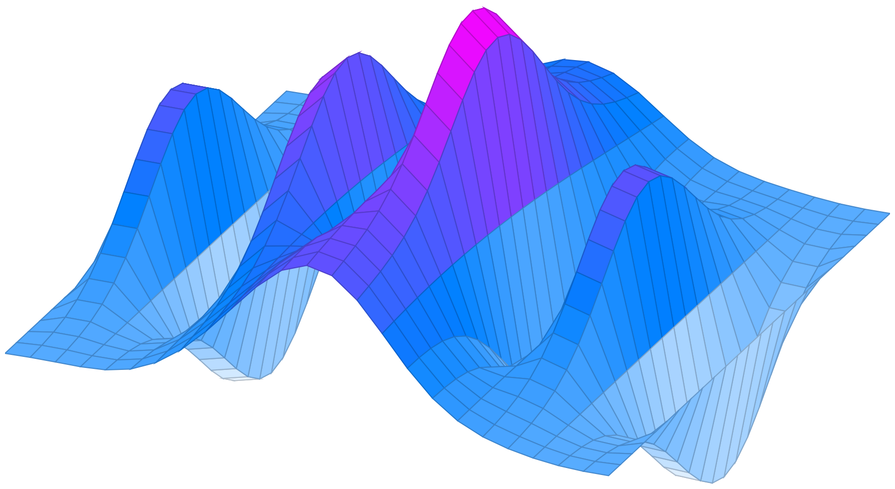
2D Procedural Map Generation

With Pascal & SwinGame



Jacob Milligan

Student ID - 100660682

Contents

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 produce each piece of content by hand. The consequences, in the case of this article, would be the production of a large 2D tile map by a manual process.

For smaller content sizes this isn't a problem; it's a relatively straight-forward process to declare each tile as an element of a statically-sized 2D array and just draw those tiles to the screen in a single pass alongside, perhaps, the drawing different sprites on top of each tile to represent NPC's or the player. But what happens as a map grows in size? As it increases from a 32 $\times$ 32 map to a 256 $\times$ 256 sized map, or even larger? Even if the programmer had created a time-saving, high-level system for creating maps as text files to be read in by the program, it can very quickly become time-consuming and tedious. Although 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), most programmers don't possess the resources or manpower of a AAA game studio and therefore often must create a better solution. So what's the appropriate solution?

Procedural Generation algorithms are the solution

Indie game title such as Minecraft, Dwarf Fortress, and the upcoming No Mans Sky all make use of procedural content generation to build enormous, beautiful, but seemingly random and unique worlds. We say *seemingly* random for two reasons:

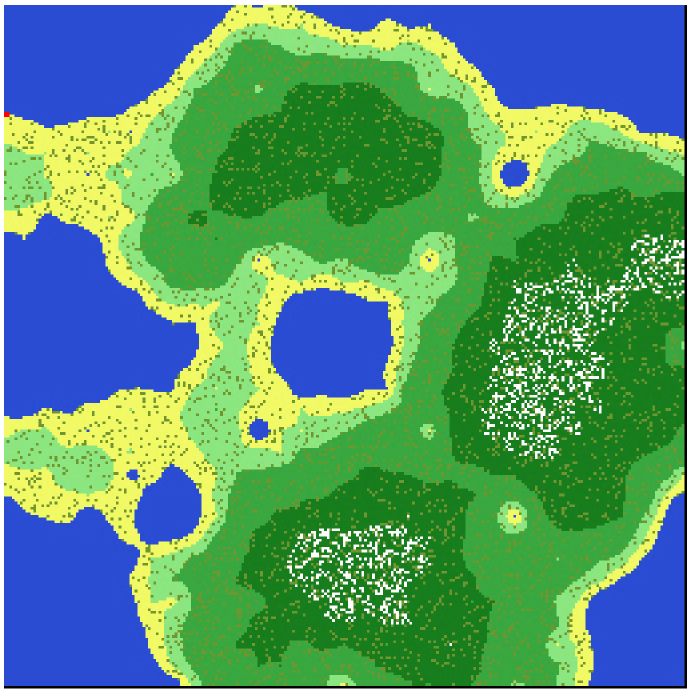
1. Computers can only produce *pseudo*-random numbers as the only truly random processes are analog, physically occurring phenomena, such as the measurement of cosmic radiation or noise signals in hardware circuits (Güneysu 2010).
2. These procedural generation algorithms are designed so that, with the same starting point, it will produce the same result.

What is the starting point for this process?

The answer to that question is wide and varied and many games, such as in indie title Dwarf Corp., begin by simulating tectonic plate activity (Klingensmith 2013), erosion, or river formation to carve out their terrain - in a similar fashion to how terrain forms in the real world (Huggett 2007, pp. 46). However, this article will be taking a different route and begin by generating a realistic height map stored in a 2D array of elements that each hold a generated elevation value which will be used to base the remaining generation procedures off. We will use this initial information to procedurally generate a 512$\times$512 sized 2D continent-like map that can be navigated by a player character. Furthermore, we will make heavy use of the SwinGame API to handle all graphics-related functionality and briefly touch on other interesting concepts such as basic collision detection and drawing algorithms, all of which will be coded using the Pascal programming language.

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, would need to be rigorously tested, and the result probably wouldn't be very effective, so the core of our program will be based off a very well-known and well-tested one named **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 resulting map will look something like this:

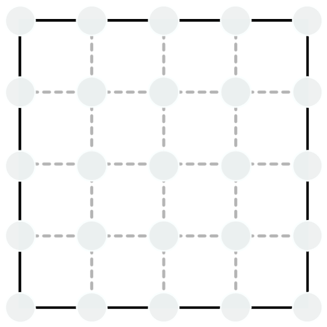


Example 1: A map generated using Diamond-Square

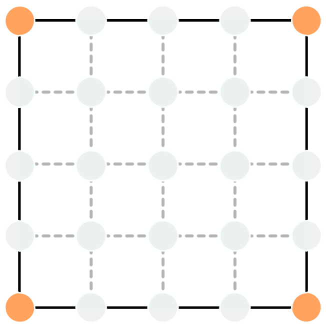
The basic concept behind Diamond-Square can be summed up like so:

- Take an empty grid, which must be of size 2^n+1 in order for the following steps to function correctly. Then assign the corners a *seed* value, a number that all other calculations are based off. This will mean that with the same seed, the same result should be produced. Then iterate the following two steps:
  - **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 up to the maximum defined height of the map.
  - **The Diamond Step -** Given the previous step, we now have a diamond shape of four points 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 up to the maximum defined height value.
- Use a `nextStep` variable to determine which point in each diamond and square step to calculate.
- Iterate until `nextStep` is less than zero.

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

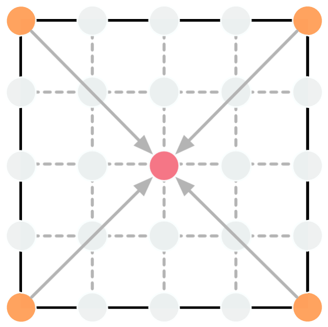


Step 1. Begin with a blank 2D array which represents a map. It can be viewed as a graph with each node representing a single tile



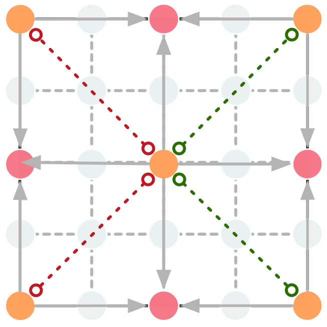
Step 2. Take the four corner nodes of the map and assign them a given seed value (in our case -1500 to force a continent-like map) which will in uence the rest of the map

The Square step

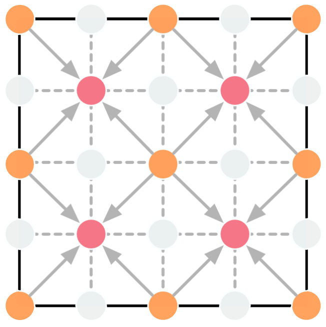


Step 3. Take the ***mid point*** between all four
nodes and assign it the average of the four corner nodes ***plus another random value.*** This gives us a ***square shape*** with a middle point.

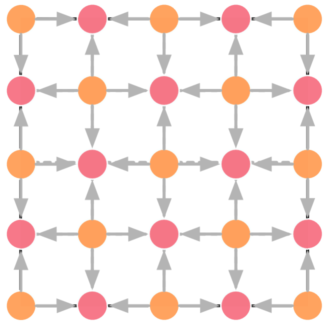
The Diamond step



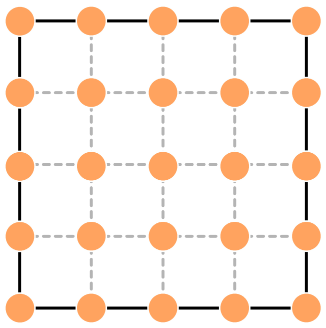
Step 4. Take the middle of each corner-to-corner edge from the previous square step (the pink nodes above) and assign them the average of all surrounding nodes in a ***diamond*** shape (in the above graph, the red and green dotted lines represent half-diamonds) plus a random value.



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 provides the ability to generate a starting point. However, before beginning an implementation, as with all software development, it would be wise to define the requirements for the final program - the functions, procedures, data structures, and features that should be included:

- First, we'll need a `Tile` `record` to hold data related to each tile in the map such as collision value, its associated bitmap, it's terrain type, and elevation. We'll also need a `MapData` `record` to contain our tile grid, the players sprite and location data, and its total size.
- We'll need two primary terrain generation procedures - `DiamondSquare()` & `GenerateTerrain()`. `DiamondSquare()` will be responsible for creating a new heightmap for a given `MapData()` parameter, whereas `GenerateTerrain()` will be responsible for deciding how each tile should be rendered based off the heightmap, alongside generating trees on the new `MapData's` tile grid.
- Now that the terrain generation functions and data structures are defined, we'll also need a `CreateMap()` function to call both of the above procedures and 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 you can 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

The primary goal of this article is to implement Diamond-Square and must be done prior to 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 (see each websites installation section for instructions on how to do this). Once this is complete, copy your downloaded SwinGame template to wherever you normally store your code, i.e. `/Users/Jacob/Dev/Repos/`. All of the programming will take place in the `/src/` folder and whenever the game needs to be built and run, 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:

```
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 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 the body of each function and procedure is actually defined.

Each unit and program can make use of the data structures, functions, and procedures created in other units via 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 either 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 tiles `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 and crash the game. 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 data type from the SwinGame library), 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
```

Writing pseudocode can often be a good idea for breaking down a complex problem, making the underlying process seem a lot simpler. Consequentially, we now 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 initially assigned the centre point of each axis in the tile grid, at first being multiplied by two to get the four corners of the map, and at each iteration will be used to determine the location of a given point in a diamond or a square. 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. However 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 value the average of the points plus a random value with a maximum possible height of our passed-in `maxHeight` variable `Random(maxHeight)`. Each time we complete the elevation assignment statement, also seen in a similar fashion the diamond step, we ensure we multiply the random value by the given `smoothness` parameter to allow the terrain to smooth out as the elevations become higher and the calculated map spaces become smaller to allow for less radical and unrealistic changes. **Very importantly, because we aren't using `for..do` loops, note that we're manually resetting both `x` & `y`**, if you forget to do this before each iteration you may lose your mind. Let's move onto the diamond step.

```
// Diamond step. Points in a diamond shape 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 in a diamond shape.
↪  Ensures that each point 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 height.
    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 reason for this is that we need to do extra checking to ensure that a given point in the diamond is within the maps 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` with their control variables manually reset and incremented at the before and at the end of each iteration. Before entering each inner loop, rather than assigning `y := nextStep` as in the square step, we assign `y` a seemingly confusing 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, at the current iteration in the process, $x = 0$ & $s = 3$ which is coincidentally where x will be at the beginning of the diamond step and what `nextStep` will be assigned 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 two elements from the right point of the diamond and will be skipped this iteration 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 left-most diamond and so on. This simple formula allows us to iterate four points of a diamond without having to resort to using numerous **`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, it must be tested. If we were to wait until we had all of our drawing, collision, and update procedures built, we would be well into finishing our program before we'd even made sure the core algorithm behind it works. To do this, we can go back to our main program file, `ProceduralGeneration.pas`, and implement the `PrintMapToConsole()` and `CreateMap()` procedures/functions we wrote in `Main()`.

```
// Prints all of the elevation data in a tilemap to the console.
// Don't use for maps larger than 16 x 16 or it will be bigger than the console window
procedure PrintMapToConsole(constref map: MapData);
var
  x, y: Integer;
begin
  for x := 0 to High(map.tiles) do
   begin
     for y := 0 to High(map.tiles) do
     begin
       Write(map.tiles[x, y].elevation, ' ');
     end;
     WriteLn();
   end;
end;
```

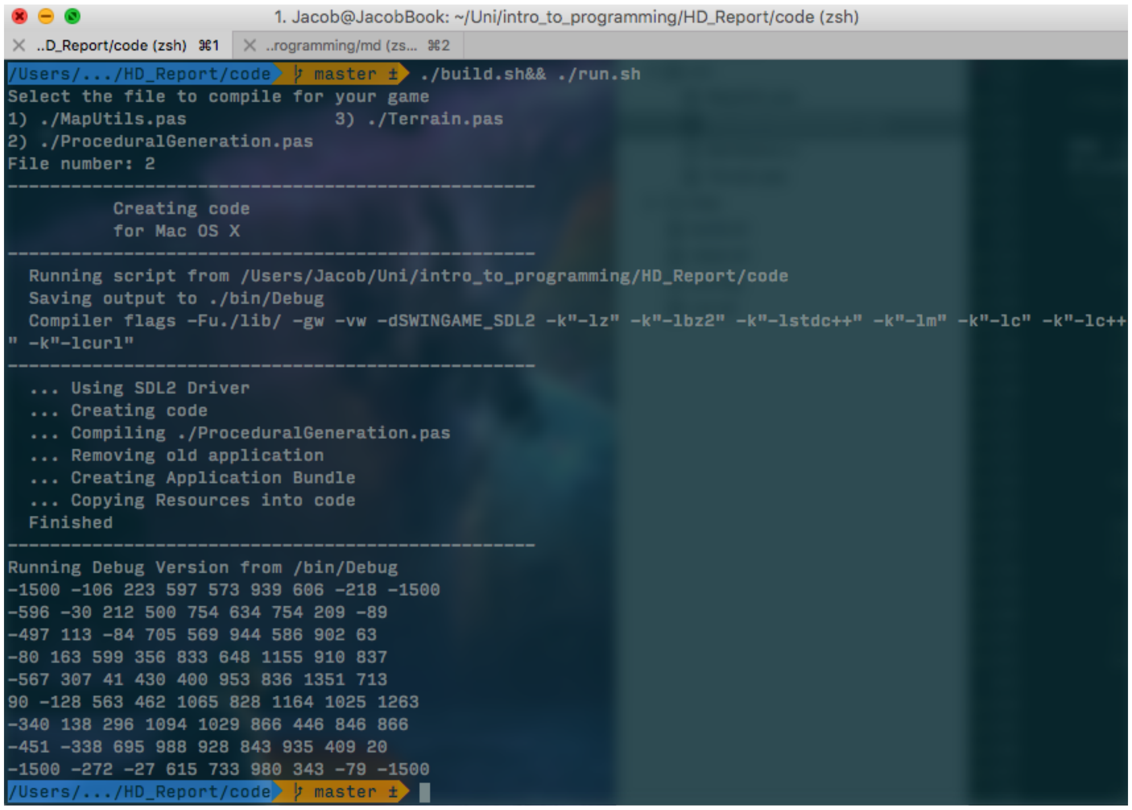
The code for this procedure is fairly self-explanatory - we iterate the 2D array and write all `y` values without appending a newline to simulate a row, then at the end of each `x` iteration, write a newline to the console to simulate a column. Then, we move onto `CreateMap()`.

```pascal
//
// Initializes a new tile grid and then generates a new map using DiamondSquare().
// The new map can be random or based off a given seed
//
function CreateMap(size: Integer; random: Boolean; seed: Integer = 0): MapData;
var
 i, j: Integer;
begin
 result.tilesize := 32;
 result.size := size;
 result.player := CreateSprite('player', BitmapNamed('player')); // We'll use
↪  this later

 // Setup seed
 if random then
 begin
  Randomize;
 end
 else
 begin
  RandSeed := seed;
 end;

 // Initialize Tile Grid
 SetGridLength(result.tiles, size);
 // Generate Heightmap
 DiamondSquare(result, 100, 20);
end;
```

`CreateMap()` takes three parameters - a value to determine the map size, a **`Boolean`** variable to determine if we should generate a random map or one from a pre-determined seed value, and the actual seed value if any exist, using the `<variable>: <type> = <value>` syntax to declare a default parameter that doesn't have to necessarily be passed as an argument when calling the procedure. We first assign a tilesize, size, and player sprite for our `MapData` `result` variable. Then, if `random` is true we call the `Randomize` **`procedure`** which initializes Pascals internal random number generator with a new, unique seed to base all random generation off. Much like our *seed* value used to base all of our Diamond-Square calculations off, the Pascal `Random()` function, a deterministic *pseudo-random* number generation algorithm, itself requires a seed value, generated by the exact time on the computers system clock down to nanoseconds, to base its calculations off. If `random` is false, we then assign our seed value to the Pascal `RandSeed` variable, which sets `Random()`'s seed value manually rather than using the system clock via `Randomize`. We then call `SetGridLength()` (listed below) to initialize our 2D array and pass our map into `DiamondSquare()` to get assigned a heightmap.

```
//   Initializes the 2D map grid with the given size and sets the default
//   values for each tile
procedure SetGridLength(var tiles: TileGrid; size: Integer);
var
  column: Integer;
  x, y: Integer;
begin

  // Set all of the column sizes to the right length
  for column := 0 to size do
  begin
    SetLength(tiles, column, size);
  end;

  // Iterate map and setup default values for all tiles
  for x := 0 to High(tiles) do
  begin
    for y := 0 to High(tiles) do
    begin
      // Setup default values
      tiles[x, y].elevation := 0;
      tiles[x, y].collidable := false;
      tiles[x, y].feature := NoFeature;
      tiles[x, y].hasBmp := false;
    end;
  end;
end;
```

Now all of our core procedures and functions are defined, we can test out our `DiamondSquare()` **`procedure`** by calling `CreateMap(9, 100, 20)` from `Main()`; we want to pass in a very small size value to `CreateMap()` otherwise we won't be able to see all of the printed elevation numbers in the console. Note that we're passing in a size of 9 rather than 8; this is because, as previously discussed, `DiamondSquare()` only works for maps of size 2^n+1 and 9 is 2^3+1. Run `./build.sh && ./run.sh` in the console from the root project folder and we can see the result:


1. Jacob@JacobBook: ~/Uni/intro_to_programming/HD_Report/code (zsh)
..D_Report/code (zsh) ⌘1
..rogramming/md (zs... ⌘2

```
/Users/.../HD_Report/code  master ± ./build.sh&& ./run.sh
Select the file to compile for your game
1) ./MapUtils.pas              3) ./Terrain.pas
2) ./ProceduralGeneration.pas
File number: 2
--------------------------------------------------
          Creating code
          for Mac OS X
--------------------------------------------------
  Running script from /Users/Jacob/Uni/intro_to_programming/HD_Report/code
  Saving output to ./bin/Debug
  Compiler flags -Fu./lib/ -gw -vw -dSWINGAME_SDL2 -k"-lz" -k"-lbz2" -k"-lstdc++" -k"-lm" -k"-lc" -k"-lc++
" -k"-lcurl"
--------------------------------------------------
  ... Using SDL2 Driver
  ... Creating code
  ... Compiling ./ProceduralGeneration.pas
  ... Removing old application
  ... Creating Application Bundle
  ... Copying Resources into code
  Finished
--------------------------------------------------
Running Debug Version from /bin/Debug
-1500 -106 223 597 573 939 606 -218 -1500
-596 -30 212 500 754 634 754 209 -89
-497 113 -84 705 569 944 586 902 63
-80 163 599 356 833 648 1155 910 837
-567 307 41 430 400 953 836 1351 713
90 -128 563 462 1065 828 1164 1025 1263
-340 138 296 1094 1029 866 446 846 866
-451 -338 695 988 928 843 935 409 20
-1500 -272 -27 615 733 980 343 -79 -1500
/Users/.../HD_Report/code  master ±
```

Example 3: The heightmap printed to the console

Although it may look fairly unimpressive now, we can at least verify that our algorithm works and produces pseudo-random values distributed across nice, smooth gradients. Let's move onto our `GenerateTerrain()` procedure to actually begin implementing a graphical representation of our map.

2.4 Generating Terrain and Features

In order to graphically represent terrain, we need to assign Bitmaps and `FeatureTypes`. This will be defined in our `GenerateTerrain()` **`procedure`**. This step in terrain generation is less complex than `DiamondSquare()` *at the present*; it's possible, and encouraged, to extend this this procedure after finishing the program and, for many procedurally generated games, `DiamondSquare()` is just the beginning with the most complex algorithms belonging to the terrain generation procedures and functions. However, our focus is on generating a base map to get started with procedural generation. If you are interested, a visit to the Procedural Content Generation Wiki (PCG 2016) is highly recommend and browsing the many articles listed there will give a good jumping off point for many varied procedural processes.

Before we begin building the `GenerateTerrain()` **`procedure`** lets outline exactly how we want it to work by using pseudocode to abstract the algorithmic problem.

Pseudocode 2 Procedure to Generate Terrain

```
procedure GenerateTerrain($map$)
    for $x \leftarrow 0$ To $High(tiles)$ do                                   ▷ Generate tile Bitmaps
        for $y \leftarrow 0$ To $High(tiles)$ do
            Assign a different Bitmap to $tiles[x, y]$ for different $elevation$ ranges
            if $tiles[x, y]$ $elevation$ value is $< 0$ then
                $tiles[x, y] \leftarrow$ Dark Water Bitmap
            else
                $tiles[x, y] \leftarrow$ Mountain Bitmap                        ▷ $elevation > 1499$
            end if
            if $Random() > 0.9$ and $tiles[x, y]$ is not a Water tile then
                $tiles[x, y]'s\ FeatureType \leftarrow$ A tree appropriate for the current tile type
            end if
        end for
    end for
end procedure
```

Essentially all we need to do is iterate over our previously generated 2D heightmap and assign each tile a Bitmap based on its elevation value. At certain elevations we may choose to apply different processes later on for more complex generation, but all we're doing at the moment is assigning the right tile Bitmap to the right elevation value. The second part of the process is generating appropriate tree Bitmaps based on a very random process; it's highly encouraged to extend this and explore forest generation algorithms that work by creating seeds, sunlight, and maturation processes (see West 2008) but for the moment we're going to apply a process to achieve a semi-procedural result.

```
// Generates the terrain type for each tile based off its elevation value
procedure GenerateTerrain(var map: MapData);
var
  x, y: Integer;
begin
  // Iterate all tiles and change their bitmap and data depending on their
  // pre-generated altitude
  for x := 0 to High(map.tiles) do
  begin
    for y := 0 to High(map.tiles) do
    begin

      // Setup the tiles
      case map.tiles[x, y].elevation of
        0..199: SetTile(map.tiles[x, y], Water, 'water', true);
        200..299: SetTile(map.tiles[x, y], Sand, 'sand', false);
        300..399: SetTile(map.tiles[x, y], Grass, 'grass', false);
        400..599: SetTile(map.tiles[x, y], MediumGrass, 'dark grass', false);
        600..799: SetTile(map.tiles[x, y], HighGrass, 'darkest grass', false);
        800..999: if Random(10) > 6 then
              // Generate patchy snow at mid-to-high elevations
              SetTile(map.tiles[x, y], SnowyGrass, 'snowy grass', false)
             else
              SetTile(map.tiles[x, y], HighGrass, 'darkest grass', false);
        1000..1499: SetTile(map.tiles[x, y], SnowyGrass, 'snowy grass', false);
```

First, we delare our `GenerateTerrain()` procedure and begin iterating the 2D tile grid. We then define several elevation ranges inside a **`case`** statement for which we assign the current tiles Bitmap using `SetTile()` (which we'll implement soon) - lower elevations get water, while higher elevations get snowy grass. At elevation ranges of 800−999 we randomly assign either snow or grass to simulate patchy snow as the terrain transitions to higher altitudes.

```
        else
          // Only generate dark water if elevation is low enough
          // otherwise, every other value, which will be higher than 1499, should
          // become mountains
          if map.tiles[x, y].elevation < 0 then
            SetTile(map.tiles[x, y], Water, 'dark water', true)
          else
            SetTile(map.tiles[x, y], Mountain, 'mountain', true)
      end;

      // Generates trees randomly. Feel free to extend this by making your own
      // procedural tree generation algorithm to make beautiful forests!
      if ( Random() > 0.9 ) and ( map.tiles[x, y].flag <> Water ) then
      begin
        SetFeature(map.tiles[x, y], Tree, true);
      end;

    end;
  end;
end;
```

If the current tiles elevation value is outside the specified ranges we fall to the `else` block and assign elevations lower than 0 deep water tiles and anything not less than 0 will be higher than 1499 so we assign it a mountain tile. Finally, we randomly set the tiles' `featureBmp` to either a tree or nothing depending on both a random value and as long as the current tile isn't a water tile as trees don't generally grow in the ocean.

You will have noticed both `SetTile()` & `SetFeature()` procedures. These handle assigning default values to tiles and assigning the right tree for the right `TileType` respectively. Let's quickly code them up:

```
// Sets a tiles values to specified. Anything with collidable set to true
// won't be able to be walked over by the player
procedure SetTile(var newTile: Tile; flag: TileType; bmp: String; collidable:
↪  Boolean);
begin
  newTile.flag := flag;
  newTile.bmp := BitmapNamed(bmp);
  newTile.collidable := collidable;
  newTile.hasBmp := true;
end;
```

`SetTile()` assigns all of the specified values alongside any required default values for a given tile which saves us the hassle of rewriting this code each time we want to setup a tile.

```
//
//  Sets up a new feature on a given tile. At the moment we only have trees
//  as valid features but it's really easy for you to add more features!
//  Why not try adding rocks, logs or even treasure?
//
procedure SetFeature(var tile: Tile; feature: FeatureType; collidable: Boolean);
begin
  tile.feature := feature;
  tile.collidable := collidable;

  if feature = Tree then
  begin
    case tile.flag of
      Water: tile.featureBmp := BitmapNamed('hidden');
      Sand: tile.featureBmp := BitmapNamed('palm tree');
      Dirt: tile.featureBmp := BitmapNamed('tree');
      Grass: tile.featureBmp := BitmapNamed('tree');
      MediumGrass: tile.featureBmp := BitmapNamed('pine tree');
      HighGrass: tile.featureBmp := BitmapNamed('pine tree');
      SnowyGrass: tile.featureBmp := BitmapNamed('snowy tree');
      Mountain: tile.featureBmp := BitmapNamed('hidden');
    end;
  end;
end;
```

`SetFeature()` is fairly self explanatory - it uses a **`case`** statement on the tile parameter's `TileType` to determine which tree bitmap to load up. Notice, a feature can also be collidable (you shouldn't be able to walk through trees) and, using the syntax **`if`** `feature = <FeatureType>` **`then`**, we can specify different logic for different types of features, such as **`if`** `feature = Rock` **`then`** *`//rock logic`*, once again future-proofing our program for extensibility.

Return to `CreateMap()` and add in a call to `GenerateTerrain(result)`. However, if you run the program now, it won't work. In order to draw the map to the screen, we need to both load our resources up and implement a `DrawMap()` procedure.

3 Drawing, Input, and Collision

3.1 Drawing the map

Before we can draw anything to the screen we need to open a SwinGame graphics window and define a main game loop, so let's go to our `Main()` procedure and edit it to look like this:

```
procedure Main();
var
 map: MapData;
begin

 LoadResources();

 // Open a SwinGame graphics window for drawing to
 OpenGraphicsWindow('Procedural Map Generation', 800, 600);

 map := CreateMap(513, true);

 repeat
  ProcessEvents();

  ClearScreen(ColorBlack);

  UpdateCamera(map);
  DrawMap(map);
  DrawSprite(map.player);

  RefreshScreen(60);
 until WindowCloseRequested();
end;
```

We define our main game loop to **`repeat..until`** `WindowCloseRequested()`, which happens when the user clicks to exit the window. Both `LoadResources()` & `UpdateCamera()` are from the `MapUtils.pas` **`unit`** file and do exactly what the say they do, loading resources and making sure the camera is always centred on our player. We call `DrawMap()` to draw our map to the screen which is the procedure we'll be implementing soon, and the SwinGame API's `DrawSprite()` procedure to draw the player to the screen. More SwinGame procedures, `ClearScreen()` & `RefreshScreen()` are called every iteration alongside `ProcessEvents()` for input handling.

Before we implement `DrawMap()`, we need to create a function to check if a particular element is inside the bounds of the tile grid.

```
//
//  Determines whether a given point is inside the tilemap or not
//
function IsInMap(constref map: MapData; x, y: Integer): Boolean;
begin
  result := false;

  // Check map bounds. As every map is (2^n)+1 in size, the bounds
  // stop at High()-1 which will be a number equal to 2^n.
  if (x > 0) and (x < High(map.tiles) - 1) and (y > 0) and (y < High(map.tiles) - 1)
↪  then
  begin
    result := true;
  end;
end;
```

Once we’ve implemented **`IsInMap()`** we can move onto **`DrawMap()`**:

```
//
//  Draws the current map data to the screen but only within the bounds of
//  the current Camera view.
//
procedure DrawMap(constref map: MapData);
var
  x, y: Integer;
  newView: TileView;
begin
  // Get a new tile view to see what should be drawn
  newView := CreateTileView(map);

  // Iterate only tiles in the tile map that correspond to a
  // visible tile
  for x := newView.x to newView.right do
   begin
     for y := newView.y to newView.bottom do
     begin

       if IsInMap(map, x, y) and map.tiles[x, y].hasBmp then
       begin
         // Draw the tile
         DrawBitmap(map.tiles[x, y].bmp, x * map.tilesize, y * map.tilesize);
         // Draw a tree or no feature
          DrawBitmap(map.tiles[x, y].featureBmp, x * map.tilesize, y *
↪  map.tilesize);
       end;

     end;
   end;
end;
```

In our `DrawMap()` procedure we create a new `TileView` record, this is from the `MapUtils.pas` unit file and is used to translate the current camera view in pixels (800×600 in the case of our game) into tile grid coordiantes (25×19). We then make use of the `x`, `right`, `y`, `bottom` **`record`** members it returns to draw whatever tiles are currently within the cameras view to the screen through a nested **`for..do`** loop, also checking that the tile is inside the map bounds and has an assigned bitmap before drawing it. The reason we use a `TileView` record to only draw what's visible rather than the entire map is that, if we were drawing $512 \cdot 512 = 262,144$ tiles to the screen at each game loop, the program would run extremely slow.

3.2 Handling Input & Collision

Now that we've implemented our drawing procedure, we can move onto input and collision detection:

```
procedure HandleInput(var map: MapData);
var
  newX, newY: Integer;
begin

  // Used to determine if they should be allowed to move in a given direction
  newX := map.playerX;
  newY := map.playerY;

  // Change values depending on direction
  if KeyDown(UpKey) then
  begin
    newY -= 1;
  end;
  if KeyDown(RightKey) then
  begin
    newX += 1;
  end;
  if KeyDown(DownKey) then
  begin
    newY += 1;
  end;
  if KeyDown(LeftKey) then
  begin
    newX -= 1;
  end;
```

Here, we pass in our `MapData` variable and save the players current tile-based (as against pixel-based) x and y values into the `newX` & `newY` variables as we'll be altering this data later. Next we use the SwinGame procedure `KeyDown()` to move the players x & y values in the correct direction. Here, we are essentially *predicting* what the players' coordinates will be on the next iteration of the game loop before we *actually* move the player, which is why we aren't assigning these new coordinates to the player directly, rather saving the predicted coordinates in order to do some collision detection:

```
  //  If either newX or newY are outside the map bounds or are a collidable tile,
↪  reset
  //  the players position and don't move them
  if (newX <= 0) or (newX >= High(map.tiles) - 1) or (map.tiles[newX,
↪  newY].collidable) then
  begin
    newX := map.playerX;
  end;
  if (newY <= 0) or (newY >= High(map.tiles) - 1) or (map.tiles[newX,
↪  newY].collidable) then
  begin
    newY := map.playerY;
  end;

  // Assign the new values to the player
  map.playerY := newY;
  map.playerX := newX;

  // Move the player according to world coordinates rather than tile
  // coordinates by multiplying their tile coordinates by the tilesize
  SpriteSetY(map.player, map.playerY * map.tilesize);
  SpriteSetX(map.player, map.playerX * map.tilesize);

end;
```

In the above **if..then** statements we check to see if the players *predicted* x or y values are outside the maps boundary or are on top of a collidable tile. If either of these conditions are true, then we reset either one or both axis' by assigning `newX` **or** `newY` the players original coordinate values. Finally, we assign the new tile-based coordinates, whether they've changed or not, to the player directly and call the SwinGame `SpriteSetX()` & `SpriteSetY()` procedures to actually change the players in-game pixel-based coordinates by multiplying the tile values by the maps `tilesize` value.

3.3 Finishing Up

We're almost done. However, first we need to append our `Main()` & `CreateMap()` procedures with some new code to allow our player to spawn in a correct location and to handle input. Edit `CreateMap()` with the following code:

```
//  Initializes a new tile grid and then generates a new map using DiamondSquare().
//  The new map can be random or based off a given seed
function CreateMap(size: Integer; random: Boolean; seed: Integer = 0): MapData;
var
  i, j: Integer;
  spawnFound: Boolean;
begin
  result.tilesize := 32;
  result.size := size;
  result.player := CreateSprite('player', BitmapNamed('player'));

  // Setup seed
  if random then
  begin
    Randomize;
  end
  else
  begin
    RandSeed := seed;
  end;

  // Initialize Tile Grid
  SetGridLength(result.tiles, size);
  // Generate Heightmap
  DiamondSquare(result, 100, 20);

  GenerateTerrain(result);

  //  Search for the first sand tile without a feature on it, thus spawning the player on a
↪  beach
  spawnFound := false;
  for i := 0 to High(result.tiles) do
  begin
    if spawnFound then
      break;

    for j := 0 to High(result.tiles) do
    begin
```

```
      if spawnFound then
        break;

      // Search for a sand tile to spawn the player on
      if (i > 1) and (result.tiles[i, j].flag = Sand) and (result.tiles[i, j].feature =
 ↪  NoFeature) then
      begin
        SpriteSetX(result.player, i * 32);
        SpriteSetY(result.player, j * 32);
        result.playerX := i;
        result.playerY := j;
        spawnFound := true;
      end;
    end;
  end;

  // Recursively call self with new random value if spawn not found
  if not spawnFound then
  begin
    CreateMap(size, true, seed);
  end;
end;
```

The primary addition here, is to search for the players spawn point. We create a nested `for..do` loop to iterate the tile grid and search for the first sand tile, then assign the players position accordingly; a very simple way to find a spawn point but effective enough for our program. The most interesting section of code in this procedure is also the shortest:

```
// Recursively call self with new random value if spawn not found
if not spawnFound then
begin
  CreateMap(size, true, seed);
end;
```

If the specified seed or random seed value produces a map with only water tiles then we use **recursion** to call `CreateMap()` itself again but with a new random seed. Recursion is an interesting concept in programming and occurs when part of a procedure or function calls itself and runs the same block of code again, but often with new parameters; the return value of which follows the chain of recursion back to the start, itself being recursive by definition (Zwart 2011). Recursive procedures for the most part are interchangeable with loops, and are often more easily followed that way, but some procedures such as `CreateMap()`, lend themselves so naturally to the concept of recursion that it only makes real sense to define them as such. This means that the `CreateMap()` procedure will keep calling itself until it produces a valid map with a valid spawn point.

Finally, edit `Main()` so that it calls the new procedures. We'll also add a little bit of code to delay the players movement so that `HandleInput()` is only called once every 3 frames rather than every frame, otherwise the movement speed of the player would be too fast.

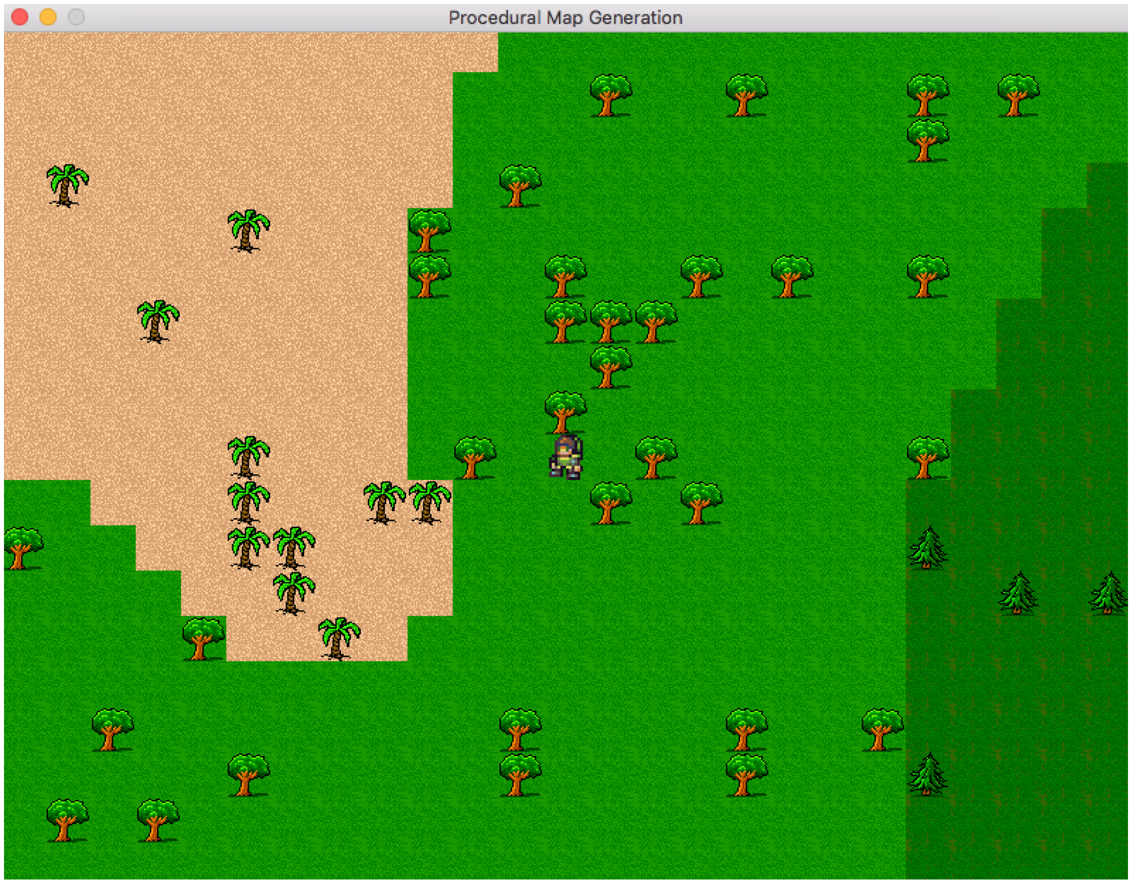
```
procedure Main();
const
  MOVE_INTERVAL = 4;
var
  map: MapData;
  moveDelay: Integer;
begin

  LoadResources();

 OpenGraphicsWindow('Procedural Map Generation', 800, 600);

  map := CreateMap(513, true);

  moveDelay := 0;
  repeat
    ProcessEvents();

    ClearScreen(ColorBlack);

    // Only moves the player at specific intervals so as to
    // limit from running super fast
    moveDelay += 1;
    if moveDelay > MOVE_INTERVAL then
    begin
      HandleInput(map);
      moveDelay := 0;
    end;

    UpdateCamera(map);
    DrawMap(map);
    DrawSprite(map.player);
```

Furthermore, we'll also check if the player has typed escape and then draw a small map overview to the screen until escape is typed again. This way the player is able to find their position on the map at any given moment:

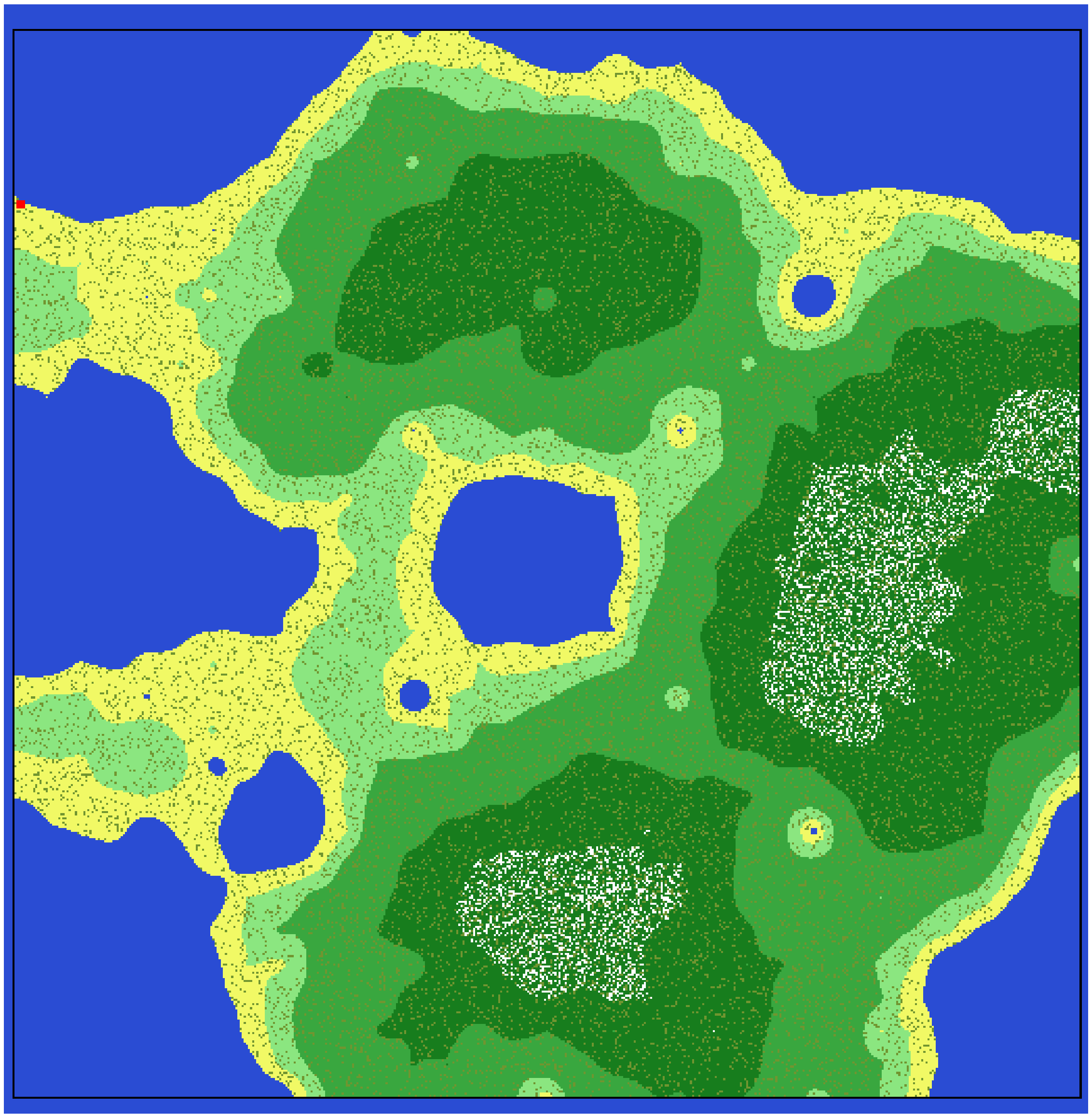
```
    // Create map drawing loop to show player where they are
    if KeyTyped(EscapeKey) then
    begin
      ProcessEvents();
      repeat
        ProcessEvents();
        DrawMapCartography(map); // Draw the map to screen
      until KeyTyped(EscapeKey) or WindowCloseRequested();
    end;

    RefreshScreen(60);
  until WindowCloseRequested();
end;
```

Note, that we've also called `CreateMap()` with a size value of $2^9+1 = 513$ in order to generate our map without hitting any errors. If you want to generate maps of different sizes, just remember they must be of size $2^n + 1$.

Finally, build and run the game and you should have a fully procedurally generated world able to be navigated by the player.



Example 4: The final in-game result



Example 5: The accompanying map overview

4 Extending & Investigating Deeper

Our goal at the beginning of this article was to implement a realistic heightmap generation algorithm and use it to procedurally create terrain representations and features on a 2D tile-based map. Ultimately, this goal has been reached while also touching on other complex problems such as collision detection and efficient drawing procedures alongside making use of an external API in the form of SwinGame.

However, the resulting program is just a starting point and has been built in such a way that it can be extended as needed. This is highly encouraged and there a plethora of algorithms and processes to be discovered and implemented in your program:

- Carving out rivers using a combination of Hart, Nilsson, and Raphaels A* Pathfinding algorithm (Peter E. Hart 1968) to find the shortest path to the ocean or a lake using elevation as the path cost, in combination with Diamond-Square to distort the river paths realistically.
- Producing dense forests via a combination of procedures previously mentioned here and various cellular automata processes
- Generating flora and fauna such as flowers and rabbits by simulating seeds, growth, breeding, and lifespan factors. The possibilities here, in particular, are truly enormous and are where many of the most interesting problems lie.
- Finally, as in the previously mentioned Dwarf Fortress, it's possible to apply narrative generation algorithms to carve out of entire centuries of history and culture, literally creating entire databases of generated lore for a game.

As demonstrated in this article, it's completely within the grasp of the reader to create realistic terrain that's unique and interesting without having to create all of the content manually. The possibilities are endless and only limited by your imagination.

References

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

Güneysu, T. (2010). "True random number generation in block memories of reconfigurable devices". In: *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, pp. 200–207. DOI: 10.1109/FPT.2010.5681499.

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).

PCG (2016). *Procedural Content Generation Wiki*. URL: http://pcg.wikidot.com/.

Peter E. Hart Nils J. Nilsson, Bertram Raphael (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107.

West, Mick (2008). *Random Scattering: Creating Realistic Landscapes*. URL: http://www.gamasutra.com/view/feature/1648/random_scattering_creating_.php.

Zwart, Jan-Wouter (2011). "Recursion in Language: A Layered-Derivation Approach". In: *Biolinguistics* 5.1-2, pp. 43–56.

Source Code & Repository

The complete finished project directory and all source code files referenced in this article can be downloaded from the repository listed on github.com by cloning or forking the project via git, or simple navigating to the 'clone or download' button and downloading as a .zip file. The repository includes all resource files alongside `MapUtils.pas`. If you have find any issues or bugs in the code provided, feel free to open a new issue on the repository page or fork and contribute any fixes or additions.