**Project Name:**

Rope Builder Tool

**Table of Contents**

Project Overview

Project Scope

Project Tasks Overview

Project Pseudocode and Design

**Project Overview**

This project aims to create a simple Unity 2019 rope mesh generator tool and will be be written in C#. An algorithm for the mesh generation will be generated along with a basic UI-addon to control the script.

The project scope is very simple and is not intended to create a robust tool.

**Project Scope**

Script that creates a mesh rope based on inputs.

Inputs:

* Start and End positions (X,Y,Z)
* At least one additional position to extrapolate a curve.
* A material to apply on the mesh.
* Radius of the bases.
* Quality of base (number of sides)

Outputs

* Rope mesh
  + Vertices in a 3D space (X,Y,Z)
  + Triangle connections

**Project Tasks Overview**

1. Create a face along a circle
2. Connect two faces
3. Add any # of points in a 3D space and connect faces
   1. Add mesh UVs
4. Add angle calculation to smooth out point connections in 3D
5. Create Unity UI for tool

**Project Pseudocode and Design**

1. **Create a face along a circle**

The base of a rope can be thought of as a circle. A circle can be thought of as a shape with infinite sides.

Each side is equally spaced apart by an angle, A°.

For infinite sides, this angle is ~ 0° .

For a triangle, it is 360° / 3 = 120°

We can create a regular polygon by using the parametric equation for a circle.

x = ( x’ + r \* cos(A°)) x’ : Center’s x-coordinate

y = ( y’ + r \* sin(A°)) y’ : Center’s y-coordinate

^ If A is negative it will move clockwise.

A° = 2 π / # of sides

Code

CreateRingVertices(Center, CullingDirection, AngleOffset)

{

Vector3[] ringVertices;

Add center to array

Create temp variables to hold x,y,z coordinates

* Set z as static for now

for (int < Numsides +1)

{

x = (center.x + Radius \* cos(A° \* CullingDir \* int + offset))

y = (center.y + Radius \* sin(A° \* CullingDir \* int + offset))

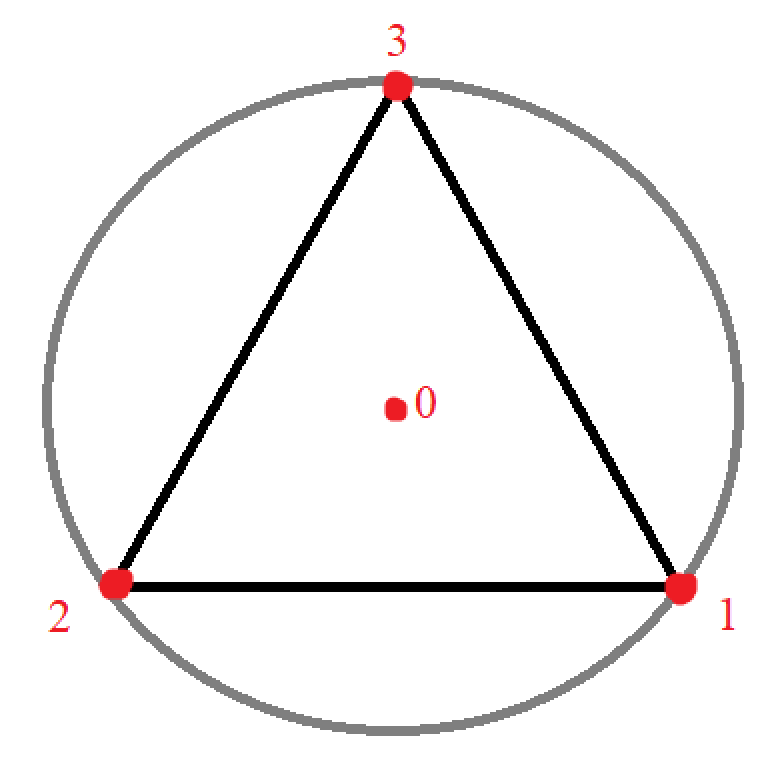
Add vector to array

}

Return

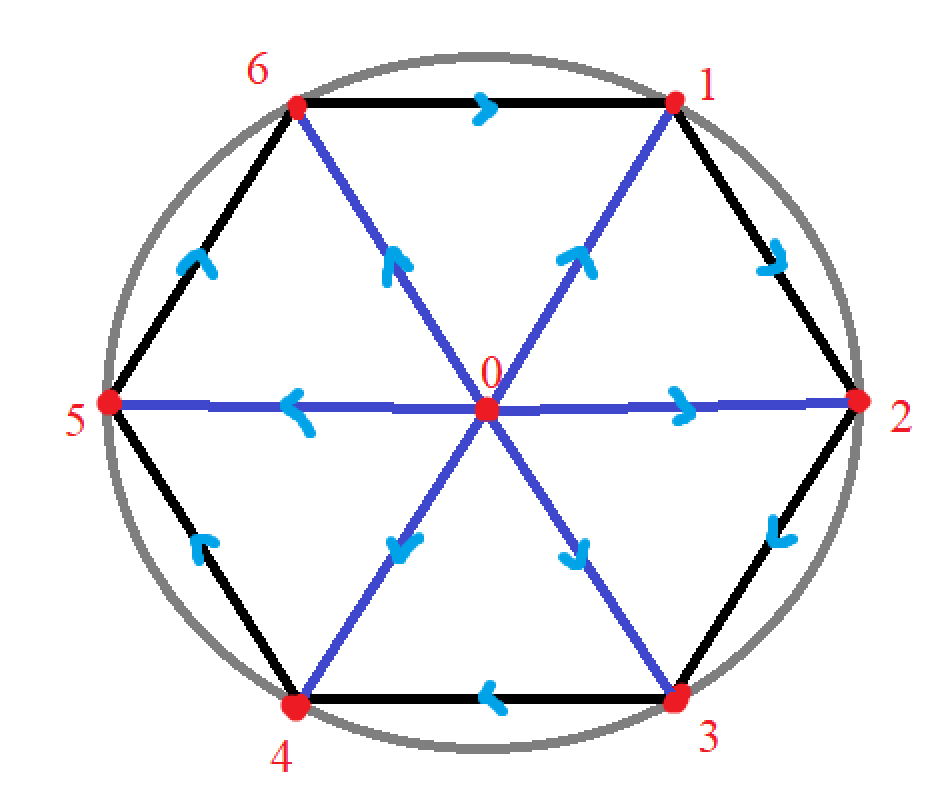
}

Once you have the vertices, we create the triangles that form the mesh. A triangle is formed by three points in a set space. The vertices that make up the face are added in a certain order and it is important to know the order (which is why we keep track of what is called the culling direction). For example, for a triangle the following vertices are added:



The numbers represent the order in which the vertices were added. We use this same numbering scheme to represent triads that create triangles when connected. Thus: 0 1 2 , 0 2 3, 0 3 1, form this triangle. Naturally for a triangle itself, 1, 2, 3 is enough. Why create three triangles for one? Indeed, it is unnecessary but the algorithm is generalized for all shapes. The triangle is the only one that has this problem.

Then, we can use a hexagon and manually write out the vertices ordering. The hexagon is chosen arbitrarily, as a guess of the amount of sides needed to observe a pattern.



[0,1,2]

[0,2,3]

[0,3,4]

[0,4,5]

[0,5,6]

[0,6,1]

It’s easy to see a pattern from this.

The first vertice is always the center (in this case 0). Let’s call this the offset (the reason for this name is understandable as we add a second polygon). Now that we have recognized a generalized variable, we can generalize the triangles.

[offset, (offset + 1) + 0, (offset + 2) + 0]

[offset , (offset + 1) + 1, (offset + 2) + 1]

[offset , (offset + 1) + 2, (offset + 2) + 2]

[offset , (offset + 1) + 3, (offset + 2) + 3]

[offset , (offset + 1) + 4, (offset + 2) + 4]

[offset , (offset + 1) + (Numsides - 1) , (offset + 2) + 5]

^ Used to represent the max reached.

First value is always offset, second value is offset + 1 + increasing integer, third value is offset + 2 + increasing integer. It’s also notable that the amount of times this is repeated is the number of sides the polygon has (in this case 6).

However, these values rollback when they reach a certain maximum. This maximum is the number of sides. This behavior can be thought of as overflowing and we can create a helper class to represent this.

LimitedInt

LimitedInt

{

Create Max and Min properties, inclusive.

The actual value of the integer, private.

A public handle for our limited integer.

{

Get {return actual value}

Set

{

If newValue > oldValue (LimitedInt), we are adding

if new value > Max, we have overflowed

Calculate difference between new value and max - 1

Set LimitedInt to minimum + difference

Else, just set LimitedInt to new value

Else if newValue < oldValue (LimitedInt), we are subtracting

If new value < Min, we have overflowed

Calculate difference between new value and min - 1

Set LimitedInt to Max – difference

Else, just set LimitedInt to new value.

}

Constructor()

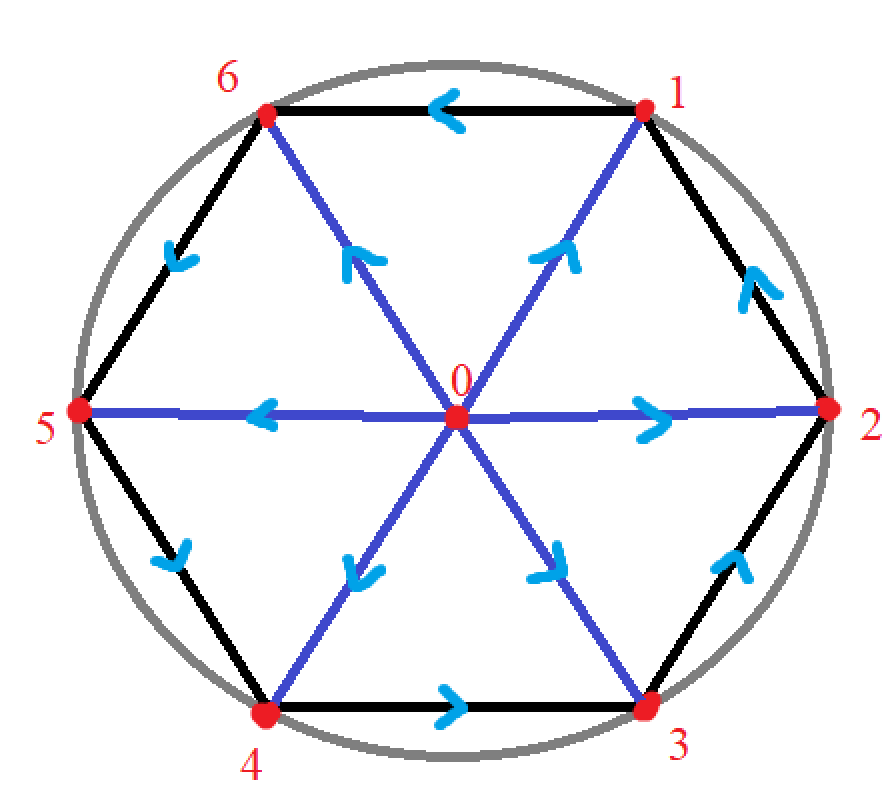
Make sure initial values are within max and min range.

If not, set default to max.

}

}

The direction in which the triangle is created termines from which side it can be seen. Since a cylinder requires to be seen in 3D, we must be able to go backwards. As mentioned, this is the culling direction.



[A,B,C]

[0,1,6]

[0,6,5]

[0,5,4]

[0,4,3]

[0,3,2]

[0,2,1]

Which can be generalized as:

[offset, (offset + 1) – 0, (offset + 6) - 0]

[offset, (offset + 1) – 1, (offset + # of sides) - 1]

[offset, (offset + 1) – 2, (offset + # of sides) - 2]

[offset, (offset + 1) – 3, (offset + # of sides) - 3]

[offset, (offset + 1) – 4, (offset + # of sides) - 4]

[offset, (offset + 1) – (# of sides - 1), (offset + # of sides) - 5]

For both directions, A = offset

For both directions, B: (offset + 1) but moves in culling direction

(offset + 1) + (culling direction \* b)

Where b is incremented NumSides times

C is complicated

It starts at offset + 1 (its minimum value), then we add the culling direction

So, if offset + 1 = 1, then

1 + (Positive Culling) = 2

1 + (Negative Culling) = 6

^for a limited int with Max = Number of Sides, and

Min = offset + 1

Therefore, the method is the following:

CreateRingBases(offset, cullingDirection, NumSides)

{

Max = Numsides + offset

Min = offset + 1

b\_Offset = (offset + 1)

LimitedInt c\_Offset = (offset + 1) + CullingDir

b, c = 0

LimitedInt B, C = Max,Min, b\_offset

c\_Offset)

int [Numsides \* 3] triangles

for (T = 0; T < (NumSides \* 3); ++T)

{

if (T % 3 == 0) // we are on A

triangles[T] = offset

else if (T % 3 == 1) // we are on B

B = b\_offset + (CullingDirection)\*b

Triangles[T] = B

++b

Else //we are on C

C = c\_Offset + (CullingDirection) \* C

Triangles[T] = C

++c

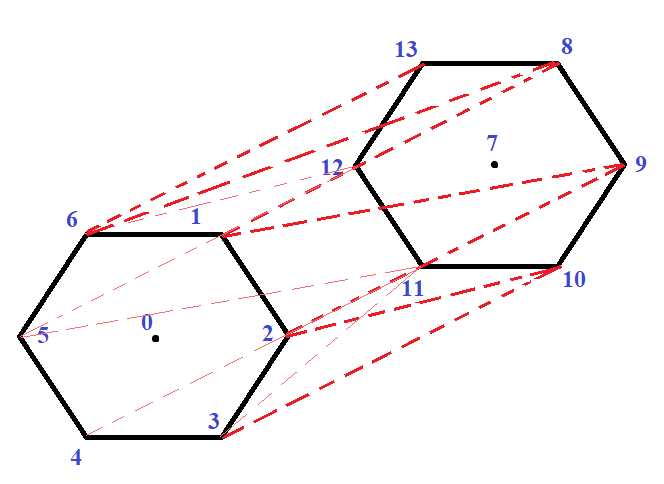
}

return triangles

}

1. **Connect two faces**

In order to connect two faces, we must develop the algorithm. The principles are the same as the base creation.



Each corresponding vertex is connected with each other, along with the next/previous vertex. For example,

A B C A’ C’ F

[1,8,9] - [1,9,2]

[2,9,10] - [2,10,3]

[3,10,11] - [3,11,4]

[4,11,12] - [4,12,5]

[5,12,13] - [5,13,6]

[6,13,8] - [6,8,1]

Two offsets are then defined.

Offset1 = offset 2 //Initial 0

Offset2 = offset1 + NumSides + 1

The pattern can then be thought of as:

1. [(offset1) + 1], (offset2) + 1, (offset2) + 2] [(offset1) + 1, (offset2) + 2, (offset1) + 2]
2. [(offset1) + 2, (offset2) + 2, (offset2) + 3] [(offset1) + 2, (offset2) + 3, (offset1) + 3]
3. [(offset1) + 3, (offset2) + 3, (offset2) + 4] [(offset1) + 2, (offset2) + 4, (offset1) + 4]
4. …
5. …
6. [(offset1) + a, (offset2) + b, (offset2) + c] [(offset1) + a, (offset2) + c, (offset1) + f]

Note that this is repeated as many sides as the shape has (hexagon has six sides).

Even more generalized, it can be described as the following,

[A, B, C] [A’, C’, F]

Where A’ and C’ are the same values given in A and C respectively but perform the additional operation of increasing the corresponding variable added.

For changing the culling direction of the sides, the ordering of the last two pairs can be changed.

[A, C, B] [A’, F, C’]

Then, the algorithm for creating sides is:

CreateSides(offset1, offset2, cullingDir)

{

Int a = 1, b =1, c = 2, f = 2;

a\_offset = f\_offset = offset1;

b\_offset = c\_offset = offset2;

LimitedInt A, B, C, F;

A = a\_offset;

B = b\_offset;

C = c\_offset;

C.Max = offset2 + NumberSides

C.Min = offset2 + 1

F = f\_offset;

F.Max = offset1 + Numbersides

F.Min = offset1 + 1

List<int> sides

Use variables to set ordering.

Change ordering depending on culling direction

Perform A, B, C, A’ C’, or F at the appropriate order.

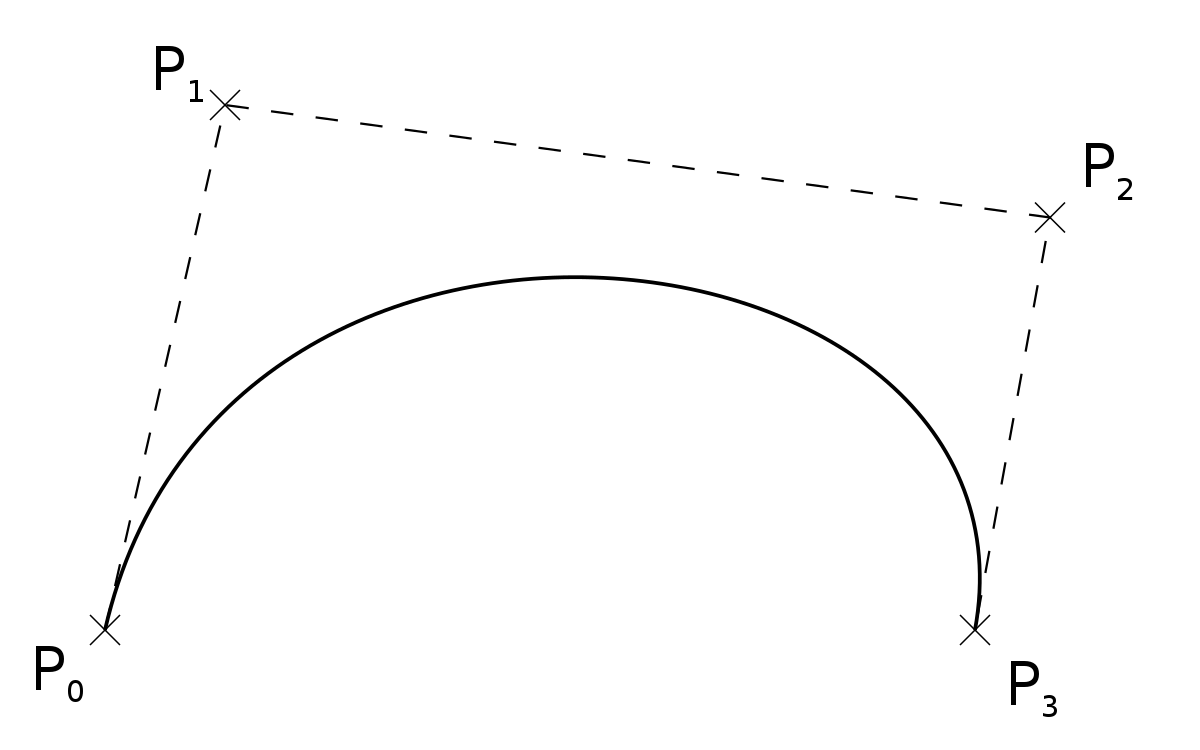
Can be implemented using a for loop for each vertex, and some if else statements.

return sides;

}

1. **Add any # of points in a 3D space and connect faces**

Ropes form curves in a 3D space. To make the creation of a rope easier, rather than specifying each point manually, four points can be used to interpolate a curve – more specifically, a Bezier curve. Here are [multiple](https://www.youtube.com/watch?v=pnYccz1Ha34) [good](https://youtu.be/dXECQRlmIaE) resources on the topic of Bezier curves.



The cubic Bezier curve is given by:

Where B(t) gives a y-coordinate for a t value between 0 and 1. Thus, for 3D we use the same formula butu on all 3 axis. In Unity, it is possible to simply multiply by a Vector3 object. We then use these positions as the centers of the mesh’s rings.

The main code-flow is as follows:

Create base vertices

Create triangles for base

offset1 = offset2

offset2 = offset1 + Numsides + 1

var Quality //from 1 to infinity

NormalizedQuality = 1 / Quality = Unit

for (Quality)

{

CreateRingVertices(GetBezierPoint(NormalizedQuality), GetBezierPoint(NormalizedQuality + unit))

CreateSides(offset1, offset2)

Attach vertices to vert list

Attach ints to triangle list

offset1 = offset2

offset2 = offset1 + Numsides + 1

NormalizedQuality = NormalizedQuality + Unit

}

Convert lists to arrays

The mesh now requires a texture to be added. This is done with a couple of method calls to the Unity library. However, a UV mapping must be created for the texture to be applied correctly. This requires an algorithm. This was not part of the original scope and as such the following is taken from [stackoverflow](https://stackoverflow.com/questions/42628741/texture-mapping-on-a-cylinder-in-c-building-a-raytracer)

*Vector2* UV\_Mapper(*Vector3* point)

    {

        float theta = (float)Math.Atan2(point.y, point.x);

        float rawU = (float)(theta / (2 \* Math.PI));

        float u = 1.0f - (rawU + 0.5f);

        float v = point.y % 1;

        return new *Vector2*(u, v);

    }

1. **Add angle calculation to smooth out point connections in 3D**

This can be done through complicated linear algebra concepts. For example, by using a plane (which represents the angle) that passes through a sphere, we the resulting intersection creates a circle (which represents the vertex rings).

Another solution is to use two angles, longitude and latitude, to represent a point on a sphere and create a circle with these inputs.

However, these solutions are much more complicated than is needed. Instead, a ring can be created with the parametric equations. Then, the angle between the current and the next point is calculated with Quaternion.LookRotation(Vector3) and used to multiply by each vertex in a given ring.

Like so:

Normalize distance between current and next position

Create Quaternion with LookRotation(normalized vector3)

for(each vertex in a ring)

{

ringVertices[i] = quaternionAngle \* (ringVertices[i] - center) + center;

}