_2]$
    $w_1 \leftarrow StretchWeights[v_1]$
    $w_2 \leftarrow StretchWeights[v_2]$
    $break \leftarrow pos_1 * (w_1/(w_1 + w_2)) + pos_2 * (w_2/(w_1 + w_2))$
    $S[v_1] \leftarrow S[v_1] \cup (v_1, break)$
    $S[v_2] \leftarrow S[v_2] \cup (v_2, break)$
**return** $S$

The process starts by calculating the distance of each cell to every area skeleton and labeling the corresponding cell with the area which has the closest skeleton. An essential aspect of this step is the definition of the distance function. It is worth mentioning that different distance functions result in area polygons with unique characteristics. Accordingly, this process relies on the distance calculator module when labeling the canvas cells. This module implements a function for calculating distances between 2D points. `JAHAN` provides the following implementations for this module.

- **Euclidean Distance Calculator:** This implementation calculates the distance between two points as the length of the straight line that connects them.
- **Manhattan Distance Calculator:** This implementation calculates the distance between two points as the sum of their horizontal and vertical distances.
- **Chebyshev Distance Calculator:** This implementation calculates the distance between two points as the maximum value between horizontal and vertical distances.

The customizability of this module provides the designers with the means to control the visual characteristics of the map geometries. Moreover, the designers can determine the overall size of areas by assigning a bound radius to every area. If the distance of a cell to its

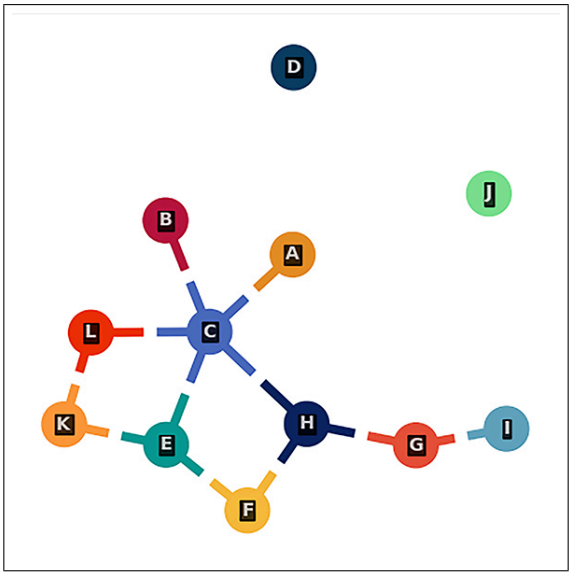



Figure 6: Area Skeletons

nearest skeleton is larger than the bound radius of that skeleton's corresponding area, then the cell is not labeled at all. In other words, the value of the bound radius acts as a limit for labeling cells that are too far from their nearest skeleton.

Finally, a post-processing routine cleans labels that form forbidden neighborhoods. By forbidden, we mean neighborhoods between areas that do not share an edge in the area layout graph. This process removes labels from cells that create such neighborhoods to ensure consistency with the edges of the area layout graph. Also, it removes labels from every cell closer to map borders than any area skeleton. We group the unlabeled cells under a predefined area called the empty area.

Now, we can create polygons that determine the geometry of areas on the map by merging cells with the same label (except the empty area). Figure 7 summarizes the polygon generation process.

Considering infinite bound radiuses, Figure 8 shows the result of are polygon generation with different distance calculator modules on a loosed square canvas for the skeletons presented in Figure 6. Furthermore, Figure 9 shows the effect of different stretch weights and bound radiuses on the resulting area polygon. As can be seen in this Figure, it is possible to generate polygons that resemble open arenas and corridors by tuning a few parameters. In other words, `JAHAN` allows designers to specify various geometric layouts with a high level of control.

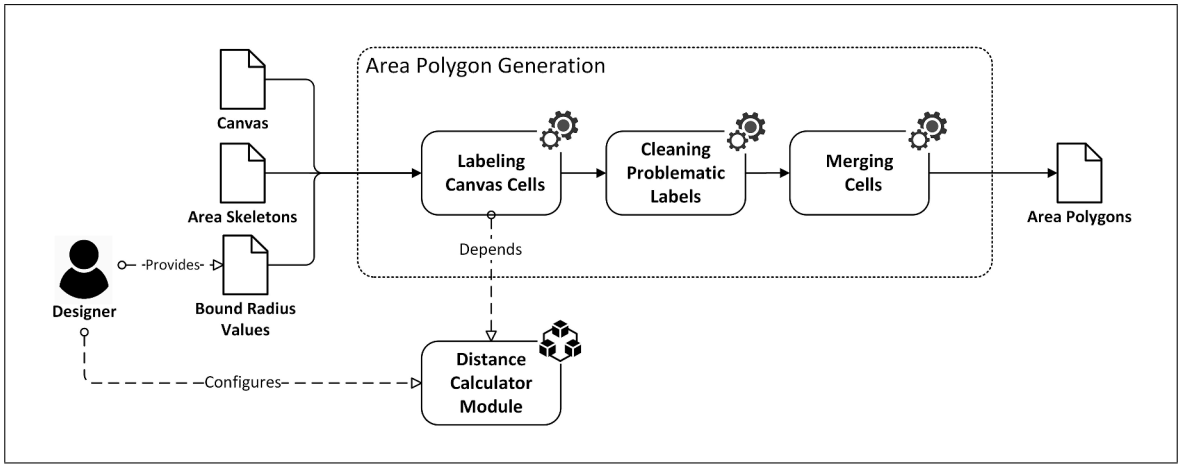



Figure 7: Sub-processes and modules of area polygon generation

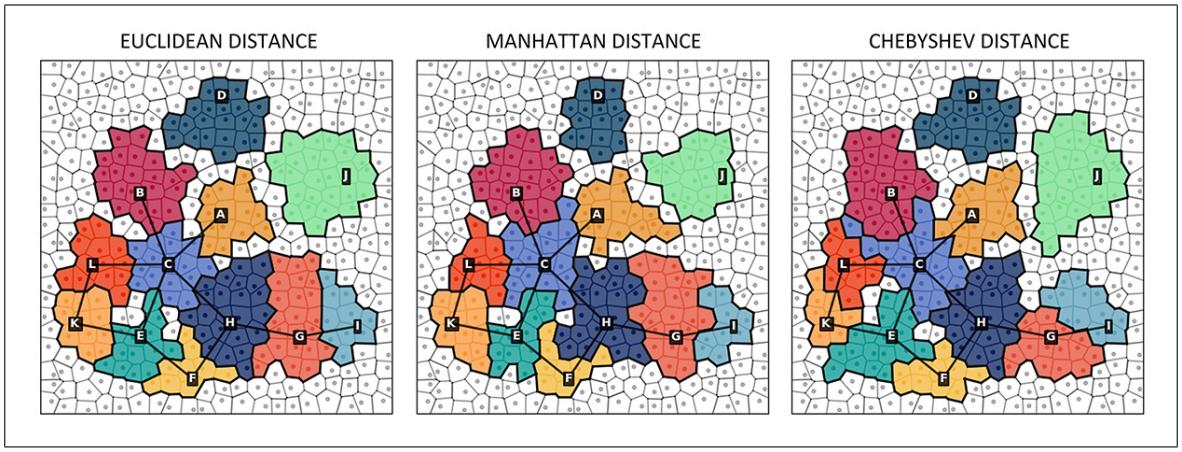



Figure 8: Generating area polygons using different distance functions

## 2.5 Influence Map Generation

This process takes the area polygons as input and rasterizes the slate. Then, it calculates the influence of every area polygon on each pixel. For every area, we store this data in an influence map, in which a scalar value from $[0, 1]$ is assigned to every map pixel. If an influence map returns 0 for a pixel, its corresponding area does not influence that pixel at all. Conversely, for the return value 1, the corresponding area has total influence over the pixel.

We later use influence maps to combine isolated information of every area when generating height and landscape for the map. Therefore, a critical property of a good influence map is having a smooth fade of influence near the borders of its corresponding polygon. This property ensures a smooth transition between area heights and landscapes. In addition, influence maps of areas of a layout graph must complement each other. In other words, for

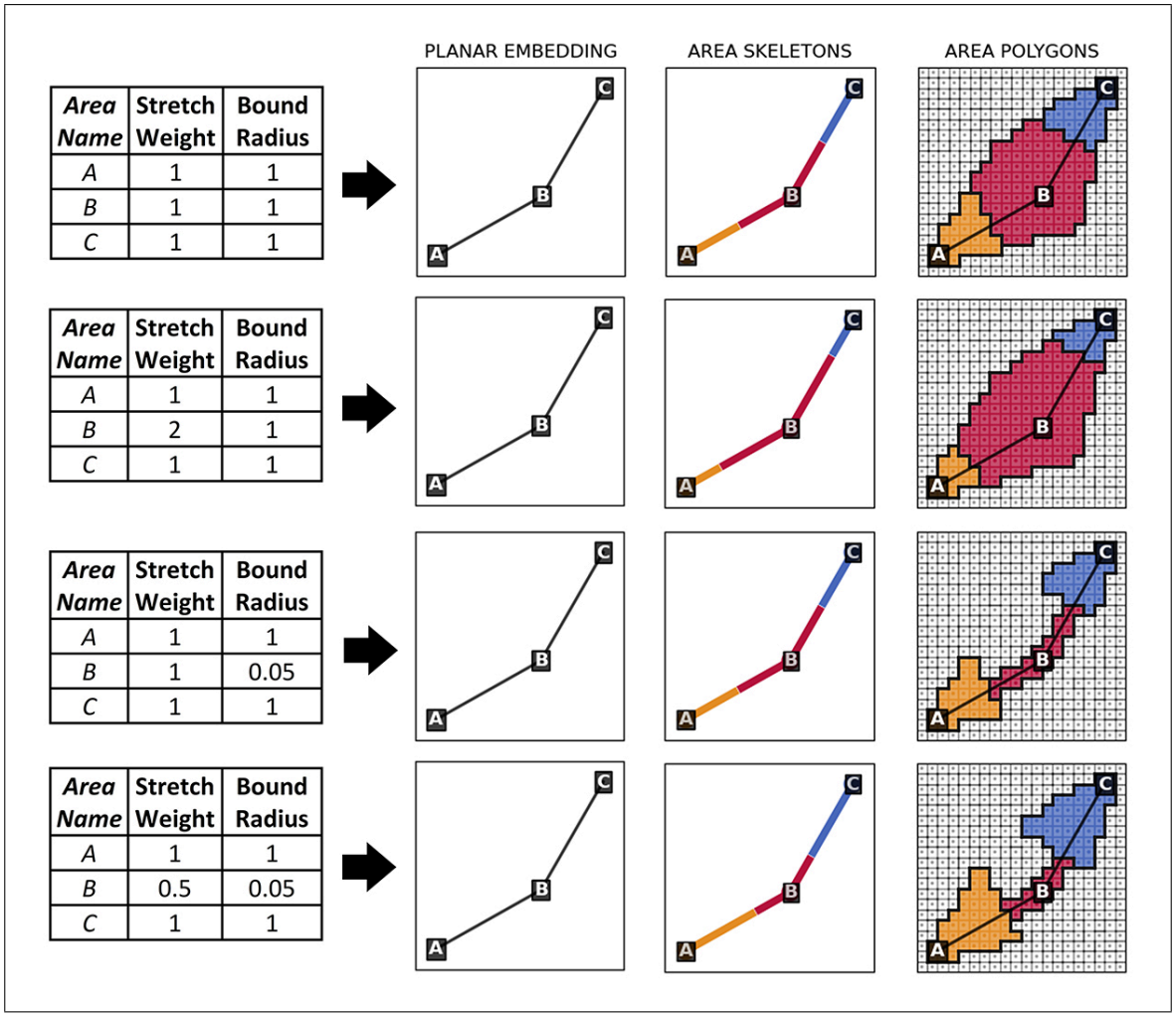



Figure 9: Controlling area polygon generation using stretch weight and bound radius parameters

every pixel, the sum of all influences must be equal to 1. This property ensures that the other areas will cover the lack of influence from one area for every pixel.

In Algorithm 2, we present a pseudo-code that takes pixel coordinates, a set of area polygons, and a fading radius as inputs and generates a set of influence maps. This algorithm iterates over polygons and pixels (lines 2 and 3). We calculate the pixel's distance to the polygon in each iteration if the pixel is not in the empty area (line 5); otherwise, we compute the pixel's distance to the nearest polygon (line 7). Then, we use this distance to calculate the linear influence of the polygon on the pixel (lines 8 to 11).

These calculations give us an influence equal to 1 (0) for pixels inside (outside) the polygon whose distance to the polygon's border is greater than or equal to the fade radius. We create a linear fade from the inner side of the polygon to its outer side for the pixels whose distance

**Algorithm 2** Generating influence maps from a set of area polygons.

**Inputs:**

- $Polygons$: a mapping that assigns a polygon to every area
- $Pixels$: coordinates of map pixels
- $Fade$: the radius of fading effect near polygon borders

**Result:**

- $InfMaps$: A mapping that assigns an influence map to every area

$InfMaps \leftarrow \emptyset$
**for** each $(area, poly)$ in $Polygons$ **do**
    **for** each $p$ in $Pixels$ **do**
        **if** $p$ is not inside the empty area **then**
            $d \leftarrow$ distance of $p$ to $poly$
        **else**
            $d \leftarrow$ distance of $p$ to the nearest polygon
        **if** $p$ is inside $poly$ **then**
            $linInf \leftarrow 0.5 + 0.5 * min(d, Fade)/Fade$
        **else**
            $linInf \leftarrow 0.5 - 0.5 * min(d, Fade)/Fade$
        **if** $linInf > 0.5$ **then**
            $infMaps[area][p] \leftarrow 2 * linInf^2$
        **else**
            $infMaps[area][p] \leftarrow 1 - ((-2 * linInf + 2)^2)/2$
**for** each $p$ in $Pixels$ **do**
    $infSum \leftarrow 0$
    **for** each $(area, poly)$ in $Polygons$ **do**
        $infSum \leftarrow infSum + infMaps[area][p]$
    **for** each $(area, poly)$ in $Polygons$ **do**
        $infMaps[area][p] \leftarrow infMaps[area][p]/infSum$
**return** $infMaps$

is less than the fade radius. These computations result in an influence equal to 0.5 precisely at the polygon's border. We apply an easing function on the calculated linear influence to create a smooth fade near the borders of the polygon (lines 12 to 15). Finally, we normalize the calculated influences across all maps and return them (lines 16 to 22).

Consider the area polygons generated in Figure 8 using Euclidean distance. If we run Algorithm 2 for these polygons on a $200 \times 200$ grid of pixels with the fade radius set to 0.05, we get 12 influence maps. Figure 10 blends these influence maps by color to depict them in a single frame.

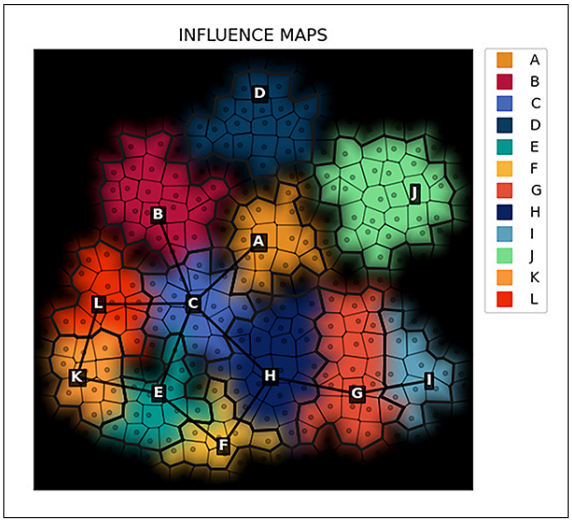



Figure 10: A set of influence maps combined into a single frame

## 2.6 Heightmap Generation

The main task of this process is to generate a heightmap from designer specifications. Here, a heightmap is a simple data structure that stores the height of every pixel of a map. The primary input of this process is the height setting that the designers provide. This setting is a mapping that assigns every area to a height profile. A height profile relies on height foundation and height detail modules. The height foundation module controls the overall shape of an area's elevation, where the height detail module provides a 2D noise function to refine the foundation values by adding more natural details. In addition to the given height setting, this process uses area polygons and influence maps generated by the previous processes.

The heightmap generation process first constructs a partial heightmap for every area and its corresponding height profile. To this end, this process depends on instances of the height foundation module and the height detail module, which both return height data for every pixel of the map. By adding the values returned by these two modules for every pixel, we

generate the partial map of the corresponding area. Afterward, we use the provided influence maps and combine the partial heightmaps to generate a complete one for the whole map. Figure 11 shows the process of heightmap generation and its different elements.

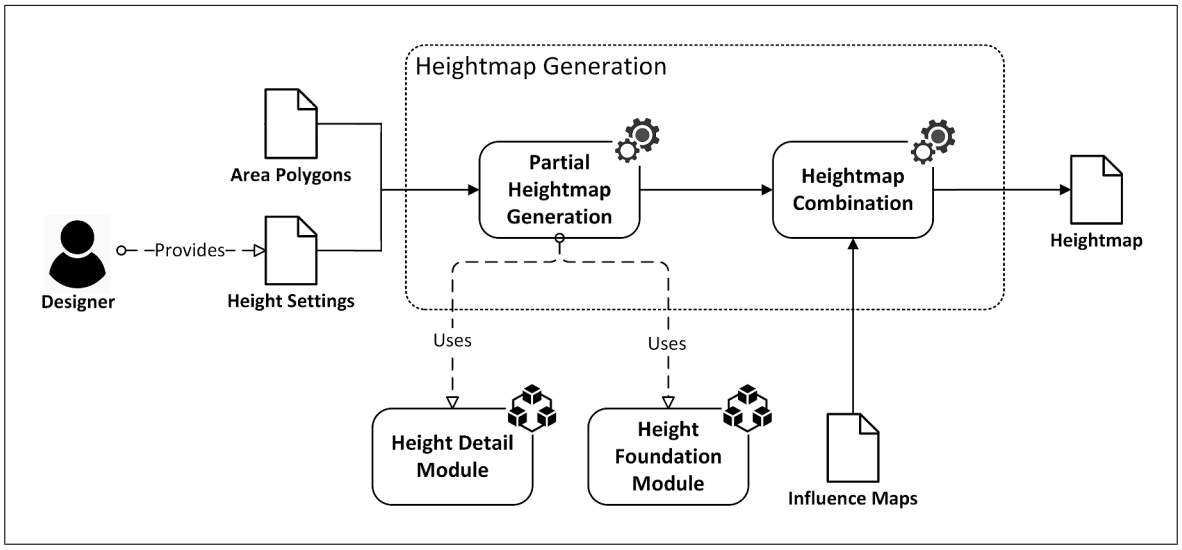



Figure 11: Sub-processes and modules of heightmap generation

### 2.6.1 Height Foundation Module

The height foundation module takes three inputs from the designer and returns a heightmap that determines an area's elevation shape. The first input determines if the foundation is ascending or descending. An ascending (descending) foundation raises (decreases) the height as the pixels get closer to the center of an area's polygon. The second and the third inputs determine the lower and upper bounds of the height values, respectively. Currently, `JAHAN` provides the following four implementations for this module.

- **Flat height foundation:** designers can use this implementation to create flat area foundations where height equals a given value all over the polygon.
- **Bell height foundation:** this implementation provides a smooth foundation with approximately zero gradients near the borders of the polygon and the polygon's center-of-mass (the average of the polygon's vertices).
- **Cone height foundation:** this implementation linearly increases or decreases the height based on the distance to the polygon's center-of-mass.
- **SDF height foundation:** this implementation creates a foundation using signed distance fields [10].

Figure 12 shows three ascending foundations generated by these implementations for a sample area polygon.

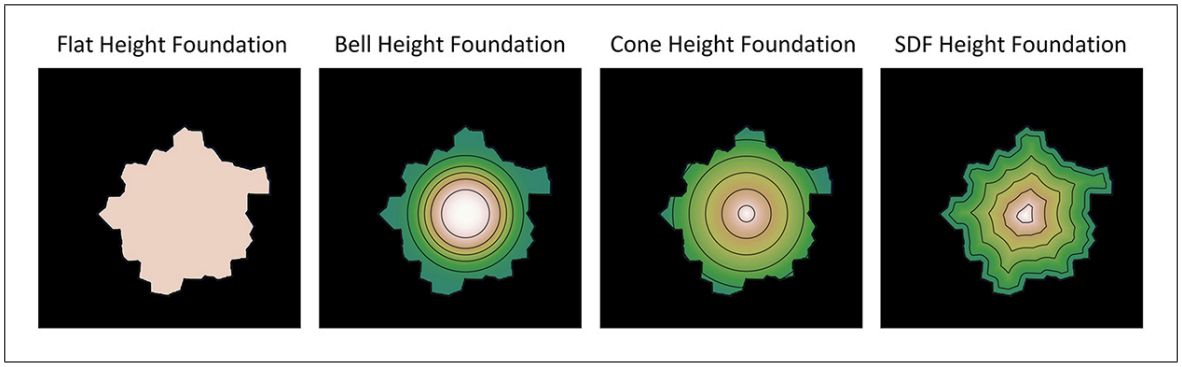



Figure 12: Four different height foundations for the same area polygon

### 2.6.2 Height Detail Module

The height detail module returns a heightmap that determines an area's detail pattern. In a nutshell, this module generates a 2D fractal noise that adds more naturality to the foundations. This module takes three inputs from the designers. The first input controls the amplitude of the noise function. The second one determines the number of octaves of the fractal noise, and the last input controls the scale of the noise. Currently, `JAHAN` provides Perlin [4], Open-Simplex [8], and Worley [14] noise functions as implementations of this module. Figure 13 shows the effect of these noises when added to the foundations presented in Figure 12. In this figure, we have excluded the flat foundation because the result of adding a noise function to a flat surface is identical to the applied noise.

### 2.6.3 Heightmap Combination

After generating partial heightmaps for every area, we have to combine them to create a complete heightmap. Assuming *areas* to be the set of all areas of the map, $infMaps$ to be a mapping that associates areas to their corresponding influence map, and *heights* to be a mapping that maps areas to partial heightmaps, we use Formula 1 to calculate the height of every pixel in the map.

$$finalHeight[p] = \sum_{a \in areas} heights[area][p] * infMaps[a][p] \tag{1}$$

Figure 14 shows a height setting for the areas in Figure 10 and also depicts the result of applying the heightmap generation process to the influence maps illustrated in that Figure.

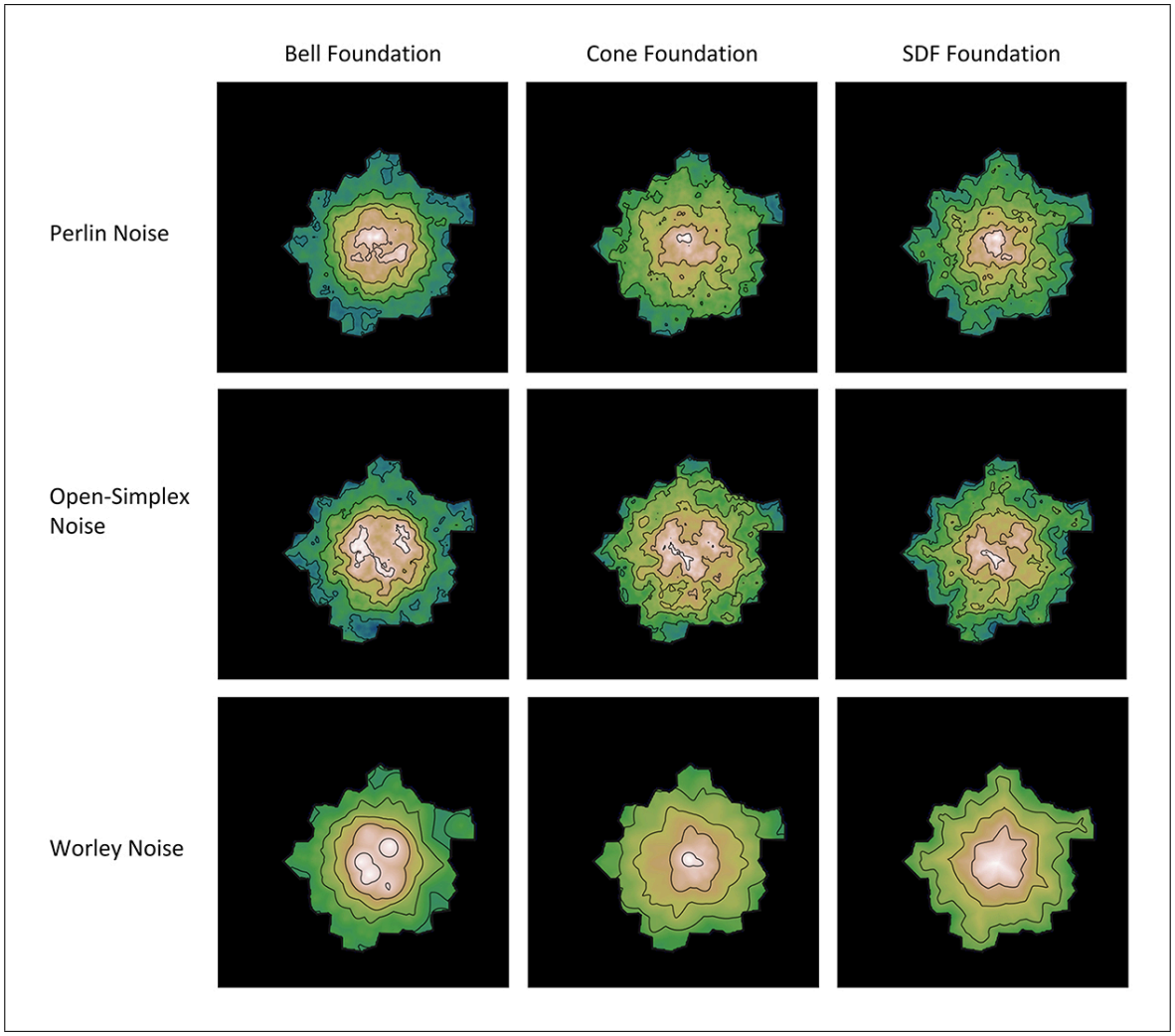



Figure 13: The result of adding different noises to different height foundations

This Figure shows that the height pattern smoothly changes from one area polygon to another because of the fading property of the influence maps.

## 2.7 Landscape Generation

This process is in charge of two tasks. The first task is determining the material of the map's terrain. The other involves scattering user-defined items across the map. To this end, this process relies on the notion of landscape profile. A landscape profile consists of the following elements:

- **Surface type:** The material of the map's surface (e.g., snow, mud, sand).

Height Setting

| Area Name | Foundation | | | | Detail | | | |
|---|---|---|---|---|---|---|---|---|
| | Type | Ascending | minH | maxH | Type | Amp | Octave | Scale |
| A | Flat | - | 7.5 | - | Perlin | 4 | 4 | 16 |
| B | Flat | - | 15 | - | Perlin | 4 | 4 | 16 |
| C | SDF | True | 10 | 30 | Perlin | 4 | 2 | 16 |
| D | Bell | False | 7.5 | 20 | Perlin | 4 | 4 | 16 |
| E | Flat | - | 7.5 | - | Perlin | 4 | 2 | 16 |
| F | Flat | - | 15 | - | Perlin | 4 | 4 | 16 |
| G | SDF | True | 10 | 30 | Perlin | 4 | 4 | 16 |
| H | Bell | False | 7.5 | 20 | Perlin | 4 | 4 | 16 |
| I | Bell | False | 7.5 | 20 | Perlin | 4 | 4 | 16 |
| J | SDF | True | 10 | 30 | Perlin | 4 | 2 | 16 |
| K | SDF | True | 10 | 30 | Perlin | 4 | 2 | 16 |
| L | Bell | False | 7.5 | 20 | Perlin | 4 | 4 | 16 |

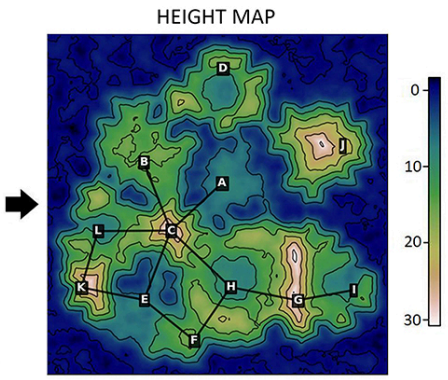



Figure 14: A complete heightmap after combining the partial ones

- **Item density:** A scalar that determines the density of scattered items.
- **Item Weights:** A mapping that determines the relative abundance of every item type.

Map designers can define any number of landscape profiles. Then, they must choose an implementation of the landscape map generator module for every area in order to determine how they wish to use the defined profiles when this process generates landscape data. For example, an implementation of this module might simply assign a single profile to an area while a more sophisticated implementation might blend different profiles based on the height data.

In summary, the landscape generation process takes an instance of the landscape map generator module along with its required landscape profiles for every area. In addition, it relies on the previously generated influence maps and heightmap. Subsequently, this process creates the following artifacts:

- **Surface Maps:** A surface map is created for every surface type. Such a map determines the effect of its corresponding surface type on pixels of the final map. For example, a pixel might be 0.3 snowy and 0.7 muddy. Technical artists can later use these data to visualize the map's surface.
- **Item Locations:** Items are scattered based on the specified item densities and weights across the map.

Figure 15 presents different steps of this process. The following sections provide details for each step.

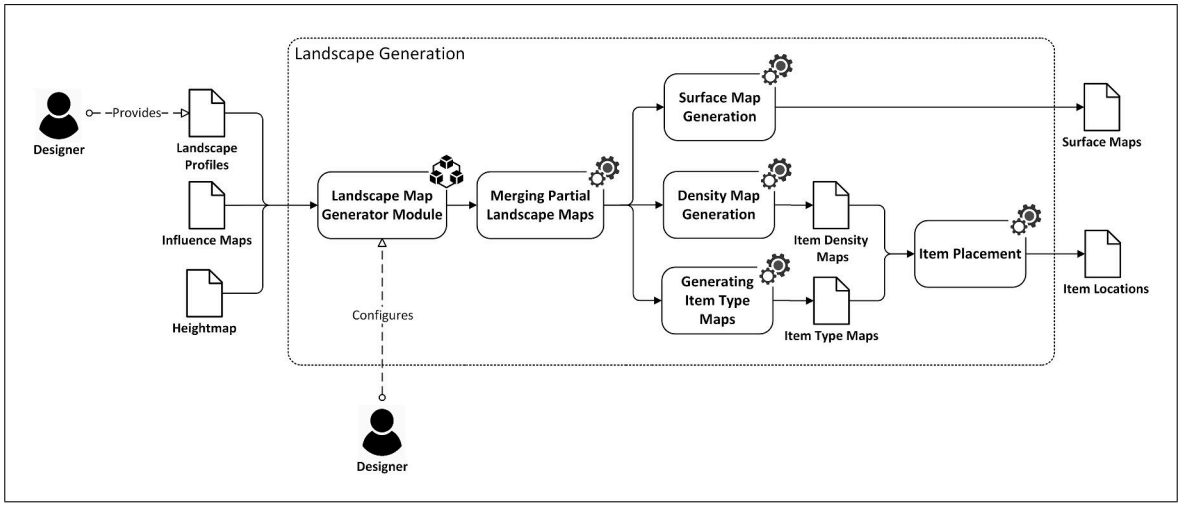



Figure 15: Sub-processes and modules of landscape generation

### 2.7.1 Landscape Map Generator Module

For an area, an instance of this module gets given landscape profiles, the influence map of the area, and the heightmap and generates a landscape map for every landscape profile. Structurally, a landscape map is very similar to an influence map. However, landscape maps determine the influence of landscape profiles over pixels instead of determining the influence of areas.

For every area, the designer must specify an instance of this module. Then, each instance generates partial landscape maps for its corresponding area. Afterward, we merge all partial landscape maps of every profile to create a complete landscape map.

Currently, `JAHAN` provides the following two implementations for the landscape generator module:

- **Single-Profile Landscape:** This implementation assigns a single landscape profile to an area. For a given landscape profile, this implementation simply returns the influence map of the area. It returns a map with zero values for all pixels for other profiles.
- **Height-based Landscape:** This implementation takes an ordered list of profiles and blends them based on the height data in the given area. For every profile not included in the list, this implementation returns a map with zero values for all pixels.

Algorithm 3 shows how the height-based implementation generates partial landscape maps for the profiles in the list. This algorithm starts by calculating a step value to determine the height difference between two consecutive profiles in the given list (lines 1 to 5). Next, it iterates over the profiles of the given list. For each profile, we first calculate the desired height for that profile (line 7). Then, we compute the absolute difference between the desired height

**Algorithm 3** Generating partial landscape maps based on height data.

**Inputs:**

- $profiles$: a list of landscape profiles with $N$ elements
- $infMap$: an influence map
- $hMap$: a heightmap
- $pixels$: coordinates of map pixels
- $fadeHeight$: a positive scalar to control the fading effect between profiles

**Result:**

- $landMaps$: A mapping that assigns a landscape map to every landscape profile in $profiles$

$landMaps \leftarrow \emptyset$
$minH \leftarrow$ minimum value in $hMap$
$maxH \leftarrow$ maximum value in $hMap$
$rangeH \leftarrow maxH - minH$
$step \leftarrow rangeH/(N-1)$
**for** $i$ from 0 to $N-1$ **do**
    $desiredH \leftarrow minH + i * step$
    **for** each $pixel$ in $pixels$ **do**
        $diffH \leftarrow |hMap[pixel] - desiredH|$
        $norm \leftarrow 1 - min(diffH, fadeHeight)/fadeHeight$
        $norm \leftarrow ease(norm)$
        $landMaps[profiles[i]][pixel] \leftarrow norm$
**for** each $p$ in $profiles$ **do**
    $landMaps[p] \leftarrow landMaps[p] * infMap$
**return** $landMaps$

and the heightmap value for every pixel (line 9). At the next stage, we use a formula that returns one if the difference is 0 and the pixel's height is precisely equal to the desired height. If the difference is greater than or equal to the provided fade height, this formula returns 0. For the pixels where the height difference is less than the fade height, this formula linearly interpolates the result between 1 to 0 (line 10). In line 11, we apply an easing function on the resulting values to create a smooth transition between height profiles. The last step of this algorithm multiplies all calculated values to their corresponding values in the influence map for each pixel. This step makes sure that the generated landscape maps are partial with respect to the area with which they are associated.

### 2.7.2 Merging Partial Landscape Maps

After generating the partial landscape maps for every area, we have to merge maps related to the same landscape profiles and generate a complete map for every profile. Assuming $A$ to be the set of all areas in the map, $partialLandMaps$ to contain outputs of the landscape map generators for every area, and $f$ to be a landscape profile, Formula 2 calculates the value of final landscape maps for the pixel $p$.

$$landMaps[f][p] = \sum_{a \in A} partialLandMaps[a][f][p] \tag{2}$$

Figure 16 presents some landscape profiles in addition to a mapping between areas depicted in Figure 10 and a set of landscape map generators. Also, this figure shows the resulting landscape maps blended by color into a single frame. As seen in this figure, areas $A$ to $I$ only have one profile each, where different profiles influence areas $J$, $K$, and $L$ because their landscape generators have used height data to blend multiple profiles in them.

### 2.7.3 Surface Map Generation

After generating landscape profiles, we are ready to generate surface maps. Structurally, a surface map is similar to an influence map. These maps determine the influence of a surface type over the pixels of the map. Assume $s$ to be a surface type mentioned in the landscape profiles, and $landMaps_s$ to be the set that includes every landscape map whose corresponding profile has s. We can use Formula 3 to calculate the surface value for every pixel $p$.

$$surfaceMap_s[p] = \sum_{map \in landMaps_s} map[p] \tag{3}$$

Figure 17 shows the surface maps generated using this formula for the landscape maps presented in Figure 16. This figure blends three surface maps by color to present them in a

Landscape Profiles

| *Landscape Profile* | Surface Type | Item Density | Item Weights |
|---|---|---|---|
| ***land_0*** | None | 0.0 | - |
| ***land_1*** | Grass | 0.01 | [Cedar: 5, Bush: 10] |
| ***land_2*** | Soil | 0.025 | [Oak: 10, Bush: 10] |
| ***land_3*** | Snow | 0.02 | [Oak: 1, Cedar: 7] |
| ***land_4*** | Snow | 0.001 | [Oak: 1, Cedar: 1] |

Areas and landscape Generators

| *Area Name* | Landscape Map Generator | |
|---|---|---|
| | Type | Profiles |
| ***A*** | Single-Profile | land_1 |
| ***B*** | Single-Profile | land_1 |
| ***C*** | Single-Profile | land_2 |
| ***D*** | Single-Profile | land_2 |
| ***E*** | Single-Profile | land_2 |
| ***F*** | Single-Profile | land_3 |
| ***G*** | Single-Profile | land_3 |
| ***H*** | Single-Profile | land_4 |
| ***I*** | Single-Profile | land_4 |
| ***J*** | Height-based | [land_0, land_1, land_2, land_3, land_4] |
| ***K*** | Height-based | [land_0, land_1, land_2, land_3, land_4] |
| ***L*** | Height-based | [land_0, land_1, land_2, land_3, land_4] |

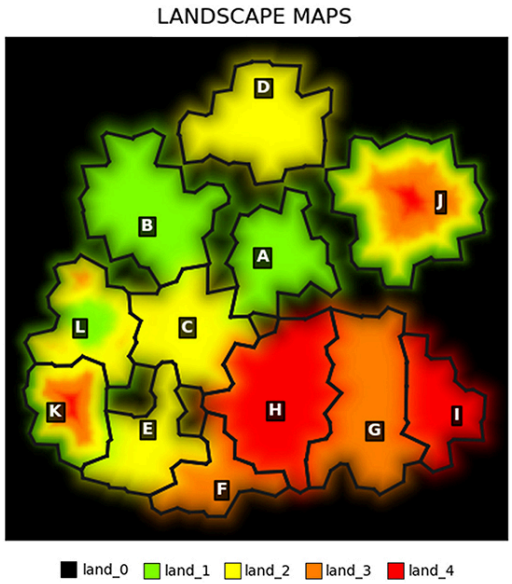



Figure 16: An example set of landscape profiles and landscape map generators along with the resulting landscape maps

single frame. As seen in this figure, the surfaces are set precisely according to the landscape profiles in Figure 16. For example, the surface of area *B* is dominantly grass which is similar to the surface type of *land_1* profile. Also, the surface of area *J* is a mixture of all surface types as its landscape generator is height-based, and its height rises as we get closer to its center (see Figure 14).

### 2.7.4 Item Placement

We can use the previously generated landscape maps for item placement. To this end, we start by generating the following maps:

- **Item Density Map:** An item density map determines the probability of a pixel to be

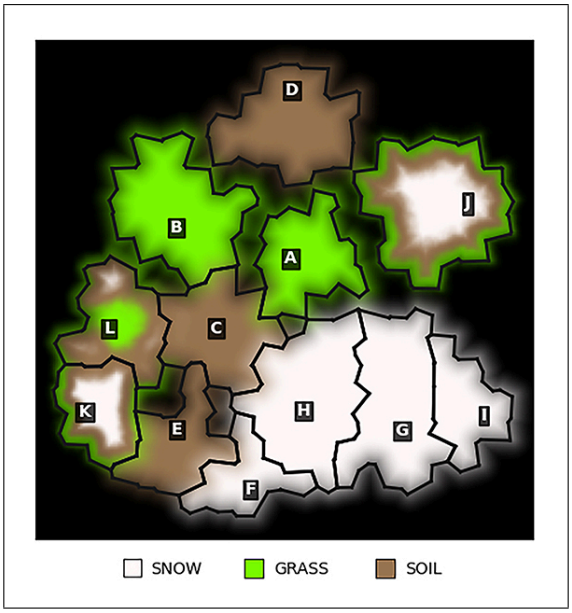



Figure 17: Three surface maps color-blended into a single map

chosen for item placement. We only need to generate one item density map.

- **Item Type Map:** For a given item type, an item type map determines the probability of a pixel to be the location of an item from the related type. For every item type mentioned in the landscape profiles, we need to generate an item type map.

Assume $landMaps$ to be a mapping that maps every landscape profile to a landscape map. We use Formula 4 to calculate item density for every pixel $p$.

$$itemDensityMap[p] = \sum_{(f,map) \in landMaps} map[p] * itemDensity_f \tag{4}$$

Furthermore, assume $itemWeightSum_f$ to be the sum of all item weights mentioned in a landscape profile like $f$. For an item type like $t$, we use Formula 5 to calculate the value of the corresponding item type map for every pixel $p$. Notice that we consider the weight of a type to be 0 if it is not mentioned in a profile.

$$itemTypeMap_t[p] = \sum_{(f,map) \in landMaps} map[p] * \frac{itemWeights_f[t]}{itemWeightSum_f} \tag{5}$$

Now, we iterate over pixels of the map. For every pixel, we use the value of the item density map as the probability of randomly choosing that pixel for item placement. If a pixel is chosen for item placement, we use the values of the item type maps as probabilities for

randomly choosing an item type. Figure 18 shows the item density map and item type maps generated from the landscape maps illustrated in Figure 16. Moreover, this figure presents the final item locations obtained from these maps.

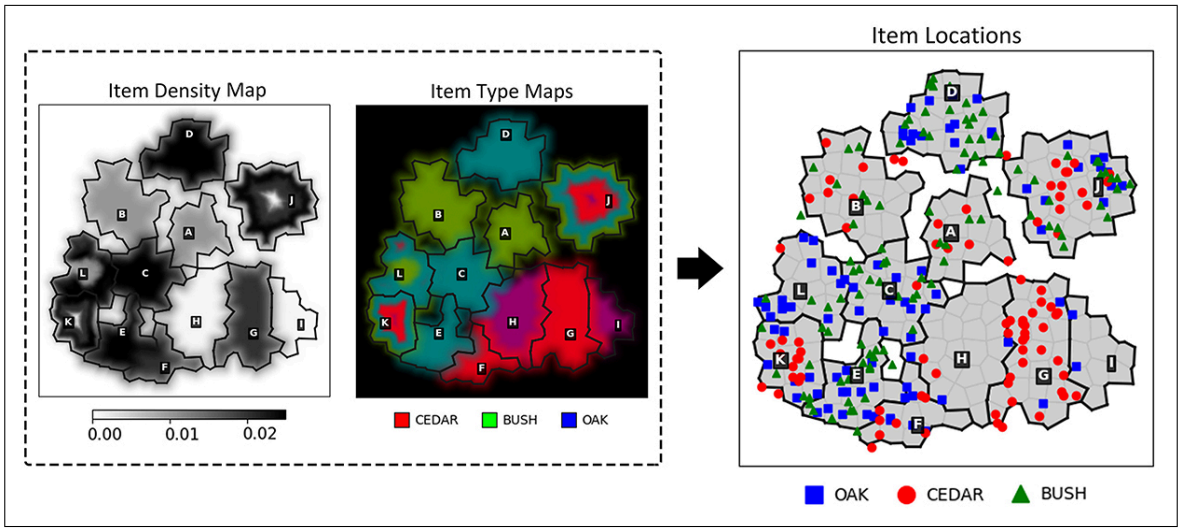



Figure 18: The result of the item placement process

As seen in this figure, the density and type of items follow the landscape profiles and landscape map generators presented in Figure 16. For example, area *G* is mainly populated by cedars because it was dominated by the *land_3* profile, assigning a relatively considerable weight to cedar item type. Moreover, area *D* is populated by a nearly equal number of bushes and oaks since it only takes effect from the *land_2* profile, which assigns equal weights to bushes and oaks. Finally, it is worth noting that area *J* was associated with a height-based landscape map generator. Thus, the population of item types changes as we move away from the borders of this area and get closer to its central parts.

## 2.8 Marker Placement

A marker is a general-purpose label on the map that designers can use for different cases. For example, a marker can locate a pickup item in an FPS map or a mission's location in an RPG map. This process takes marker specifications from designers and finds the best possible locations for each specification. A marker specification includes the following properties:

- **Area:** The containing area of the marker.
- **Height Preference:** Preference to more elevated locations.
- **Height Importance:** The weight of the height preference. Greater values increase the effect of height preference on the final placement.

- **Center Preference:** Preference to being near the center of the containing area.
- **Center Importance:** The weight of the center preference. Greater values increase the effect of center preference on the final placement.
- **Effector Areas:** A list of areas where markers are supposed to be near them.
- **Effector Importance:** The weight of proximity to effector areas. Greater values increase the tendency to be closer to these areas.

For any marker specification like

$$\begin{aligned} M = <& area, \\ & hgtPref, hgtImp, \\ & cntrPref, cntrImp, \\ & effAreas, effImp > \end{aligned}$$

we define the function $effDist$ that takes a coordinate and a mapping of area polygons as inputs and returns the sum of the input coordinate distances to the effector areas' polygons. Later, we use this function in our marker placement algorithm. Formula 6 formally presents this function.

$$effDist(p, polygons) = \sum_{a \in effAreas} \text{distance of } p \text{ to } polygons[a] \tag{6}$$

Algorithm 4 involves the pseudo-code to find a location for a given marker specification. This algorithm calculates a penalty for every canvas seed in the containing area polygon. The final goal is to return the seed coordinates with the lowest penalty. To this end, we consider three penalty components. The first component captures height preferences (lines 11 to 13). The second component calculates the penalty regarding preferring borders of the containing polygon (lines 14 to 16). Finally, the third component computes the penalty of being far from effector areas (lines 17 to 19). We multiply all of these penalties by their corresponding importance values mentioned in the specification (lines 13, 16, and 19) and obtain the total penalty of the seed by summing the results.

Figure 19 presents three marker specifications and the result of running the marker placement algorithm for these specifications, the heightmap presented in Figure 14, and the Euclidean area polygons indicated in Figure 8.

## 2.9 Assertion Checking

JAHAN allows designers to formally evaluate generated maps against their desired properties. To this end, JAHAN provides a set of pre-build blocks that query different aspects of the resulting map. Designers can combine these queries or use them programmatically to formally

**Algorithm 4** Finding the best location for a marker specification

**Inputs:**

- $polygons$: a mapping that maps every area to a polygon
- $canvas$: a canvas for the map
- $hMap$: a heightmap
- $M$: the marker specification

**Result:**

- $markerLocation$: a coordinate on the map

$cntr \leftarrow$ average of vertices of $polygons[area]$
$S \leftarrow \{s \in seeds_{canvas} \mid s \text{ is inside } polygons[area]\}$
$rangeH \leftarrow \max_{s \in S} hMap[s] - \min_{s \in S} hMap[s]$
$desiredH \leftarrow \min_{s \in S} hMap[s] + rangeH * hgtPref$
$rangeD \leftarrow \max_{s \in S} |s - cntr| - min_{s \in S}|s - cntr|$
$desiredD \leftarrow \min_{s \in S} |s - cntr| + rangeD * (1 - cntrPref)$
$minE \leftarrow \min_{s \in S} effDist(s, polygons)$
$rangeE \leftarrow \max_{s \in S} effDist(s, polygons) - minE$
$minPenalty \leftarrow$ inf
**for** each $s \in S$ **do**
    **if** $rangeH == 0$ **then** $penaltyH \leftarrow 0$
    **else** $penaltyH \leftarrow |hMap[s] - desiredH|/rangeH$
    $penaltySum \leftarrow penaltyH * hgtImp$
    **if** $rangeD == 0$ **then** $penaltyD \leftarrow 0$
    **else** $penaltyD \leftarrow ||s - cntr| - desiredD|/rangeD$
    $penaltySum \leftarrow penaltySum + penaltyD * cntrImp$
    **if** $rangeE == 0$ **then** $penaltyE \leftarrow 0$
    **else** $penaltyE \leftarrow |effDist(s, polygons) - minE|/rangeE$
    $penaltySum \leftarrow penaltySum + penaltyE * effImp$
    **if** $penaltySum < minPenalty$ **then**
        $markerLocation \leftarrow s$
        $minPenalty \leftarrow penaltySum$
**return** $markerLocation$

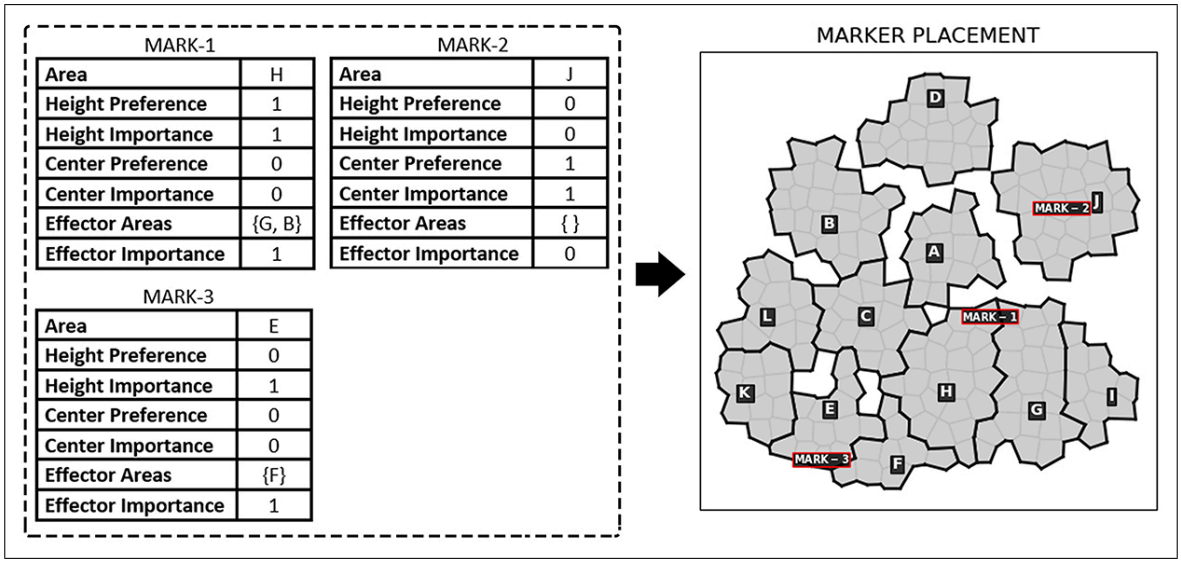



Figure 19: Marker placement

declare assertions about maps. Table 1 presents names and descriptions of these queries. It is worth mentioning that these queries only rely on the format of the data generated by the pipeline processes. These data formats are independent of the algorithms and implementations; therefore, adding new query blocks is not a complicated task and only requires understanding related data formats.

As an example for the map presented in Figure 18, the assertion presented in Formula 7 checks if the density of oak trees is greater in area $E$, compared to area $L$.

$$\frac{NumberOfItemsInArea(E, Oak)}{SizeOfArea(E)} > \frac{NumberOfItemsInArea(L, Oak)}{SizeOfArea(L)} \tag{7}$$

We present more asssertions in Section 3 and show how `JAHAN`'s assertion checking feature can be used to fomrulate actual design requirements.

# 3 Case Studies

We used `Python 3.8` to implement `JAHAN` because of its popularity, ease of use, and abundance of its packages. We utilized `NetworkX` [7] to perform graph theory tasks like planar embedding and force-directed graph drawing. Moreover, `SciPy` [12] has been applied to generate Voronoi diagrams and Delaunay triangulation problems. Lastly, we used `Shapely` [11] to solve our geometry-related problems.

This section presents two case studies to demonstrate `JAHAN`'s capabilities for generating different types of maps and the level of control it gives to designers. We also show how

Table 1: Building blocks of assertion decleration

| **Building Block** | **Description** |
|---|---|
| `ShortestPathFromTo` | Takes names of two areas and returns the shortest path between them as a list of areas. |
| `IsAreaInPath` | Takes the name of an area and a path. Returns TRUE if the area is on the path. Otherwise, returns False. |
| `LengthOfPath` | Takes a path and returns its length. |
| `SizeOfArea` | Takes the name of an area and returns its size. |
| `BorderLegnth` | Takes names of two areas and returns the length of their border. Returns $-1$ if given areas do not share a border. |
| `ItemsInArea` | Takes the name of an area and returns its items as a list of names. |
| `NumberOfItemsInArea` | Takes the name of an area and the name of an item. Returns the number of item instances in that area. |
| `NearestItemToMarker` | Takes the name of a marker and the name of an item. Returns the location of the nearest item instance to that marker. |
| `FarthestItemToMarker` | Takes the name of a marker and the name of an item. Returns the location of the farthest item instance to that marker. |
| `NearestPointToMarker` | Takes the name of a marker and the name of an area. Returns the closest point to that marker on the area's border. |
| `FarthestPointToMarker` | Takes the name of a marker and the name of an area. Returns the farthest point to that marker on the area's border. |
| `CenterOfMassForArea` | Takes the name of an area and returns its center of mass. |
| `Distance2D` | Takes two inputs. Inputs can be marker names or coordinates of a location. Returns the total displacement of the shortest path between inputs regardless of their height. |
| `Distance3D` | Takes two inputs. Inputs can be marker names or coordinates of a location. Returns the total displacement of the shortest path between inputs while considering the map's elevation. |

`JAHAN`'s assertion checking feature can help use evaluate the generated maps and find if they are not consistent with their related design requirements. To this end, we first generate a simple multiplayer FPS map for the diffuse-the-bomb game mode. Then, we generate an RTS campaign map.

## 3.1 Generating an FPS Map

In FPS games, players view the map from a first-person perspective and carry different types of firearms. In Multiplayer FPS maps, players should collaborate or fight with other players to perform a unique task for every game mode.

A popular mode of multiplayer FPS games is *diffuse-the-bomb*. In this mode, players split into two teams of terrorists and counter-terrorists. Players start the match in a specific place on the map, which the designers predefine for every team, known as spawn points. Then, the terrorist group must fight with counter-terrorist players to plant a bomb in a specific location on the map. After successfully planting the bomb, the terrorist team must defend it against the counter-terrorist players who are now supposed to defuse the bomb before it explodes. The terrorist team will win the game if the bomb explodes before the match time out. Otherwise, the counter-terrorist team wins. Also, each team can win the match if they eliminate all of the opposing team's players.

Here, fairness is a critical aspect of a good map designed for such a mode. In other words, the map should be balanced for both teams. For example, the shortest distance from every spawn location to the bomb location should be equal. Moreover, both teams should have equal access to strategic points on the map, such as covers or sniping positions. Consider the following specifications for a classic 8-shaped FPS map based on the requirements mentioned above:

1. The map contains two areas named *BASE-A* and *BASE-B*, each headquarter of one of the teams.
2. The map contains a central arena that we call *CHOKE* because most of the fighting will happen there.
    - The bomb location is in the middle of this arena.
    - This arena is filled with uniformly distributed cover points.
    - The size of this arena should be relatively larger than the base sites.
3. Each base is connected to the *CHOKE* with a long and short corridor.
    - All long corridors must have the same length.
    - All short corridors must be equally long.
4. There are two spawn points on the map, each located in one of the bases near the gates of the short corridors.
5. Players can climb two towers on the map to obtain a strategic advantage over enemies. These towers reside in the long corridors near the gates of the central arena.

To generate a map from these specifications, we define an area layout graph as presented in Figure 20. In this figure, *L-CORR-A* and *S-CORR-A* are respectively the long and short corridors that connect *BASE-A* to the *CHOKE*. Similarly, *L-CORR-B* and *S-CORR-B* do the same for the *BASE-B*.

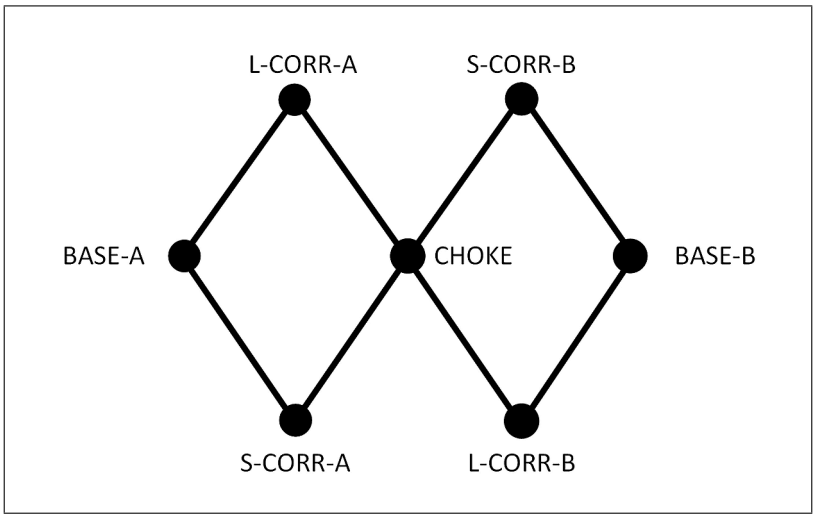



Figure 20: The area layout graph of the FPS case study

Applying the Euclidean distance calculator on a canvas generated by a radial seed generator, we use this graph to generate polygons. We start by embedding this graph on the slate. Then, we provide the stretch weights of areas to the skeleton generator module to obtain the area skeletons. Next, we set the bound radius of every area and generate area polygons. Figure 21 depicts the values we used for stretch weights and bound radiuses, along with the generated area skeletons and resulting polygons.

| *Area Name* | Stretch Weight | Bound Radius |
|---|---|---|
| *BASE-A* | 0.75 | 0.2 |
| *BASE-B* | 0.75 | 0.2 |
| *CHOKE* | 1 | 0.1 |
| *S-CORR-A* | 1 | 0.04 |
| *L-CORR-A* | 2.5 | 0.04 |
| *S-CORR-B* | 1 | 0.04 |
| *L-CORR-B* | 2.5 | 0.04 |

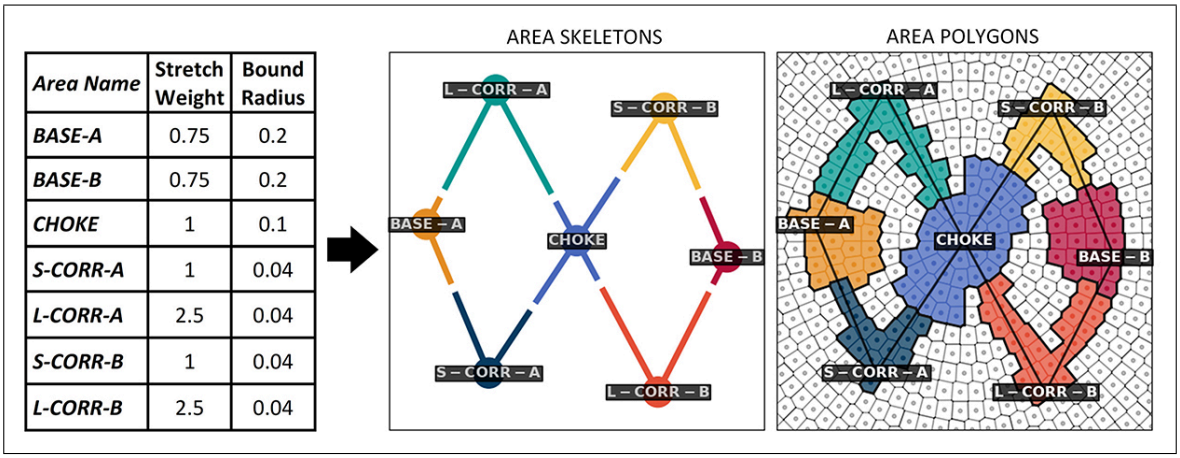



Figure 21: Polygon generation for the FPS case study

As seen in Figure 21, we set the stretch weights of base areas a little less than the stretch

weight of the central arena in order to make the final base polygons' size less than that of the central arena. Also the stretch weights of the long corridors are set relatively higher in comparison to the short ones. Finally, to make corridors narrow compared to other areas, we set their bound radiuses significantly less.

We have to define five markers for this map based on the given specification. The first marker should determine the bomb's location. Two markers are needed to determine the spawn locations for each team and two more markers are required for identifying the location of towers. Figure 22 specifies these five markers, along with their resulting locations on the map.

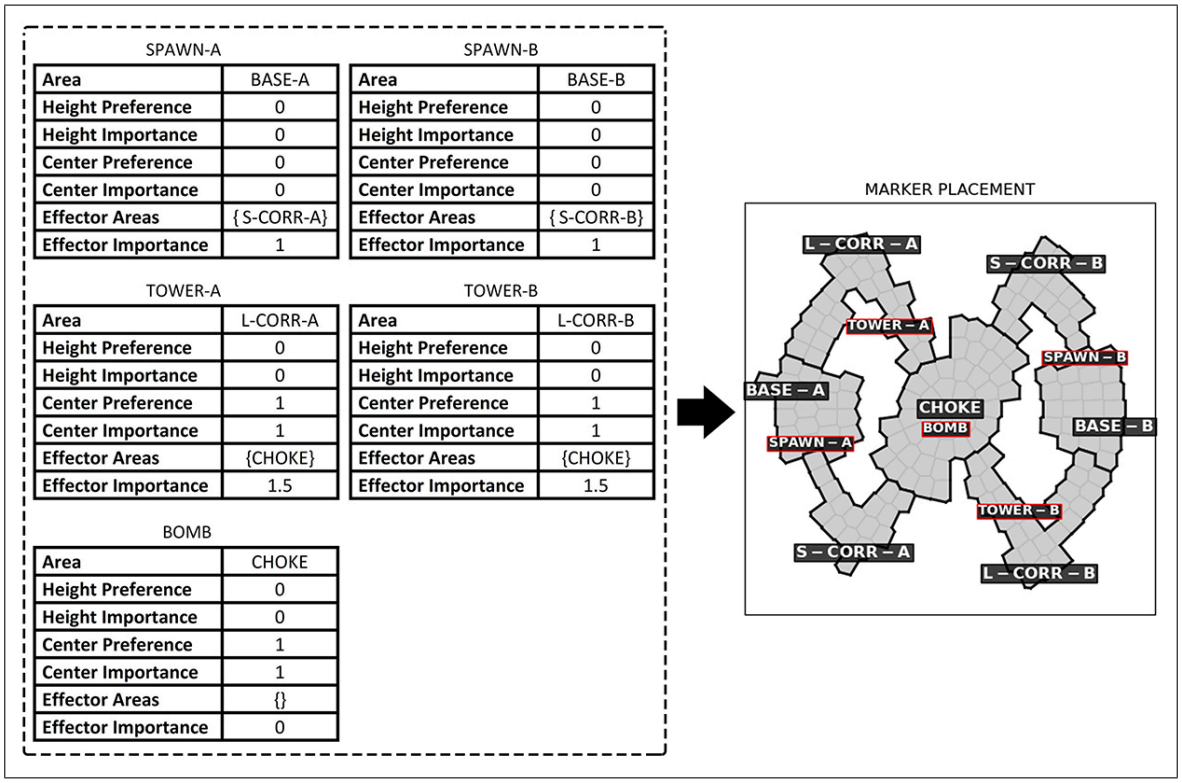



Figure 22: Marker placement for the FPS case study

As found in this figure, the markers follow the given specification. For example, the bomb marker prefers being in the center of its containing area, the *CHOKE* area; or *TOWER-A* prefers to be near the center of the *L-CORR-A area*, but not as much it prefers closeness to the *CHOKE* area.

At the final stage of the map generation, we utilize single-profile landscape generators to scatter cover points across the central arena. We define two landscape profiles, one for the central arena and another for the other areas. There is no need to specify a surface type for these profiles due to the nature of this case study. For the central arena profile, we set the item density to 0.01 and define a single item type with its weight set to 1. We do not specify

any item type for the other profile because we want some map areas to be empty of any covers. Figure 23 shows the locations of the cover points generated from this configuration.

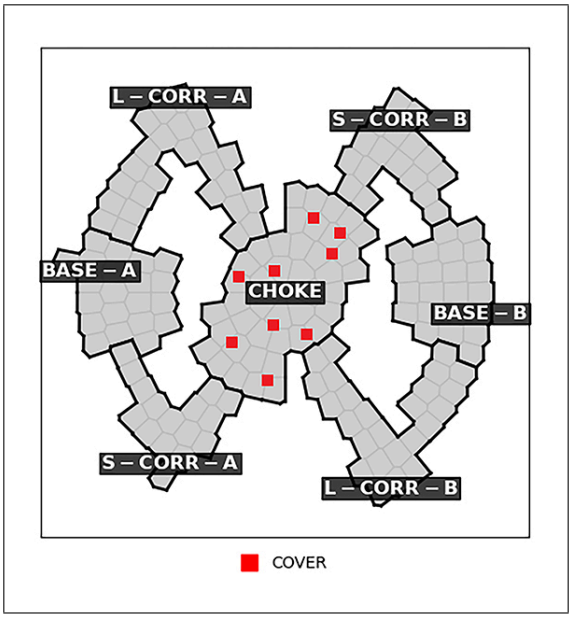



Figure 23: Cover points obtained from landscape generation

To make sure the map is balanced for both teams, we declare a set of assertions to formulate the fairness of the map. The assertion presented in Formula 8 checks if the bomb is approximately equally distant from spawn points. Also, the assertion specified by Formula 9 checks if the distance from the first team's spawn point to its sniping tower is nearly equal to that of the other team. Finally, we want to check if both teams have fair access to cover locations from their spawn points. The assertion shown in Formula 10 checks this requirement. It is worth mentioning that all of these three assertions are satisfied by the map we generated for this case study.

$$\begin{aligned} DistanceA &= Distance2D(BOMB, SPAWN\text{-}A) \\ DistanceB &= Distance2D(BOMB, SPAWN\text{-}B) \\ |DistanceA - DistanceB| &< 5 \end{aligned} \tag{8}$$

$$\begin{aligned} DistanceA &= Distance2D(SPAWN\text{-}A, TOWER\text{-}A) \\ DistanceB &= Distance2D(SPAWN\text{-}B, TOWER\text{-}B) \\ |DistanceA - DistanceB| &< 5 \end{aligned} \tag{9}$$

$$
\begin{aligned}
NearestCoverA &= NearestItemToMarker(SPAWN\text{-}A, COVER) \\
NearestCoverB &= NearestItemToMarker(SPAWN\text{-}B, COVER) \\
DistanceA &= Distance2D(SPAWN\text{-}A, NearestCoverA) \\
DistanceB &= Distance2D(SPAWN\text{-}B, NearestCoverB) \\
|DistanceA - DistanceB| &< 5
\end{aligned}
\tag{10}
$$

## 3.2 Generating an RTS Map

In an RTS game, each player constructs buildings and controls multiple units to secure areas of the map or destroy the assets of their enemies. In a typical RTS game, it is possible to create additional units and buildings by spending specific resources. These resources are gathered by controlling specific sites on the map or having certain units and buildings for this purpose. To summarize, the typical game in the RTS genre features resource-gathering, base-building, in-game technological development, and indirect control of units. For example, consider the following design for a simple RTS game:

1. Players must mine gold or refine the oil to make money. They can spend their revenue on constructing barracks and factories.
    - Barracks can train soldiers while factories can train engineers.
    - Training soldiers and engineers require money and time.
2. Players can mine gold from gold tracks.
    - Gold tracks only exist in specific areas of the map.
    - To mine gold, the player has to send engineers to the gold mines to bring gold back to the base. Only then is the gold converted into money.
3. Players can capture old refineries to make revenue from them.
    - Oil refineries only exist in the predefined locations on the map.
    - The player must send an engineer into an oil refinery to capture it.
4. There might be other factions on the map. The computer controls these factions.
    - Every faction has a headquarter. A faction is defeated when soldiers destroy their headquarter.
    - These factions can gather resources, make units, capture refineries, and attack other factions.

5. Some map areas are heavily polluted, and ordinary units quickly die if they enter them. Players can spend money to upgrade their barracks and factories to train more resistive units.

Accordingly, consider the following specifications for a map based on the above design.

1. The player starts the game in their base (*P-BASE*).
    - Players have access to two gold mines (*GM-1* and *GM-2*) from their base. *GM-1* must have twice as much gold track as *GM-2*.
    - In general, all the gold mines are areas with gold tracks scattered all across them.
2. A group of outlaws on the map has their territory (*O-BASE*). The computer controls outlaws.
    - *O-BASE* should be twice as big as *P-BASE*
    - The headquarter of the outlaw faction resides at the center of its territory.
    - The outlaw territory is connected to *GM-1* by a bridging corridor (*BRDG-1*). Also, it is connected to *GM-2* by two other bridges (*BRDG-2* and *BRDG-3*). Bandits will use these bridges to attack player engineers who are mining gold. Moreover, they have three outposts at the gates of each bridge.
    - The outlaw territory has access to an oil field (*OF*) with several oil refineries through a narrow passage that is heavily polluted (*BRDG-4*).
3. The primary enemy of the player has a headquarter in their territory (*E-BASE*), which has borders with the player base.
    - *E-BASE* should be as big as *P-BASE*
    - Enemies have their own gold mine (*GM-3*) which has as many tracks as the total number of tracks in *GM-1* and *GM-2*.
    - There is a polluted wasteland (*WASTE*) between the enemy gold mine and the oil field.
4. The primary objective of the player is to destroy the enemy headquarter. Destroying outlaws and capturing refineries is an optional objective.

We define an area layout graph based on these specifications as presented in Figure 24.

Using Euclidean distances, we use the defined graph to generate polygons on a hexagonal canvas. Figure 25 shows the values of the stretch weights and bound radiuses, along with the generated area skeletons and resulting polygons.

Based on the given specifications, we have to describe six markers for this map:

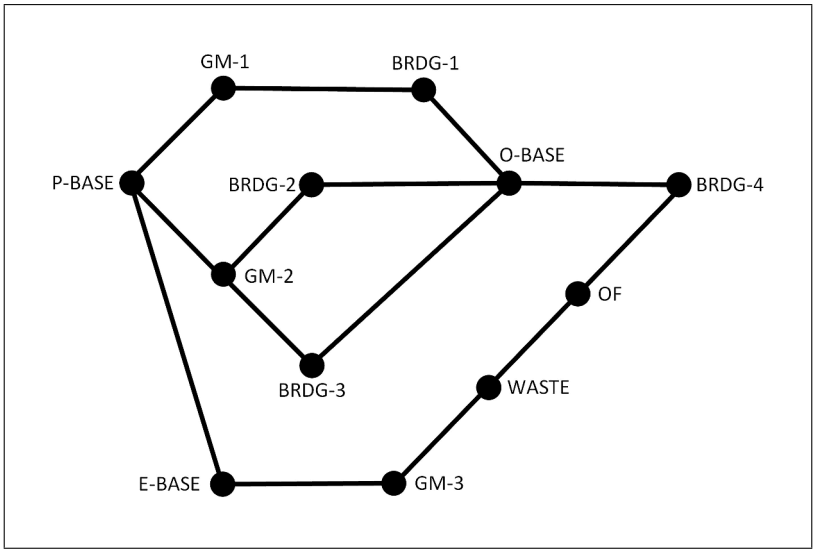



Figure 24: Area layout graph of the RTS case study

- *P-HQ*, *O-HQ*, and *E-HQ* markers should be at the center of *P-BASE*, *O-BASE*, and *E-BASE*, respectively.
- *OUTPOST-1*, *OUTPOST-2*, and *OUTPOST-3* markers should be in *O-BASE* near *BRDG-1*, *BRDG-2*, *and BRDG-3*, respectively.

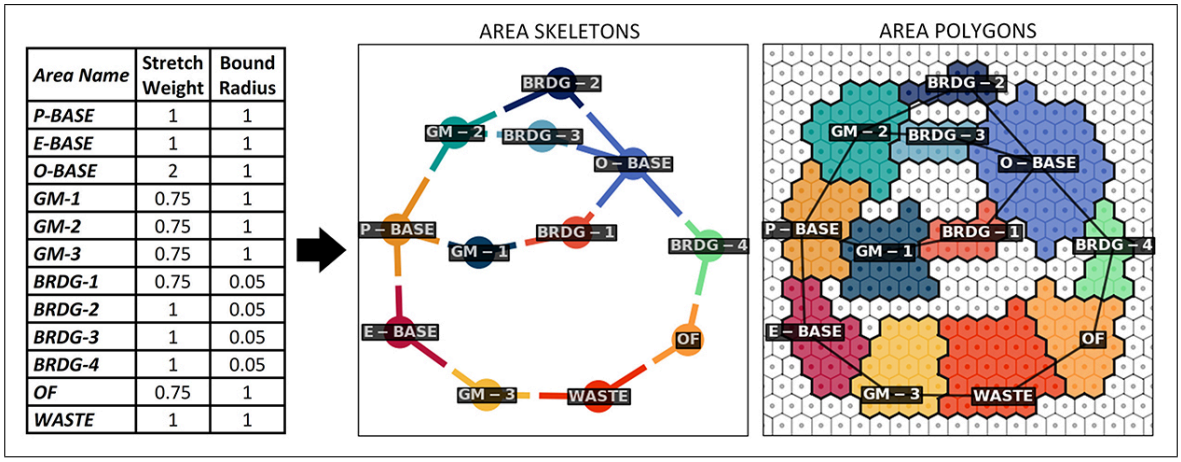
Figure 26 specifies these six markers, along with their resulting locations on the map.

Now, we utilize single-profile landscape generators to scatter gold tracks and oil refineries across the map and determine if the surface of each area is polluted. To this end, we define four landscape profiles as follows:

- For *GM-1*, *GM-2*, and *GM-3*, we define a specific profile. This profile does not specify any surface type but includes gold track items and has an item density of 0.01.
- The second profile has a polluted surface and no items. This profile provides the required surface for *BRDG-4* and *WASTE*.
- The third profile has a polluted surface, including oil refinery items, and has an item density of 0.01. We assign this profile to *OF*.
- We assign the last profile to the areas with neither gold mines nor oil fields not polluted.

Figure 27 shows the gold tracks' locations, oil refineries' locations, and the map's surface.

Lastly, we declare the following assertions to check constraints mentioned in the map specification. To this end, Formula 11 and Formula 12 check the constraints over the relative size




| Area Name | Stretch Weight | Bound Radius |
|---|---|---|
| *P-BASE* | 1 | 1 |
| *E-BASE* | 1 | 1 |
| *O-BASE* | 2 | 1 |
| *GM-1* | 0.75 | 1 |
| *GM-2* | 0.75 | 1 |
| *GM-3* | 0.75 | 1 |
| *BRDG-1* | 0.75 | 0.05 |
| *BRDG-2* | 1 | 0.05 |
| *BRDG-3* | 1 | 0.05 |
| *BRDG-4* | 1 | 0.05 |
| *OF* | 0.75 | 1 |
| *WASTE* | 1 | 1 |

Figure 25: Polygon generation for the RTS case study

of areas. Formula 13 encapsulates the constraint over the ratio of gold tracks in *GM-1* and *GM-2*. Finally, Formula 14 checks if the number of gold tracks in *GM-3* is approximately equal to the sum of the number of gold tracks in *GM-1* and *GM-2*.

$$|SizeOfArea(P\text{-}BASE) - SizeOfArea(E\text{-}BASE)| < 5 \tag{11}$$

$$|SizeOfArea(O\text{-}BASE) - 2 * SizeOfArea(P\text{-}BASE)| < 10 \tag{12}$$

$$\begin{gathered} GoldTracksInGM1 = NumberOfItemsInArea(GM\text{-}1, GoldTrack) \\ GoldTracksInGM2 = NumberOfItemsInArea(GM\text{-}2, GoldTrack) \\ |GoldTracksInGM1 - 2 * GoldTracksInGM2| < 2 \end{gathered} \tag{13}$$

$$\begin{gathered} GoldTracksInGM3 = NumberOfItemsInArea(GM\text{-}3, GoldTrack) \\ |GoldTracksInGM3 - (GoldTracksInGM1 + GoldTracksInGM2)| < 2 \end{gathered} \tag{14}$$

The map we presented in this case study satisfies assertions that were mentioned in Formula 11 and Formula 12. However, assertions that were presented in Formulas 13 and 14 are not satisfied by this map. As Figure 27 shows, there are 14 gold tracks in *GM-1*, 21 gold tracks in *GM-2*, and 17 gold tracks in *GM-3*. Accordingly, the constraint presented by Formula 13 is not satisfied because *GM-1* does not have twice as much gold track as *GM-2*. Also, the constraint shown in Formula 14 is not satisfied because the number of gold tracks in *GM-3* is not equal to the sum of gold tracks in *GM-1* and *GM-2*.

The failed assertions are addressing constraints over the distribution of gold tracks in the map. Therefore, we must revisit the setup of our landscape profiles to refine this map with regard to these design constraints. For example, we can define three different landscape

P-HQ

| Area | P-BASE |
|---|---|
| Height Preference | 0 |
| Height Importance | 0 |
| Center Preference | 1 |
| Center Importance | 1 |
| Effector Areas | {} |
| Effector Importance | 0 |

OUTPOST-1

| Area | O-BASE |
|---|---|
| Height Preference | 0 |
| Height Importance | 0 |
| Center Preference | 0 |
| Center Importance | 0 |
| Effector Areas | {BRDG-1} |
| Effector Importance | 1 |

E-HQ

| Area | E-BASE |
|---|---|
| Height Preference | 0 |
| Height Importance | 0 |
| Center Preference | 1 |
| Center Importance | 1 |
| Effector Areas | {} |
| Effector Importance | 0 |

OUTPOST-2

| Area | O-BASE |
|---|---|
| Height Preference | 0 |
| Height Importance | 0 |
| Center Preference | 0 |
| Center Importance | 0 |
| Effector Areas | {BRDG-2} |
| Effector Importance | 1 |

O-HQ

| Area | O-BASE |
|---|---|
| Height Preference | 0 |
| Height Importance | 0 |
| Center Preference | 1 |
| Center Importance | 1 |
| Effector Areas | {} |
| Effector Importance | 0 |

OUTPOST-3

| Area | O-BASE |
|---|---|
| Height Preference | 0 |
| Height Importance | 0 |
| Center Preference | 0 |
| Center Importance | 0 |
| Effector Areas | {BRDG-3} |
| Effector Importance | 1 |

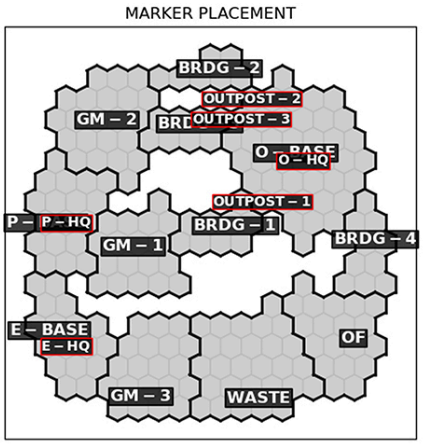



Figure 26: Marker placement for the RTS case study

profiles with proper item densities for *GM-1*, *GM-2*, and *GM-3* instead of using a single profile for all of them.

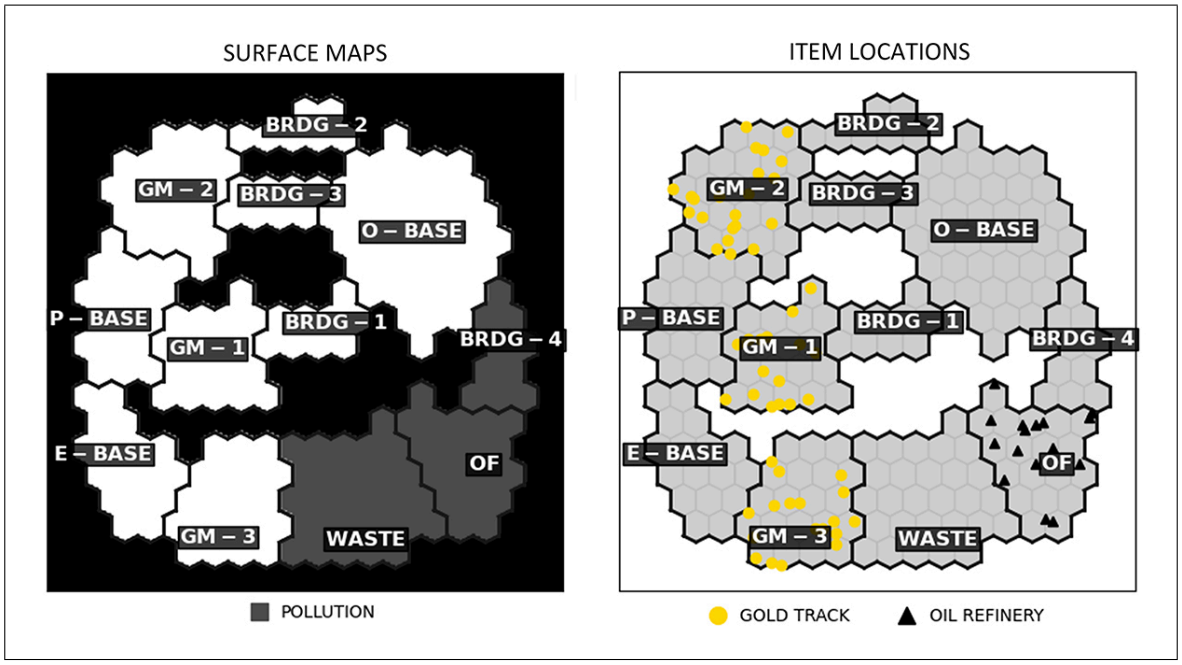



Figure 27: Item locations and the surface of the RTS case study

# References


[1] Franz Aurenhammer. “Voronoi Diagrams—a Survey of a Fundamental Geometric Data Structure”. In: *ACM Comput. Surv.* 23.3 (Sept. 1991), pp. 345–405. ISSN: 0360-0300. DOI: 10.1145/116873.116880. URL: https://doi.org/10.1145/116873.116880.

[2] M. de Berg et al. *Computational Geometry: Algorithms and Applications*. Springer, 2008. ISBN: 9783540779735.

[3] M. Chrobak and T. H. Payne. “A Linear-Time Algorithm for Drawing a Planar Graph on a Grid”. In: *Inf. Process. Lett.* 54.4 (May 1995), pp. 241–246. ISSN: 0020-0190. DOI: 10.1016/0020-0190(95)00020-D. URL: https://doi.org/10.1016/0020-0190(95)00020-D.

[4] David S. Ebert et al. *Texturing and Modeling: A Procedural Approach*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1558608486.

[5] J. Joseph Fowler and Stephen G. Kobourov. “Planar Preprocessing for Spring Embedders”. In: *Graph Drawing*. Ed. by Walter Didimo and Maurizio Patrignani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 388–399. ISBN: 978-3-642-36763-2.

[6] Thomas M. J. Fruchterman and Edward M. Reingold. “Graph drawing by force-directed placement”. In: *Software: Practice and Experience* 21.11 (1991), pp. 1129–1164. DOI: https://doi.org/10.1002/spe.4380211102. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380211102. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211102.

[7] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*. Pasadena, CA USA, 2008, pp. 11–15.

[8] Ares Lagae et al. “A survey of procedural noise functions”. In: *Computer Graphics Forum*. Vol. 29. 8. Wiley Online Library. 2010, pp. 2579–2600.

[9] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. USA: John Wiley & Sons, Inc., 1992. ISBN: 0471934305.

[10] Taejung Park et al. “CUDA-based Signed Distance Field Calculation for Adaptive Grids”. In: *2010 10th IEEE International Conference on Computer and Information Technology*. 2010, pp. 1202–1206. DOI: 10.1109/CIT.2010.217.

[11] *Shapely Documentation*. https://shapely.readthedocs.io/en/latest/. Accessed: 2020-03-14. 2008.

[12] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature Methods* 17.3 (Mar. 2020), pp. 261–272. ISSN: 1548-7105. DOI: 10.1038/s41592-019-0686-2. URL: https://doi.org/10.1038/s41592-019-0686-2.

[13] D.B. West. *Introduction to Graph Theory*. Featured Titles for Graph Theory. Prentice Hall, 2001. ISBN: 9780130144003.

[14] Steven Worley. “A Cellular Texture Basis Function”. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 291–294. ISBN: 0897917464. DOI: 10.1145/237170.237267. URL: https://doi.org/10.1145/237170.237267.