How can naturally occurring mountain terrains be modeled using a recursive subdivision process and 3D visualization systems?

Kai Junge

School: Osaka International School

Session: May 2017

Word Count: 3952(excluding in text citations)

Abstract

This essay focuses on the creation of a realistic mountain terrain addressed by the research question, how can naturally occurring mountain terrains be modeled using a recursive subdivision process and 3D visualization systems? To answer this question, I created a computer program using javascript to display a 3D mountain terrain(see the website I created, http://3dmountain.weebly.com/). In the process of creating this program, I explored two terrain creation algorithms based on fractal geometry, and visualization systems to add realism to the fractal structure, so it resembles a natural terrain.

To simplify the problem, this essay first explores the creation of a simpler 2D terrain using the midpoint displacement algorithm. Using similar ideas to the 2D terrain, the creation process of a 3D mountain terrain is then explained. To model the 3D terrain, I first used the same algorithm as the 2D terrain. To improve the terrain created with this algorithm, I explored a second algorithm, the diamond square algorithm. The essay also explains the problems and difficulties faced throughout the creation of the program.

I used two methods to make the mountain terrain look as realistic as possible. Perspective mathematics was needed to project the mountain with 3D coordinates onto the screen with 2D coordinates. Vectors were used to determine the angle between faces of the mountain and the direction of sunlight to calculate the amount of shading.

The final product from this investigation is the mountain drawn by the computer program. The research question is answered to an extent because the mountain created by the program definitely resembles a naturally produced mountain range. However, there are properties of the mountain which deviate from a real mountain terrain, acknowledged in the conclusion.

Word Count: 281 words

Table of Contents

Part 1 : Introduction

1a: Fractals

Complex shapes and structures created by nature can be seen anywhere, from snowflakes to clouds. While these natural structures are common to everyday life, their irregular shapes cannot be explained easily with Euclidian geometry(“Fractal Geometry”). Perfect spheres and straight lines do not appear often in nature.

To explain naturally occurring shapes and patterns using mathematics, Benoit Mandelbrot developed fractal geometry(Mandelbrot)(“Fractal Geometry”). Mandelbrot realized, although naturally occurring shapes seem irregular and random, there is a degree of self-similarity; a property where one section of the shape seems identical to other sections or the whole(“Fractal Geometry”). A classic example is a fern leaf(Mandelbrot)(“Fractal Geometry”). While each leaf seems complex, one leaf looks very similar to the whole fern it self.

Fig. 1. Dokie. Fern. “2592x1944px, 737.27 KB, Fern, #761425, by Dokie.” Lemur.com, 8 May 2015. http://lemerg.com/761425.html. Accessed 23 November 2016.

Like the fern, fractals have both properties of self-similarity and complexity. A classic example of a fractal is the Sierpinski Triangle(“Fractals: Useful Beauty”).

Fig. 2. “Making a Sierpinski Triangle.” Resonance’s Mathematics Page, http://spanishplus.tripod.com/maths/FractalSierpinski.htm. Accessed 23 November 2016.

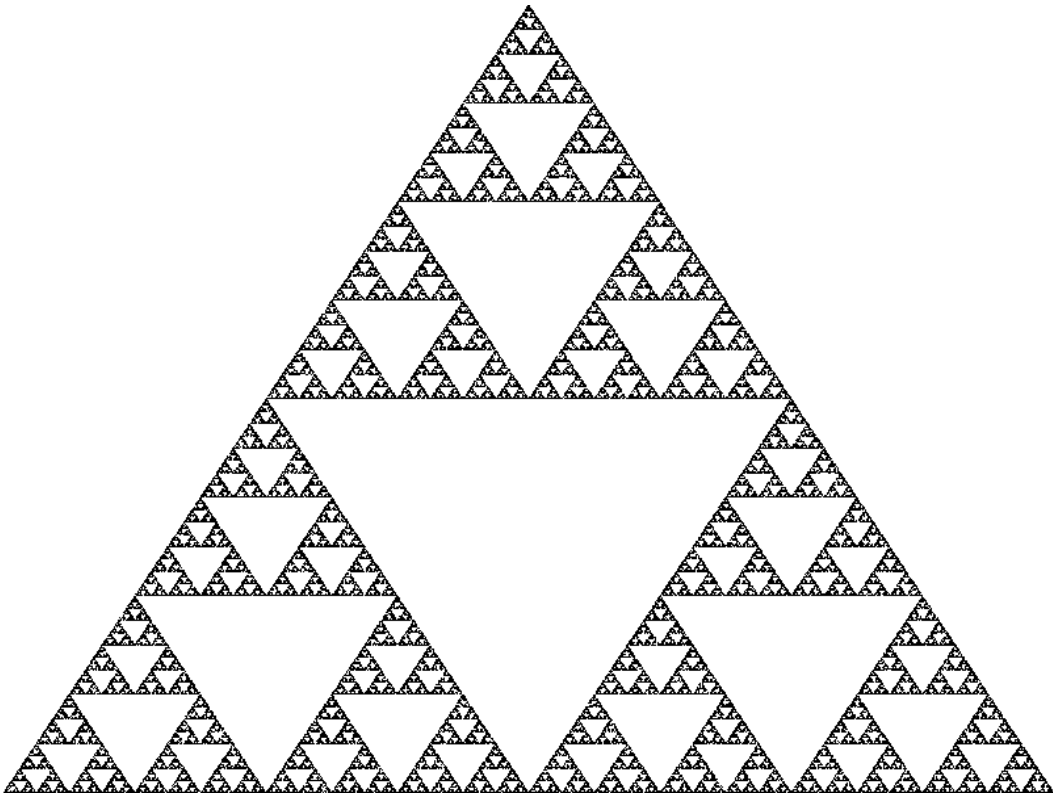


Fig. 3. bilalCh213. “Sierpinski Triangle Fractal - The easiest way to produce randomness.” cplusplus.com, 13 October 2015, http:///articles/LyTbqMoL/. Accessed 23 November 2016.

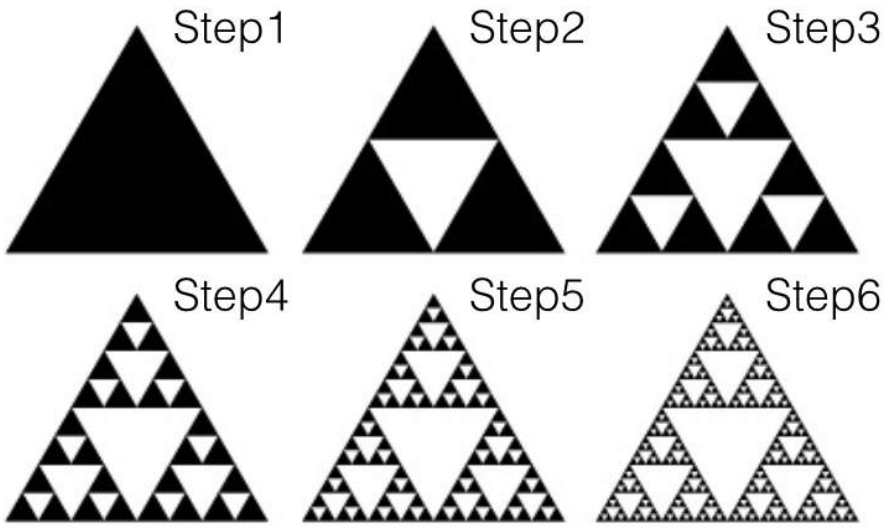


Figure 2 is the Sierpinski Triangle. While the Sierpinski triangle is a complex shape, it is produced by a simple process known as recursive subdivision, shown by figure 3. Recursive subdivision is a repetitive process which begins with a simple shape, and dividing it into smaller pieces. In this example, we begin with an equilateral triangle, which is divided into smaller equilateral triangles in each step. This process is repeated infinitely to create the Sierpinski Triangle.

1b: Fractal Mountain Terrain

Like the fern, a mountain terrain has fractal properties. The random unevenness of a terrain is indistinguishable when a terrain is observed form a far away or from a close distance.

The purpose of the investigation is to show a mountain terrain can be modeled using a recursive subdivision process, given by the research question, how can naturally occurring mountain terrains be modeled using a recursive subdivision process and 3D visualization systems? In this investigation, I created a computer program in javascript which displays a 3D fractal mountain. I focused on the mathematics of the algorithm of creating a fractal mountain terrain and the visual features. It is important for the fractal mountain to closely resemble a real mountain since it has applications in animations, gaming, virtual reality, and movie production.

Part 2 : Recursion

To understand the recursive subdivision process, we must first understand recursion. Recursion is a method used in mathematics and computer programming to determine the solution to a question in terms of itself(Germain)(Riley “What on Earth is Recursion”). There are always two parts to recursion, a function defined by itself and an endpoint used to terminate the recursion process(Germain). The factorial function is famous recursive function(Germain)(Riley “What on Earth is Recursion”).

We know that the factorial function is,

$$factorial(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$$

The factorial function expressed recursively is,

$$factorial(n) = n \cdot factorial(n-1)$$

Another part needed for this function is the endpoint to end the recursive process. If we do not have an endpoint, the function will keep recursing it self. In this case the endpoint is,

$$factorial(0) = 1$$

With these rules we can determine a solution for a factorial function for any non-negative integer value of n. For example:

$$\begin{aligned} factorial(3) &= 3 \cdot factorial(2) \\ &= 3 \cdot 2 \cdot factorial(1) \\ &= 3 \cdot 2 \cdot 1 \cdot factorial(0) \\ &= 3 \cdot 2 \cdot 1 \cdot 1 = 6 \end{aligned}$$

A recursive function defined by the function itself will have sections within the function which have properties of the whole. This is similar property to a subdivided fractal with self-similarity. Therefore, the principle of recursion will be essential for creating fractal mountain terrains.

Part 3 : 2D Mountain

3a: Midpoint Displacement Algorithm for 2D mountains

Before investigating methods to create a 3D mountain, we will first look at a simpler version, a 2D fractal mountain. The method to create such a mountain is known as the midpoint displacement method, first introduced by Benoit Mandelbrot in his *Fractal Geometry of Nature*(Mandelbrot), but explained more clearly in the website *Generating Random Fractal Terrain*(Martz).

Fig. 4.

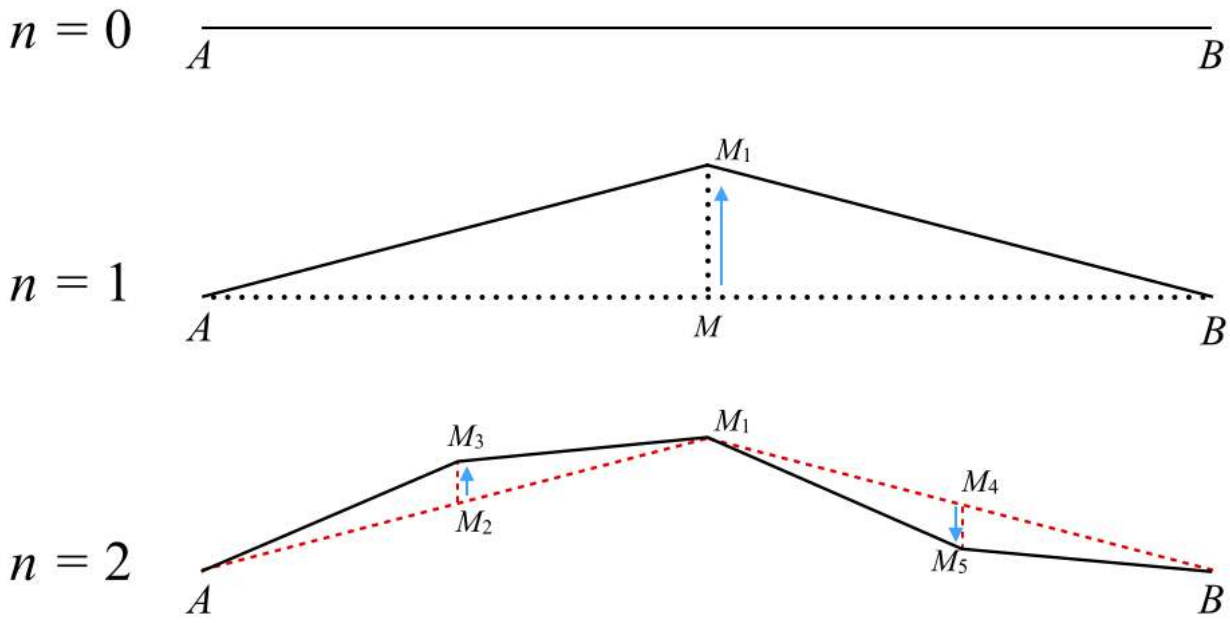


Figure 4 is a diagram that shows the first two iterations of creating a 2D mountain, where n is the number of iterations. First, we start with a line segment with AB. This line segment will be the base for the mountain(Martz). In the first iteration(when $n = 1$), we first find the midpoint of AB, labeled as point M on the diagram(Martz). Once we found the midpoint, it is displaced along the vertical axis, giving us M_1(Martz). The displacement is dependent on a random value and the original length of the line(Martz). Finally, we complete the first iteration by connecting the three points A, M_1 and B(Martz).

For the second iteration(n=2), we apply the same rules as the first iteration separately, to each of the line segments, AM_1 and M_1B. In the diagram above, by applying these rules to AM_1, its midpoint M_2 is displaced to M_3. For M_1B, its midpoint M_4 is displaced to M_5.

Further iterations can be determined with the same rules. For example, the third iteration would apply the same rules to the four line segments determined in the second iteration.

3b: Producing a 2D mountain using the Midpoint Displacement Algorithm

By using the method discussed in 3a, a computer program to draw 2D mountains can be made.

Fig. 5.

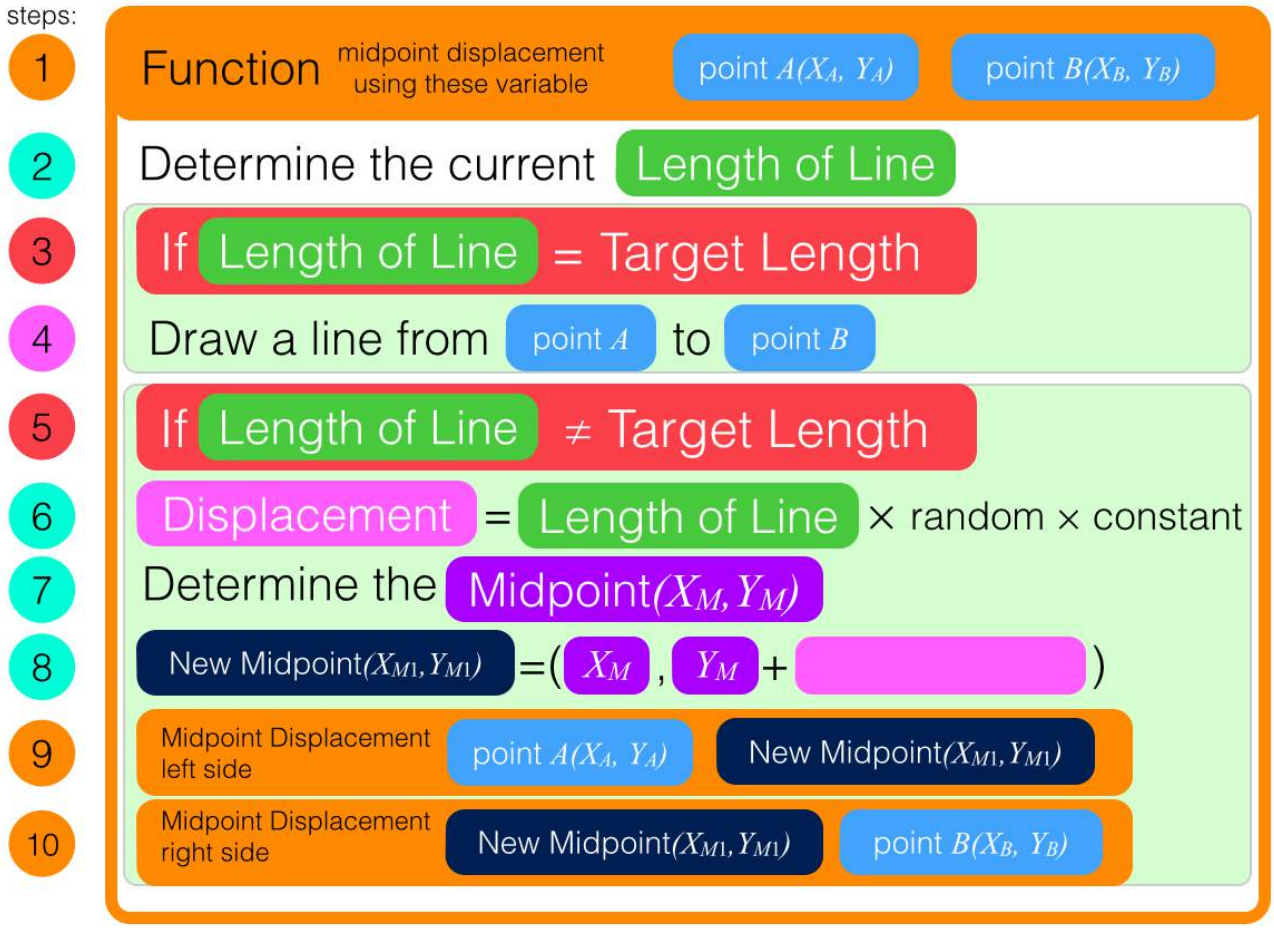


Figure 5 shows the structure of the program. The program will create a 2D mountain with n iterations, where n is inputed by the user before the program is executed. To begin the program, we need to know the coordinates of the ends of the line segment, which is represented by point A and point B, shown in step 1.

In steps 3 and 5, the program will check whether the mountain has reached the target length, which depends on n. This step acts as the endpoint of the recursion. To do this, we must determine the distance between the x-coordinates of two consecutive points in the n^{th} iteration and compare it to the distance between the x-coordinates of the current two points the program is working with. Since

the algorithm only displaces the midpoint in the vertical direction, the distance between two consecutive *x*-coordinates at a particular iteration will all equal to each other.
The current distance is,

$$current\ distance = x_B - x_A$$

Which is the "Length of Line" determined in step 1. *A* and *B* represent the current two points. The number of line segments after *n* iterations is,

$$number\ of\ lines = 2^n$$

Therefore, the distance between the *x*-coordinates after *n* iterations would be

$$Target\ Length = \frac{the\ distance\ between\ the\ original\ two\ points}{2^n}$$

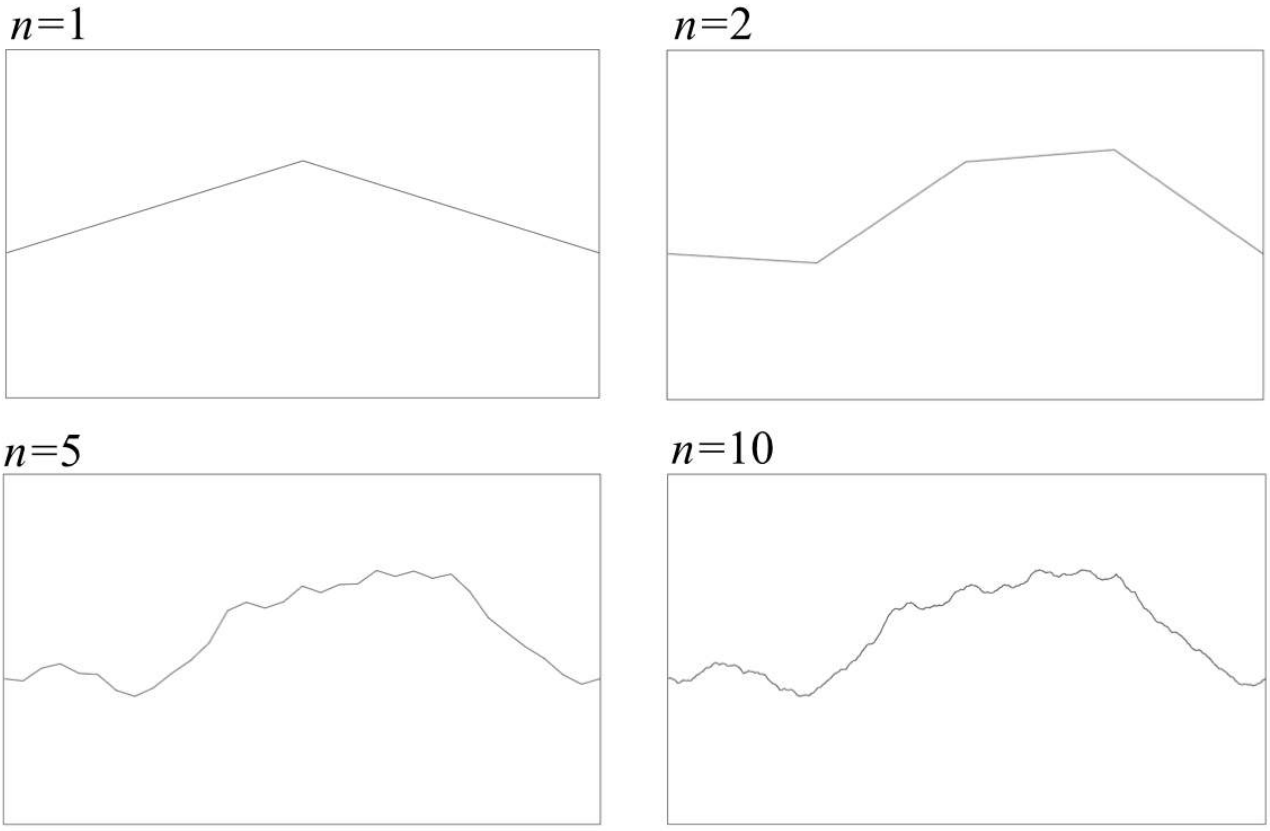
In step 3, when the current distance equals the target length, we stop the recusing process and draw a line to one point to another as in step 4. When the two quantities are not equal, shown in step 5, we continue the iteration process.

In steps 6~8 we are calculating the new displaced midpoint, in the same way explained in 3a. In step 6, the length of line is being multiplied by a random value, ranging from -1 to 1, and a constant. This constant is responsible for the steepness of the mountain. Typically, a constant between 0.05 and 0.5 creates a realistic looking 2D mountain.

Steps 9 and 10 are key to making this function recursive, since it defines the function in terms of itself. These two steps apply the same process to each side of the line.

By completing these steps, we end up with a shape(see figure 6) created using javascript that resembles a 2D mountain.

Fig. 6.



Part 4 : 3D Mountain

4a: 3D Perspective

In order to create a 3D fractal mountain, we first must determine a method to draw a 3D object on a 2D screen relative to the viewer. We have to do this because the 3D mountain with 3 coordinates will be drawn on a computer screen with only 2 coordinates. In this investigation, we will use the *x*-*y*-*z* coordinate system shown in figure 7.

Fig. 7

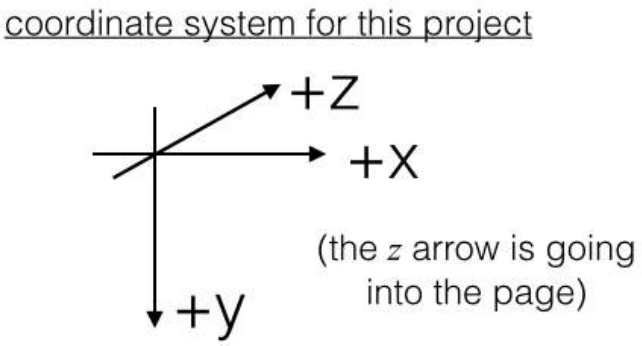


Fig. 8.

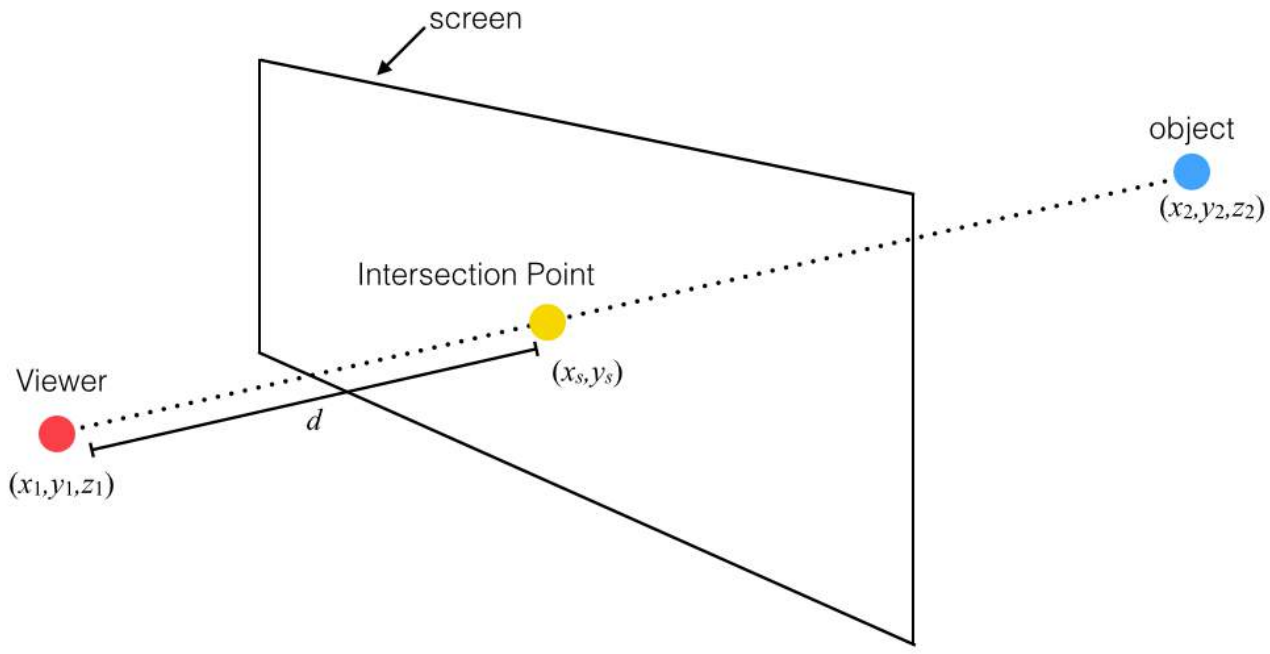


Figure 8 shows the current situation visualized. If we draw a straight line from the viewer and the object in 3D space, it will intersect with a screen that is distance *d* away from the viewer. Our goal is to determine the coordinates of the intersection point using the coordinates of the viewer, object, and the distance of the screen from the viewer.

First, we will determine the *x*-coordinate of the point on the screen.

Fig. 9.

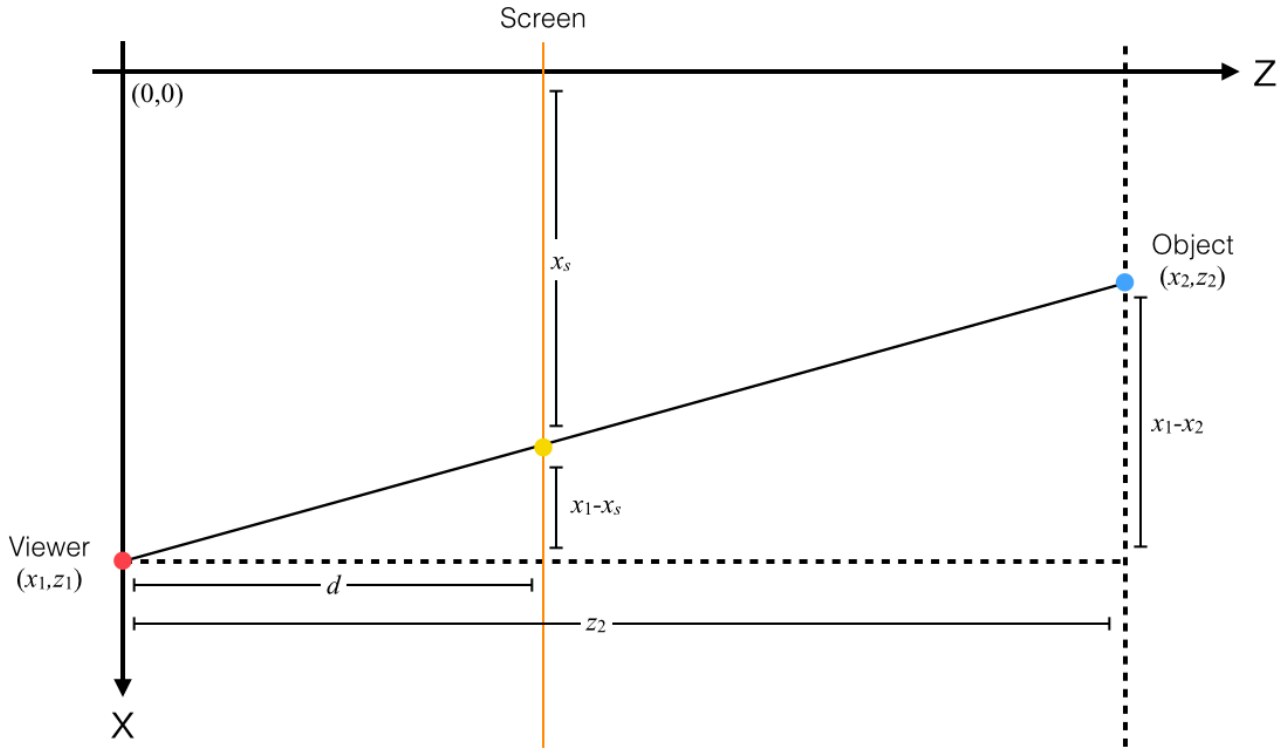


Figure 9 shows the same situation when viewed from directly above the *x*-*z* plane. This way, any differences in the *y*-axis will not affect the diagram, eliminating the *y*-coordinate from the expression.

From this, we can determine an expression for x_s

By using similar triangles ,

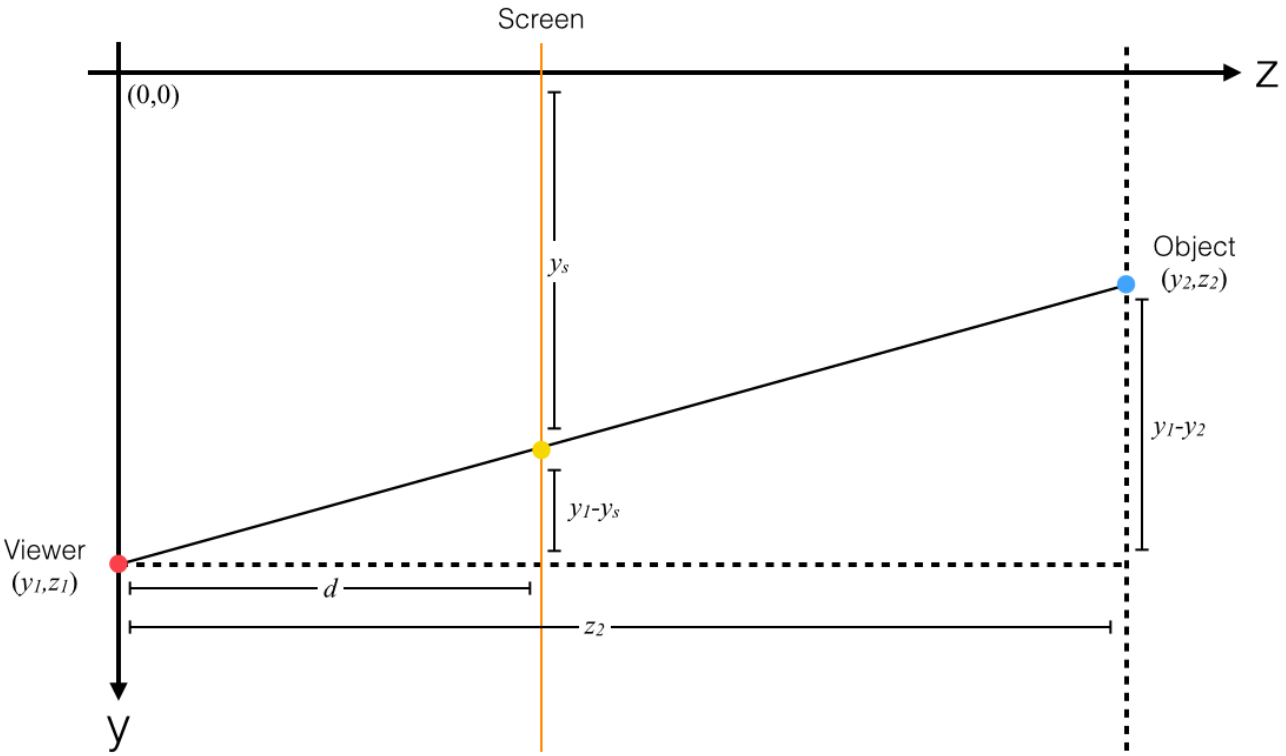
$$\frac{x_1-x_s}{x_1-x_2}=\frac{d}{z_2}$$

$$x_1-x_s=(x_1-x_2)\cdot\frac{d}{z_2}$$

$$-x_s=(x_1-x_2)\cdot\frac{d}{z_2}-x_1$$

$$x_s=x_1-(x_1-x_2)\cdot\frac{d}{z_2}$$

Fig. 10.



Similarly, the *x*-coordinate of each of the points can be eliminated by viewing the diagram directly above the *y*-*z* plane, shown in figure 10.

Using the same method, we can conclude that,

$$y_s = y_1 - (y_1 - y_2) \cdot \frac{d}{z_2}$$

These equations allows us to correctly determine the coordinates of the point of where the object should be drawn on the computer screen. However, one assumption was made in the derivation, and it is that the *z*-coordinate of the viewer will always be 0 to simplify the problem. This is acceptable, because we will not move the position of the viewer in this project.

To conclude, we can convert 3D coordinates into 2D coordinates represented by (x_s, y_x) as such,

$$(x_s, y_s) = \{ x_1 - (x_1 - x_2) \cdot \frac{d}{z_2} \quad , \quad y_1 - (y_1 - y_2) \cdot \frac{d}{z_2} \}$$

4b: Midpoint Displacement Algorithm for 3D mountains

The midpoint displacement algorithm discussed in part 3a can be used to create a 3D mountain. With the 2D mountain, a line was used to begin the recursive. For a 3D mountain, we can begin with a plane, then apply similar algorithms to displace the midpoints.

Fig. 11.

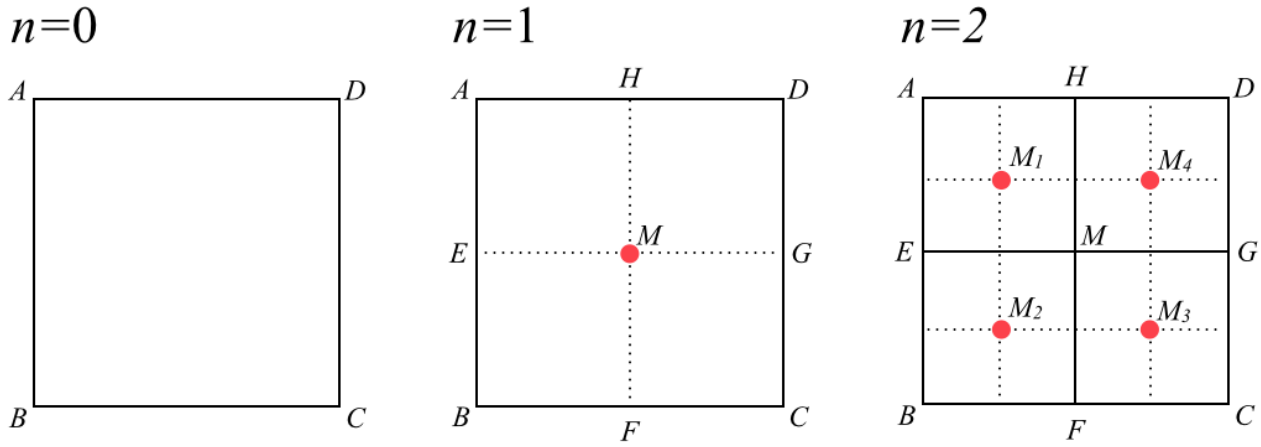


Figure 11 shows the progression of a fractal mountain from directly above it, when the midpoint displacement algorithm is applied(*n*=number of iterations). The process begins with a 2D square. For the first iteration, just like in the 2D mountain, we find the midpoint of the square(point *M*) and displace it in the vertical direction by a random amount. To find the midpoint, we simply take the average of the coordinates of each of the corners.

$$(x_M, y_M, z_M)=\left(\frac{x_A+x_B+x_C+x_D}{4}, \frac{y_A+y_B+y_C+y_D}{4}, \frac{y_A+y_B+y_C+y_D}{4}\right)$$

With this midpoint *M*, we have created 4 new quadrilaterals: *AEMH*, *EBFM*, *MFCG*, and *HMGD*(when the midpoint is displaced, the squares become distorted in 3D). For the second iteration, we take each of the four new squares and apply the same function to it. This creates 4 new midpoints, $M_1 \sim M_4$.

4c: Producing a 3D mountain using the Midpoint Displacement Algorithm

By using the method discussed in 4b, we can produce a computer program which draws a 3D mountain.

Fig. 12.

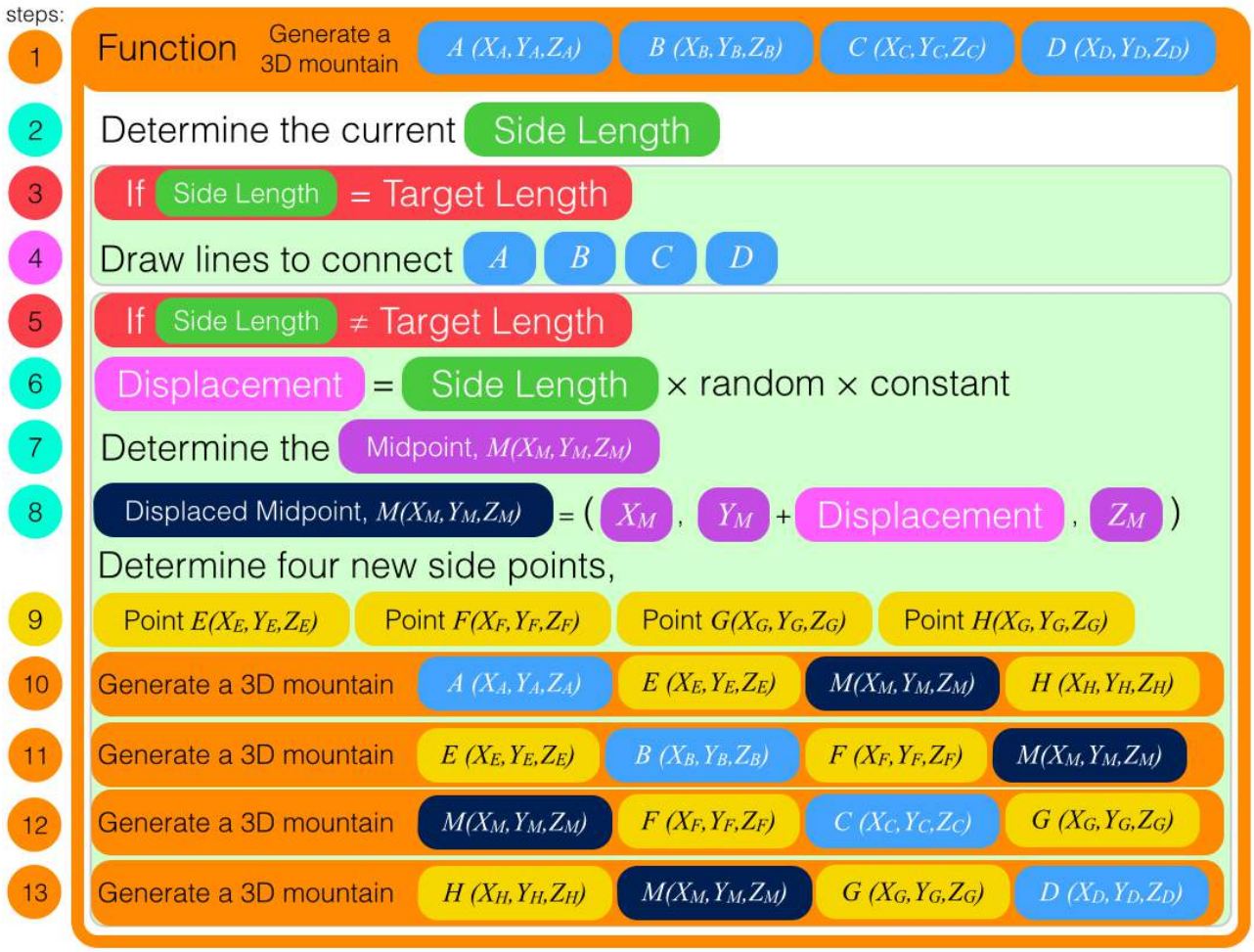


Figure 12 shows how the program would be structured. The points correspond to figure 11. This program will create a 3D mountain until the mountain reaches the target length, which depends on n, where n is inputed by the user. The program structure is very similar to the creation of the 2D mountain. We have to input the x,y,z coordinates of each of the four corners(A,B,C,D) to begin the program, shown in step 1.

To check the program has reached the n^{th} iteration, it will check the side length of the current square, and compare it to what the side length of the square will be in the n^{th} iteration, which is done in steps 3 and 5.

The current side length is,

$$\sqrt{(x_B - x_A)^2 + (z_B - z_A)^2}$$

With each iteration, the number of squares will quadruple but the side length only halves. Therefore,

$$Target\ Length = \frac{the\ distance\ between\ the\ original\ two\ points}{2^n}$$

Once the side length and the target length are equal, the program will draw a line to each corner as shown in step 4. Here, each of the four points, *ABCD* are converted into 2D coordinates with the method discussed in 4a.

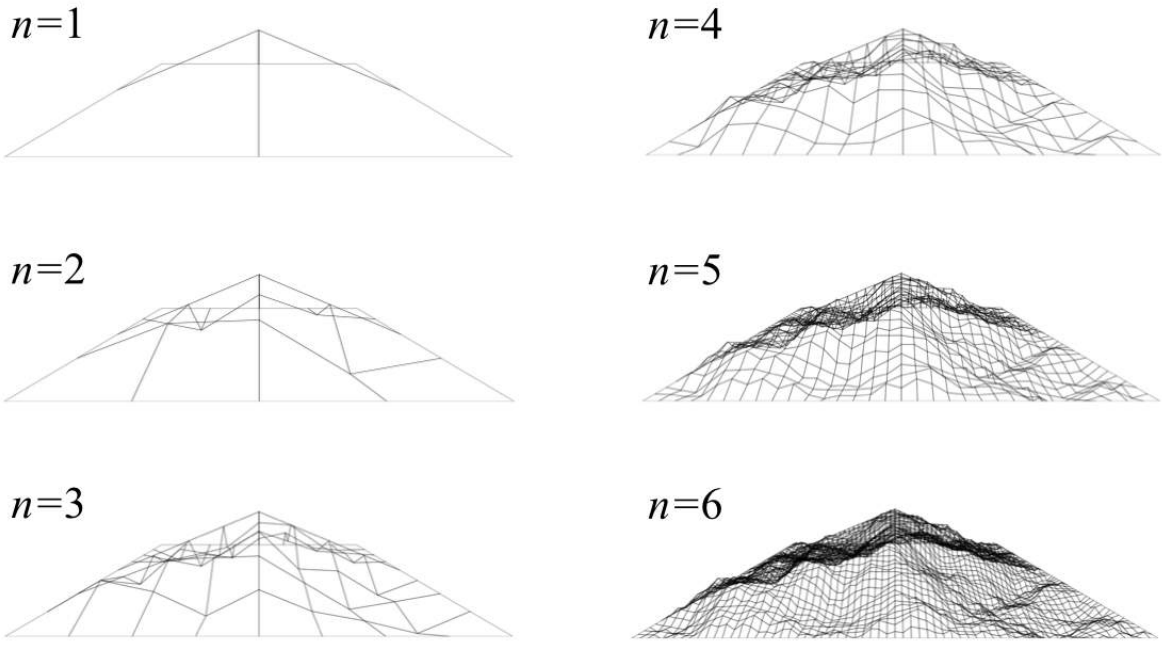
Steps 6 to 8 calculate the displacement value, finds the midpoint, and then displaces the *y*-coordinate of the midpoint. To continue the recursive process, we need 4 midpoints on the sides of the square, marked *EFGH* on figure 11, shown in step 9. To find one of these points, for example point E, we take the midpoint of the two ends of the side it is on.

$$point\ E = \left(\frac{x_A + x_B}{2}\ ,\ \frac{y_A + y_B}{2}\ ,\ \frac{z_A + z_B}{2} \right)$$

Finally, steps 10 to 13 take each of the four quadrilaterals(*AEMH, EBFM, MFCG, HMGD*), and applies the same function to it, making this function a recursive process.

The product of this program written in javascript can be seen in figure 13, which resembles a 3D mountain.

Fig. 13.



4d: Diamond Square Algorithm

As the number of iterations is increased, a flaw in the mountain created using the midpoint displacement algorithm becomes visible.

Fig. 15.

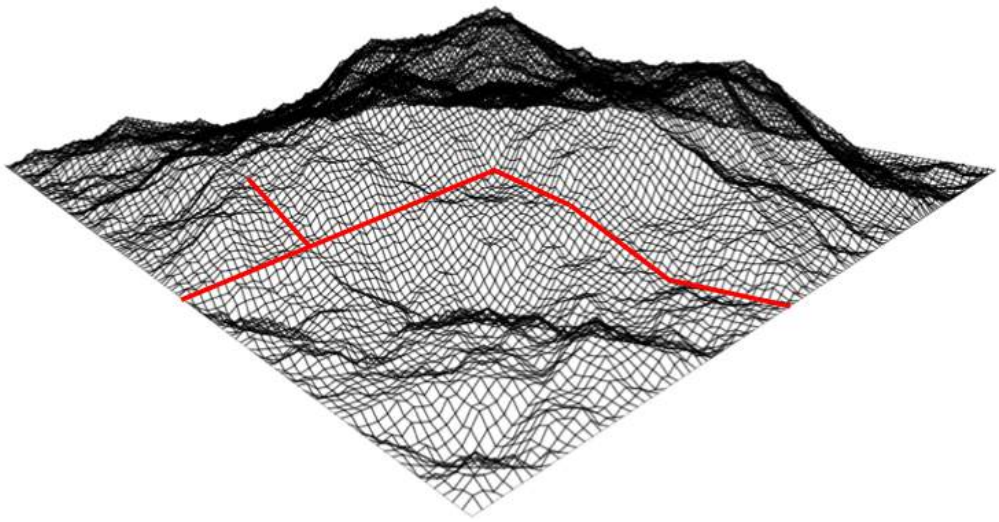


Figure 15 is the same mountain used figure 14, but when n=7. There are obvious straight and unnatural looking sections, marked in red. These lines make the mountain look artificial. This is in fact an inevitable result of the midpoint displacement algorithm..

Fig. 16.

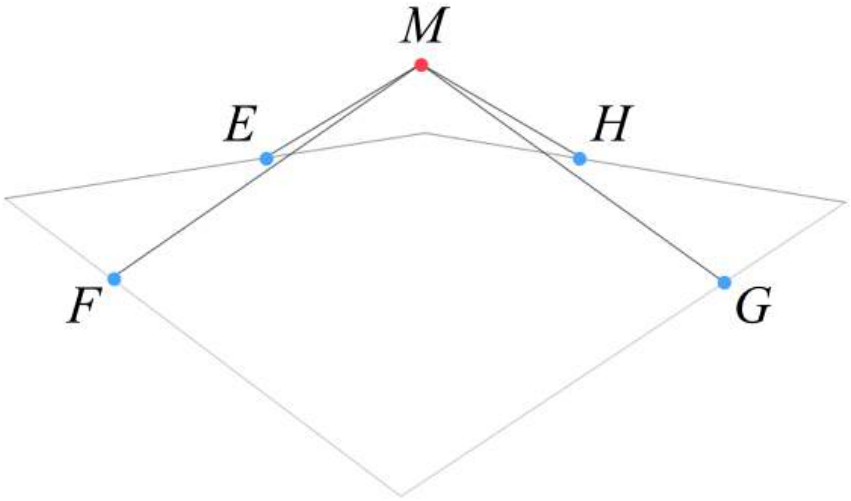


Figure 16 is a mountain at n=1. Lines *EM, FM, GM, and HM,* will remain as they appear regardless of the number of iterations, since it is only the center of the quadrilateral which gets displaced.

To solve this problem, we must use a different algorithm to create the mountain. A solution is the diamond square algorithm developed in an article, *Computer Rendering of Stochastic Models* in the journal *Communications of the ACM*(Fournier 371-384),and was explained again in the article *The Definition and Rendering of Terrain Maps* in the journal *ACM SIGGRAPH Computer Graphics*(Miller 39-41) as an effective effective method for terrain generation. The method is more clearly explained in the webpage *Generating Random Fractal Terrain*(Martz).

Fig. 17.

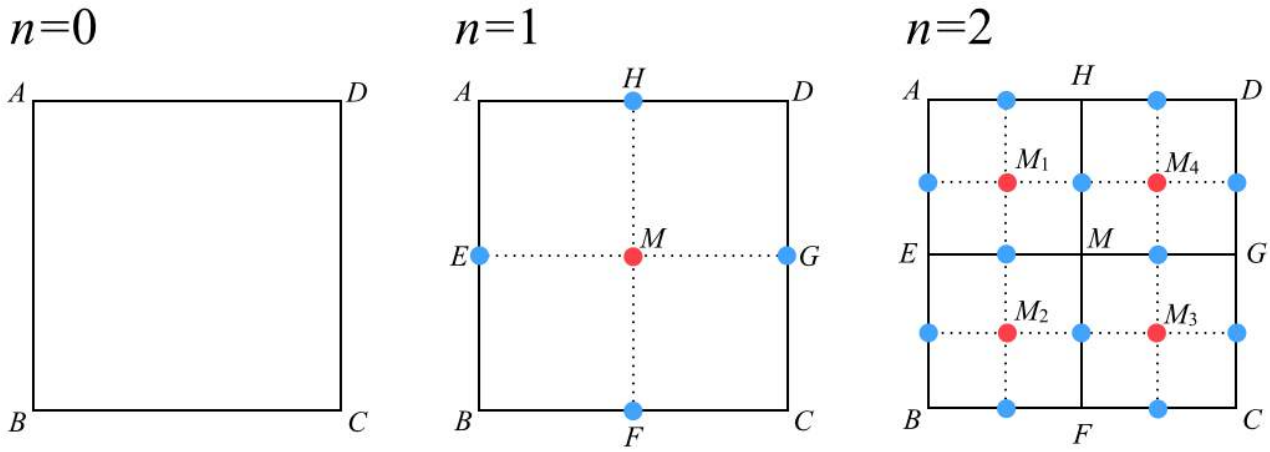


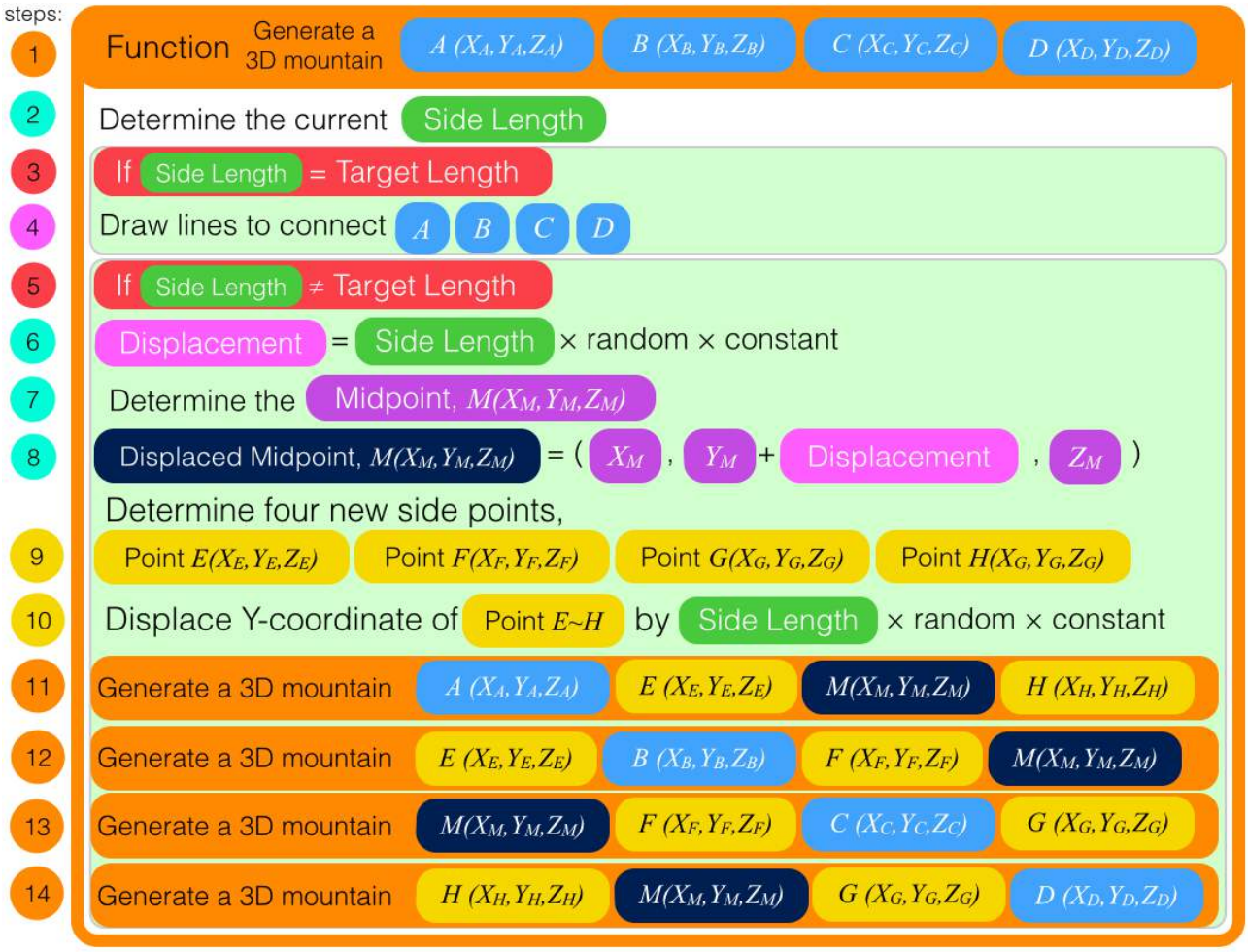
Figure 17 shows the progression of a fractal mountain similar to figure 11, but using the diamond square algorithm. In this algorithm, there are two steps in each iteration, the diamond step and the square step, represented by the red and blue dots respectively(Martz). Without the blue dots, this algorithm is identical to to the midpoint displacement algorithm. After displacing the midpoint of the square in the vertical direction, the four midpoints of each side of the square are determined. Finally, all the four points are displaced along the vertical direction as well(Martz).

Although it is a simple step, the diamond square algorithm solves the problem seen in the midpoint displacement algorithm. Since the side midpoints also get displaced, the straight line in figure 16 will no longer be one straight line when the number of iterations are increased.

4e: Producing a 3D mountain using the Diamond Square Algorithm(Part 1)

To change the midpoint displacement computer program into the diamond square algorithm does not require a significant change in the program.

Fig. 18.



The only difference between figure 18 and figure 12 is the addition of step 10. With this step, each of the midpoint of the four side points, Y_E, Y_F, Y_G, Y_G, are displaced in the same way as point M, the central midpoint.

While the problem with the midpoint displacement algorithm is fixed, another problem arises with this algorithm.

Fig. 19

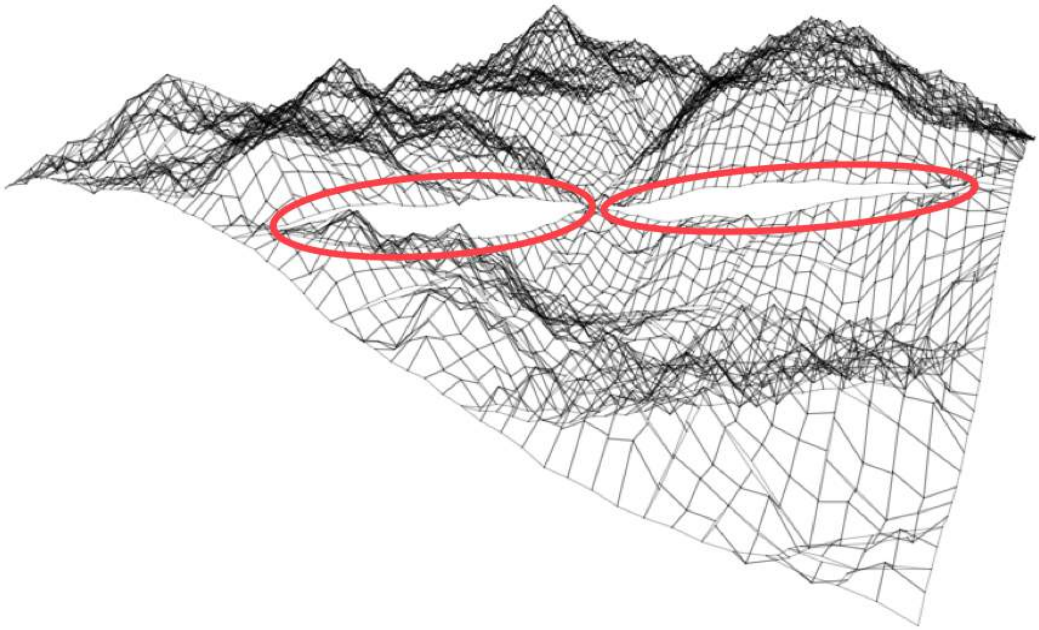


Figure 19 is a mountain created by this algorithm. There are obvious creases or gaps in the mountain marked in red. Unlike the problem with the midpoint displacement algorithm, this is not a problem of the diamond square algorithm, and is fixable. This problem is caused by the use of the random variable.

Fig. 20.

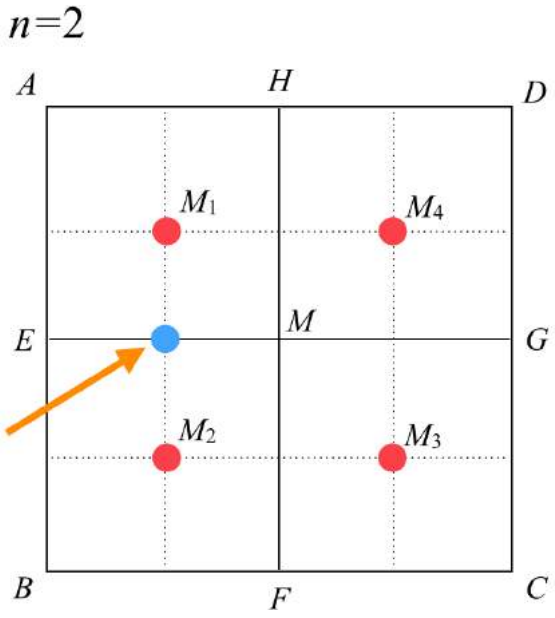


Figure 20 shows one section of the second iteration. When the program is running, it is applying the diamond square algorithm to all of the four quadrilaterals, *AEMH, EBFM, MFCG, HMGD*, independently. For example, when the computer is working with square *EBFM*, it does not know any transformations that were done to any of the other squares. Each square is transformed irrespective of how the other squares have changed.

Consider the displacement of the midpoint of *EM*. When the program is dealing with square *AEMH*, it will displace this point dependent on a random variable. When the program is dealing with square *EBFM*, it will displace this point by some other random value. Here, the two random values will most likely not equal each other. In extreme cases such as figure 19, the large difference in the random value displaces the same point into two opposing directions, resulting in a such a crease.

4f: Producing a 3D mountain using the Diamond Square Algorithm(Part 2)

To fix this problem, we must develop a system, so each quadrilateral is not being transformed independently. A solution is to introduce a matrix containing a specific random value for each point created in the process of iteration. This way, each point on the mountain will have only one random value assigned. Thus, there will be no disagreements in how a point will be displaced.

The size of the matrix should correspond to the number of points at the n^{th} iteration. Since number of line segments of one side of the original square at the n^{th} iteration is 2^n lines, the number of points on one side is 2^n+1 points. Therefore, the matrix will have a size of 2^n+1 by 2^n+1.

In the program, the size of the matrix must be defined with an actual number. The time taken to execute the program in javascript increases exponentially as n is increased. With a normal computer, any number of iterations above $n=11$ would cause the program to fail. Thus, in the program, the size of the matrix is set to be $2^{11}+1$ by $2^{11}+1$.

Now the matrix is created, each space in the matrix contains a random number, which corresponds to a specific point of the mountain. When the recursive process is taking place, the program will know the position of the point it is trying to displace, on the mountain, and will fetch the random value from the matrix corresponding to that point.

Fig. 21.

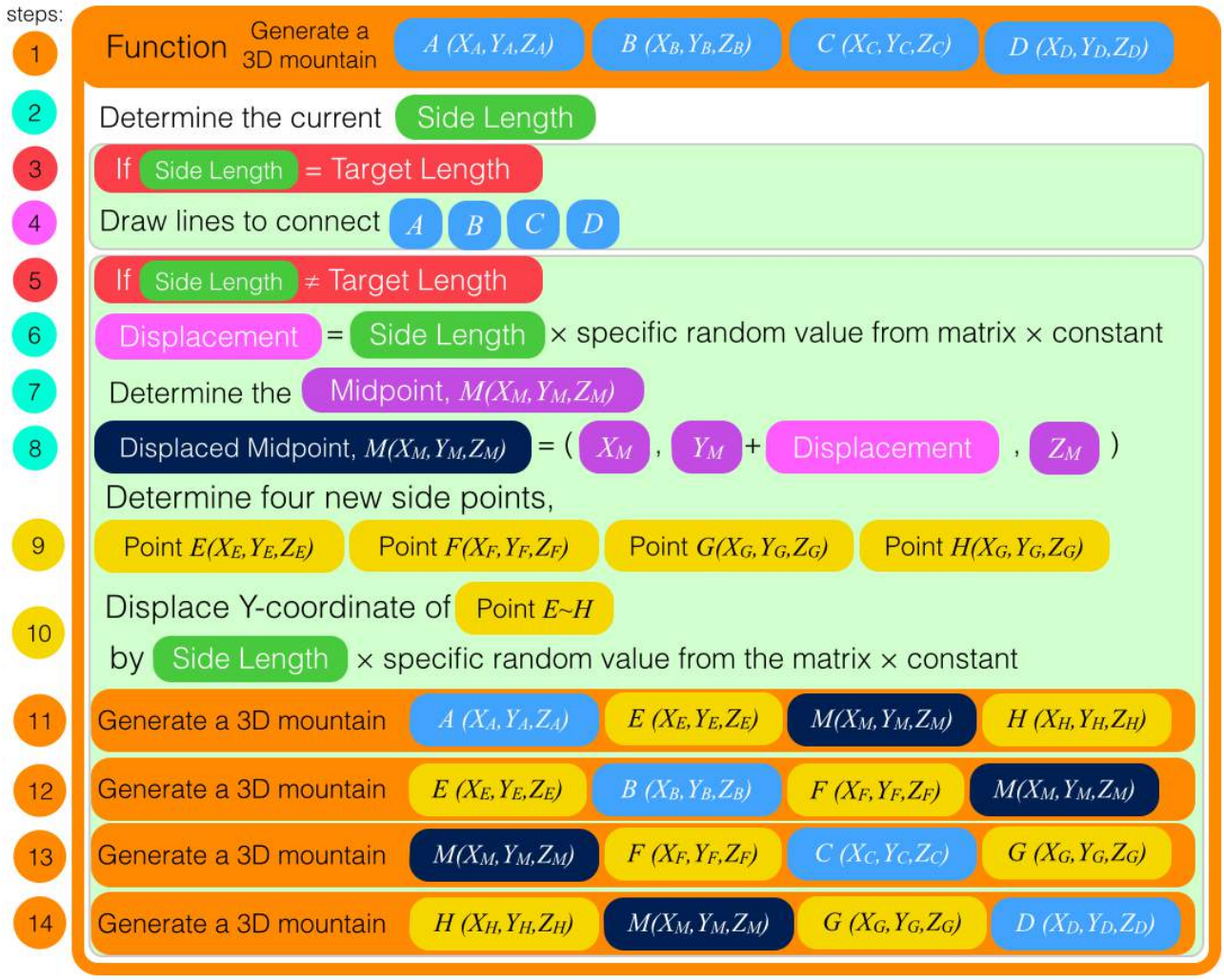
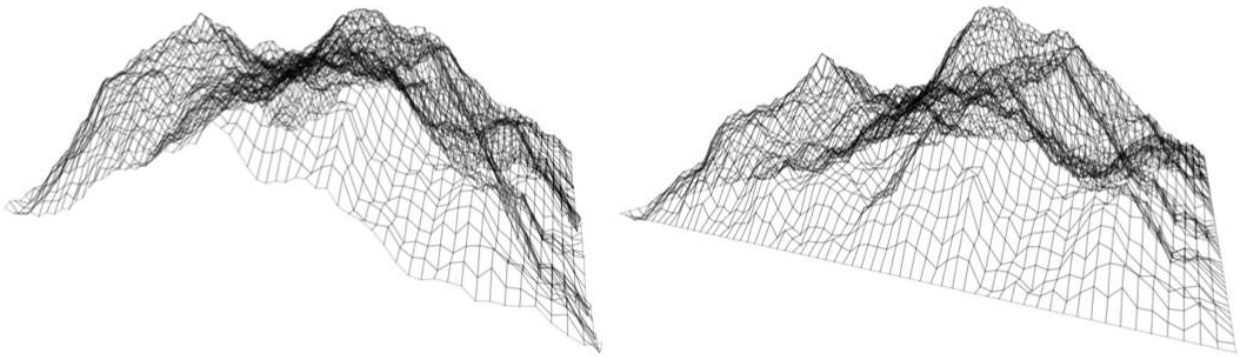


Figure 21 shows the structure of the modified program. The differences from the previous program can be seen in steps 6 and 10.

Fig. 22.



Both of these mountains shown in figure 22 are made from the same matrix of random numbers and are iterated the same number of times. The mountain on the left is purely made from the diamond

square algorithm. The one on the right is adjusted to look cleaner on the edge by eliminating any displacement for the points on the edge of the original square.

4g: Shading the mountain

The final step in producing a 3D mountain is shading all the subdivided spaces. Before we begin the shading process, we have to divide the quadrilaterals into triangles(Hughes). A triangle, having three points, will always be on one plane(Hughes). If we leave the subdivided spaces as quadrilaterals, in many cases we will end up shading two planes with one shade.

Fig. 23.

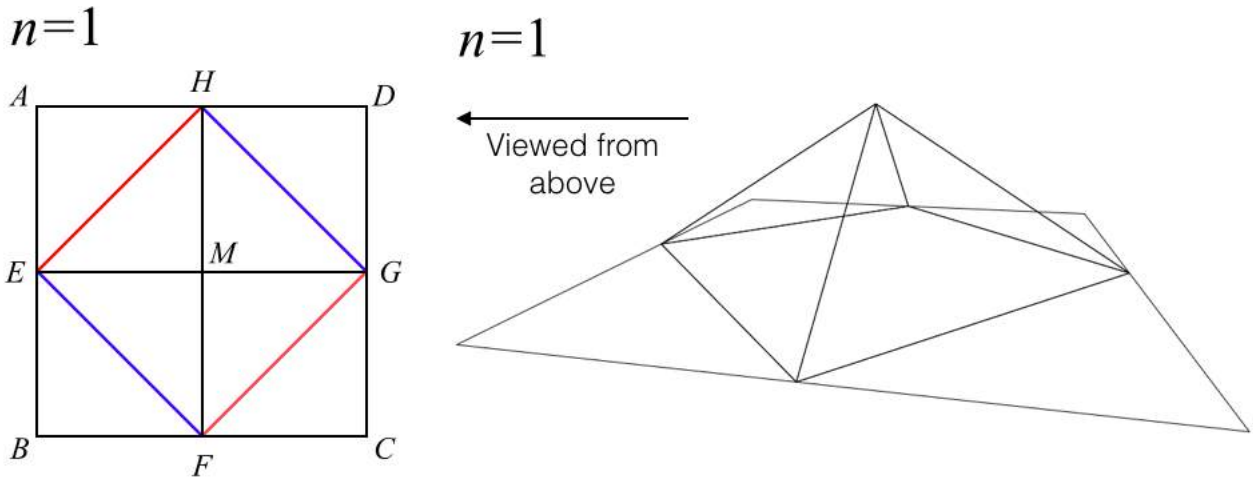


Figure 23 shows how the squares will be subdivided into triangles for n=1. The same method will be applied for dividing quadrilaterals in higher iterations. For the diagram on the left, there are two ways the squares are divided, red and blue. Instead of dividing the squares one way(eg: from the top left corner to the bottom right corner), this method makes the mountain symmetrical.

The brightness of each triangle depends on the amount of sunlight hitting on its surface. To do this, we must first visualize the sun's rays as vectors. Since the distance between the sun and the mountain is so large, we can assume all the vectors representing the sun's rays to be the same.

The brightness would be proportional to the magnitude of the component of the sun's vector which is perpendicular to the surface of the triangle.

Fig. 24

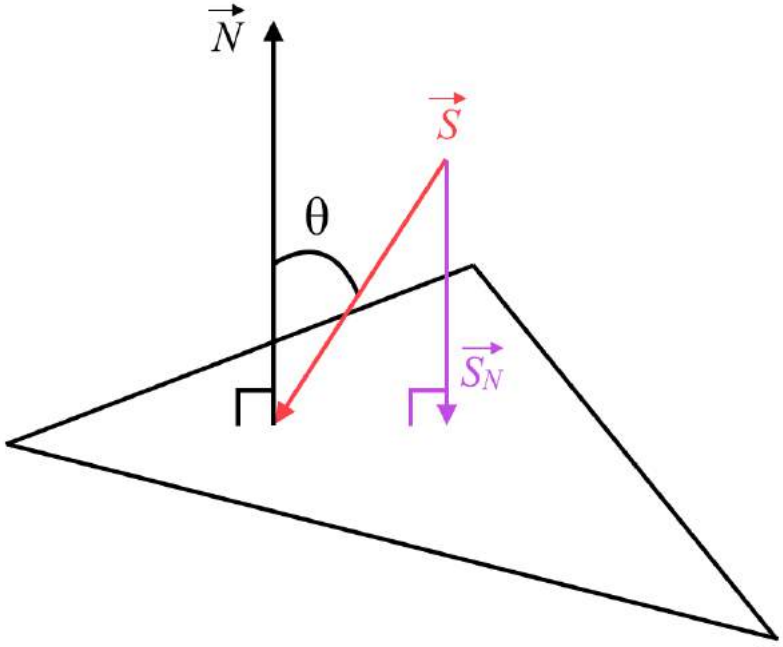


Figure 24 shows the situation for one triangle, where vector N is the normal vector to the triangle, and vector S is the sun's rays. We must determine the magnitude of vector S_N.

$$|\vec{S}_N|=|\vec{S}|\cos\theta$$

To calculate $\cos\theta$ we need to determine the normal vector N. For each divided triangle shown in figure 23, we can calculate the vector of two of the sides, and take the cross product to determine the normal vector.

Fig. 25.

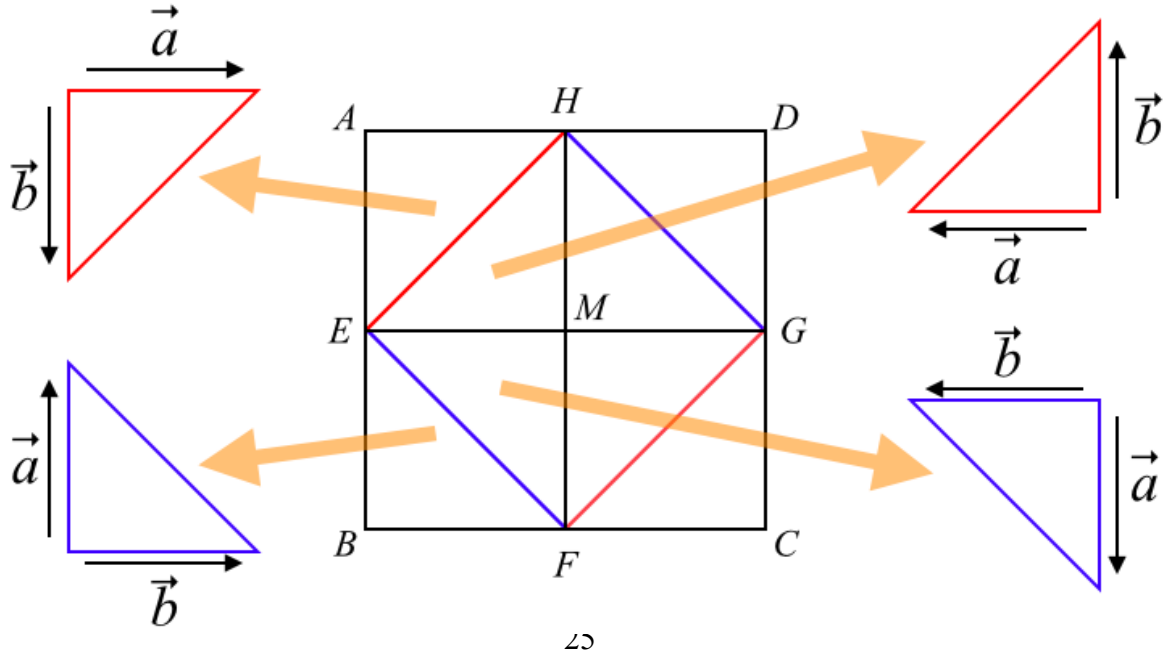


Figure 25 shows how the two vectors are selected on each triangle. From the vector right hand rule, all the normal vectors are going into the page. If vector *a* and vector *b* on each triangle is selected by random, not all the normal vectors will be going into the page, which would cause problems when shading the triangles.

The normal vector is,

$$if\ \vec{a}=\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}\quad \text{and}\quad \vec{b}=\begin{pmatrix}b_x\\b_y\\b_z\end{pmatrix}$$

$$\vec{a}\times\vec{b}=\begin{pmatrix}a_y b_z-a_z b_y\\a_z b_x-a_x b_z\\a_x b_y-a_y b_x\end{pmatrix}=\vec{N}=\begin{pmatrix}N_x\\N_y\\N_z\end{pmatrix}$$

Now, we can find $cos\theta$ by taking the dot product of vector S and vector *N*.

$$\vec{S}=\begin{pmatrix}S_x\\S_y\\S_z\end{pmatrix}$$

$$\vec{S}\cdot\vec{N}=S_x N_x+S_y N_y+S_z N_z=|\vec{S}||\vec{N}|cos\theta$$

$$\frac{S_x N_x+S_y N_y+S_z N_z}{|\vec{S}||\vec{N}|}=cos\theta$$

Since,

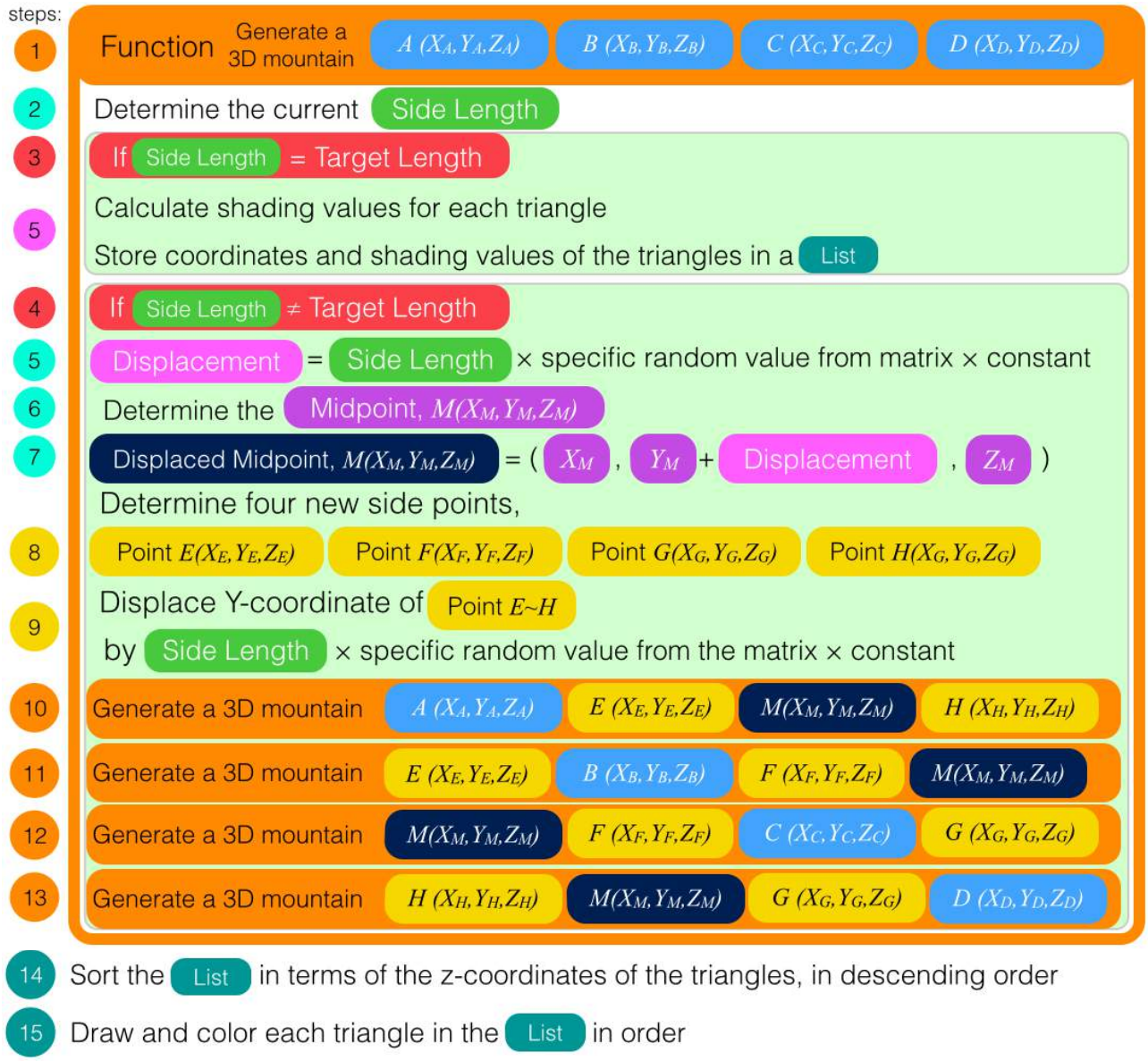
$$Brightness\propto|\vec{S}_N|=|\vec{S}|cos\theta$$

$$Brightness\propto|\vec{S}_N|=\frac{S_x N_x+S_y N_y+S_z N_z}{|\vec{N}|}$$

$$Brightness\propto\frac{S_x N_x+S_y N_y+S_z N_z}{|\vec{N}|}$$

Now in the program, a constant(which is adjusted) can be multiplied to $\frac{S_x N_x+S_y N_y+S_z N_z}{|\vec{N}|}$, to produce a value that can be directly used to represent the of brightness of each triangle.

Fig. 26



In terms of the computer program, we can incorporate these vector calculations in step 5 of the program, as shown in figure 26. In addition to the calculations, now the triangles are not being drawn in step 5, but the information(such as the coordinates and brightness) is stored in a list.

A list is used to overcome a problem when drawing these triangles. If a triangle further away from the screen is drawn after a triangle closer to the screen is drawn, the mountain will look chaotic, because seeing an object, which should be furthest away, in front of an object which is closer to the viewer does not make sense. To correctly draw the mountain, we must draw each triangle in descending order of of the *z*-coordinate, meaning the triangles will be drawn from back to front, relative to the viewer(Riley “The Visibility Problem - Computerphile”). With a sorting function in javascript, we can rearrange the information of each triangle in the list in decreasing order of the *z*-

coordinate. Now, the program can draw the triangles through the list from beginning to end, without causing any chaos.

If we set a green hue for the base color, we get a colored mountain with correct shading(see figure 27). We can also change the color of each triangle based on its y-coordinate and create a mountain with a height map(see figure 28).

Fig. 27.

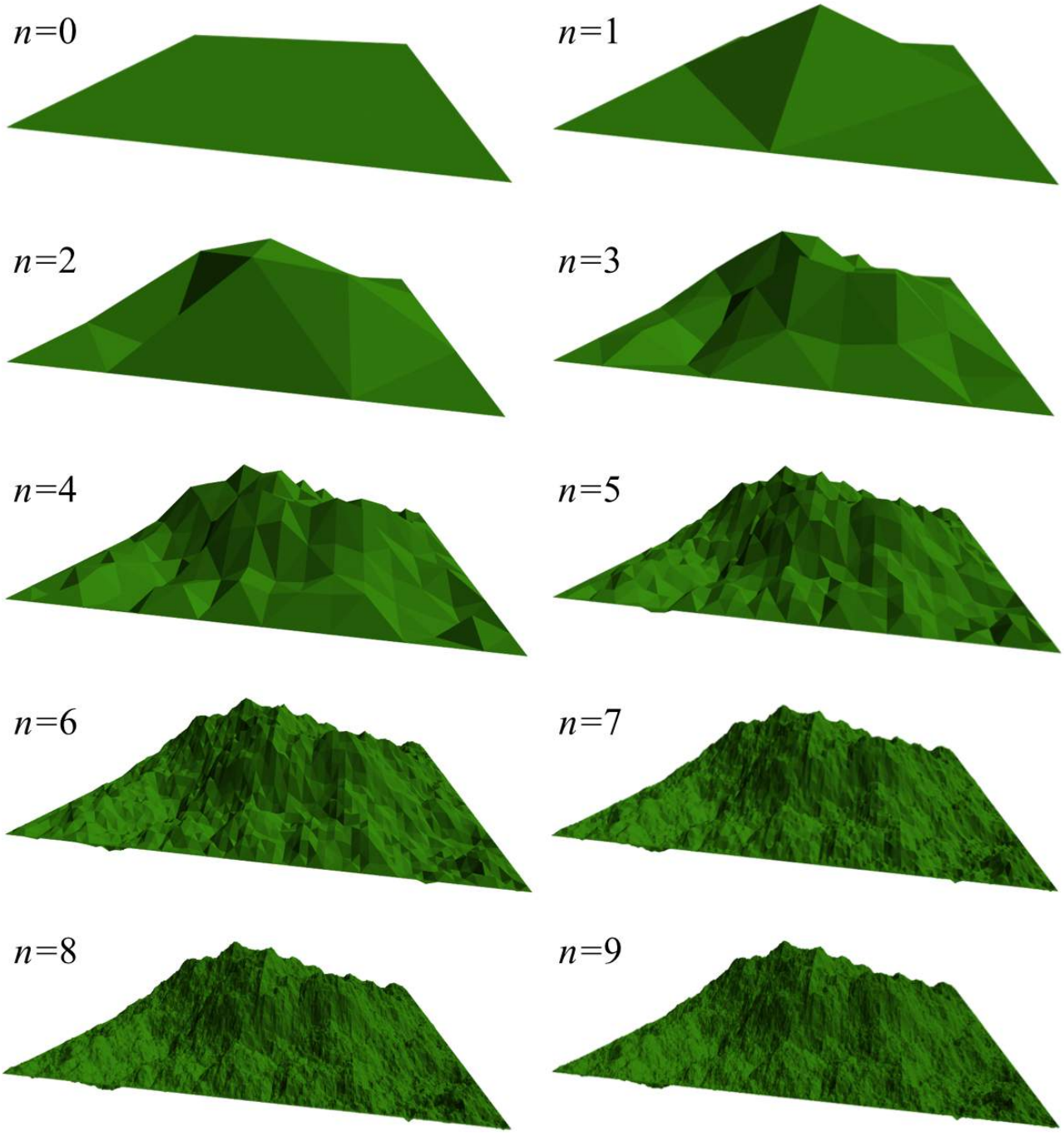
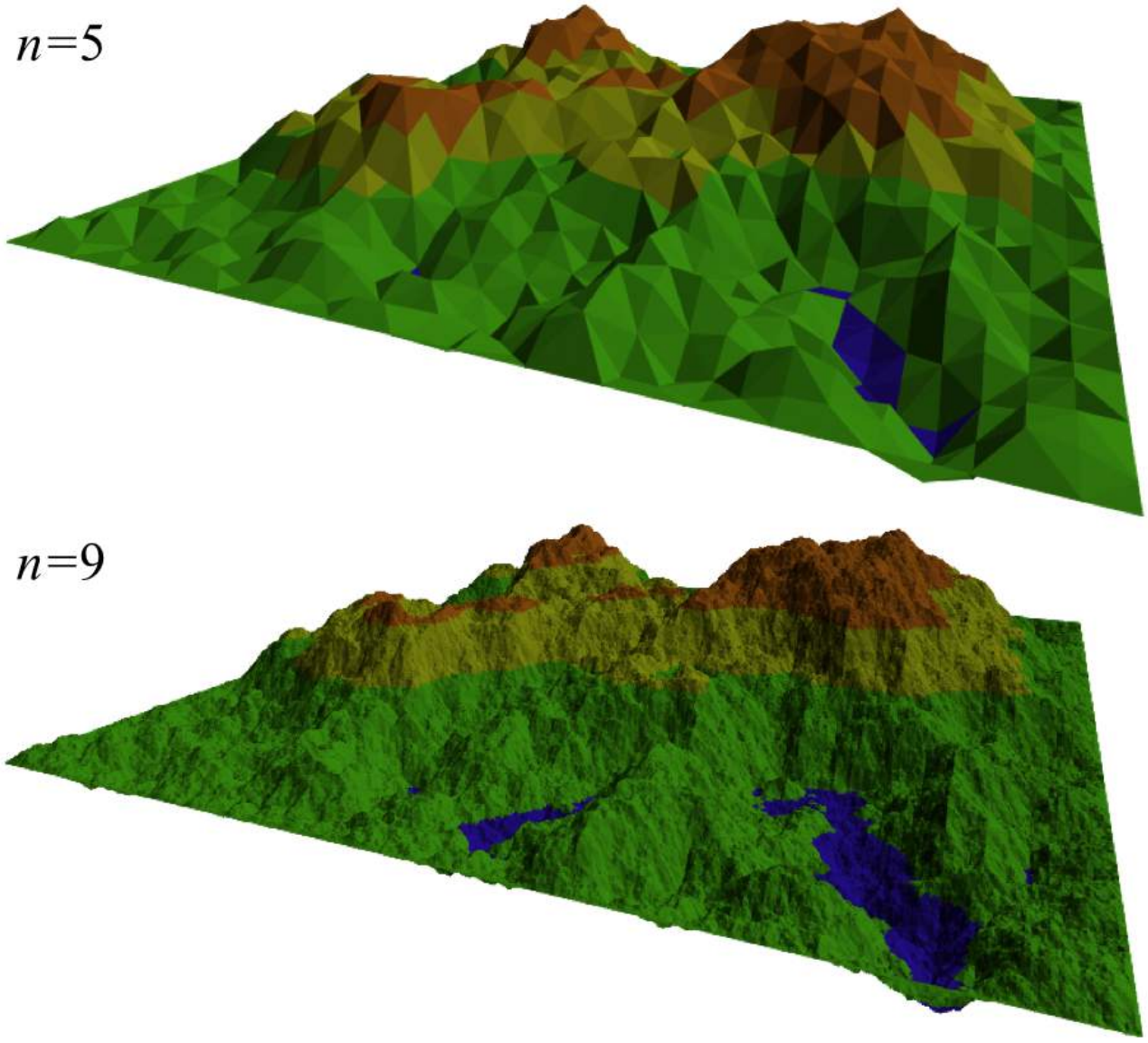


Fig. 28.



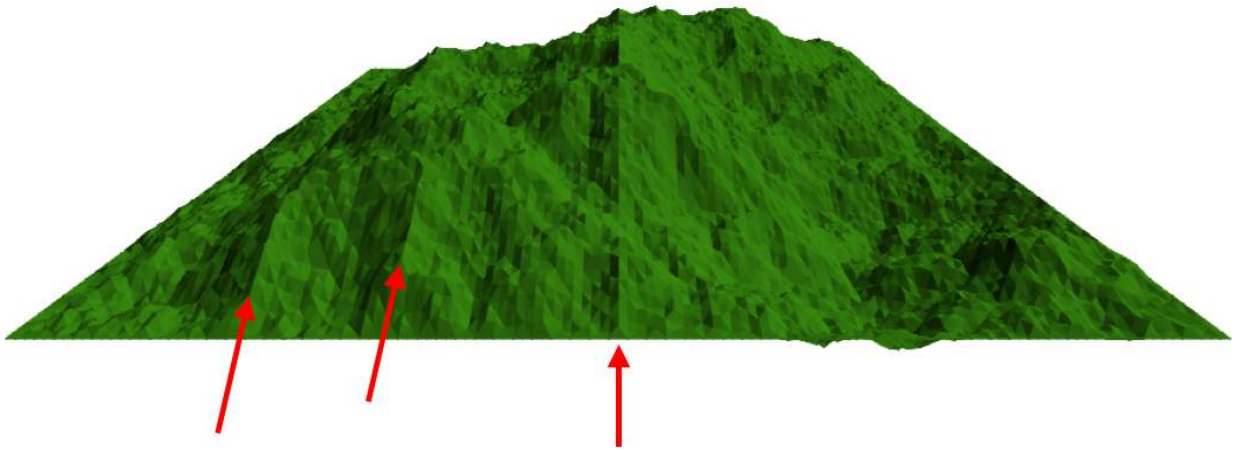
Part 5 : Conclusion

By using the diamond square algorithm and 3D visualization systems a 3D mountain was successfully produced. The mountain looks the most realistic with a larger iteration value. For example in figure 28 the mountain at n=9 will have $2\times4^9 = 524288$ triangles. A large number of triangles results in a complex shape which are similar to complex shapes produced in nature. The objective of this investigation was to explore algorithms to produce a fractal mountain which look as realistic as possible. This objective was met to a certain extent as the mountain in figure 28 strongly resembles a mountain range seen in nature.

However, this fractal mountain does have flaws which deviates from a real mountain. The first limitation is roughness. Although a huge number of triangles are being drawn, they are still triangles. In nature, there are rough terrains such as the one created, but many surfaces can be much more smooth, and will consist of different geometric shapes than triangles.

The second limitation is the mountain having straight lines, similar to the problem encountered with the midpoint displacement algorithm.

Fig. 29.



This algorithm only displaces points in the vertical direction. Therefore, when the mountain is viewed from the front, in some cases straight lines can be seen(see figure 29), which makes the mountain look artificial. One solution to this problem is to begin the recursive subdivision process with an equilateral triangle, rather than a square. We can divide an equilateral triangle into four smaller equilateral triangles in each iteration, similar to the

Sierpinski Triangle. This method may be able to eliminate multiple straight lines seen in figure 29 because subdividing an equilateral triangle will not result in multiple parallel straight lines going across the shape, which is the case with subdividing a square.

The third limitation is the case of shading a triangle with a wrong color.

Fig. 30.

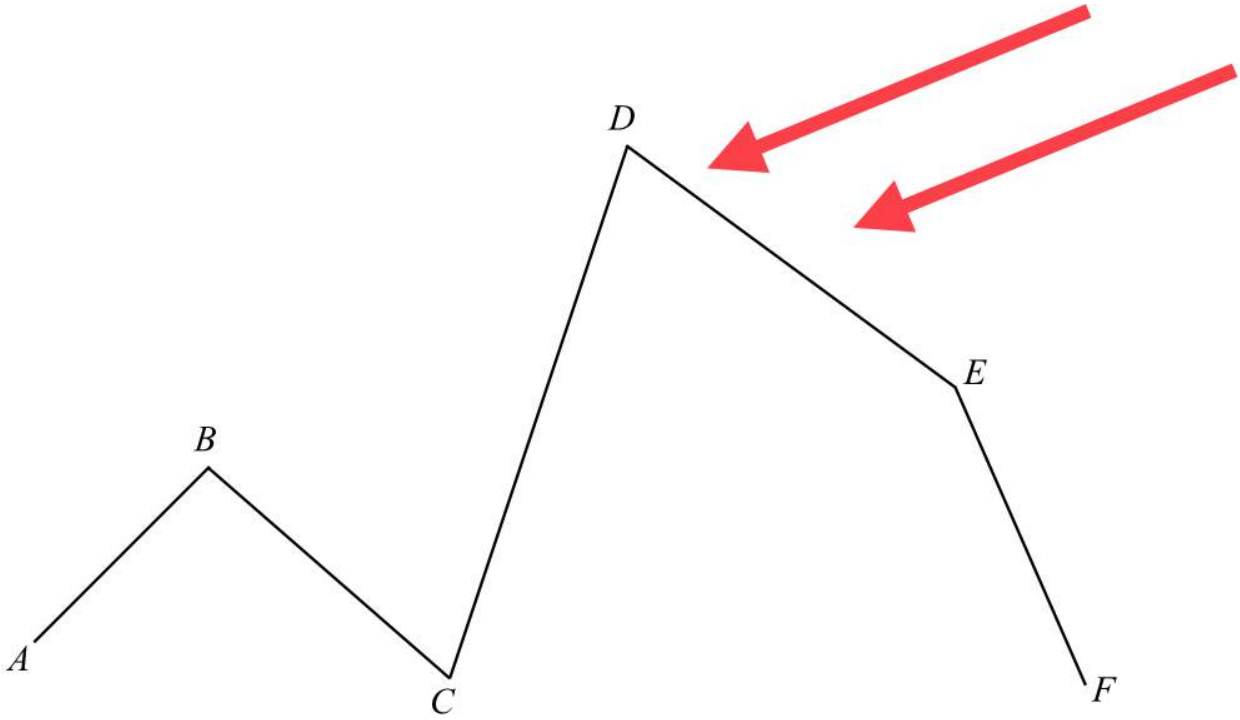


Figure 30 shows one possible terrain, and the arrows represent the direction of the sun's rays. In reality, the surfacing containing segment *BC* would be in the shade because sunlight is blocked by segments *DE* and *EF*. However, in the algorithm, each surface is independent of the others. Thus the surface with segment *BC* which is almost perpendicular to the sun's rays will be shaded with a bright color, which is unrealistic.

Work Cited

1. "Fractal Geometry." *IBM 100*, http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/fractal/. Accessed 30 September 2015.

2. "Fractals: Useful Beauty." *Fractal.org*, http://www.fractal.org/Bewustzijns-Besturings-Model/Fractals-Useful-Beauty.htm. Accessed 20 October 2016.

3. Fournier, Alain. Fussell, Don. Carpenter, Loren. "Computer Rendering of Stochastic Models" *Communications of the ACM*, vol. 25, no. 6, 19Fournier82, pp. 371-384, http://excelsior.biosci.ohio-state.edu/~carlson/history/PDFs/p371-fournier.pdf. Accessed 20 February 2016.

4. Germain, Jim. "Recursion." *The University of Utah School of Computing*, http://www.cs.utah.edu/~germain/PPS/Topics/recursion.html. Accessed 10 February 2016.

5. Riley, Sean. "What on Earth is Recursion? - Computerphile" *YouTube*, uploaded by Computerphile, 16 May 2014, https://www.youtube.com/watch?v=Mv9NEXX1VHc.

6. Hughes, Merlin. "3D Graphic Java: Render fractal landscapes." *JavaWorld*, 1 August 1998, http://www.javaworld.com/article/2076745/learn-java/3d-graphic-java--render-fractal-landscapes.html. Accessed 22 March 2016.

7. Mandelbrot, Benoit B. *The Fractal Geometry of Nature*. W. H. Freeman and Co., 1982

8. Martz, Paul. "Generating Random Fractal Terrain" *Game Programmer*, http://www.gameprogrammer.com/fractal.html, Accessed 14 February 2016.

9. Miller, Gavin S. P. "The Definition and Rendering of Terrain Maps." *ACM SIGGRAPH Computer Graphics*, vol. 20, no. 4, 1986, pp. 39-48, http://metalbyexample.com/wp-content/uploads/miller-terrain.pdf. Accessed 20 February 2016.

10. Riley, Sean. “The Visibility Problem - Computerphile.” *YouTube*, uploaded by Computerphile, 3 January 2014. https://www.youtube.com/watch?v=OODzTMcGDD0&list=PLzH6n4zXuckrPkEUK5iMQrQyvj9Z6WCrm&index=4. Accessed 26 June 2016.

Note: Figures 4 ~ 30 are created by Kai Junge

Appendix 1

This is the program in javascript used to create the 2D mountain.

```
<html>
    <h1>2D terrain generator</h1>

  <input type="search" id="input" onsearch="proceed_fractal()">
   <p> enter a value between 0 and 10 </p>

<body>
<canvas id="myCanvas" width="1025" height="600" style="border:1px solid
#000000;">
    //style=background-color
</canvas>

<script>
var canvas = document.getElementById("myCanvas"); //load the canvas
var ctx = canvas.getContext("2d"); // make the canvas ediable

    var size = Math.pow(2,10)+1;
    var left_point = [1,100];
    var right_point = [size,100];
    var iteration;


    function terrain(left,right,N,color)
    {
          if(N == iteration)
           {
            // draw a line from the left to the right
           ctx.beginPath();
           ctx.moveTo(left[0],left[1]);
           ctx.lineTo(right[0],right[1]);
           ctx.strokeStyle = color;
           ctx.stroke();
           ctx.closePath();

            }
    else{

    var displacement = (right[0]-
left[0])*(Math.cos(Math.random()*Math.PI))*0.2;
    var midpoint = [(left[0]+right[0])/2,
((left[1]+right[1])/2)+displacement];

    terrain(left,midpoint,N+1,color);
    terrain(midpoint,right,N+1,color);
    }

    }
```

```
    function proceed_fractal()
    {
        ctx.clearRect(0,0,innerWidth,innerHeight);

        iteration = document.getElementById("input").value;

        terrain([1,left_point[1]+150],
[size,right_point[1]+150],0,"#000000");
    }

</script>
</body>
</html>
```

Appendix 2

This is the program in javascript used to create the final version of the 3D mountain using the diamond square algorithm.

```
<html>
<h1>3D Mountain</h1>
 <input id="refresh" type="button" value="refresh" onclick="refresh()">
press "refresh" to produce a new mountain

<p>number of iteration <input type="search" id="input"
onsearch="update_iteration()">  enter a value between and including 0 and
9(9 will take 50 seconds to load) </p>
   Effects:
    <input  type="button" value="rotate left" onclick="left()">
    <input  type="button" value="rotate right" onclick="right()">

    <input  type="button" value="black line" onclick="show_line()">
    <input  type="button" value="colored line" onclick="hide_line()">

    <input  type="button" value="show height color" onclick="show_color()">
    <input  type="button" value="show only green" onclick="only_green()">
    <input  type="button" value="hide color" onclick="hide_color()">

    <input  type="button" value="+roughness" onclick="plus_roughness()">
    <input  type="button" value="-roughness" onclick="minus_roughness()">

    <p id ="one"></p>  <p id ="two"></p> <p id="three"></p>

<canvas id="myCanvas" width="1000" height="600" style="border:1px solid
#000000;">
</canvas>

    <script>
        var canvas = document.getElementById("myCanvas");// load the canvas
        var ctx = canvas.getContext("2d"); //make the canas editable

        var camera = [500,100,0];
        var screen = 600;

        var max_shade = 30;  //40
        var min_shade = -10;   //5
        var original_Hue = 123;
        var Saturation_index = 80;
        var Saturation = 80;
        var Saturation_line = 80;
        var FILL = 1;
        var store = 0;
        var size = 0;
        var n = 0;
```

```
        var theta = Math.PI/4; //angular distance from x/z axis to any
hypotenuse, clockwise
        var smoothness = 0.08;
        var side_roughness = 0.15;
        var center_roughness = 0.2;
        var angular_displacement = 0;
        var increment = 0.125*0.5*Math.PI;
        var height_setting = 110;
        var GREEN = 1;
        var max_iteration = 9;

        var draw_data = []; //array to store drawing data to draw in the
end.

        var background_color = "#ffffff";

        document.getElementById("one").innerHTML= "angular distance:  " +
Math.abs(angular_displacement) + " degrees";

        function refresh()
        {
            function create_storage_array()
            {
                var array = [];
                for(i=0;i<Math.pow(2,max_iteration)+1;i++)
                    {
                            array[i] = new Array(Math.pow(2,max_iteration)
+1);
                        for(j=0;j<Math.pow(2,max_iteration)+1;j++)
                            {
                                array[i][j] = new Array(3);
                                array[i][j][0]=
(Math.random()).valueOf(); //x coordinate
                                array[i][j][1]= i; //x of array coordinate
                                array[i][j][2]= j; //y of array coordinate
                            }
                    }
                return array;
            }
            store = create_storage_array();
            proceed_fractal();
        }

        function left()
        {
            if(theta != 2*Math.PI+Math.PI/4)
            {
            angular_displacement += 11.25;
            }
            theta += increment;

            if(theta<2*Math.PI+Math.PI/4)
                {
```

```
            proceed_fractal()
        }
    else
        {
          theta =  2*Math.PI+Math.PI/4;
          proceed_fractal()
        }
}

function right()
{
    if(theta != Math.PI/4)
    {
        angular_displacement -= 11.25;
    }
    theta -= increment;

    if(theta>Math.PI/4)
        {
            proceed_fractal()
        }
    else
        {
          theta =  Math.PI/4;
          proceed_fractal()
        }
}

function show_line()
{
    Saturation_line = 0;
    proceed_fractal();
}
function hide_line()
{
    Saturation_line = Saturation_index;
    proceed_fractal();
}
function show_color()
{
    FILL = 1;
    GREEN = 0;
    proceed_fractal();
}
function hide_color()
{
    FILL = 0;
    proceed_fractal();
}
function only_green()
{
    FILL = 1;
    GREEN = 1;
```

```
            proceed_fractal();
        }
        function plus_roughness()
        {
            center_roughness += 0.05;
            side_roughness += 0.05;
            proceed_fractal();
        }

        function minus_roughness()
        {
            center_roughness -= 0.05;
            side_roughness -= 0.05;
            if(center_roughness <= 0)
            {
                center_roughness = 0.000001;
            }
            if(side_roughness <= 0)
            {
                side_roughness = 0.000001;
            }
            proceed_fractal();
        }

        function update_iteration()
        {
            var iteration = document.getElementById("input").value;

            if(iteration <= max_iteration)
            {
                n = iteration;
            }

            size = Math.pow(2,n)+1;
            proceed_fractal()
        }

        function proceed_fractal()
        {
            draw_data = [];
            var Hue = 100;

            document.getElementById("one").innerHTML= "angular distance:  "
+ Math.abs(angular_displacement) + " degrees";

            var sun = [-5,5,0]; //vector of the sun ray
            var sun_size =
Math.sqrt(Math.pow(sun[0],2)+Math.pow(sun[1],2)+Math.pow(sun[2],2));
//vector size of sun
            var original_Hue = 123;
            var N = 11;  //side length of the mountain = 2^N
            var center = [500,850,2400]; //center point of the mountain
```

```
            var hyp = Math.pow(2,N)/Math.sqrt(2); //distance from center to
a point of the mountain

            ctx.clearRect(0,0,innerWidth,innerHeight); //clear screen
            var phi = (Math.PI/2)-theta;

            function origin(point) //create the coordiantes for the four
centers for a given center point
            {
                var result = [[point[0]-
hyp*Math.cos(theta),point[1],point[2]+hyp*Math.sin(theta)],[point[0]-
hyp*Math.cos(phi),point[1],point[2]-hyp*Math.sin(phi)],
[point[0]+hyp*Math.cos(theta),point[1],point[2]-hyp*Math.sin(theta)],
[point[0]+hyp*Math.cos(phi),point[1],point[2]+hyp*Math.sin(phi)]];
                return result;
            }

            var sq1 = origin(center);  //generate the four corners for one
mountain

            function projection(Point) //convert 3D points to 2D points
            {
                var final = new Array(2);

                final[0]=camera[0]-(camera[0]-Point[0])*(screen/(Point[2]-
camera[2]));
                final[1]=camera[1]-(camera[1]-Point[1])*(screen/(Point[2]-
camera[2]));

                return final;
            }

            function RNG(random,side)  //create the displacement for the
side length, not the center
            {
                // "random" is the random value obtained from the common
array
                return side*(Math.cos(random*Math.PI))*side_roughness;
            }

            // set background color
            ctx.beginPath();
            ctx.lineTo(0,0);
            ctx.lineTo(1200,0);
            ctx.lineTo(1200,600);
            ctx.lineTo(0,600);
            ctx.closePath();
            ctx.fillStyle=background_color;
            ctx.fill();


            function setcolor(height)
            {
```

```
                var base = 850;
                var hue_setting;
                if(GREEN == 0)
                {
                    if(height>=base+50)
                    {
                        hue_setting = 250;
                    }
                    if(height<=base+50&&height>base-height_setting)
                    {
                        hue_setting = 100;
                    }
                    if(height<=base-height_setting&&height>base-
height_setting*2)
                    {
                        hue_setting = 65;
                    }
                    if(height<=base-height_setting*2)
                    {
                        hue_setting = 30;
                    }
                }
                else
                {
                    hue_setting = 100;
                }

                return hue_setting;
            }

        function terrain(P1,P2,P3,P4,p1,p2,orientation) // terrain creation
        {
            //variable orientation is to determine whether the square is in
position 1,3 or 2,4
            //this position will determine how to draw triangle 1 and
triangle 2

            var side_length = Math.round(Math.sqrt(Math.pow((P1[0]-
P2[0]),2)+Math.pow((P1[2]-P2[2]),2)));
            var half_length = (p2[2]-p1[2])/2; // the half length in the
array

            if(side_length == Math.pow(2,N)/Math.pow(2,n)) // if length =
target length
            {
                /*
                DEFINITION:

                triangle 1 is made from points P1,P2,P4

                triangle 2 is made from points P2,P3,P4
                */
```

```
                if(orientation == 0)
                    {
                       var a1 = [P4[0]-P1[0],P4[1]-P1[1],P4[2]-P1[2]];
                       var b1 = [P2[0]-P1[0],P2[1]-P1[1],P2[2]-P1[2]];
                       var a2 = [P3[0]-P2[0],P3[1]-P2[1],P3[2]-P2[2]];
                       var b2 = [P3[0]-P4[0],P3[1]-P4[1],P3[2]-P4[2]];
                    }
                else
                    {
                       var a1 = [P1[0]-P2[0],P1[1]-P2[1],P1[2]-P2[2]];
                       var b1 = [P3[0]-P2[0],P3[1]-P2[1],P3[2]-P2[2]];
                       var a2 = [P4[0]-P3[0],P4[1]-P3[1],P4[2]-P3[2]];
                       var b2 = [P4[0]-P1[0],P4[1]-P1[1],P4[2]-P1[2]];
                    }


                var normal1 = [a1[1]*b1[2]-a1[2]*b1[1],a1[2]*b1[0]-
a1[0]*b1[2],a1[0]*b1[1]-a1[1]*b1[0]]; //cross product of a1 x b1
                var normal2 = [a2[1]*b2[2]-a2[2]*b2[1],a2[2]*b2[0]-
a2[0]*b2[2],a2[0]*b2[1]-a2[1]*b2[0]]; //cross product of a2 x b2

                var normal_size1 =
Math.sqrt(Math.pow(normal1[0],2)+Math.pow(normal1[1],2)+Math.pow(normal1[2],
2));
                var normal_size2 =
Math.sqrt(Math.pow(normal2[0],2)+Math.pow(normal2[1],2)+Math.pow(normal2[2],
2));

                var dot1 =
normal1[0]*sun[0]+normal1[1]*sun[1]+normal1[2]*sun[2]; //dot product of
normal1 and sun
                var dot2 =
normal2[0]*sun[0]+normal2[1]*sun[1]+normal2[2]*sun[2]; //dot product of
normal2 and sun

                var cosine1 = dot1/(normal_size1*sun_size);  //find cosine
of normal1 and sun
                var cosine2 = dot2/(normal_size2*sun_size);  //find cosine
of normal2 and sun

                var amplitude = (max_shade-min_shade)/2;

                var color1 = Math.round(amplitude*cosine1+
(amplitude+min_shade));
                var color2 = Math.round(amplitude*cosine2+
(amplitude+min_shade));


                var z_value;
                var hue_color;
                if(orientation == 0)
                    {
                        z_value = (P1[2]+P2[2]+P4[2])/3;
```

```
                        hue_color =
setcolor(Math.min(Math.min(P1[1],P2[1]),P4[1]));

draw_data.push([z_value,projection(P1),projection(P2),projection(P4),color1,
hue_color]);

                        z_value = (P2[2]+P4[2]+P3[2])/3;
                        hue_color =
setcolor(Math.min(Math.min(P2[1],P4[1]),P3[1]));

draw_data.push([z_value,projection(P2),projection(P4),projection(P3),color2,
hue_color]);
                    }
                 else
                    {
                        z_value = (P1[2]+P2[2]+P3[2])/3;
                        hue_color =
setcolor(Math.min(Math.min(P1[1],P2[1]),P3[1]));

draw_data.push([z_value,projection(P1),projection(P2),projection(P3),color1,
hue_color]);

                        z_value = (P1[2]+P3[2]+P4[2])/3;
                        hue_color =
setcolor(Math.min(Math.min(P1[1],P3[1]),P4[1]));

draw_data.push([z_value,projection(P1),projection(P3),projection(P4),color2,
hue_color]);
                    }
            }
            else
            {
               var displacement =
side_length*(Math.cos(store[p1[1]+half_length][p1[2]+half_length]
[0]*Math.PI))*center_roughness;

               var C = [(P1[0]+P2[0]+P3[0]+P4[0])/4,
((P1[1]+P2[1]+P3[1]+P4[1])/4)+displacement,(P1[2]+P2[2]+P3[2]+P4[2])/4];

               var M1 = [(P1[0]+P2[0])/2,((P1[1]+P2[1])/2)+RNG(store[p1[1]]
[p1[2]+half_length][0],side_length),(P1[2]+P2[2])/2];

                if(p1[1]==0)
                    {
                        M1[1] = (P1[1]+P2[1])/2;
                    }

               var M2 = [(P2[0]+P3[0])/2,
((P2[1]+P3[1])/2)+RNG(store[p1[1]+half_length][p2[2]][0],side_length),
(P2[2]+P3[2])/2];


                if(p2[2]==Math.pow(2,max_iteration))
```

```
                    {
                        M2[1] = (P2[1]+P3[1])/2;
                    }

                var M3 = [(P3[0]+P4[0])/2,
((P3[1]+P4[1])/2)+RNG(store[p1[1]+half_length*2][p1[2]+half_length]
[0],side_length),(P3[2]+P4[2])/2];

                if(p1[1]+half_length*2==Math.pow(2,max_iteration))
                    {
                        M3[1] = (P3[1]+P4[1])/2;
                    }

                var M4 = [(P4[0]+P1[0])/2,
((P4[1]+P1[1])/2)+RNG(store[p1[1]+half_length][p1[2]][0],side_length),
(P4[2]+P1[2])/2];

                if(p1[2]==0)
                    {
                        M4[1] = (P4[1]+P1[1])/2;
                    }


                 terrain(P1,M1,C,M4,p1,store[p1[1]][p1[2]+half_length],0);
                 terrain(M1,P2,M2,C,store[p1[1]][p1[2]+half_length],p2,1);
                 terrain(C,M2,P3,M3,store[p1[1]+half_length]
[p1[2]+half_length],store[p2[1]+half_length][p2[2]],0);
                 terrain(M4,C,M3,P4,store[p1[1]+half_length]
[p1[2]],store[p1[1]+half_length][p1[2]+half_length],1);
            }
        }
            terrain(sq1[0],sq1[1],sq1[2],sq1[3],store[0][0],store[0]
[Math.pow(2,max_iteration)],1);

            var sorted = draw_data.sort();
            var final_data = sorted.reverse();
            var data_amount = final_data.length;
            for(i=0;i<data_amount;i++)
                {
                    ctx.beginPath();
                    ctx.moveTo(final_data[i][1][0],final_data[i][1][1]);
                    ctx.lineTo(final_data[i][2][0],final_data[i][2][1]);
                    ctx.lineTo(final_data[i][3][0],final_data[i][3][1]);
                    ctx.closePath();
                    ctx.strokeStyle='hsl('+ final_data[i][5] +', '+
Saturation_line +'%, ' + final_data[i][4] + '%)';
                    ctx.stroke();

                    if(FILL == 1)
                    {
                        ctx.fillStyle='hsl('+ final_data[i][5] +', '+
Saturation +'%, ' + final_data[i][4] + '%)';
                        ctx.fill();
```

```
                    }
                }
            }

    </script>
</html>
```