Institute of Education

UCL

BA Media Practice-based dissertation 2023-24

Student Name:	Peiying Zhang	**Candidate Number:**	HSWS4
Module Title & Code: ☐ CCME0161: Research Dissertation and Publication - Creative Media Lab III. ☒ CCME0162: Creative Project, Degree Show and Practiced Based Dissertation – Creative Media Lab III.			
Programme Title: BA Media			
Dissertation Title: Voronoi Diagram Based City Generator: A Procedural Generation Approach to Create Adjustable City in Games			
Word Count: 3390 (Excluding abstract, acknowledgements, bibliography/references list, appendices, and cover sheet). Expected word count by module: • CCME0161 – 6000-8000 words. • CCME0162 – 4000 words.			
Media Artefact URL: https://liveuclac-my.sharepoint.com/:f:/g/personal/dtnvpz1_ucl_ac_uk/ElaRbsjM5rZBoj_k88rh9t4BED66gpjMbPBGuYYCufMl8A			
Did you use AI in this assessment? ☐ **Yes** ☒ **No**	*If yes, please provide an acknowledgement statement below with a critical reflection of how it helped you and how it was less helpful (You should also add a footnote on each page for which AI was used saying briefly what you used it for).*		

Student Declaration: ☒ ←Tick this box to agree

By ticking this box, I affirm that the work in this assignment is my own and that any material derived or quoted from the published or unpublished work of other persons has been duly acknowledged. I confirm that I have read the UCL and Departmental guidance on plagiarism.

Student Waiver: ☒ ←Tick this box to agree

By ticking this box, I give permission for my assignment to be used as an exemplar so that future students can use it as a learning tool.

Extenuating Circumstances

Please only complete this section **only** if you have applied for and received an approved EC (extenuating circumstances) from either the deputy programme leader or the EC panel:

I have an approved EC in place for this assignment ☐←Tick this box to agree

Case Reference Number:

New Deadline:

Please note that after the original deadline whatever you submit will be deposited on Turnitin so you cannot submit multiple times. If you want to check your Turnitin score before you submit, you can do this be uploading a 'test' submission on the following Moodle page:

https://moodle.ucl.ac.uk/enrol/index.php?id=34

VORONOI DIAGRAM BASED PROCEDURAL CITY GENERATOR DOCUMENTATION

Creative Project Documentation

JUNE 8, 2024

PEIYING ZHANG
UCL BA MEDIA

PROCEDURAL CITY GENERATOR:

Create a City on Yourself

Scan QR Code to Get Author's Info

Voronoi Diagram Based City Generator: A Procedural Generation Approach to Create Adjustable City in Games

Developed in Unreal Engine 5 and implemented in C++

Table of Contents

Abstract 3
1. Introduction 3
2. Inspiration 4
3. Project's Showcase 6
3.1. Road Network Density 6
3.2. Road Type 8
3.3. Road/Pavement Width 9
3.4. Network Seed 11
3.5. Building Distribution 12
3.6. Highest Building 12
3.7. City Centre 14
3.8. Building's Variance 15
3.9. Building's Seed 16
3.10. Terrain's Seed 18
3.11. Green Area Amount 19
4. Project's Workflow 20
4.1. Terrain Generation 21
4.2. Road Map Generation 26
4.3. Building Block's Generation 35
4.4. Building Generation 39
4.5. Pre-made Infrastructures Generation 43
4.6. UI Design 45
4.7. Viewport Design 46
5. Conclusion 48
6. Appendices 50
7. Acknowledgements 53
8. Bibliography 54

Abstract

This documentation presents the development of Voronoi Diagram Based Procedural City Generator, detailing the creative project's workflow within Unreal Engine 5. The generator utilizes Voronoi diagrams for efficient spatial partitioning, enabling the creation of detailed and customizable city layouts. The core of the documentation showcases various city generator parameters like road network density, road types, building distribution, and height etc., illustrating their direct impact on the procedural generation of cityscapes. Additionally, the project workflow is thoroughly explored, from terrain and road map generation to the creation of buildings and infrastructural elements. Code snippets and explanations provide insight into the underlying algorithms that automate and enhance the city creation process. This procedural approach represents an efficient shift from traditional manual methods, offering a streamlined and flexible tool for city design in game.

1. Introduction

The procedural city generator is created based on the Voronoi diagram, which is a mathematical approach to conduct space partitioning. Voronoi diagram was initially introduced by an imperial Russian mathematician named Georgy Voronoy (Negm, 2021), and firstly applied by Doctor John Show who used a Voronoi diagram to trace the spread of cholera in London's Soho district, demonstrating that the disease was transmitted through water sources (Reid, 2022). The Voronoi Tessellation is mathematical method to searching nearest neighbour, determining vertices, finding maximal empty circles (cells) and path planning (edges) etc. (Ai, Yu, & He, 2015). Using the Voronoi Diagram provides a solid foundation to build the procedural city

generator because of its own graphic structure, and it also promotes the city generator a better adjustment with different parameters. The procedural city generator, performing as a “software” inside Unreal Engine 5, provides an approach for people to generate their own city with customized parameters. To some extent, the procedural generation method addresses the drawbacks of the traditional manual approach to city creation by providing a more efficient, automated way to streamline the process with algorithmic rules and limited human input. The following sections will illustrate how the city generator works with different parameters and explain the process of making the generator.

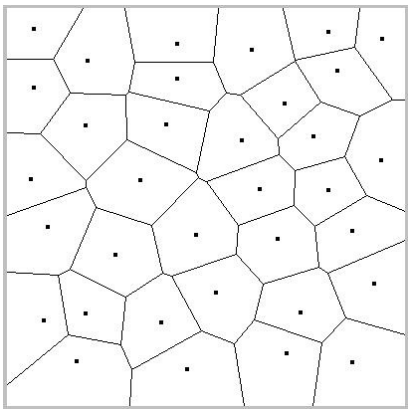


Figure 1: Voronoi Diagram, Spin-heads, 2013

2. Inspiration

The inspiration of the Voronoi Diagram Based Procedural City Generator is two sided. Firstly, it is initially inspired by the Unreal Engine 5 City Sample called "The Matrix Awakens: An Unreal Engine 5 Experience". It is a Houdini-generated city that uses extensive rule processing and procedural generation for city design and construction (Epic Games, n.d., 2022). As a city project, the City Sample is quite graphically intensive that requires a powerful video card to run, which is a not quite suitable for ordinary players who doesn’t have a video card. To solve

this problem, I had a though on make a city generator for games through other method instead of using the layer-based generation logic from Epic Games.

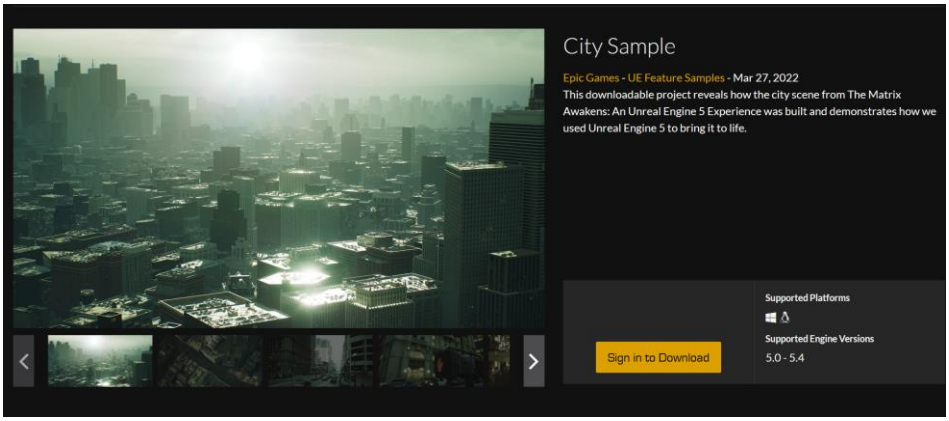


Figure 2: City Sample, Epic Games, n.d., 2022

Secondly, Joseph Chet Redmon's (2011) procedural city generation project gives me the exact approach inspiration to create the procedural city – Voronoi Diagram method. Actually, to create a city, the Voronoi Diagram method is not a new method to employ, but Joseph's project has a great potential to make the city fully adjustable by parameters, including the window size, building's height etc. These elements all give me the direction to create a good adjustable procedural city.

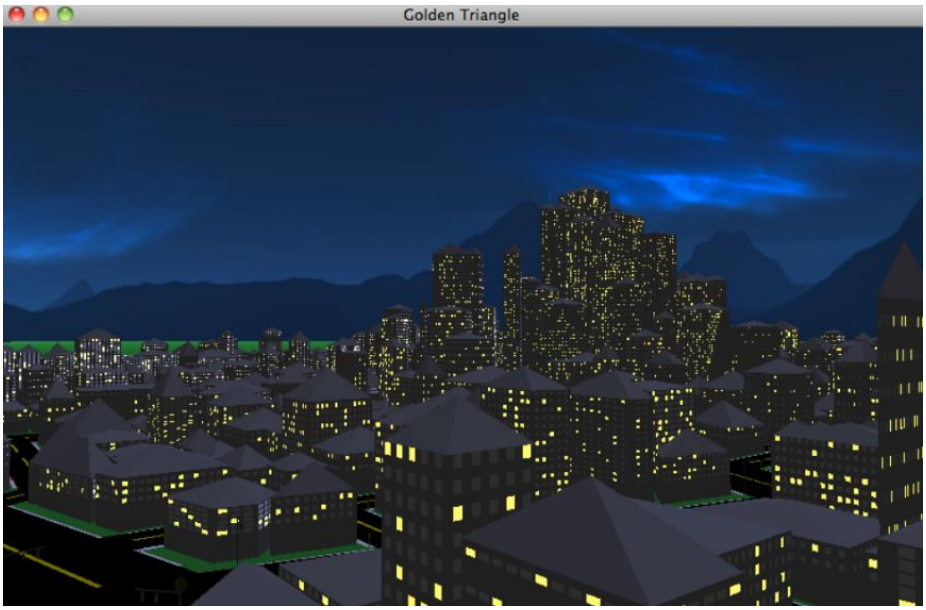


Figure 3: Finished Procedural City Generation Project, Joseph Chet Redmon, 2011

3. Project's Showcase

This section is the showcase of how to create your own city by employing different parameters.

Those parameters will be explained by contrasting the city layout with different values put in.

3.1. Road Network Density

Figure 4: Road Network Density Parameter

1. HIGH

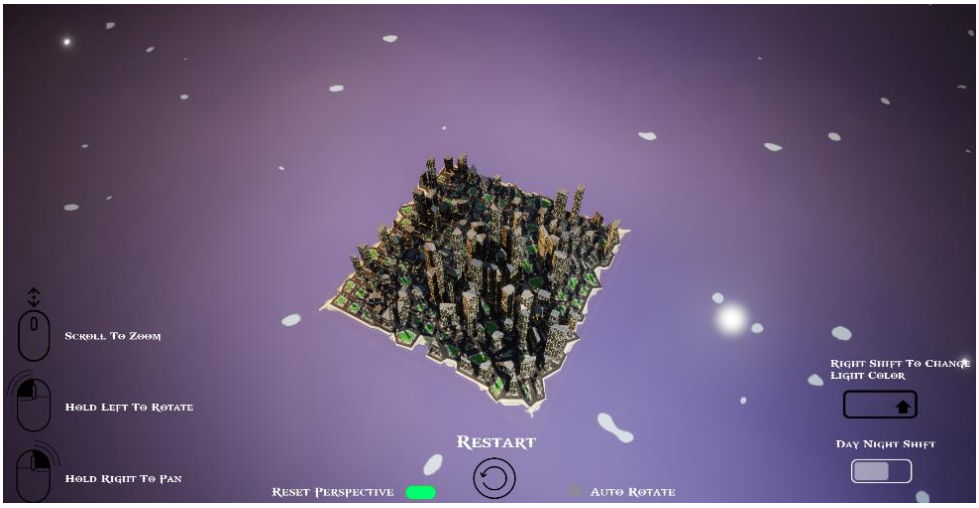


Figure 5: Road Network High Density

2. MEDIUM

Figure 6: Road Network Medium Density

3. LOW

Figure 7: Road Network Low Density

The road network density is different with the parameter input, which is quite visible in the pictures above. The difference is created through assign different cell size for the texture itself, so generating different results.

3.2. Road Type

Figure 8: Road Type Parameter

1. VORONOI

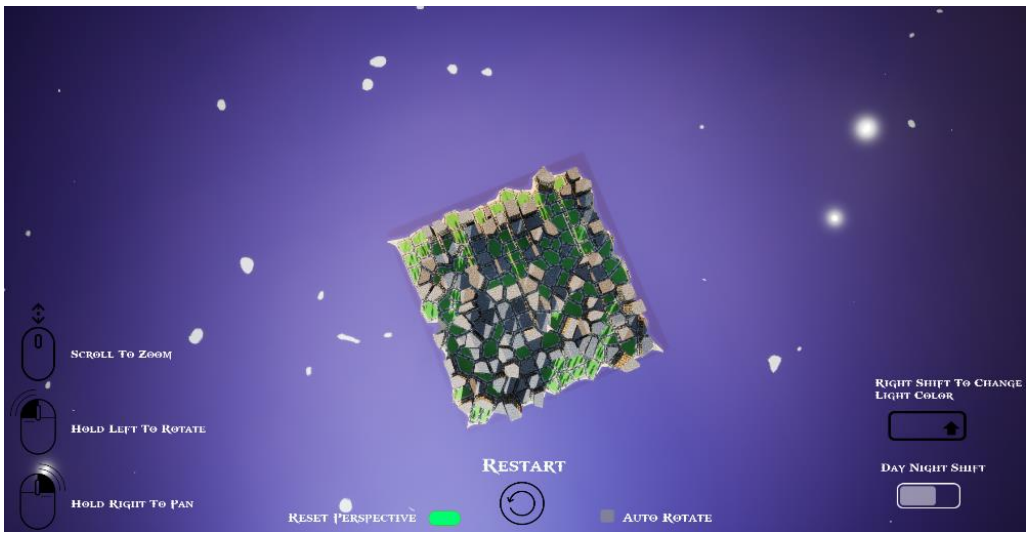


Figure 9: Voronoi Road Type

2. SQUARE

Figure 10: Square Road Type

3. RECTANGLE

Figure 11: Rectangle Road Type

The different road types will create different road shapes and building shapes. The difference is created through controlling the seed when generating the Voronoi Diagram. When the seed is free on the range of both X-axis and Y-axis, it's Voronoi shape. If it only frees on X/Y-axis, it's rectangle shape. If it does not free on both X/Y-axis, it's square shape.

3.3. Road/Pavement Width

Figure 12: Road Width Parameter

Figure 13: Pavement Width Parameter

1. HIGHER WIDTH

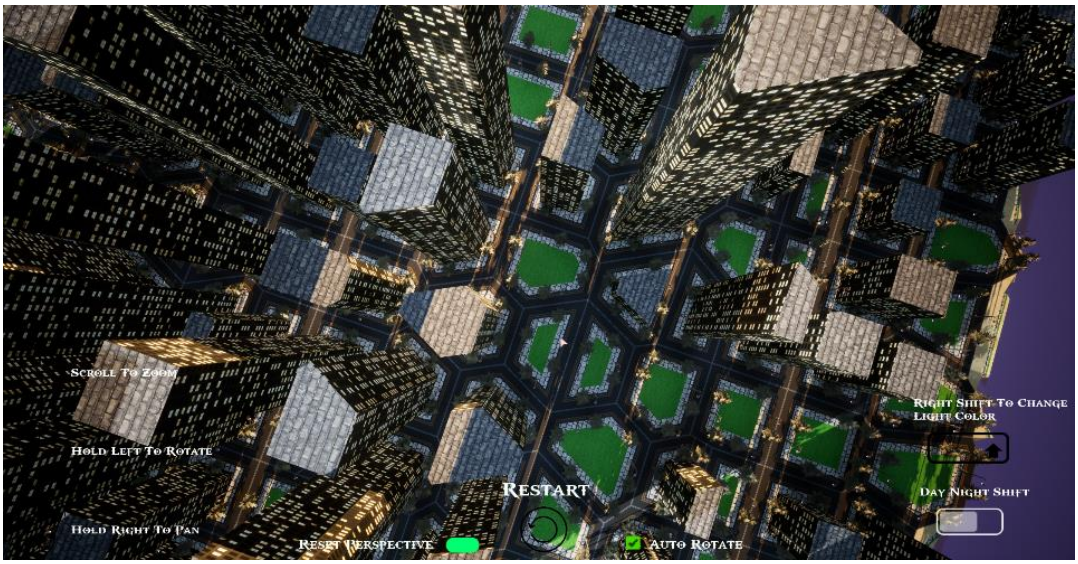


Figure 14: Higher Width

2. LOWER WIDTH

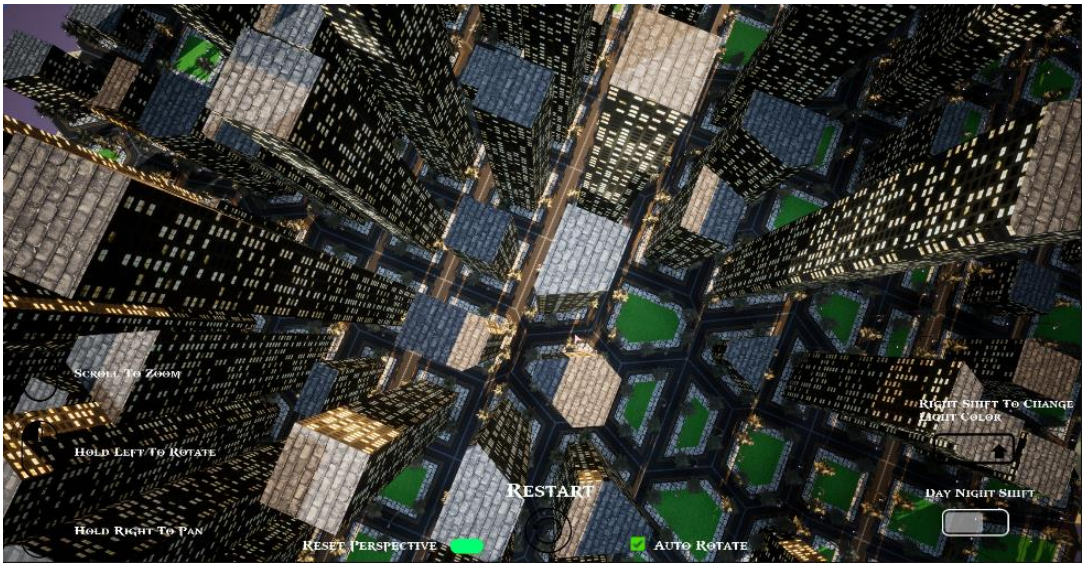


Figure 15: Lower Width

The different road/pavement width is not as visible as other parameters, since it has a smaller limit to avoid bugs. The way to create different width is through adjusting the triangle's height when creating the geometry, it also used to adjust the UV mapping.

3.4. Network Seed

Figure 16: Network Random Seed Parameter

1. ONE SEED

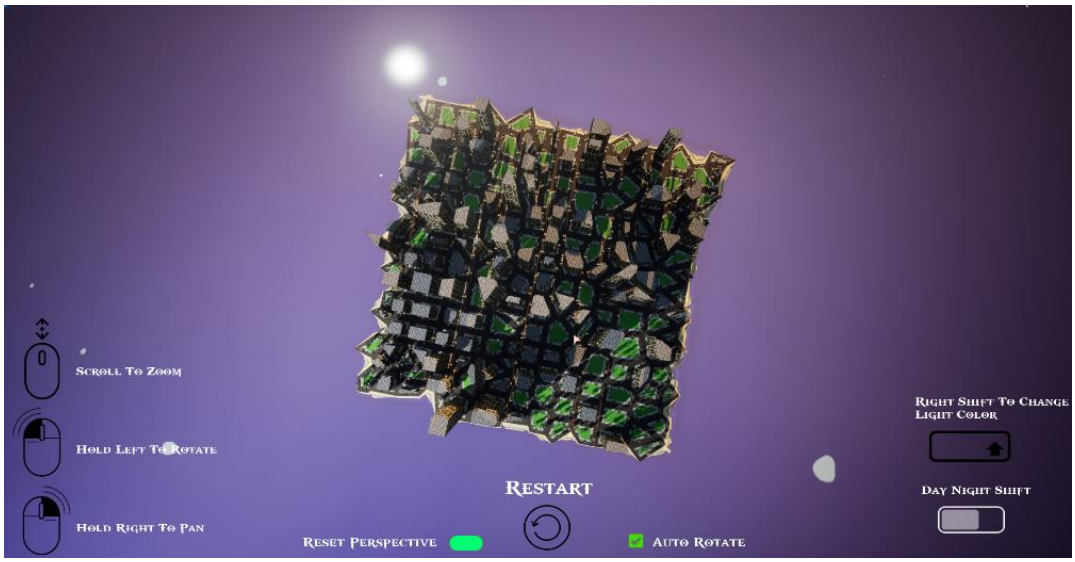


Figure 17: One Seed

2. OTHER SEED

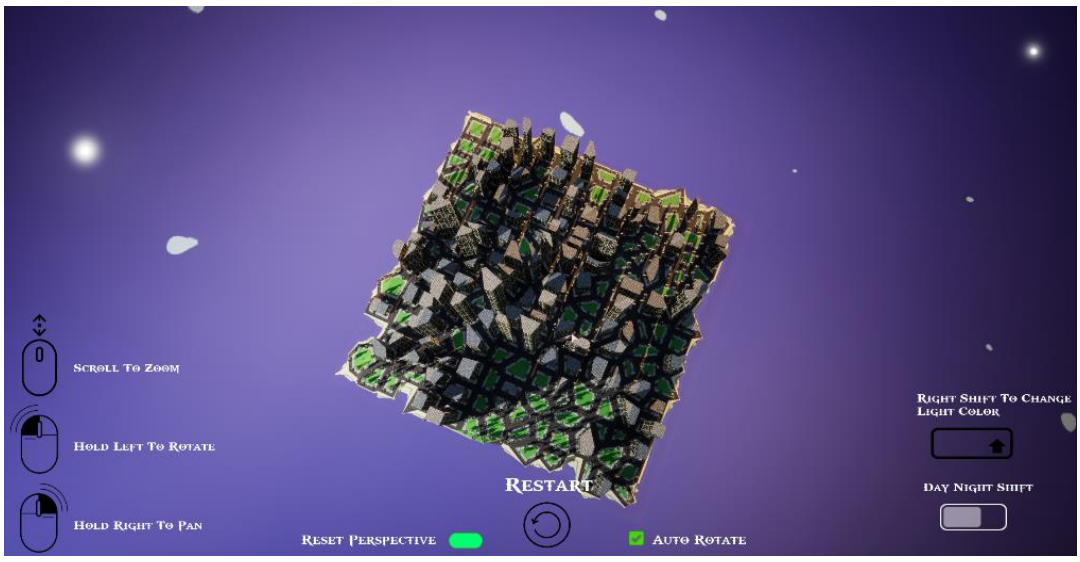


Figure 18: Other Seed

It’s quite clear that when putting different random seed number into the project, it will create a different road network and building layout looking. To achieve this result, I used some Hash code to create a Pseudo random noise which gives back a Pseudo random number ranged from 0-1 according to a certain input.

3.5. Building Distribution

Figure 19: Building Distribution Type Parameter

Because of the time limitation, I only used the normal distribution and Perlin noise to create the smoothly randomized building layout. I assumed that the highest building is the centre of the normal distribution, other buildings will all decrease from the highest building.

3.6. Highest Building

Figure 20: Highest Building Parameter

1. HIGHER HIGHEST BUILDING

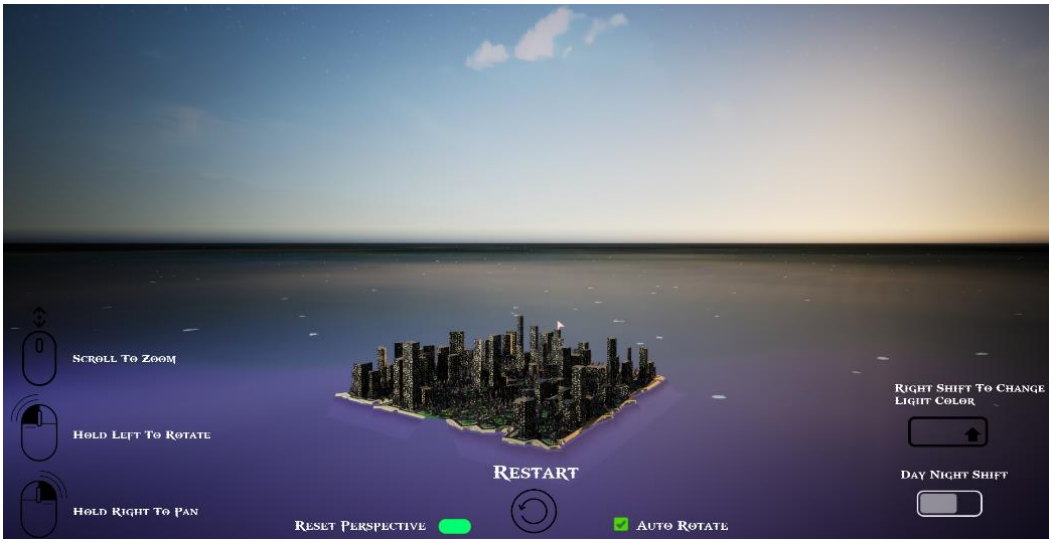


Figure 21: Higher Highest Building

2. LOWER HIGHEST BUILDING

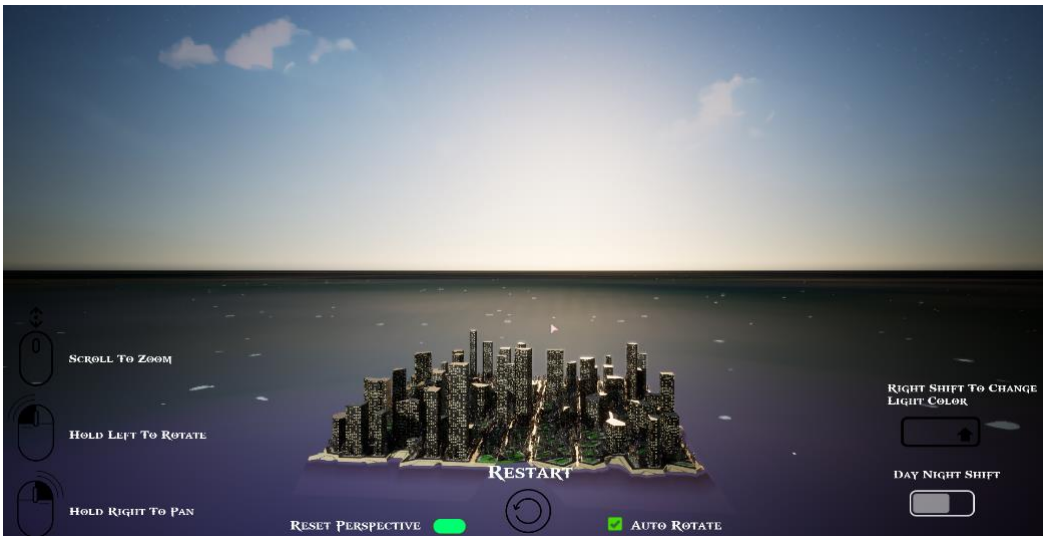


Figure 22: Lower Highest Building

The different highest building will influence the whole city's building height, which indicates the average height of the buildings. Since it's according to the normal distribution, if the highest building's height has been determined then other building's height will decrease from that value, which essentially indicates the average buildings' height.

3.7. City Centre

Figure 23: City Centre Parameter

1. ONE POSITION OF CITY CENTER

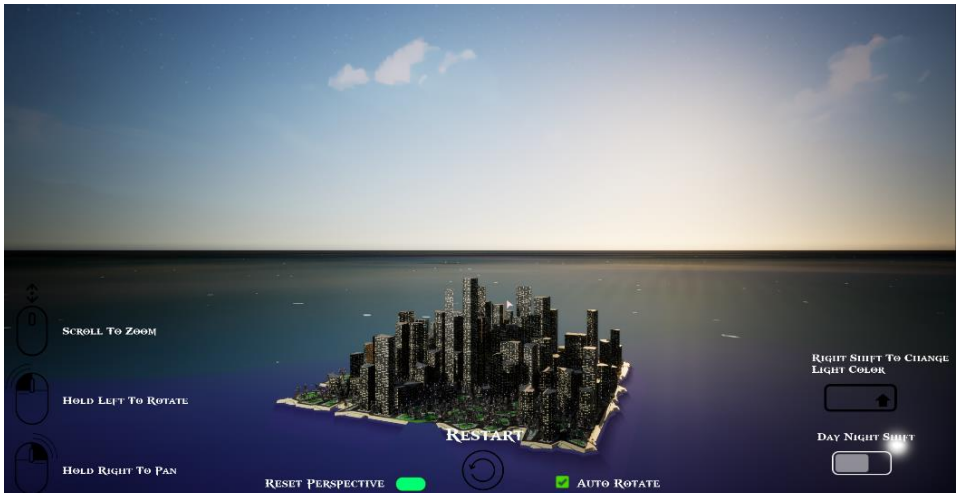


Figure 24: One Position of City Centre

2. OTHER POSITION OF CITY CENTER

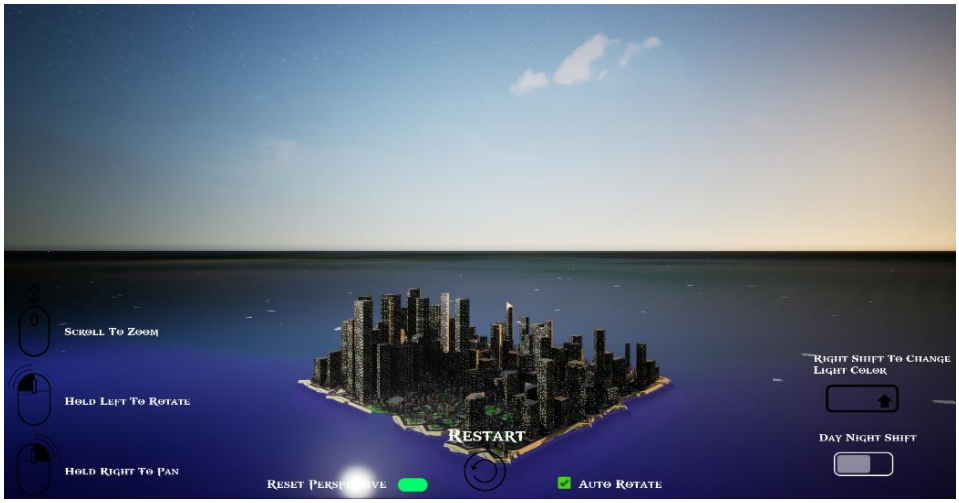


Figure 25: Other Position of City Centre

The different city centre position will influence the building's layout. For example, in the first picture (Figure 24), the city centre is located at the geometry centre of the whole square-shaped land. In the second picture (Figure 25), the city centre is located at the left-bottom corner of the square-shaped land. It's created through adjusting the highest building's location on the land, in which it assumes that the highest building's location is the city centre.

3.8. Building's Variance

Figure 26: Building Variance Parameter

1. LOWER VARIANCE

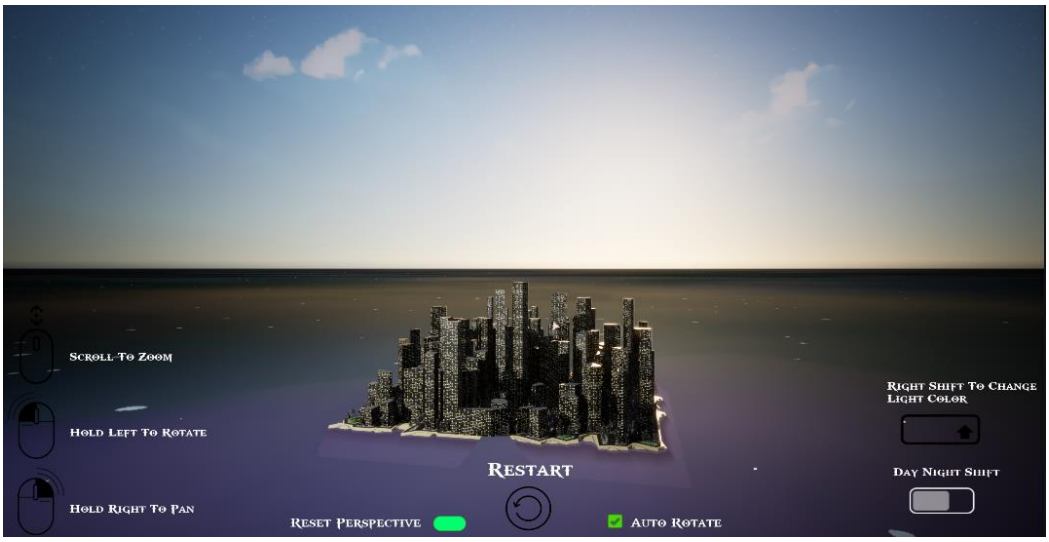


Figure 27: Lower Variance

2. HIGHEE VARIANCE

Figure 28: Higher Variance

The different variance indicates the different falloff of the building's normal distribution. With lower variance, statistically, the normal distribution will be more liked a rounded bell shape, on the other hand, with higher variance, the normal distribution will be more liked a sharp bell shape. Also, since the normal distribution I applied is 2D, so it has both X-axis and Y-axis for distribute the buildings.

3.9. Building's Seed

Figure 29: Building Random Seed Parameter

1. ONE SEED

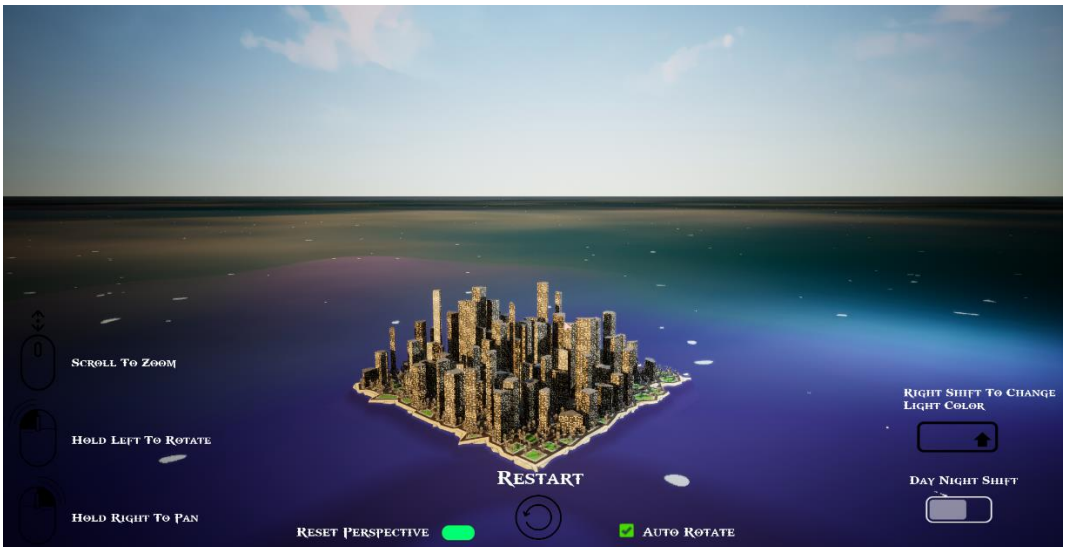


Figure 30: One Seed

2. OTHER SEED

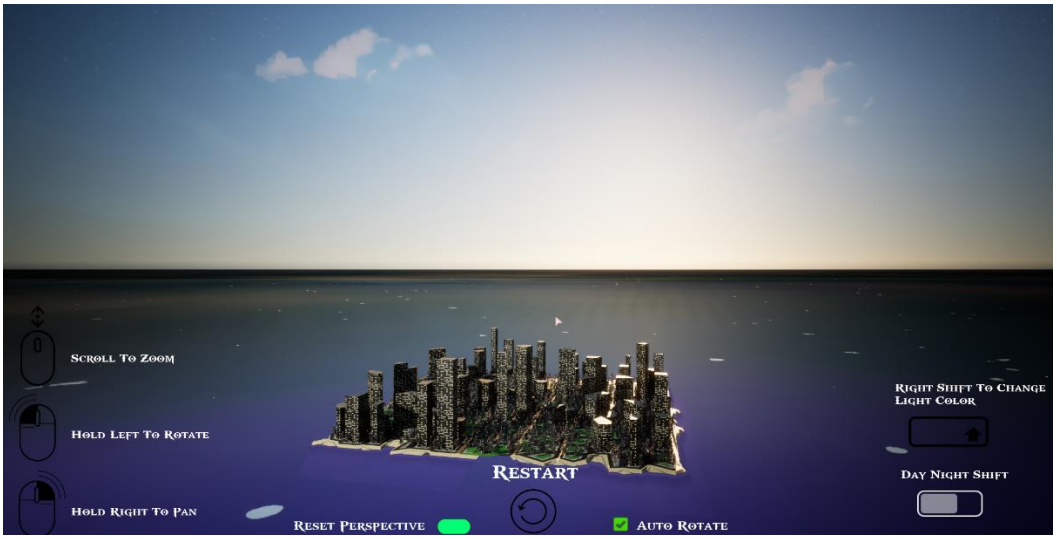


Figure 31: Other Seed

With different seed input, the building's layout will be different. It's the same logic with the road network seed, both using the Hash code to create Pseudo randomized number ranged from 0-1 to create different layouts.

3.10. Terrain's Seed

Figure 32: Terrain Seed Parameter

1. ONE SEED

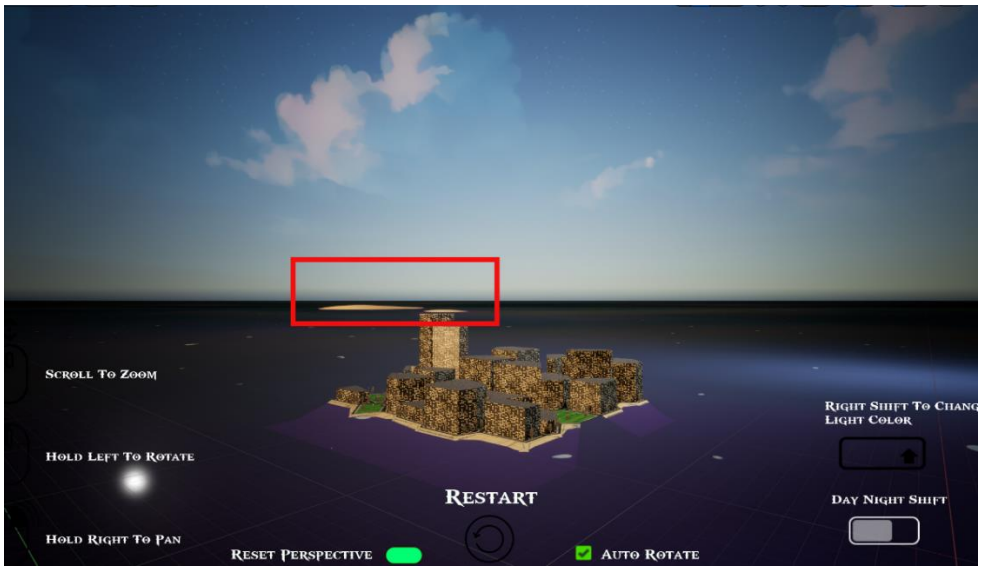


Figure 33: One Seed

2. OTHER SEED

Figure 34: Other Seed

The different terrain seed will create the different terrain shape of the of heightfield that underneath the ocean. Since the heightfield is created through Multi-layer Perlin noise, so the seed will essentially be filled to the Perlin noise to generate different heightfield shapes to create the different-shaped small island surrounded by the ocean.

3.11. Green Area Amount

Figure 35: Green Area Amount Parameter

1. LOWER VALUE OF GREEN AREA AMOUNT

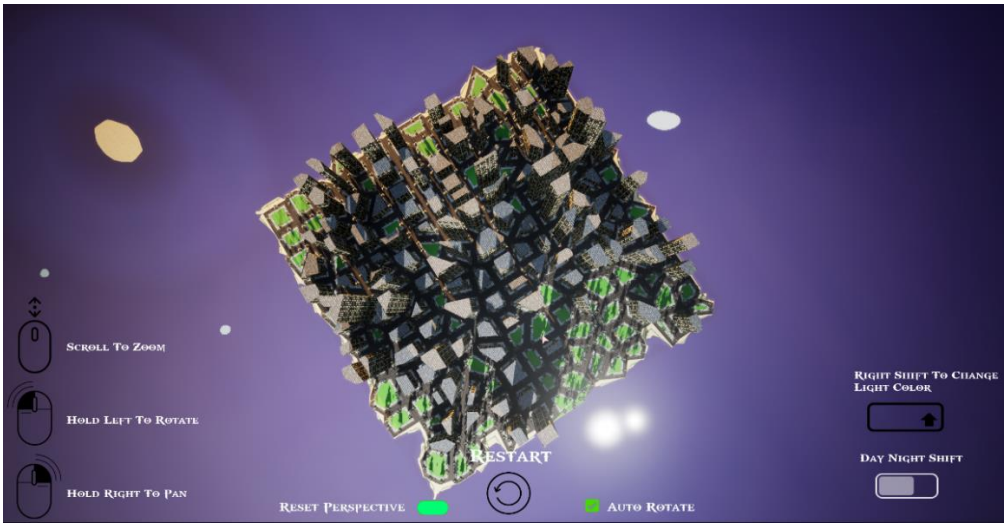


Figure 36: Lower Value of Green Area Amount

2. HIGHER VALUE OF GREEN AREA AMOUNT

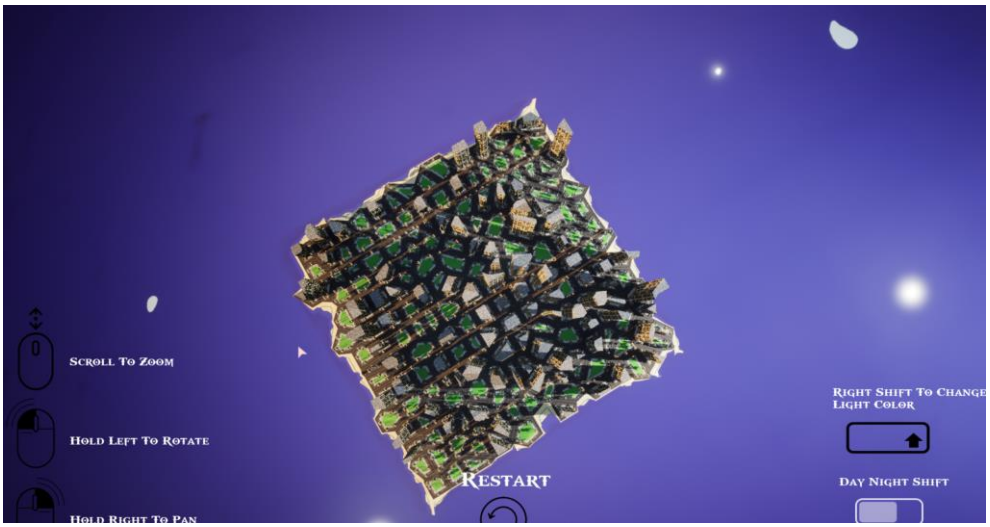


Figure 37: Higher Value of Green Area Amount

The different green area amount will influence how much green areas that exits on this land. The logic behind is quite straightforward, if a building's height is lower than certain value, the area will be the green space with the grass texture been put into the UV channel.

The section above is about the project's showcase of how different parameters worked together to create a featured customizable city. The following section will explain how those algorithmic logic is been written in code.

4. Project's Workflow

For the industry's standard purpose, the whole project is written in Unreal Engine 5 with C++ implemented. Around 3000 lines of code are included in the project, the core algorithm is about 2000 lines. With C++ code as the backend, the blueprint has been used for the front UI

demonstration and actor communication purposes. Since the purpose of the project is to use code to create geometry, so no other assets are included in the project except two models, one is a tree, the other is a streetlight, both will include in the appendices section. Also, no premade function library is included in this project, every function that used to implement the algorithm is written by myself to fully control the project's logic.

The algorithmic logic of the project is demonstrated in the picture below (Figure 38), each section will be explained with its running result (pictures) and the exact code logic behind.

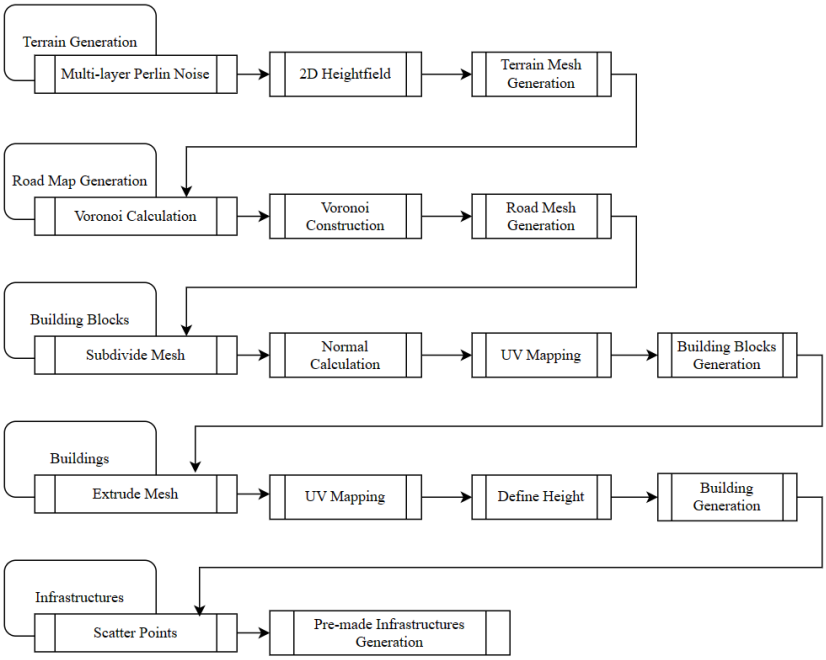


Figure 38: Algorithmic Logic Flowchart

4.1. Terrain Generation

The first step is to generate the terrain through Perlin noise. Perlin noise is a type of gradient noise, developed by Ken Perlin in 1983.

1. Single Layer Generation (with Perlin noise only applied once)

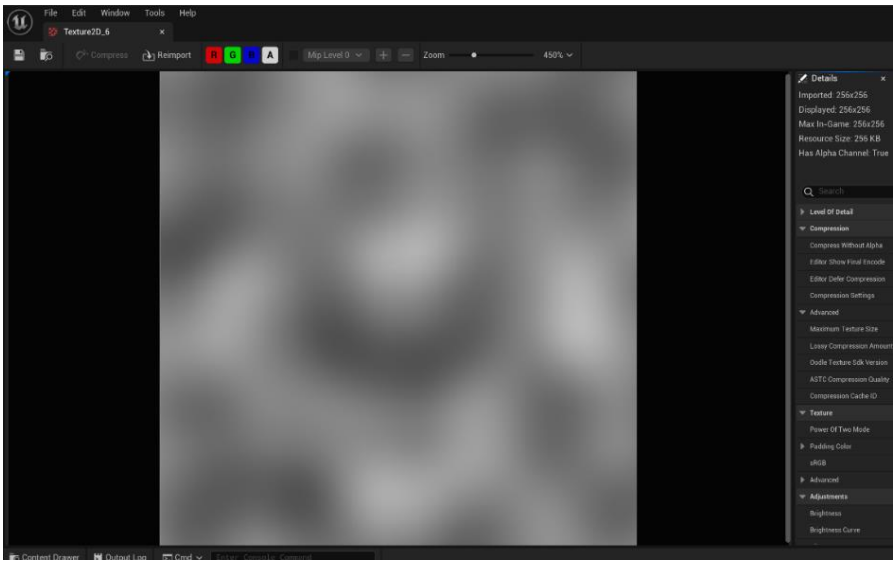


Figure 39: Single Layer Perlin Noise Generation

My Code Section Below:

```
float UMyBlueprintFunctionLibrary::PerlinNoiseCalculation(FVector2D WidthAndHeight, int32 GridCount, float perlinNoiseSeed, FVector2D PixelPos)
{
    const int32 Width = WidthAndHeight.X;
    const int32 Height = WidthAndHeight.Y;

    const int32 PixelInEachGridX = Width/GridCount;
    const int32 PixelInEachGridY = Height/GridCount;
    int32 i = PixelPos.X;
    int32 j = PixelPos.Y;

    //left bottom ---A
    FVector2D CurrentGridLeftBottomIndex = FVector2D( inX: FMath::Floor(i/(PixelInEachGridX)), inY: FMath::Floor( f: j/(PixelInEachGridX)));
    FVector2D CurrentGridLeftBottomPseudoVector = PelinNoiseGeneratePseudoRandomVector2D(CurrentGridLeftBottomIndex, perlinNoiseSeed);
    FGridPointWithPseudoRandomVector CurrentLeftBottom (CurrentGridLeftBottomPseudoVector,CurrentGridLeftBottomIndex);

    //right bottom ----D
    FVector2D CurrentGridRightBottomIndex = FVector2D( inX: CurrentLeftBottom.GridIndex.X+1, CurrentLeftBottom.GridIndex.Y);
    FVector2D CurrentGridRightBottomPseudoVector = PelinNoiseGeneratePseudoRandomVector2D(CurrentGridRightBottomIndex, perlinNoiseSeed);
    FGridPointWithPseudoRandomVector CurrentRightBottom (CurrentGridRightBottomPseudoVector,CurrentGridRightBottomIndex);

    //left top ---B

    FVector2D CurrentGridLeftTopIndex = FVector2D(CurrentLeftBottom.GridIndex.X, inY: CurrentLeftBottom.GridIndex.Y+1);
    FVector2D CurrentGridLeftTopPseudoVector = PelinNoiseGeneratePseudoRandomVector2D(CurrentGridLeftTopIndex, perlinNoiseSeed);
    FGridPointWithPseudoRandomVector CurrentLeftTop (CurrentGridLeftTopPseudoVector,CurrentGridLeftTopIndex);

    //right top ---C
    FVector2D CurrentGridRightTopIndex = FVector2D( inX: CurrentLeftBottom.GridIndex.X+1, inY: CurrentLeftBottom.GridIndex.Y+1);
    FVector2D CurrentGridRightTopPseudoVector = PelinNoiseGeneratePseudoRandomVector2D(CurrentGridRightTopIndex, perlinNoiseSeed);
    FGridPointWithPseudoRandomVector CurrentRightTop (CurrentGridRightTopPseudoVector,CurrentGridRightTopIndex);

    //FVector2D PPos = FVector2D(i,j);
    FVector2D PPos = FVector2D( inX: (i-CurrentLeftBottom.GridIndex.X*PixelInEachGridX)/(PixelInEachGridX), inY: (j-CurrentLeftBottom.GridIndex.Y*PixelInEachGridY)/(PixelInEachGridY));
    FVector2D AP = (PPos - FVector2D( inX: 0, inY: 0));
    FVector2D BP = (PPos - FVector2D( inX: 0, inY: 1));
    FVector2D CP = (PPos - FVector2D( inX: 1, inY: 1));
    FVector2D DP = (PPos - FVector2D( inX: 1, inY: 0));

    float ADotProduct = FVector2D::DotProduct( A: CurrentLeftBottom.GridPseudoRandomVector, B: AP);
    float BDotProduct = FVector2D::DotProduct( A: CurrentLeftTop.GridPseudoRandomVector, B: BP);
    float CDotProduct = FVector2D::DotProduct( A: CurrentRightTop.GridPseudoRandomVector, B: CP);
    float DDotProduct = FVector2D::DotProduct( A: CurrentRightBottom.GridPseudoRandomVector, B: DP);

    float ADLerp = PerlinNoiseLerp( t: ADotProduct, r: DDotProduct, t: PPos.X);
    float BCLerp = PerlinNoiseLerp( t: BDotProduct, r: CDotProduct, t: PPos.X);
    float NoiseValue = PerlinNoiseLerp( t: ADLerp, r: BCLerp, t: PPos.Y);

    NoiseValue = (NoiseValue + 1.0)/2;

    return NoiseValue;
```

Figure 40: Single Layer Perlin Noise Generation Code

Perlin noise is a quite mature algorithm, the logic behind it is shown below:

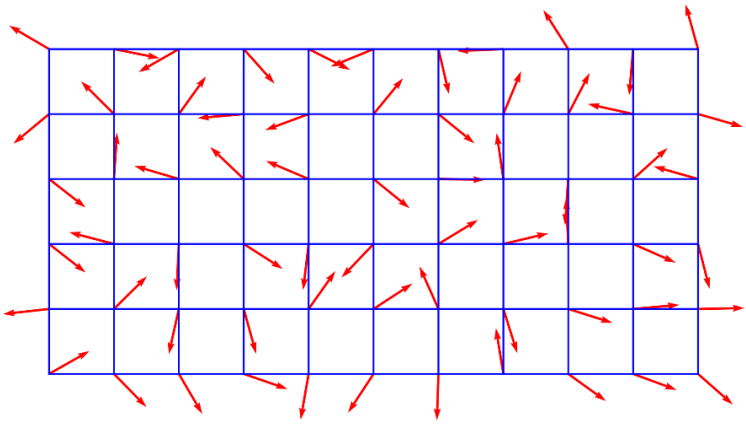


Figure 41: Perlin Noise Logic

By calculating a dot product of a random vector, it will give back a smooth Pseudo value to demonstrate the grey scale of the current pixel. My code is similar to the majority Perlin noise code.

2. Multilayer Generation (with Perlin noise applied three times)

For each time generation, octaves increase one, frequency multiplied by two, strength divided by two.

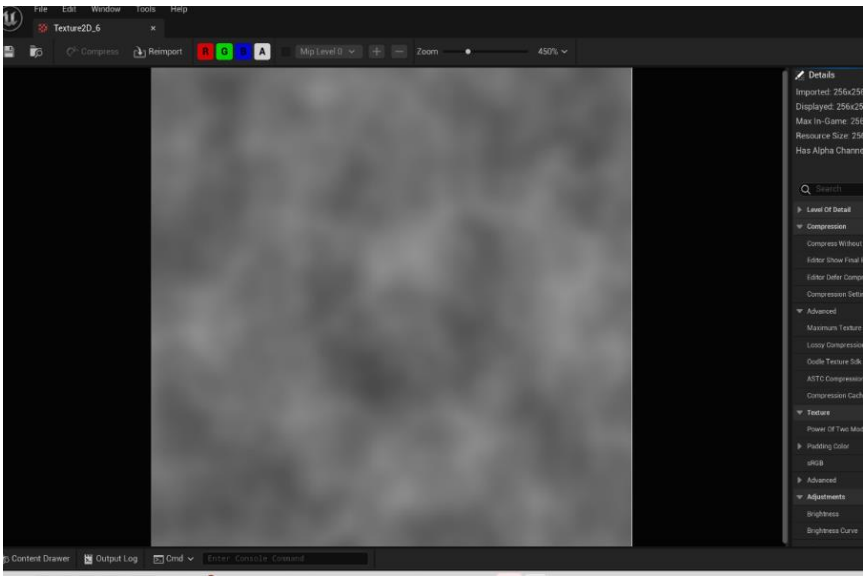


Figure 42 : Multilayer Perlin Noise Generation

My Code Section Below:

```
float UMyBlueprintFunctionLibrary::FractualBrownMotion(int32 octaves, int32 initialFrequency, float initialStrength,FVector2D PixelPos,FVector2D WidthAndHeight)
{
    //const int32 Width = Texture2D->GetSizeX();
    //const int32 Height = Texture2D->GetSizeY();
    //FByteBulkData* RawImageDataOut = &Texture2D->GetPlatformData()->Mips[0].BulkData;
    //FColor* FormatedImageDataOut = static_cast<FColor*>(RawImageDataOut->Lock(LOCK_READ_WRITE));



    //for(int32 i =0;i<Width; i++)
    //{
        //for(int32 j =0; j<Height;j++)
        //{

    const int i = PixelPos.X;
    const int j = PixelPos.Y;

    float NoiseValue = 0;
    float frequency = initialFrequency;
    float strength = initialStrength;


    for (int32 k = 0; k < octaves; k++)
    {
        strength /= 2;
        float layerNoise = PerlinNoiseCalculation(WidthAndHeight, frequency, PerlinNoiseSeed + frequency,  PixelPos: FVector2D( inX: i,  inY: j));
        NoiseValue += layerNoise * strength;

        frequency *= 2;

    }

    NoiseValue = FMath::Clamp( X: NoiseValue, Min: 0.0f, Max: 1.0f);
    return NoiseValue;

            //FColor PixelColor = FColor(NoiseValue * 255, NoiseValue * 255, NoiseValue * 255, 255);
            //FormatedImageDataOut[(j * Width) + i] = PixelColor;

            //UE_LOG(LogTemp, Log, TEXT("NoiseValue at (%d, %d): %f"), i, j, NoiseValue);




        //}
    //}

    //RawImageDataOut->Unlock();
    //Texture2D->UpdateResource();
```

Figure 43: Multilayer Perlin Noise Generation Code

The logic is to iterate Perlin noise three times in the multi-layer calculation.

3. Heightfield Generation

By generating multilayer Perlin noise, each pixel's grey scale will represent the height of this mesh piece. Through iterating every pixel, the heightfield can be constructed through certain vertex sorting logic.

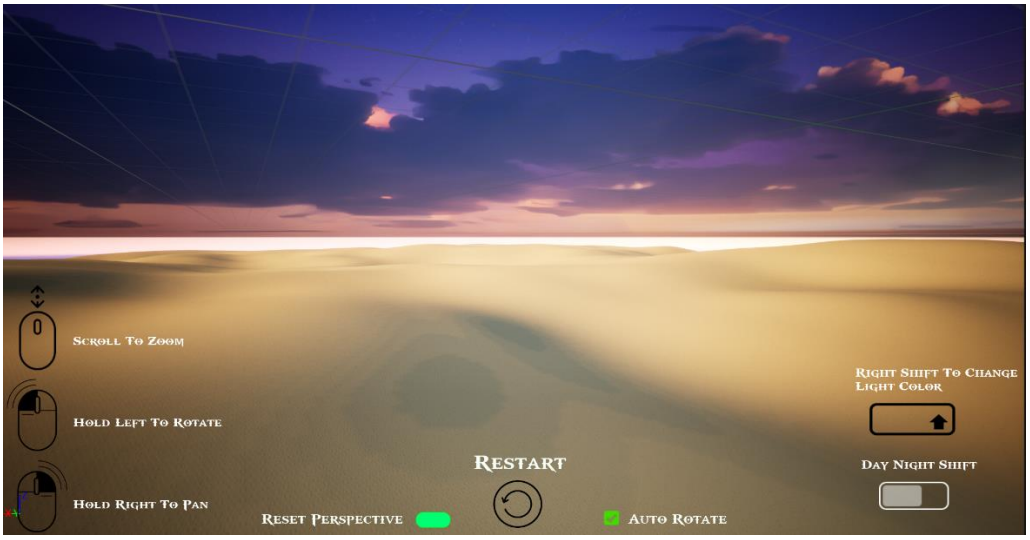


Figure 44: Heightfield Generation (Mountain)

My Code Section Below:

```
void UMyBlueprintFunctionLibrary::GenerateTerrian(UProceduralMeshComponent* TerrainProceduralMesh,
    UMaterialInterface* TerrainMaterialInstance, FVector2D terrainHeightAndWidth)
{

    const int32 Width = terrainHeightAndWidth.X;
    const int32 Height = terrainHeightAndWidth.Y;
    float TallScale = 90.0f;

    TArray<FVector> TerrainVertices;
    TArray<int32> TerrainTriangles;
    TArray<FVector2D> TerrainUVs;
    TArray<FVector> TerrainNormals;
    TArray<FProcMeshTangent> TerrainTangents;

    TerrainNormals.Init( Element: FVector( InX: 0.0f, InY: 0.0f, InZ: 0.0f), Number: Width * Height);


    for (int32 y = 0; y < Height; y++)
    {
        for (int32 x = 0; x < Width; x++)
        {
            float TerrainHeight = FractalBrownMotion( octaves: 3, initialFrequency: 4, initialStrength: 1, PixelPos: FVector2D(x,y),terrainHeightAndWidth ) * TallScale;
            TerrainVertices.Add( Item: FVector(x, y, InZ: TerrainHeight));
            TerrainUVs.Add( Item: FVector2D(x, y));
        }
    }


    for (int32 y = 0; y < Height - 1; y++)
    {
        for (int32 x = 0; x < Width - 1; x++)
        {
            int32 Index1 = y * Width + x;
            int32 Index2 = Index1 + Width;
            int32 Index3 = Index1 + 1;
            int32 Index4 = Index2 + 1;

            // 第一个三角形
            TerrainTriangles.Add(Index1);
            TerrainTriangles.Add(Index2);
            TerrainTriangles.Add(Index3);

            // 第二个三角形
            TerrainTriangles.Add(Index3);
            TerrainTriangles.Add(Index2);
            TerrainTriangles.Add(Index4);

            // 计算面法线并累加到顶点法线
            FVector Normal1 = -FVector::CrossProduct( A: TerrainVertices[Index2] - TerrainVertices[Index1], B: TerrainVertices[Index3] - TerrainVertices[Index1]).GetSafeNormal();
```

Figure 45: Heightfield Generation Code

```
            int32 Index2 = Index1 + Width;
            int32 Index3 = Index1 + 1;
            int32 Index4 = Index2 + 1;

            // 第一个三角形
            TerrainTriangles.Add(Index1);
            TerrainTriangles.Add(Index2);
            TerrainTriangles.Add(Index3);

            // 第二个三角形
            TerrainTriangles.Add(Index3);
            TerrainTriangles.Add(Index2);
            TerrainTriangles.Add(Index4);

            // 计算面法线并累加到顶点法线
            FVector Normal1 = -FVector::CrossProduct(A: TerrainVertices[Index2] - TerrainVertices[Index1], B: TerrainVertices[Index3] - TerrainVertices[Index1]).GetSafeNormal();
            FVector Normal2 = -FVector::CrossProduct(A: TerrainVertices[Index2] - TerrainVertices[Index3], B: TerrainVertices[Index4] - TerrainVertices[Index3]).GetSafeNormal();

            TerrainNormals[Index1] += Normal1;
            TerrainNormals[Index2] += Normal1 + Normal2;
            TerrainNormals[Index3] += Normal1 + Normal2;
            TerrainNormals[Index4] += Normal2;
        }
    }

    // 标准化顶点法线
    for (FVector& TerrainNormal : TerrainNormals)
    {
        TerrainNormal.Normalize();
    }

    // 创建网格
    TerrainProceduralMesh->CreateMeshSection(0, TerrainVertices, TerrainTriangles, TerrainNormals, TerrainUVs, VertexColors: TArray<FColor>(), TerrainTangents, bCreateCollision: true);
    if (TerrainMaterialInstance)
    {
        TerrainProceduralMesh->SetMaterial(ElementIndex: 0, TerrainMaterialInstance);
    }

    TerrainVertices.Empty();
    TerrainTriangles.Empty();
    TerrainUVs.Empty();
    TerrainNormals.Empty();
    TerrainTangents.Empty();

}
```

Figure 46: Heightfield Generation Code

4.2. Road Map Generation

The second step is to generate the road map. The road map generation is the core of the project, which is being implemented by Voronoi diagram. In Voronoi diagram, there are three steps of road map generation, which are colouring, constructing and tessellation.

1. **Voronoi Colouring (Define the colour)**

Figure 47: Voronoi Colouring

My Code Section Below:

```
void UMyBlueprintFunctionLibrary::VoronoiCalculation(UTexture2D* Texture2D)
{
    InitializeClosestCellVoronoiSeedXY(Texture2D);
    const int32 Width = Texture2D->GetSizeX();
    const int32 Height = Texture2D->GetSizeY();
    FByteBulkData* RawImageDataOut = &Texture2D->GetPlatformData()->Mips[0].BulkData;
    FColor* FormatedImageDataOut = static_cast<FColor*>(RawImageDataOut->Lock(LOCK_READ_WRITE));

    const int32 PixelsInEachCellX = Width/CellCount;
    const int32 PixelsInEachCellY = Height/CellCount;



    for(int X=1; X<Width-1; X++)
    {
        for (int Y =1; Y<Height-1; Y++)
        {
            const FVector2D CellXY = FVector2D( inX: FMath::Floor( In: X/PixelsInEachCellX),  inY: FMath::Floor( In: Y/PixelsInEachCellY));
            float MinDist = FLT_MAX;
            FVector2D ClosestCell = FVector2D( inX: 0, inY: 0);
            FVector2D PixelXYToClosestCellSeed = FVector2D( inX: 0, inY: 0);

            for(int i = -1; i<=1; i++)
            {
                for (int j = -1; j <=1; j++)
                {
                    const FVector2D CurrentCellXY = FVector2D( inX: CellXY.X + i,  inY: CellXY.Y +j);
                    if(CurrentCellXY.X < 0 || CurrentCellXY.Y < 0 || CurrentCellXY.X >=CellCount || CurrentCellXY.Y >=CellCount) continue;
                    const FVector2D CurrentCellPixelXY = FVector2D(PixelsInEachCellX* CurrentCellXY.X,PixelsInEachCellY* CurrentCellXY.Y);
                    int32 RandomX = 0;
                    int32 RandomY = 0;

                    if(IsVoronoi)
                    {
                        RandomX = FMath::Floor( In: FMath::Lerp( A: 0,  B: PixelsInEachCellX, Vector2DtoGeneratePseudoRandomVector2D(CurrentCellPixelXY).X));
                        RandomY = FMath::Floor( In: FMath::Lerp( A: 0,  B: PixelsInEachCellY, Vector2DtoGeneratePseudoRandomVector2D(CurrentCellPixelXY).Y));
                    }

                    if(IsSquare)
                    {
                        RandomX = PixelsInEachCellX/2;
                        RandomY = PixelsInEachCellY/2;
                    }

                    if(IsRectangle)
                    {
                        RandomX = PixelsInEachCellX/2;
                        RandomY = FMath::Floor( In: FMath::Lerp( A: 0,  B: PixelsInEachCellY, Vector2DtoGeneratePseudoRandomVector2D(CurrentCellPixelXY).Y));
                    }
```

Figure 48: Voronoi Colouring Code

```
                    FVector2D VoronoiSeedPixelXY = FVector2D( inX: (CurrentCellPixelXY.X + RandomX),  inY: (CurrentCellPixelXY.Y + RandomY));

                    VoronoiSeeds.Add(VoronoiSeedPixelXY);
                    FVector2D CurrentPixelToVoronoiSeed = (FVector2D(X,Y) - VoronoiSeedPixelXY);
                    const float Dist = CurrentPixelToVoronoiSeed.Length();
                    if(Dist < MinDist)
                    {
                        PixelXYToClosestCellSeed = CurrentPixelToVoronoiSeed;
                        MinDist = Dist;
                        ClosestCell = CurrentCellXY;
                        ClosestCellVoronoiSeedXY[X][Y] = VoronoiSeedPixelXY;
                    }

                }
            }



            const float NoiseColor = Vector2DtoGeneratePseudoRandomVector1D( Vector2D: ClosestCell,  A: FVector2D( inX: 289.89,  inY: 38.02),  B: 4.98,  C: 60.90);
            const FVector Color = Vector1DtoGeneratePseudoRandomVector3D(NoiseColor);
            const FColor PixelDrawCol = FColor( inR: Color.X * 255,  inG: Color.Y * 255, inB: Color.Z * 255,  inA: 255);


            const FColor PixelColor = PixelDrawCol;
            FormatedImageDataOut[(Y * Width) + X] = PixelColor;

        }
    }


    for(int X1 =0; X1<Width; X1++)
    {
        for (int Y1 =0; Y1<Height; Y1++)
        {
            ClosestCellVoronoiSeedXY[X1][0] = FVector2D( inX: Width/2, inY: 0);
            ClosestCellVoronoiSeedXY[X1][Height - 1] = FVector2D( inX: Width/2,  inY: Height-1);
            ClosestCellVoronoiSeedXY[0][Y1] =  FVector2D( inX: 0, inY: Height/2);
            ClosestCellVoronoiSeedXY[Width - 1][Y1] = FVector2D( inX: Width-1,  inY: Height/2);
        }
    }
    RawImageDataOut->Unlock();
    Texture2D->UpdateResource();
}
```

Figure 49: Voronoi Colouring Code

The Voronoi Colouring is also a mature algorithm, my code's different is from 418 – 427 lines, in which I assigned some unique numbers for calculating the edges of the Voronoi Diagram. This part is not necessary for every developer, I'm using that only for the purpose of facilitate later geometry calculations.

The Voronoi calculation is highly depends on the lattice division, on each lattice only one seed is randomly generated. For each pixel, they need to find the nearest seed and contain this seed for next step calculation.

2. Voronoi Constructing (Define pointes and edges)

- *Point Calculation*

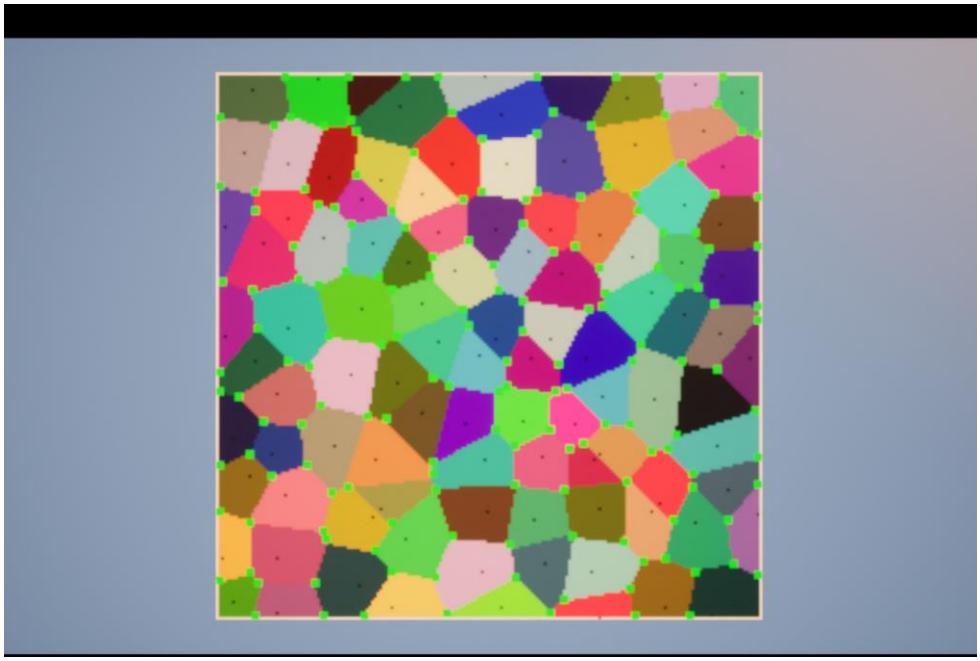


Figure 50: Voronoi Point Calculation

My Code Section Below:

```
// }
void UMyBlueprintFunctionLibrary::CalculateVertices(UTexture2D* Texture2D)
{
    const int32 Width = Texture2D->GetSizeX();
    const int32 Height = Texture2D->GetSizeY();
    Vertices.Empty();

    for(int X=1; X<Width-1; X++)
    {
        for (int Y =1; Y<Height-1; Y++)
        {

            TSet<FVector2D> UniqueSeeds;
            for(int i = -1; i <=1; i++)
            {
                for(int j = -1; j <= 1; j++)
                {

                    FVector2D CurrentClosestSeed = ClosestCellVoronoiSeedXY[X+i][Y+j];

                    UniqueSeeds.Add(CurrentClosestSeed);

                }
            }



            if(UniqueSeeds.Num() >= 3)
            {

                Vertices.Add(Item: FVector2D(X, Y));


            }

        }
    }


}

```

Figure 51: Voronoi Point Calculation Code

```
void UMyBlueprintFunctionLibrary::MergeSpecialCaseWithFourOrMoreVertices(TArray<FVerticesEdgesStruct>& Array)
{

    for (int32 i = 0; i < Array.Num(); i++)
    {
        FVerticesEdgesStruct CurrentVertex = Array[i];
        if(CurrentVertex.CurrentCellsUniqueNumbers.Num() >= 4)
        {
            for(int32 j = 0; j<Array.Num(); j++)
            {
                if(j!=i && CurrentVertex.IsContainMoreThan3Element( Array[j]))
                {
                    TSet<int32> CombinedCellNum;
                    CombinedCellNum.Empty();
                    CombinedCellNum.Append(CurrentVertex.CurrentCellsUniqueNumbers);
                    CombinedCellNum.Append(Array[j].CurrentCellsUniqueNumbers);
                    Array[i].CurrentCellsUniqueNumbers.Empty();
                    for(int32 Element : CombinedCellNum)
                    {
                        Array[i].CurrentCellsUniqueNumbers.Add(Element);
                    }

                    Array.RemoveAt(j);


                }
            }
        }

    }
}

```

Figure 52: Voronoi Point Calculation Code

```cpp

void UMyBlueprintFunctionLibrary::DrawMergedVerticesOnTexture2D(UTexture2D* Texture2D, FColor color)
{
    const int32 Width = Texture2D->GetSizeX();
    const int32 Height = Texture2D->GetSizeY();
    FByteBulkData* RawImageDataOut = &Texture2D->GetPlatformData()->Mips[0].BulkData;
    FColor* FormatedImageDataOut = static_cast<FColor*>(RawImageDataOut->Lock(LOCK_READ_WRITE));
    for(int i = 0; i<Merged4CellCountVerticesEdges.Num(); i++)
    {

        for (int X = 0; X < Width; X++)
        {
            for(int Y = 0; Y < Height; Y++)
            {

                if(Merged4CellCountVerticesEdges[i].VertexPosition.X == X && Merged4CellCountVerticesEdges[i].VertexPosition.Y ==Y)
                {
                    FormatedImageDataOut[(Y * Width) + X] = color;
                }
            }
        }
    }
    RawImageDataOut->Unlock();
    Texture2D->UpdateResource();
}


```

Figure 53: Voronoi Point Calculation Code

```cpp
USTRUCT(BlueprintType)
struct FVerticesEdgesStruct
{
    GENERATED_BODY()

public:

    FVector2D VertexPosition;
    TSet<int32> CurrentCellsUniqueNumbers;

    bool IsContainOtherArray( const FVerticesEdgesStruct Other) const
    {

        for (const int32& Element : Other.CurrentCellsUniqueNumbers)
        {
            if (!CurrentCellsUniqueNumbers.Contains(Element))
            {
                return false;
            }
        }

        return true;


    }

    bool IsTwoEquivalent(const FVerticesEdgesStruct& Other) const
    {
        if (CurrentCellsUniqueNumbers.Num() != Other.CurrentCellsUniqueNumbers.Num())
        {
            return false;
        }

        for (const int32 Element : CurrentCellsUniqueNumbers)
        {
            if (!Other.CurrentCellsUniqueNumbers.Contains(Element))
            {
                return false;
            }
        }

        return true;
    }

    bool IsContainMoreThan3Element( const FVerticesEdgesStruct Other) const
    {

        int Count = 0;
        for (const int32 Element : CurrentCellsUniqueNumbers)
        {
            if (Other.CurrentCellsUniqueNumbers.Contains(Element))
            {
```

Figure 54: Voronoi Point Calculation Code

Since this is not a mature algorithm, I write a lot of code to conduct the logic to find the vertices. This section's logic is to iterate the 8 near pixels for each of the pixel, if the unique seed number is larger than or equal to 3, then I think this pixel is the corner – vertex. But in this approach, many similar vertices will be calculated as a corner. To solve this issue, I also applied some logic to cluster these similar vertices as one exact vertex. I think this is most important code section in my whole project. Without this code section, the whole project will crash (though I think this code still need be polished in many aspects).

- *Point Cluster and Edge Construction*

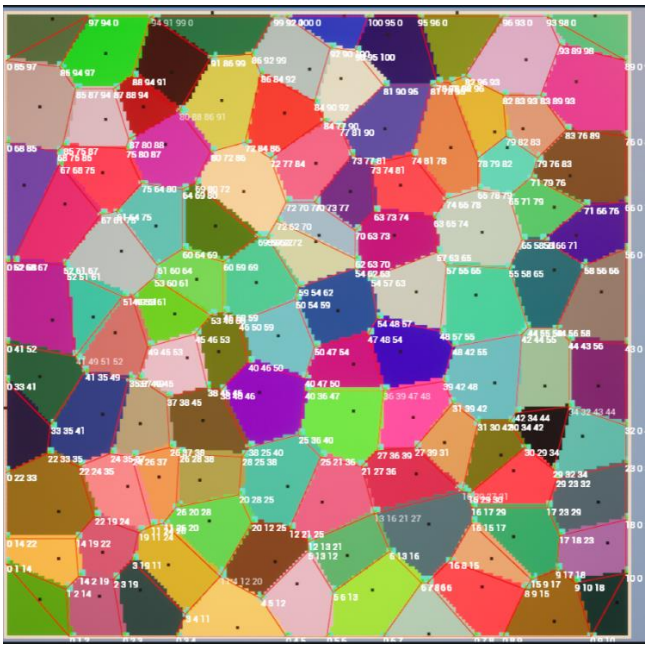


Figure 55: Voronoi Point Cluster and Edge Construction

My Code Section Below:

```
void UMyBlueprintFunctionLibrary::GroupVerticesWithSharedCells(UTexture2D* Texture2D)
{
    const int32 Width = Texture2D->GetSizeX();
    const int32 Height = Texture2D->GetSizeY();

    for(int i = 0; i<Merged4CellCountVerticesEdges.Num();i++)
    {
        FVerticesEdgesStruct CurrentVertex = Merged4CellCountVerticesEdges[i];
        int32 X = CurrentVertex.VertexPosition.X;
        int32 Y = CurrentVertex.VertexPosition.Y;

        for(int j = i+1; j<Merged4CellCountVerticesEdges.Num();j++)
        {
            int SharedCellNumber = 0;
            for(const int CellNumber : CurrentVertex.CurrentCellsUniqueNumbers)
            {
                if(Merged4CellCountVerticesEdges[j].CurrentCellsUniqueNumbers.Contains(CellNumber))
                {
                    SharedCellNumber++;
                }
            }



            if(SharedCellNumber>=2)
            {
                FPairedVertices CurrentPairedVertices ( InFirstVertex: CurrentVertex.VertexPosition, InSecondVertex: Merged4CellCountVerticesEdges[j].VertexPosition);
                bool bIsNewPairUnique = true;
                for(const FPairedVertices& ExistingPair : PairedVertices)
                {
                    if(CurrentPairedVertices.IsEquivalent(ExistingPair))
                    {
                        bIsNewPairUnique = false;
                        break;
                    }
                }


                if(bIsNewPairUnique)
                {
                    PairedVertices.Add(CurrentPairedVertices);
                }

            }


        }
```

Figure 56: Voronoi Point Cluster and Edge Construction Code

This code is simply group two points as an edge, if the unique seed number on these two points is larger than or equal to 2, then I consider them as a pair. This step is prepared for the next step's geometry calculations.

3. Tessellation (Mesh construction)

- In wireframe mode

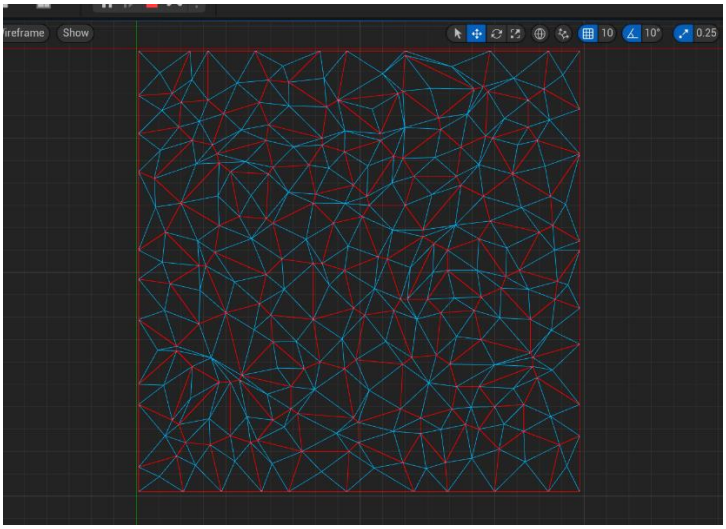


Figure 57: Voronoi Tessellation in Wireframe Mode

- In view mode

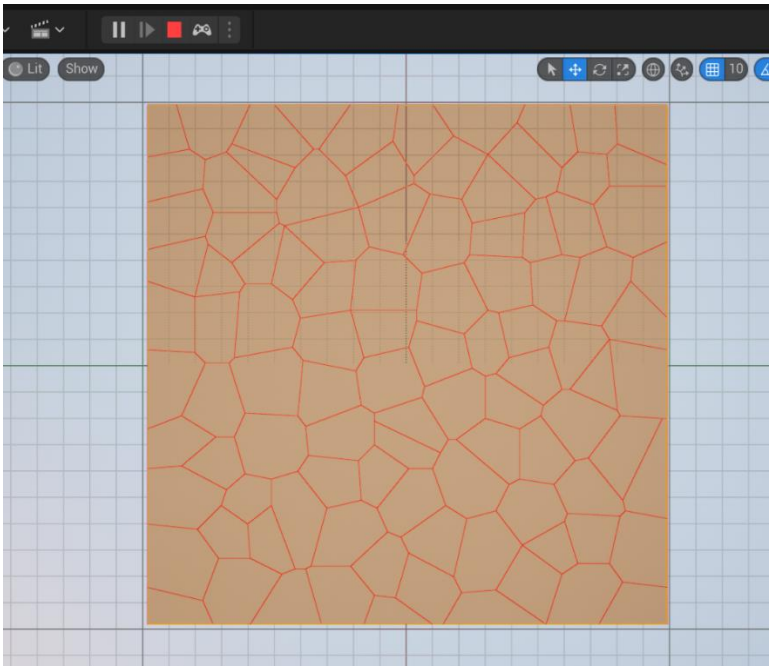
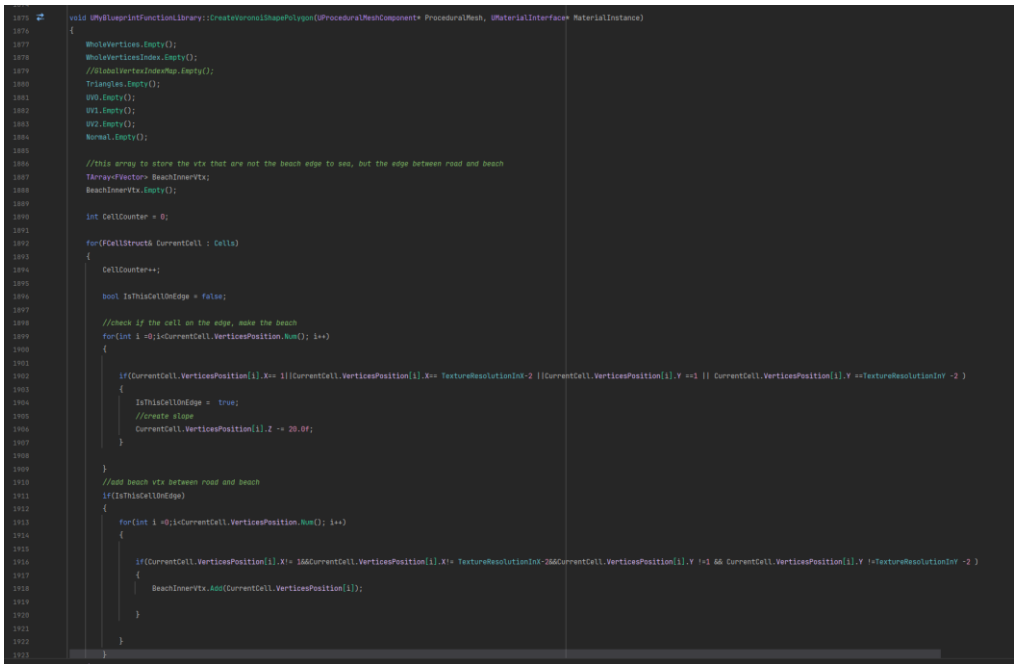


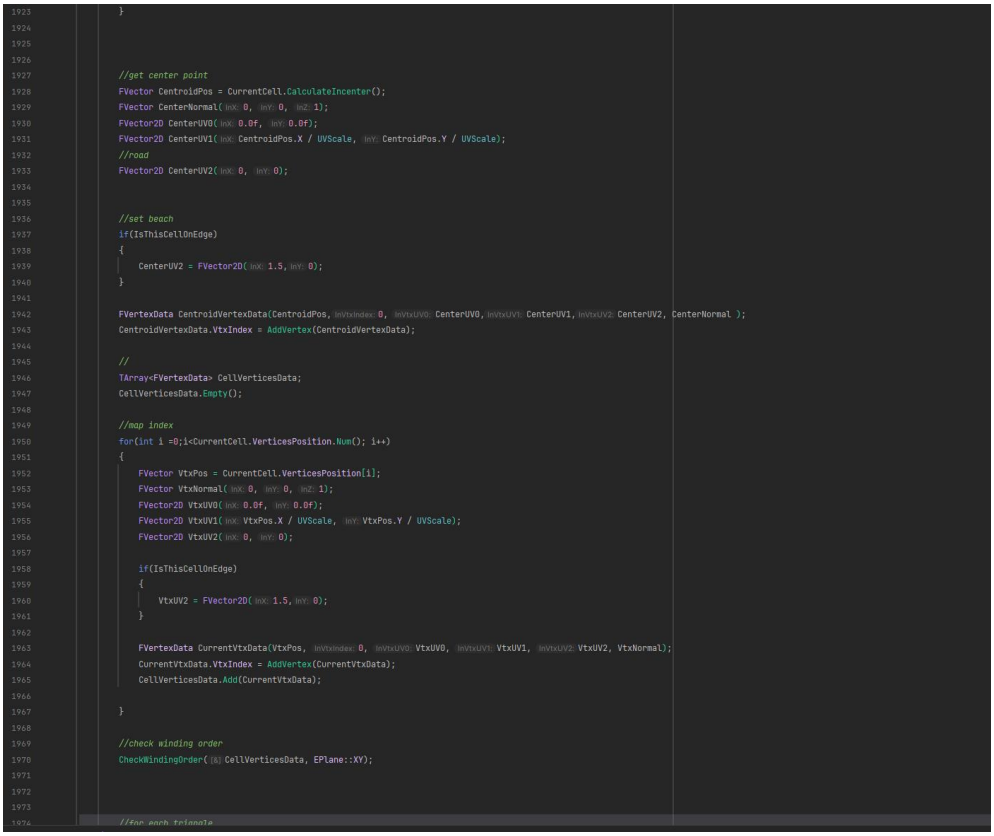
Figure 58: Voronoi Tessellation in Render Mode

My Code Section Below:



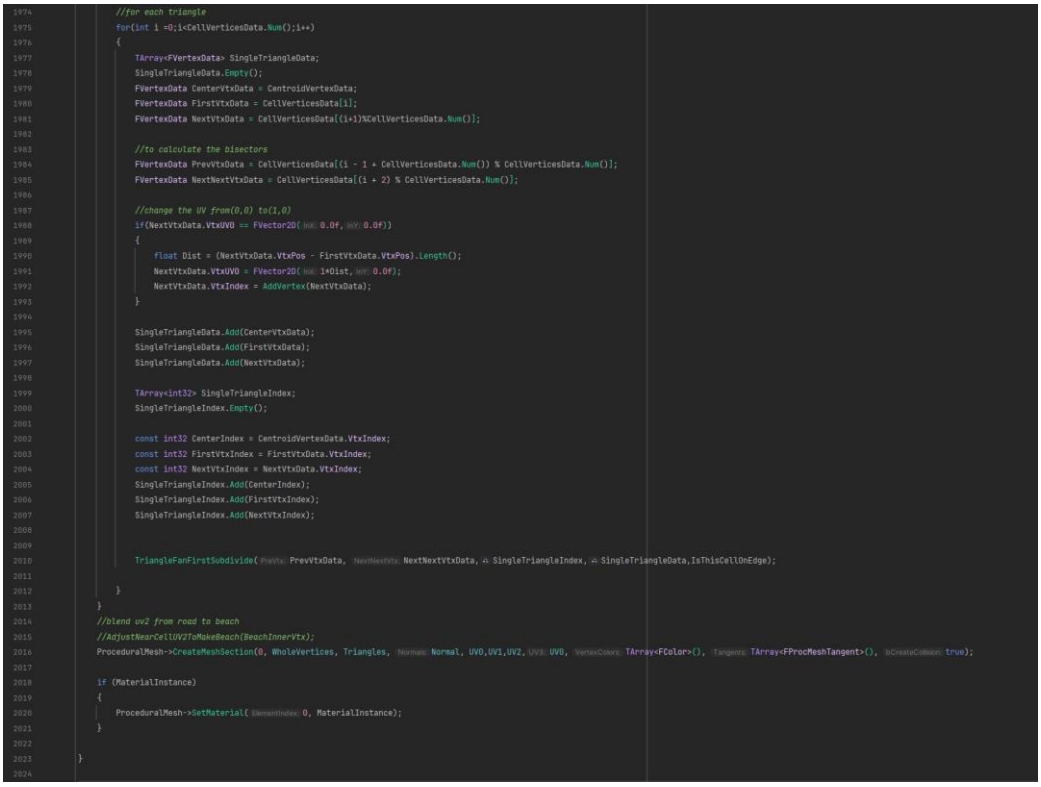
```
void UMyBlueprintFunctionLibrary::CreateVoronoiShapePolygon(UProceduralMeshComponent* ProceduralMesh, UMaterialInterface* MaterialInstance)
{
    WholeVertices.Empty();
    WholeVerticesIndex.Empty();
    //GlobalVertexIndexMap.Empty();
    Triangles.Empty();
    UV0.Empty();
    UV1.Empty();
    UV2.Empty();
    Normal.Empty();

    //this array to store the vtx that are not the beach edge to sea, but the edge between road and beach
    TArray<FVector> BeachInnerVtx;
    BeachInnerVtx.Empty();

    int CellCounter = 0;

    for(FCellStruct& CurrentCell : Cells)
    {
        CellCounter++;

        bool IsThisCellOnEdge = false;

        //check if the cell on the edge, make the beach
        for(int i =0;i<CurrentCell.VerticesPosition.Num(); i++)
        {

            if(CurrentCell.VerticesPosition[i].X== 1||CurrentCell.VerticesPosition[i].X== TextureResolutionInX-2 ||CurrentCell.VerticesPosition[i].Y ==1 || CurrentCell.VerticesPosition[i].Y ==TextureResolutionInY -2 )
            {
                IsThisCellOnEdge =  true;
                //create slope
                CurrentCell.VerticesPosition[i].Z -= 20.0f;
            }

        }
        //add beach vtx between road and beach
        if(IsThisCellOnEdge)
        {
            for(int i =0;i<CurrentCell.VerticesPosition.Num(); i++)
            {

                if(CurrentCell.VerticesPosition[i].X!= 1&&CurrentCell.VerticesPosition[i].X!= TextureResolutionInX-2&&CurrentCell.VerticesPosition[i].Y !=1 && CurrentCell.VerticesPosition[i].Y !=TextureResolutionInY -2 )
                {
                    BeachInnerVtx.Add(CurrentCell.VerticesPosition[i]);

                }

            }
        }
```

Figure 59: Voronoi Tessellation Code



```
        }



        //get center point
        FVector CentroidPos = CurrentCell.CalculateIncenter();
        FVector CenterNormal( inX: 0,  inY: 0,  inZ: 1);
        FVector2D CenterUV0( inX: 0.0f,  inY: 0.0f);
        FVector2D CenterUV1( inX: CentroidPos.X / UVScale,  inY: CentroidPos.Y / UVScale);
        //road
        FVector2D CenterUV2( inX: 0,  inY: 0);


        //set beach
        if(IsThisCellOnEdge)
        {
            CenterUV2 = FVector2D( inX: 1.5, inY: 0);
        }

        FVertexData CentroidVertexData(CentroidPos, inVtxIndex: 0, inVtxUV0: CenterUV0, inVtxUV1: CenterUV1, inVtxUV2: CenterUV2, CenterNormal );
        CentroidVertexData.VtxIndex = AddVertex(CentroidVertexData);

        //
        TArray<FVertexData> CellVerticesData;
        CellVerticesData.Empty();

        //map index
        for(int i =0;i<CurrentCell.VerticesPosition.Num(); i++)
        {
            FVector VtxPos = CurrentCell.VerticesPosition[i];
            FVector VtxNormal( inX: 0,  inY: 0,  inZ: 1);
            FVector2D VtxUV0( inX: 0.0f,  inY: 0.0f);
            FVector2D VtxUV1( inX: VtxPos.X / UVScale,  inY: VtxPos.Y / UVScale);
            FVector2D VtxUV2( inX: 0,  inY: 0);

            if(IsThisCellOnEdge)
            {
                VtxUV2 = FVector2D( inX: 1.5, inY: 0);
            }

            FVertexData CurrentVtxData(VtxPos,  inVtxIndex: 0,  inVtxUV0: VtxUV0,  inVtxUV1: VtxUV1,  inVtxUV2: VtxUV2, VtxNormal);
            CurrentVtxData.VtxIndex = AddVertex(CurrentVtxData);
            CellVerticesData.Add(CurrentVtxData);

        }

        //check winding order
        CheckWindingOrder( [&] CellVerticesData, EPlane::XY);



        //for each triangle
```

Figure 60: Voronoi Tessellation Code



```
        //for each triangle
        for(int i =0;i<CellVerticesData.Num();i++)
        {
            TArray<FVertexData> SingleTriangleData;
            SingleTriangleData.Empty();
            FVertexData CenterVtxData = CentroidVertexData;
            FVertexData FirstVtxData = CellVerticesData[i];
            FVertexData NextVtxData = CellVerticesData[(i+1)%CellVerticesData.Num()];

            //to calculate the bisectors
            FVertexData PrevVtxData = CellVerticesData[(i - 1 + CellVerticesData.Num()) % CellVerticesData.Num()];
            FVertexData NextNextVtxData = CellVerticesData[(i + 2) % CellVerticesData.Num()];

            //change the UV from(0,0) to(1,0)
            if(NextVtxData.VtxUV0 == FVector2D( inX: 0.0f, inY: 0.0f))
            {
                float Dist = (NextVtxData.VtxPos - FirstVtxData.VtxPos).Length();
                NextVtxData.VtxUV0 = FVector2D( inX: 1*Dist, inY: 0.0f);
                NextVtxData.VtxIndex = AddVertex(NextVtxData);
            }

            SingleTriangleData.Add(CenterVtxData);
            SingleTriangleData.Add(FirstVtxData);
            SingleTriangleData.Add(NextVtxData);

            TArray<int32> SingleTriangleIndex;
            SingleTriangleIndex.Empty();

            const int32 CenterIndex = CentroidVertexData.VtxIndex;
            const int32 FirstVtxIndex = FirstVtxData.VtxIndex;
            const int32 NextVtxIndex = NextVtxData.VtxIndex;
            SingleTriangleIndex.Add(CenterIndex);
            SingleTriangleIndex.Add(FirstVtxIndex);
            SingleTriangleIndex.Add(NextVtxIndex);


            TriangleFanFirstSubdivide( PrevVtx: PrevVtxData, NextNextVtx: NextNextVtxData, & SingleTriangleIndex, & SingleTriangleData,IsThisCellOnEdge);

        }
    }
    //blend uv2 from road to beach
    //AdjustNearCellUV2ToMakeBeach(BeachInnerVtx);
    ProceduralMesh->CreateMeshSection(0, WholeVertices, Triangles, Normals: Normal, UV0,UV1,UV2, UV3: UV0, VertexColors: TArray<FColor>(), Tangents: TArray<FProcMeshTangent>(), bCreateCollision: true);

    if (MaterialInstance)
    {
        ProceduralMesh->SetMaterial( ElementIndex: 0, MaterialInstance);
    }

}
```

Figure 61: Voronoi Tessellation Code

This code is to conduct the Voronoi Tessellation (Mesh construction). To the purpose of demonstration, I skipped many data processing code to make it clear. Essentially, this step is foundation of the geometry creation part, and it requires a lot of mathematical calculations to get the result. I had a thought on many scenarios, and finally used this structure and it also have a deep influence on the later subdivision and building extrusion section.

4.3. Building Block's Generation

After road map generation, it is the time to generate building blocks. The blocks are based on road map but splitting the whole plane into many small pieces. There are three main steps in this process, which are subdivide meshes (each Voronoi cell), normal calculation as well as UV mapping.

1. Subdivision

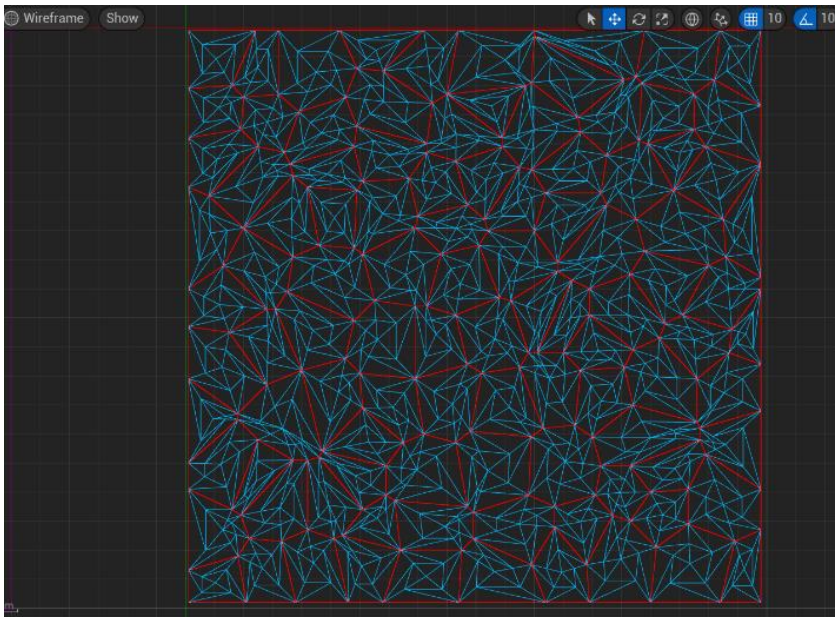
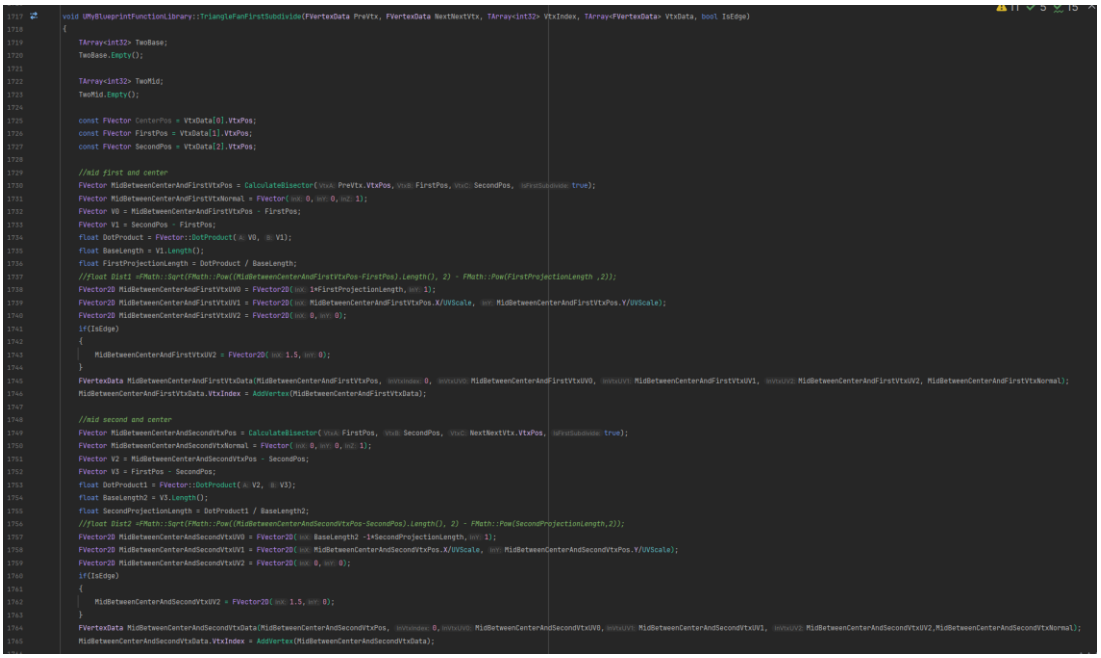


Figure 62: Subdivision

My Code Section Below:



```
void UMyBlueprintFunctionLibrary::TriangleFanFirstSubdivide(FVertexData PreVtx, FVertexData NextNextVtx, TArray<int32> VtxIndex, TArray<FVertexData> VtxData, bool IsEdge)
{
    TArray<int32> TwoBase;
    TwoBase.Empty();

    TArray<int32> TwoMid;
    TwoMid.Empty();

    const FVector CenterPos = VtxData[0].VtxPos;
    const FVector FirstPos = VtxData[1].VtxPos;
    const FVector SecondPos = VtxData[2].VtxPos;

    //mid first and center
    FVector MidBetweenCenterAndFirstVtxPos = CalculateBisector(VtxA: PreVtx.VtxPos, VtxB: FirstPos, VtxC: SecondPos, bFirstSubdivide: true);
    FVector MidBetweenCenterAndFirstVtxNormal = FVector(nX: 0, nY: 0, nZ: 1);
    FVector V0 = MidBetweenCenterAndFirstVtxPos - FirstPos;
    FVector V1 = SecondPos - FirstPos;
    float DotProduct = FVector::DotProduct(A: V0, B: V1);
    float BaseLength = V1.Length();
    float FirstProjectionLength = DotProduct / BaseLength;
    //float Dist1 =FMath::Sqrt(FMath::Pow((MidBetweenCenterAndFirstVtxPos-FirstPos).Length(), 2) - FMath::Pow(FirstProjectionLength ,2));
    FVector2D MidBetweenCenterAndFirstVtxUV0 = FVector2D(nX: 1*FirstProjectionLength, nY: 1);
    FVector2D MidBetweenCenterAndFirstVtxUV1 = FVector2D(nX: MidBetweenCenterAndFirstVtxPos.X/UVScale, nY: MidBetweenCenterAndFirstVtxPos.Y/UVScale);
    FVector2D MidBetweenCenterAndFirstVtxUV2 = FVector2D(nX: 0, nY: 0);
    if(IsEdge)
    {
        MidBetweenCenterAndFirstVtxUV2 = FVector2D(nX: 1.5, nY: 0);
    }
    FVertexData MidBetweenCenterAndFirstVtxData(MidBetweenCenterAndFirstVtxPos, nVtxIndex: 0, nVtxUV0: MidBetweenCenterAndFirstVtxUV0, nVtxUV1: MidBetweenCenterAndFirstVtxUV1, nVtxUV2: MidBetweenCenterAndFirstVtxUV2, MidBetweenCenterAndFirstVtxNormal);
    MidBetweenCenterAndFirstVtxData.VtxIndex = AddVertex(MidBetweenCenterAndFirstVtxData);

    //mid second and center
    FVector MidBetweenCenterAndSecondVtxPos = CalculateBisector(VtxA: FirstPos, VtxB: SecondPos, VtxC: NextNextVtx.VtxPos, bFirstSubdivide: true);
    FVector MidBetweenCenterAndSecondVtxNormal = FVector(nX: 0, nY: 0, nZ: 1);
    FVector V2 = MidBetweenCenterAndSecondVtxPos - SecondPos;
    FVector V3 = FirstPos - SecondPos;
    float DotProduct1 = FVector::DotProduct(A: V2, B: V3);
    float BaseLength2 = V3.Length();
    float SecondProjectionLength = DotProduct1 / BaseLength2;
    //float Dist2 =FMath::Sqrt(FMath::Pow((MidBetweenCenterAndSecondVtxPos-SecondPos).Length(), 2) - FMath::Pow(SecondProjectionLength,2));
    FVector2D MidBetweenCenterAndSecondVtxUV0 = FVector2D(nX: BaseLength2 -1*SecondProjectionLength, nY: 1);
    FVector2D MidBetweenCenterAndSecondVtxUV1 = FVector2D(nX: MidBetweenCenterAndSecondVtxPos.X/UVScale, nY: MidBetweenCenterAndSecondVtxPos.Y/UVScale);
    FVector2D MidBetweenCenterAndSecondVtxUV2 = FVector2D(nX: 0, nY: 0);
    if(IsEdge)
    {
        MidBetweenCenterAndSecondVtxUV2 = FVector2D(nX: 1.5, nY: 0);
    }
    FVertexData MidBetweenCenterAndSecondVtxData(MidBetweenCenterAndSecondVtxPos, nVtxIndex: 0, nVtxUV0: MidBetweenCenterAndSecondVtxUV0, nVtxUV1: MidBetweenCenterAndSecondVtxUV1, nVtxUV2: MidBetweenCenterAndSecondVtxUV2,MidBetweenCenterAndSecondVtxNormal);
    MidBetweenCenterAndSecondVtxData.VtxIndex = AddVertex(MidBetweenCenterAndSecondVtxData);
```

Figure 63: First Subdivision Code

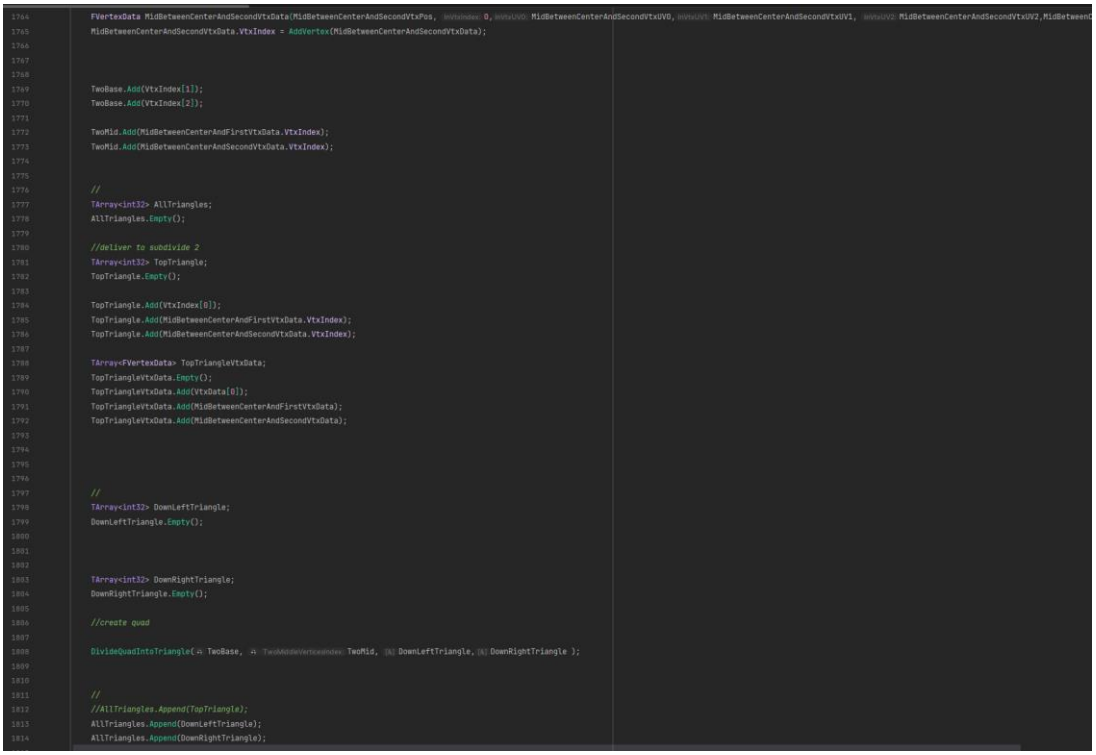


Figure 64: First Subdivision Code

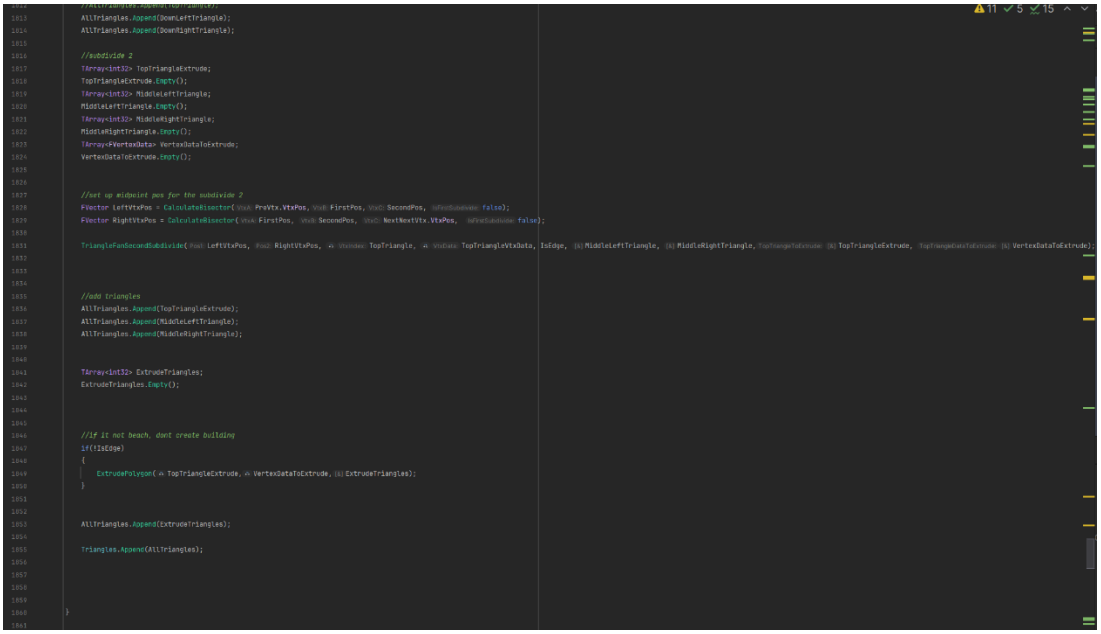


Figure 65: First Subdivision Code

This code section is to conduct the first subdivision, the logic behind it is to divide each cell into different triangles. For each of the triangle, using the bisector a direction to create and subdivided the triangle. In this way, I can create and split the road, pavement as well as the buildings' section. Later I conduct a second subdivision in 1831 line. But for space-conscious,

it won’t put the exact code in the documentation since it’s the similar logic with the first-time subdivision.

2. Normal Calculation & UV Mapping

Normal calculation and UV mapping are not very visible in the project, which means that, if looks correct then it’s correct, otherwise it need further adjustments.

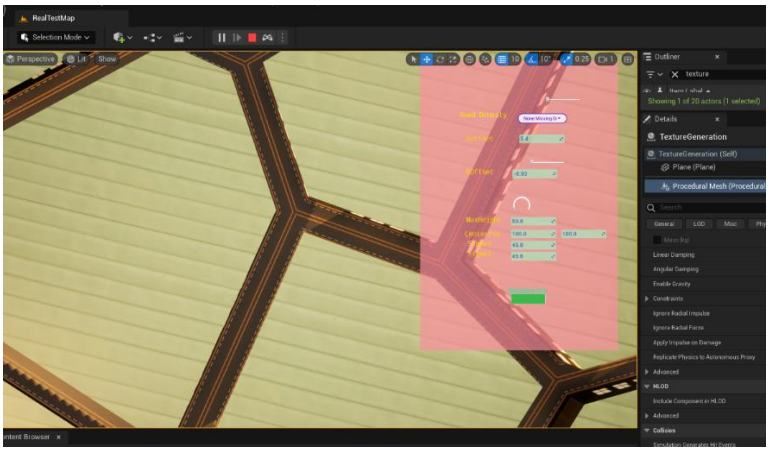


Figure 66: Normal Calculation & UV Mapping

My Code Section Below:

```
USTRUCT(BlueprintType)
struct FVertexData
{
    GENERATED_BODY()

public:
    // Position
    FVector VtxPos;
    // Index
    int32 VtxIndex;
    // Local UV
    FVector2D VtxUV0;
    // World UV
    FVector2D VtxUV1;
    // Distinguish UV Channel
    //road(0 ,0),building(0.5 ,0), roof(1.0 ,0), beach(1.5, 0), pavement(2.0, 0)
    FVector2D VtxUV2;
    // Normal
    FVector VtxNormal;

    // Default constructor with initializers
    FVertexData()
        : VtxPos(0.0f, 0.0f, 0.0f)
        , VtxIndex(0)
        , VtxUV0(0.0f, 0.0f)
        , VtxUV1(0.0f, 0.0f)
        , VtxUV2(0.0f, 0.0f)
        , VtxNormal(0.0f, 0.0f, 0.0f)
    {
    }

    // Parameterized constructor for custom initializztion
    FVertexData(const FVector& InVtxPos, int32 InVtxIndex, const FVector2D& InVtxUV0, const FVector2D& InVtxUV1, const FVector2D& InVtxUV2, const FVector& InVtxNormal)
        : VtxPos(InVtxPos)
        , VtxIndex(InVtxIndex)
        , VtxUV0(InVtxUV0)
        , VtxUV1(InVtxUV1)
        , VtxUV2(InVtxUV2)
        , VtxNormal(InVtxNormal)
    {
    }

    // Equality operator to support comparison
    friend bool operator==(const FVertexData& Lhs, const FVertexData& Rhs)
    {
        return Lhs.VtxPos == Rhs.VtxPos &&
               Lhs.VtxIndex == Rhs.VtxIndex &&
               Lhs.VtxUV0 == Rhs.VtxUV0 &&
               Lhs.VtxUV1 == Rhs.VtxUV1 &&
               Lhs.VtxUV2 == Rhs.VtxUV2 &&
               Lhs.VtxNormal == Rhs.VtxNormal;
    }
```

Figure 67: Normal Calculation & UV Mapping Code

Since this code should be contained in each of code sections mentioned above, so it will not provide too much code example here. But the logic behind is, when calculating normal, use cross product on the building's normal calculation. And for the UV Mapping, I used three UV Channels to define the right UV. The first UV0 is for local UV, second UV1 is for global UV, third UV2 is for distinguish whether to use UV0 or UV1 and use which texture in the material to demonstrate this polygon.

4.4. Building Generation

The generation of building blocks give a solid foundation on building's generation. The building generation section is based on building block's subdivision process, for each of the subdivided mesh, there will be a building extrude here. There are three main steps of the building generation, which are extrude buildings, UV mapping, and defining height.

1. **Extrusion**

- In wireframe mode

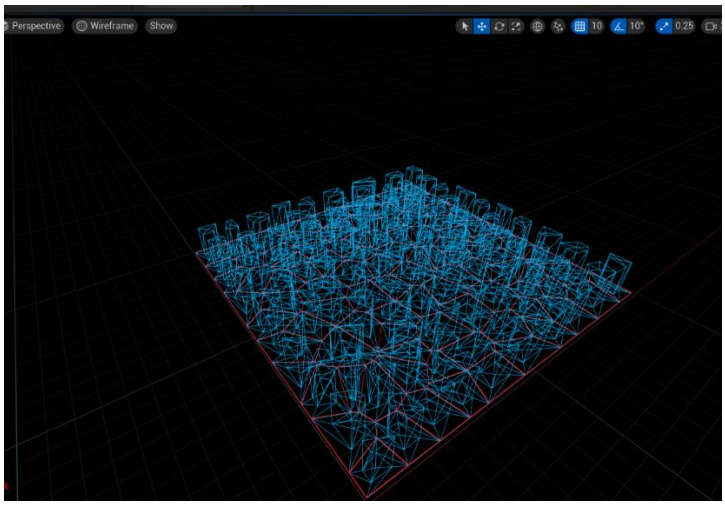


Figure 68: Extrusion in Wireframe Mode

- In view mode

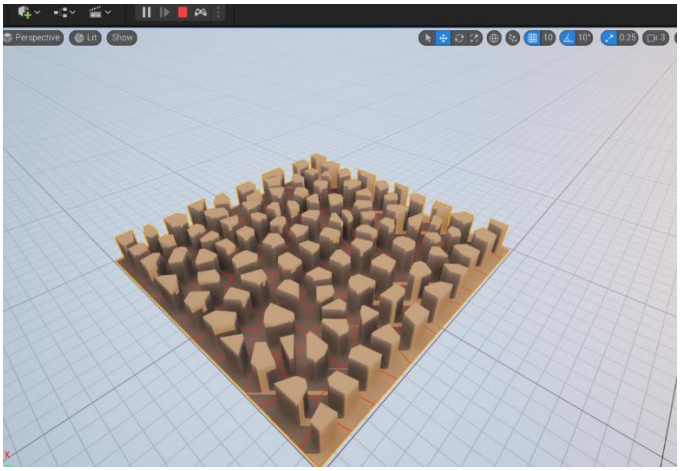
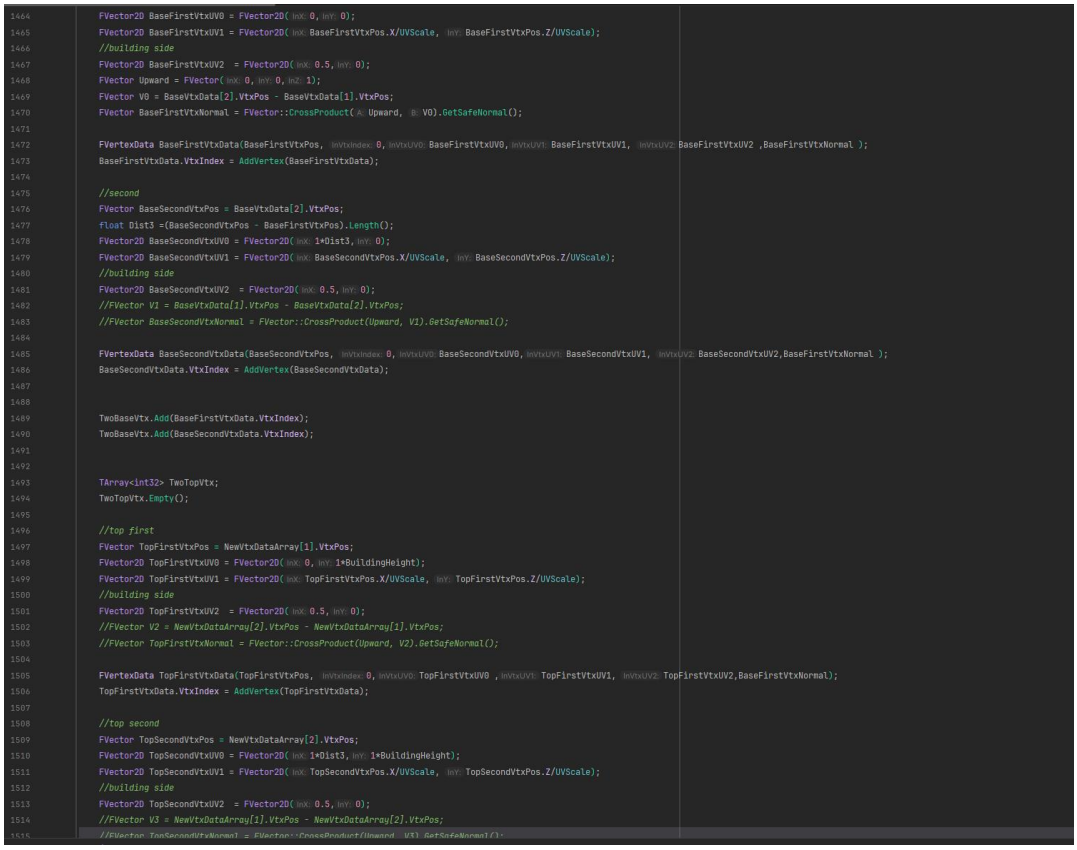


Figure 69: Extrusion in View Mode

My Code Section Below:

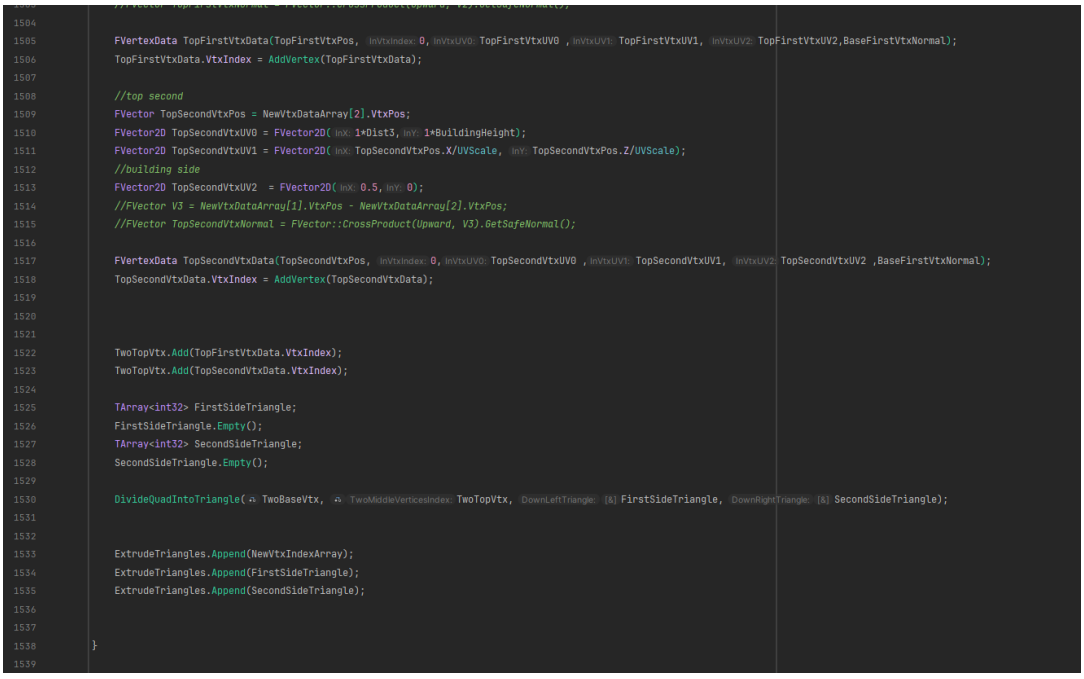
```
void UMyBlueprintFunctionLibrary::ExtrudePolygon(TArray<int32> BaseTriangle, TArray<FVertexData> BaseVtxData, TArray<int32>& ExtrudeTriangles)
{

    ExtrudeTriangles.Empty();

    TArray<int32> NewVtxIndexArray;
    NewVtxIndexArray.Empty();
    TArray<FVertexData> NewVtxDataArray;
    NewVtxDataArray.Empty();
    //setup building height
    FVector2D BuildingCenter = FVector2D(BaseVtxData[0].VtxPos.X,BaseVtxData[0].VtxPos.Y);
    float BuildingHeight = UseNormalDistributionToGetBuildingHeight(BuildingCenter);
    float NoiseScale = MaxHeight * 2.0f;
    float Noise = FMath::PerlinNoise2D( Location: BuildingCenter / 10.0f) * NoiseScale;
    BuildingHeight+=Noise;
    //change building as grass
    if(BuildingHeight<=GreenAreaAmount)
    {
        BuildingHeight = 0.01;
    }

    for(int i =0; i<BaseVtxData.Num();i++)
    {
        FVector TopNewVtxPos = BaseVtxData[i].VtxPos;
        //float Height = 20.0f;
        TopNewVtxPos.Z += BuildingHeight;
        FVector NewVtxNormal = FVector( inX: 0, inY: 0, inZ: 1);
        FVector2D NewVtxUV0 = FVector2D( inX: 0, inY: 0);
        FVector2D NewVtxUV1 = FVector2D( inX: TopNewVtxPos.X/UVScale, inY: TopNewVtxPos.Y/UVScale);
        //roof
        FVector2D NewVtxUV2 = FVector2D( inX: 1, inY: 0);
        //set as green
        if(BuildingHeight<=GreenAreaAmount)
        {
            NewVtxUV2 = FVector2D( inX: 2.5, inY: 0);
        }
        FVertexData NewVtxData(TopNewVtxPos, inVtxIndex: 0, inVtxUV0: NewVtxUV0, inVtxUV1: NewVtxUV1, inVtxUV2: NewVtxUV2,NewVtxNormal );
        NewVtxData.VtxIndex = AddVertex(NewVtxData);
        NewVtxIndexArray.Add(NewVtxData.VtxIndex);
        NewVtxDataArray.Add(NewVtxData);

    }


    TArray<int32> TwoBaseVtx;
    TwoBaseVtx.Empty();
    //first
    FVector BaseFirstVtxPos = BaseVtxData[1].VtxPos;
    FVector2D BaseFirstVtxUV0 = FVector2D( inX: 0, inY: 0);
```

Figure 70: Extrusion Code



```
FVector2D BaseFirstVtxUV0 = FVector2D( inX: 0, inY: 0);
FVector2D BaseFirstVtxUV1 = FVector2D( inX: BaseFirstVtxPos.X/UVScale, inY: BaseFirstVtxPos.Z/UVScale);
//building side
FVector2D BaseFirstVtxUV2  = FVector2D( inX: 0.5, inY: 0);
FVector Upward = FVector( inX: 0, inY: 0, inZ: 1);
FVector V0 = BaseVtxData[2].VtxPos - BaseVtxData[1].VtxPos;
FVector BaseFirstVtxNormal = FVector::CrossProduct( A: Upward,  B: V0).GetSafeNormal();

FVertexData BaseFirstVtxData(BaseFirstVtxPos,  inVtxIndex: 0, inVtxUV0: BaseFirstVtxUV0, inVtxUV1: BaseFirstVtxUV1,  inVtxUV2: BaseFirstVtxUV2 ,BaseFirstVtxNormal );
BaseFirstVtxData.VtxIndex = AddVertex(BaseFirstVtxData);

//second
FVector BaseSecondVtxPos = BaseVtxData[2].VtxPos;
float Dist3 =(BaseSecondVtxPos - BaseFirstVtxPos).Length();
FVector2D BaseSecondVtxUV0 = FVector2D( inX: 1*Dist3, inY: 0);
FVector2D BaseSecondVtxUV1 = FVector2D( inX: BaseSecondVtxPos.X/UVScale,  inY: BaseSecondVtxPos.Z/UVScale);
//building side
FVector2D BaseSecondVtxUV2  = FVector2D( inX: 0.5, inY: 0);
//FVector V1 = BaseVtxData[1].VtxPos - BaseVtxData[2].VtxPos;
//FVector BaseSecondVtxNormal = FVector::CrossProduct(Upward, V1).GetSafeNormal();

FVertexData BaseSecondVtxData(BaseSecondVtxPos,  inVtxIndex: 0, inVtxUV0: BaseSecondVtxUV0, inVtxUV1: BaseSecondVtxUV1,  inVtxUV2: BaseSecondVtxUV2,BaseFirstVtxNormal );
BaseSecondVtxData.VtxIndex = AddVertex(BaseSecondVtxData);


TwoBaseVtx.Add(BaseFirstVtxData.VtxIndex);
TwoBaseVtx.Add(BaseSecondVtxData.VtxIndex);


TArray<int32> TwoTopVtx;
TwoTopVtx.Empty();

//top first
FVector TopFirstVtxPos = NewVtxDataArray[1].VtxPos;
FVector2D TopFirstVtxUV0 = FVector2D( inX: 0, inY: 1*BuildingHeight);
FVector2D TopFirstVtxUV1 = FVector2D( inX: TopFirstVtxPos.X/UVScale,  inY: TopFirstVtxPos.Z/UVScale);
//building side
FVector2D TopFirstVtxUV2  = FVector2D( inX: 0.5, inY: 0);
//FVector V2 = NewVtxDataArray[2].VtxPos - NewVtxDataArray[1].VtxPos;
//FVector TopFirstVtxNormal = FVector::CrossProduct(Upward, V2).GetSafeNormal();

FVertexData TopFirstVtxData(TopFirstVtxPos,  inVtxIndex: 0, inVtxUV0: TopFirstVtxUV0 , inVtxUV1: TopFirstVtxUV1,  inVtxUV2: TopFirstVtxUV2,BaseFirstVtxNormal);
TopFirstVtxData.VtxIndex = AddVertex(TopFirstVtxData);

//top second
FVector TopSecondVtxPos = NewVtxDataArray[2].VtxPos;
FVector2D TopSecondVtxUV0 = FVector2D( inX: 1*Dist3, inY: 1*BuildingHeight);
FVector2D TopSecondVtxUV1 = FVector2D( inX: TopSecondVtxPos.X/UVScale,  inY: TopSecondVtxPos.Z/UVScale);
//building side
FVector2D TopSecondVtxUV2  = FVector2D( inX: 0.5, inY: 0);
//FVector V3 = NewVtxDataArray[1].VtxPos - NewVtxDataArray[2].VtxPos;
//FVector TopSecondVtxNormal = FVector::CrossProduct(Upward, V3).GetSafeNormal();
```

Figure 71: Extrusion Code



```
FVertexData TopFirstVtxData(TopFirstVtxPos,  inVtxIndex: 0, inVtxUV0: TopFirstVtxUV0 , inVtxUV1: TopFirstVtxUV1,  inVtxUV2: TopFirstVtxUV2,BaseFirstVtxNormal);
TopFirstVtxData.VtxIndex = AddVertex(TopFirstVtxData);

//top second
FVector TopSecondVtxPos = NewVtxDataArray[2].VtxPos;
FVector2D TopSecondVtxUV0 = FVector2D( inX: 1*Dist3, inY: 1*BuildingHeight);
FVector2D TopSecondVtxUV1 = FVector2D( inX: TopSecondVtxPos.X/UVScale,  inY: TopSecondVtxPos.Z/UVScale);
//building side
FVector2D TopSecondVtxUV2  = FVector2D( inX: 0.5, inY: 0);
//FVector V3 = NewVtxDataArray[1].VtxPos - NewVtxDataArray[2].VtxPos;
//FVector TopSecondVtxNormal = FVector::CrossProduct(Upward, V3).GetSafeNormal();

FVertexData TopSecondVtxData(TopSecondVtxPos,  inVtxIndex: 0, inVtxUV0: TopSecondVtxUV0 , inVtxUV1: TopSecondVtxUV1,  inVtxUV2: TopSecondVtxUV2 ,BaseFirstVtxNormal);
TopSecondVtxData.VtxIndex = AddVertex(TopSecondVtxData);




TwoTopVtx.Add(TopFirstVtxData.VtxIndex);
TwoTopVtx.Add(TopSecondVtxData.VtxIndex);

TArray<int32> FirstSideTriangle;
FirstSideTriangle.Empty();
TArray<int32> SecondSideTriangle;
SecondSideTriangle.Empty();

DivideQuadIntoTriangle( TwoBaseVtx,  TwoMiddleVerticesIndex: TwoTopVtx,  DownLeftTriangle: [&] FirstSideTriangle,  DownRightTriangle: [&] SecondSideTriangle);


ExtrudeTriangles.Append(NewVtxIndexArray);
ExtrudeTriangles.Append(FirstSideTriangle);
ExtrudeTriangles.Append(SecondSideTriangle);


}
```

Figure 72: Extrusion Code

This code is simply to extrude each of the triangle fan. The logic behind it is to create new set of points according to the buildings height and create other two face (top and side) of this fan's section.

2. UV Mapping

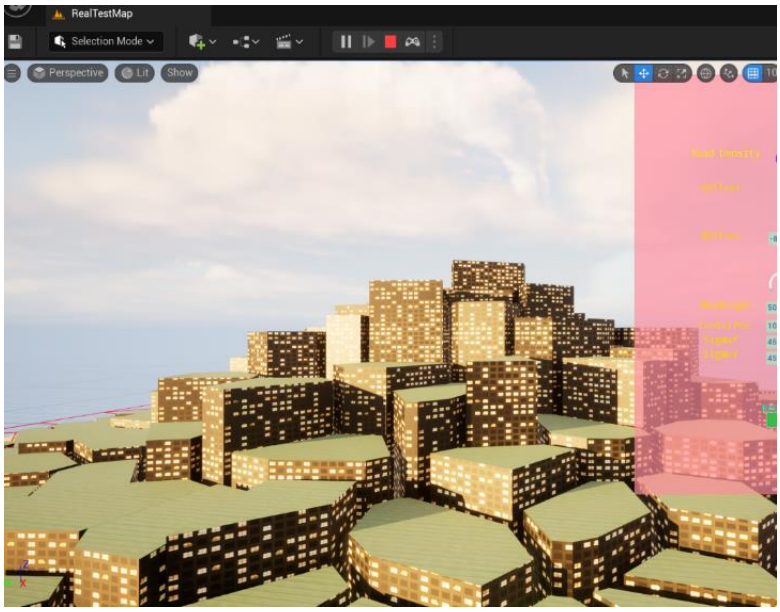


Figure 73: UV Mapping

The building's UV Mapping code is similar to the UV Mapping section mentioned before, so the code will not be shown here.

3. Defining Building Height

The way to define the building's height is through 2D Gaussian distribution with the combination of Perlin noise. Using a certain weight to balance these two values, the city's shape can be more smoothly natural and fit the statistically significance.

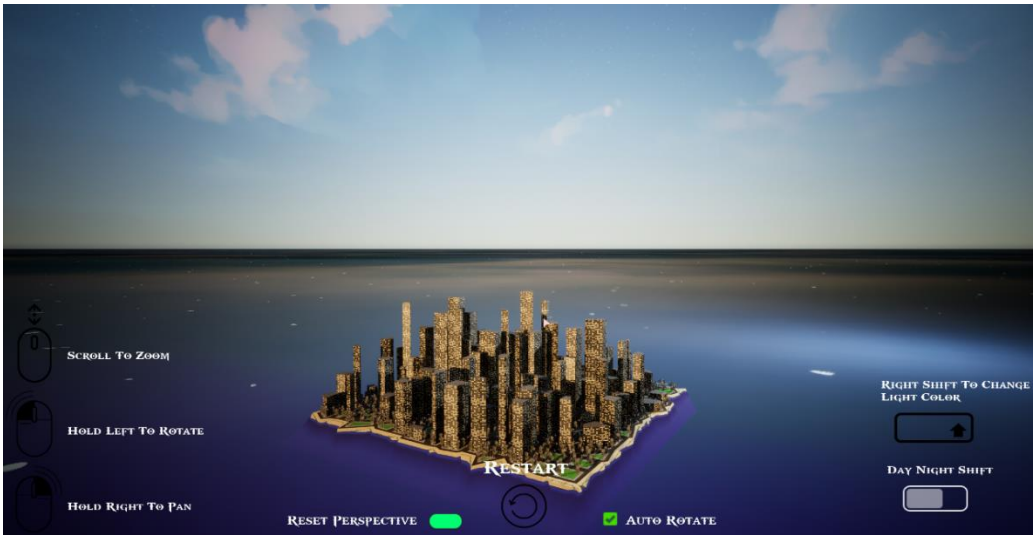


Figure 74: Defining Building Height

My Code Section Below:

```
float UMyBlueprintFunctionLibrary::UseNormalDistributionToGetBuildingHeight(FVector2D CurrentPos)
{
    const float exponent = FMath::Exp(Value: -(FMath::Square(A: CurrentPos.X - CityCenterPos.X) / (2 * SigmaX * SigmaX)
                         + FMath::Square(A: CurrentPos.Y - CityCenterPos.Y) / (2 * SigmaY * SigmaY)));
    UE_LOG(LogTemp, Warning, TEXT("exponent: %f"), exponent);

    return MaxHeight * exponent;


}
```

Figure 75: Defining Building Height Code

This code is the 2D Gaussian distribution implementation. The Pelrin noise implementation has already shown in the building extrusion code section.

4.5. Pre-made Infrastructures Generation

The infrastructures generation is the last part of the generation algorithm code section. To generate pre-made infrastructures, some random points is scattered around the building blocks. Then, the scattered points can use used as the location reference of the object's mesh.

The tree and streetlight are scattered as below (Figure 76):

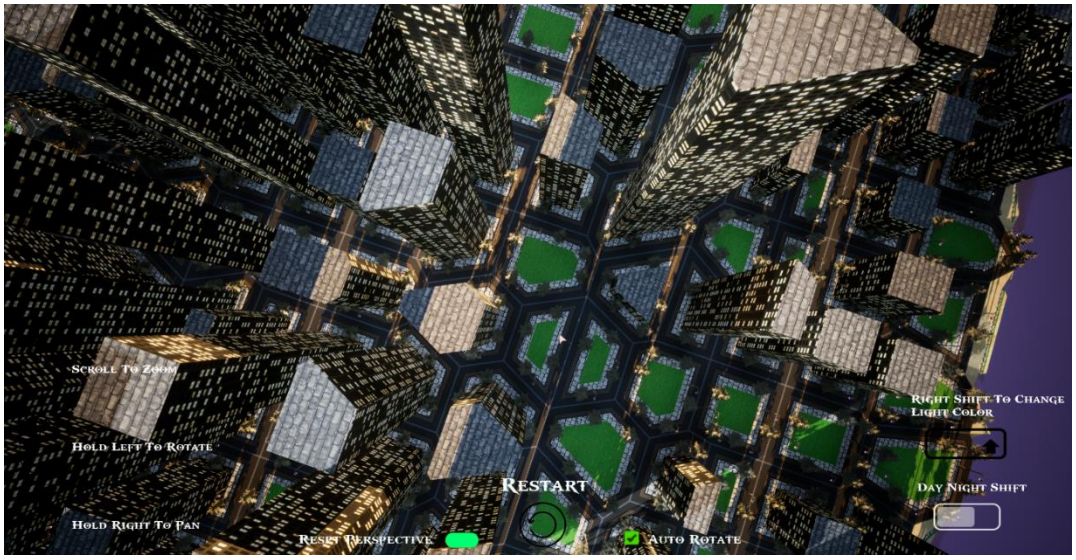


Figure 76: Pre-made Infrastructures Generation

My Code Section Below (Figure 77):

```
void UMyBlueprintFunctionLibrary::InstantiateObject(UInstancedStaticMeshComponent* StaticMesh, FVector WorldLocation)
{

    WorldLocation.Z += 0.01f; // Adjust Z position slightly
    const FTransform InstantiateTransform = FTransform( InRotation: FRotator( InPitch: 0, InYaw: 0, InRoll: 0), InTranslation: WorldLocation, InScale3D: FVector( InX: 0.01, InY: 0.01, InZ: 0.01));

    StaticMesh->AddInstance(InstantiateTransform);

}

void UMyBlueprintFunctionLibrary::ScatterPointsInSquare(FVector BaseLeftPos, FVector BaseRightPos, FVector TopLeftPos,FVector TopRightPos,int32 NumberOfObjects)
{
    constexpr int32 Slider = 0.2;
    const FVector LeftMiddlePos = BaseLeftPos+Slider*(TopLeftPos-BaseLeftPos);
    const FVector RightMiddlePos = BaseRightPos+Slider*(TopRightPos-BaseRightPos);
    //divide into n points

    const FVector Dist = RightMiddlePos - LeftMiddlePos;
    for(int i = 1; i<NumberOfObjects; i++)
    {
        const float Ratio = static_cast<float>(i) / static_cast<float>(NumberOfObjects);
        const FVector InstantiateLocation = LeftMiddlePos + Ratio*Dist;
        if(i%2!=0)
        {
            InstantiateObject(Tree, InstantiateLocation);
        }

        else
        {
            InstantiateObject(StreetLight, InstantiateLocation);
        }

    }

}
```

Figure 77: Pre-made Infrastructures Generation Code

This code's logic is quite straightforward. Through distinguish the scattered point is odd or even, to copy the tree or streetlight to this point's location.

4.6. UI Design

The UI Section is designed on Figma as below (Figure 78):

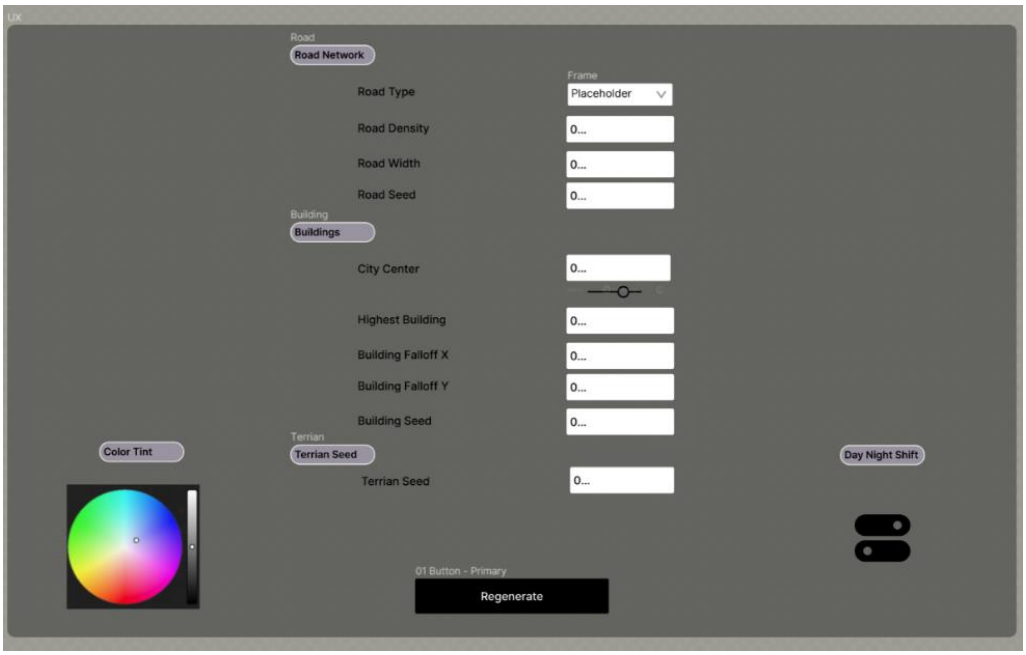


Figure 78: UI Design on Figma

The UI is implemented in Unreal Engine 5 as below (Figure 79):

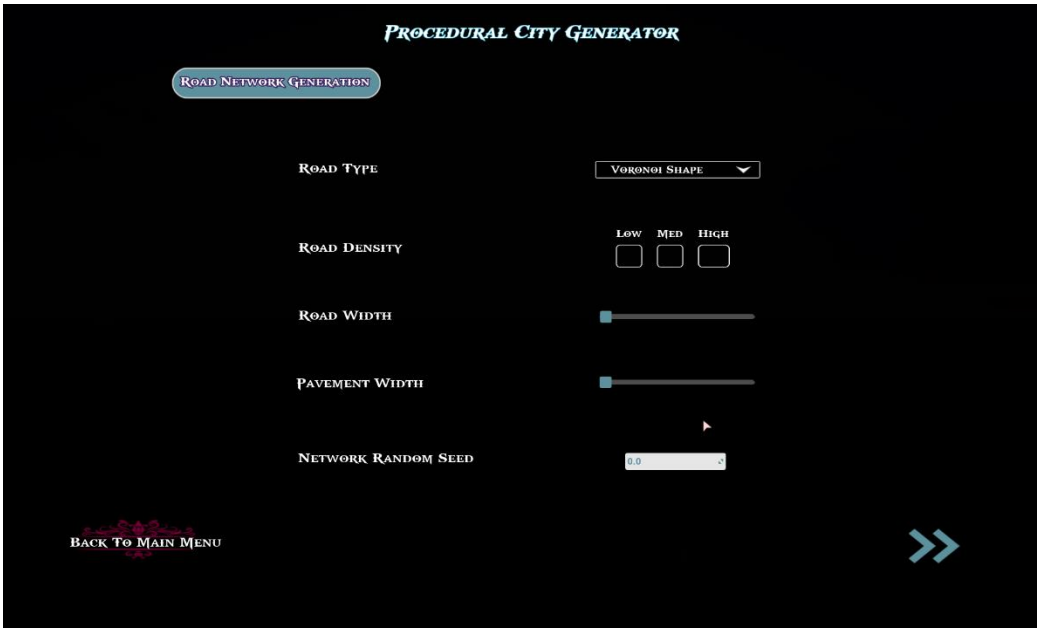


Figure 79: UI Implemented in Unreal Engine 5

The Blueprint interface as the front to connect with the C++ code as below (Figure 80):

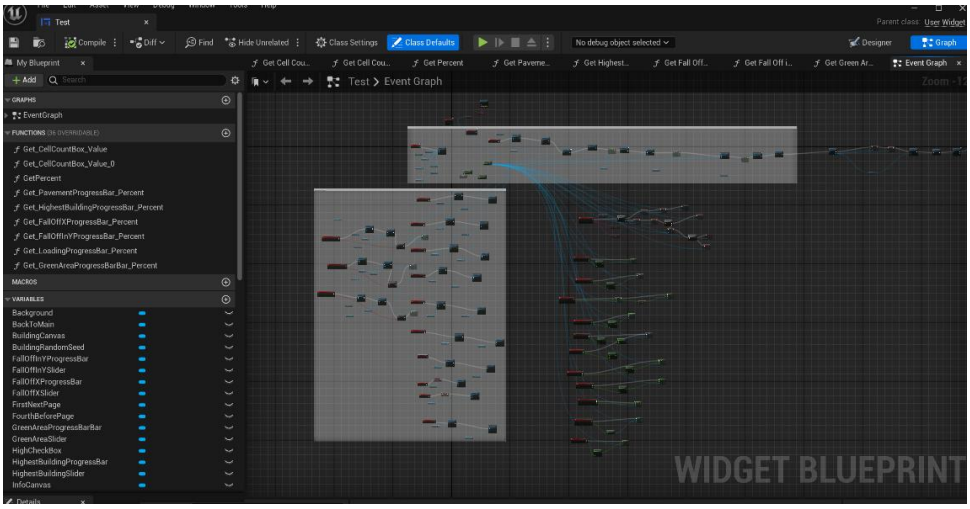


Figure 80: UI Blueprint

Since this UI section is essentially quite easy to implement, the only necessary part is to write the parameters' interface decently in the C++ code and use the blueprint to clamp or lerp each parameters' value and pass it back to backend.

4.7. Viewport Design

The viewport is designed as below (Figure 81):

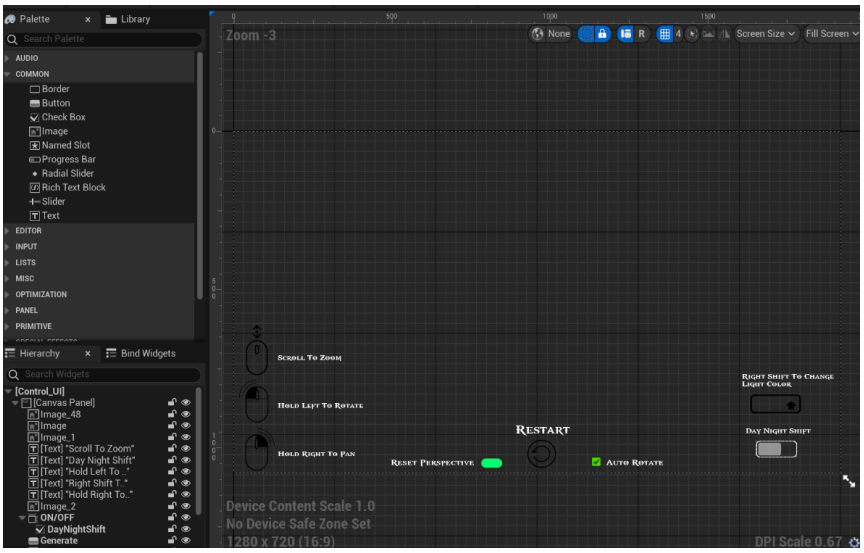


Figure 81: Viewport Design

Hold left to rotate, hold right to pan, middle mouse scrolls to zoom. A button designed for day-night shift. Right shift for change the light colour. Green button for reset the perspective. A checker for start or finish the auto rotation of the city mesh. And a restart button for regenerate a city with different parameters.

The Blueprint logic is below (Figure 82&83):

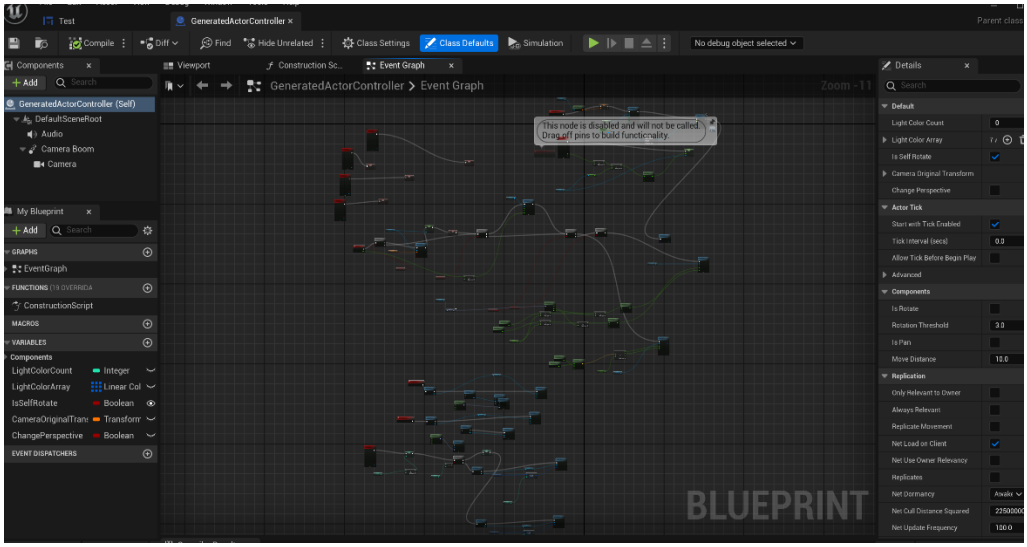


Figure 82: Viewport Design Blueprint

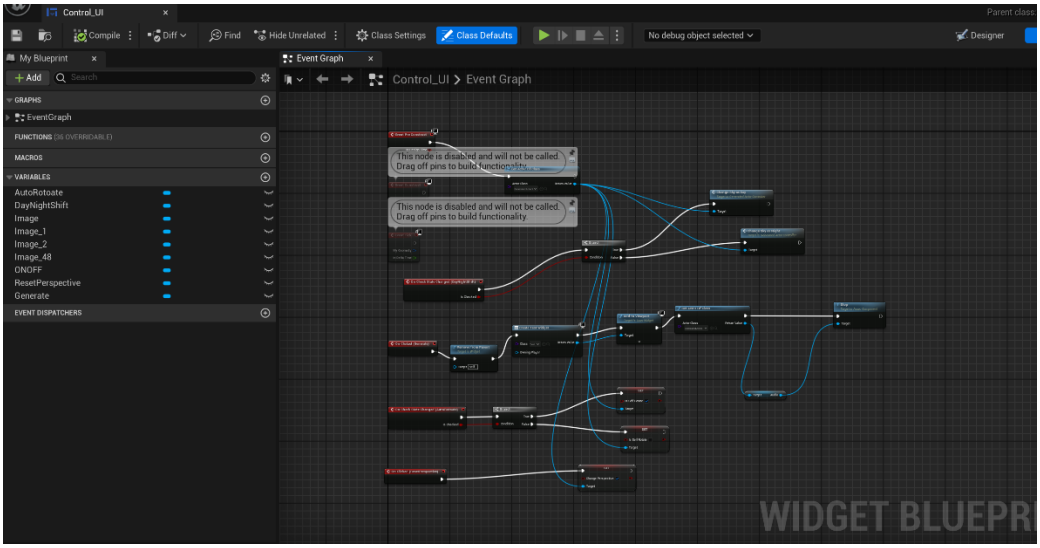


Figure 83: Viewport Design Blueprint

The logic behind the blueprint is to adjust the camera's rotation and offset according to the mouse's movement in X-axis and Y-axis. In this way, we can observe the city in different perspectives. For the day-night shift, I chose to change the skybox to create the day/night effects. Changing the light colour is also quite simple, just directly change the directional light colour in the blueprint.

This section above is the exact working process of the whole city generator in different stages. Due to the constraints of space, only core code and blueprints will be demonstrated in this documentation. But essentially, the whole procedural city generator is written in a quite tight, precise and complex approach. Every step took a long time to think, create, iterate and recreate.

5. Conclusion

In conclusion, Voronoi Diagram Based Procedural City Generator can generate a customisable city through adjusting different parameters by users. However, after reflecting on the project, I have identified a few issues for further improvement.

A major issue is the complexity and length of the current codebase, which may hinder maintenance and extensibility. Despite the fact that I have written the code in a very modular way, there are still some functions that are not very readable. A further way to improve this might be to refactor some of the code modules and add more comments inside of the code.

Additionally, using a single Voronoi diagram limits the complexity and variety of city patterns that can be generated. Implementing a Voronoi Treemap (Figure 84) can provide more layers and detail for the generated city layout, and it will be more similar to the organic growth patterns of real-world cities.

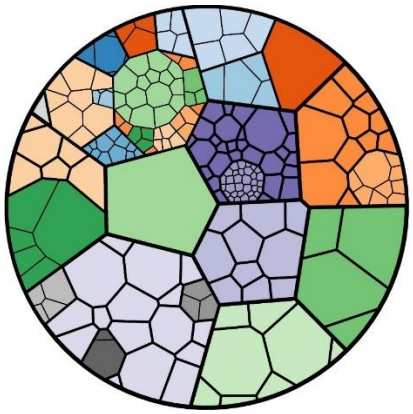


Figure 84: Voronoi Treemap, Vines & Henry, 2014

Finally, the current form of procedural generation tends to produce outputs that are repetitive and cookie-cutter, which can reduce the realism and engagement of the simulation. Introducing more meaningful narrative and interactive elements into the city generation process can greatly enhance the user experience. By embedding story-driven components and interactive features, users can engage more deeply with the generated environment, and it helps to add a layer of uniqueness and personalisation to each generated city.

6. Appendices

- **Git Commits**

Every commit of the project was recorded by Git with exact commit date and contents, the preview as below (Figure 85):

	A	B	C	D
1	7ec7e67	AudreyPeiyingZhang	2024-06-04 22:00:09 +0100	FixBugs
2	80b95b4	AudreyPeiyingZhang	2024-05-31 21:46:29 +0100	FixRotation
3	887669c	AudreyPeiyingZhang	2024-05-29 07:37:08 +0100	FinishFirstBuild
4	319bf01	AudreyPeiyingZhang	2024-05-28 21:26:50 +0100	ChangeUI5
5	609d7f5	AudreyPeiyingZhang	2024-05-28 08:33:27 +0100	ChangeUI4
6	150312e	AudreyPeiyingZhang	2024-05-27 04:54:34 +0100	ChangeUI3
7	436a04f	AudreyPeiyingZhang	2024-05-25 21:47:34 +0100	ChangeUI2
8	b5c4504	AudreyPeiyingZhang	2024-05-24 21:57:56 +0100	ChangeUI
9	6b66eff	AudreyPeiyingZhang	2024-05-24 00:49:37 +0100	AddUI
10	41d39fe	AudreyPeiyingZhang	2024-05-23 01:36:15 +0100	FinishScatter
11	6c72fea	AudreyPeiyingZhang	2024-05-22 03:07:59 +0100	FinishSecondSubdivide
12	4ca93c1	AudreyPeiyingZhang	2024-05-10 21:17:35 +0100	AddBeachArea
13	9cf4d33	AudreyPeiyingZhang	2024-05-10 00:52:57 +0100	FixPerlinNoiseAndFBM
14	d01f0e7	AudreyPeiyingZhang	2024-05-09 00:49:15 +0100	AddPerlinNoise
15	f7769ad	AudreyPeiyingZhang	2024-05-07 21:01:49 +0100	AddUIMaterials
16	903210a	AudreyPeiyingZhang	2024-05-07 01:43:06 +0100	FixConCaveCornerAndHashResetUI
17	5c89ffe	AudreyPeiyingZhang	2024-05-06 01:47:23 +0100	FinishRoadUVMapping
18	0d2a9a6	AudreyPeiyingZhang	2024-05-04 18:29:44 +0100	FixClearArray
19	ca8f585	AudreyPeiyingZhang	2024-05-03 21:18:26 +0100	FixUVBug
20	ae2a330	AudreyPeiyingZhang	2024-05-02 02:28:34 +0100	FixMergeBug
21	62596de	AudreyPeiyingZhang	2024-05-01 21:13:58 +0100	FixVoronoiBug
22	47bbb4e	AudreyPeiyingZhang	2024-04-30 21:33:12 +0100	FinishUVCalculationAndAddMaterials
23	bc2d4aa	AudreyPeiyingZhang	2024-04-29 21:52:35 +0100	AddNormalDistributionForBuildings
24	c0827e9	AudreyPeiyingZhang	2024-04-27 18:06:17 +0100	AddMaterial
25	68178e4	AudreyPeiyingZhang	2024-04-26 21:54:40 +0100	HalfWayUVAndNormalCalculations
26	43d8000	AudreyPeiyingZhang	2024-04-24 03:22:24 +0100	AddUVsAndNormalMapping
27	e216174	AudreyPeiyingZhang	2024-04-21 23:53:09 +0100	FinishExtrudeBuilding
28	3713f13	AudreyPeiyingZhang	2024-04-21 20:00:35 +0100	FinishFirstSubdivision
29	f87444d	AudreyPeiyingZhang	2024-04-20 20:55:04 +0100	ChangeTrianglatedLogic
30	a718e2e	AudreyPeiyingZhang	2024-04-19 21:50:24 +0100	AddTriangleSubdivision
31	d332f7f	AudreyPeiyingZhang	2024-04-18 02:11:24 +0100	FinishFirstStepPolyCalculation

Figure 85: Git Commits Preview

The whole excel file has been uploaded here:

https://liveuclac-my.sharepoint.com/:f:/g/personal/dtnvpz1_ucl_ac_uk/EpTJ6p_2ssFGj3Y8MvwyqNkBUejs-c-a1kIcCjCZe2jPzw

- **Git Repository**

The project is totally open-sourced, which is accessible here (includes code, scene, UI etc.):

https://github.com/AudreyPeiyingZhang/PCG-GAMEAI

- **Paper Draft**

The paper draft contains all steps' consideration when creating the procedural city generator, the preview as below (Figure 86):

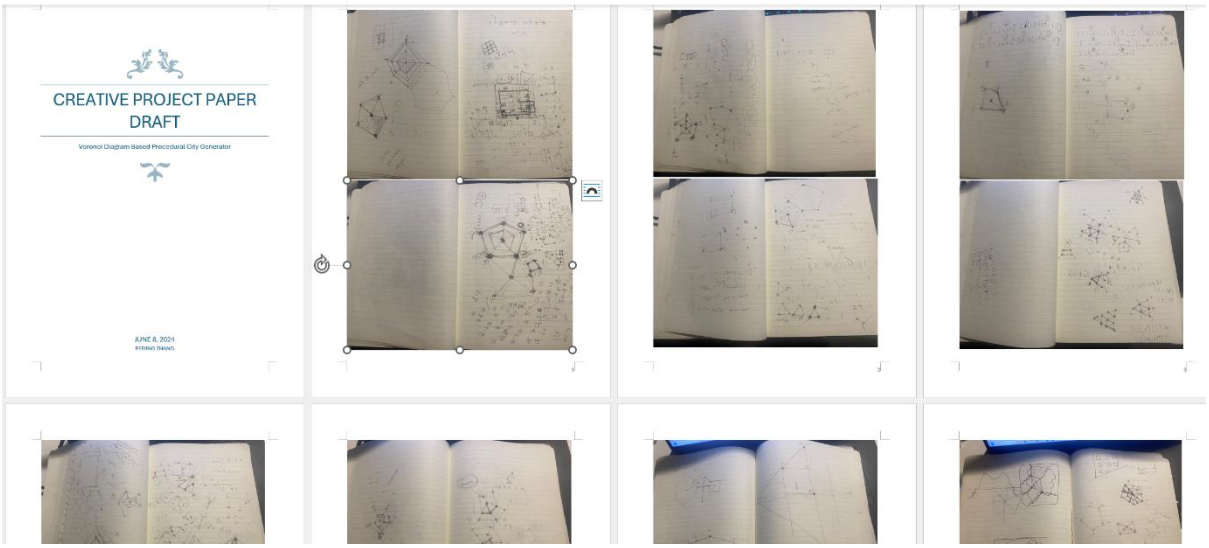


Figure 86: Paper Draft Preview

The whole PDF file has been uploaded here:

https://liveuclac-my.sharepoint.com/:f:/g/personal/dtnvpz1_ucl_ac_uk/EvCsTsqLvnNMr2ucglvw6mwB8XZR_s-yHLAD4_ELbgbmdg

- **Assets Citation**

Only two premade models were used in this project, which are Tree and Streetlight models.

The link to those models is listed below:

1. Tree:

https://sketchfab.com/3d-models/free-tree-1-fc97bf9c59e344078ec728148f210e66

2. Streetlight:

https://sketchfab.com/3d-models/various-low-poly-street-lights-1173b0c4d9b0400bbeaafbee0e94ca59

Other asset that I used in this project is a water shader, the link is listed below:

3. Water Shader:

https://www.unrealengine.com/marketplace/en-US/product/stylized-water-pack

No tutorial is used as a reference for this project.

Apart from that, I can guarantee that other mesh/material/scene/code/blueprint in this project is created by myself (not include some texture that downloaded from browser images).

7. Acknowledgements

I would like to express my gratitude to my first supervisor, Miss Jelena Viskovic, and my second supervisor, Mr. William Sykes, for their invaluable guidance and support throughout my dissertation and final creative project.

8. Bibliography

Epic Games. (n.d.). *City Sample Project: Unreal Engine Demonstration*. Unreal Engine Documentation. https://dev.epicgames.com/documentation/en-us/unreal-engine/city-sample-project-unreal-engine-demonstration?application_version=5.0

Negm, R. I. T. A. M. (2021). The Concept of "Voronoi Diagram" and its impact on the formation of Scenic Design. *International Design Journal, 11*(2), 185-199. https://doi.org/10.21608/idj.2021.152352

Redmon, J. (n.d.). Procedural city generation. Retrieved [June 8, 2024], from https://www.pjreddie.com/projects/procedural-city-generation/

Reid, M. (2022, March 8). John Snow Hunts the Blue Death. *Distillations Magazine*. Science History Institute. https://www.sciencehistory.org/stories/magazine/john-snow-hunts-the-blue-death/

Vines, P., & Henry, P. (2014). *Voronoi Treemaps in D3*. https://cse512-14w.github.io/fp-plvines-djpeter/