

HPC report

by George Anthony Thomas

FILE	1387_GEORGE_ANTHONY_THOMAS_HPC_REPORT_107698_1364066509.DOCX (427.2K)		
TIME SUBMITTED	08-MAY-2019 07:59PM (UTC+0100)	WORD COUNT	4370
SUBMISSION ID	106689063	CHARACTER COUNT	22667

Parallelising a Perlin Noise Height Map

Name: George Thomas

P-number: P149706

Module Title: High Performance Computing for Games and Visualisation

Assignment Title: HPC for games assignment

Module Lecturer: Gordon Dickers

Abstract

In this report, the developer will discuss the steps taken to implement and parallelise a Perlin noise height map using a mesh grid. The developer will briefly discuss previous work that has been done on this topic, followed by an in depth look into the method used to implement this project. Finally, the developer will discuss the results found when testing the speed of the CPU algorithm against the speed on the GPU algorithm, followed by a conclusion of the project.

Table of Contents

Table of Contents	3
Table of figures	3
Abstract	2
Background Literature	4
Introduction	4
Noise functions	4
Perlin Noise	4
Simplex Noise.....	5
Noise and Terrain Generation.....	5
Cuda OpenGL interoperability	5
Algorithm and Implementation	7
Grid setup.....	7
Perlin Noise	7
Rendering	7
Parallelisation.....	10
Results and Evaluation	12
Conclusion and future work.....	13
References	14
Appendix	15
CPU Implementation.....	15
GPU Implementation	20

Table of figures

Figure 1 terrain created using Perlin noise function.....	4
Figure 2 Height map using Perlin Noise	8
Figure 3 Height map using Perlin Noise	8
Figure 4 Perlin noise function running at lower frequency	9
Figure 5 Perlin noise function running at higher frequency	9
Figure 6 Perlin height map on GPU.....	10

Background Literature

Introduction

In this section of the report the developer will investigate the current literature about Perlin noise, noise functions and the use that they have in creating height maps. Also the developer will look into the interoperability between Nvidia's Cuda and OpenGL.

Noise functions

A noise function outputs values which vary between 0 and 1. The values are generated by using pseudo-random numbers which has been determined in the region of which noise is to be introduced [1]. There are three main types of noise functions, value noise functions, gradient noise functions and value + gradient noise functions. Value noise uses the pseudo random numbers as the input to the noise function and the slope of the function is 0 at each point [1]. Value noise creates a lattice of points that are given random or pseudo-random numbers [2]. For gradient noise, every point is used as the input for every point. Gradient noise creates gradients using the pseudo-random numbers. Gradient noise creates a lattice of pseudo-random gradient, the dot product of the gradients are interpolated to be used as values in-between the lattice [3] Perlin noise is a very famous type of gradient noise. Gradient + value noise is a combination of the two methods.

Perlin Noise

Perlin noise is a form of gradient noise and was created in the early 1980s by Ken Perlin while he was working on the original Tron movie [4]. The function has a wide variety of uses ranging from terrain generation to procedural textures such as the one shown below in figure 1.

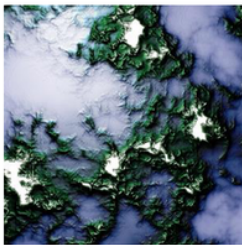


Figure 1 terrain created using Perlin noise function

Perlin Noise provides a more organic appearance than many other noise functions that are available, it uses a naturally ordered sequence of pseudo-random numbers which allow for a smooth step from one point to another where as other methods will provide a much more erratic effect with harsh peaks [5]. There are 3 main stages to the Perlin noise function, first of which being the grid definition. As mentioned, a gradient noise function creates a lattice of pseudo-random gradients at each point. This is what happens in Perlin noise, each point is assigned a random unit length gradient

vector. For 1 and 2 dimensions a simple random number generator can be used but for higher dimensions other methods of random number generation are required [5].

The next step is to find which cell a given point falls. To achieve this, a distance vector is calculated between the point and each corner node of the cell. The dot product is then calculated between the gradient vector and the distance vector which has just been calculated [5]. The last step of the Perlin noise function is to interpolate between the resulting dot products of the cell, this will give the final value of the function which will be between 1 and 0.

Simplex Noise

Simplex noise is a noise function which is also created by Ken Perlin. It was created to solve many issues that were in his original noise function. It has more success when working in a higher dimension and requires a lot less computational power due to it needing fewer calculations. The main difference between the two functions is that simplex noise splits the space up into simplices [6]. This reduces data points because a hypercube has 2^n corners whereas a simplex has $n+1$ corners, where n is the number of dimensions. This means that only in 1 dimension will simplex not have more corners [6].

Noise and Terrain Generation

A common used method of creating terrain is to create building blocks using noise functions like Perlin noise [7]. Of course, noise is just a series of numbers which range from 0 to 1, however, these can be used to create terrain by using the values to affect the elevation of points. This is referred to as a height map [7]. Using noise to affect elevation will lead to a uniform set of hills, this already resembles terrain, but it can be improved in various ways to further enhance the illusion of terrain. For example, changing the frequency of the noise will cause the “hills” will cause the amount of peaks to increase or decrease depending on the value of the frequency [7]. Another way to improve the terrain is to add extra noise into the function by adding different frequencies of noise at different sizes. This will cause the terrain to look more like fractal terrain [7]. This method is referred to as adding octaves. There are a few other methods such as raising the elevation to an exponent, this can lead to flat land or extreme heights depending on the exponent provided [7].

Cuda OpenGL interoperability

Nvidia’s GPU language, Cuda, provides features that are known as Cuda-OpenGL interoperability [8]. This allows for Cuda and OpenGL to work with each other to provide faster computation time. This process has a distinct order in which code needs to be run. Firstly, a buffer needs to be created using OpenGL, the type of buffer used depends on the dimension in which you are working in. 2 dimensions would use a pixel buffer and 3 dimensions would use a vertex buffer. The next step is to map the buffer into Cuda’s memory. A kernel is then applied to the data which will modify the data. The data is then unmapped from Cuda memory and drawn by OpenGL using functions provided by

OpenGL [8]. According to Nvidia, the interoperability is very fast because the framebuffer allows for Cuda and OpenGL to share data through common memory [8].

Algorithm and Implementation

The developer has chosen to implement a height map using Perlin noise to create a terrain generator. The algorithm will use a 2D mesh grid that will be set in the x and z axis. The Perlin noise function will provide values that will be used to change the y values of the mesh Grid which will create the height map. This section will discuss the methods used to implement this

Grid setup

The first step that needed to be done was to set up the mesh grid. This was simple enough to implement. The developer used a 2D array which had a max grid size² because both dimensions of the array have the same max values which represent the maximum size in each axis. The grid is initialised by looping through all points in the array and setting each elements x and z coordinates equal to the current values of i and j in the loop. When the mesh grid is set up the developer also initialises the gradient vectors at each point using a rand function that produces values between 0 and 1 for the x, y and z coordinates.

Perlin Noise

The next step is to apply Perlin noise to each point. To do this the x and z value of each point is passed into the Perlin noise function. This function will use these coordinates to find the current grid cell by using the x and z which were passed in as the minimum values of the cell. It will then add 1 to both of these values and use them as the maximum values of the cell. Once the cell has been determined, the next step is to find the dot product between the gradient and distance vector. There are four distance vectors on each cell, one for each corner. The distance vectors are calculated by subtracting each corner by the current position, as soon as the distance vector is calculated, the dot product between the gradient vector at that corner and the distance vector at that corner is calculated. Once the dot product is found for each corner, the next step is to interpolate between them. The bottom left and bottom right dot products are interpolated together and the top left and top right dot products are interpolated together. Immediately after that, the bottom and top interpolations are interpolated together to give the final value for the y value of the current point. The reason the two interpolated values are interpolated again is to ensure all corners are included in the final value. To allow for a more realistic terrain, different octaves of the Perlin function are added to the y value. This creates more hills that are smaller and have a lower max height.

Rendering

To test that the Perlin noise function is working correctly, the program is rendered using OpenGL. The mesh grid is passed into a display function which will loop through all of the points and render the x, y and z coordinates using OpenGL's line loop function. This will display a mesh grid that resembles terrain.

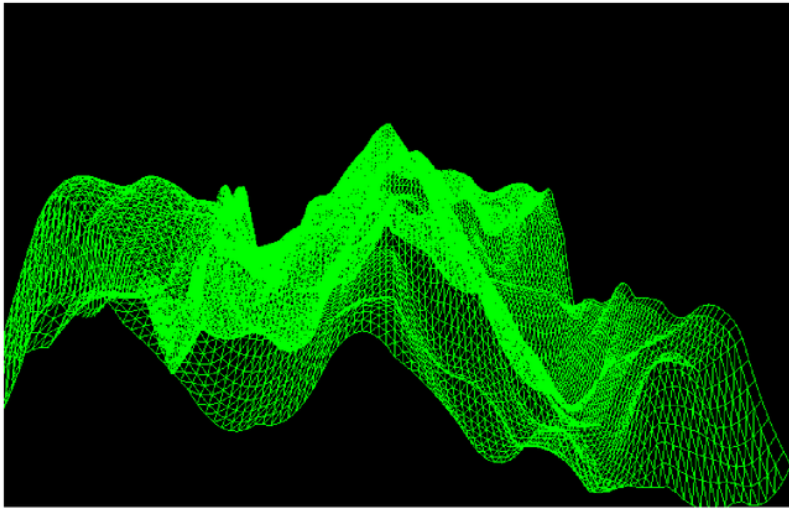


Figure 2 Height map using Perlin Noise

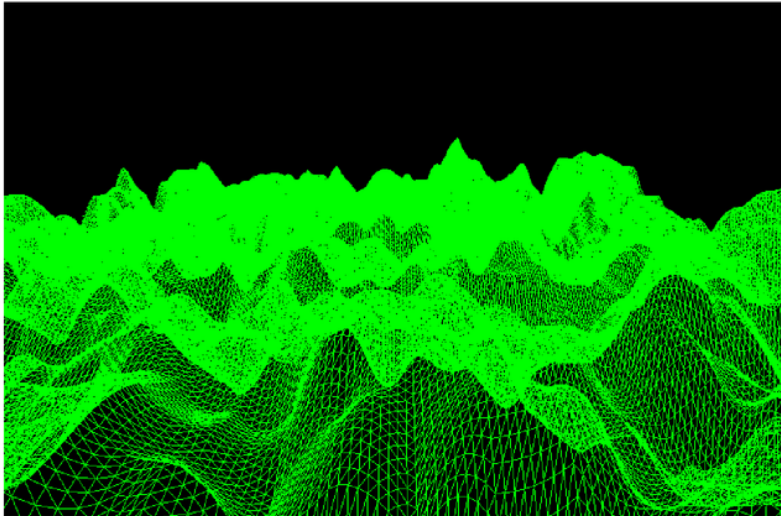


Figure 3 Height map using Perlin Noise

Figures 2 and 3 above show the Perlin noise function applied to two different mesh grids of different sizes. As can be seen from the figures, the y values of the mesh grid have been changed by the function which causes the mesh to look like terrain. There is only a small change between adjacent y positions which is as intended for the Perlin noise function. The user can change the frequency of the Perlin noise function by using the z and x keys, figures 4 and 5 below show the algorithm running at a lower and higher frequency. The user will also be able to manipulate the position of the grid by using the q, w, e, a, s and d keys.

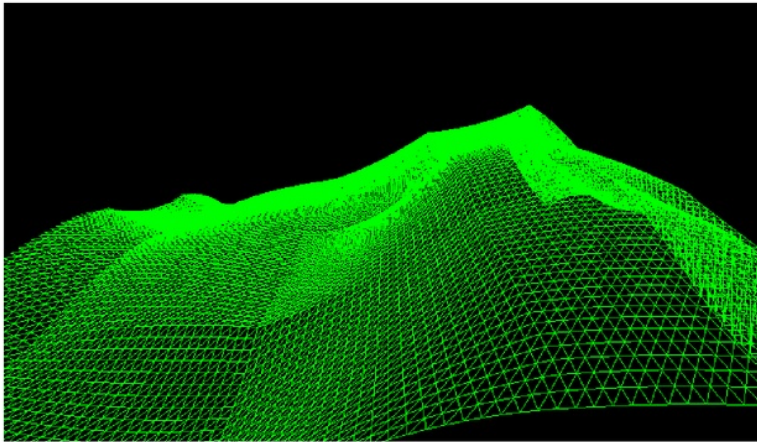


Figure 4 Perlin noise function running at lower frequency

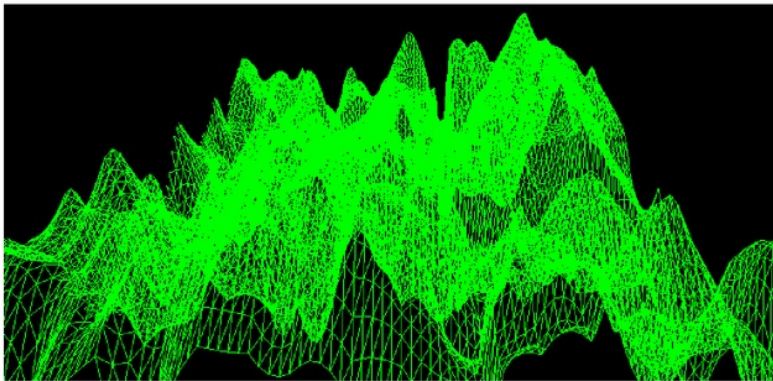


Figure 5 Perlin noise function running at higher frequency

Parallelisation

This section will discuss the aspects of the code that will be parallelised.

The implementation into Cuda uses the same algorithm that the CPU implementation uses. However, the algorithm has been adjusted to utilise the GPU. For the GPU implementation, the developer has decided to place all of the Perlin noise calculations into a kernel, so that all points in the grid can be calculated simultaneously. Before calling the kernel, the mesh grid and gradient vectors are initialised as they were for the CPU implementation. After initialisation, these arrays are allocated and copied in Cuda memory. Once Cuda has access to the mesh grid and the gradient vectors, the kernel is called and these arrays are passed through. Cuda will then run a thread for each point in the mesh grid and apply the new y value to each point using the Perlin noise function. Using Cuda's devices, which work much like functions, the kernel can successfully calculate the distance vector, the dot product and the final interpolated value for each point in the grid. The kernel will also apply octaves to the mesh grid much like the CPU implementation, which means that because there are three octaves, all of the Perlin noise calculations are completed three times for each point. One for each octave. The developer decided that this would be enough calculations to be worth parallelising. The algorithm is limited to the size of the grid because the developer didn't make use of the grids in Cuda. This means that the algorithm is only currently using blocks and threads. The developer expects the algorithm to run faster on the GPU than the CPU.

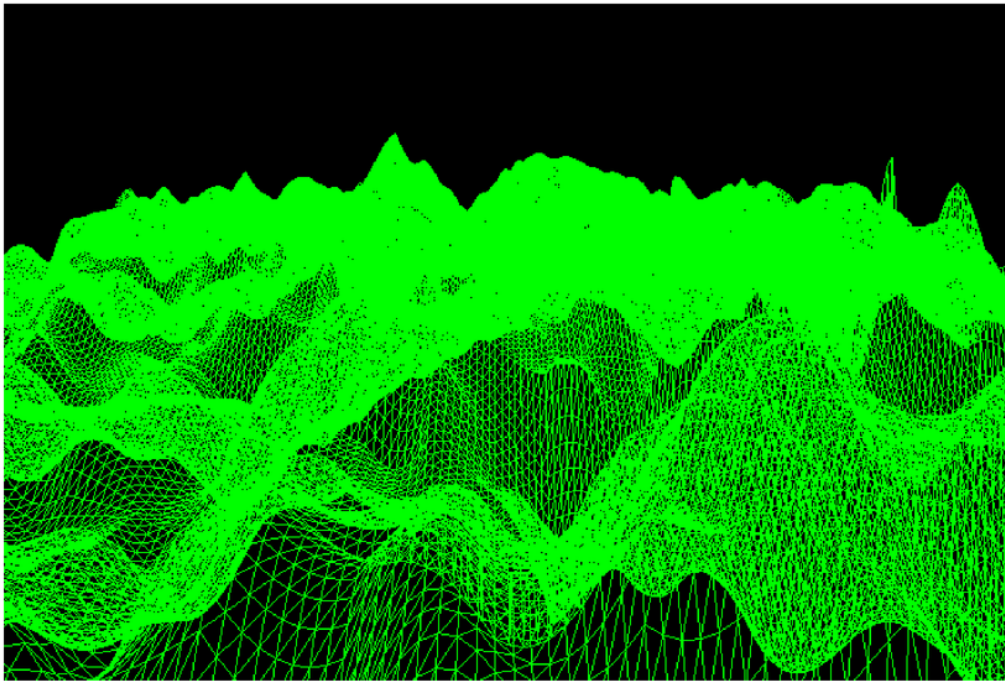


Figure 6 Perlin height map on GPU

Figure 6 above shows the algorithm running using Cuda. The algorithm was successfully implemented and gives the expected results. Both CPU and GPU output the exact same height map.

Results and Evaluation

In this section the developer will discuss the results that were found from testing the speeds of both CPU and GPU algorithms. Both algorithms are using a mesh grid that has 72900 points and both are running on visual studio 2015. All other data that is used is identical.

Results from testing in Room 204, IQ building		
	CPU (C++) (seconds)	GPU (Cuda) (seconds)
Test 1	2.3	0.869
Test 2	5.3	0.729
Test 3	2.3	0.653
Test 4	2.3	0.564
Test 5	2.7	0.712
Average	2.98	0.7054

As can be seen from the tests in the tables above, the GPU implementation is faster than the CPU implementation. It takes on average 2.2 seconds less to run on the GPU which is roughly 4.22 times faster. The tests were done without rendering the image, this is because OpenGL will cause the code to run continuously which will cause results which are not suitable for this test, however, all calculations are being completed.

Conclusion and future work

After getting the results of the test, the developer found that the tests were as expected. The GPU implementation increases the speed of the algorithm making it suitable for parallelisation on the GPU. The Developer thinks that the approach taken to parallelise is an efficient method to reduce computation time. In future implementation the developer will improve the Cuda implementation so that it takes grids into consideration as at the moment it only uses blocks and threads. This would allow for bigger mesh grids to be used.

References

- [1] "Noise in Computer Graphics- A brief introduction — Harold Serrano." [Online]. Available: <https://www.haroldserrano.com/blog/noise-in-computer-graphics-a-brief-introduction>. [Accessed: 11-Dec-2018].
- [2] "Value Noise and Procedural Patterns: Part 1 (Introduction)." [Online]. Available: <http://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/introduction>. [Accessed: 06-May-2019].
- [3] D. S. Ebert, *Texturing and modeling : a procedural approach*. AP Professional, 1994.
- [4] P. Kerman, *Macromedia Flash 8 @work : projects and techniques to get the job done*. Sams, 2006.
- [5] K. Perlin, "An Image Synthesizer," 1985.
- [6] K. Perlin, "Improving Noise."
- [7] "Making maps with noise functions." [Online]. Available: <https://www.redblobgames.com/maps/terrain-from-noise/>. [Accessed: 12-Dec-2018].
- [8] "What Every CUDA Programmer Should Know About OpenGL Motivation."

Appendix

CPU Implementation

```
#include <iostream>
#include "glut.h"
#include "math.h"

using namespace std;
const int MAX_TERRAIN_SIZE = 270;
float gridSize = 0.1;
struct vec3 {
    float x = 0, y = 0, z = 0;
};

float maxY = 5.0;
vec3 terrain[MAX_TERRAIN_SIZE][MAX_TERRAIN_SIZE];

float movex = 0, movey = 0, movez = 0;

vec3 gradient[MAX_TERRAIN_SIZE][MAX_TERRAIN_SIZE];

float turn = 0;
float scalar = 0.3;

int transition = 0;

float lerp(float a0, float a1, float w) {
    return (1.0f - w)*a0 + w*a1;
}

float dotterrainGradient(int ix, int iz, float x, float z) {

    // Compute the distance vector
    float changeX = x - (float)ix;
    float changeZ = z - (float)iz;

    // Compute the dot-product
    return (changeX*gradient[iz][ix].x + changeZ *gradient[iz][ix].z);
}

// Compute Perlin noise at coordinates x, y
float perlin(float x, float z) {

    // Determine grid cell coordinates
    int x1 = (int)x;
    int x2 = x1 + 1;
    int z1 = (int)z;
    int z2 = z1 + 1;

    // Determine interpolation weights
    float sizex = x - (float)x1;
    float sizey = z - (float)z1;

    // Interpolate between grid point gradients
    float n1, n2, interpx1, interpx2, value;
```

```

        n1 = dotterrainGradient(x1, z1, x, z);
        n2 = dotterrainGradient(x2, z1, x, z);
        interpx1 = lerp(n1, n2, sizex);

        n1 = dotterrainGradient(x1, z2, x, z);
        n2 = dotterrainGradient(x2, z2, x, z);
        interpx2 = lerp(n1, n2, sizex);

        value = lerp(interpx1, interpx2, sizey);
        return value;
    }
    void terrainSet()
    {
        for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
        {
            for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
            {
                terrain[j][i].x = j * gridSize;
                terrain[j][i].y = 0;
                terrain[j][i].z = i * gridSize;
            }
        }

        //terrain[5][5].y = 2;

        for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
        {
            for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
            {
                gradient[i][j].x = 2.0f * ((float)rand() / (float)RAND_MAX) -
1.0f;
                gradient[i][j].y = 2.0f * ((float)rand() / (float)RAND_MAX) -
1.0f;
                gradient[i][j].z = 2.0f * ((float)rand() / (float)RAND_MAX) -
1.0f;

                //cout << gradient[i][j].x << "," << gradient[i][j].y << "," <<
gradient[i][j].z << endl;
            }
        }
    }

    void noise()
    {
        for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
        {
            for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
            {
                /*float e = 1 * perlin(1 * terrain[j][i].x, 1 * terrain[j][i].z);
+0.5 * perlin(2 * terrain[j][i].x, 2 * terrain[j][i].z);
+0.25 * perlin(4 * terrain[j][i].x, 4 * terrain[j][i].z);
                terrain[j][i].y = maxY * pow(e, exponent);*/
                terrain[j][i].y = maxY * (1 * perlin((1*(scalar *
terrain[j][i].x)), (1*(scalar * terrain[j][i].z)))));
                terrain[j][i].y += maxY * (0.5 * perlin((2 * (scalar *
terrain[j][i].x)), (2 * (scalar * terrain[j][i].z)))));
            }
        }
    }

```

```

        terrain[j][i].y += maxY * (0.25 * perlin((4 * (scalar *
terrain[j][i].x)), (4 * (scalar * terrain[j][i].z))));
    }
}
void init()
{
    terrainSet();
    noise();
    /*for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
    {
        for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
        {
            cout << terrain[j][i].y << "  ";

        }
    }*/
}
void update()
{
    if (transition > 2)
    {
        for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
        {
            for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
            {
                if (j == MAX_TERRAIN_SIZE - 1)
                {
                    terrain[j][i].y = terrain[0][i].y;
                }
                else
                {
                    terrain[j][i].y = terrain[j + 1][i].y;
                }
            }
        }
        transition = 0;
    }
    transition++;
    glutPostRedisplay();
}
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);           // clear screen
    and depth buffer]
    glEnable(GL_DEPTH_TEST);

    glLoadIdentity();
    // reset modelview matrix

    gluLookAt(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); //reset camera position
    glColor3f(0.0, 1.0, 0.0);

```

```

        glTranslatef(((terrain[0][0].x + ((float) - MAX_TERRAIN_SIZE / 2) * gridSize)),
-2.0, (((float)-MAX_TERRAIN_SIZE - 10) * gridSize));

//glRotatef(turn, 0.0, 1.0, 0.0);
//turn++;

glTranslatef(movex, movey, movez);

for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
{
    glBegin(GL_LINE_STRIP);
    for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
    {
        if (i != MAX_TERRAIN_SIZE - 1 && j != MAX_TERRAIN_SIZE - 1)
        {
            glVertex3f(terrain[j][i].x, terrain[j][i].y,
terrain[j][i].z);
            glVertex3f(terrain[j + 1][i].x, terrain[j + 1][i].y,
terrain[j + 1][i].z);
            glVertex3f(terrain[j + 1][i + 1].x, terrain[j + 1][i +
1].y, terrain[j + 1][i + 1].z);
            glVertex3f(terrain[j][i + 1].x, terrain[j][i + 1].y,
terrain[j][i + 1].z);
        }
    }
    glEnd();
}

glFlush(); //force drawing

glutSwapBuffers(); // bring backbuffer to foreground
//any errors then display error
codes
GLenum errCode;
const GLubyte *errString;
if ((errCode = glGetError()) != GL_NO_ERROR)
{
    errString = gluErrorString(errCode);
    fprintf(stderr, "OpenGL error : %s\n", errString);
}

update();

}

void keyboardread(unsigned char key, int x, int y)
{
    float move = 1;
    switch (key)
    {

```

```

        case 'w':
        {
            movez += move;

            break;
        }
        case 's':
        {
            movez -= move;

            break;
        }
        case 'a':
        {
            movex += move;

            break;
        }
        case 'd':
        {
            movex -= move;

            break;
        }
        case 'q':
        {
            movey += move;

            break;
        }
        case 'e':
        {
            movey -= move;

            break;
        }
        case 'z':
        {
            scalar += 0.1;
            noise();

            break;
        }
        case 'x':
        {
            scalar -= 0.1;
            noise();

            break;
        }

        default:
            break;
    }
    if (scalar <= 0)
        scalar = 0;
    glutPostRedisplay();
}

void reshape(int w, int h)
//reshape
//*****

```

```

//OpenGL reshape callback is called when the window is resized
{
    float size = 100;
    glViewport(0, 0, (GLsizei)w, (GLsizei)h); //respecify size of view port to
match new w and h
    glMatrixMode(GL_PROJECTION); //go into projection mode
    glLoadIdentity(); //reset projection matrix
    //glOrtho(-size, size, -size, size, 0.0, 200.0); //defines orthoganal
projection (parrallel
                                // viewing volume
                                // window limits of display
                                // left =0.0, right =2.0,
                                // bottom = 0.0, top = 2.0
                                // near =0.0, far = 100.0
    gluPerspective(90, (GLfloat)w / (GLfloat)h, 0.1, 1000); //redefine projection
matrix to match changes in w and h
    glMatrixMode(GL_MODELVIEW); //switch back to modelview mode
    glLoadIdentity(); //reset modelview matrix
    gluLookAt(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); //reset camera position
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800, 800);
    glutInitWindowPosition(100, 100);
    init();
    glutCreateWindow(argv[0]);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboardread);
    glutMainLoop();
    return 0;
}

```

GPU Implementation

```

#include <iostream>
#include "glut.h"
#include "math.h"

using namespace std;
const int MAX_TERRAIN_SIZE = 270;
float gridSize = 0.1;
struct vec3 {
    float x = 0, y = 0, z = 0;
};

float maxY = 5.0;
vec3 terrain[MAX_TERRAIN_SIZE][MAX_TERRAIN_SIZE];

float movex = 0, movey = 0, movez = 0;

```

```

vec3 gradient[MAX_TERRAIN_SIZE][MAX_TERRAIN_SIZE];

float turn = 0;
float scalar = 0.3;

int transition = 0;

float lerp(float a0, float a1, float w) {
    return (1.0f - w)*a0 + w*a1;
}

float dotterrainGradient(int ix, int iz, float x, float z) {

    // Compute the distance vector
    float changeX = x - (float)ix;
    float changeZ = z - (float)iz;

    // Compute the dot-product
    return (changeX*gradient[iz][ix].x + changeZ *gradient[iz][ix].z);
}

// Compute Perlin noise at coordinates x, y
float perlin(float x, float z) {

    // Determine grid cell coordinates
    int x1 = (int)x;
    int x2 = x1 + 1;
    int z1 = (int)z;
    int z2 = z1 + 1;

    // Determine interpolation weights
    float sizex = x - (float)x1;
    float sizey = z - (float)z1;

    // Interpolate between grid point gradients
    float n1, n2, interpx1, interpx2, value;

    n1 = dotterrainGradient(x1, z1, x, z);
    n2 = dotterrainGradient(x2, z1, x, z);
    interpx1 = lerp(n1, n2, sizex);

    n1 = dotterrainGradient(x1, z2, x, z);
    n2 = dotterrainGradient(x2, z2, x, z);
    interpx2 = lerp(n1, n2, sizex);

    value = lerp(interpx1, interpx2, sizey);
    return value;
}

void terrainSet()
{
    for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
    {
        for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
        {
            terrain[j][i].x = j * gridSize;
            terrain[j][i].y = 0;
            terrain[j][i].z = i * gridSize;
        }
    }
}

```

```

    }
}

//terrain[5][5].y = 2;

for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
{
    for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
    {
        gradient[i][j].x = 2.0f * ((float)rand() / (float)RAND_MAX) -
1.0f;
        gradient[i][j].y = 2.0f * ((float)rand() / (float)RAND_MAX) -
1.0f;
        gradient[i][j].z = 2.0f * ((float)rand() / (float)RAND_MAX) -
1.0f;

        //cout << gradient[i][j].x << "," << gradient[i][j].y << "," <<
gradient[i][j].z << endl;
    }
}

}

void noise()
{
    for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
    {
        for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
        {
            /*float e = 1 * perlin(1 * terrain[j][i].x, 1 * terrain[j][i].z);
+0.5 * perlin(2 * terrain[j][i].x, 2 * terrain[j][i].z);
+0.25 * perlin(4 * terrain[j][i].x, 4 * terrain[j][i].z);
terrain[j][i].y = maxY * pow(e, exponent);*/
            terrain[j][i].y = maxY * (1 * perlin((1*(scalar *
terrain[j][i].x)), (1*(scalar * terrain[j][i].z)))));
            terrain[j][i].y += maxY * (0.5 * perlin((2 * (scalar *
terrain[j][i].x)), (2 * (scalar * terrain[j][i].z)))));
            terrain[j][i].y += maxY * (0.25 * perlin((4 * (scalar *
terrain[j][i].x)), (4 * (scalar * terrain[j][i].z)))));
        }
    }
}

void init()
{
    terrainSet();
    noise();
    /*for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
    {
        for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
        {
            cout << terrain[j][i].y << " ";

        }
    }*/
}

void update()
{
    if (transition > 2)

```

```

{

    for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
    {

        for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
        {
            if (j == MAX_TERRAIN_SIZE - 1)
            {
                terrain[j][i].y = terrain[0][i].y;
            }
            else
            {
                terrain[j][i].y = terrain[j + 1][i].y;
            }
        }

        transition = 0;
    }
    transition++;
    glutPostRedisplay();
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);           // clear screen
    glEnable(GL_DEPTH_TEST);
    glLoadIdentity();
    // reset modelview matrix

    gluLookAt(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); //reset camera position

    glColor3f(0.0, 1.0, 0.0);
    glTranslatef(((terrain[0][0].x + ((float) - MAX_TERRAIN_SIZE / 2) * gridSize)),
-2.0, (((float)-MAX_TERRAIN_SIZE - 10) * gridSize));

    //glRotatef(turn, 0.0, 1.0, 0.0);
    //turn++;

    glTranslatef(movex, movey, movez);

    for (int i = 0; i < MAX_TERRAIN_SIZE; i++)
    {
        glBegin(GL_LINE_STRIP);
        for (int j = 0; j < MAX_TERRAIN_SIZE; j++)
        {
            if (i != MAX_TERRAIN_SIZE - 1 && j != MAX_TERRAIN_SIZE - 1)
            {

```

```

        glVertex3f(terrain[j][i].x, terrain[j][i].y,
terrain[j][i].z);
        glVertex3f(terrain[j + 1][i].x, terrain[j + 1][i].y,
terrain[j + 1][i].z);
        glVertex3f(terrain[j + 1][i + 1].x, terrain[j + 1][i +
1].y, terrain[j + 1][i + 1].z);
        glVertex3f(terrain[j][i + 1].x, terrain[j][i + 1].y,
terrain[j][i + 1].z);
    }
    glEnd();
}

glFlush(); //force drawing

glutSwapBuffers(); // bring backbuffer to foreground
//any errors then display error
codes
GLenum errCode;
const GLubyte *errString;
if ((errCode = glGetError()) != GL_NO_ERROR)
{
    errString = gluErrorString(errCode);
    fprintf(stderr, "OpenGL error : %s\n", errString);
}

update();

}

void keyboardread(unsigned char key, int x, int y)
{
    float move = 1;
    switch (key)
    {
        case 'w':
        {
            movez += move;

            break;
        }
        case 's':
        {
            movez -= move;

            break;
        }
        case 'a':
        {
            movex += move;

            break;
        }
        case 'd':
        {
            movex -= move;

```

```

        break;
    }
    case 'q':
    {
        movey += move;

        break;
    }
    case 'e':
    {
        movey -= move;

        break;
    }
    case 'z':
    {
        scalar += 0.1;
        noise();

        break;
    }
    case 'x':
    {
        scalar -= 0.1;
        noise();

        break;
    }
}

default:
    break;
}
if (scalar <= 0)
    scalar = 0;
glutPostRedisplay();
}

void reshape(int w, int h)
//reshape
//*****
//OpenGL reshape callback is called when the window is resized
{
    float size = 100;
    glViewport(0, 0, (GLsizei)w, (GLsizei)h); //respecify size of view port to
match new w and h
    glMatrixMode(GL_PROJECTION); //go into projection mode
    glLoadIdentity(); //reset projection matrix
    //glOrtho(-size, size, -size, size, 0.0, 200.0); //defines orthoganal
projection (parrallel
// viewing volume
// window limits of display
// left =0.0, right =2.0,
// bottom = 0.0, top = 2.0
// near =0.0, far = 100.0
    gluPerspective(90, (GLfloat)w / (GLfloat)h, 0.1, 1000); //redefine projection
matrix to match changes in w and h
    glMatrixMode(GL_MODELVIEW); //switch back to modelview mode
    glLoadIdentity(); //reset modelview matrix
    gluLookAt(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); //reset camera position
}

```

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800, 800);
    glutInitWindowPosition(100, 100);
    init();
    glutCreateWindow(argv[0]);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboardread);
    glutMainLoop();
    return 0;
}
```

HPC report

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

74/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

GRADING FORM: SMU GENERIC_GJ

GEORGE ANTHONY THOMAS

73.50

RESEARCH-INFORMED LITERATURE



Check formatting errors in references. Generally appropriate. Could have included more.

5

KNOWLEDGE AND UNDERSTANDING

A good review of the literature related to procedural



landscape generation using noise functions. A detailed explanation of Perlin noise technique together with example is missing here. Refs used appropriately. Cuda functionality discussed too.

11

ANALYSIS



Some good results providing results as expected - you might, in future work, consider the organisation of grids/blocks etc. Initialisation times are also of interest.

6.75

PRACTICAL APPLICATION AND DEP.



Some good work here both CUDA and CPU code evaluation in evidence. Timings reflect expectations. Good.

48

SKILLS FOR PROFESSIONAL PRAC.



Structured, abstract included. Refs appropriate. Res incomplete/formatting issues in places. Too few?

2.75