# Advanced Technologies: Procedural City Generation

**William Whitehouse (19019239)**
*University of the West of England*

October 14, 2022

## 1 Introduction

Procedural city generation involves the use of algorithms and processes to automatically create virtual cities. It has multiple potential uses, including creating realistic environments for media and designing real-world cities. This report will explore the techniques used in procedural city generation, including 3D model synthesis, noise functions, and other elements of a virtual city. Existing work, challenges/limitations and future directions for this project will be discussed.

## 2 Related Work

### 2.1 City Engine

In "Procedural Modeling of Cities" (Parish and Müller, 2001), a method of generating a road system using an extended L-System is discussed, which is used in the commercial product "City Engine". The generation process involves evaluating potential roads and creating new ones using both local and global rules. The global rules determine the overall direction of road creation based on a population density texture map and the type of road (highway or urban street for example), while the local rules impose constraints based on the environment, such as prohibiting roads from crossing illegal areas or detecting road intersections.

In a blog post, Barret argues that using an L-System is unnecessary, slow and complicated (Barrett, n.d.). As having to rely on a string rewriting algorithm is not ideal for larger cities, therefore a simple queue can be used instead. The queue is initialised with a road and new potential roads are added only when the local constraints on the segment have succeeded. The algorithm continues evaluating the roads in the queue until no more can be added and the queue is empty. The pseudo-code below shows how the simplified and faster road network generation algorithm might be implemented:

```
initialise queue Q with a single road entry
initialise road list R to empty

while (Q has elements)
{
    pop smallest road from Q
    compute local constraints
    if (local constraints succeeded)
    {
        add road to R
        add new roads using global goals to Q
    }
}
```

### 2.2 Shape Grammar

Müller presents a new shape grammar in a paper for generating buildings procedurally (Müller et al., 2006). Shape grammars provide a set of rules that dictate how to transform one shape into another through a series of iterations. This allows for the creation of complex building geometries and facades through the application of simple, parameterised rules. By using shape grammars, it is possible to generate a wide range of building designs using a relatively small set of rules such as split, repeat and scale.

### 2.3 Wave Function Collapse

Wave Function Collapse (WFC) is a Model Synthesis technique used in 3D procedural generation (Gumin, 2016). It generates grid-based content by starting with an empty grid and filling it with tiles that follow specific rules and constraints. The system uses constraint satisfaction to ensure that the chosen tiles are compatible with their neighbours and to minimize contradictions. It tracks the possible tiles that can be placed in each cell using a wave function, which represents the probability of each tile being selected. As the grid is filled, the wave function is updated until it collapses at a point where only one tile can be placed in a cell, at which point that tile is chosen and the process continues.

Golding discussed a technique at a GDC talk that uses rule-based constraints to generate buildings using pre-designed tile sets (Golding, 2010). The main benefit of this method is that all the tiles and constraints are created by an artist, ensuring that the generated buildings are physically feasible and predictable. Instead of using an existing model as

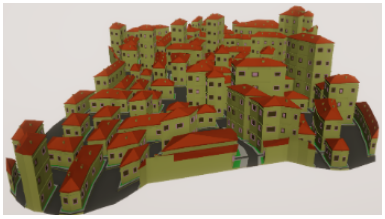


**Figure 1:** *City generated with WFC and Growing Grids Neural Network (Nordvig Møller, Billeskov, and Palamas, 2020)*

input, artists only need to create small, reusable tiles that can be used in multiple tile sets to generate a wide range of building styles by following the same constraints. This allows for the efficient creation of diverse and coherent building designs using a small set of varied tiles.

Sandhu et al. have extended WFC to include non-local constraint reasoning, weight recalculation, and area propagation (Sandhu, Chen, and McCoy, 2019). Non-local constraints enable the use of non-tile items and allow for the control of their frequency. Weight recalculation allows external forces to affect the probabilities of tiles within an area - for instance, the probability of enemies appearing in an area might increase if a lock and key item are added. Area propagation restricts the propagation of certain tiles to specific areas.

Nordvig et al. conducted further research on the WFC algorithm and successfully used it with a Growing Grid (GG) neural network to generate a city - see figure 1 (Nordvig Møller, Billeskov, and Palamas, 2020). An important aspect of their research was the inclusion of irregular grids by adding noise to the vertices of the grid to distort it. Stålberg proposed an alternative method for generating an irregular quadrilateral grid by dividing a hexagon into randomized quads and applying a relaxation algorithm (Stålberg, 2019 and Stålberg, 2021). This approach allows for the creation of a small smooth, irregular grid that is a part of a larger hexagonal grid. He implemented this method within the Townscaper game, which used WFC to procedurally model buildings (Stålberg and Raw Fury, 2021 and Stålberg, 2018).

# 3 Method

## 3.1 Tile-Based Generation

A 2D WFC algorithm generates the road network using a set of road tiles with basic neighbour relationships. A 3D version of the same algorithm is utilized to generate the buildings, which require more complex neighbour relationships but follow a similar implementation.

### 3.1.1 Tiles

Tiles are prefab objects, they hold all the necessary information in the "TileObject" script, as shown in Figure 2. This example tile has two valid anchor points, which can be seen as position gizmos in the top image and array values in the inspector. The rotate axis booleans indicate which axes the tile is allowed to rotate around, with this road tile only able to rotate around the Y axis. The normal direction vector holds the direction the tile is facing, which is unused for road tiles but necessary for tiles with identical anchors on different sides, such as walls in the building tile set.

A "TileSet" (see Figure 3) is an object which holds all tiles used during the city generation for a specific set. The tile set handles all valid tile neighbour generation. This is done by comparing each tile's anchor points with each other while applying the appropriate rotation options, if the tile's normal direction is valid, they are compared using a dot product in the following code.

```
// ...compare anchors...

if (ValidNormal(tile1Normal) &&
    ValidNormal(tile2Normal))
{
  // Dot Product:
  //  1 if they point in same direction,
  // -1 if they point in opposite direction,
  //  0 if they are perpendicular
  float dot = Vector3.Dot(tile1Normal,
                          tile2Normal);

  // no anchors on either tile
  if (anchors1.Count == 0 &&
      anchors2.Count == 0)
  {
    // ignore this tile if they are facing the same
    direction
    if (dot > -0.5f) continue;
  }
  else
  {
    // ignore this tile if they are facing opposite
    directions
    if (dot < 0.25f) continue;
  }
}

// tile is valid! add it to the neighbour list
neighbours.Add(tile2Index);
```

Each tile in the set has a frequency value representing its probability of being chosen during the WFC algorithm. The alias sampling method is used to randomly select a tile based on its frequency, as it allows for random selection regardless of the number of frequencies (prxq, 2006, rampion, 2017).

### 3.1.2 Wave Function Collapse

The WFC algorithm is run in a simple loop until all tiles have been propagated or there has been a contradiction (no valid tile for cell). There are two main steps for the WFC algorithm, firstly, the propagate step is responsible for determining which of the remaining tiles is valid in the grid cells. When this step has finished considering each cell the observe step runs, this checks the overall state of the grid (complete or contradiction) and selects which cell to next propagate if the algorithm should continue.

The algorithm selects the cell with the lowest entropy, or the fewest remaining tile options, to propagate next. If possible, it prefers to choose a cell with a completed neighbour to avoid creating disconnected "islands." If no such cell is available, it will fall back to choosing any cell. This is a modified version of the original algorithm that aims to

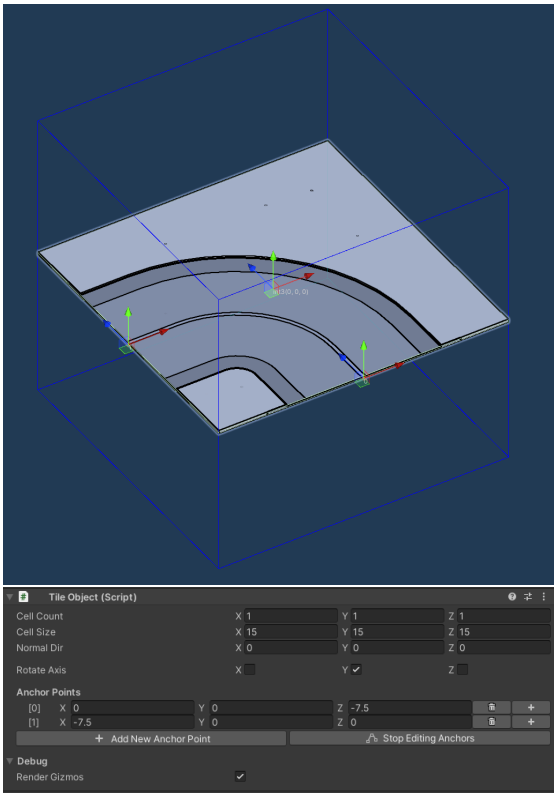



**Figure 2:** *Road bend tile (top), with its tile object script inspector (bottom).*

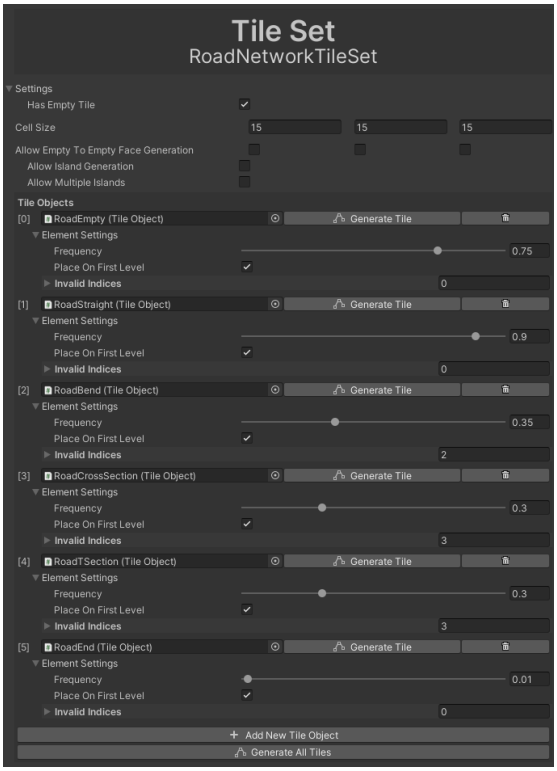



**Figure 3:** *TileSet Inspector.*

improve the chances of generating a complete output while avoiding disconnected tiles (e.g. a smaller road network not connected to the main one).

To improve the output of the algorithm, an additional step was added to remove invalid tiles from the edge of the grid before the main WFC steps. This ensures that only tiles that "end" can be chosen at the edge of the grid, resulting in a road network that is internally complete within the city.

## 3.2 City Districts

City districts are implemented using a queue-based flood fill algorithm. This process involves checking each road grid cell to see if it is empty and if it is, adding it to a district list. The flood fill algorithm is then used to identify all empty cells adjacent to the cell in question. The implementation of the flood fill process follows the pseudocode provided.

```
initialise cell queue Q with the first cell
initialise searched cell list S to empty
initialise found cell list F to empty

while (Q has elements)
{
    pop first cell from Q
    add cell to searched list S
    if (cell is an empty tile)
    {
        add cell to found list F
        for (each neighbour tile)
        {
            if (S doesn't contain neighbour cell)
            {
                add neighbour cell to Q
            }
        }
    }
}
```

Districts are selected based on a set of rules specific to each district type. These rules include a range of minimum and maximum values between 0 and 1 that represent the relative distance from the center of the city where a district can appear. For example, the highrise district has a range of 0 to 0.3, indicating that it will only appear in the center of the city. Minimum and maximum cell count parameters control the size of each district, for example, the green space district is limited to a maximum of 6 cells. Each district type also has a maximum count to ensure that it can only appear a certain number of times in the city. Finally, each district has a probability frequency that is used when randomly selecting a district using the alias sampling method mentioned earlier. Figure 4 shows a visualisation of the chosen districts in a final city model.

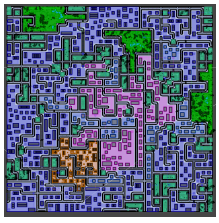
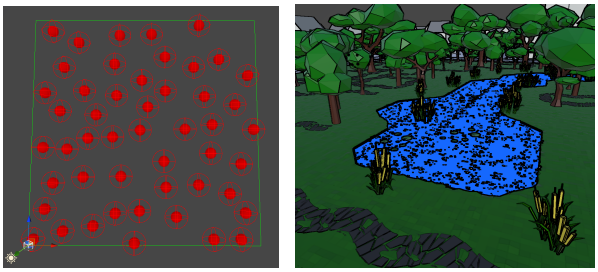


**Figure 4:** *City district visualisation.*

### 3.3 Prop Placement

Props are objects that populate the city scene, such as trees, reeds, lily pads in park areas, and barrels and pallets in industrial districts - see figure 5b. It is easy to add more props as prefabs in the inspector, each with its own probability frequency. Prop placement is achieved using blue noise, specifically Poisson Sphere Distributions. Based on the research of Lagae and Dutré (Lagae and Dutré, 2005, Lagae and Dutré, 2006), a relaxation dart-throwing technique was selected to implement this feature.

The dart-throwing technique involves randomly selecting a point within a designated area and verifying that it does not overlap with any existing points - see figure 5a. This process is repeated until a certain number of failed attempts have occurred, at which point the algorithm terminates. The overlap radius is typically defined by a user-supplied constant in the standard algorithm. The relaxation algorithm initially increases this constant by a scale factor multiple times, and if the algorithm fails to place new points, the overlap constant is decreased and the algorithm is run continuously until it reaches the original user-supplied value. This ensures that points are initially evenly distributed across the area, with additional points being added between them during later runs of the algorithm.



**(a)** **(b)**

**Figure 5:** *Image a is showing Poisson Sphere Distribution generating example points (red spheres) in an area. Image b shows prop positions being generated in the park district (trees, reeds and lily pads).*

### 3.4 Exporting City Model To FBX

Unity provides an FBX exporter plugin on GitHub (Unity Technologies, n.d.), simply downloading and installing it in the project is enough to export the city model during editor time. Some minor code changes were made to the plugin to expose relevant functions to the city generation system. This allowed the addition of an "Export To FBX" button on the generation system that would automatically export the correct objects to an FBX file.

## 4 Future Work

The WFC algorithm randomly places buildings between roads, which requires refinement in order to achieve more believable placement. One solution could be to perform building placement in a separate pass, using the WFC algorithm to generate a building to fit the designated plot.

Incorporating a height map would result in interesting varied heights in the city, currently, the model is created on a 2D grid. The height map could be supplied by the user or procedurally generated from noise.

Improving the generation time of the city is a crucial area that requires further attention. Currently, the entire city is generated on a single thread, which can be slow for large grid sizes and when creating numerous objects. One solution is to implement multi-threading, which would allow different districts to be processed asynchronously and improve computation time. Unity has a built-in job system that could be utilised for this purpose. Another potential optimization is to combine the mesh objects into a single entity, as spawning individual tiles and objects can be costly.

## 5 Evaluation

### 5.1 City Layout

Comparing the generated road network to images of real-life road networks (figure 7) key features such as roundabouts and motorways aren't generated. This is a drawback of the WFC algorithm as it relies on neighbour relationships rather than viewing the network as a whole. A solution could be to use multiple passes of WFC for each type of road and slowly refine the layout or use another algorithm such as extended L-Systems proposed by Müller (2001).

The cities are generated on a regular grid, meaning roads only connect at 90-degree intersections; compared to Bristol's road network, this looks unrealistic (figure 7a). When compared to New York (figure 7b) these intersections mimic the "block" style road architecture, although the main roads still follow the landscape which is missing from the generated cities. Adding noise to the model vertices to distort the city (Nordvig Møller, Billeskov, and Palamas, 2020) or using irregular quadrilateral hexagon-based grids (Stålberg, 2019) could improve the realism and more closely match the look of UK cities. By integrating terrain into the city generation process, the resulting cities can become more diverse and lifelike, as the roads would follow the natural topography of the land, and must avoid natural obstacles such as rivers.

### 5.2 City Detail

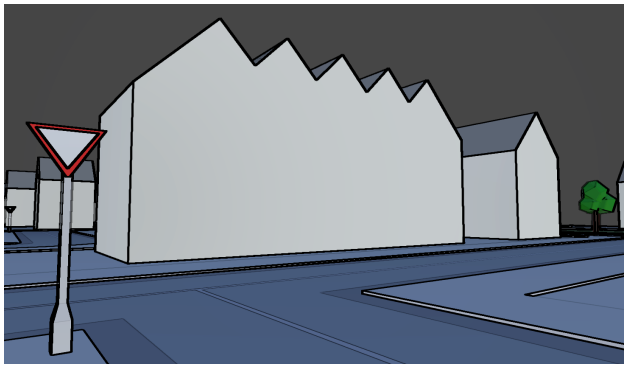
The generated cities offer a realistic inclusion of districts where different areas have varying population densities and building use. For example, the park areas appear considerably more on the outer edges of the city which emulate the urban areas of a real city and the highrise district only appears in the city centre where the population density is at its highest.

The fine-grain details of the city are missing in some areas. For example, buildings don't have any windows or doors (see figure 6). Using shape grammar proposed by Müller (2006) could help accurately generate building facades with these necessary details, procedural texturing could also be employed to increase detail without necessarily spawning new objects. The park areas are well detailed with the inclusion of pedestrian paths, ponds/lakes and props such as trees.

# 6 Conclusion

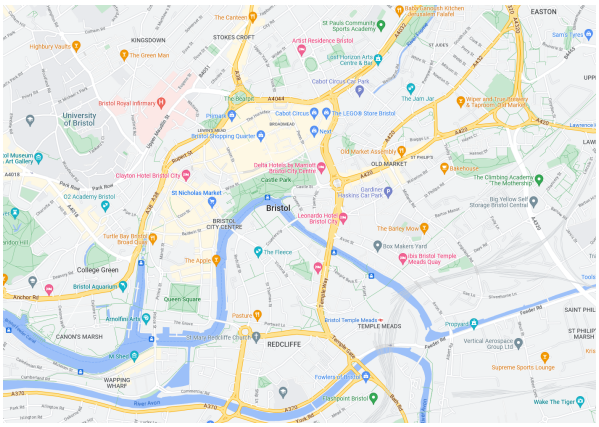
In summary, the project should have prioritized the WFC algorithm more, as time was spent on features that were not discussed in this report and did not contribute to the final project (L-System road network, splines, hexagon-based grids). While the parks in the generated city are well-detailed, the buildings could benefit from more detail, as mentioned in the evaluation. The road network is complete and believable, with various road details such as intersections and crossings included. The tile sets used for generating the city are easily expandable to include more road and building types, and the districts within the city can be expanded by adding new scriptable objects and tweaking their parameters.

(a)



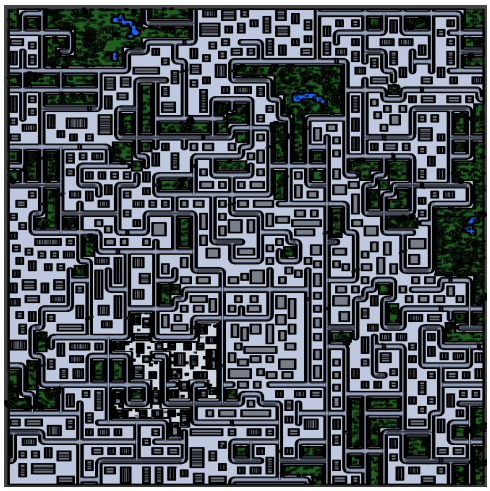
(b)

**Figure 6:** *Visually comparing real-life residential housing (6a) to the generated housing (6b).*



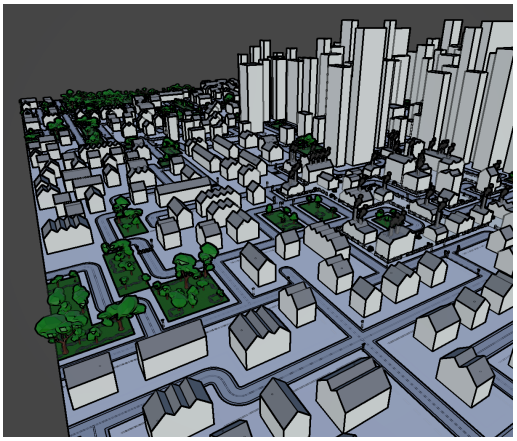

(a)

(b)



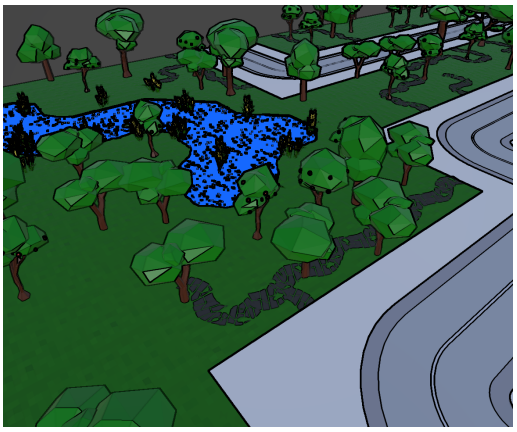
(c)

**Figure 7:** *Visually comparing the generated road network (7c) with the network from Bristol, UK (7a) and New York, USA (7b). Figures 7a & 7b provided by Google Maps.*

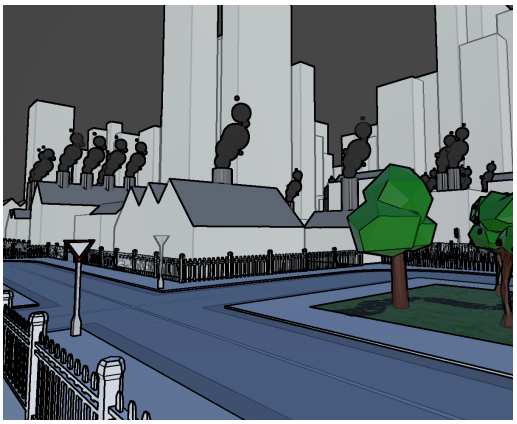


**(a)** *Generated city.*

**(b)** *Traffic light intersection and zebra crossing.*



**(c)** *Park district with paths, pond and props.*



**(d)** *Industrial district, with highrise in the distance.*

**Figure 8:** *Final generated city screenshots.*

# References


Barrett, Sean (n.d.). *L-Systems Considered Harmful*. URL: https://nothings.org/gamedev/l_systems.html. (accessed: 02.01.2023).

Golding, James (2010). *GDC - Building Blocks: Artist Driven Procedural Buildings*. URL: https://www.gdcvault.com/play/1012219/Building-Blocks-Artist-Driven-Procedural. (accessed: 02.01.2023).

Gumin, Maxim (2016). *GitHub - Wave Function Collapse*. URL: https://github.com/mxgmn/wavefunctioncollapse. (accessed: 02.01.2023).

Lagae, Ares and Philip Dutré (2005). "A procedural object distribution function". In: *ACM transactions on graphics (TOG)* 24.4, pp. 1442–1461.

– (2006). "Poisson sphere distributions". In: *Vision, Modeling, and Visualization*. Akademische Verlagsgesellschaft Aka GmbH, pp. 373–379.

Müller, Pascal et al. (2006). "Procedural modeling of buildings". In: *ACM SIGGRAPH 2006 Papers*, pp. 614–623.

Nordvig Møller, Tobias, Jonas Billeskov, and George Palamas (2020). "Expanding wave function collapse with growing grids for procedural map generation". In: *International Conference on the Foundations of Digital Games*, pp. 1–4.

Parish, Yoav IH and Pascal Müller (2001). "Procedural modeling of cities". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 301–308.

prxq (2006). *The Alias Method*. URL: https://prxq.wordpress.com/2006/04/17/the-alias-method/. (accessed: 02.01.2023).

rampion (2017). *Efficient algorithm to randomly select items with frequency*. URL: https://stackoverflow.com/a/873430/13195883. (accessed: 02.01.2023).

Sandhu, Arunpreet, Zeyuan Chen, and Joshua McCoy (2019). "Enhancing wave function collapse with design-level constraints". In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pp. 1–9.

Stålberg, Oskar (2018). *YouTube - EPC2018 - Oskar Stalberg - Wave Function Collapse in Bad North*. URL: https://www.youtube.com/watch?v=0bcZb-SsnrA. (accessed: 02.01.2023).

– (2019). *YouTube - ORGANIC TOWNS FROM SQUARE TILES - a talk by OSKAR STALBERG at INDIECADE EUROPE 2019*. URL: https://www.youtube.com/watch?v=1hqt8JkYRdI. (accessed: 02.01.2023).

– (2021). *YouTube - SGC21- Oskar Stalberg - Beyond Townscapers*. URL: https://www.youtube.com/watch?v=Uxeo9c-PX-w. (accessed: 02.01.2023).

Stålberg, Oskar and Raw Fury (2021). *Townscaper*. URL: https://store.steampowered.com/app/1291340/Townscaper/.

Unity Technologies (n.d.). *GitHub - Unity FBX Exporter Plugin*. URL: https://github.com/Unity-Technologies/com.unity.formats.fbx. (accessed: 02.01.2023).