Swinburne University of Technology
Faculty of Science, Engineering, and Technology

COS20007: Object Oriented Programming

Research Report: Midpoint Displacement Algorithm

Date of report submission	11/06/19
Lab Supervisor	Medhi Naseriparsa
Group	10:30am Wednesday

Name	**Student ID**
Jimmy Trac	101624964

This page is mostly blank.

The purpose of this page is to separate the cover page from the report when printed double-sided.

Midpoint Displacement

Research report (version two), written by Jimmy Trac (101624964)

I. Preface

This report is designed to provide short and concise insight into the implementation of the Midpoint Displacement Algorithm as well as the implementation of the algorithm within all iterations of Artillery. Notably, this document was written for the Artillery 3s (three-S) branch, hence, some details may be different to other branches. This is prevalent in the location and design of the model from 3R to 3Rx, which are both parents of 3s.

This report may contain instances of the name *Mikan*, it is a reference to an online alias (username) and is the author of this report. Other forms include *Mikanwolfe* and *Mikan Lumi*.

This document borrows off the design document for the original Artillery, and therefore, may incur some similarities with services such as TurnItIn.

II. Table of Contents

`

I. Abstract

A quick and concise overview of terrain generation and implementation in Artillery.

Destructible terrain within games is a useful and tactical mechanic that gives certain game genres the ability to create effective player-versus-player combat. Artillery describes terrain in a one-dimensional array of heights that is randomly generated on a per-round basis both in the logical foreground terrain and the background for aesthetics. This provides realistic terrain that is both immersive and provides a unique playthrough each round. The successful implementation of the Midpoint Displacement Algorithm, also known as Diamond-Square or the Recursive Subdivision Algorithm, provides a simple and effective method of terrain generation within Artillery that produces fractal terrain with a dimension based on the provided displacement reduction coefficient.

I. a. Keywords

Terrain Generation, Midpoint Displacement Algorithm, Diamond-Square Algorithm, Recursive Subdivision Algorithm, Platformer, Game.

I. b. Acknowledgements

I would like to give my heartfelt thanks for all the friends that have supported me throughout the development of Artillery3 and its derivatives. The amount of time I've sunk into this is easily measurable based on the impact it has had on my other units, hence, I would both like to apologise and extend my thanks to group members for their support during the development of A3.

A special thanks goes out to Bi Wan Low for bringing me back into reality and changing the colour theme from a garish green/blue to the much more aesthetic snow theme that exists within A3s. It would have not been possible without her spark of genius.

II. Terrain Generation

A literature review of destructible terrain and the implementation within Artillery.

The use of terrain generation within the Artillery Series, including both Artillery 1, 2, and all iterations of Artillery 3, is twofold: two produce realistic-looking terrain, and to provide unique terrain on a per-session basis. This report, however, will not go into detail on the design goals and decisions made within Artillery 3 (A3), that will be covered in a separate design document.

2.1. Destructible terrain within games

The Artillery Series relies heavily on damageable terrain as game mechanic, and therefore requires extra consideration when it comes to the maps and terrain in which A3 is based on. There are two major types of destructible terrain discussed within this document: Bitmasks and Tile maps.

Bitmask terrain is terrain generated from pre-made images where the image is treated as terrain and can be destroyed by removing them from the game. A good example of this is within old titles such as *Worms (Team 17 Digital Limited, 1995)*, where players can destroy and manipulate terrain to their advantage, such as hiding behind obstacles or destroying them. (Monteiro, 2012), or in *GunBound SoftNyx, 2003)*, where Artillery derives a lot of concepts as seen in Figure 2.1.

Figure 1.1 – Screenshot of GunBound, showing the terrain is an image that can be destroyed. (MMOBomb, 'Gunbound')

The second type of destructible terrain is a Tile Map, which consists of a grid of "tiles" that can be destroyed and constructed by players. Seeing extensive use in survival-type games such as *Terraria (Re-Logic,* 2011*)* or *Starbound (Chucklefish,* 2016*)*, tile-maps offer the player flexibility when it comes to how they want to design or destroy the terrain as the 'resolution' of these tiles are often quite low. Compared to *Worms* where each individual pixel can be destroyed, the tiles used in *Starbound* and *Terraria* are one-third the height of the player, leaving very blocky-looking terrain. This is, for the most part, a variable affected by the design of the game and player usability, alongside performance. As the size of each individual tile is reduced, the number of tiles is increased by the square of the side length, meaning that halving tile size increases the number of tiles required by four. The benefits of a tile map is the ability to specify the type of tile. On the implementation level, the performance of tile maps can be reduced using the *Flyweight* (Nystrom, 2014) design pattern.

Due to time and design requirements, Artillery does not implement either of the above systems. Owing to the simple nature of Artillery, the Artillery Series implements a different and much more simplistic form of destructible terrain, allowing for the use of the Midpoint Displacement Algorithm as a natural and simplistic way of generating aesthetically pleasing terrain.

2.2. Artillery3's Implementation of Terrain

Artillery3 uses a simplified terrain method which consists of treating the terrain as a one-dimensional array of "height" values, effectively creating a "line of terrain" which acts as the terrain the players are positioned on. This is useful as it simplifies several aspects of terrain generation which would have increased the complexity significantly, such as mapping terrain generation to a tile map or the implementation of bitmasks. There are limitations of this system, such as the inability to have "caves" or "roofs" as there can only be one height value for each x-position.

A3 utilises the Midpoint Displacement Algorithm (MDA), also known as the Diamond-Square Algorithm or in the original 1982 SIGGRAPH paper, the *Recursive Subdivision Algorithm* (Fournier et al., 1982). For the sake of simplicity, this report will refer to the algorithm as the Midpoint Displacement Algorithm due to the descriptiveness of the name The MDA produces fractal terrain which closely resembles the stochastic (read: can be analysed but not predicted; random) terrain present in the real world, particularly coastlines. This report will focus on a slightly more modern and less boring interpretation of the MDA.

The pseudocode is as follows:

```
Draw a line across the screen

Repeat for a sufficiently large amount of times
   For each segment
      Calculate the midpoint
      Displace the midpoint between a random range
   Reduce random range
End
```

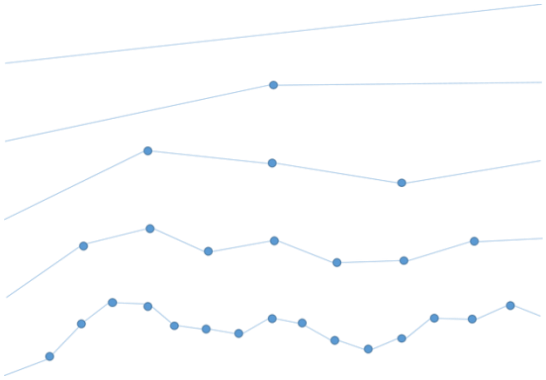


Figure 2.2.1 - Successive iterations of the Midpoint Displacement Algorithm (Gallostra, 2016)

Figure 2.2.2 - Terrain generated using the Midpoint Displacement Algorithm (Gallostra, 2016)

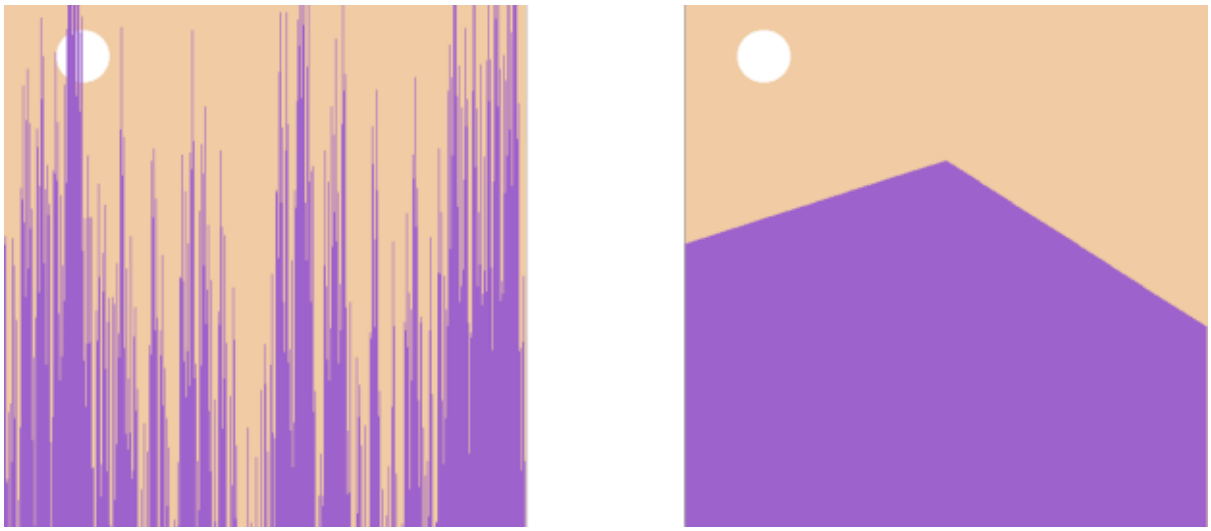


Figure 2.2.3 - Left: No displacement reduction. Right: Displacement reduction set to 0 after first iteration (Gallostra, 2016)

Figure 2.2.1 is a visual representation of 4 iterations of the MDA performed on a line. Despite the simplicity, there are two main factors that influence the appearance of the generated landscape: initial displacement and the reduction function used to reduce the displacement range at each iteration. Figure 2.2.2 shows the application a linear reduction at various displacement reduction coefficients to show the varied terrain produced, whilst the two figures of 2.2.3 show the extreme ends of the displacement reduction, being 0 and 1 respectively (Gallostra, 2016).

2.3 Implementation-Level MDA

Whilst the original MDA program was written in Pascal, the reduction function and the general complexity of the code was rather high. Hence, Artillery utilises a cut-down method to generate terrain based on points within an array rather than conceptual lines.

The implementation utilised within the Artillery Series has some requirements: there must be 2^n points for n iterations of the algorithm within the array and the first and last numbers must be present for the algorithm to successfully iterate. Due to time constraints, this is not the exact definition utilised, however, the function simply contains a few failsafe measures. The pseudocode for the MDA within the original Artillery is as follows:

Code 2.3.1 – The pseudocode for the MDA

```
Constant: SCREEN_WIDTH, SCREEN_HEIGHT
Height-Map: One-dimensional array of numbers.
D: Displacement maximum value, number.

Terrain Width = A power of 2 that is greater than SCREEN_WIDTH + 1
Set random heights for the first and last point


For a number, I = 0, to 2^I = Terrain-Width
   Number-Segments = 2^I
   Segment-Length = Half the length of the full segment

   For each segment
      Current-Index = Midpoint of current segment
      Height-Map [Current-Index] = Average of previous and next midpoint
      Height-Map [Current-Index] += Random value between [-D, D]
   End

   Reduce Displacement
End
```

Conceptually, the algorithm treats each two-point combination as a line segment. As there are 2^n segments, each division into two will contain the same distance on either side the midpoint. Taking an example of 1024 points (and hence, 10 iterations), the first iteration will split the line into one with 512 points on either side the middle point 511. In practice, there are some issues with the definition of counting, hence, the actual length of the array is set to $2^n + 1$ to allow for 512 points on either side of the now-not-included middle 511th point.

The midpoint is then displaced, and the algorithm iterates and there are now two segments to split—hence, the number of segments is defined by the first FOR loop for 1, 2, 4, 8, 16… segments. The second FOR loop iterates through each segment and finds the midpoint of each segment and averages the value to linearize the three points and proceeds to displace the midpoint by the range reduction.

One technical note is finding the next value of 2^n above the designated size of the terrain. This can be found by taking the ceiling of $Log_2 w$, where w is the current width of the terrain.

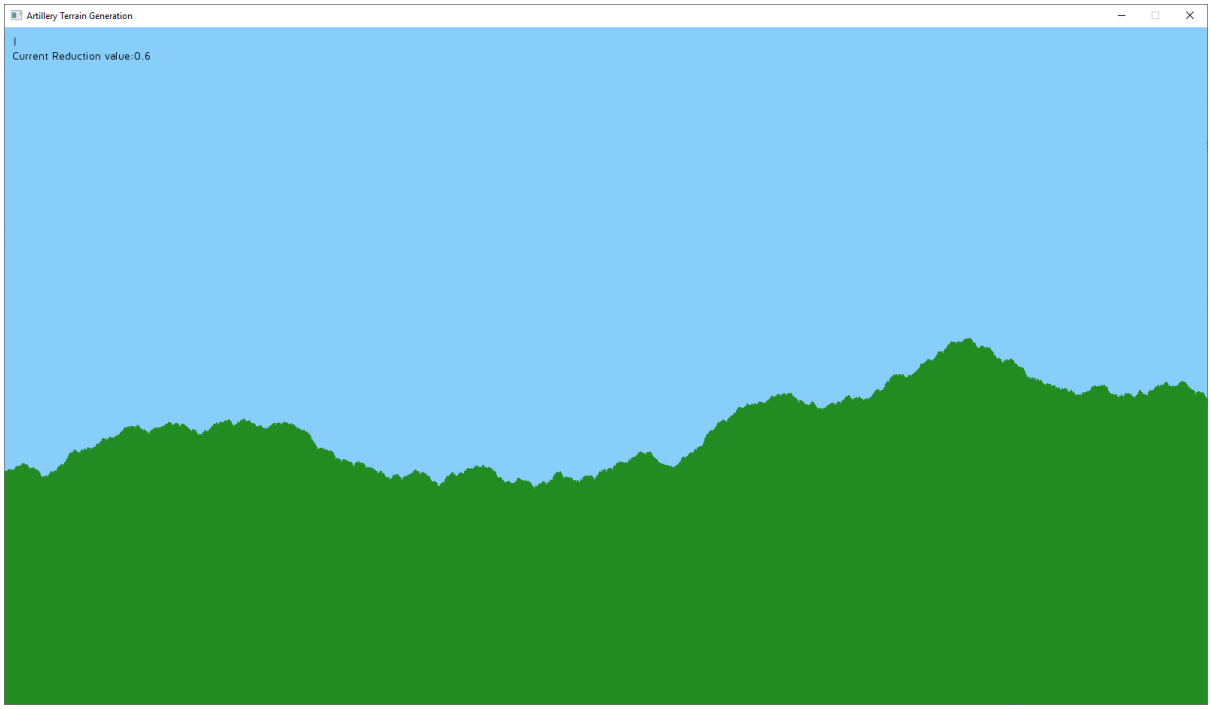
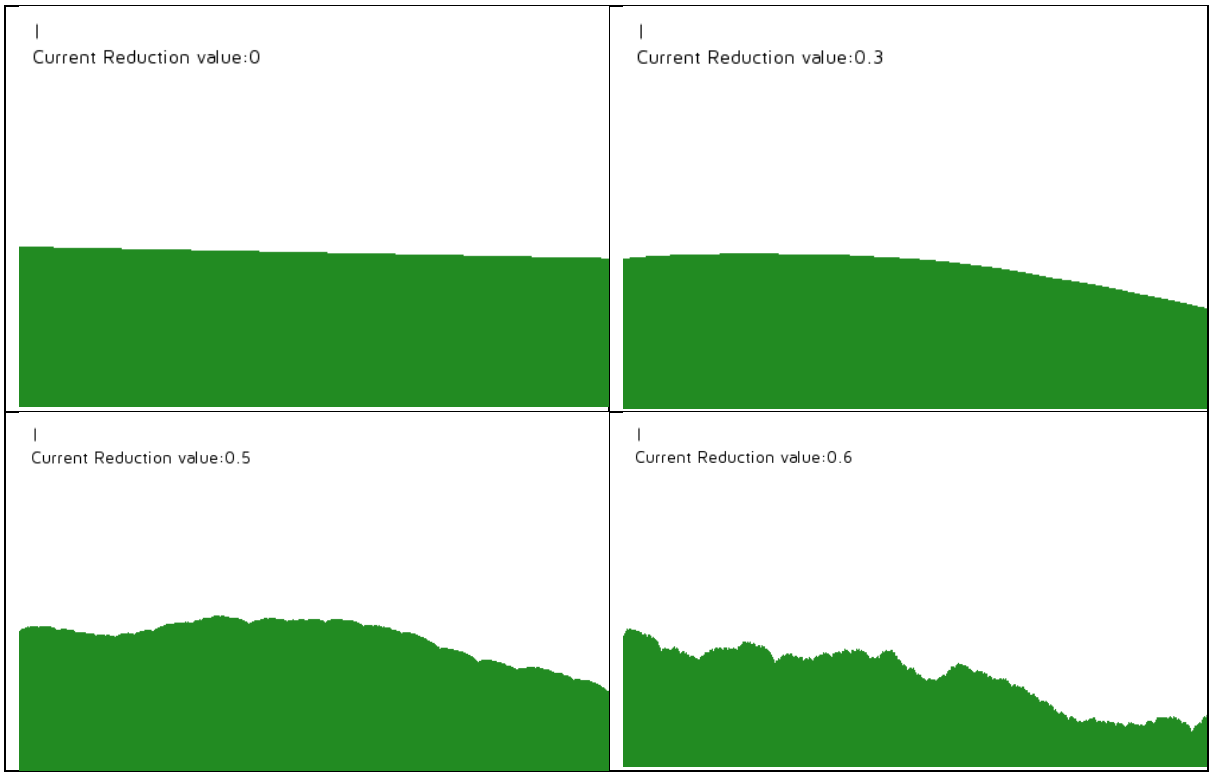
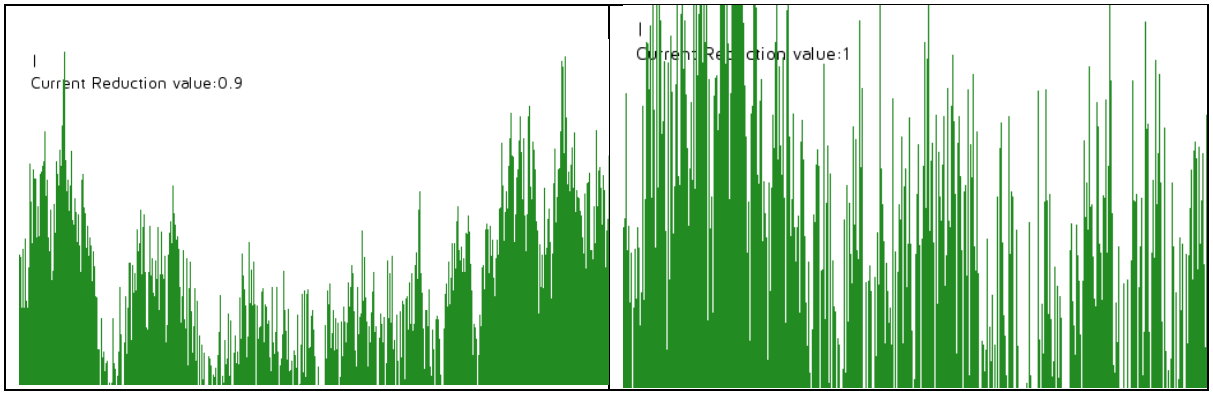


Figure 2.3.1 – Implementation of MDA with a reduction of 0.6

The implementation within C# can be found within the online repository once it has been released. The algorithm implemented is similar to the original implementation within Pascal and can be found within Code 2.3.2. Figure 2.3.1 shows terrain that is rough but believable when framed in the context of a mountain range. Further tests of different reduction coefficients can be found in Table 2.3.3.

Table 2.3.3 – Terrain Reduction at various reduction coefficients





Code 2.3.2 – The C# Implementation of the MDA

```
int requiredExponent = PowerCeiling(2, TerrainBox.Width);
            int requiredWidth = (int)Math.Pow(2f, requiredExponent); //Should be 2^x
- 1. e.g. 0..1024
            float[] generatedMap = new float[requiredWidth + 1];
            int numberOfSegments;
            int segmentLength;
            int xVal;
            float displacement = 200;

            /* -- Terrain Generation, Start! -- */

            /* -- Generating the starting and the ending points -- */
            generatedMap[0] =
                _averageTerrainHeight + RandDisplacement(50);
            generatedMap[requiredWidth] =
                _averageTerrainHeight + RandDisplacement(50);

            for (int step = 0; step <= requiredExponent; step++)
            {
                numberOfSegments = (int)Math.Pow(2, step); //2, 4, 6, 8, 16...
                segmentLength = requiredWidth / numberOfSegments;

                for (int i = 1; i <= numberOfSegments; i++)
                {
                    //Increment through each segment
                    xVal = i * segmentLength - (segmentLength / 2);
                    generatedMap[xVal] = (generatedMap[xVal - (segmentLength / 2)] +
generatedMap[xVal + (segmentLength / 2)]) / 2;
                    generatedMap[xVal] += RandDisplacement(displacement);
                }
                displacement *= ReductionCoef;
            }
```

2.4. Serialising Terrain

The use of a simple array of floats implies that the serialisation and deserialization (read: saving and loading) of terrain is rather straightforward. Toward this end, Artillery3X and the Artillery Terrain Generation (ATG) example both employ NewtonSoft.JSON, a library for C# that facilitates the de/serialisation of objects into JSON strings.

The reason for the use of JSON over XML relates to the readability of JSON over XML. Furthermore, NewtonSoft.JSON is a straightforward implementation of JSON serialisation which converts objects into primitive strings. Further research into this area and a test program with XML may move interesting as the currently implementation does not fully serialise objects such as SwinGame Rectangles.

Code 2.4.1 is an example of a saved terrain file from ATG with the actual file containing all 560 entries for the height of each pixel.

Code 2.4.1 – Reduced example to show the general layout of a serialised JSON file.

```
{
   "Map":[
      188.0,
      186.8556,
      186.88559,
      186.831818,
      187.699142,
      187.069244,
      187.360443,
      187.5679,
      188.696442,
      187.99736,
      188.21936,
      188.3576,
      189.416946,
      ...
      215.768036
   ],
   "Color":{
      "R":34,
      "G":139,
      "B":34,
      "A":255
   },
   "Pos":{
      "X":0.0,
      "Y":0.0
   },
   "WindowRect":{
      "X":0.0,
      "Y":0.0,
      "Width":540.0,
      "Height":300.0,
      "Bottom":300.0,
      ...
   }
}
```

2.4.1. Serialising Objects

The serialisation of terrain is derived off A3X's use of serialisation and deserialization of A3 elements, such as constants and what was originally intended. The limited application of the MVC model highlighted it's usefulness when it came to the reading and writing of model information as it was all contained in primitive types; that is—the NewtonSoft JSON serializer is not necessarily built to handle SwinGame objects nor unique A3 objects, however, as most objects are constructed from primitive types, it holds to reason that with a generous amount of time and the correct implementation of a A3 Model, the serialisation and deserialization of A3 as loading and saving is possible. This weighs in favour of A3X, A3R, and A3S, which all utilise the pseudo-MVC design. Code 2.4.1.1 shows the current implementation of Constants within a JSON file

Code 2.4.1.1 – The current implementation of constants as a deserialize-able JSON file

```
{

  "DataFile" : "data.json",
  "WindowWidth" : 1600,
  "WindowHeight" : 900,
  "Gravity" : 0.6,
  "CameraEaseSpeed" : 10,
  "CameraPadding" : 100,
  "CameraMaxHeight" : 5000,
  "TerrainWidth" : 3000,
  "TerrainHeight" : 5000,
  "TerrainReductionCoef" : 0.50,
  "AverageTerrainHeight" : 600,
  "TerrainBoxPadding" : 100,
  "BaseTerrainInitialDisplacement" : 100,
  "NumBgTerrain" : 3,
  "WeaponChargeSpeed" : 0.5,


  "InitString" : "Hello, World! -- From settings.json"

}
```

2.5. Conclusions

Within the final iterations of A3, notably A3s, the terrain is used both for aesthetic strategic purposes. The terrain, with its hills and valleys, produces somewhat difficult to manoeuvre terrain that can restrict one player's ability to fire at another, and vice-versa.

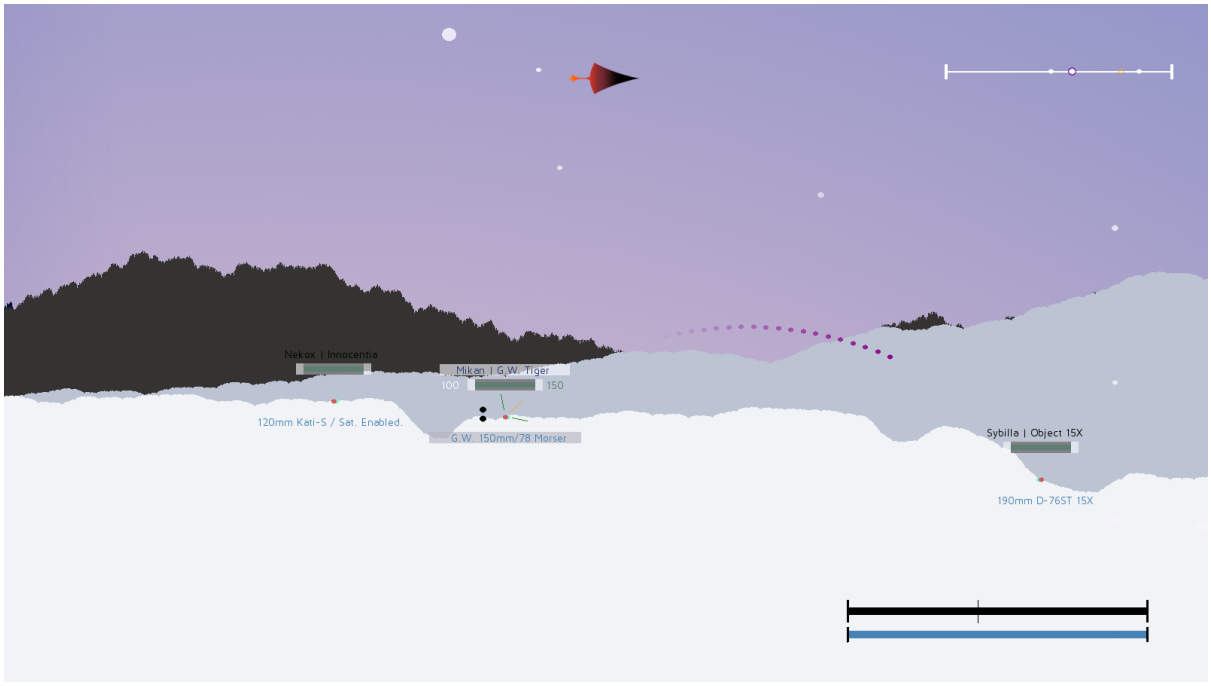


Figure 2.4.1 – An example of a player using terrain within A3s

Figure 2.4.1 shows the rightmost player taking advantage of being on the bottom of a hill to reduce the chance they themselves would be hit.

The advantage of MDA is that it allows sufficiently realistic and believable terrain generation that is both straightforward to implement in A3 due to the nature of the terrain being a one-dimensional array. The flexibility in terrain generation allows A3 to produce a relatively interesting game experience whilst not feeling particularly repetitive.

3. References

Worms 1995 [Game], Team 17 Digital Limited. < https://www.team17.com/>

GunBound 2003 [Game], SoftNyx. <http://www.softnyx.net/>

Monteiro, R. *The guide to implementing 2D platformers*, Higher-Order Fun: http://higherorderfun.com/blog/2012/05/20/the-guide-to-implementing-2d-platformers/, Last Accessed 03/06/2019.

'Gunbound', in MMOBomb, viewed 09 June 2019, < https://www.mmobomb.com/review/gunbound>

Terraria 2011 [Game], Re-Logic. < https://terraria.org/>

Starbound 2016 [Game], Chucklefish. < https://playstarbound.com/>

Nystrom, R 2014, *Game Programming Patterns*, Genever Banning.

Fournier, A., Fussell, D. and Carpenter, L., 1982. Computer rendering of stochastic models. *Communications of the ACM*, *25*(6), pp.371-384.

Gallostra, J 2016, 'Landscape Generation using midpoint dispacement', Bites of code, Viewed 03 June 2019, < https://bitesofcode.wordpress.com/2016/12/23/landscape-generation-using-midpoint-displacement/>