REPORT FOR CMP 505:

ADVANCED PROCEDURAL METHODS

# DUNGEON GENERATION BASED ON VORONOI REGIONS

Naman Merchant

## INSTRUCTIONS:

Press **L** to switch to Free Camera Mode, **B** to toggle the radial blur and **G** to toggle the bloom effect.

Player can move in all 8 directions while facing one of the cardinal directions.

Player movement: **Up, Down, Left** and **Right**.

In the free camera mode, press **R** to reset the terrain, **V** to create the dungeon again, **F** for faulting on the terrain, **S** to smoothen the terrain, **Space** to add some randomized noise and **X** to add some Perlin Noise.

Camera movement in the free camera mode is the same as how it was provided by the module: i.e. **Up Down Left Right, A, Z, Page Up** and **Page Down**

### SPECIAL INSTRUCTIONS

You can change the resolution of the terrain (line 114 of "applicationClass.cpp") to create larger/smaller rooms. Additionally, you can also change the number of rooms that can be generated in the voronoi diagram (line 53 of "Voronoi.cpp").

To show all the voronoi regions, go to the file: "*Voronoi.cpp*" and on line no. 80, change the variable: *showFullDiagram* from false to true.

To toggle full screen mode, go to "applicationClass.h" and at line no. 11, change the variable *FULL_SCREEN*.

## *NOTE:

a) In the free cam mode, if you press **V** before you press **R** (Generate a dungeon before resetting the terrain), A dungeon will be drawn on top of the previous dungeon. This way the rooms will be more spatial and it gives the dungeon an entirely new feel. (everything should, in theory, still be functional)
b) The entire scene takes about 16 seconds to load in debug mode and appx 2-3 seconds in release mode. (release and debug refer to visual studio's configuration settings)
c) Release mode might sometimes have issues with the spawning of the player initially and movement in the scene, but the same issues do not persist in debug mode.

# INTRODUCTION

## PROPOSED PROJECT

The project that I chose to work on for procedural methods was *Dungeon Generation Based on Voronoi Diagrams*. The final product would have a player in top down view moving around in the dungeon collecting objects on the way. The shape of the rooms would be the largest square/rectangle based on a generated voronoi diagram and if possible, be the same shape as that of a voronoi region. The terrain of the dungeon would not be a flat plane. The project would also have a post-processing effect which would enhance the looks of the project.

## FINAL PROJECT

The final project consists of a scene which loads into a dungeon generated with different rooms and connecting corridors. The player is in control of a cube model which serves as a player character for the project. The scene is in a top-down perspective and the player is always visible in the center of the scene. The player emits a light of his own which makes only the environment around the player visible.

The player collides with the sides of the dungeon walls and moves on the surface of the terrain. All the collectables in the scene are also lit up and the player can collect them while moving around.

The terrain is altered just before the generation of the dungeon. The terrain runs multiple algorithms to finally achieve a structure which does not resemble a plane.

The terrain shader of the dungeon is also modified to have 3 different textures and a warm look to it. The rocks and the slopes are a different texture than that of the floor. There are patches of cracks and dried blood in various regions of the dungeon.

Along with that, the project also has implemented a post process effect of a radial blur and bloom on specific objects.

# METHOD

## TERRAIN:

After the generation of the terrain, the height map of the terrain has been altered and passed through certain algorithms to give it a less plane-like look.

1) ***Perlin Noise:***
   At first the height map's 'x' and 'z' co-ordinates are passed through a Perlin Noise generator which outputs a single float value. This value is added to the current height map's 'y' value. This gives the terrain a look similar to that of a wavy/hilly region.
   The code to generate Perlin Noise was provided in the tutorial 2 and converted to a C++ class. [Gustavson 2005] *Perlin Noise can also be added to the terrain by entering the Free Cam mode and Pressing '**X**'.*

2) ***Faulting:***
   The faulting algorithm that I run uses a line equation to increase/decrease the height of an entire area on the left/right of a given line. This changes the look of the terrain if run multiple times. I run this algorithm 45 times when the scene is loaded.
   *Faulting can also be applied to the terrain by entering the **Free Cam** mode and Pressing '**F**'.*

3) ***Adding Random noise:***
   This function is a basic function which adds/subtracts a randomized value from each location of the height map.
   *Random Noise can also be added to the terrain by entering the* ***Free Cam*** *mode and Pressing '****Space****'.*

4) ***Smoothen Terrain:***
   For each index(*i*) in the height map, the algorithm adds up all the values in the indices around it (all 8 directions) and aggregates their value. This aggregated value is then applied to the value at the index of *i*. This algorithm is applied 20 times after all the algorithms mentioned above have been run.
   *Smoothen Terrain can also be applied to the terrain by entering the* ***Free Cam*** *mode and Pressing* ***'S'****.*

## DUNGEON GENERATION

The dungeon generation consists of 2 major parts.

1) Voronoi Region Generation and selection of rooms.
2) Delaunay Triangulation and Corridor formation.

### Voronoi Regions:

The process of generating the room is summed up in a point format below:

1) Creating a grid of uniform points in the terrain.
2) Adding a randomized value to the points.
3) Generating Voronoi regions using the set of points generated above as *Voronoi Seeds*.
4) Storing each region in a data structure.

(The code for this process can be found in *Voronoi.cpp*)

The grid of points is created at every uniform location on the height map provided. These points are then added with a *randomized value(v)*. If the distance between two points in the grid is 'x' the randomized value (v) will be between -x/2 and x/2. This way the uniformity of the grid will remain intact while retaining the procedural nature of the voronoi regions that are to be generated. This will also make sure that each room will not have a marginally different size (Which would have been the case if the points had been completely random, where one room could have been the size of half the map while another room would be much smaller than that.)

With the given points, a voronoi diagram is generated. There are a few ways to do this, and the method that I used was as follows:

a) Obtain a set of points **V** from the graph generated.
b) For each point on the height map **H(i),** find the closest voronoi point **V(x)** in the set of points **V**.
c) For each point **V(x)**, create a new array **R(x)** of the type HeightMapType*
d) The array that is stored **R** will contain all the voronoi regions on the map as separate arrays **R(x)**, where n(R) = n(V)
e) Parse through each index of the height map and add it to the right region array **R(x)** depending on the closest voronoi seed **V(x)**.

As voronoi regions are defined as:

*That set of points (called seeds, sites, or generators) is specified beforehand, and for each seed there is a corresponding region consisting of all points closer to that seed than to any other. These regions are called Voronoi cells. [Wikipedia]*

The process mentioned above finds **the closest voronoi seed for each index of the height map** and stores them accordingly into a new array (or in this case a dynamic vector) to store each voronoi region separately.

### Selection of Rooms

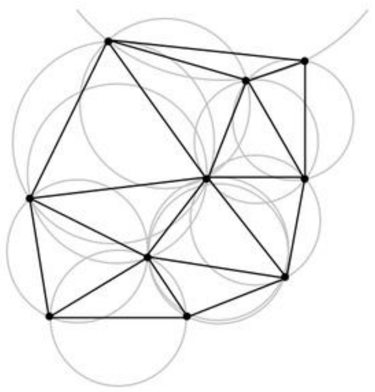
Once the Regions have been obtained, we need to select a small number of these regions to be the rooms of the dungeon we wish to create. In the selection of the dungeons, we need to keep 2 things in mind:

a) Not to keep a room at any of the borders of the map.
b) No two rooms can be attached to each other.

At this level, the grid of points generated in the beginning of the process is extremely helpful. Here we select N number of rooms randomly. If any room is negative with either of the two clauses mentioned above (a, b), another random room is selected in its place.

This process could eventually lead to an infinite loop process (if the number of rooms required > number of rooms available) so we need to makes sure that the total N must be smaller than {n(V)/10} which is the total number of available rooms (Depending on the two clauses mentioned above.)

### Delaunay Triangulation



To make sure that each room is only connected to the room closest to it, we run a Delaunay Triangulation algorithm. This triangulation ensures the connectivity of the rooms closest to each other. In this project, I have used an external library which performs the triangulation and returns a graph data structure as a list of edges. [Blackbone 2016] I have modified this return type a little so that each edge also stores the distance between the two nodes(rooms) and the index for each voronoi seed of a room.

The code for the library can be found in *"/Engine/DelaunayTriangles/"*

### Minimum Spanning Tree:

Once we have an edge list graph obtained from the Delaunay triangulation algorithm, we need to reduce the number of corridors that would be connected to a bare minimal. To do this, we create a minimum spanning tree (according to distances) such that there is no circulation between any two rooms in the dungeon. This way each room is only connected to the rooms closest to it and there is always a path from one room to every other room in the map.

The process of creating a minimal spanning tree is as follows (using Kruskal's Algorithm) [Kruskal 1956]:

a) First we sort every edge given to us depending on their weights.
b) Then we add every edge into the back end of a new edge list
c) Check if the edge currently added causes a circulation in the new graph (edge list).
    a. To do this we need to convert the provided edge list to an Adjacency List. [Drozdek, 2013]
    b. From the first node in the graph, we mark it as 'traveled' and check recursively if the nodes adjacent to it lead to a 'traveled' node.
    c. If the clause above is true the graph is circular, otherwise it isn't.
d) If the previous clause is true (graph is circular), we remove the recently added edge from the list.
e) Repeat the process until the size of the new edge list = n(Rooms) -1 or all the edges have been processed through.

After creating the minimum spanning tree, a few of the corridors which create a circularity are added to this tree and then passed into the next step. Each extra corridor (not part of the minimum spanning tree) has a 10% chance of being added to the minimum spanning tree. This step is added to the process to increase the connectivity of the dungeon.

### Generating Corridors

Once we have the edge list of corridors from the previous step, we generate the corridors with the given indices of the rooms (from the edge list). We use the height map and parse through each index between the x axis of the first point of an edge and the x axis of the second point of an edge to create one corridor while adding some width to it. We repeat the process for the y axis of the second point to the y axis of the first point to obtain the second corridor. This way the joint corridor always forms an 'L' shape.

We store each corridor as another data structure of the type HeightMapType* for easy access later.

### Creating the dungeon from the given data structures:

We now have each room and each corridor stored in easily accessible data structures.

As the dungeon was meant to be a top down map where the player will be able to walk around, the best option was to indent each vertex of the rooms and corridors into the terrain. (subtract the height by a certain value). This worked well and there was no need to create a new mesh for the walls of the dungeons.

### Generating Collectables positions and Player Position

With the given rooms and corridors, the **N** number of collectables in the scene need spawn points every time the dungeon is generated. These spawn points are also generated procedurally. The rooms could hold between 0 to N/2 collectables whereas the corridors will hold the rest of the collectables. Each corridor can only hold 1 collectable (at the midpoint location of the corridor). The code also makes sure that each spawn point is always unique.

## COLLISIONS AND WALKING ON THE SURFACE

### Walking on the Surface

Walking on the surface works on the principles explained from the rastertek tutorial: Height based Movement [Rastertek (no date)]. In this tutorial, the camera is used instead of the player, where the camera casts a ray (passes only the x and z axis values) straight down to the terrain. The terrain will then parse through all its triangles and check if the given x and z co-ordinates lie within a triangle of the terrain. If the triangle is found, then the height of at the given co-ordinates is calculated and passed to the camera. This way the camera will always stay at a certain height just above the terrain. (This is implemented in the ***Free Cam*** Mode where if the camera gets closer to the terrain, it will latch onto the height of the terrain.)

As the terrain has been divided into a quad tree, this technique is efficient to be run in real-time.

The player character in my game also follows the same process as above to move on the surface of the terrain.

### Collisions

As the player in my game was only going to collide with the walls of the terrain, instead of using a complex triangle intersection between the mesh of the player and a mesh of the terrain, I just added a new variable to each index of the terrain called "walkable". This value is passed from the terrain to the player with the same function as the one when it looks for the height of the triangle in the terrain. If the 3 vertices of the triangle that the player is in have the value of walkable as 1, the player will be able to walk without any problems. But if the value of any one of the 3 vertices is < 1.0f the player will not be able to walk on it and the player's position will be set to the player's position in the previous frame.

## TEXTURING

The texturing of the terrain is done using a blend of 3 different textures. The shader for this texture is found in the file: "terrain_ps.hlsl" and "terrain_vs.hlsl".

The three textures are:

a) Dungeon Floor ("Engine/data/dirt01.dds")
b) Dungeon walls ("Engine/data/rock.dds")
c) Cracks ("Engine/data/cracks.dds")

2 of these 3 textures have a very easy implementation. The *Dungeon Wall* is applied to the texture if the slope of the current pixel's normal is greater than a certain value otherwise the *Dungeon Floor* texture is applied.

The third texture of the cracks is slightly more complicated than the ones above. At first, the value of *walkable* for each vertex is passed to the vertex shader and then the pixel shader of the terrain. In the ***Free Cam*** Mode, the value of walkable is multiplied to the red value of each pixel to differentiate the collidable terrain from the walkable terrain.

In the dungeon (in game), this values of walkable is used to apply the cracks texture. When the value of walkable is assigned at the generation of the dungeon, the x and z co-ordinates of that particular vertex are passed through Perlin noise and clamped so that there are no negative values. This value obtained is added to 1.0 and then assigned to walkable. (1 is added because if the value is smaller than 1 the player will take that vertex as collidable). This value when accessed by the pixel shader lerps the textures between the *Dungeon floor* and *Cracks* textures with the value of (2.0 – *walkable)* as its weight. The red value is also directly proportional to the value of walkable for a pixel.

This way the variable walkable has a dual purpose. In the game mode, if the value of walkable = 0, the pixel will not be rendered. Because of this, the top of the dungeon walls will never be visible.

## POST-PROCESSING:

This project involves 2 different post-processing effects:

### Bloom:

The bloom that has been applied is only for the game-objects and not the entire scene. [Learn OpenGL 2015] The process for the bloom shader is explained below:

a) The entire scene is rendered to a texture.
b) Render only the game-objects to another texture and brighten this color.
c) Run a horizontal blur and then a vertical blur and after down-sampling the texture produced in (b)
d) Sample the texture from the blur (c) back to the original size onto a new texture.
e) Blend the texture rendered in (a) and the texture obtained in (d) and render this onto a new texture to process later.

To add to this effect, instead of rendering the collectables with the light shader, they have been rendered with a color shader where the entire cube is rendered as white.

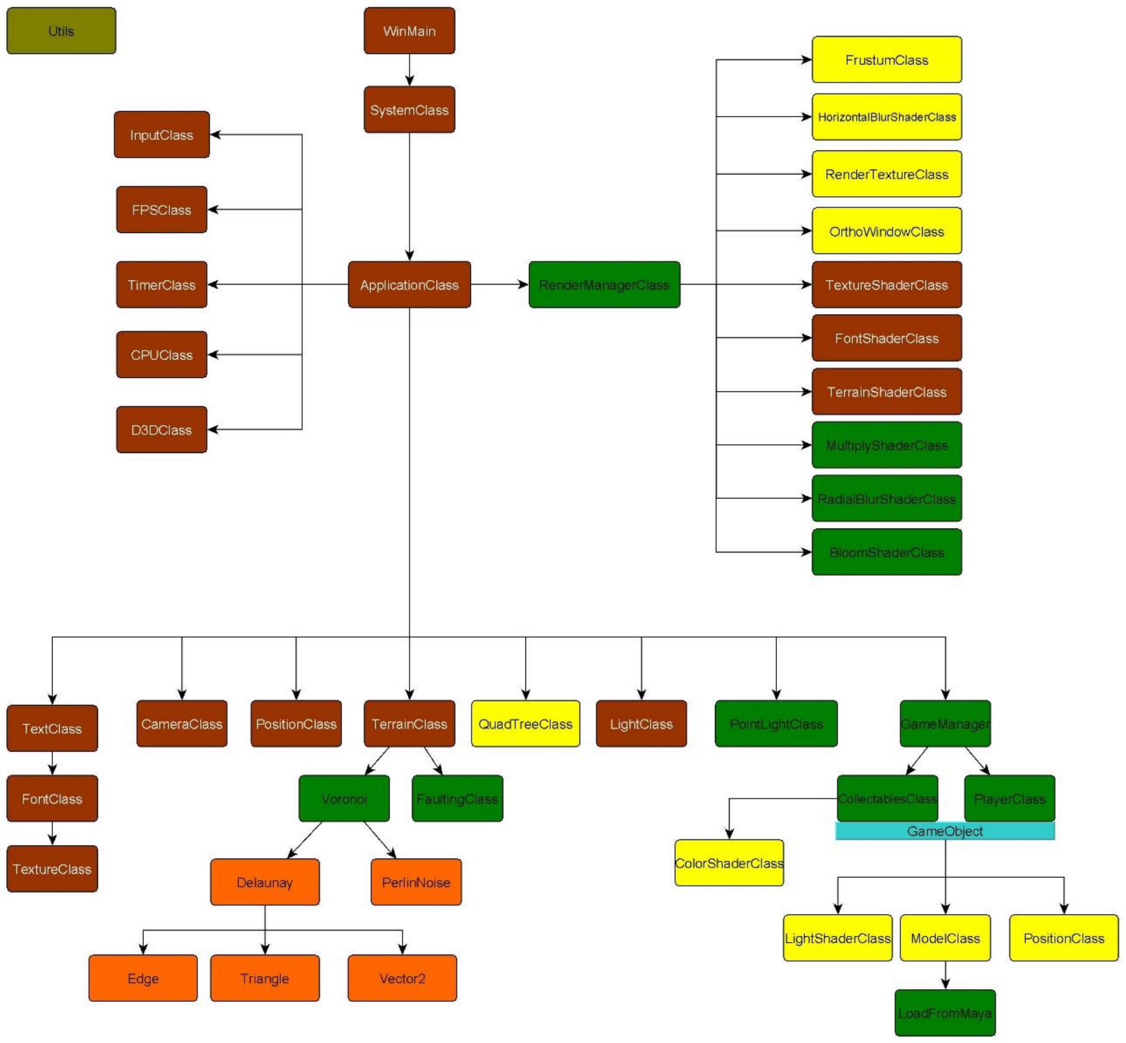
### Radial Blur:

After completing the process above, we have a single texture which renders the entire scene with the bloom effect enabled on the player and the collectables. This texture is then passed through the *radialBlurShader* and rendered to the scene.

This effect is a single shader which calculates the *aggregate* between 10 pixels in its diagonal/radial direction and depending on the distance from the center, applies a lerp between the normal pixel's color and the color of the calculated *aggregate*. This texture is then rendered to the screen using an orthographic rectangle.

## CODE ORGANIZATION

### CLASS ORGANIZATION:



Legend:

| | |
|---|---|
| 🟫 | Classes given by module |
| 🟧 | Classes from external Source |
| 🟨 | Classes taken from rastertek and modified |
| 🟩 | Classes Created / Refactored |
| ■ | Purely Static Classes |

The framework is similar to that which was provided in the initial phase of the module. (Tutorial 1). The new classes have been added and organized to suit the framework provided. As I was learning about using direct x in C++ for the first time, most of the classes here (The ones in yellow) have been understood, modified (where required) and implemented from the tutorial website: Rastertek.com.

The **RenderManagerClass** encapsulates all the shaders within it and handles all the rendering functions that are required in the game. As the rendering process requires several steps (Render to multiple textures before rendering to scene), the refactoring of this class was essentially required.

**MultiplyShaderClass**, **RadialBlurShaderClass** and **BloomShaderClass** are classes which are used to initialize their own respective shaders. A new class was made for each as each shader requires different parameters. For example, the Multiply Shader requires 2 textures while the Radial Blur Shader requires only one. In each of them, only the pixel (fragment) shader is changed, while their vertex shader is common.

The **GameManager** class initializes all the game-related models (Except for the terrain which is initialized in the applications class) which includes spawning of game objects (Collectables and the player), passing of variables, enabling and disabling objects and keeping track of the collectables remaining.

The class **GameObject** has two derived classes namely: **PlayerClass** and **CollectablesClass**. These are the only two types of game objects available in the scene and deriving them from the *GameObjects* class makes them easier to work with as they both have many similar functionalities. Each class encapsulates the *ModelClass* and the *LightShaderClass* such that every object can have its own model and its own specifications as required. Each class also holds a pointer to an attached light which gives them access to manipulate the light object very easily.

The class **Voronoi** is solely present for the generation of the dungeon. The class uses the data structures provided from the **Delaunay** class and follows a step by step process to generate the Rooms and then the corridors of the dungeon. The function *GenerateVoronoi* when called returns the data structures of the rooms and corridors after the generation process is completed.

The class **PerlinNoise** is a class with static only members which need to be initialized before and released after the process is completed. This class follows the same code as that was provided in the tutorial 2 of this module [Gustavson, 2005]. The code has only been converted from java to C++.

Apart from the classes present in rastertek's tutorials, I have added a new class called **PointLightClass** which will hold all the functionalities of a point light in the scene. The game has an array of these point lights which are sent to the terrain shader and the light shader so that the terrain and the models are lit up by these point lights. Each point light moves with a game object and the light attached to it. The player's light has a flickering effect which makes the environment look like it's lit up by a torch.

The class **Utils** was created for ease of use and performing purely mathematical functions without the use of objects.

## DATA STRUCTURES

Below is a list of the data structures used in the generation of the voronoi region dungeon:

The grid of the voronoi seeds (**V**) is a vector of the type: **<VoronoiPoint*>**. A VoronoiPoint is a structure that holds the float co-ordinates of the location of the voronoi point in world space along with the height of the entire voronoi region and 2 indices. The first index is corresponding to the index of the point in the height map while the second one relates to the index in the grid of points generated (**V**).

The struct **HeightMapType** provided in the source of the module has been modified. This structure now holds a pointer to **VoronoiData*** (which holds the VoronoiPoint and the distance from this point for each index on the height map) and a float value for walkable.

The final struct that was used in the generation of voronoi dungeons was called the **VoronoiRegion** struct. This struct holds a pointer to the *VoronoiPoint* that resides in it along with a list of all the indices that are a part of this region. This struct makes the process of parsing through all the indices within this region in real time very quick as each index of a region can now be accessed in constant time.

The class Voronoi has a function: *GenerateVoronoiDungeon* which returns 2 major data structures (arrays as Vectors) to the TerrainClass.

a) A vector of type: <VoronoiRegions*> called m_rooms which stores all the rooms and a pointer to their vertices.
b) A 2D vector of the type <HeightMapType*> called m_corridors to store all the corridors and a pointer to their vertices.

### EXTRA FILES

The Model class that I have used is "cube.txt". As model loading was not the main aim of the project, cubes are the only models that I have worked with in this game.

There are many different vertex and pixel shaders that were used in this game. The terrain had a different shader than that of the models as the terrain required the "walkable" in its shader. Along with that another shader is used for the radial blur on the texture where the scene is rendered. These shaders are loaded and compiled from their own respective shader classes. (e.g. The *TerrainShaderClass* will load and compile "*terrain_vs.hlsl*" and "*terrain_ps.hlsl*")

The font shader and the texture shaders are simple shaders which are used to render the text to the scene and scene to a texture respectively.

Many of the vertex shader have been re-used. For example: the radial blur vertex shader and the texture shader's vertex shader share the same code. For this reason, another new vertex shader has not been created. Instead, the same vertex shader has been loaded for both the shader classes.

## CRITICAL APPRAISAL:

### VORONOI DUNGEON:

The code for the dungeon is encapsulated inside the Voronoi class which is called from the TerrainClass. The functionality is easy to operate with and the data structures of the rooms and corridors obtained are used for generation of the spawn points. These spawn points can be generated with the complexity of O(n).

In terms of efficiency the dungeon generation code works well for a pre-loading session, but the total amount of time taken to generate the dungeon is high and cannot be used for real time generation.

The major problem that lies in this code is the generation of Voronoi regions. The amount of time taken to process the entire height map and obtain the voronoi points is O($n^2$) which is quite a lot to process if the resolution of the height map is high. (In this game it is 256x256)

The solution to this will be to change the algorithm used. The algorithm that is being used currently checks the seed at the minimum distance from the current region. One way to improve the efficiency would be to use a quad based system (same as that of the terrain) and only check the distances with the point in the current quad. But this method will also create problems if the voronoi point of the current index is in another quad altogether.

The next best option would be to use the midpoint bisection formula which calculates a voronoi diagram by parsing through each voronoi seed and draws a perpendicular bisector with the points around it calculated through the Delaunay triangulation algorithm. The voronoi diagram is a connection of all these bisectors. While this process is less expensive than the one used, it still takes O($n^2$) amount of time which is exactly what we are trying to prevent.

Another way to improve the efficiency of this algorithm is to use the Sweep Line algorithm for this process. This algorithm passes a line from the top of the array to the bottom. Every time the line touches a voronoi seed, it generates a new parabola. If one parabola touches another parabola, it starts drawing an edge for a voronoi region. This algorithm was founded by Steven Fortune in 1986. [Fortune 1986]. This algorithm takes a steady O(n*log(n)) amount of time to calculate the voronoi regions and can also be used for real time processing. [Kuckir (no date)]

The only problem that lies here is that this algorithm will only give me the edges of the voronoi diagram. If I want to access each pixel in the diagram, it will take me another O(n) amount of time to do so which takes me back to my current algorithm.

Certainly, the amount of time taken to generate the voronoi diagram can be slightly faster, but as I do not need my generation to be real time, the algorithm currently in place works well enough for this game.

## GAMEPLAY AND COLLISIONS

The use of the current framework enabled me to use the value *walkable* as a suitable substitute to a collision using a triangle intersection check which would've been a lot more expensive on a real-time basis.

The problem that arose with this collision was that most of the times the player hit a wall, the player would vibrate because of the high current velocity. Initially the camera's position was directly attached to the player's position, but when the player hit a wall, because of the vibration, the entire screen would shake. To overcome this problem, I used a lerp function to smoothly transform the camera's position to the player's position.

Another way of improving the collisions of the player would be to implement wall sliding. This will make the movement a lot smoother than it already is.

The framework for the game objects and the game manager made it extremely easy to work with the gameplay programming of the game. With this framework, it was easy to make the camera smoothly follow the player (using a lerp function) and the player rotate smoothly from one direction to another. It was also easy to work with lights and reduce their intensity slowly rather than turn them off immediately once the collectable was collected.

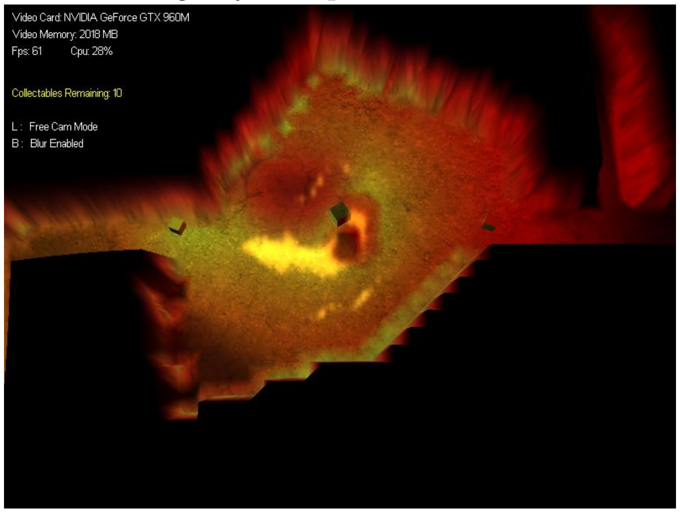
## Models

This current game only has cubes loaded into it but it also has the capacity to load other smaller models. The problem occurs when we try to implement larger more complicated models. I spent a lot of time initially trying to load more complicated models, but eventually stopped because this project's aim was to explore procedural content rather than load complicated models. Later in the process, I stumbled upon this library called *Assimp* which loads models into the scene by providing a list of vertices. If later I was to load complex assets, I would use this free library instead of creating one of my own.

## Post Processing:

Post processing (Render to texture and then render that to screen) had a few permutations and combinations. Initially I had implemented a Gaussian blur which involved down sampling a rendered texture, applying a horizontal and then a vertical blur and up sampling the texture, but this result of this process did not fit very well with the visuals of the game.

After that, I tried applying a bloom to the scene which involved selecting a few pixels from the screen whose brightness is above a certain threshold, and then blurring only those pixels. This blurred texture is then added to the main texture which fakes the intensity of light to be greater than it already is. This works well in environments where the light is not attached to the player, but in this game, the added intensity did not fit the scene very well. Additionally, I have reduced all the blue light in the game so the chances of a pixel reaching a high brightness value are low. The bloom (with a lower threshold) will only affect the areas around a point light aggressively. Therefore, I changed the approach and instead of applying a bloom to the entire scene, I only applied it to the game-objects in the scene. (Process is explained in the Methods section above)



*Visuals for full screen bloom in this scene*

With the bloom effect only on the game objects, one issue that I faced was that the bloomed texture was passing through the walls of the dungeon. To overcome this problem, keeping in mind that the collectables are always lit up and the top of the dungeon walls are always black, I have kept a simple check to add the bloomed effect to the rendered texture only if the current color on the rendered texture is not black. This is clearly a work around and the ideal solution would be to create a mask for which areas need to be bloomed and which not. But in the context of this game, this solution (despite not being ideal) works well.

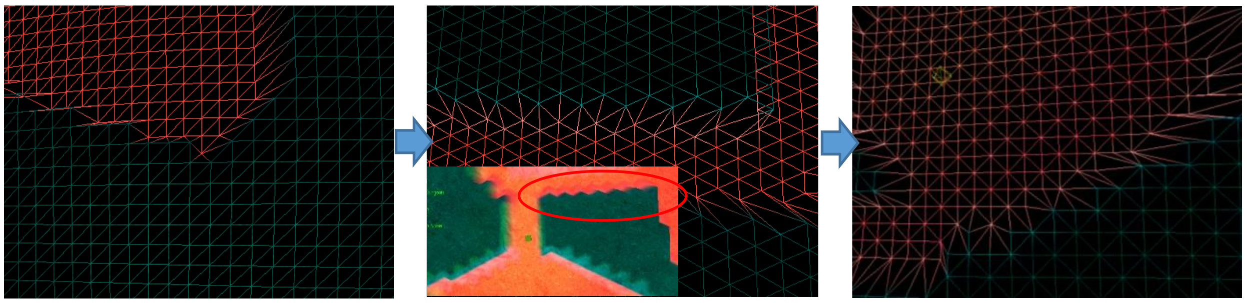
The radial blur also had an issue that I faced. When checking the pixels outside the range of the current texture, I had an option to either clamp them or leave the pixels accessed to black. Neither of these options looked good so I changed the blurring algorithm a little. Instead of aggregating with 5 pixels in front and 5 pixels behind, I only aggregated the color with 10 pixels in front of the current pixel (in the same direction towards the center). This solved the issue entirely and the blur looked like a part of the environment.

## The corners

The entire system of voronoi dungeon generation worked well throughout the process. The game, according to me, looks like a player with a torch in a dungeon.

There is one issue that I faced throughout the project. The corners of the voronoi dungeon's rooms are not of a square or rectangular shape. Because of this when the rooms of the dungeons are indented, the corners of some of the voronoi rooms are very edgy. This doesn't look extremely good and I tried a few ways to solve them.

First I tried to change the triangulation of the terrain. Instead of keeping the triangles in a square-like formation, I changed them to a set of equilateral triangles which improved the edges of the voronoi rooms, but because the corridors are straight lines, their walls became a set of edgy lines.



Another way that worked slightly better was to increase the resolution of the height map. Here it improved the visuals, but not to a massive extent. The edginess of the slopes was still visible.

The best way to solve this problem would be to create a new mesh for the walls of the dungeon by taking each corner point as a vertex.

## The corridors

a) Even though these corridors work well enough for the dungeon generated, some of the corridors pass through other rooms which looks alright in the game mode but doesn't look extremely good in the Free Cam mode. The reason why I implemented the minimum spanning tree was so that this room intersection is minimalized, but the problem persisted. One way to solve this problem completely would be to create a pathfinder from one room to another and make sure that the path doesn't intersect with another room or corridor.

b) Because of the use of the minimum spanning tree, there would've been no circular path in the dungeon. This would've made the journey from one room to another very tedious as the player needs to traverse through many rooms in the middle to get to the distant room. To reduce this effect, a few of the corridors that are not a part of the minimum spanning tree (the ones which create circularity) are added to the structure of the minimum spanning tree

c) Even though the corridors intersecting rooms [mentioned in (a) above] don't look very good, at present, it's adding to the circularity of the dungeon which makes the dungeon a little more fun to explore.

## REFLECTION

This semester was a complete learning experience for me. I started off without any experience in C++, Direct x and graphics programming but ended up learning a lot about each of these aspects in great depth.

I started off by learning more about how direct x works and learnt in depth about the graphics pipeline and how shaders are loaded and compiled. The tutorial websites: Rastertek [Rastertek (No Date)] and DirectX Tutorials [DirectX (no date)] were the most helpful at this stage. Later I progressed into loading models into the scene and adding directional and ambient lights while implementing a specular power to them. I then learnt about how the terrain system works in the framework provided.

The entire learning process is what took away most of my time in the entire semester. Soon after I started to get a grasp of how direct x works, I started researching on how I could implement the Voronoi based dungeon that I planned to implement for this coursework.

The process of research helped me understand different algorithms and how each can be used to generate better results. For instance, if I had not implemented the grid system to generate the voronoi seeds, I would have used the K-means algorithm for clustering to reduce the spread of the voronoi seeds throughout the map.

This entire process also taught me about how to work with shaders. Towards the end, I wrote the entire shader for the terrain, models and the post processing by myself which was a great leap in terms of learning.

I've always been used to working with game engines and having everything given to me in a platter. In this project, I was forced to work without any inbuilt structures for game objects and scenes. The entire process of writing my own game object system helped me understand how game engines work on a deeper level. I also conducted some research on how I can implement collisions between two meshes which gave me a deeper insight into how a physics engine would work. The entire lighting system in this game has been implemented by me which also gave me a better understanding of how lights work in a 3D environment.

I have still only touched the surface and there is a lot more to learn to reach the level of another game engine, but this entire module helped me get an entirely new perspective on how I can build my own game engine.

Dungeon generation was a very important part of this project and creation of this dungeon was a long process. The gamasutra article on Dungeon Generation was the most helpful to get a better understanding of how the dungeon generation algorithm works. [Adonaac 2015]

In terms of procedural methods, I have learnt and implemented some different methods that had been taught to us in the classes. I understand better how the terrain can be modified without having an artist to work with it. I've also understood methods like particle deposition and how that can be used in various tools throughout different parts of the pipeline. The deep study of the algorithms to generate the voronoi dungeons made me see more ways of applying them. For example, I can use the Delaunay triangulation algorithm to create meshes out of certain given points, or voronoi regions to break a mesh apart as a destruction algorithm. This has opened many new windows and ways of perceiving different problems.

## REFERENCES

Adonaac A. (2015), *Procedural Dungeon Generation Algorithm,* Available from: http://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php [Accessed on: 25th April 2017]

Blackbone (2016), Available from: https://github.com/Bl4ckb0ne/delaunay-triangulation [Accessed on 25th April 2017] <External Library>

DirectX Tutorials, Available from: http://www.directxtutorial.com/LessonList.aspx?listid=11 [Accessed on 25th April 2017]

Drozdek A. (ed. 2013) *Graph Representation* In: *Data Structures and Algorithms in C++* pp. 393-397

Fortune S., (1986), "A sweepline algorithm for Voronoi diagrams" SCG '86 Proceedings of the second annual symposium on Computational geometry, pp. 313-322

Gustavson S. (2005). "Simplex Noise Demystified" *Linköping University, Sweden* [Provided in tutorial material from the module] <External Library>

Kruskal, J. B. (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem". Proceedings of the American Mathematical Society. 7: 48–50.

Kuckir I. Available from: http://blog.ivank.net/fortunes-algorithm-and-implementation.html#impl_cpp [Accessed on: 25th April 2017]

Learn OpenGL, 2015, *Bloom* Available from: https://learnopengl.com/#!Advanced-Lighting/Bloom [Accessed on: 25th April 2017]

Rastertek, Available from: http://www.rastertek.com/ [Accessed on: 25th April 2017] <External Library>