This document is the logbook for James Moran’s FMP. This logbook details key references to sources of information, key ideas drawn from these sources, technical and design information, discussions with my project supervisor (and any other parties involved in the project), preliminary thoughts and ideas, list of work to complete, along with plans, development of criteria for choices to be made, plus metrics for project tasks and results of technical proving exercises, experiments or technical/product evaluations.

James Moran

James Moran – FMP – Logbook

Contents

[Considered Methods for Balancing the Space Filling Algorithm 2](#_Toc511650008)

[Setting-Up a Plugin in Unreal Engine 4 (UE4) 4](#_Toc511650009)

[Balanced FPS Level Generation System 6](#_Toc511650010)

[First Row 9](#_Toc511650011)

[Second Row 10](#_Toc511650012)

[Third Row 10](#_Toc511650013)

[Screenshots from the Implementation 11](#_Toc511650014)

[Improvements to the First Prototype 12](#_Toc511650015)

[Level Generation Heuristics: First Phase 12](#_Toc511650016)

[First Phase Conclusions 14](#_Toc511650017)

[First Row 16](#_Toc511650018)

[Second Row 16](#_Toc511650019)

[Third Row 17](#_Toc511650020)

[Adding New Wang Tiles to the Set: Second Phase 18](#_Toc511650021)

[Second Phase Conclusions 20](#_Toc511650022)

[Phase Three: Balancing the Placement of Zones 21](#_Toc511650023)

[Considering Zone Defensiveness, Flanking and Dispersion Coefficients 21](#_Toc511650024)

[Software Development Analysis of Classes for the Method Detailed In: ‘Procedural Generation of Balanced Levels for a 3D Paintball Game’ 22](#_Toc511650025)

[Appendix A: Code Snippets 23](#_Toc511650026)

[Appendix B: Past Coefficient Descriptions 27](#_Toc511650027)

[Bibliography 28](#_Toc511650028)

[References 29](#_Toc511650029)

# Considered Methods for Balancing the Space Filling Algorithm

First, looking at a paper, entitled ‘Procedural Generation of Balanced Levels for a 3D Paintball Game’. (Raul Lara-Cabrera et al, 2017)

In this paper, the idea of using zones, with edges between the zones (up to a certain threshold) will be utilised. A vector representation, holding three specific groups of values will be used for each zone. First, the coordinates of the zone, then the density of obstacles in that zone and finally, the density of obstacles for edges, are taken into consideration. (Raul Lara-Cabrera et al, 2017)

In order to maintain a balanced map (level), with no clear advantage for either team, Coefficients for defensiveness, flanking and dispersion are used, along with the mean and Standard Deviation of the defensiveness and flanking values of the zones. (Raul Lara-Cabrera et al, 2017)

The defensiveness of a zone, is determined by the following factors:

* The density of the obstacles within that zone
* The density of the obstacles between that zone and the nearest zones (on the edges)

(Raul Lara-Cabrera et al, 2017)

Considerations for the flanking Coefficient, are shown on the next page.

To alter how the level is generated, the generator will use a mutation-operator. The mutation operator applies pseudo-random permutations to the values of an individual vector (as noted in the 2nd paragraph of this section), adding to or multiplying by a respective pseudo-random value. The decision of adding or multiplying is also decided upon by chance, with the same probability. If an individual vector is mutated to such an extent, that it becomes invalid for the FPSLevelGenerator’s requirements, the algorithm will not consider this vector for the map. The map graph is then recalculated after this mutation, to include new edges between zones, if they become close enough because of it. (Raul Lara-Cabrera et al, 2017)

This ties in with Wang Tiles quite well, as a tile and its edges can represent a zone and its edges. A unit measurement for this project is one Unreal Unit (UU), which equates to 1 metre.

Using the Defensiveness, Flanking and Dispersion values, in an equation, to calculate an ‘Overall Compatibility’ value, comes out as follows:

(Raul Lara-Cabrera et al, 2017)

Where α, β and Γ are the Coefficients for defensiveness, flanking and dispersion respectively. (µd µk) is the mean and (σd, σk) is the Standard Deviation, of the defensive and flanking values respectively. (Raul Lara-Cabrera et al, 2017)

For the defensiveness Coefficient, di, the following equation is used:

(Raul Lara-Cabrera et al, 2017)

Where the density of objects in that zone (Densityi), is calculated with the following equation:

(Raul Lara-Cabrera et al, 2017)

With δi equating to the object count of zone i and δMAX equating to the highest object count. For example, in the Level-Generator, δMAX = 5 (as there are at most, 5 objects in a Zone), then for a given zone (i), there are 2 objects in that zone, so .

For the TotalZoneObjectArea, this is simply the sum of each object’s X-Scale value, multiplied by its Y-Scale value:

. Where ZoneObjectCount is the number of objects (components) present in the Zone, ScaleX is the X-Scale of object-j in the Zone and ScaleY is the Y-Scale of object-j in the Zone. Following through with Zone i having two objects (one with an X-Scale value of 1 and a Y-Scale value of 0.1. Whilst the other has an X-Scale value of 0.1 and a Y-Scale value of 0.9):

For the density of the paths to and from that zone (Densitypaths), the following equation is used:

(Raul Lara-Cabrera et al, 2017)

Where γi represents the degree of node i (the degree is the number of edges connected to each node, (The MathWorks, ©1994-2018) in this case though, the number of adjacent zones will be used instead) and δj represents the density of edge j (for ease of understanding, this is simply the number of edges connected to j, from other zones, not including the ‘virtual’ edges of the Level-Generation Area).

An example run-through of this calculation is as follows (using δMAX equating to 5, as in the above sample calculation and a bottom-left corner Zone being considered as the node):

For the flanking Coefficient (ki), this equation is used:

(Raul Lara-Cabrera et al, 2017)

Where φi is the number of adjacent Zones to the Zone being considered and γi is the number of zones surrounding the Zone being considered (diagonally and horizontally). For example, a zone is in between 8 other zones (with 4 adjacent zones), so . (Raul Lara-Cabrera et al, 2017)

# Setting-Up a Plugin in Unreal Engine 4 (UE4)

To start off, I initiated the creation of a toolbar plugin, providing me with a base plugin, that I could build the Balanced FPS Level Generator on top of.

After attempting to have a sub-menu display to the user, via the button added to the top tool-bar of the editor, I decided it would make more sense to extend the editor’s menu listings (up to the top left corner of the default editor window).

For this, I followed a Live Training video, entitled ‘C++ Extending the Editor’, which has shown me how to add custom menu options, to the menu items list (for the ‘Edit’ sub-menu). (Noland, 2015)

This menu item will be added to the custom ‘Procedural’ category, on the ‘Edit’ sub-menu, as shown below (currently a blank box is shown, as there are no children of the UBaseEditorTool class, to be displayed in this category, from which, the Balanced FPS Level Generator will derive from):



This will then show an Edit Properties dialog for the user to choose constraints/other settings, before initiating the level generation (such as the area to generate a level in and bias of the generation for certain aspects of the level). An example of this is shown below:



This currently produces a structure to encapsulate the level, for the respective dimensions, with a light source (point light), at the centre of this structure. This can be seen below:

Level Generation Bounds Exterior, Unlit:

Level Generation Bounds Interior, Lit:

This provides a platform, to generate the level on top of, which is detailed in the next section.

In forming this encapsulation-object, a source from the UE4 documentation and the UE4 AnswerHub, were used for reference. (Epic Games, 2017), (Digi Labs Dan, 2016)

# Balanced FPS Level Generation System

The following diagram demonstrates the use of these equations:

(MS Paint Pixel-Based Calculations)

749: Depth (80 cm), 551: Width (60 cm)

(Average pixels to cm = ((749 / 80) + (551 / 60)) / 2 = (9.3625 + 9.1833) / 2 = 9.273 (to 3 d. p.).

(308, 21): Centre Edge Point.

Distance ((462, 175), (308, 21)) =

Distance ((220, 379), (308, 21)) =

Distance ((214, 606), (308, 21)) =

Distance ((389, 606), (308, 21)) =

As this value lies between 2500 and 5000, the edge colour for the top edge of this zone, is red.

Another example of its use, is detailed in this section. Note that no matter which algorithm is used to fill the area, the set-up shown immediately below, is always used, before the space is filled with the respective space-filling algorithm:

The width and depth for this area, are expressed by the user (in cm). This will encapsulate the volume to generate a level within (filling space from the ‘Generation Area Floor’, to the default height of the level (100cm or 1UU).

For the use of Wang Tiles, each Wang Tile (Zone class instance) in this implementation, will take up a volume of 1m3 (1x1x1 UU). Specifically, as shown in the diagram immediately to the right, is this set-up for the placement of these Zones (for a 3x3m area (9UU2)):

As there are walls encompassing this area, they are presumed to have a red-edge (matching up to the tiles).

A top-down view (to scale), of all the Wang Tiles in the set, is provided below:

There are 11 Wang Tiles in this set used by the level-generator.

The anchor point of each object on the tile is 5 cm from the edge of that object.

The calculations of the edge-colour, for each of the edges of each wang tile, is provided below (starting at the top left of the above diagram, for WangTile1, moving rightwards for the next Wang Tile, on a row by row basis, with the ranges for the edge-colour as: Where if ): (Omitted) The Geometry for Zones 9 and 11, allow the Player to pass through them, by (from the west or east to the opposite direction, over the obstacles in the Zone).

These Edge colours, are represented on the diagram of Wang Tiles, as per the calculations, as shown below (along with a number to identify each tile, when mentioned in a text passage of this document):

As per the example grid layout (shown earlier in this section), these tiles will now be placed onto the grid, making sure that they match up to the edges correctly.

If there is more than one tile available, in the set of sub-tiles, that match the current grid position, a random number will be used to decide on which tile to use from this sub-set. This has been generated by Random.org. (Random.org, 2018)

Placement of tiles will begin on the lowest row of the grid, in the first column, moving rightwards. The colour of the geometry on the tiles, has been changed to black (to have them stand out).

## First Row

First Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 10.

Second Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 8.

Third Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 4.

The generation area, after selecting tiles for the first row, is shown to the left:

## Second Row

First Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 7. Second Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 4. Third Column: Valid tiles: 5. Tile chosen: 5. The generation area, after selecting tiles for the second row, is shown below:



## Third Row

First Column: Valid Tiles: 2, 7, 8, 9 and 10. Tile Chosen: 7. Second Column: Valid Tiles: 5. Tile Chosen: 5. Third Column: Valid Tiles: 2, 7, 8, 9 and 10. Tile Chosen: 9. The generation area, after selecting tiles for the third column, is shown to the right:

After completing these sample calculations, it has become clear that the equation for determining the edge-density of a Zone’s Edges, provides inconsistent values. This can result in output, as is shown, in the diagram to the right. This shows a level, where the Player is not able to move between all the Zones, no matter where their Spawn Point is placed in the level. Modifications to the Edge Colour calculations, will have to be undertaken to resolve this issue. Heuristics will be considered for this along with reconsidering the tile design (as in this example, the 11th tile will have never been considered, as there is no configuration of tiles that would allow for the placement of it).

# Screenshots from the Implementation

After resolving certain issues with the plugin’s use of Blueprint assets, these screenshots have been taken of the level. These are to scale (from a top-down perspective, with the floor and ceiling of the level generation area, removed so that the tile geometry can be seen):





This shows that the steps for the generation of a level, fall into place, but as this is based off the current set of logical steps, it is still subject to the issues identified in the previous section. Rules will have to be put in place, to have the generator put together a level that allows the Player to move between most of the Zones of the level. After this, factors for balancing the level can be considered.

# Improvements to the First Prototype

After reflecting on the improvements, I could add to the first prototype (as detailed in the previous section), I would want to consider the following phases for the next prototype (in this order):

* Using heuristics to determine the edge-colour of a tile, instead of an equation (to make sure the level is mostly traversable)
* Adding new tiles to the set of tiles to use (at least for copies of the tiles currently present in the set, but at a different orientation)
* Considering the Defensiveness, Flanking and Distribution characteristics of a Zone, along with the Heuristics, to ensure a level is balanced for both teams, as well as mostly traversable

To initiate these improvements, will be that of designing with heuristics (for a traversable level, even though, more than likely, not a balanced level).

## Level Generation Heuristics: First Phase

For these heuristics, I will want them to use the Blue edge-colour, to indicate a solid wall in parallel with that edge, that is the closest object in that Zone, to the respective Zone-Edge. This means that the ‘virtual-edges’ of the ‘virtual-zones’ , for the walls that encapsulate the level-generation area (with the respective north, east, south or west ‘virtual-edge’), will have a blue edge-colour, as well as the edges of any Zone, which completely obstruct Player movement (as in, the Player is not able to leave that Zone via that Edge).

This logic will be applied to the previously, ‘lower’ edge-colours (given the range for edge-density values used to determine the edge-colour), for green equating to an edge with chokepoints (either a narrow opening between two Zones, or an obstacle that must be vaulted over), red equating to edge with a wider opening (but not completely clear, or multiple smaller openings and/or minor obstacles that the Player can walk over, experiencing a minor ascent or descent) and grey equating to an edge with no obstructions to Player movement at all (from this Zone to another).

This is presented as the following sequence (for determining the edge-colour, of any edge):

1. Consider not only the extent of objects that are blocking entry into the Zone via that Edge, but also what percentage of the Zone’s total area, a Player is able to access via that Edge.
2. If entry into the Zone via that edge is either completely blocked (with objects taking-up all the area within 1cm to that Edge), and the Player is only able to access up-to 10% of the Zone’s total area, via that Edge, that Edge-Colour will be Blue.
3. If entry into the Zone via that edge is either only accessible via one chokepoint (between 10 and 20cm in width), and the Player is only able to access up-to 25% of the Zone’s total area, via that Edge, that Edge-Colour will be Green.
4. If entry into the Zone via that edge is either accessible via one entry-point 21-80cm in width, or four or more entry points 5-10cm in width, or a barrier high enough so that the Player can just about jump over it and the Player can access 26-75% of the Zone’s total area, via that Edge, that Edge-Colour will be Red.
5. If entry into the Zone via that edge is either completely open (with no objects taking-up the complete area within 1cm to that Edge (so height and width)), or there is an object with a height half or less than half of a Player’s maximum jump height, and the Player is can access greater than 75% of the Zone’s total area, via that Edge, that Edge-Colour will be Grey.

The above heuristics have been applied to the same set of 11 tiles as before:



This produces the following results:

No calculations have to be performed to determine the Edge colours, as one is simply able to apply the heuristics to tiles, so long as they know the extents of the objects in that zone.

There is also logic for how edge colours match-up to each other, as shown in this table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Edge Colour to Compare** | **Against Blue Edge (%)** | **Against Green Edge (%)** | **Against Red Edge (%)** | **Against Grey Edge (%)** |
| Blue | 85 | 5 | 5 | 5 |
| Green | 5 | 10 | 75 | 10 |
| Red | 5 | 10 | 75 | 10 |
| Grey | 5 | 5 | 85 | 5 |
| Colourless | 5 | 10 | 75 | 10 |

These values have been chosen, so that space to move through the level is maximised (comparisons between a Zone’s edge, to a blank slot in the level-generation area, is presumed to be comparing that edge to a grey edge).

Edit: Now for certain Edge-Colours, there is a specific set of tiles that will match up to that set (as follows, north first, clockwise order):

* Blue, Colourless, Colourless and Blue: 1 and 3
* Blue, Colourless, Colourless and Any Colour: 2, 8 and 9
* Blue, Blue, Colourless and Any Colour: 4 and 11
* Any Colour, Colourless, Blue and Blue: 6 and 8

## First Phase Conclusions

After finishing the implementation of the first of the three phases (for heuristics), it seems as though, either the logic for such has not been followed through it its entirety, or the chance-values, for how the Edges match up against each other, are too lenient.

An example of a generated level using this new system is shown below:



For the sake of implementing the remaining improvements, this phase will be marked as completed for now (subject to change).

Edit: There is a new set of heuristics, for the level-generator to use, to produce levels where Edge or corner spaces, will have a pre-defined set of tiles to choose from (for Edges, this set only has one tile in it). An example level, generated with these new rules, is shown below:

An example of the intended logic, for these heuristics, is shown as a manual calculation (with results), below though:

For these sample calculations, Random.org will be used again and the set of 11 tiles will be used. (Random.org, 2018)

Setup (starting point is the top row, left column):

### First Row

First Column: Target Edge Colours (for this space the chosen edge-colours of this space and all other spaces, the values are shown for the north, east, south and west edges respectively):

Blue, Colourless, Colourless and Blue.

Chosen Edge Colours: Red, Red, Red and Grey. Closest matching tile(s): 7. Tile Chosen: 7.

Second Column: Target Edge Colours: Blue, Colourless, Colourless and Red. Chosen Edge Colours: Blue, Red, Red and Grey. Closest Matching Tile(s): 3. Tile Chosen: 3.

Third Column: Target Edge Colours: Blue, Blue, Colourless and Red. Chosen Edge Colours: Blue, Green, Green and Green. Closest Matching Tile(s): 1, 2, 3, 4, 8, 9 and 11. Tile Chosen: 2.

Results for the first row:

### Second Row

First Column: Target Edge Colours: Red, Colourless, Colourless and Blue. Chosen Edge Colours: Red, Red, Red and Blue. Closest Matching Tile(s): 3, 6 and 7. Tile Chosen: 7.

Second Column: Target Edge Colours: Red, Colourless, Colourless and Red. Chosen Edge Colours: Red, Red, Green and Red. Closest Matching Tile(s): 7. Tile Chosen: 7.

Third Column: Target Edge Colours: Blue, Blue, Colourless and Red. Chosen Edge Colours: Blue, Blue, Red and Grey. Closest Matching Tile(s): 4 and 11. Tile Chosen: 4.

Results for the second row:

### Third Row

First Column: Target Edge Colours: Red, Colourless, Colourless and Blue. Chosen Edge Colours: Red, Red, Blue and Grey. Closest Matching Tile(s): 6. Tile Chosen: 6.

Second Column: Target Edge Colours: Red, Colourless, Blue and Red. Chosen Edge Colours: Blue, Red, Blue and Blue. Closest Matching Tile(s): 1. Tile Chosen: 1.

Third Column: Target Edge Colours: Red, Blue, Blue and Red. Chosen Edge Colours: Red, Blue, Blue and Red. Closest Matching Tile(s): 5. Tile Chosen: 5.

Final Results, with green arrows showing the accessibility between tiles (for how many tiles one can access from one end of the line, to the other). Lowest accessibility value is 2 and the greatest is 6:

These results show that with some tweaking, the generator will get to a point quite soon, where it is able to generate a level where Players can traverse most of the level (irrespective of balance). Then with the aspects for a balanced level being taken into consideration, this should allow for a balanced level, that Players are able to traverse most of.

Adding New Wang Tiles to the Set: Second Phase

For this phase, 7 new Wang Tiles have been added to the set, for alternate orientations of existing Wang Tiles in the set (to account for as many possibilities in traversal that have been foreseen).

The set of Wang Tiles (Zones) is now as follows:

With the edge colours as so:

The process of implementing the Blueprints for each of the new Wang Tiles, has been successful and the new Wang Tiles are part of the set of Wang Tiles, that the system can pick from. Eighteen Tiles are in this new set of Tiles.

Edit: Four new Wang Tiles have been added to the set (as shown below), for wall-tiles lining-up with the Edges of the level-generation area. This increases the size of the set, to twenty-two Wang Tiles:

With Edge-Colours:

## Second Phase Conclusions

The second phase of the improvements, has turned out as planned, putting together new Wang Tiles for the set of Wang Tiles, to accommodate for Zones at a different orientation to some of the pre-existing zones. Even so, additional Zones for some ‘Filler’ Wang Tiles, could be added to the set, for pieces of a level that currently have no representation and should be considered (such as single wall-piece Zones, for each of the Edges of a Zone, with a wall-piece at the north, east, south and west Edges).

These additional Wang Tiles can be added to the set of Wang Tiles, in forthcoming updates to the Wang Tiles set, when such tiles are deemed necessary in the set.

# Phase Three: Balancing the Placement of Zones

## Considering Zone Defensiveness, Flanking and Dispersion Coefficients

The equations listed in ‘Considered Methods for Balancing the Space Filling Algorithm’, will be used for this (as described in that section).

For the defensiveness Coefficient (di): .

For the density of objects in the zone (Densityi): .

For the density of the paths to and from the Zone (Densitypaths):

For the Flanking Coefficient:

In that section, Dispersion was not described though, so that will be described here as the median value of the variance and Standard Deviation (σ), for the volume of the objects in the zone. For variance and Standard Deviation, the following equation will be used: . Where xr is an object’s volume, µ is the Zone’s mean object volume and n is the number of objects in the Zone. If there are n objects in the Zone, with all objects having the same volume, the Dispersion will be 1/n. (Revision Maths, 2018)

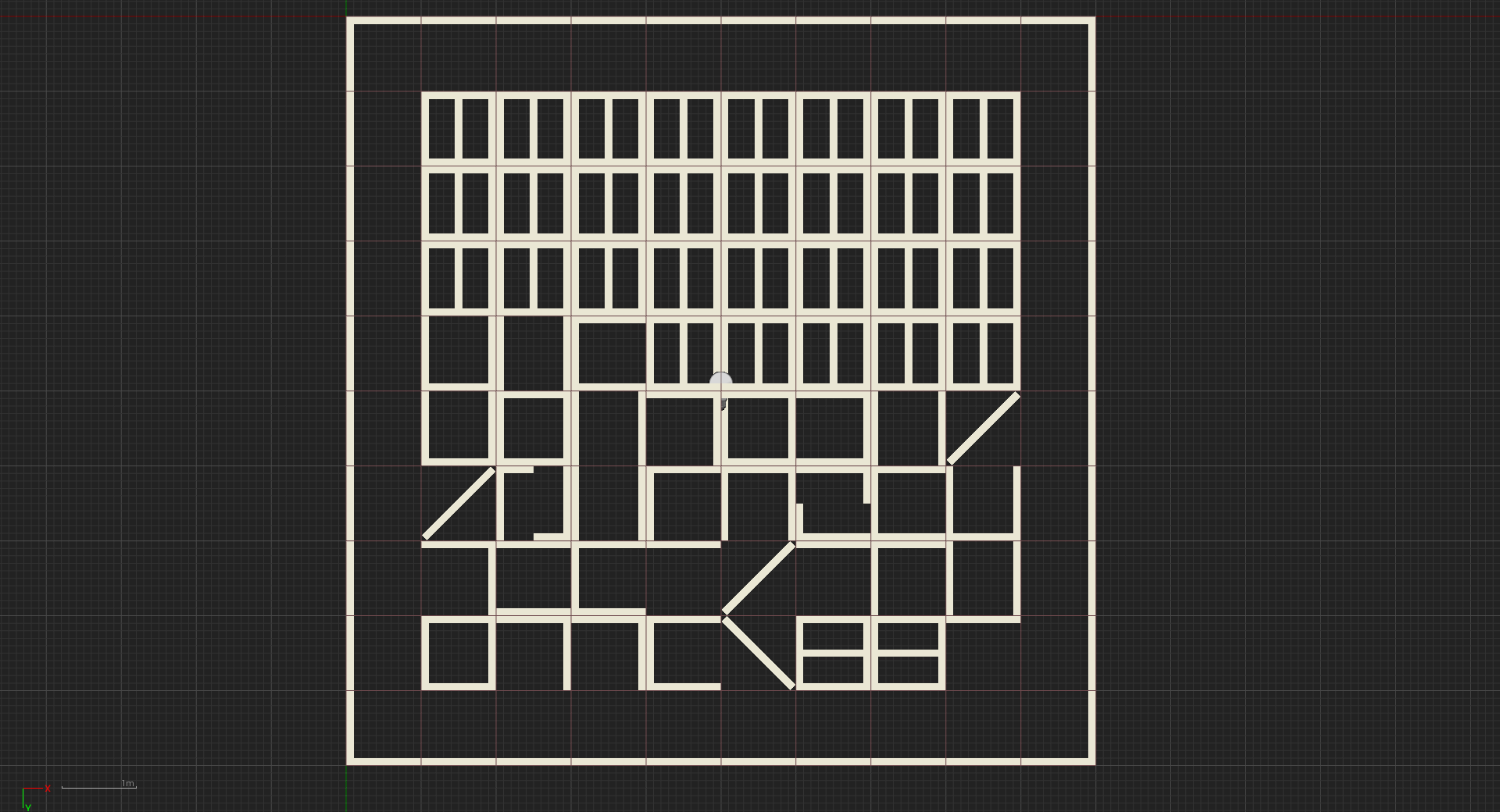
An example calculation (for a zone consisting of 3 objects, with volumes (m3) of 0.09, 0.09 and 0.1), is given below: . Then as the Dispersion is the median of the Variance and the Standard Deviation (σ), . This value will be multiplied by a factor (with a default value of 100), to bring it to a similar level as the defensiveness and flanking values, so the Dispersion in this case, now equals 0.75 (to 2 d. p.). If the Dispersion was 0.013 (to 3 d. p.), then it would be multiplied by 10, instead.

The Defensiveness and Flanking Coefficients can only be determined when the Zone is placed, but the Dispersion Coefficient can be determined before placement of the Zone (knowing which Zone the Dispersion Coefficient is being calculated for).

## Comparison Between the Coefficients of Adjacent Zones

This will start with comparisons between the Dispersion Coefficients of the Zones, following this set of rules (first to last):

1. If the placed Zone is one that fits for a corner or edge of the level-generation area, find a Zone to place, that has the lowest possible Dispersion Coefficient.
2. If the placed Zone has a Dispersion Coefficient of exactly 0.5 find an appropriate Zone for to put next to the placed Zone, given that this Zone will be either WangTile2 or WangTile10.
3. If the placed Zone is not a corner, or has a Dispersion Coefficient that is not 0.5, find a Zone that will have a lower or higher Defensiveness Coefficient, depending on whether the placed Zone’s Defensiveness Coefficient is higher or lower than a threshold for balancing the area taken-up, by objects in that Zone.

An example of a level generated, considering the Coefficients, is shown below:

Although, the tool would not consider both Zones (to the west and south of the current position for Zone placement), in Zone placement. An attempt will be made, of implementing functionality in the tool, so that it will consider the Zones that have been placed to the south and west of the current placement position (during generation of the level).

Software Development Analysis of Classes for the Method Detailed In: ‘Procedural Generation of Balanced Levels for a 3D Paintball Game’

From this considered method for balancing the space filling algorithm, the following classes can be discerned from it:

* Area (with position coordinates, extents and area object density)
* Zone (inheriting from Area, including the zone’s edges)
* Edge (that of the connection point to and from zones, that are close enough to each other)

These can be represented in a Class Diagram (as shown below):



Edit: There is no requirement for a dedicated Edge and Area class, so the Class Diagram simply becomes as follows:

# Appendix A: Code Snippets

The implementation for choosing an appropriate corner tile, for the corners of the level generation area:

// Choose from Zones 1, 3 or 14:

if (CurrentPlacementPosition == TopLeftCorner)

{

RandomDistribution = std::uniform\_int\_distribution<int>(0, 2);

RNG.seed(time(NULL));

int RandomChoice = RandomDistribution(RNG);

std::vector<int> ZoneIndices = { ZONE\_ONE\_INDEX, ZONE\_THREE\_INDEX,

ZONE\_FOURTEEN\_INDEX };

ZoneChoice = ZoneIndices[RandomChoice];

PlacementInCornerOrAlongEdge = true;

}

// Choose from Zones 4, 14 or 15:

else if (CurrentPlacementPosition == TopRightCorner)

{

RandomDistribution = std::uniform\_int\_distribution<int>(0, 2);

RNG.seed(time(NULL));

int RandomChoice = RandomDistribution(RNG);

std::vector<int> ZoneIndices = { ZONE\_FOUR\_INDEX, ZONE\_FOURTEEN\_INDEX,

ZONE\_FIFTHTEEN\_INDEX };

ZoneChoice = ZoneIndices[RandomChoice];

PlacementInCornerOrAlongEdge = true;

}

// Choose from Zones 5, 15 or 16:

else if (CurrentPlacementPosition == BottomRightCorner)

{

RandomDistribution = std::uniform\_int\_distribution<int>(0, 2);

RNG.seed(time(NULL));

int RandomChoice = RandomDistribution(RNG);

std::vector<int> ZoneIndices = { ZONE\_FIVE\_INDEX, ZONE\_FIFTHTEEN\_INDEX,

ZONE\_SIXTEEN\_INDEX };

ZoneChoice = ZoneIndices[RandomChoice];

PlacementInCornerOrAlongEdge = true;

}

// Choose from Zones 1, 6 or 16:

else if (CurrentPlacementPosition == BottomLeftCorner)

{

RandomDistribution = std::uniform\_int\_distribution<int>(0, 2);

RNG.seed(time(NULL));

int RandomChoice = RandomDistribution(RNG);

std::vector<int> ZoneIndices = { ZONE\_ONE\_INDEX, ZONE\_SIX\_INDEX,

ZONE\_SIXTEEN\_INDEX };

ZoneChoice = ZoneIndices[RandomChoice];

PlacementInCornerOrAlongEdge = true;

}

To only use a certain tile for a corner:

// Zone 3:

if (CurrentPlacementPosition == TopLeftCorner)

{

ZoneChoice = ZONE\_THREE\_INDEX;

PlacementInCornerOrAlongEdge = true;

}

// Zone 4:

else if (CurrentPlacementPosition == TopRightCorner)

{

ZoneChoice = ZONE\_FOUR\_INDEX;

PlacementInCornerOrAlongEdge = true;

}

// Zone 5:

else if (CurrentPlacementPosition == BottomRightCorner)

{

ZoneChoice = ZONE\_FIVE\_INDEX;

PlacementInCornerOrAlongEdge = true;

}

// Zone 6:

else if (CurrentPlacementPosition == BottomLeftCorner)

{

ZoneChoice = ZONE\_SIX\_INDEX;

PlacementInCornerOrAlongEdge = true;

}

For the old horizontal encapsulation of the level generation area (tiles fulfil this encapsulation now):

// Front face:

for (float CurrentXPosition = LevelGenerationStartPoint.X;

CurrentXPosition < LevelExtents.X; CurrentXPosition +=

DEFAULT\_TILE\_WIDTH)

{

FVector CurrentPosition = FVector(CurrentXPosition,

LevelGenerationStartPoint.Y - DEFAULT\_ENCAPSULATION\_OFFSET,

CurrentZPosition - DEFAULT\_ENCAPSULATION\_OFFSET);

FRotator FrontFaceRotation = FRotator::ZeroRotator;

FTransform LevelPanelTransform = FTransform(FrontFaceRotation.Quaternion(), CurrentPosition, DefaultRelativePanelScale);

// \*GEditor->GetEditorWorldContext().ExternalReferences[0] just to get the world context...

WallPanelActor = UGameplayStatics::BeginSpawningActorFromBlueprint(

GEditor->GetEditorWorldContext().World()->GetCurrentLevel(),

WallPanelBlueprintAsset, LevelPanelTransform, false);

WallPanelActor->ExecuteConstruction(LevelPanelTransform, nullptr, nullptr, true);

FrontFaceLoopCycleCount++;

//UGameplayStatics::FinishSpawningActor(WallPanelActor, LevelPanelTransform);

}

// Right face:

for (float CurrentYPosition = LevelGenerationStartPoint.Y;

CurrentYPosition < LevelExtents.Y; CurrentYPosition +=

DEFAULT\_TILE\_WIDTH)

{

FVector CurrentPosition = FVector(LevelGenerationStartPoint.X + LevelExtents.X + DEFAULT\_ENCAPSULATION\_OFFSET,

CurrentYPosition, CurrentZPosition - DEFAULT\_ENCAPSULATION\_OFFSET);

FRotator RightFaceRotation = FRotator(0.0f, 90.0f, 0.0f);

FTransform LevelPanelTransform = FTransform(RightFaceRotation.Quaternion(), CurrentPosition, DefaultRelativePanelScale);

// \*GEditor->GetEditorWorldContext().ExternalReferences[0] just to get the world context...

WallPanelActor = UGameplayStatics::BeginSpawningActorFromBlueprint(GEditor->GetEditorWorldContext().World()->GetCurrentLevel(),

WallPanelBlueprintAsset, LevelPanelTransform, false);

WallPanelActor->ExecuteConstruction(LevelPanelTransform, nullptr, nullptr, true);

RightFaceLoopCycleCount++;

}

// Back face:

for (float CurrentXPosition = LevelGenerationStartPoint.X + LevelExtents.X;

CurrentXPosition > LevelGenerationStartPoint.X; CurrentXPosition -=

DEFAULT\_TILE\_WIDTH)

{

FVector CurrentPosition = FVector(CurrentXPosition,

LevelGenerationStartPoint.Y + LevelExtents.Y + DEFAULT\_ENCAPSULATION\_OFFSET, CurrentZPosition - DEFAULT\_ENCAPSULATION\_OFFSET);

FRotator BackFaceRotation = FRotator(0.0f, 180.0f, 0.0f);

FTransform LevelPanelTransform = FTransform(BackFaceRotation.Quaternion(), CurrentPosition, DefaultRelativePanelScale);

// \*GEditor->GetEditorWorldContext().ExternalReferences[0] just to get the world context...

WallPanelActor = UGameplayStatics::BeginSpawningActorFromBlueprint(GEditor->GetEditorWorldContext().World()->GetCurrentLevel(),

WallPanelBlueprintAsset, LevelPanelTransform, false);

WallPanelActor->ExecuteConstruction(LevelPanelTransform, nullptr, nullptr, true);

BackFaceLoopCycleCount++;

}

// Left face:

for (float CurrentYPosition = LevelGenerationStartPoint.Y + LevelExtents.Y;

CurrentYPosition > LevelGenerationStartPoint.Y; CurrentYPosition -=

DEFAULT\_TILE\_WIDTH)

{

FVector CurrentPosition = FVector(LevelGenerationStartPoint.X - DEFAULT\_ENCAPSULATION\_OFFSET, CurrentYPosition, CurrentZPosition - DEFAULT\_ENCAPSULATION\_OFFSET);

FRotator FrontFaceRotation = FRotator(0.0f, -90.0f, 0.0f);

FTransform LevelPanelTransform = FTransform(FrontFaceRotation.Quaternion(), CurrentPosition, DefaultRelativePanelScale);

// \*GEditor->GetEditorWorldContext().ExternalReferences[0] just to get the world context...

WallPanelActor = UGameplayStatics::BeginSpawningActorFromBlueprint(GEditor->GetEditorWorldContext().World()->GetCurrentLevel(), WallPanelBlueprintAsset, LevelPanelTransform, false);

WallPanelActor->ExecuteConstruction(LevelPanelTransform, nullptr, nullptr, true);

LeftFaceLoopCycleCount++;

}

Old matching against system used, when matching against Edge-colours only:

// Not necessary in the latest version, this class will instead determine which Zone to

// use, based on how Edges match up against each other.

// Get a set of Zones that match the TargetEdgeColours...

for (int ZoneSetIterator = 0; ZoneSetIterator < LevelZoneTiles.Num(); ZoneSetIterator++)

{

// For the Edge-colours of this current Zone in the set:

std::vector<FPSLevelGeneratorEdge::EdgeColour> CurrentZoneEdgeColours = LevelZoneTiles[ZoneSetIterator]->GetZoneEdgeColours();

// For validating if a tile matches up to the TargetEdgeColours:

bool ZoneMatchesToTargetEdgeColours = true;

// To check each EdgeColour against CurrentZoneEdgeColours:

for (int TargetEdgeColoursIterator = 0; TargetEdgeColoursIterator < TargetEdgeColours.size() - 1; TargetEdgeColoursIterator++)

{

If(TargetEdgeColours[TargetEdgeColoursIterator] > FPSLevelGeneratorEdge::EdgeColour::Grey - 1)

{

// This CurrentEdgeColour would not match up to TargetEdgeColour, so this zone would not go in the CurrentPlacementPosition. For the meantime, grey is assumed to be colourless (so matching with any colour):

if (!(CurrentZoneEdgeColours[TargetEdgeColoursIterator] == TargetEdgeColours[TargetEdgeColoursIterator]) &&

TargetEdgeColours[TargetEdgeColoursIterator] != FPSLevelGeneratorEdge::EdgeColour::Grey)

{

ZoneMatchesToTargetEdgeColours = false;

break;

}

}

if (ZoneMatchesToTargetEdgeColours)

{

ZoneSubSet.Add(LevelZoneTiles[ZoneSetIterator]);

}

}

// ...then pick one of these Zones from the sub-set:

// But if there are no Zones in the sub-set:

if (ZoneSubSet.Num() == 0)

{

return nullptr;

}

Previous Edge-colour comparison values:

// Previous values:

/\*\* Colourless against another. \*/

//const int COLOURLESS\_TO\_BLUE = 5;

//const int COLOURLESS\_TO\_GREEN = 10;

//const int COLOURLESS\_TO\_RED = 75;

//const int COLOURLESS\_TO\_GREY = 10;

///\*\* Blue against another. \*/

//const int BLUE\_TO\_BLUE = 85;

//const int BLUE\_TO\_GREEN = 5;

//const int BLUE\_TO\_RED = 5;

//const int BLUE\_TO\_GREY = 5;

///\*\* Green against another. \*/

//const int GREEN\_TO\_BLUE = 5;

//const int GREEN\_TO\_GREEN = 10;

//const int GREEN\_TO\_RED = 75;

//const int GREEN\_TO\_GREY = 10;

///\*\* Red against another. \*/

//const int RED\_TO\_BLUE = 5;

//const int RED\_TO\_GREEN = 10;

//const int RED\_TO\_RED = 75;

//const int RED\_TO\_GREY = 10;

///\*\* Grey against another. \*/

//const int GREY\_TO\_BLUE = 5;

//const int GREY\_TO\_GREEN = 5;

//const int GREY\_TO\_RED = 85;

//const int GREY\_TO\_GREY = 5;

For choosing a Zone based on Coefficients:

// The coefficients will be considered here, for the choice of Zone:

int UBalancedFPSLevelGeneratorTool::GetZoneConsideringCoefficients(int ZoneToCompareTo, ZoneAdjacencyDirection PlacedZoneAdjacency)

{

// To Pick from an index-subset on a random basis:

std::default\_random\_engine RNG;

std::uniform\_int\_distribution<int> RandomDistribution(0, 0);

// Seed the RNG before using it:

RNG.seed(time(NULL));

// For the Coefficients to consider:

float ConsideredZoneDefensivenessCoefficient = ZoneSubSet[ZoneToCompareTo]->GetDefensivenessCoefficient();

float ConsideredZoneFlankingCoefficient = ZoneSubSet[ZoneToCompareTo]->GetFlankingCoefficient();

float ConsideredZoneDispersionCoefficient = ZoneSubSet[ZoneToCompareTo]->GetDispersonCoefficient();

for (int ZoneIterator = 0; ZoneIterator < ZoneSubSet.Num() - 1;

ZoneIterator++)

{

// To populate with applicable Zone indices:

std::vector<int> ApplicableZoneIndices;

float CurrentZoneDefensivenessCoefficient = ZoneSubSet[ZoneIterator]->GetDefensivenessCoefficient();

float CurrentZoneFlankingCoefficient = ZoneSubSet[ZoneIterator]->GetFlankingCoefficient();

float CurrentZoneDispersionCoefficient = ZoneSubSet[ZoneIterator]->GetDispersonCoefficient();

// Consider dispersion first (of the Zone already placed in the level):

// A Zone is to be placed in an adjacent position to a corner of the level-generation area,

// or an edge Zone for such:

if (RuleOneIsTrue(CurrentZoneDispersionCoefficient) &&

ZoneIsCornerPiece(CurrentZoneDispersionCoefficient))

{

// Choose a Zone with a lower value than this piece's Dispersion

// Coefficient:

for (int ZoneIterator = 0; ZoneIterator < ZoneSubSet.Num() - 1;

ZoneIterator++)

{

if (ZoneSubSet[ZoneIterator]->GetDispersonCoefficient() <

CORNER\_PIECE\_ZONE\_DISPERSION)

{

ApplicableZoneIndices.push\_back(ZoneIterator);

}

}

// Pick from the index-subset on a random basis:

std::default\_random\_engine RNG;

std::uniform\_int\_distribution<int>

RandomDistribution(0, ApplicableZoneIndices.size() - 1);

// Seed the RNG before using it:

RNG.seed(time(NULL));

// Make a call to 'flush' the stream before using it, as well as seeding it:

RandomDistribution(RNG);

return ApplicableZoneIndices[RandomDistribution(RNG)];

}

if (RuleOneIsTrue(CurrentZoneDispersionCoefficient) &&

ZoneIsEdgePiece(ZoneToCompareTo))

{

// Choose a Zone with a lower value than this piece's Dispersion

// Coefficient:

for (int ZoneIterator = 0; ZoneIterator < ZoneSubSet.Num() - 1;

ZoneIterator++)

{

if (ZoneSubSet[ZoneIterator]->GetDispersonCoefficient() <

CORNER\_PIECE\_ZONE\_DISPERSION)

{

ApplicableZoneIndices.push\_back(ZoneIterator);

}

}

// Pick from the index-subset on a random basis:

std::default\_random\_engine RNG;

std::uniform\_int\_distribution<int>

RandomDistribution(0, ApplicableZoneIndices.size() - 1);

// Seed the RNG before using it:

RNG.seed(time(NULL));

// Make a call to 'flush' the stream before using it, as well as seeding it:

RandomDistribution(RNG);

return ApplicableZoneIndices[RandomDistribution(RNG)];

}

// The placed Zone is WangTile2 or WangTile10:

if (RuleTwoIsTrue(ConsideredZoneDispersionCoefficient) &&

ZoneIsWangTile2(ZoneToCompareTo))

{

// Consider the adjacency:

switch (PlacedZoneAdjacency)

{

// Same for either here (concerning the traversable direction of

// WangTile2):

case ZoneAdjacencyDirection::Westwards:

case ZoneAdjacencyDirection::Southwards:

RandomDistribution = std::uniform\_int\_distribution<int>

(0, APPLICABLE\_ZONE\_INDICES\_FOR\_WANG\_TILE\_2.size() - 1);

// Seed the RNG before using it:

RNG.seed(time(NULL));

// Make a call to 'flush' the stream before using it, as well as seeding it:

RandomDistribution(RNG);

return APPLICABLE\_ZONE\_INDICES\_FOR\_WANG\_TILE\_2[RandomDistribution(RNG)];

break;

default:

break;

}

}

if (RuleTwoIsTrue(ConsideredZoneDispersionCoefficient) &&

ZoneIsWangTile10(ZoneToCompareTo))

{

// Consider the adjacency:

switch (PlacedZoneAdjacency)

{

// Same for either here (concerning the traversable direction of

// WangTile10):

case ZoneAdjacencyDirection::Westwards:

case ZoneAdjacencyDirection::Southwards:

RandomDistribution = std::uniform\_int\_distribution<int>

(0, APPLICABLE\_ZONE\_INDICES\_FOR\_WANG\_TILE\_10.size() - 1);

// Seed the RNG before using it:

RNG.seed(time(NULL));

// Make a call to 'flush' the stream before using it, as well as seeding it:

RandomDistribution(RNG);

return APPLICABLE\_ZONE\_INDICES\_FOR\_WANG\_TILE\_10[RandomDistribution(RNG)];

break;

default:

break;

}

}

}

return 0;

}

# Appendix B: Past Coefficient Descriptions

For the flanking Coefficient (ki), this equation is used:

(Raul Lara-Cabrera et al, 2017)

Where φi is the number of connected zones in the sub graph, created from the adjacent zones to the node being considered, not including that zone itself and γi is the number of zones connected to the node being considered. If this zone has no connected zones, its flanking Coefficient is zero (for example, a zone is in between 5 other zones, with the top 2 zones only being connected to each other and the centre zone, with the bottom 3 zones only being connected to each other and the centre zone. After taking the centre zone out of consideration, there are two sets of connected zones, so ki = 1 – 2/5 = 3/5 = 0.6). (Raul Lara-Cabrera et al, 2017)

For the edges, edge density (δj) is used to represent the colour of an edge. This is calculated from the volume taken up by objects in the zone (in cm3), divided by the mean distance (in cm) of these objects, from the centre point of the edge. These equations are used for such:

δj is then checked against the range of values used to determine the colour of an edge. Where if

# Bibliography

**There are no sources in the current document.**

# References

CABRERA, R., L., *et al*, 2017. *Procedural Generation of Balanced Levels for a 3D Paintball Game*. Madrid: Universidad Autonoma de Madrid

DIGI LABS DAN, 2016. *Create a new Level from C++ code* [Viewed on the 19/03/2018]. Available from: <https://answers.unrealengine.com/questions/440347/create-a-new-level-from-c-code.html>

Epic Games, Inc., ©2004-2017. *UWorld* [Viewed on the 19/03/2018]. Available from: <http://api.unrealengine.com/INT/API/Runtime/Engine/Engine/UWorld/index.html>

NOLAND, M., 2015. C++ Extending the Editor | Live Training | Unreal Engine. [Viewed on the 04/03/2018]. Available from: <https://www.youtube.com/watch?v=zg_VstBxDi8&t=1364s>

RANDOM.ORG, © 1998-2018. *What’s this fuss about true randomness?* [Viewed on the 18/03/2018]. Available from: <https://www.random.org/>

REVISION MATHS, © 2004-2018. Measures of Dispersion [Viewed on the 14/04/2018]. Available from: <https://revisionmaths.com/advanced-level-maths-revision/statistics/measures-dispersion>

THE MATHWORKS, © 1994-2018. *Degree* [viewed on the 05/04/2018]. Available from: <https://uk.mathworks.com/help/matlab/ref/graph.degree.html?s_tid=gn_loc_drop#buofhro-1-D>