Desired Path-Dependent Enemy Placement in Stealth Video Games

Ivan Miloslavov

April 16, 2018

Abstract

A common intention amongst game developers is to make the players take a path that traverses the level in a way that the developers intend. This could be used in making a long, compact path in a small level, encourage collectibles discovery, or encourage obtaining stealth or no-kill achievements for the level, such as the Ghost and Shadow achievements in Dishonored[1]. So far, such a desired path could only be forced by static obstacles (such as physical objects or camera-like static enemies) or carefully engineered using game testers. We propose a novel idea that would be able to place dynamic enemies, i.e. guards in strategic locations across the level to permit the proposed path as a safe one, without restricting it to be the only path. This approach relies on two stages: first, the level is divided into zones using circle packing, whose approximate intersection graph is taken; then, a grammar-based approach is used on the graph to create guard routes. The result is analyzed using the GRAM's Stealth Tool[2] to verify that the intended path is in fact taken by theoretical players.

Contents

1 Introduction

A stealth video game offers the player a navigational puzzle: the player must avoid non-player characters, also known as guards, while traversing the level with all its obstacles in order to reach some location. Those guards are modeled as points moving along a closed polygonal chain, or polyline, with a Field of View (FOV) modeled as a circle sector of a set radius and angle. A victory condition in a discrete-step version of the problem ensues from finding a path in both space and time (i.e. a point in space for each time step) such that for each time step, the corresponding position is not inside an obstacle or an enemy's FOV, the distance between two points is upper-bounded by the player's maximum velocity divided by the time step, and the sequence of points starts at the player's start position and ends at the intended destination.

While the original problem relies on finding said path given a full level layout, the game developers face a different challenge in coming up with a layout that allows the existence of such a path. They thus must plan out the enemies in a way that does not completely block the player, but that remains challenging and interesting to the player. Currently, finding a selection of enemies to follow this relies on iterative subjective placement followed by tests using play testers or developed automatic testers such as the GRAM's UnityTool [2]. As such, a tool that could create enemy placements would be convenient; even better so, the tool presented in this paper attempts to do so while taking into account a path intended by the developer to be accessible to the player.

2 Background and Previous Work

Procedural content generation is a very popular technique in modern video games, with applications from Borderlands' equipment and weapons selection to No Man's Sky's entire universe. In general, PCG in games is used throughout the development process as well as the end result; for example, level generation reduces the developer's costs by not requiring as many level designers that would fix the tiny details of level geometry. Also, asset generation such as the work of the company Speedtree that was used in Witcher 3 and the Star Wars movies provides an easy interface for developers to place trees in their simulated environments.[3]

Some research already exists on PCG within the video game development setting, such as Qihan Xu's approach to guard and camera placement[5]. While not path-dependent, it serves as an introduction to the topic of pro-

Figure 1: Randomly generated flora as assets for the Witcher 3 video game[4]

cedurally generated enemies. Two entirely different approaches for guard generation were proposed in that paper. The first one produced single segment paths that would be created inside each region of a Voronoi tessellation, a geometrical partitioning of the surface based on proximity to a set of node points. As such, this approach lacked diversity by making all paths straight, as well as made enemies sparse and unable to interact with each other, as exactly one would be located in each partitioned region of the surface. The second proposal relied on a grammar system, which replaced with multiple iterations parts of the path - such as some stationary waypoint - by a more complicated path - such as a small, back and forth detour. This approach gave more interesting enemy paths, but did not have a guarantee of covering the level with enemies nor a guarantee on their lengths, as all nodes were random points on the walkable surface.

Both it and the present paper extend the existing Stealth tool, an API for developers on the Unity game engine developed by McGill University's GRAM lab[2]. It provides a testing framework that can simulate players navigating a level populated with enemies using a Rapidly exploring Random Tree, which is a tree whose nodes are random points in the spacetime of the level progression, created randomly in space and always increasing in time.

To analyze results, a comparison metric is required between the intended and generated paths. This experiment uses the Frechet distance [6] as the metric between the polygonal curves, or line strings, that both types of paths are. However, the original algorithm was not found to be precise enough. An edge case that makes this algorithm fail any approximation factor can be created by drawing an isosceles triangle with a base length of a and a height of ϵ, then using the base as one path and the two congruent segments as the second path. The true Frechet distance of this setup is of course ϵ, but the algorithm described in the paper would return $\sqrt{a^2 + \epsilon^2}$. As such, a preprocessing step that adds vertices onto segments closest to verticies of the other path allows the precise value to be computed for line strings.

3 Methodology

Since enemy FOV's are represented as circular sectors in this model of the stealth video game, a geometrical representation of the level's accessible surface was generated using circle packing. Given such a packing, each circle's center represents a potential vertex of an enemy's path, with segments between different circle centers being edges in the enemy's path. While the concept of intersection graphs exists for circle packings, we do not necessarily obtain perfect circle packings with touching circles using a greedy algorithm; as such, a pseudo-intersection graph is taken from a given circle packing by drawing edges between any two circle centers that can "see" each other.

The outline of the generation process is as follows. First, the intended level's geometry as well as the intended path is extracted as two-dimensional geometrical data from the Unity editor. Then, a Python script imports the data and generates a circle packing pseudo-intersection graph. Another Python script then generates enemies over the graph and save their paths as two-dimensional geometrical data. Finally, the Unity editor imports the results and places the enemies per the specification.

An overview of the methodology flow is presented as a flowchart in Appendix 1[Figure 6].

3.1 Level geometry extraction

Since the core part of the project was written in Python, an import-export system within Unity had to be created to accomodate it. In particular, the Geometry Recovery Toolkit[7] was partially used for this task. It allows to create a tree of two-dimensional boundaries representing the level bounds as well as its obstacles and any further nested polygons. To fit with the GeoJSON format, the resulting tree is parsed by taking every polygon that represents the true level - whether the level's exterior or any holes within it, which can be selected by the Unity user and saved under the MultiPolygon GeoJSON type, which represents a list of holed polygons. In addition to that, a user-defined path is also extracted into a GeoJSON format under the LineString type.

3.2 Graph generation

A common stage to any path within a particular level is preprocessing the level's accessible terrain to create the graph on which the next stage will rely. Two distinct methods were devised for the project, of which one ultimately trumped the other. Both relied on an observation of the long-term enemy field of view (FOV). Since the FOV tends to have a particular maximum view distance, a circle with that distance as the radius represents the potentially visible field for a stationary, rotating enemy. However, while the enemy moves, the effective view can be modeled by lower-radius circles, with lower sizes representing the navigation speed of the enemy, as at higher speed the edges of the FOV cone were seen only for fractions of seconds. A certain lower bound could be imposed as the "passive perception" radius, one at which the enemy will detect the player no matter the direction. As such, the problem was reduced to a complex circle packing problem, with the intent of retracting a pseudo-intersection graph.

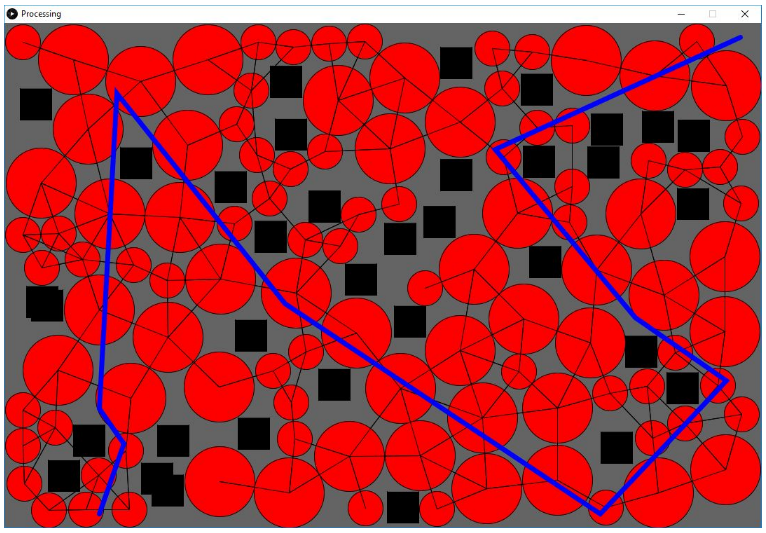

Figure 2: Draft of the NG approach on a random level

3.2.1 Neighbourhood growth approach

The first method relies on a desired guarantee of graph connectivity for the next stage. This algorithm starts by placing a fixed number of seed circles at random location in the level. Once they are generated, each new circle is generated close to some previous circle by looking for the available space to place the center of the new circle a radius r away from the former circle. This available space would be a set of intervals obtained by removing all points r or closer to an obstacle, level bound or another circle. Of them, a random point was selected which served as the center of the new circle.

This was a custom-devised method that was not relying on any other software. It also allowed exact circle calculations and relatively fast execution. However, some drawbacks were in the requirement of a number of seed circles, the inability to precisely find the set of points r and closer to a concave polygon, as well as loss of precision and speed when allowing a range of circle radii instead of only the two extreme radii.

3.2.2 Remainder boundary approach

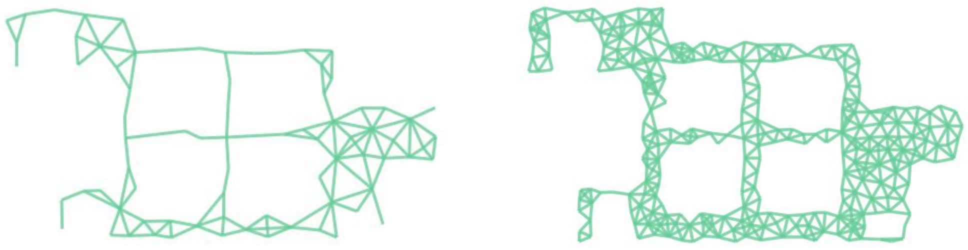

Figure 3: Different resolutions of the RB approach on the Metal Gear Solid Docks level

One of the research projects using the GRAM Stealth Tool has referenced the usage of geometrical algorithms in Python[8]. In particular, the project was using the Shapely package[9], a library of functions on polygons and 2D geometry. As such, the use of Shapely provided a new approach based on the iterative remaining level area.

At each iteration, a random radius r from the allowed range is picked. The boundary of the remaining level area (after removing obstacles and previously placed circles) would then be extended said radius away to create a new boundary on which all points are exactly r away from all obstacles, previous circles and level bounds. If this boundary is non-empty, a random point from it becomes the center of a new circle with radius r. Else, the range is reduced to make r the maximum radius allowed for following circles. If the range becomes too small, the algorithm terminates.

Since this method uses an external package to its extents, it ends up being slower and requires additional installation but provides more robust and extendable functionality with minimal (and thus, more likely to be correct) custom code. This was the selected method for proceeding.

3.3 Heuristics-based route generation

The biggest question of the project is the generation of the enemies given a graph representing the level structure. Due to the complexity and certain subjectivity as to what set of enemy paths constitutes an interesting layout for the player, a heuristics approach was selected to generate the paths. In particular, the enemy paths are generated as random closed walks on a weighted undirected graph. Iteratively, a vertex v is selected from the neighbourhood of the previous vertex u in the walk W such that

$$\sum_{e \in W} w(e) + w((u,v)) + d_G(s,v) \leq C,$$

where s is the starting vertex and C, the maximum cost. This ensures that the walk is closed and always under the maximum cost. Only after generating the full path that its total cost is compared to the minimum cost, discarding the path if it does not reach it.

The heuristic function selected was based on two factors, which are the Euclidean distance between the nodes as well as the Euclidean distance of the edge to the path. While the first factor provides a direct and deterministic cost value, the cost based on the distance is more complicated. Increasing significantly the cost of edges close to or crossing the path would enforce a guarantee of having the requested path free of guards, but not increasing it enough could potentially create a single-edge path for a guard effectively blocking the path.

3.4 Finalizing

Once the paths have been created, they are placed into a file that the Stealth Tool[2] can load to place enemies in the level and test the results using the RRT/Mapper scripts devised by the GRAM. The resulting paths - if any are in fact found - can then be analyzed to modify the arguments of the route generation and fuel successive attempts. That analysis comes from compiling the Frechet distance between the generated and the intended routes, selecting a few whose distances fall under a certain set threshold and saving them separately. As such, quantifying the fraction of those close paths to all generated paths, as well as their individual Frechet distances gives us specific results for the experiment.

4 Experiments and Results

The experiments were conducted by repeatedly creating enemies using the algorithm described above, then generating player paths using the GRAM's Unity tool, and compiling the Frechet distances between those paths and the intended one. Given the lack of an automated, or batch, testing framework as well as time limitations, the results were gathered by fixing all but one variable and testing different values of the remaining variable, for different variables.

Most of the testing was executed on the Smith corridor level[10]. The level is a straight corridor with alternating alcoves; it has been chosen as it was shown in the Unity Tool's presentation paper that the players do not pass through the alcoves with the single straight-path guard, and thus a set of guards forcing the players to pass through the alcoves is an important result to obtain.

The original values were picked randomly until a decent solution was found. As such, most of the Unity tool's settings were left as defaults, except for fixing the number of attempts to 10000, the number of iterations to 60, and fixing the random seed 9481.

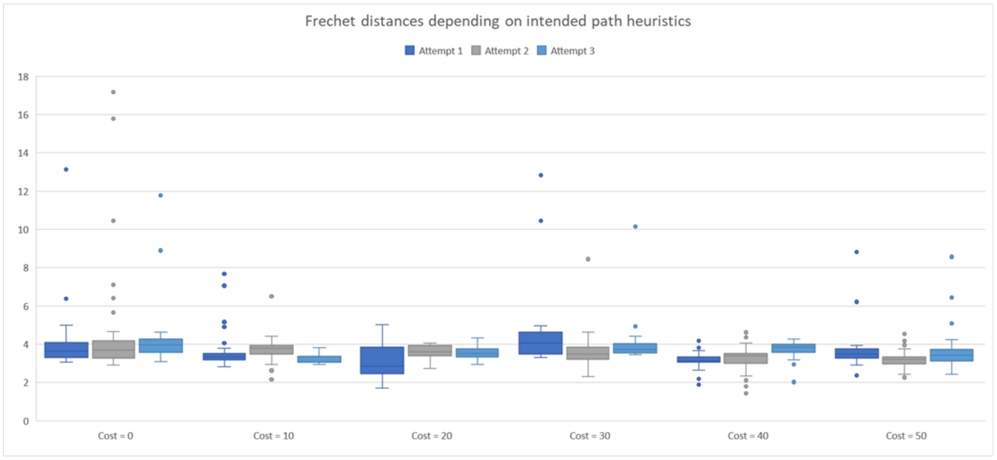

Figure 4: Comparison of path heuristic costs on the Corridor level

As one of the most important variables in the algorithm, the heuristic cost to pass close to the intended path was one of the variables tested[Figure 4]. Intuitively, an extremely high cost would forbid the guards to pass through the intended path. Given enough guards or long enough paths, the remainder of the level is covered and the players are forced onto the intended path. A low, but non-zero cost however could be an issue given the randomness of placing enemies. If the random walk goes along or across the intended path with a segment cost approximately equal to half the maximum path cost, the only way that enemy can complete its cyclic behaviour is by coming back on that same segment. This makes an enemy fixed on top of the intended path, which goes against the wanted result of allowing players to take the intended path exactly.

The number of guards can further restrict the players' movement on the level, but is also subject to a certain randomness[Figure 5]. From the experiments taken, we can see that a low number of guards can indeed give paths that are very far from the intended path. However, a high number of guards does not necessarily raise the coverage of the level by enemies, leading to similar results with a slightly lower number of guards. As such, a heuristic cost can be added to each guard's walk for ensuing ones, to distribute the guards further across the level, but it is not very effective in very granular packings. In those, each node may have a higher number of neighbors, and as such a higher number of incident edges; adding guard costs to some edges still leaves multiple ones for another guard to take.

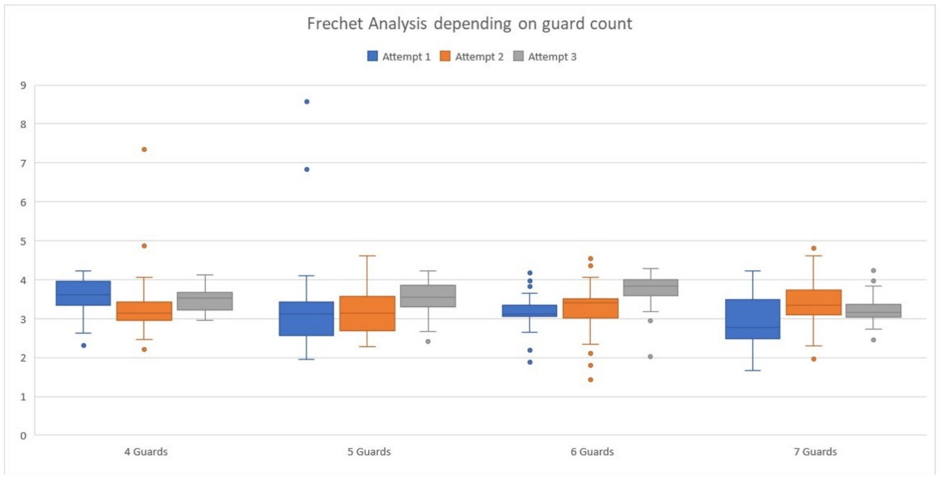

Figure 5: Comparison of the number of guards on the Corridor level

5 Conclusion

This paper presented a solution for the problem of procedural generation of enemies in a stealth video game design setting. In particular, an algorithm intended for 2D levels was devised and written as a Python addon to the GRAM's Unity Stealth tool, then tested with said tool. The algorithm generates a circle packing over the level's surface, then uses a pseudo-intersection graph of those circles to create enemy paths from random weighted walks. Analysis of the Frechet distances between the intended path and the generated paths revealed that the current algorithm is perhaps too random, as it treats each enemy independently and thus might not cover the level entirely, even when introducing additional heuristic costs to disperse the enemies across the level. However, given a strong guarantee of lacking enemies on the intended path (by increasing the cost of enemy walks through the intended path to a high value), the players can in fact both pass close to the intended path and find paths of their own.

A lot of optimization can still be beneficial for this exact approach. In particular, normalizing all variables with respect to either the level surface or the surface gathered by the circle packing may lead to more intuitive variable selection, as well as a higher probability that a set of variables optimized for some level can give good solutions for a level with less or more surface. Linking the enemies' FOV to the size of the maximum circle was motivated by the fact that an enemy would then definitely see its sector of any circle in the packing; however, the FOV angle was not taken into account and thus results may vary greatly for narrow or wide FOV angles. Additionally, computing added guard heuristic costs in the same way as the intended path heuristic costs - that is, using a linear dropoff depending on the distance of a graph segment to the path in question - could also benefit the distribution of guards across the level.

Future work for the first stage may include trying other surface partitioning algorithms, like Voronoi tesselations or generalized circle packing, or perhaps an entirely different approach that creates a (not necessarily planar) graph covering the level's accessible surface. Furthermore, grammar-based approaches to encourage particular guard behaviours could be researched to develop the enemy generation stage further. Finally, the addition of enemy archetypes or unique enemies that would influence the size of their FOV, addition of combat encounters and the application of this approach on surfaces of a 3D level are research topics that would lead to a more enjoyable experience as a player.

References

[1] Bethesda Softworks. Dishonored, 2012.

[2] Jonathan Tremblay, Pedro Andrade Torres, Nir Rikovitch, and Clark Verbrugge. An exploration tool for predicting stealthy behaviour. In *The Second Workshop on Artificial Intelligence in the Game Design Process*, 2013.

[3] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016.

[4] CD Project Red. The Witcher 3: Wild Hunt, 2015.

[5] Qihan Xu. Procedural guard placement and behaviour generation. Master's thesis, McGill University, Montréal, Canada, April 2015.

[6] Thomas Eiter and Heikki Mannila. Computing discrete Frechet distance. 05 1994.

[7] Luke Jones. GRTK : Geometry Recovery toolkit, 2017.

[8] Muntasir Chowdhury and Clark Verbrugge. Exhaustive exploration strategies for NPCs. In *Proceedings of the 1st International Joint Conference of DiGRA and FDG: 7th Workshop on Procedural Content Generation*, August 2016.

[9] Sean Gillies. Shapely - Python software library, January 2018.

[10] Randy Smith. Level building for stealth gameplay. In *Game Developer Conference*, 2006.

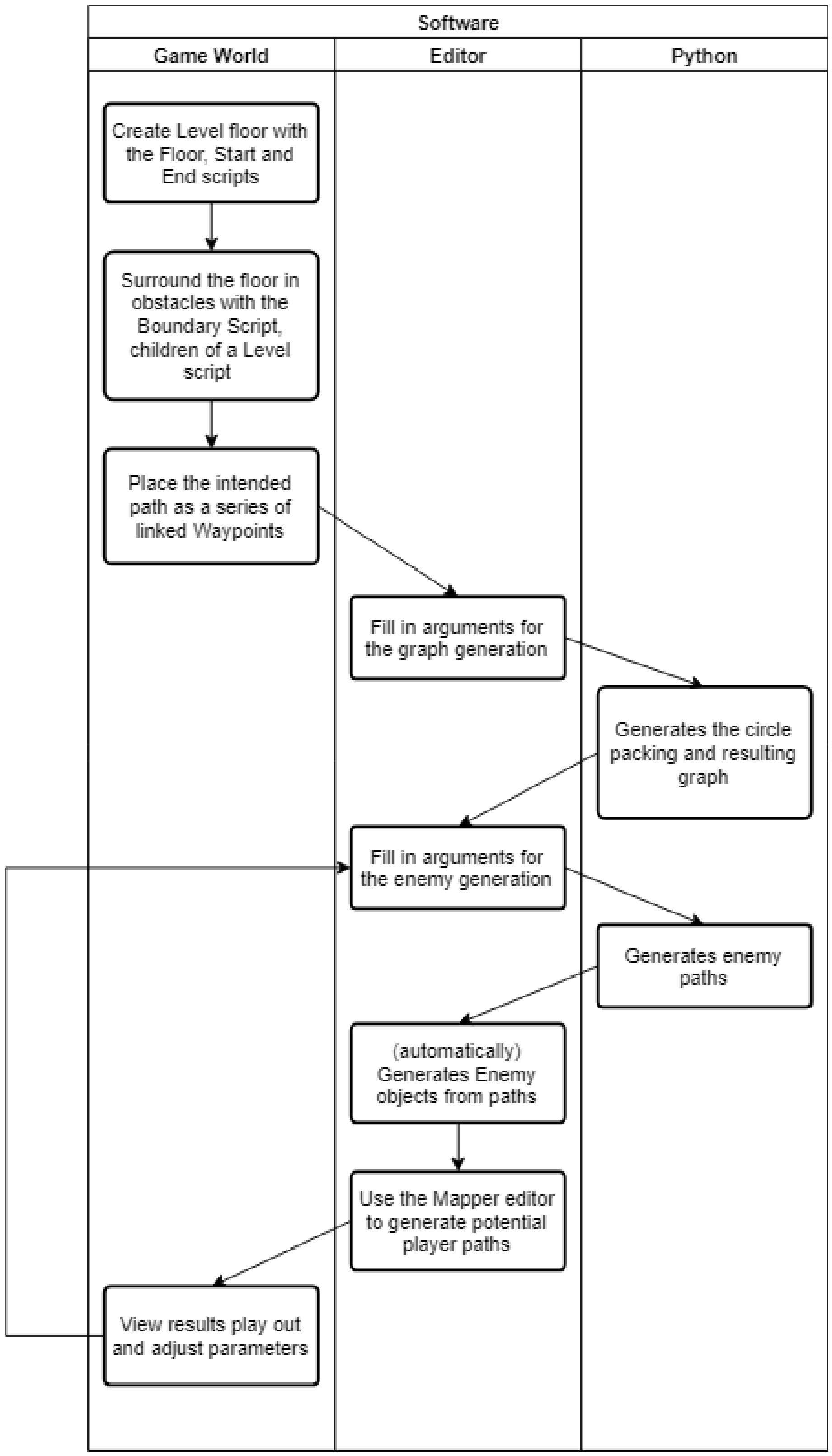
Software
Game World
Editor
Python
Create Level floor with the Floor, Start and End scripts
Surround the floor in obstacles with the Boundary Script, children of a Level script
Place the intended path as a series of linked Waypoints
Fill in arguments for the graph generation
Generates the circle packing and resulting graph
Fill in arguments for the enemy generation
Generates enemy paths
(automatically) Generates Enemy objects from paths
Use the Mapper editor to generate potential player paths
View results play out and adjust parameters

Figure 6: Methodology flow