2D Procedural Map Generation
With Pascal & SwinGame

Jacob Milligan
Student ID - 100660682

1 Procedural Generation

1.1 Procedural over Manual

Very broadly speaking, in game development there are two primary ways to generate content for a project. The most common and controllable way is to make each part by hand - in our case we'll be referring to a 2D tile map as our content.

For small maps this isn't a problem; it's very straight-forward to declare each tile as an element of a statically-sized 2D array (we'll go into how this is done later) and just draw those tiles to the screen, perhaps also drawing different sprites on top of each tile for NPC's or the player. But what happens as our map grows in size? As it goes from a 32×32 map to a 256×256 sized map, or even larger? Even if we've created a system for writing our maps out as text files to be read in, already saving lots of time, this can very quickly become time-consuming. This is a valid way of generating content, in fact the developers on CD Projekt Red's The Witcher 3: Wild Hunt did just that (Klepek 2015), a pretty amazing feat. However, we don't have the resources or manpower of CD Projekt Red, so what's the solution?

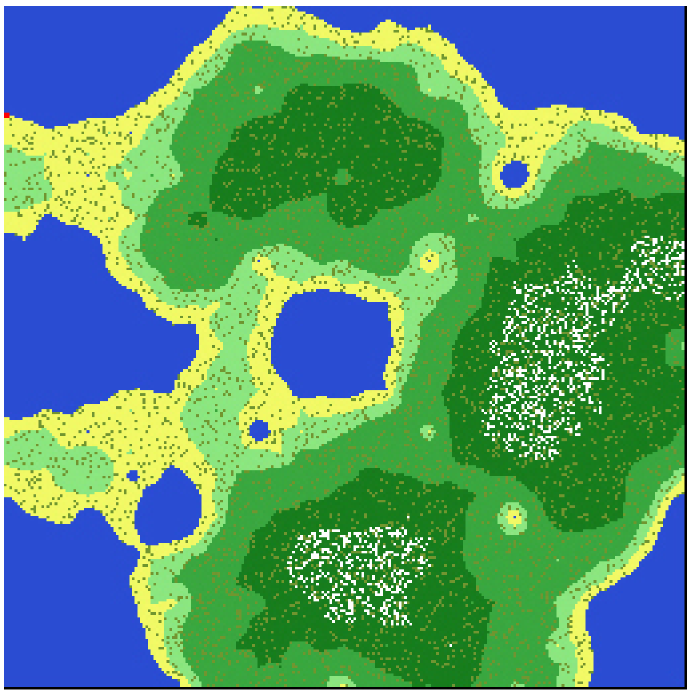
Procedural Generation algorithms are the solution

Games such as Minecraft, Dwarf Fortress, and the upcoming No Mans Sky all make use of procedural generation to generate enormous, beautiful, but seemingly random worlds. We say *seemingly* random because, aside from computers only being able to generate *pseudo*-random numbers, these algorithms are designed so that, with the same starting point, it will produce the same result.

"So then where do we start?" Good question. Many games, such as in indie title Dwarf Corp. (Klingensmith 2013), begin by simulating tectonic plate activity, erosion, and river formation to carve out their terrain - in a similar process to how terrain forms in the real world (Huggett 2007, pp. 46). However, we're going to go a different route and start by generating a realistic height map, a 2D array of elements that hold a generated elevation value, that we'll use to base the rest of our map off. We will use this starting point to procedurally generate a 512×512 sized map that can be navigated by the player. Along the way, we will make heavy use of the SwinGame API to handle all graphics-related functionality and briefly touch on other concepts such as basic collision detection, all of which we will code using Pascal.

1.2 Diamonds & Squares

To generate a heightmap, it would be possible to design an algorithm from scratch, however that would take a long time and the result probably wouldn't be very effective, so we're going to borrow a very well-known and academically sound one called **Random Midpoint Displacement** (Fournier, Fussell, and Carpenter 1982), also known as **the Diamond-Square Algorithm**. At its core, the purpose of this algorithm is to generate pseudo-random noise in a desirable pattern, i.e. one that resembles a realistic spread of terrain height values. Each point of noise is stored in a data structure (in our case, a 2D array) and holds a single value - a number representing its elevation. The result is something like this:

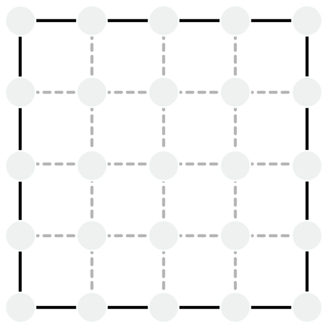


Example 1: A map generated using Diamond-Square

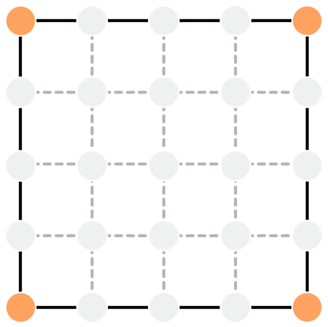
The basic concept behind Diamond-Square can be summed up like so:

- Take an empty grid which must be of size $2^n + 1$ in order to work. Then assign the corners a *seed* value, a number that all other calculations are based off. This means that with the same seed, we should get the exact same result.
- **The Sqaure Step** - Take the grids four corners, average their total, find their mid point and assign that point the average plus a random value.
- **The Diamond Step** - Given the previous step, we now have a diamond shape surrounding a new mid point. Take the average of all points in the diamond and assign the new midpoint that value plus a random amount.
- Use a `nextStep` variable to determine the next point to calculate.
- Iterate until `nextStep` is smaller than zero.

This process can be best visualized using graphs, seen in the example pictured below.

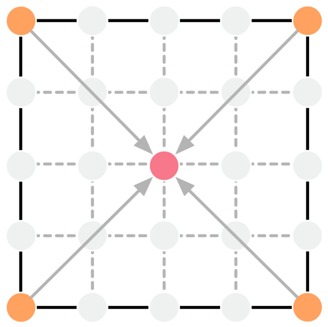


Step 1. Begin with a blank 2D array which represents a map. It can be viewed as a graph with each node representing a single tile



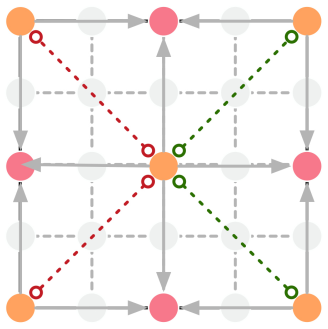
Step 2. Take the four corner nodes of the map and assign them a given seed value (in our case -1500 to force a continent-like map) which will influence the rest of the map

The Square step

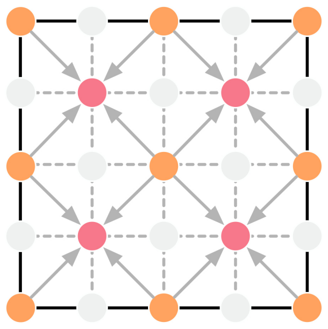


Step 3. Take the ***mid point*** between all four
nodes and assign it the average of the four corner nodes ***plus another random value.*** This gives us a ***square shape*** with a middle point.

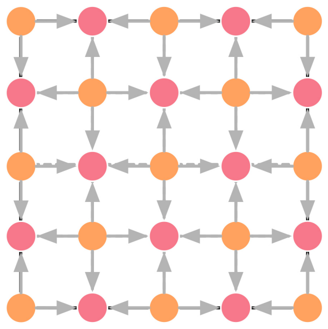
The Diamond step



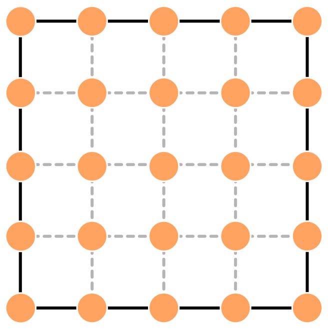
Step 4. Take the middle of each corner-to-corner edge from the previous square step (the pink nodes above) and assign them the average of all surrounding nodes in a ***diamond*** shape (in the above graph, the red and green dotted lines represent half-diamonds) plus a random value.



Step 5. The result of the previous step is another square (the orange nodes above). Therefore we now do the square step again.



Step 6. As before, the result of the previous step is a diamond shape surrounding each of the pink nodes. Therefore we complete the diamond step one more time.



Step 7. We have our complete tile grid!

Example 2: Summary of the Diamond-Square algorithm

Using this algorithm, we now have the ability to generate a starting point. However, before starting on an implementation, as with all software development, it's a good idea to define the requirements for our program - what functions, procedures, data structures, and features do we want to include?

We'll need the following data structures, functions, and procedures:

- First, we will define the data structures that will be required by our program. We're going to first need a `Tile` `record` to hold data related to each tile in the map such as if it's collidable, the bitmap that should be drawn for it, it's type, and elevation. We'll also need a `MapData` `record` to contain our tile grid, the players sprite and location data, and its size.
- Very importantly, we'll also need our terrain generation procedures - `DiamondSquare()` & `GenerateTerrain()`. `DiamondSquare()` will be responsible for creating a new heightmap for a passed-in `MapData()` record, whereas `GenerateTerrain()` will be responsible for deciding how each tile should be rendered based off the heightmap, alongside generating trees on the passed-in `MapData()` record.
- Now that our terrain generation functions and structures are defined, we'll also need a `CreateMap()` function to call both of the above procedures and then to search for an appropriate place on the map to spawn the player.
- Finally, we'll need both a `HandleInput()` procedure and a `DrawMap()` procedure to move the player around while detecting collision tiles and to draw the tile grid to the screen respectively.

There will also be several functions and resources referenced later on that we won't be building as they aren't directly related to procedural generation and are just utilities for allowing our map to render properly. This code sits in the `MapUtils.pas` file and can be downloaded from github, as part of the source for the finished project, alongside the bitmap resources we'll be using (if you don't have git installed just click the 'clone or download' link and download as a zip file). These extra files are important for loading bitmaps, updating the camera position relative to the edge of the map, and drawing a map overview to the screen.

2 Coding Terrain Generation

2.1 Setting Up

We need to implement Diamond-Square before we can do any other terrain generation. First, download and install the latest version of FPC (Free Pascal Compiler) and a Pascal SwinGame template from the SwinGame Website. Once this is complete, copy your downloaded SwinGame template to wherever you normally store your code (on my Mac it sits in `/Users/Jacob/Dev/Repos/` - all our coding will take place in the `/src/` folder and whenever you need to build and run the game, type the command `./build.sh && ./run.sh` (drop the `./` on Windows machines). Rename the `GameMain.pas` file to something a bit more descriptive, such as `ProceduralGeneration.pas` and open it up in your favourite text editor.

The first thing we need to do is to replace the code in the stock `Main()` procedure with the following:

```
procedure Main();
var
  map: MapData;
begin
  DiamondSquare(map, 100, 20);
  PrintMapToConsole(map);
end;
```

Before we render anything to a graphics window, we should first implement our algorithm and ensure that it functions correctly by printing it to the console, both procedures that will be called from `Main()`. We've also declared a new `MapData` variable which we'll be creating soon.

Next, create a new file in the `/src/` directory called `Terrain.pas`, open it up and write a new Unit file skeleton:

```pascal
unit Terrain;

interface
  uses SwinGame;

  type
    // Valid tile types for building maps with. Used as a terrain flag for different
↪   logic.
    TileType = (Water, Sand, Dirt, Grass, MediumGrass, HighGrass, SnowyGrass, Mountain);

    // Represents a feature on top of a tile that can have a bitmap, collision, and be
↪   interactive
    FeatureType = (NoFeature, Tree);

    // Represents a tile on the map - has a terrain flag, elevation and bitmap
    Tile = record
      flag: TileType; // Terrain type
      feature: FeatureType; // Type of feature if any
      collidable: Boolean; // Tile uses collision detection
      elevation: Integer; // The tiles elevation - zero represents sealevel.
      bmp: Bitmap; // Tiles base bitmap
      featureBmp: Bitmap; // If has feature, its bitmap
      hasBmp: Boolean;
    end;

    // Array used to hold a tilemap
    TileGrid = array of array of Tile;

    // Main representation of the current map. Holds a tile grid, alongside data related
↪   to size, smoothness, seed values.
    MapData = record
      tiles: TileGrid; // All of the tiles on a map
      player: Sprite;
      playerX, playerY: Integer; // Tile-based coordinates
      size, seed, tilesize, playerIndicator: Integer; // Map settings
    end;

  // Fills a MapData's TileGrid with generated heightmap data using the Diamond-Square
↪   fractal generation algorithm
  // This heightmap data gets used later on to generate terrain realistically
  procedure DiamondSquare(var map: MapData; maxHeight, smoothness: Integer);

  // Uses elevation values generated by DiamondSquare to assign appropriate bitmaps and
↪   randomly generate trees
  procedure GenerateTerrain(var map: MapData);

implementation

  procedure DiamondSquare(var map: MapData; maxHeight, smoothness: Integer);
  begin
    //code
  end;

  procedure GenerateTerrain(var map: MapData);
  begin
    //code
  end;

end.
```

That's a lot of code, so let's step through it.

```
unit Terrain;

interface
  uses SwinGame;

  type

    //
    //  Valid tile types for building maps with.
    //  Used as a terrain flag for different logic.
    //
    TileType = (Water, Sand, Dirt, Grass, MediumGrass, HighGrass, SnowyGrass, Mountain);

    //
    //  Represents a feature on top of a tile that can have a bitmap,
    //  collision, and be interactive
    //
    FeatureType = (NoFeature, Tree);
```

First, we create a new `unit` file named `Terrain`. A `unit` file has two sections of code:

- The `interface` where all of our types are declared alongside **forward-declared** functions and procedures. This is the part of the `unit` file that other units and the main program will actually see.
- The `implementation` section where we actually define the body of our functions and procedures.

Both of these sections of code can make use of the types, functions, and procedures created in other units through the `uses <UnitName>` syntax.

In the `type` section of the unit, we declare two enumeration types.- `TileType` & `FeatureType`. These will be used by our `GenerateTerrain()` procedure and the `MapUtils.pas` `unit` file to determine how to treat different tiles. Of note is the `FeatureType enumeration` which at the moment can only be a Tree or nothing. Generally speaking, If we only wanted to represent Trees in the game world, we would be better off using a `hasTree Boolean` variable but the reason we've used an `enumeration` is to future-proof our program; if we wanted to, later on, add logs or rocks to the game we would only need to add a new element to `FeatureType` and alter the terrain generation code.

```
//
//  Represents a tile on the map - has a terrain flag,
//  elevation and bitmap
//
Tile = record
  // terrain type
  flag: TileType;

  // type of feature if any
  feature: FeatureType;

  // uses collision detection
  collidable: Boolean;

  //
  //  Represents the tiles elevation - zero represents sea
  //  level.
  //
  elevation: Integer;

  // tiles base bitmap
  bmp: Bitmap;

  hasBmp: Boolean;

  // bitmap for whatever feature is on top of the tiles
  featureBmp: Bitmap;
end;

//
//  Array used to hold a tilemap
//
TileGrid = array of array of Tile;

//
//  Main representation of the current map. Holds a tile grid, alongside
//  data related to size, smoothness, seed values.
//
MapData = record
  tiles: TileGrid;
  player: Sprite;
  playerX, playerY: Integer;
  size, seed, tilesize, playerIndicator: Integer;
end;
```

Here, we declare our most important records and types. The `Tile` `record` is what represents a single element of our tile grid and contains a `TileType` `flag`, a `FeatureType`, a `Boolean` variable, `collidable` to communicate that the particular tile is subject to collision detection, the `elevation` `Integer` value, its attached base tile `Bitmap` and its feature `Bitmap` (in this case either a Tree or an invisible bitmap) to render alongside a `hasBmp` `Boolean` value used to stop our drawing procedures from trying to draw a non-existent bitmap. We've also declared a new `open array of dynamic Tile arrays` to function as our tile grid. As 2D arrays are essentially just an array in which each of its elements is just another array of elements of a specified type, we've used the syntax `array of array of Tile` to declare this type.

Finally, we declare our `MapData` type. This `record` will hold our tile grid, the players `Sprite` variable (A SwinGame library data type), the size of the map, the size of each tile, it's seed or starting value, and an indicator used by the `MapUtils.pas DrawMapCartography()` `procedure` to locate where the player is relative to the drawn tile map. Important to note is the `playerX` & `playerY` variables as these aren't the players position in pixel coordinates (there will be a total of 275952697344 pixels on the final map, way too large a number to even fit in a `LongWord` type), they are the players current pixel position translated to 2D array index equivalents - these variables will be used to calculate simple collision detection later on. Lastly, we forward declare our two terrain generation procedures.

```
//
//  Fills a MapData's TileGrid with generated heightmap data
//  using the Diamond-Square fractal generation algorithm
//  This heightmap data gets used later on to generate terrain realistically
//
procedure DiamondSquare(var map: MapData; maxHeight, smoothness: Integer);

//
//  Uses elevation values generated by DiamondSquare to assign appropriate
//  bitmaps and randomly generate trees
//
procedure GenerateTerrain(var map: MapData);
```

2.2 Implementing Diamond-Square

Moving onto the `implementation` section, we can now build our terrain generation algorithms. Starting with `DiamondSquare`. The basic pseudocode for the algorithm looks something like this:

Pseudocode 1 The Diamond-Square algorithm

```
procedure DiamondSquare($map$, $maxHeight$, $smoothness$)
    Initialize the four corners of the map with a seed value
    $nextStep \leftarrow \frac{Length(tileGrid)}{2}$
    while $nextStep > 0$ do
        for all $midPoints$ of each square in the grid do                    ▷ Do square step
            $midPoint \leftarrow$ Average four corners + $(Random(maxHeight) \times smoothness)$
        end for
        for all Diamonds in the map do                    ▷ We now have diamonds, do diamond step
            for all $point$ in a diamond do
                $pointCount \leftarrow 0$
                if Within boundaries of the tile grid then
                    $midPoint \leftarrow midPoint + point$
                    $pointCount \leftarrow pointCount + 1$
                end if
            end for
            $midPoint \leftarrow \frac{midPoint}{pointCount} + (Random(maxHeight) \times smoothness)$
        end for
        $nextStep \leftarrow \frac{nextStep}{2}$                    ▷ Smaller diamonds and squares
        $smoothness \leftarrow \frac{smoothness}{2}$        ▷ Higher elevations have less radical difference in height
    end while
end procedure
```

When broken down like this, the process becomes a lot simpler and we have a good abstraction to reference when implementing the algorithm, so let's start on building it into our source code.

```
implementation

  procedure DiamondSquare(var map: MapData; maxHeight, smoothness: Integer);
  var
    x, y: Integer;
    midpointVal: Double;
    nextStep, cornerCount: Integer;
  begin
    x := 0;
    y := 0;
    midpointVal := 0;
    nextStep := Round(Length(map.tiles) / 2 ); // Center of the tile grid

    // Seed upper-left corner extremely low elevation to force it to
    // start with water
    map.tiles[x, y].elevation := -1500;
```

Initially, we declare the x & y variables to track our iterations through the heightmap generation process, then the `midpointVal` `Double` which we will use to calculate the current midpoint value (average plus a random value) at both the diamond and square steps. The `nextStep` variable is an important one and will control which tile in the grid we're analysing at any given moment and will be made smaller at each iteration until it equals 0, at which point the algorithm is finished. This is set to the centre point of the tile grid and at each iteration will be used to determine the location of a midpoint. Finally, we 'seed' the top-left corner of the map with an elevation of -1500 to ensure that the map will always have some ocean as its starting point which will help achieve our goal of producing a continent-like map.

```
// Initialize four corners of map with the same value as above
while x < Length(map.tiles) do
begin
  while y < Length(map.tiles) do
  begin
    map.tiles[x, y].elevation := map.tiles[0, 0].elevation;
    y += 2 * nextStep;
  end;

  x += 2 * nextStep;
  y := 0;
end;
```

We then iterate all four corners of the map, stepping the entire length of the map at a time, and assign each corner the same elevation value as the top-left corner. Something that you may notice is that we aren't using the more obvious `for..do loop` to iterate the 2D tile grid. This is due to a quirk that's relatively unique to Pascal and some Pascal-derived languages in that a `for..do loop` can only increment the control variable by 1 at a time, yet we need to increment $2 \cdot nextStep$ (currently half the size of the map) at each iteration to get to the next corner of the map, therefore we'll be using `while..do loops`.

```
x := 0;
y := 0;

while nextStep > 0 do
begin
  midpointVal := 0;
```

Here begins the core of the generation process, essentially we want to continue to iterate the process into smaller and smaller sizes until `nextStep` is smaller than the size of the tile grid. Inside this loop we start with the **square step**:

```
x := nextStep;
while x < Length(map.tiles) do
begin

  y := nextStep;
  while y < Length(map.tiles) do
  begin

    //
    // Sum surrounding points equidistant from the midpoint
    // in a square shape
    //
    midpointVal := map.tiles[x - nextStep, y - nextStep].elevation
           + map.tiles[x - nextStep, y + nextStep].elevation
           + map.tiles[x + nextStep, y - nextStep].elevation
           + map.tiles[x + nextStep, y + nextStep].elevation;

    // Set midpoint to the average + Random value and multiply by smoothing factor
    map.tiles[x, y].elevation := Round( (midpointVal / 4) + (Random(maxHeight) *
↪   smoothness) );
    y += 2 * nextStep;
  end;

  x += 2 * nextStep;
  y := 0;
end;
```

Here, we're scanning the tile grid for squares and their corners of the current iteration size, as in Example 2, step 5. `midpointVal` is assigned the sum of the four corners in a square surrounding the current midpoint, `map.tiles[x, y]`, then we assign the midpoints elevation the average of the points plus a random value with a maximum possible height of our passed-in `maxHeight` variable. Each time we do this part of the step, also seen in the diamond step, we ensure to multiply the random value by the passed-in smoothness value to allow the terrain to smooth out as the elevations become higher and the calculated map spaces become smaller. **Very importantly, because we aren't using `for..do` loops, note that we're manually resetting both `x` & `y`**, don't forget to do this before each iteration. Let's move onto the diamond step.

```
//
// Diamond step.
// Check surrounding points in a diamond around a given midpoint. Checks if they're
↪  within the bounds
// of the map
//
x := 0;
while x < Length(map.tiles) do
begin

  y := nextStep * ( 1 - Round(x / nextStep) mod 2);
  while y < Length(map.tiles) do
  begin
    midpointVal := 0;
    cornerCount := 0;

    // Sum the surrounding points equidistant from the current midpoint, checking in a
↪   diamond shape, then calculating their
    // average and adding a random amount. Ensures that the corner checking is within
↪   the bounds of the map
    if ( y - nextStep >= 0 ) then
    begin
      midpointVal += map.tiles[x, y - nextStep].elevation;
      cornerCount += 1;
    end;
    if ( x + nextStep < Length(map.tiles) ) then
    begin
      midpointVal += map.tiles[x + nextStep, y].elevation;
      cornerCount += 1;
    end;
    if ( y + nextStep < Length(map.tiles) ) then
    begin
      midpointVal += map.tiles[x, y + nextStep].elevation;
      cornerCount += 1;
    end;
    if ( x - nextStep >= 0 ) then
    begin
      midpointVal += map.tiles[x - nextStep, y].elevation;
      cornerCount += 1;
    end;

    //
    // If at least one corner is within the map bounds, calculate average plus
    // a random amount less than the map size.
    //
    if cornerCount > 0 then
    begin
      // Set midpoint to the average of corner amt + Random value and multiply by
↪   smoothing factor
      map.tiles[x, y].elevation := Round( (midpointVal / cornerCount) +
↪   Random(maxHeight) * smoothness );
    end;

    y += 2 * nextStep;
  end;

  x += nextStep;
end;
```

The diamond step is a little more complicated. The only reason for this is that we need to do a bit more checking to ensure that a given point in the diamond is within the map boundary to avoid both calculation errors and a `EAccessViolation` error for accessing a non-existent memory address.

Once again we have two, nested `while..do loops`. Before entering each inner loop, `y` is assigned a new value which will be equal to the next point in the diamond. Why is this the case? Well, given the formula, where $s =$ `nextStep`:

$$y = s \cdot (1 - \left[\frac{x}{s}\ mod\ 2\right] \tag{1}$$

If we were to plug in $x = 0$ & $s = 3$ which is where x will start at the beginning of the diamond step and what `nextStep` will be set to in the first iteration of a 5×5 grid, we would get:

$$\begin{aligned} y &= 3 \cdot (1 - \left[\frac{0}{3}\ mod\ 2\right]) \\ &= 3 \cdot (1 - [0\ mod\ 2]) \\ &= 3 \cdot (1 - [0]) \\ &= 3 \end{aligned} \tag{2}$$

`y` is now the right point of the diamond, then, given `x += nextStep` in the next iteration, `x` will now be 3, so if we plug that into the formula again, we get:

$$\begin{aligned} y &= 3 \cdot (1 - \left[\frac{3}{3}\ mod\ 2\right]) \\ &= 3 \cdot (1 - [1\ mod\ 2]) \\ &= 3 \cdot (1 - [1]) \\ &= 3 \cdot 0 \\ &= 0 \end{aligned} \tag{3}$$

So now `y` is set to the top point of the diamond and so on. This simple formula allows us to iterate four points of a diamond without having to resort to using `if` statements.

In the body of the inner loop, we then check each point surrounding the mid-point to see if it's inside the bounds of the map. If it is, we increment `cornerCount` once. Then, after checking all points we assign the mid-point the average of all surrounding points within the bounds of the map plus a random value limited to our `maxHeight` variable multiplied by the current smoothness factor. Finally, we increase the value of `y` in the inner loop by the length of the current diamond to get the lower point and iterate until all diamonds in the current loop are calculated.

```
    nextStep := Round(nextStep / 2); // Make the next space smaller

    //
    //  Increase smoothness for every iteration, allowing
    //  less difference in height the more iterations that are completed
    //
    smoothness := Round(smoothness / 2);
  end;
end;
```

We're almost done. At the end of the current diamond-square iteration, we make the size of our diamonds and squares smaller by halving `nextStep` alongside halving `smoothness` so that higher elevations have a less radical difference in height. Changing our smoothness value is what creates a realistic gradient in heights and a more visually pleasing result.

2.3 Testing

The most complex and difficult part is out of the way. But before we do anything else, we need to test it. If we were to wait until we had all of our drawing, collision, and update functions coded up, we'd be well into finishing our program before we've even made sure the core algorithm behind it even works! To do this, we can go back to our main program file, `ProceduralGeneration.pas`, and implement the `PrintMapToConsole()` `procedure` we wrote in `Main()`.

References

Fournier, Alain, Don Fussell, and Loren Carpenter (1982). "Computer Rendering of Stochastic Models". In: *Commun. ACM* 25.6, pp. 371–384.

Huggett, John (2007). *Fundamentals of Geomorphology*. Ed. by John Huggett. 2nd Edition. London: Routledge, p. 46.

Klepek, Patrick (2015). *How The Witcher 3's Developers Ensured Their Open World Didn't Suck*. URL: http://kotaku.com/how-the-witcher-3s-developers-ensured-their-open-world-1735034176.

Klingensmith, Matt (2013). *How we Generate Terrain in DwarfCorp*. URL: http://www.gamasutra.com/blogs/MattKlingensmith/20130811/198049/How_we_Generate_Terrain_in_DwarfCorp.php (visited on 08/11/2013).