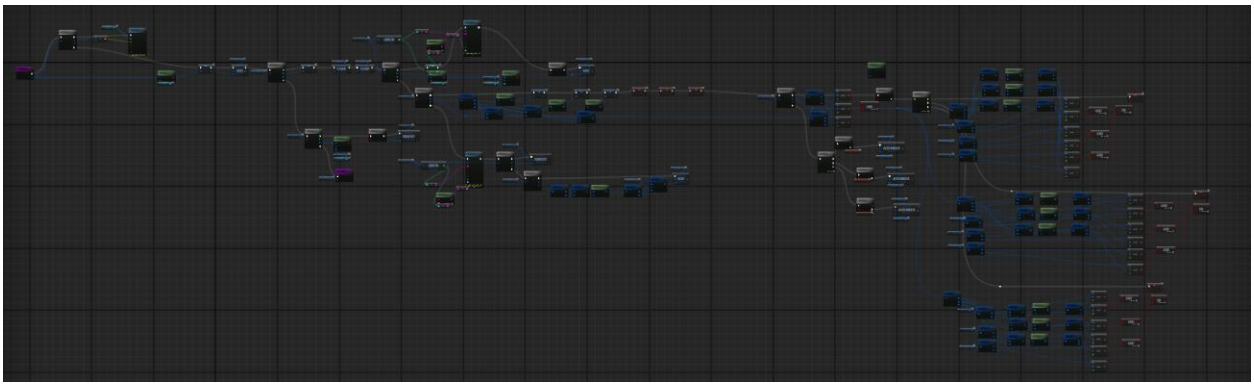
4/3/2025

Translating the C++:

The steps for generating a dungeon layout are:

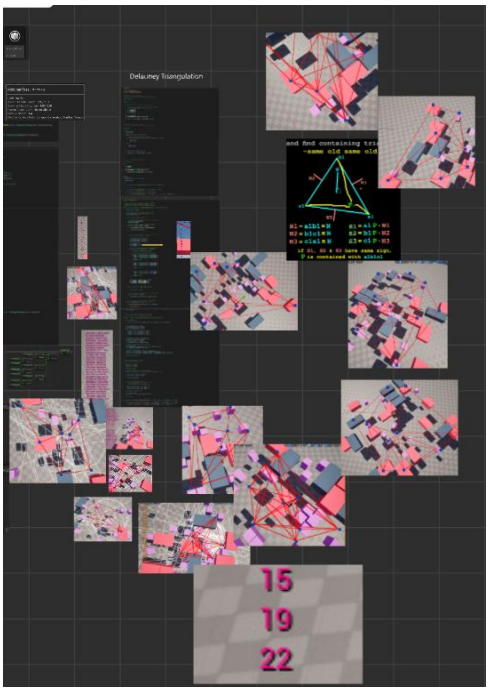
1. Spawn in a bunch of boxes (rooms) with random locations and rotations within bounds.
2. These boxes have physics simulations running to separate themselves out from other boxes. Run checks to see when all boxes stop simulating physics.
3. Once all of the boxes have settled, get the central points of a random selection of them (usually the largest boxes) (usually about a 4^{th} of the total boxes spawned).
4. Use these points to calculate Delauney Triangulation and make a graph connecting each point without any edge intersecting another edge.
5. Use this graph to calculate a Minimum Spanning Tree- a path that connects every room with no loops.
6. (Optionally) add some loops from the graph back into the tree for fun
7. Calculate corridors connecting close together rooms
8. Add back in rooms that the corridors intersect
9. Spawn a minimum corridor width to make sure all rooms connect with enough space to walk through.

I comfortably made it to step 4: Delauney Triangulation. This is the most intimidating step, so I'm not completely worried about it being a hurdle. This week was also particularly busy, and after Friday, (GDN Competition deadline), I will be able to fully focus on this project. I would like to be further through this process than I am, but I do think progress will speed up significantly after this week. I plan to attend office hours this coming Friday to talk about where I think the problem may lie.



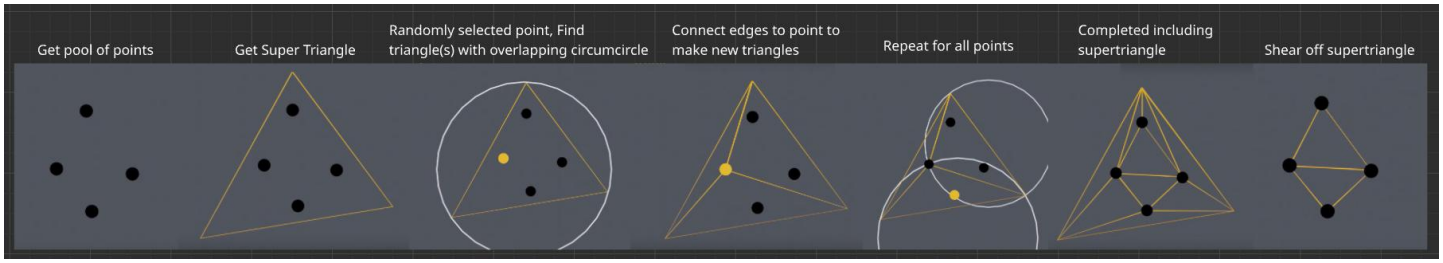
Delauney Triangulation:

The most common way to calculate Delauney Triangulation is the **Bowyer-Watson** algorithm. Its steps are as follows:

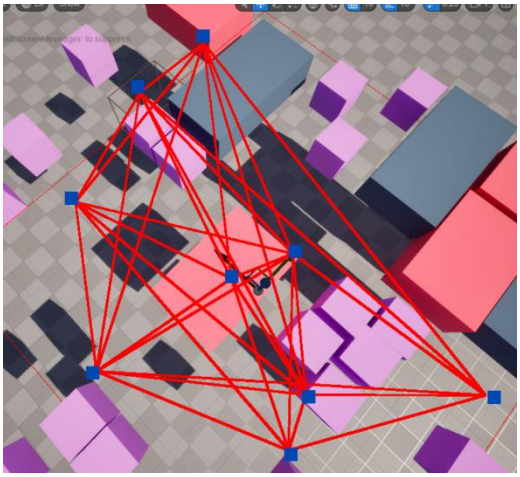


1. Generate a massive "Super Triangle" that contains all the points we would like to connect.
2. Select a random point from our pool
3. Look for triangles whose circumcircle contains the selected point. (This means looping through each generated triangle and calculating their circumcircle, then doing a distance check from their circumcenter and our point and comparing it to the circle's radius)
4. When the algorithm is complete, no triangles will contain any of the points (they will instead connect each point), so any triangle whose circumcircle does overlap our point is BAD and needs to go. Add each overlapping triangle to an array of "Bad Triangles".
5. Loop through your array of "Bad Triangles" and find its edges. Connect each point of the bad triangle back to the point we are checking (this way our point falls in the center of three triangles instead of the middle of one). (During this step, check each edge to make sure it doesn't duplicate the edge of a pre-existing triangle)
6. Remove the bad triangle from our triangles array, add the three new triangles to our triangles array, and repeat steps 2-6 until each point has been checked.
7. Once each point has been checked, remove all vertices left over from the super triangle and every triangle that contains those vertices

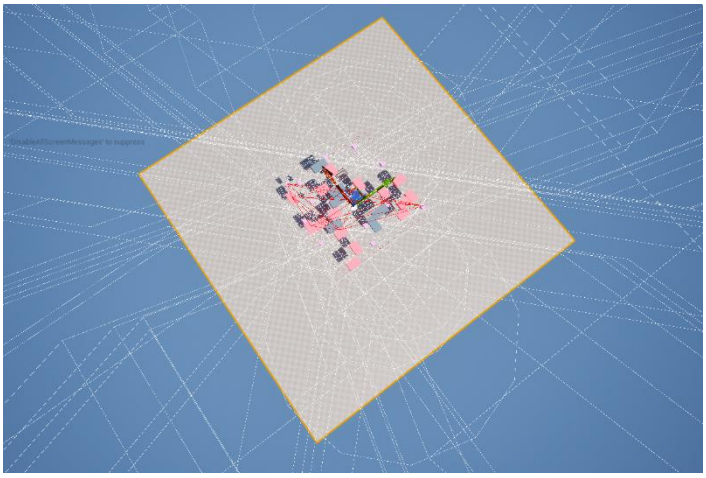
I have collected frames from this strangely cropped youtube video as a visual:
https://www.youtube.com/watch?v=GctAunEuHt4&t=77s

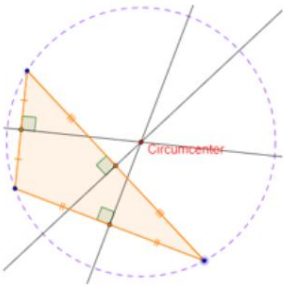


The problem is that my points are generating like this, which is not ideal.



I investigated a few suspects. First, I made debug circles to show my Circumcenter calculations (white lines), and those looked weird enough that I thought something had gone terribly wrong with them. After (a lot) of further research into how circumcenters are calculated, I realized that they are just super weird. I realized that the circumcenter could lay outside of the triangle itself. I wish I had Ritalin when I took math in highschool. The example code I looked at seemed to use a specific cartesian coordinates calculation that I could only find on Wikipedia and no other Youtube explanation. Eventually I checked it against another Unreal Engine Delauney Triangulation project found online that did the same thing in blueprints. When they lined up, I deemed the circumcenter calculation innocent.





The circumcenter of an obtuse triangle is outside the triangle opposite the obtuse angle.

Reference C++ Code, Wikipedia, Reference triangulation project, my function:

```cpp
/**
* Calculates circumcircle of a triangle
* Used to check Delaunay condition
*/
{
    FVector2D A = Triangle.A;
    FVector2D B = Triangle.B;
    FVector2D C = Triangle.C;

    SCircumcircle Circle;

    // Calculate determinant for circumcenter calculation
    float D = 2 * (A.X * (B.Y - C.Y) + B.X * (C.Y - A.Y) + C.X * (A.Y - B.Y));

    // Handle degenerate case (collinear points)
    if (FMath::Abs(D) < KINDA_SMALL_NUMBER)
    {
        return SCircumcircle();
    }

    // Calculate circumcenter coordinates
    float Ux = ((A.X * A.X + A.Y * A.Y) * (B.Y - C.Y) +
        (B.X * B.X + B.Y * B.Y) * (C.Y - A.Y) +
        (C.X * C.X + C.Y * C.Y) * (A.Y - B.Y)) / D;

    float Uy = ((A.X * A.X + A.Y * A.Y) * (C.X - B.X) +
        (B.X * B.X + B.Y * B.Y) * (A.X - C.X) +
        (C.X * C.X + C.Y * C.Y) * (B.X - A.X)) / D;

    Circle.Center = FVector2D(Ux, Uy);

    // Calculate the radius
    Circle.Radius = FVector2D::Distance(Circle.Center, A);

    return Circle;
}
```

Cartesian coordinates [edit]

The Cartesian coordinates of the circumcenter $U = (U_x, U_y)$ are

$$U_x = \frac{1}{D}\left[(A_x^2 + A_y^2)(B_y - C_y) + (B_x^2 + B_y^2)(C_y - A_y) + (C_x^2 + C_y^2)(A_y - B_y)\right]$$

$$U_y = \frac{1}{D}\left[(A_x^2 + A_y^2)(C_x - B_x) + (B_x^2 + B_y^2)(A_x - C_x) + (C_x^2 + C_y^2)(B_x - A_x)\right]$$

with

$$D = 2\left[A_x(B_y - C_y) + B_x(C_y - A_y) + C_x(A_y - B_y)\right].$$

Without loss of generality this can be expressed in a simplified form after translation of the vertex A to the origin of the Cartesian coordinate systems, i.e., when $A' = A - A = (A'_x, A'_y) = (0, 0)$. In this case, the coordinates of the vertices $B' = B - A$ and $C' = C - A$ represent the vectors from vertex A' to these vertices. Observe that this trivial translation is possible for all triangles and the circumcenter $U' = (U'_x, U'_y)$ of the triangle $\triangle A'B'C'$ follow as

$$U'_x = \frac{1}{D'}\left[C'_y(B'^2_x + B'^2_y) - B'_y(C'^2_x + C'^2_y)\right],$$

$$U'_y = \frac{1}{D'}\left[B'_x(C'^2_x + C'^2_y) - C'_x(B'^2_x + B'^2_y)\right]$$

with

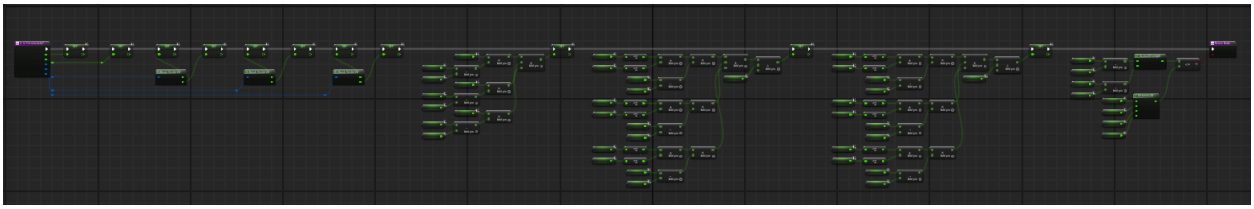
$$D' = 2(B'_x C'_y - B'_y C'_x).$$

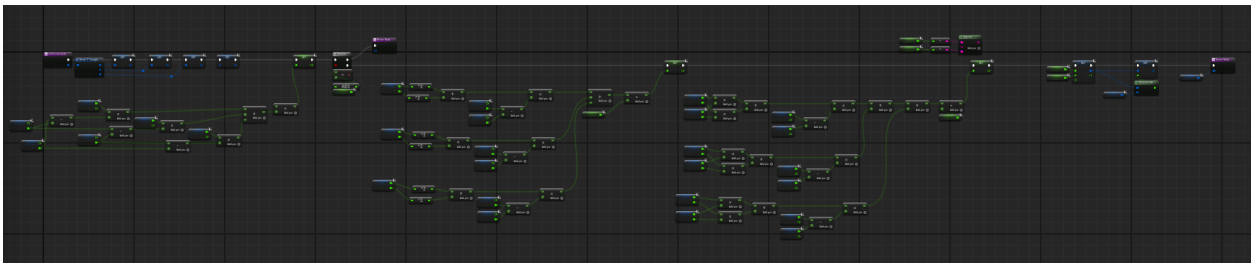
Due to the translation of vertex A to the origin, the circumradius r can be computed as

$$r = \|U'\| = \sqrt{U'^2_x + U'^2_y}$$

and the actual circumcenter of $\triangle ABC$ follows as

$$U = U' + A$$



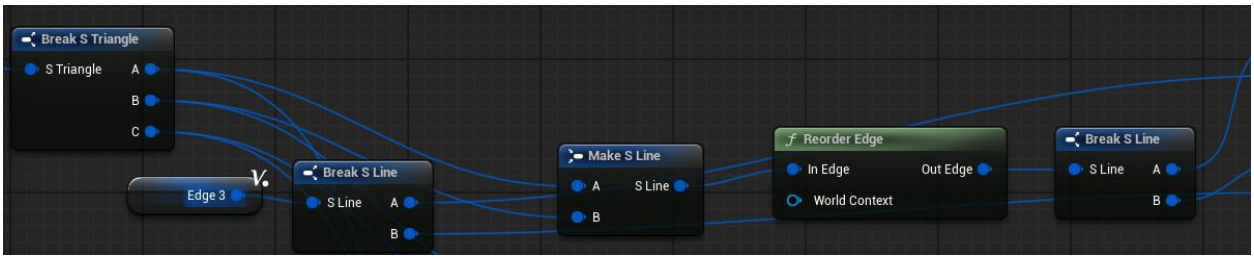


I did find an issue with my “Edge” structs. The reference code uses a struct “Edge”. I assumed that “Edge” was synonymous with “line” and just reused my “line” struct. But it seems that in C++, one is able to make structs that dynamically adjust themselves with built in functions. I went back through my own lines and attempted to replicate this by plugging them into an equivalent function when they are created.

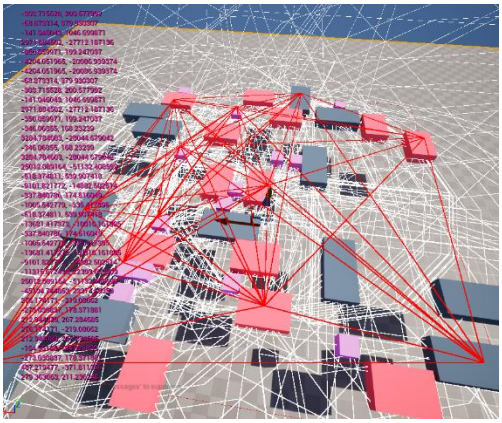
```cpp
struct SEdge
{
public:
    FVector2D Start;
    FVector2D End;

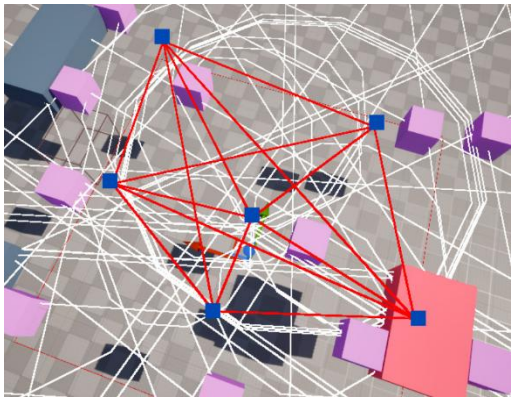
    SEdge(FVector2D InStart, FVector2D InEnd)
    {
        // Ensure consistent ordering by sorting points
        if (InStart.X < InEnd.X || (InStart.X == InEnd.X && InStart.Y < InEnd.Y))
        {
            Start = InStart;
            End = InEnd;
        }
        else
        {
            Start = InEnd;
            End = InStart;
        }
    }

    bool operator==(const SEdge& Other) const
    {
        return Start == Other.Start && End == Other.End;
    }
};
```



I do think this "Reorder Edge" function helped somewhat. My graphs went from looking like the first image (most of the time) to the second image (most of the time).





Occasionally, my grid will even generate perfectly. This is deeply annoying. It is taunting me.

If I do not get this to work shortly I will pivot and try again to use this for my top-down project. I plan to next investigate my points-generating function, because as I look at these graphs now, I am a little bit confused as to why the points are where they are. They should be in the center of each block (which is still sometimes true).

