# Procedural Methods

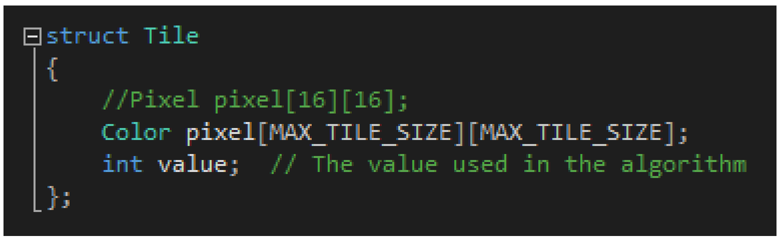
## Outline

This program's main feature is a procedural level generator that uses markov chains to create content for tile based games. The program uses a level from the game to learn about likely and repeated patterns, as well as what does not appear at all in the levels. Using this information it generates a new level and prints it to a PNG file.

This program also features a plane that displays smoothed perlin noise [9] and a gaussian blur post processing effect.

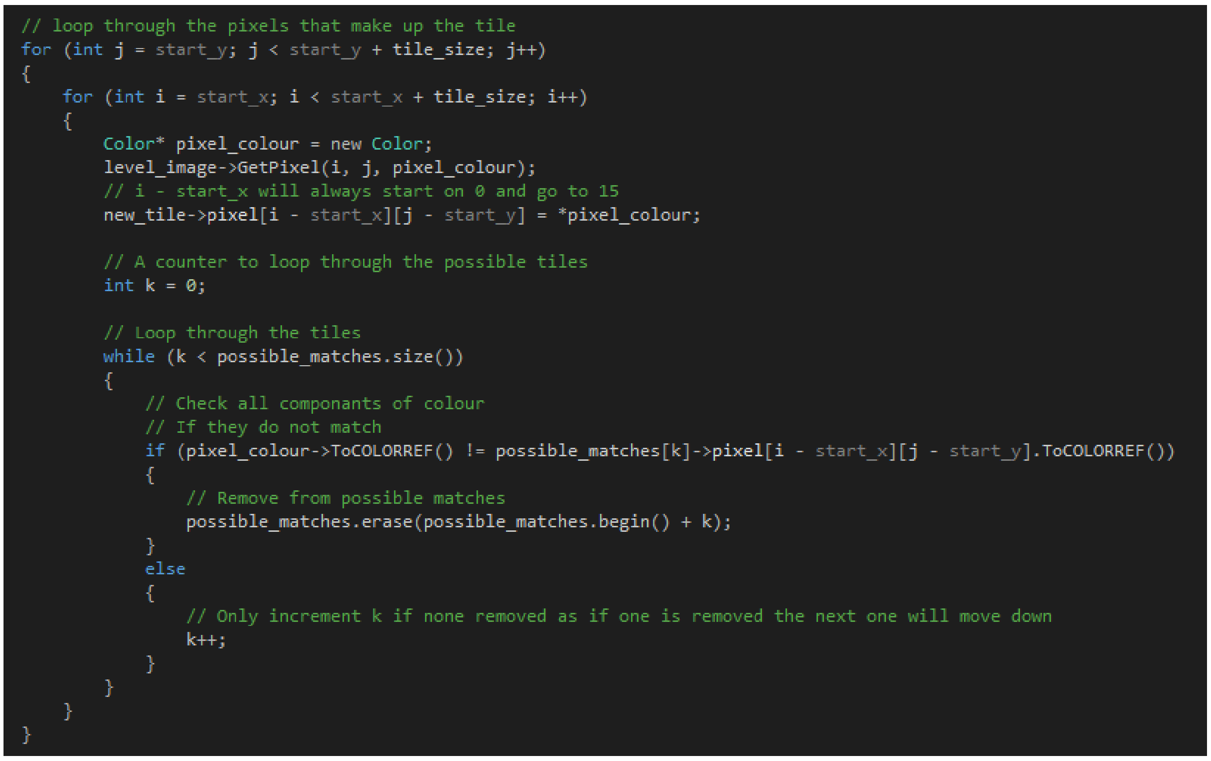
## Description of features

**Markov chain level generator -** The program's level generator takes a level from an existing game as input. This image must only contain the tiles that make up the level, not any sprites of items or enemies. The program uses the GDI+ library [10] to analyse the image once it's loaded in. The program contains Level 1-1 of "Kid Icarus" [1], Level 1-1 of "Super Mario Bros" [2] and the overworld map from "The Legend of Zelda" [3]. A custom image can be loaded in if a PNG is added to the "Levels" folder with the name "custom". The program must know the tile size of this image and then each tile can be passed to a function to be sorted.



```
struct Tile
{
    //Pixel pixel[16][16];
    Color pixel[MAX_TILE_SIZE][MAX_TILE_SIZE];
    int value;  // The value used in the algorithm
};
```

This function loops through all of the pixels in the input tile and checks them against the corresponding pixel in all known tiles (At the beginning there is no known tiles). If this pixel is different in any of the known tiles, the program stops checking them as it knows this tile is not the same as them. When the function is finished checking the tile there will either be one possible match left (meaning this tile is identical to it) or there are none left (meaning this is a new tile. If the tile is new it is added to the list of known tiles and is given a new tile value (an integer number equal to the tile's position in the vector of known tiles) and if it is an instance of an existing tile it is given the tile value of that tile. These tile values are written to a 2D array of integers representing the tiles of the level.



```
// loop through the pixels that make up the tile
for (int j = start_y; j < start_y + tile_size; j++)
{
    for (int i = start_x; i < start_x + tile_size; i++)
    {
        Color* pixel_colour = new Color;
        level_image->GetPixel(i, j, pixel_colour);
        // i - start_x will always start on 0 and go to 15
        new_tile->pixel[i - start_x][j - start_y] = *pixel_colour;

        // A counter to loop through the possible tiles
        int k = 0;

        // Loop through the tiles
        while (k < possible_matches.size())
        {
            // Check all componants of colour
            // If they do not match
            if (pixel_colour->ToCOLORREF() != possible_matches[k]->pixel[i - start_x][j - start_y].ToCOLORREF())
            {
                // Remove from possible matches
                possible_matches.erase(possible_matches.begin() + k);
            }
            else
            {
                // Only increment k if none removed as if one is removed the next one will move down
                k++;
            }
        }
    }
}
```

The probabilities are found using a two dimensional markov chain (Based on a paper by Sam Snodgrass and Santiago Ontanon [6] [7]). This approach calculates the probability of a tile being any particular state given the state of the tile to its left, above it and to its top-left.

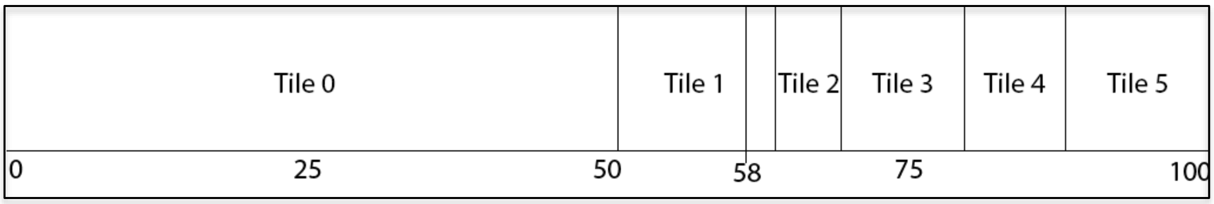
$$\begin{tikzcd}
S_{1,1} \arrow[r] \arrow[d] \arrow[dr] & S_{2,1} \arrow[r] \arrow[d] \arrow[dr] & S_{3,1} \arrow[r] \arrow[d] \arrow[dr] & S_{4,1} \arrow[d] & \cdots \\
S_{1,2} \arrow[r] \arrow[d] \arrow[dr] & S_{2,2} \arrow[r] \arrow[d] \arrow[dr] & S_{3,2} \arrow[r] \arrow[d] \arrow[dr] & S_{4,2} \arrow[d] & \cdots \\
S_{1,3} \arrow[r] & S_{2,3} \arrow[r] & S_{3,3} \arrow[r] & S_{4,3} & \cdots \\
\cdots & \cdots & \cdots & \cdots &
\end{tikzcd}$$

$$P(S_{t,r} \mid S_{t-1,r}, S_{t,r-1}, S_{t-1,r-1})$$

(Taken from the paper by Sam Snodgrass and Santiago Ontanon [6])

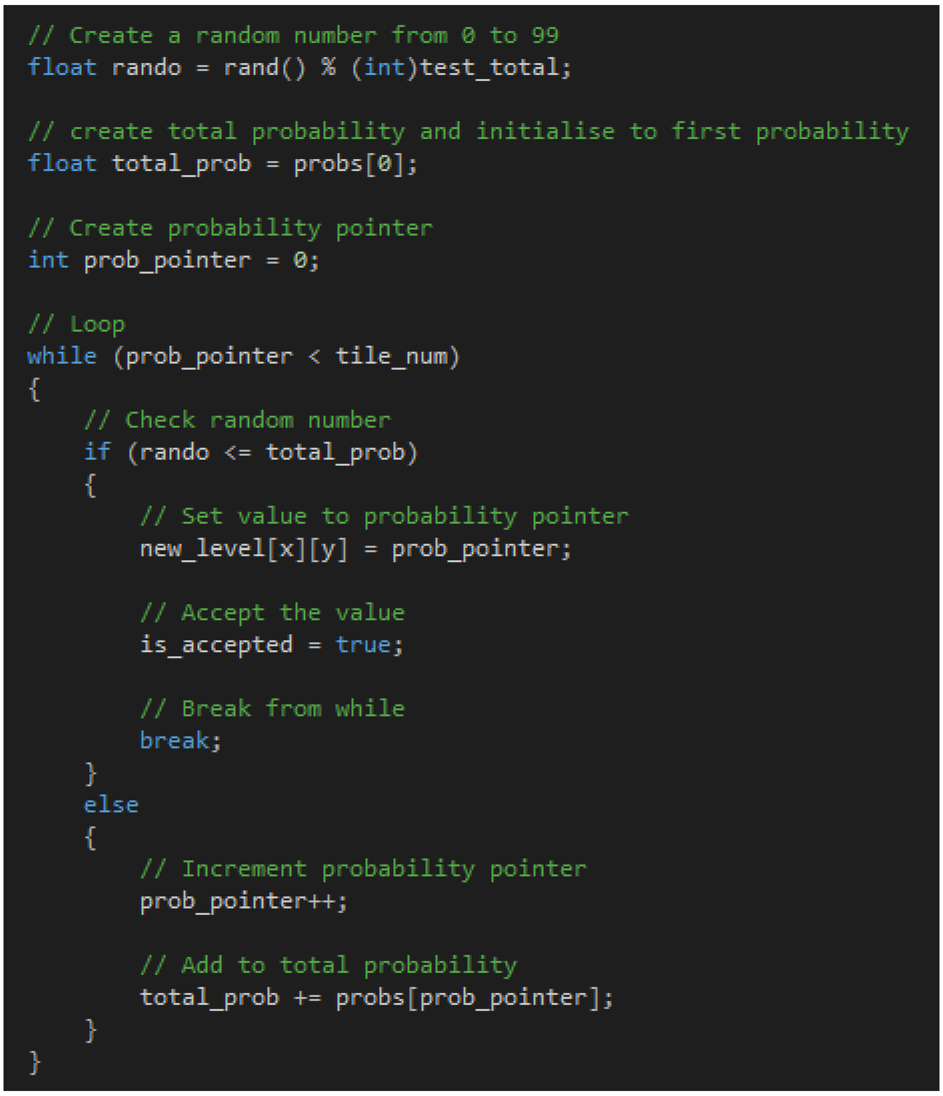
These probabilities are found for every tile and every arrangement of three tiles that could be around it.

Using these probabilities the program generates another 2D array of tile values, with the same width and height as the input level. The left and top edge are initialised to zero to give the program something to start from and then loops through the remaining tiles. A random number is generated between 0 and 99 and that number decides on which tile is chosen (tile 0 is N%, tile 1 is tile 0 + N% etc.)

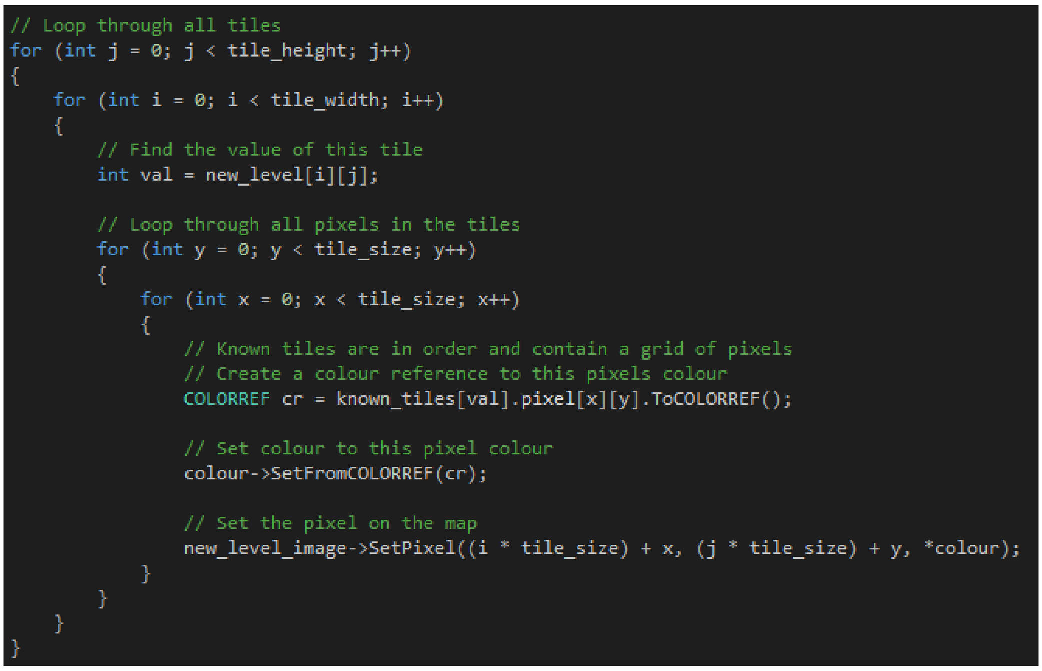


(A random value of 58 would select tile one as 58 is less than probability 1 + probability 0)

In some cases the program will find a combination of three tiles surrounding the target tile that was not present in the input level. These combinations are considered impossible as the formation may not exist because it would not make sense in the game. In these cases the program goes back to the tile one above and one left of the target tile and does the random generation again. This repeats until all tiles have been given a value and all arrangements are considered possible.


```
// Create a random number from 0 to 99
float rando = rand() % (int)test_total;

// create total probability and initialise to first probability
float total_prob = probs[0];

// Create probability pointer
int prob_pointer = 0;

// Loop
while (prob_pointer < tile_num)
{
    // Check random number
    if (rando <= total_prob)
    {
        // Set value to probability pointer
        new_level[x][y] = prob_pointer;

        // Accept the value
        is_accepted = true;

        // Break from while
        break;
    }
    else
    {
        // Increment probability pointer
        prob_pointer++;

        // Add to total probability
        total_prob += probs[prob_pointer];
    }
}
```

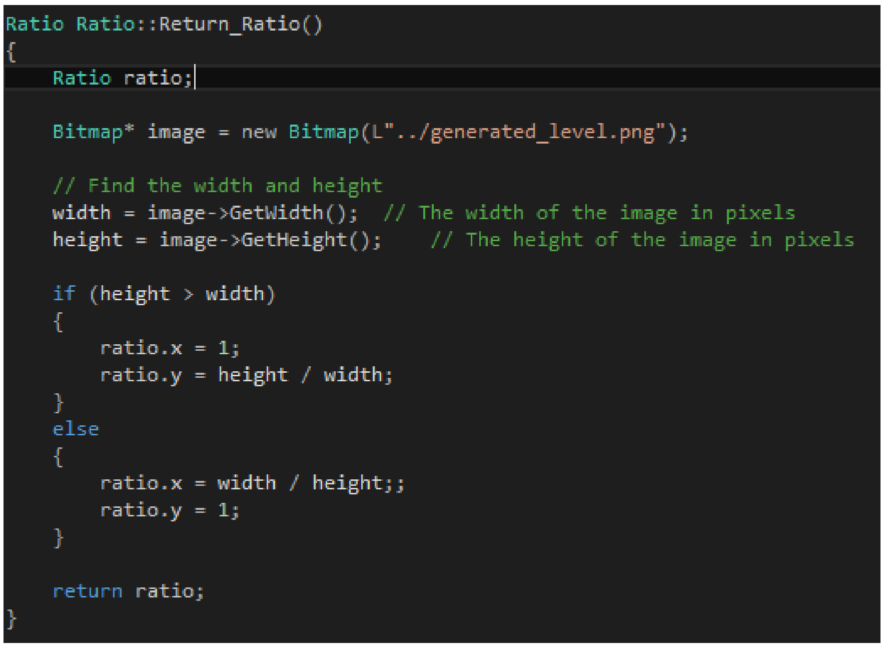
The Final thing this part of the program does is write the procedurally generated level to a PNG image. This is done using GDI+ [10] also. The program loops through every integer in the 2D array of tile values and then a nested for loop goes through each pixel that makes up that tile. The pixels that make up the tiles are stored in the vector of known tiles and can be found using the tile values as this is their position in the vector.



```
// Loop through all tiles
for (int j = 0; j < tile_height; j++)
{
    for (int i = 0; i < tile_width; i++)
    {
        // Find the value of this tile
        int val = new_level[i][j];

        // Loop through all pixels in the tiles
        for (int y = 0; y < tile_size; y++)
        {
            for (int x = 0; x < tile_size; x++)
            {
                // Known tiles are in order and contain a grid of pixels
                // Create a colour reference to this pixels colour
                COLORREF cr = known_tiles[val].pixel[x][y].ToCOLORREF();

                // Set colour to this pixel colour
                colour->SetFromCOLORREF(cr);

                // Set the pixel on the map
                new_level_image->SetPixel((i * tile_size) + x, (j * tile_size) + y, *colour);
            }
        }
    }
}
```

The program then finds the class ID of a PNG using the “GDIHelperFunction.h” [10] file and uses this to output the new level as “generated_level” to be used in the main application.

The PNG of the generated level is loaded into the shaders framework as a texture and then applied to a quad. The quad is scaled either horizontally or vertically depending on if the image is wider than it is tall or vice versa.



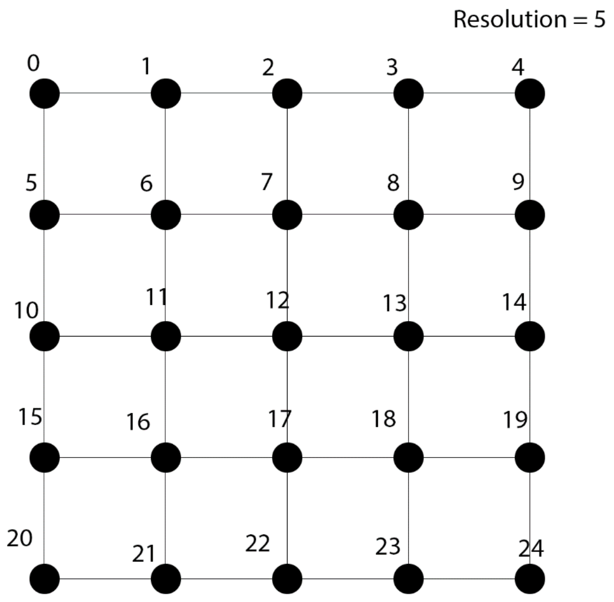
```
Ratio Ratio::Return_Ratio()
{
    Ratio ratio;

    Bitmap* image = new Bitmap(L"../generated_level.png");

    // Find the width and height
    width = image->GetWidth();  // The width of the image in pixels
    height = image->GetHeight();    // The height of the image in pixels

    if (height > width)
    {
        ratio.x = 1;
        ratio.y = height / width;
    }
    else
    {
        ratio.x = width / height;;
        ratio.y = 1;
    }

    return ratio;
}
```

This means that the quad will always be scaled to show the image at the correct aspect ratio and large enough to be seen properly.

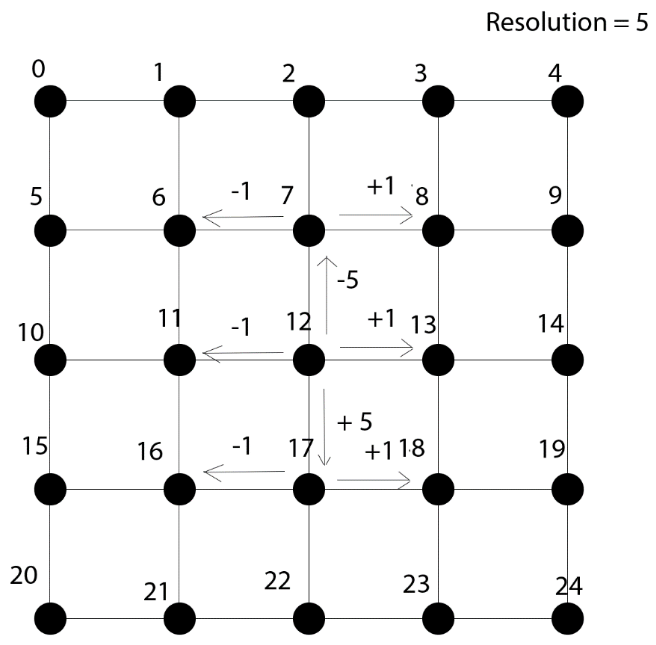
**Smoothed perlin noise plane -** A plane is generated using a modified version of the code that the shaders framework provides to create a plane.

Ken Perlin's improved noise class [9], which is written in java, was modified to work in C++. This was first created and integrated in the rastertek framework, however it was created to be easily integrated into any system.

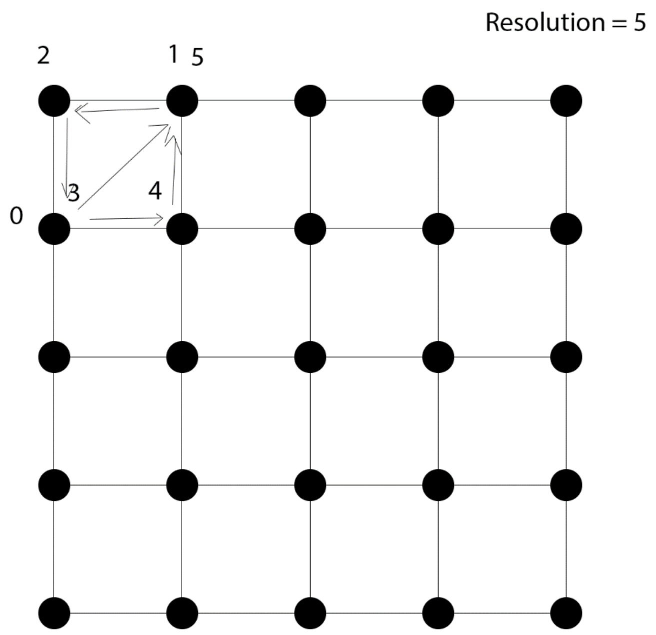
Perlin noise and a smoothing function was first implemented in the rastertek framework in which the plane was stored as a one dimensional array of points (starting at the top right and moving horizontally through each row or vertices).



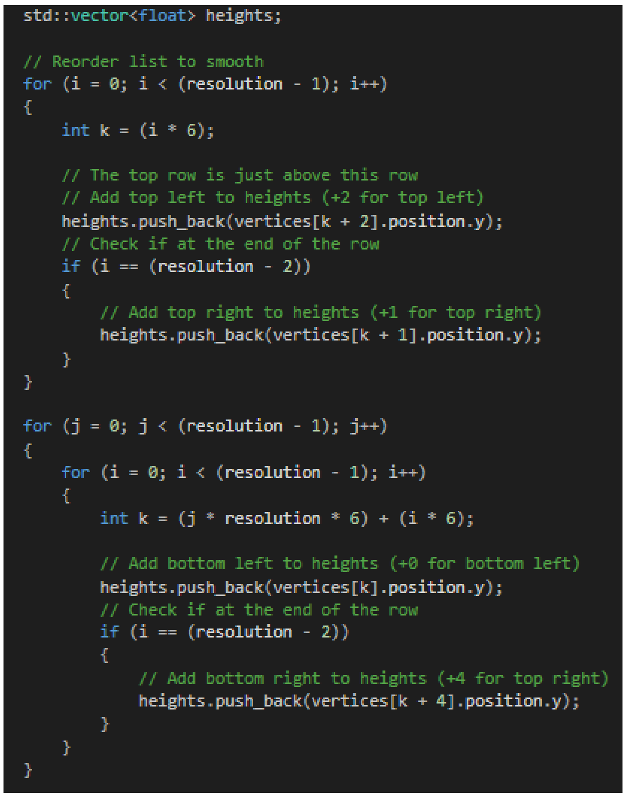
The smoothing function was designed for this. The smoothing function sets the height of each point and the average height of the eight points surrounding it. These points are easy to locate given the arrangement of the plane as the left and right point are just one unit above and below in the array. The top and bottom are one row away making them equal to the point's value plus and minus the resolution of the plane. The corner points are just one above and below the top and bottom points in the array.



The shaders framework stores the plane in a very different way. Each square on the plane is stored as six points (The vertices of two right angle triangles) and so there are multiple instances of each point in the array of vertices.

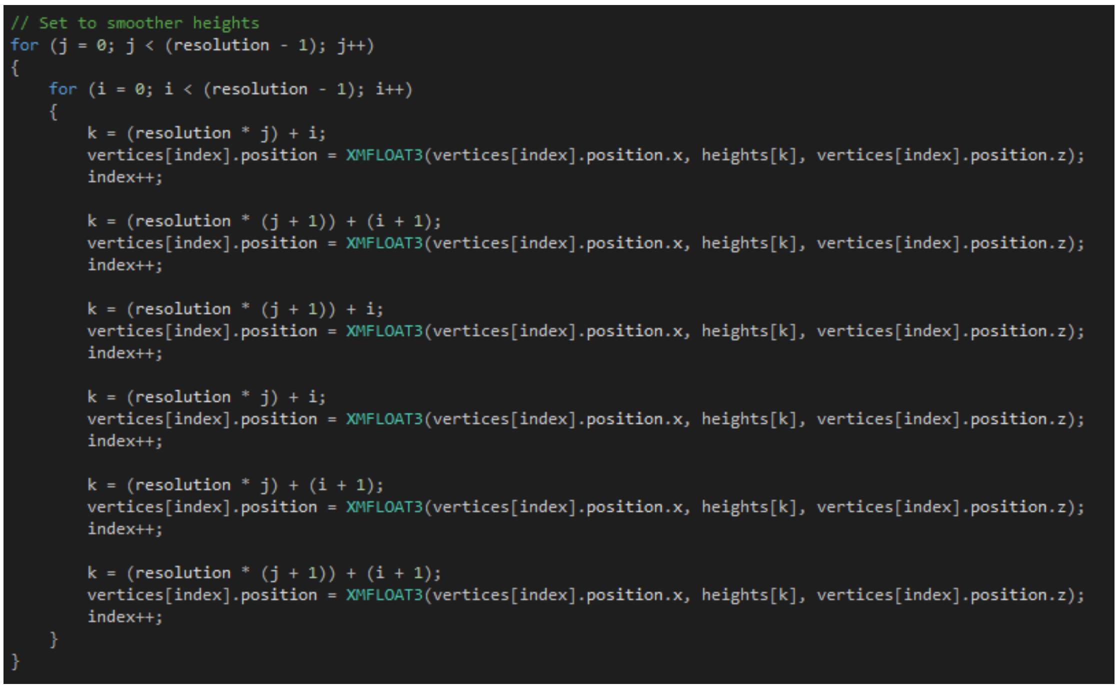


As there are more points in the vertices array than there are visible points on the plane, a vector was created and a formula was made to add only one instance of each point’s height to this vector in the same order that they are stored in the array in the rastertek framework.



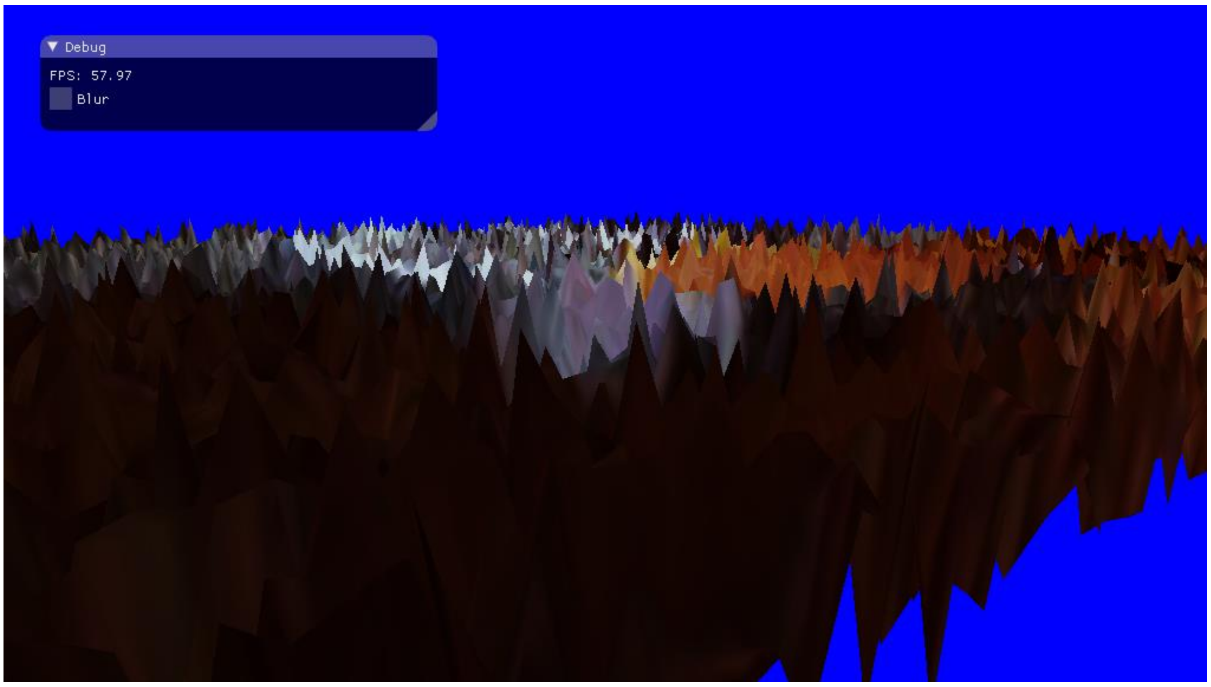
```
std::vector<float> heights;

// Reorder list to smooth
for (i = 0; i < (resolution - 1); i++)
{
    int k = (i * 6);

    // The top row is just above this row
    // Add top left to heights (+2 for top left)
    heights.push_back(vertices[k + 2].position.y);
    // Check if at the end of the row
    if (i == (resolution - 2))
    {
        // Add top right to heights (+1 for top right)
        heights.push_back(vertices[k + 1].position.y);
    }
}

for (j = 0; j < (resolution - 1); j++)
{
    for (i = 0; i < (resolution - 1); i++)
    {
        int k = (j * resolution * 6) + (i * 6);

        // Add bottom left to heights (+0 for bottom left)
        heights.push_back(vertices[k].position.y);
        // Check if at the end of the row
        if (i == (resolution - 2))
        {
            // Add bottom right to heights (+4 for top right)
            heights.push_back(vertices[k + 4].position.y);
        }
    }
}
```

After they are in this order the smoothing function that was implemented in the rastertek framework will work as expected but this will not actually change the heights’ of the points on the plane, however it will store the heights the points should be at in the vector. Another formula goes through the array of vertices and figures out which point in the list of smoothed heights it correlates to.

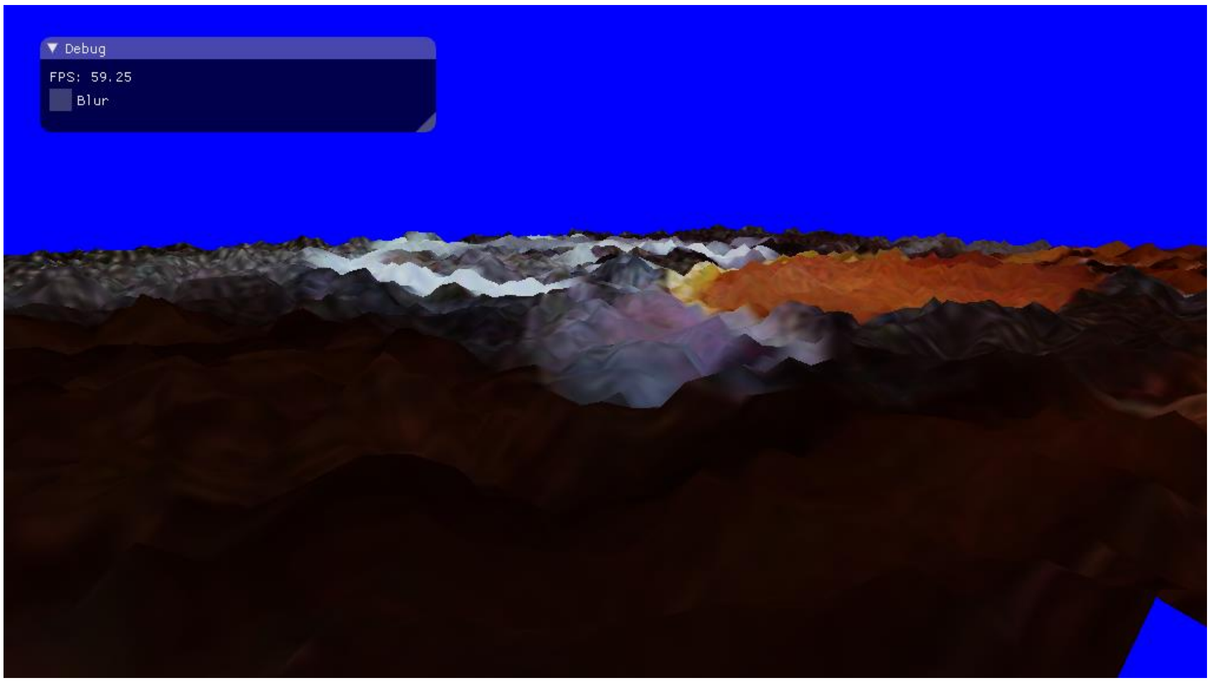


```
// Set to smoother heights
for (j = 0; j < (resolution - 1); j++)
{
    for (i = 0; i < (resolution - 1); i++)
    {
        k = (resolution * j) + i;
        vertices[index].position = XMFLOAT3(vertices[index].position.x, heights[k], vertices[index].position.z);
        index++;

        k = (resolution * (j + 1)) + (i + 1);
        vertices[index].position = XMFLOAT3(vertices[index].position.x, heights[k], vertices[index].position.z);
        index++;

        k = (resolution * (j + 1)) + i;
        vertices[index].position = XMFLOAT3(vertices[index].position.x, heights[k], vertices[index].position.z);
        index++;

        k = (resolution * j) + i;
        vertices[index].position = XMFLOAT3(vertices[index].position.x, heights[k], vertices[index].position.z);
        index++;

        k = (resolution * j) + (i + 1);
        vertices[index].position = XMFLOAT3(vertices[index].position.x, heights[k], vertices[index].position.z);
        index++;

        k = (resolution * (j + 1)) + (i + 1);
        vertices[index].position = XMFLOAT3(vertices[index].position.x, heights[k], vertices[index].position.z);
        index++;
    }
}
```

This causes the plane to be smoothed effectively.



(Unsmoothed Perlin noise)



(Smoothed Perlin noise)

**Gaussian blur -** The gaussian blur in this application is made by combining a horizontal and vertical blur, which gives a very similar effect to a true gaussian blur but is easier to perform.
This blur requires the scene to go through several rendering stages before being output to the screen. The stages are as follows.

Render to texture - The scene is rendered as it normally would be however instead of rendering to the back buffer so that it can be output to the screen, it is rendered to a render texture that can be altered in the various other stages of the rendering process.

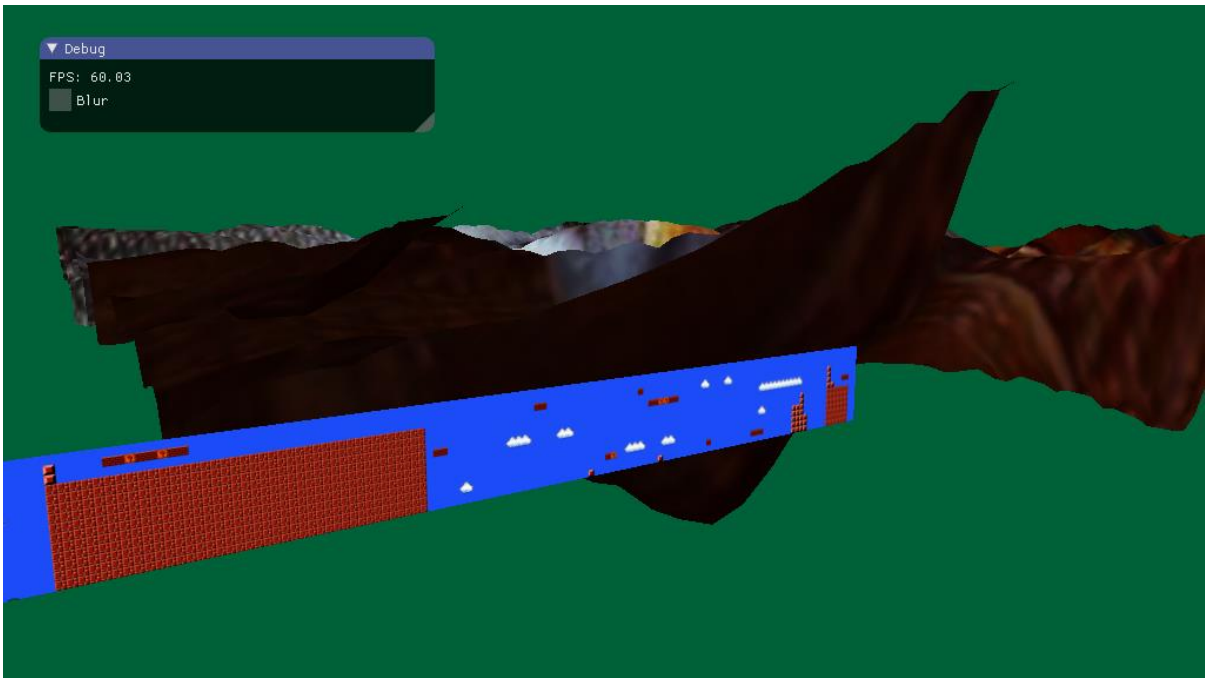
Down sample - The render texture of the scene is applied to an ortho mesh and is rendered again to another texture that is half the width and height of the screen. This means that the images that will be worked with in the horizontal and vertical blur stages will be a lower resolution and therefore will take less time to blur.

Horizontal blur - The lower resolution texture is rendered using the horizontal blur shader. The vertex shader finds the texture coordinates of the current pixel and the three pixels to its left and right. The pixel shader then takes the colour of all of these pixels and multiplies them by weighting values that all add to one. This blurs the pixels with the pixels next to them horizontally.

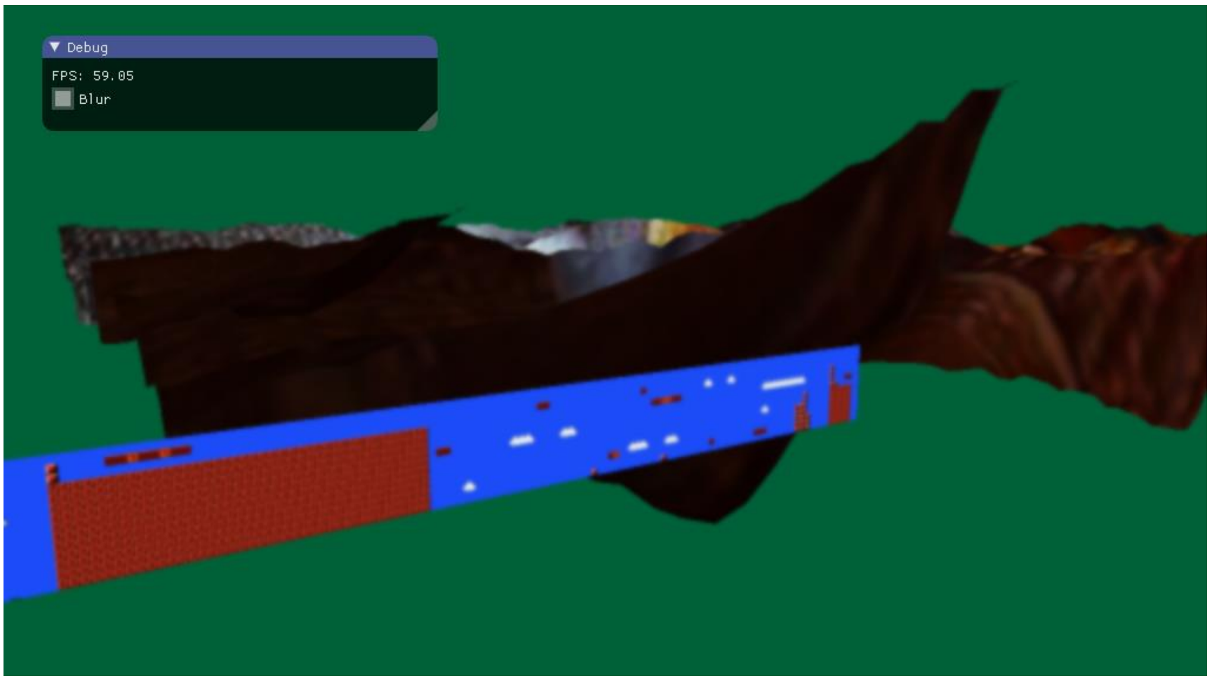
Vertical blur - The blurred texture is rendered again using the same blurring technique however this time the vertex shader finds the three pixels above and below the current pixel. This blurs the pixels with the pixels next to them vertically.

Up sample - The image is rendered to a texture that is the same size as the screen to bring it back up to the resolution it is supposed to be at.

Render scene - The blurred texture is rendered to the screen at the correct resolution.



(The image rendered without blur)



(The image rendered with blur)

**Description of organisation**

The Markov chain level generator is stored in a separate class from the rest of the application. As the level is stored as a PNG image that can then be loaded into the application by the texture manager, there is no need for the Markov class to directly communicate with the application. The Markov class keeps this complex part of the code separate from the main application.

The Markov class is split up into the different steps that it goes through in order to generate a level.

- Create a list of all tiles in the input level.
- Calculate all probability values for the tiles.
- Generate a grid of tile values based on these probabilities.
- Create the bitmap of the new level using the list of tiles and the grid of tile values.

There was also a function created to print the level to the output console by translating the tile values into colours. This was used only for debugging purposes.

Each step of the Markov class relies on the information created in the last step, however it was easy to isolate issues when they could be narrowed down to a specific function and also using test data, each part of the process could be tested independently.

The ratio class is used to find out how to best scale the quad to display the level correctly. It stores a float for the scale in the x and y axis and will calculate these using the height and width of the generated level (found using the GDI+ library [10]). Then it can return this to the main application.

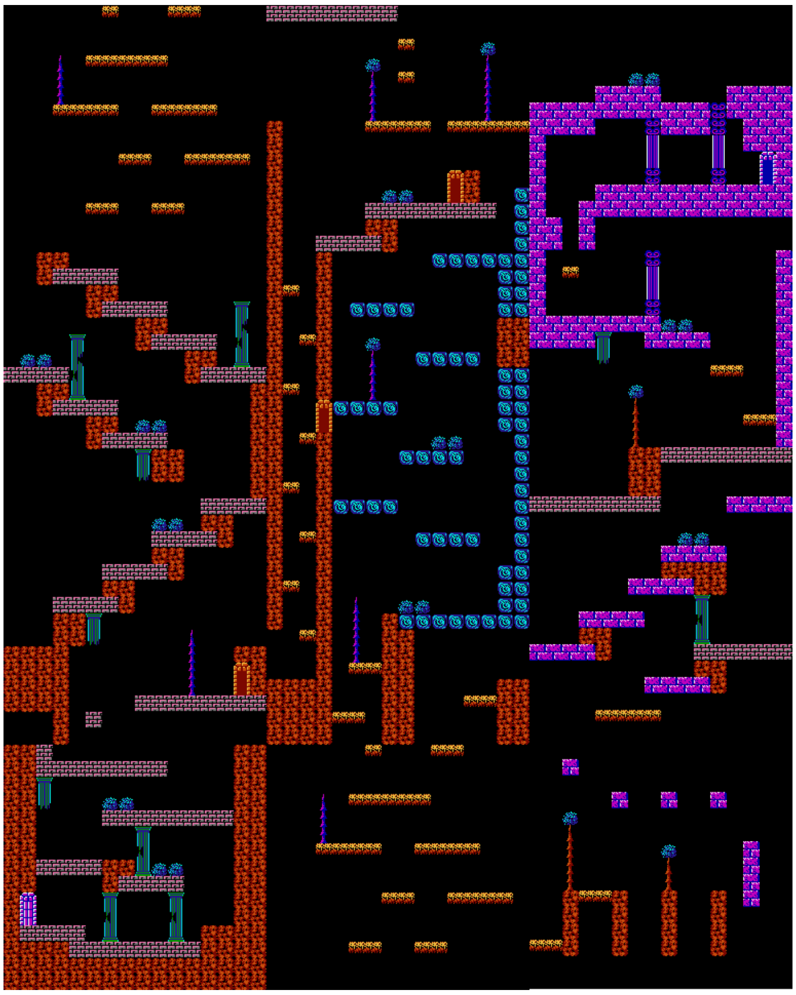
The noise generation code is also stored in a separate class in order to keep the main class easily readable. This class in designed to take in any input and return a double so that it can be used wherever is necessary.

Each stage of the rendering process is in its own function so that they can be read easily. The render function calls all of these functions in turn to create the full render. This is useful as several of these render functions are only used in the gaussian blur which can be turned off and so an if statement can bypass the parts of the rendering process that blur the image.

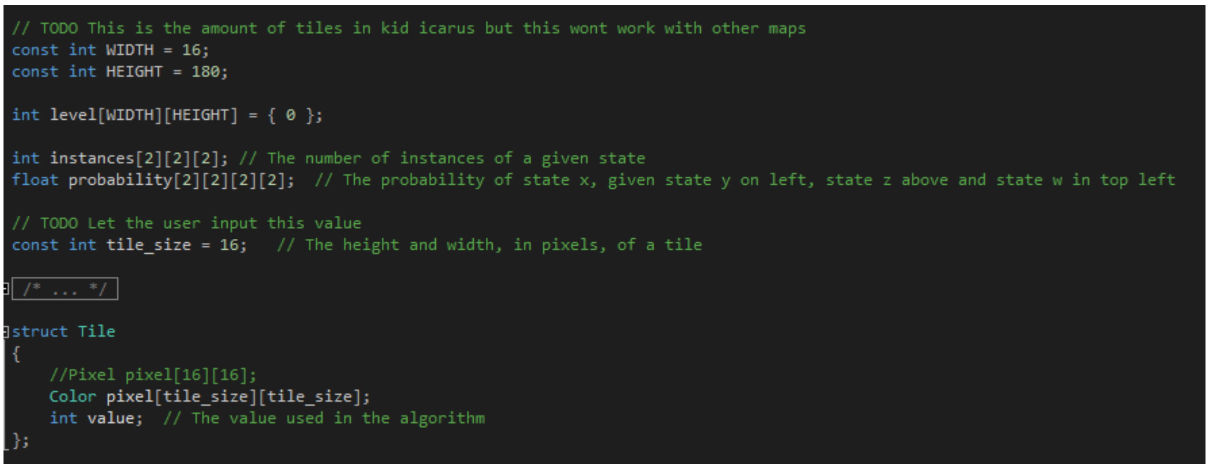
The smoothing function is contained within the plane mesh class as it is the only place it needs to be used.

## **Critical appraisal**

The main issue with the application is that it runs quite slowly. There were several attempts made to speed up the application however it was at its fastest when it was built specifically to create levels for one game, which was level 1-1 form Kid Icarus [1]. (This level is completely vertical however it has been split into three parts for formatting)

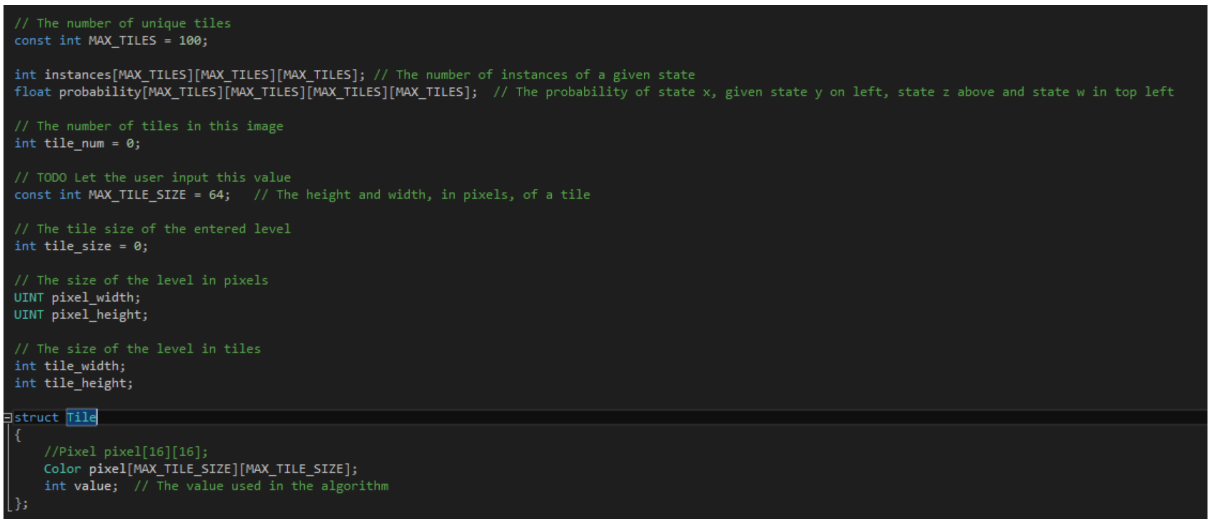


This used multi-dimensional arrays with the specific values necessary for this level however the concept of the application was for it to create a level for any tile based game that was input.



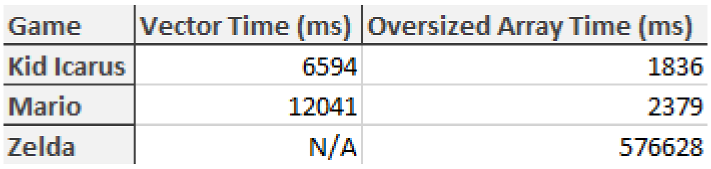
```
// TODO This is the amount of tiles in kid icarus but this wont work with other maps
const int WIDTH = 16;
const int HEIGHT = 180;

int level[WIDTH][HEIGHT] = { 0 };

int instances[2][2][2][2]; // The number of instances of a given state
float probability[2][2][2][2];  // The probability of state x, given state y on left, state z above and state w in top left

// TODO Let the user input this value
const int tile_size = 16;   // The height and width, in pixels, of a tile

/* ... */

struct Tile
{
    //Pixel pixel[16][16];
    Color pixel[tile_size][tile_size];
    int value;  // The value used in the algorithm
};
```

In order to allow these arrays to be changed to the necessary size for other inputs they were first replaced with vectors. This worked for the other inputs however this slowed the application by a huge amount. This was remedied by replacing the vectors with arrays of a fixed length that is greater than will be necessary and only using part of them. This means the system will work for any game as long as the image fulfils a few requirements.



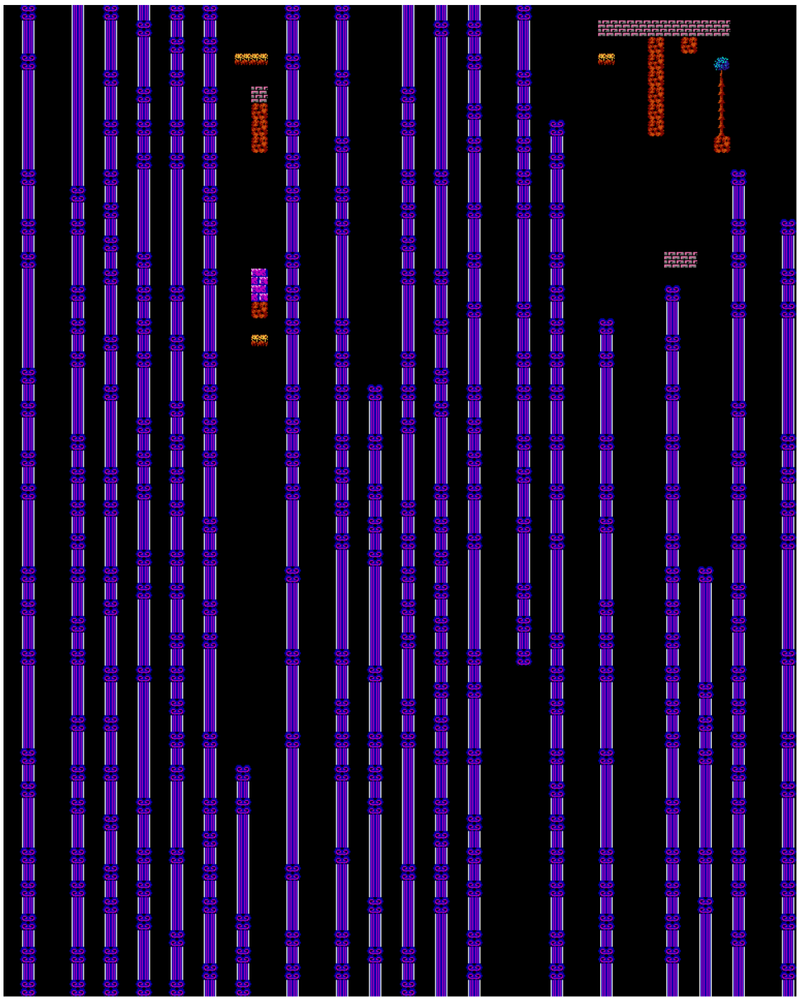
```
// The number of unique tiles
const int MAX_TILES = 100;

int instances[MAX_TILES][MAX_TILES][MAX_TILES]; // The number of instances of a given state
float probability[MAX_TILES][MAX_TILES][MAX_TILES][MAX_TILES];  // The probability of state x, given state y on left, state z above and state w in top left

// The number of tiles in this image
int tile_num = 0;

// TODO Let the user input this value
const int MAX_TILE_SIZE = 64;   // The height and width, in pixels, of a tile

// The tile size of the entered level
int tile_size = 0;

// The size of the level in pixels
UINT pixel_width;
UINT pixel_height;

// The size of the level in tiles
int tile_width;
int tile_height;

struct Tile
{
    //Pixel pixel[16][16];
    Color pixel[MAX_TILE_SIZE][MAX_TILE_SIZE];
    int value;  // The value used in the algorithm
};
```

This system was far quicker than the vectors. The Zelda overworld [3] was left to generate for over two hours but did not complete.

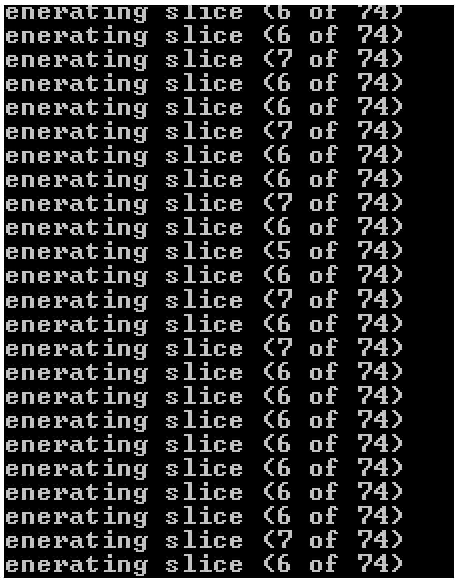


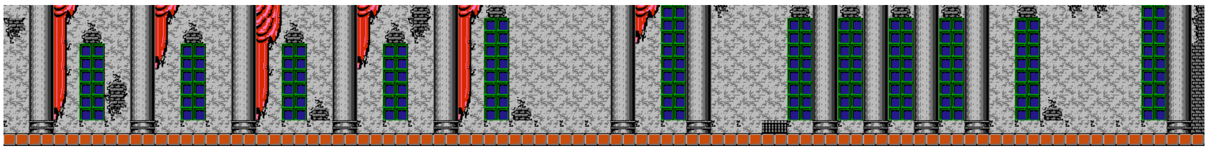
| Game | Vector Time (ms) | Oversized Array Time (ms) |
|---|---|---|
| Kid Icarus | 6594 | 1836 |
| Mario | 12041 | 2379 |
| Zelda | N/A | 576628 |

In cases where there are some tiles which are used very infrequently, the conditions for them to appear could be very specific and there could be a very limited range of tiles that could follow it. In the case of Kid Icarus level 1-1 [1], There was a purple pillar that, if it appeared in the level, would most likely be followed by more purple pillar tiles for the rest of the level.



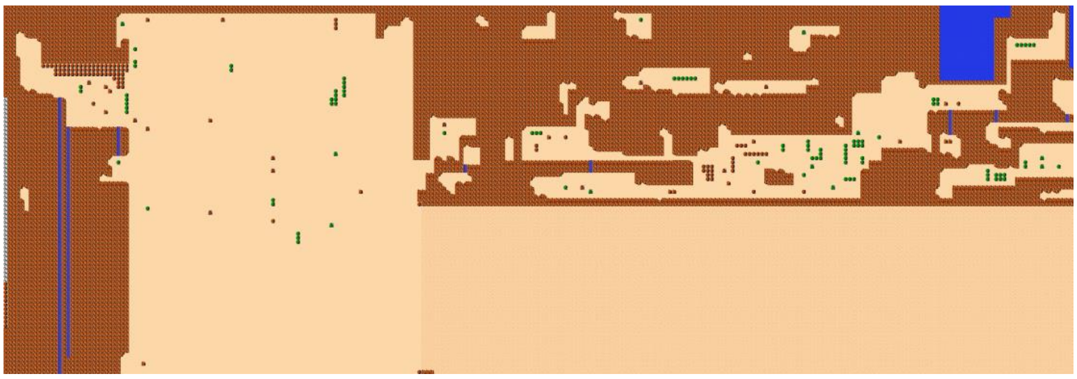
A section of the first level from Contra [4] was used in the system also and because of the multitude of different tiles with stars on them that make up the sky, none of the individual tiles are used enough times and so the system often gets stuck unless very specific tiles appear. A similar problem occurred with level 1 of Castlevania [5] as there were a lot of tiles used very infrequently to make very detailed backgrounds.


```
enerating slice (6 of 74)
enerating slice (6 of 74)
enerating slice (7 of 74)
enerating slice (6 of 74)
enerating slice (6 of 74)
enerating slice (7 of 74)
enerating slice (6 of 74)
enerating slice (6 of 74)
enerating slice (7 of 74)
enerating slice (6 of 74)
enerating slice (5 of 74)
enerating slice (6 of 74)
enerating slice (7 of 74)
enerating slice (6 of 74)
enerating slice (7 of 74)
enerating slice (6 of 74)
enerating slice (6 of 74)
enerating slice (6 of 74)
enerating slice (6 of 74)
enerating slice (6 of 74)
enerating slice (6 of 74)
enerating slice (6 of 74)
enerating slice (7 of 74)
enerating slice (6 of 74)
```
(The first level of Contra [4] cannot progress past column 7)

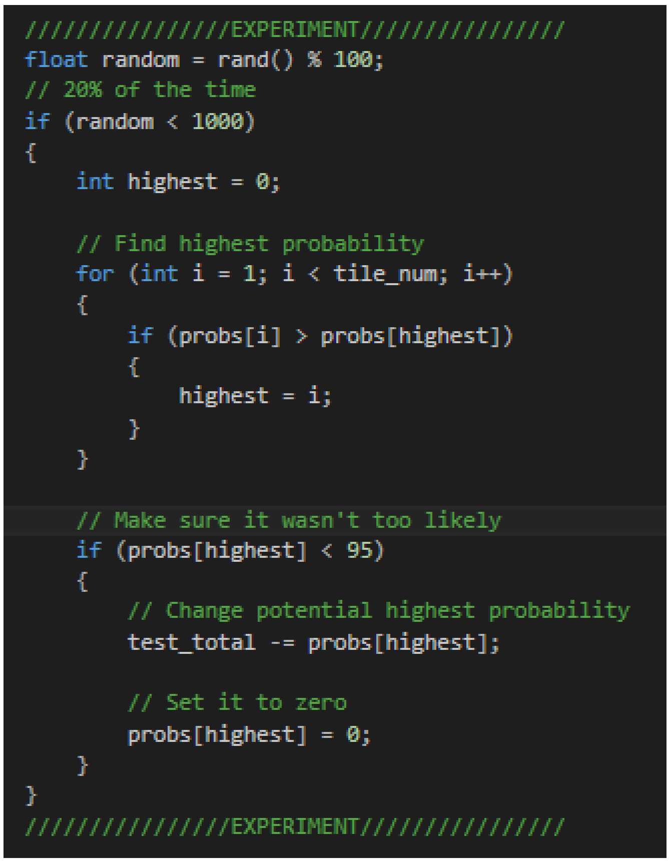

(Castlevania [5] becomes very repetitive)

The conflict resolution currently implemented makes the system repeat itself a lot. Parallel programming was considered to help speed up the program by splitting the level grid up and having multiple threads calculate the tile values of different parts of the level simultaneously. This would not work however as the tiles must be calculated in order because they use the previous tiles in the calculations.

The code doesn’t consistently create playable levels. It has a tendency to create large blocks of the same tile instead of varying the tiles the same amount as the original image. The level generator is working as expected under the rules it uses (Tiles are not appearing in places that should be impossible) and so this is likely just a flaw in the concept or a problem with the random number generation (the C++ rand function).


(There are large chunks of sand as these are the most probable tiles.)

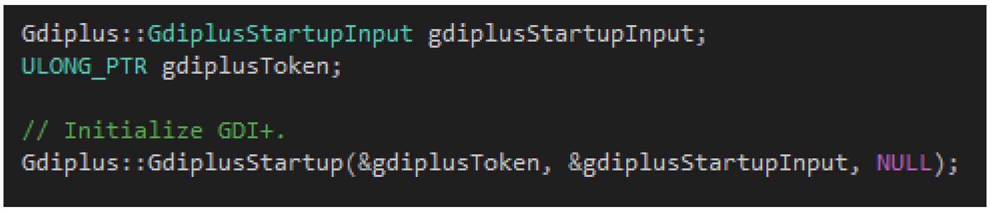
There was an attempt to combat this by adding to the random tile generator a 20% chance to remove the most probable tile from the list of possibilities however this did not work and so was removed.



```
////////////////EXPERIMENT///////////////
float random = rand() % 100;
// 20% of the time
if (random < 1000)
{
    int highest = 0;

    // Find highest probability
    for (int i = 1; i < tile_num; i++)
    {
        if (probs[i] > probs[highest])
        {
            highest = i;
        }
    }

    // Make sure it wasn't too likely
    if (probs[highest] < 95)
    {
        // Change potential highest probability
        test_total -= probs[highest];

        // Set it to zero
        probs[highest] = 0;
    }
}
////////////////EXPERIMENT////////////////
```

## Reflection

Before the beginning of this project I had never had a reason to read and analyse an image in code like I do in this project. I had only been able to read in image files in existing game engines or frameworks and so I had no prior experience of doing this. After doing some research and getting some help from a classmate [11] I started using the GDI+ library [10]. There is a specific way to set this up that was quite difficult to find online and the documentation was quite difficult to search through to find the functions that are used in this project. After some experimentation I found out how to use the get and set pixel functions which fulfilled the majority of the tasks I needed.



```
Gdiplus::GdiplusStartupInput gdiplusStartupInput;
ULONG_PTR gdiplusToken;

// Initialize GDI+.
Gdiplus::GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);
```

(This is necessary to use GDI+)

I had not used markov chains before and didn’t know exactly how to use them. I did a lot of research into how to use them and more specifically I researched other times they had been used to generate tile based game maps. The papers I had found by Sam Snodgrass and Santiago Ontanon [6] [7] was very helpful in understanding what a markov chain was, how to implement it mathematically and how a two dimensional markov chain might work. The method used was designed myself based on the logic of a 2D markov chain.

I wrongfully assumed during this project that a vector would be more efficient than an array for a system that could take input from many different games. I thought this because I had to make the array very large and would usually end up not using most of it but with a vector it would be made to be the exact size for the given input. This however was completely false and adding the vectors slowed down the application a great deal. This project has taught me more about what data structures are more appropriate in different situations.

I originally implemented perlin noise [9] in the rastertek framework. This framework was new to me however it was very easy to implement because, as stated above, the geometry was saved as a linear array of points. I moved my application to the shaders framework as I more familiar with it after working in it last semester however implementing the perlin noise and the smoothing function required far more maths and work than expected.

## References

| | | | |
|---|---|---|---|
| **1** | **Computer Game** | Nintendo (1986) *Kid Icarus - Nintendo Entertainment System* [Video Game]. Nintendo. | **(Nintendo, 1986)** |
| **2** | **Computer Game** | Nintendo (1985) *Super Mario Bros - Nintendo Entertainment System* [Video Game]. Nintendo. | **(Nintendo, 1985)** |
| **3** | **Computer Game** | Nintendo (1986) *The Legend of Zelda - Nintendo Entertainment System* [Video Game]. Nintendo. | **(Nintendo, 1986)** |
| **4** | **Computer Game** | Konami (1987) *Contra - Nintendo Entertainment System* [Video Game]. Konami. | **(Konami, 1987)** |
| **5** | **Computer Game** | Konami (1986) *Castlevania - Nintendo Entertainment System* [Video Game]. Konami. | **(Konami, 1986)** |
| **6** | **Paper** | Sam Snodgrass and Santiago Ontanon (no date given) ‘Experiments in Map Generation using Markov Chains’, Available at: http://www.fdg2014.org/papers/fdg2014_paper_29.pdf | **(Sam Snodgrass and Santiago Ontanon)** |
| **7** | **Paper** | Sam Snodgrass and Santiago Ontanon (no date given) ‘A Hierarchical MdMC Approach to 2D Video Game Map Generation’, Available at: https://pdfs.semanticscholar.org/99b6/d919fd88d507e45d3a692fdc3b1f242ed8e0.pdf | **(Sam Snodgrass and Santiago Ontanon)** |
| **8** | **Website** | NESMaps (2006), Available at: http://www.nesmaps.com/#E | **(NESMaps, 2006)** |
| **9** | **Website** | Ken Perlin (2002), *Improved Noise reference implementation,* Available at: http://mrl.nyu.edu/~perlin/noise/ | **(Ken Perlin, 2002)** |
| **10** | **Website** | GDI+ (2012), Available at: https://msdn.microsoft.com/en-us/library/windows/desktop/ms533798(v=vs.85).aspx<br>GDI+ helper function, Available at: http://read.pudn.com/downloads12/sourcecode/windows/49942/Starting%20GDIPlus/GdiplusHelperFunctions.h__.htm | **(GDI+, 2012)** |
| **11** | **Classmate** | Andrew Milne, Helped me set up GDI+ | |