GOLD SMITH UNIVERSITY

MSC COMPUTER GAMES AND ENTERTAINMENT

Maths & Graphics Assignment 2

Student: Phuong Linh Nguyen
Professor: Frederic Fol Leymarie

BEZIER CURVE AND DIAMOND SQUARE ALGORITHM

1.Overview

Applying Bezier curve in creating curvature surface and organic shape
Attempt at diamond square algorithm for 3d terrain generation.
Using Unity API and C#.

2.Project analysis

Bezier curve

The Maths behind my 3D surface generation is Bezier curve, all boiled down to this formula

$$Bézier(n,t) = \sum_{i=0}^{n} \underbrace{\binom{n}{i}}_{binomial\ term} \cdot \underbrace{(1-t)^{n-i} \cdot t^{i}}_{polynomial\ term}$$

Bézier curves are polynomials of *t*, with the value for *t* fixed being between 0 and 1, here is more simple interpretation of the formula above, representing linear, square and cubic curve respectively:

$$
\begin{aligned}
linear &= (1-t) + t \\
square &= (1-t)^2 + 2 \cdot (1-t) \cdot t + t^2 \\
cubic &= (1-t)^3 + 3 \cdot (1-t)^2 \cdot t + 3 \cdot (1-t) \cdot t^2 + t^3
\end{aligned}
$$

Binomial formula:

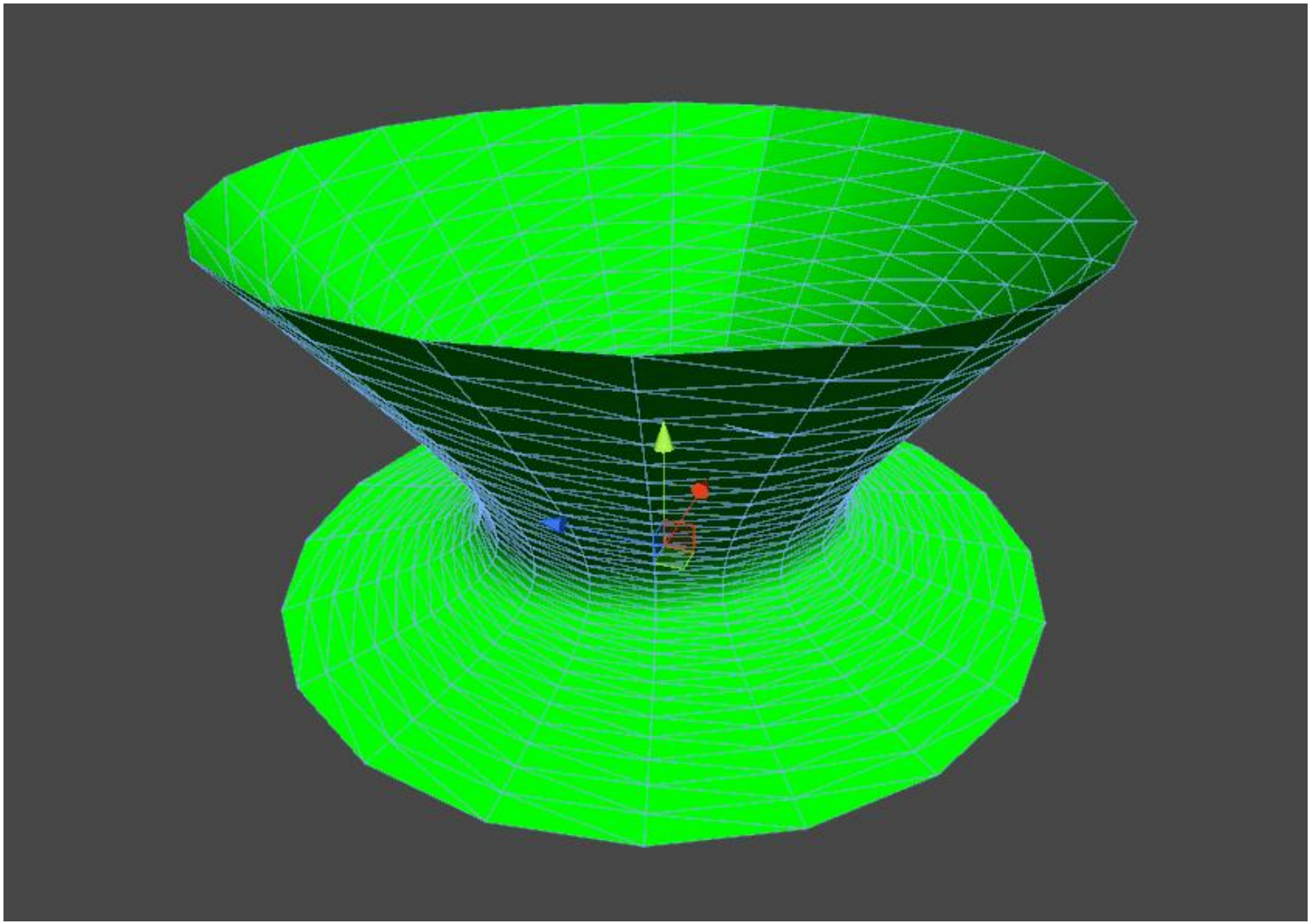
$$
\begin{aligned}
linear &= 1 + 1 \\
square &= 1 + 2 + 1 \\
cubic &= 1 + 3 + 3 + 1 \\
hypercubic &= 1 + 4 + 6 + 4 + 1
\end{aligned}
$$

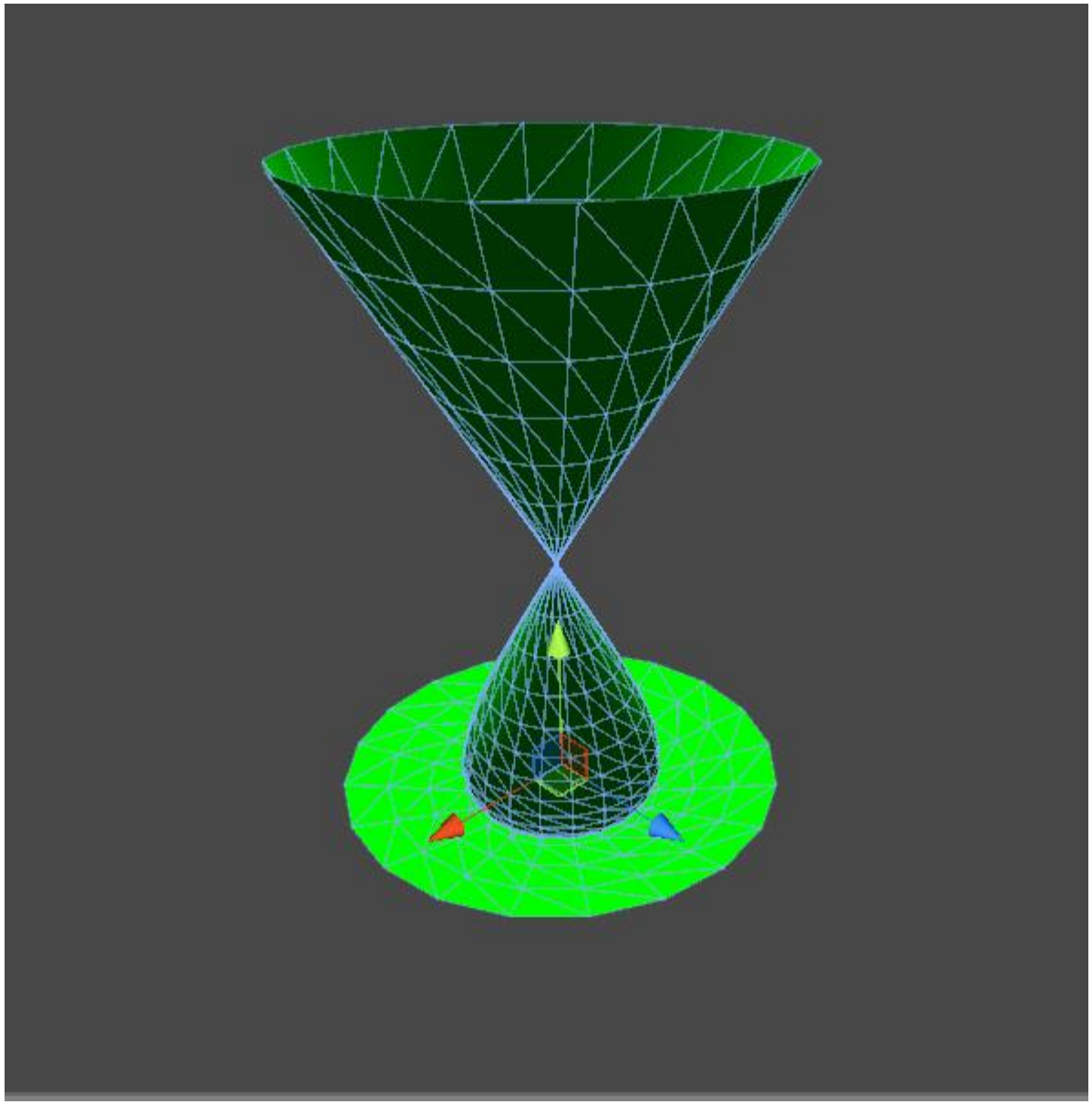
Code sample

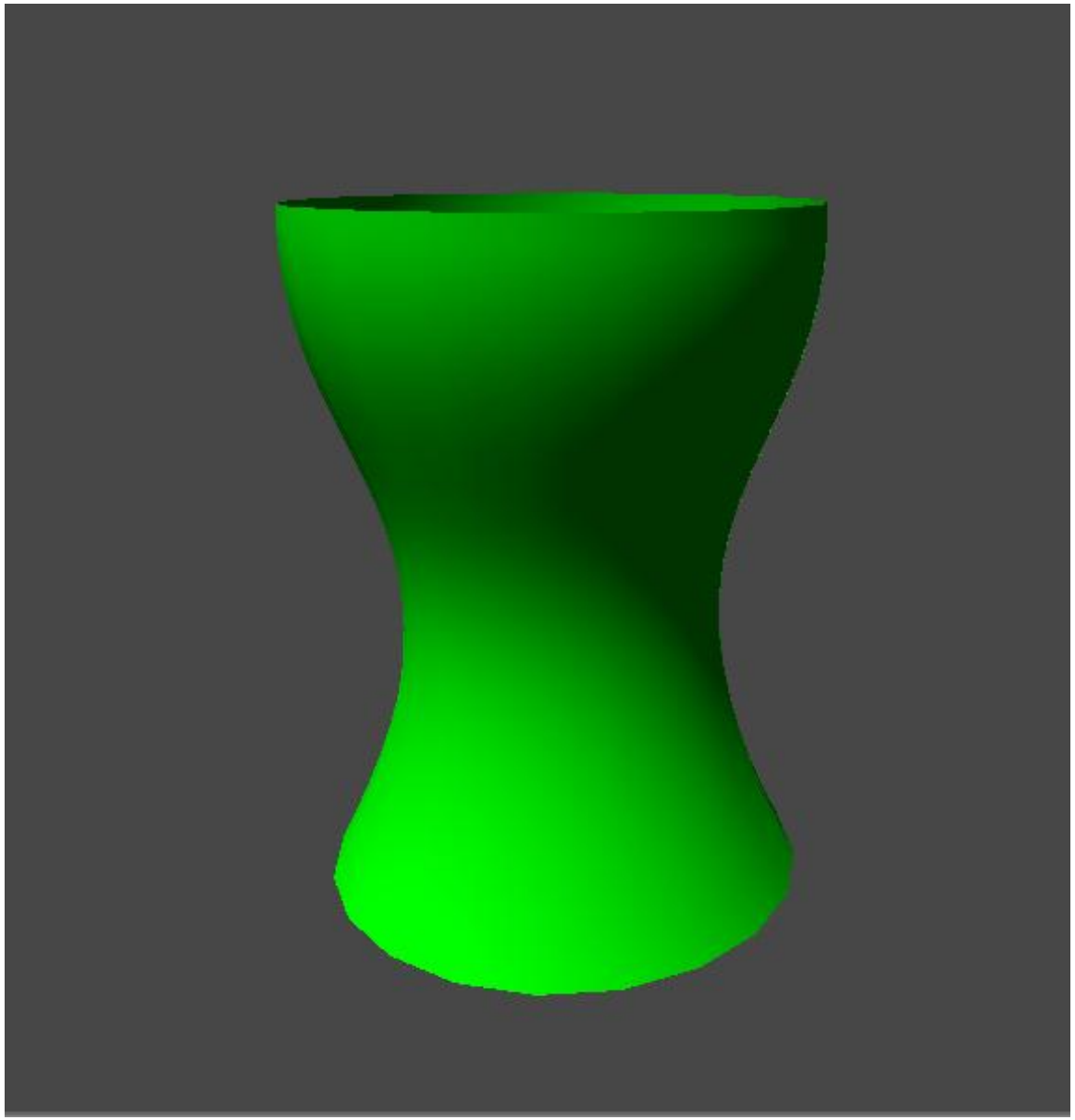
t is 1/segments and when t = 0, the point is w0, t = 1, the point is w3, in between are points interpolated between w0 and w3.

```
float n = 1 - t;
float tSquare = t*t;
float tCubic= t * t * t;
float nSquare = n * n;
float nCubic = n * n * n;
return w0*nCubic + 3 *w1* nSquare * t + 3 *w2* n * tSquare + w3*tCubic;
```

Organic shape created from Cubic Bezier curve:







Curve surface created from hypercubic Bezier curve:

Formula for calculating tangent (as derivatives of the curve):

$$B\acute{e}zier'(n,t) = \sum_{i=0}^{k} \underbrace{\binom{k}{i}}_{binomial\ term} \cdot \underbrace{(1-t)^{k-i} \cdot t^{i}}_{polynomial\ term} \cdot \underbrace{n \cdot (w_{i+1} - w_i)}_{derivative\ weight}, \ with\ k = n-1$$

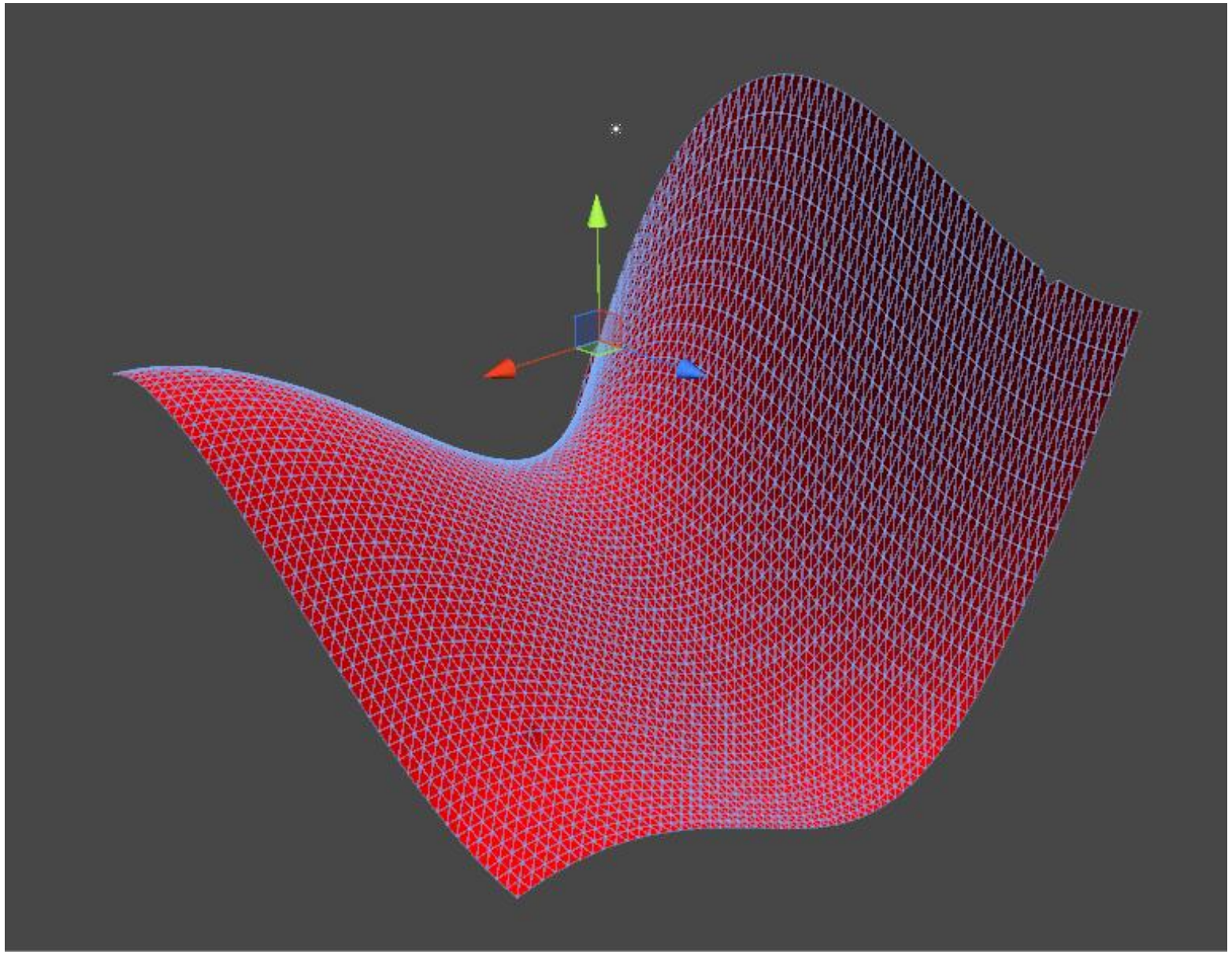
Derivative points:

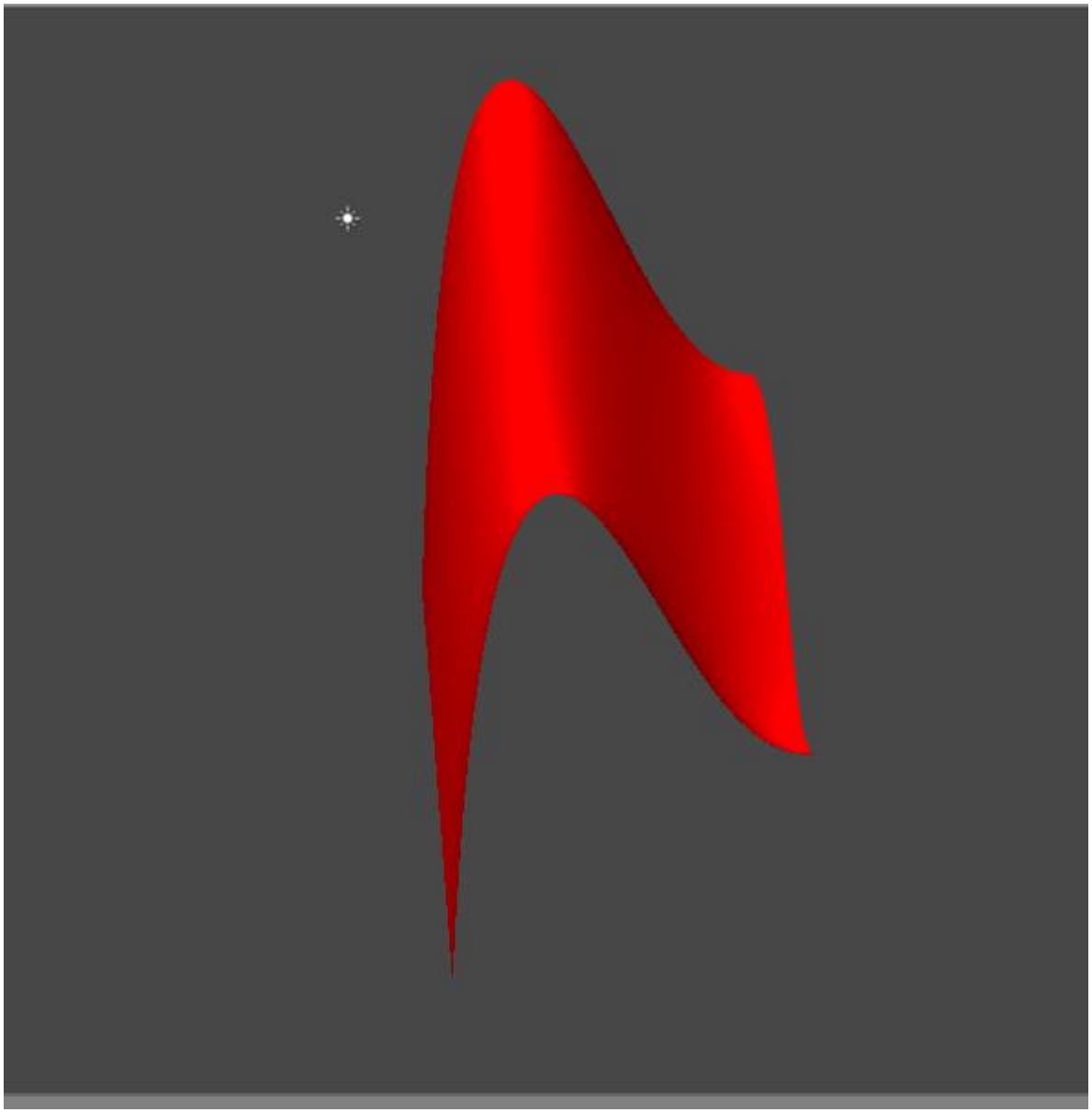
$$\begin{array}{llll} B(n,t), & & w = \{A,B,C,D\} & \\ B'(n,t), & n=3, & w' = \{A',B',C'\} & = \{3\cdot(B-A),\ 3\cdot(C-B),\ 3\cdot(D-C)\} \\ B''(n,t), & n=2, & w'' = \{A'',B''\} & = \{2\cdot(B'-A'),\ 2\cdot(C'-B')\} \\ B'''(n,t), & n=1, & w''' = \{A'''\} & = \{1\cdot(B''-A'')\} \end{array}$$

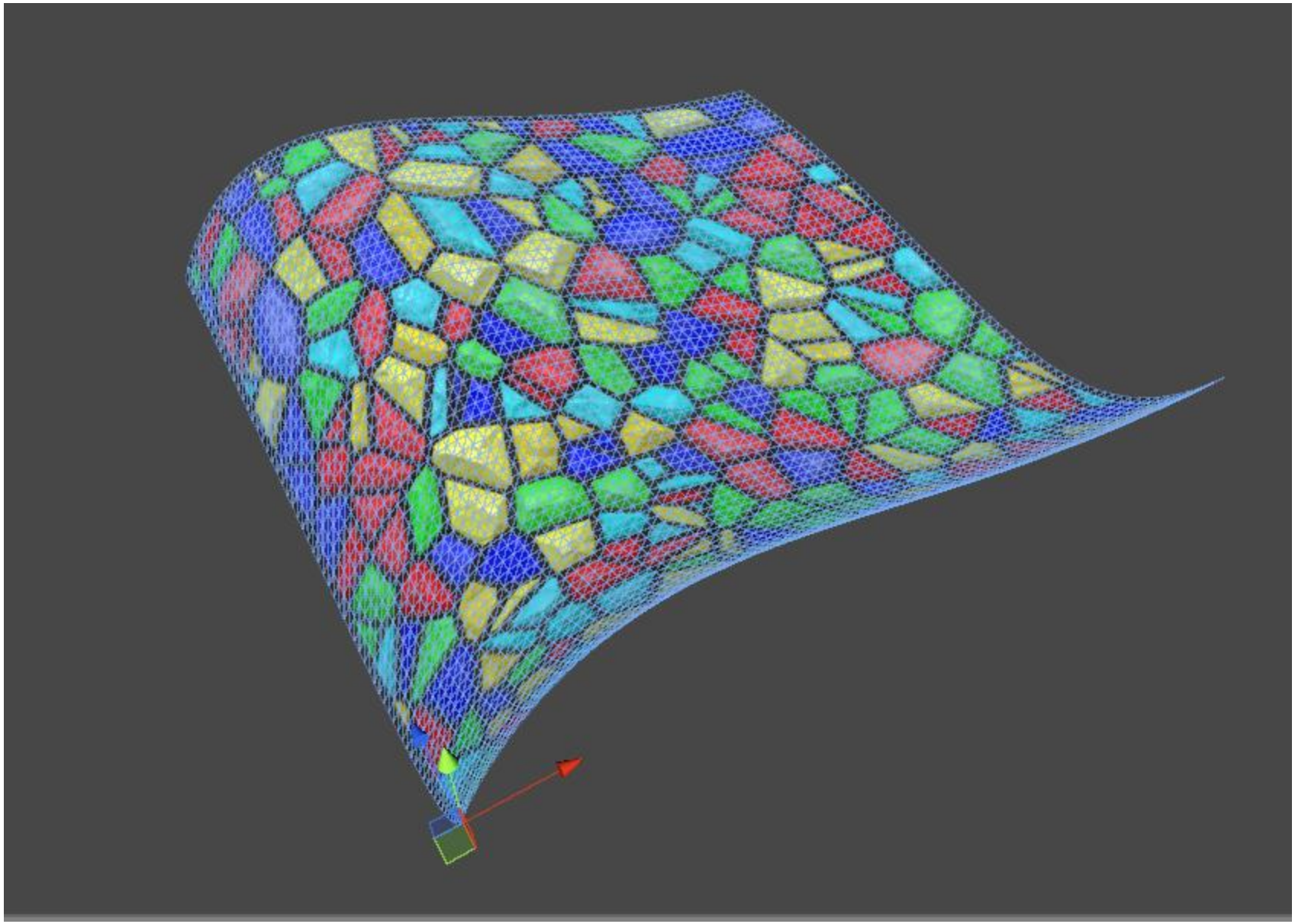
Code sample:

```
float n = 1 - t;
float tSquare = t*t;
float tTripple = t * t * t;
float nSquare = n * n;
Vector3 _tangent = (w1 - w0) * nSquare + 2 * (w2 - w1) * n * t + (w3 - w2) * tSquare;
return _tangent.normalized;
```

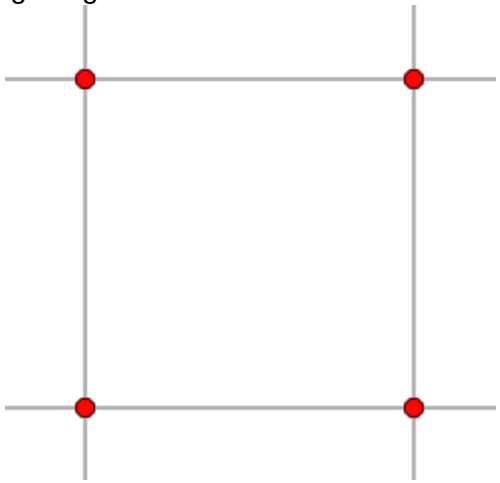
Rotate tangent 90 degree for normal



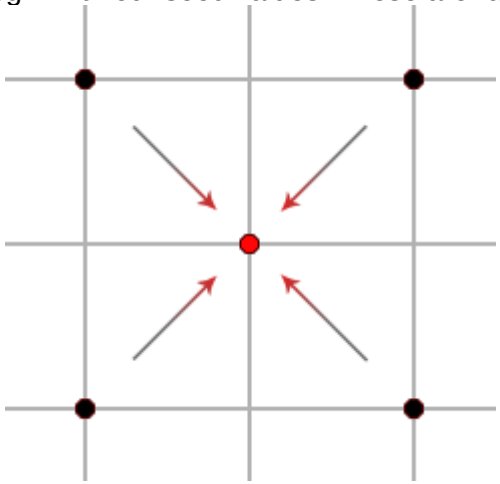




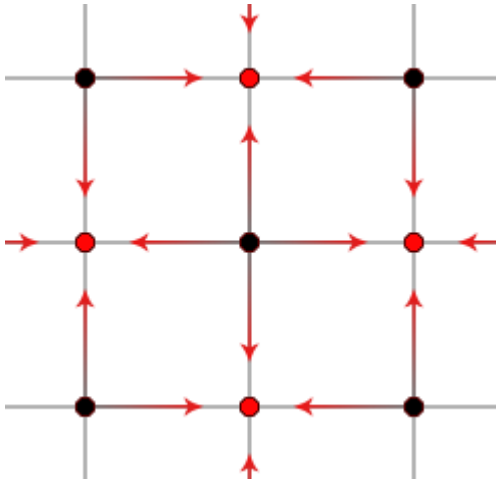
The diamond-square algorithm is an extension of midpoint displacement. The basic idea is to offset points on a grid, with the grid (and offset) getting smaller and smaller with each pass.



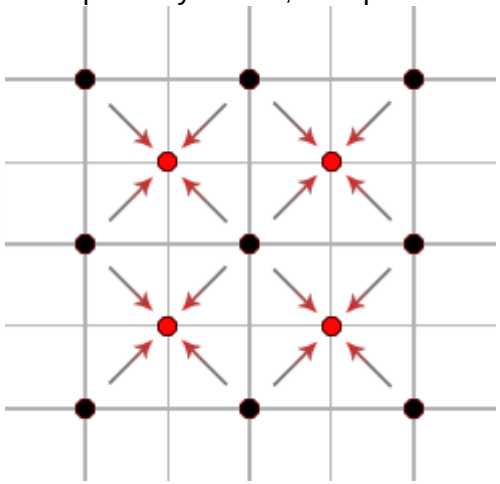
We begin with our seed values. These are random values that will define the overall layout of the terrain.

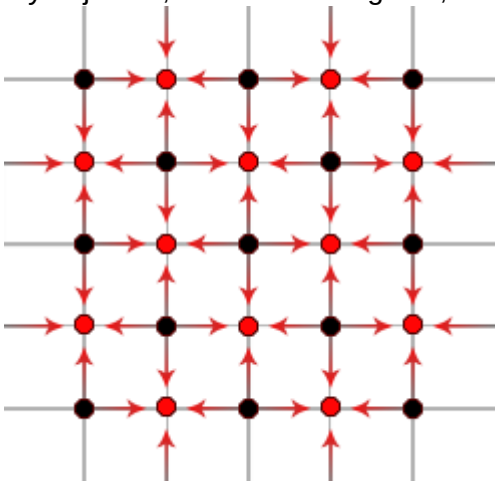


Next, we calculate the height at the centre of each group of four seed points, averaging the outer values and adding a small random offset.



The next step is very similar, except that we calculate from the values directly adjacent, rather than diagonal, from the centre point.





The steps repeat, getting smaller and smaller, until the grid is full.

```
int getIndex(int x, int y)

    {
```

```
            return (x & (segments - 1)) + (y & (segments - 1)) * segments;
        }
```

Then come the centre part of diamond square algorithm:

```
void squareSample(int x,int y,int step,float value)
void diamondSample(int x,int y,int step,float value)
```

quareSample function calculating the middle point inside a rectangular shape:

```
P0        P1          float P0 = getIndex (x - half_step, y + half_step);
                      float P1 = getIndex (x - half_step, y - half_step);
     x                float P2 = getIndex (x + half_step, y + half_step);
                      float P3 = getIndex (x + half_step, y - half_step);
P2        P3
```

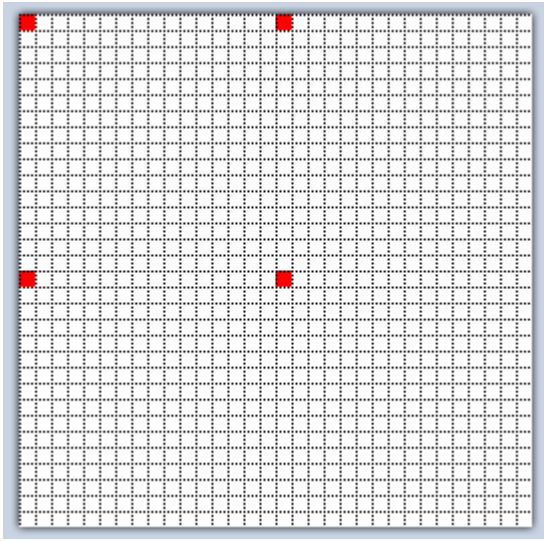
diamondSample calculating the middle point inside a diamond shape:

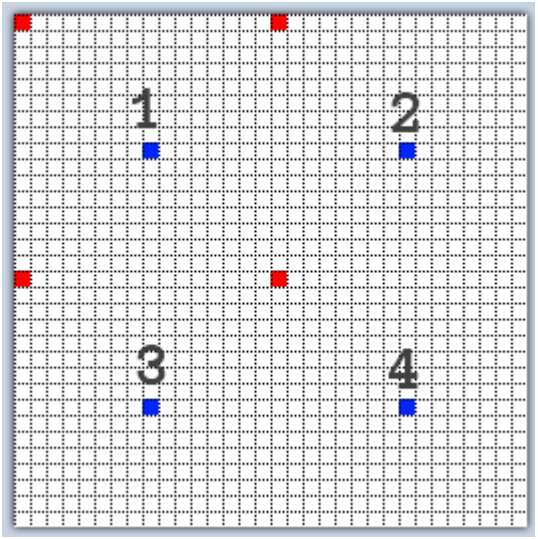
```
     P0               float P0 = getIndex (x - half_step, y);
                      float P1 = getIndex (x, y - half_step);
P2   x   P1           float P2 = getIndex (x + half_step, y);
                      float P3 = getIndex (x, y - half_step);
     P3
```

:

```
while(step>1)
            {
                  diamonSquare(step, Random.value*scale);
                  step/=2;
                  scale *= 0.5f;
            }
```

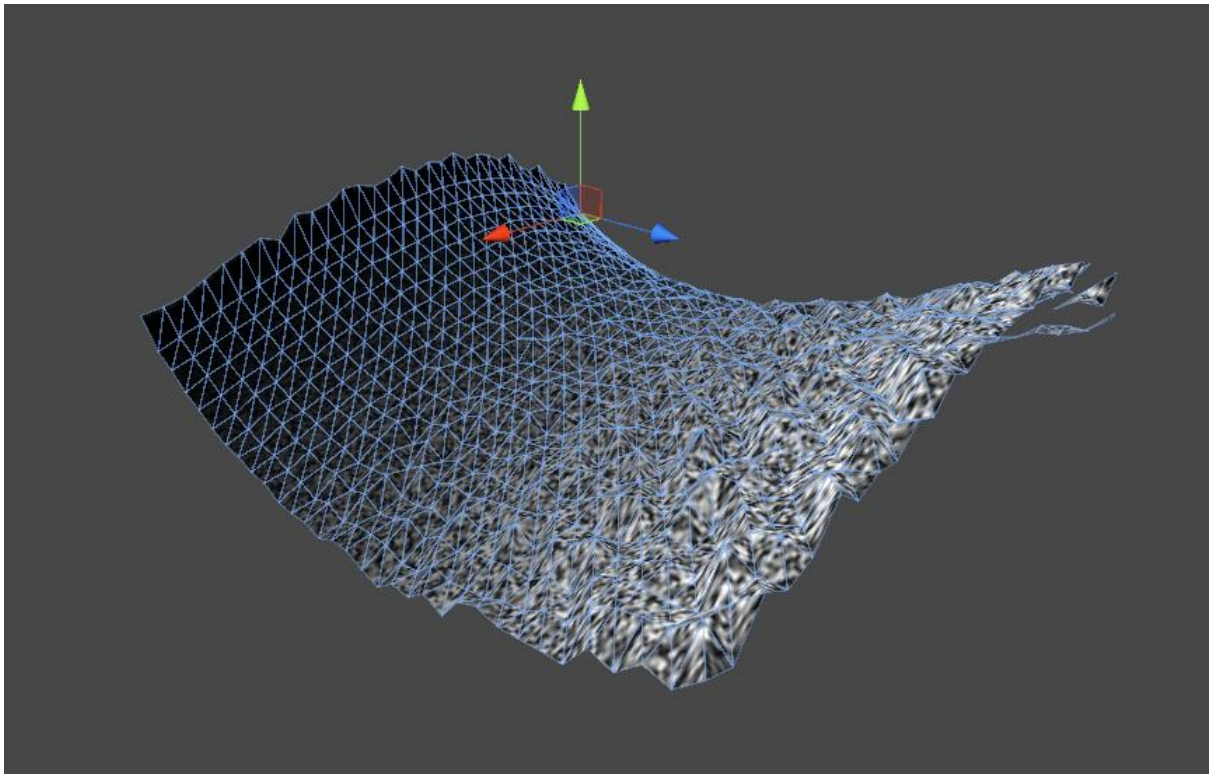
Seed value:





Then the code to set texture from the mid-point values that were to be generated and stores in array:

```
texture = new Texture2D(segments,segments);
                    Color[]color = new Color[midPoint_value.Length];
                    float normalised_value = 1.0f / (max_value - min_value);
                    for (int i = 0; i < midPoint_value.Length; i++)
                    {
                            float input = (float)(midPoint_value[i] - min_value)*normalised_value;//Random.Range(-1.0f,1.0f);
                            color[i] = new Color(input,input,input,input);
                    }
                    renderer.material.mainTexture = texture;
                    texture.SetPixels (color);
                    texture.Apply();
```



3.Conclusion

Here the list of reference:

e-book:

3D Math Primer for Graphics and Game Development (2nd Ed) – Fletcher Dunn & Ian Parrbery.

Online source:

http://pomax.github.io/bezierinfo/

http://gru.bz/2012/10/diamond-square-algorithm-help/

http://www.bluh.org/code-the-diamond-square-algorithm/