2015

322COM – Advanced Graphics Programming

PROCEDURAL GENERATION OF GAMING ENVIRONMENTS

SAMUEL MCKAY 4906379

Friday, December 11, 2015

Abstract

As game worlds become larger and larger, it is becoming less feasible for these environments to be individually hand-crafted. As such more and more research is being carried out into the procedural generation of content such as terrains, trees and sky boxes.

Introduction

The aim of this project is to produce a procedurally generated gaming environment. The task presents several challenging new algorithms to implement as well as learning how work with shaders and passing information between the central processing unit (CPU) and the graphics processing unit (GPU). This report will explain the main concepts and methods behind procedurally generating terrain and trees. It will also cover the use of OpenGL shaders to optimize the rendering of the scene.

Procedural generation of environments allows developers to create vast and varied terrains considerably faster than traditional hand crafting methods. This combined with the procedural generation of trees and other environmental elements means that’s new worlds can be fully generated very quickly.

Background theory

Diamond-Square

In order to create the procedurally generated terrain it is necessary to first understand the main algorithm behind it. The tried and proven Diamond-Square algorithm is one of better known algorithms for procedural generation (Boxley, 2011). This algorithm is easiest implemented in a recursive function, unlike most recursive functions it is better to calculate the points as at each level, rather than on the way back out. This results in a semi-recursive function but is necessary to ensure all the points on the previous layer have been set before calculating the next set of points. The diamond square algorithm consists of two main components: the diamond part; and the square section.

Diamond

The Diamond section involves taking the average value of the corner sections and adding a random value (within a pre-defined range). This value then becomes the height of them middle point between the four corners.

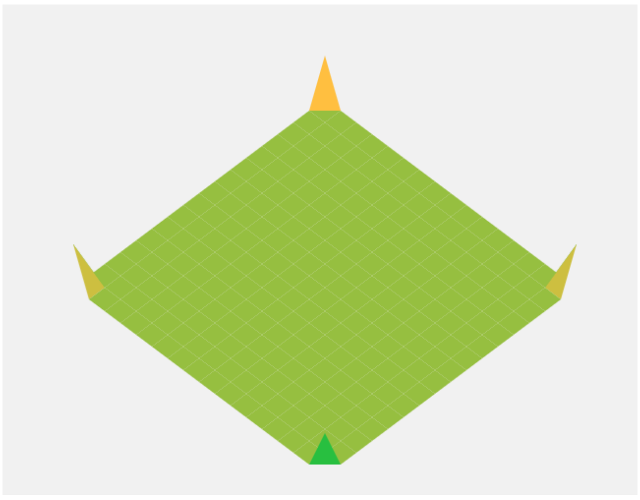


Figure 1 Diamond Step (Boxley, 2011).

Square

For the square step a value is assigned to each of the midpoints of the lines making up the square. For each line calculate the midpoint of the corner values and add another random value to this point.

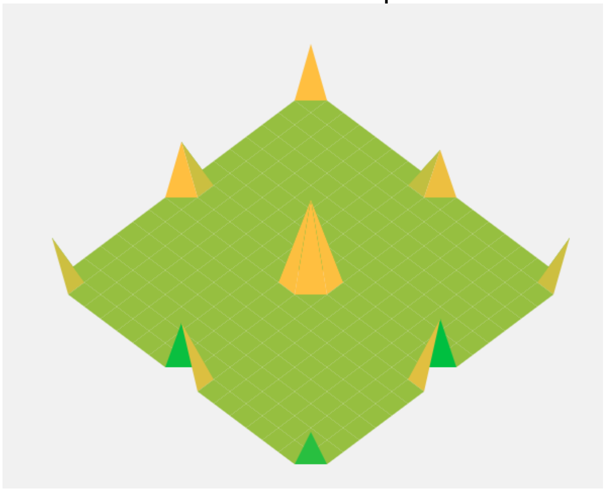


Figure 2 Diamond & Square step applied (Boxley, 2011).

Fractal Trees

Another important aspect of creating a gaming environment is to populate the world with trees. The use of procedural generation is the quickest way to produce realistic trees. This will allow the rapid creation of multiple variations of a tree without having lots of assets and will make the world seem less repetitive.

The tree generation is another recursive algorithm that essentially works by generating each branch using depth first method. Each branch is a copy of the initial “trunk” section and is then scaled, translated and rotated into the corresponding position. This is then stored into an array of Vertex’s which contain all of the tree data. Along with the index buffer this is then passed into the OpenGL shaders ready to be drawn to the screen.

OpenGL shaders

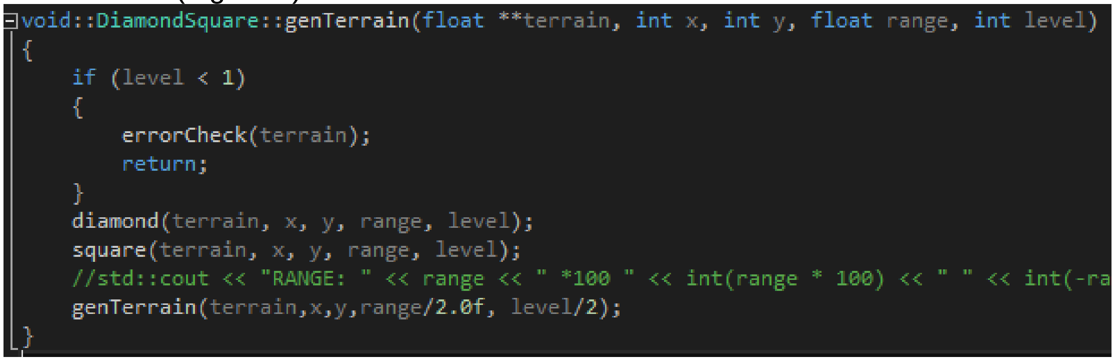
In order to optimise the code, it is ideal to perform the majority of calculation on the graphics hardware. This is because it is very powerful containing far more processing power than the main CPU. Due to differences in graphics hardware and drivers, AMD GPUs are unable to run the OpenGL shaders with the current setup and as such the code discussed will only run with an Nvidia GPU. These shaders can be very difficult to work with at first and will require a lot of research to fully understand how to use them. However, they are a very effective way of programming and will massively boost the performance of this program (Guha, 2014).

Methodology and results

Although initially the task of procedural generation seems like a complex task the individual aspects of the project were fairly simple to implement.

Terrain

Using a recursive function is the easiest way to procedurally generate terrain using the Diamond-Square algorithm. This is because the last step is when the distance between two squares is 1 or less. This acts as the stopping point for the recursive function, and prevents it from attempting to put data into an area of the array that doesn't exist (Figure 3).

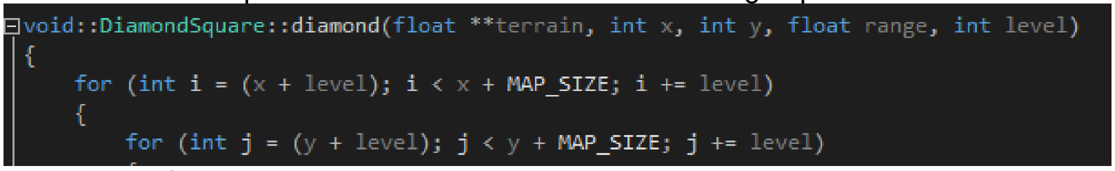


```
void::DiamondSquare::genTerrain(float **terrain, int x, int y, float range, int level)
{
    if (level < 1)
    {
        errorCheck(terrain);
        return;
    }
    diamond(terrain, x, y, range, level);
    square(terrain, x, y, range, level);
    //std::cout << "RANGE: " << range << " *100 " << int(range * 100) << " " << int(-ra
    genTerrain(terrain,x,y,range/2.0f, level/2);
}
```

Figure 3 Recursive Terrain Generation

Diamond

To implement the diamond step a nested for loop (Figure 4) was used to obtain the y values of the corners of the current square. These values were then used to calculate an average value to which a random number was added. This random number was generated with a function that takes in the current height value which is degraded over time. This helps to smooth our terrain and avoids large spikes.

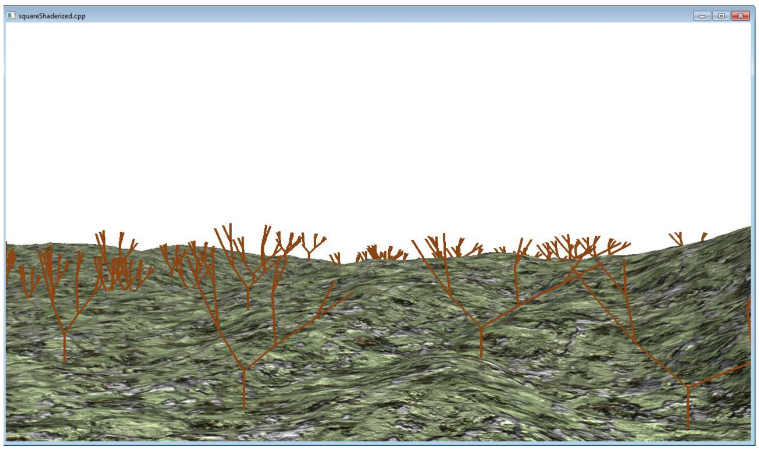


```
void::DiamondSquare::diamond(float **terrain, int x, int y, float range, int level)
{
    for (int i = (x + level); i < x + MAP_SIZE; i += level)
    {
        for (int j = (y + level); j < y + MAP_SIZE; j += level)
```

Figure 4 Nested for loop to obtain corner vertices

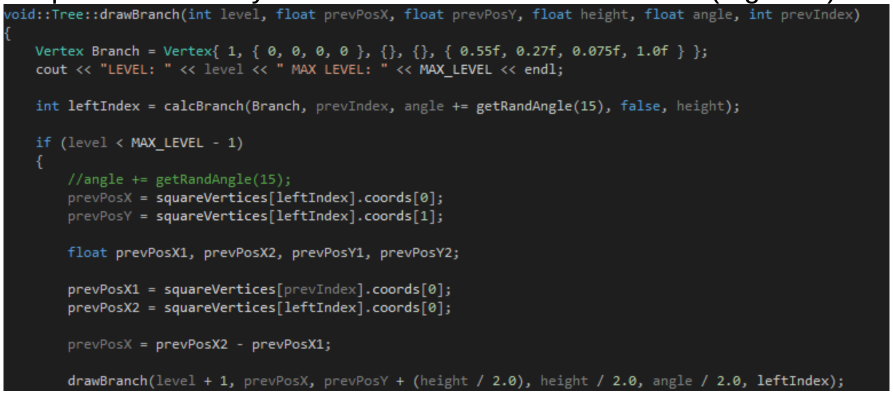
Square

The square function uses the same loop as the diamond step to obtain the corner values but this time only calculates the average between two corners at a time. This result along with another random value is used to set the value for the midpoint between each corner.



Tree

Another recursive function is used to generate fractal trees this is because a MAX_LEVEL variable can be used to control the amount of branch levels within the tree (EDU, 2015). The way this works is by having a drawBranch function as the recursive function and control whether left or right branches are being drawn. Another function, calcBranch, is used to carry out the actual branch calculations. This keeps the code very neat and therefore easier to read (Figure 5).

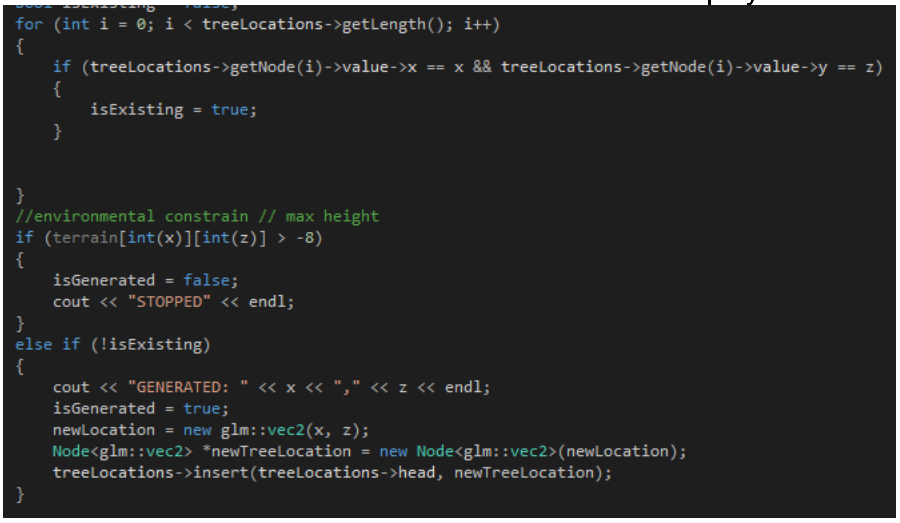


```
void::Tree::drawBranch(int level, float prevPosX, float prevPosY, float height, float angle, int prevIndex)
{
    Vertex Branch = Vertex{ 1, { 0, 0, 0, 0 }, {}, {}, { 0.55f, 0.27f, 0.075f, 1.0f } };
    cout << "LEVEL: " << level << " MAX LEVEL: " << MAX_LEVEL << endl;

    int leftIndex = calcBranch(Branch, prevIndex, angle += getRandAngle(15), false, height);

    if (level < MAX_LEVEL - 1)
    {
        //angle += getRandAngle(15);
        prevPosX = squareVertices[leftIndex].coords[0];
        prevPosY = squareVertices[leftIndex].coords[1];

        float prevPosX1, prevPosX2, prevPosY1, prevPosY2;

        prevPosX1 = squareVertices[prevIndex].coords[0];
        prevPosX2 = squareVertices[leftIndex].coords[0];

        prevPosX = prevPosX2 - prevPosX1;

        drawBranch(level + 1, prevPosX, prevPosY + (height / 2.0), height / 2.0, angle / 2.0, leftIndex);
```

Figure 5 drawBranch() function in Tree Class

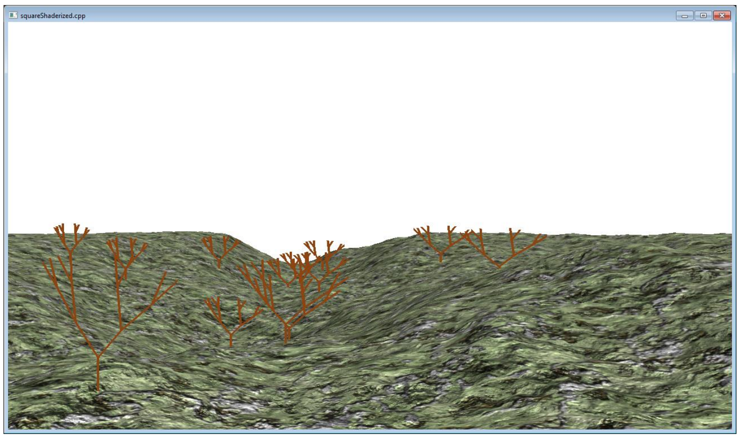
Environmental constraints

To prevent trees from being drawn at the top of a mountain, environmental constraints are necessary to stop it drawing if the y value is too high (Figure 6). This helps to achieve better realism, because if a tree was sticking out of the top of a mountain it would look bizarre. This in turn breaks the player's immersion.



```
for (int i = 0; i < treeLocations->getLength(); i++)
{
    if (treeLocations->getNode(i)->value->x == x && treeLocations->getNode(i)->value->y == z)
    {
        isExisting = true;
    }


}
//environmental constrain // max height
if (terrain[int(x)][int(z)] > -8)
{
    isGenerated = false;
    cout << "STOPPED" << endl;
}
else if (!isExisting)
{
    cout << "GENERATED: " << x << "," << z << endl;
    isGenerated = true;
    newLocation = new glm::vec2(x, z);
    Node<glm::vec2> *newTreeLocation = new Node<glm::vec2>(newLocation);
    treeLocations->insert(treeLocations->head, newTreeLocation);
}
```

Figure 6 Environmental Constraints



Conclusion

In conclusion creating procedural elements has been a tough challenge, especially whilst understanding the new OpenGL shaders and how to access these. This was not helped by the incompatibility of the AMD graphics cards. Better understanding and a large amount of research would be required in order to have this running on an AMD graphics card.

A better approach would have been to use the same vertex buffer data so that we don't have to pass a lot of empty data into the shader. In a larger program this would eventually reduce the frame rates by considerable amounts. Both of the classes created in this project are reusable and can easily be ported into new applications.

I now have a much better understanding of vertex and fragment shaders and would feel comfortable implementing this for a games studio as part of a graphics engine. I am very happy with results of this project and will use an updated version as part of a portfolio in the future.

References

Boxley, P., 2011. *Paul Boxley*. [Online]
Available at: http://www.paulboxley.com/blog/2011/03/terrain-generation-mark-one
[Accessed 11 December 20145].
EDU, D., 2015. *Fractal Trees.* [Online]
Available at: http://davis.wpi.edu/~matt/courses/fractals/trees.html
[Accessed 12 December 2015].
Guha, S., 2014. *Computer Graphics Through OpenGL.* 2nd ed. s.l.:CRC Press.

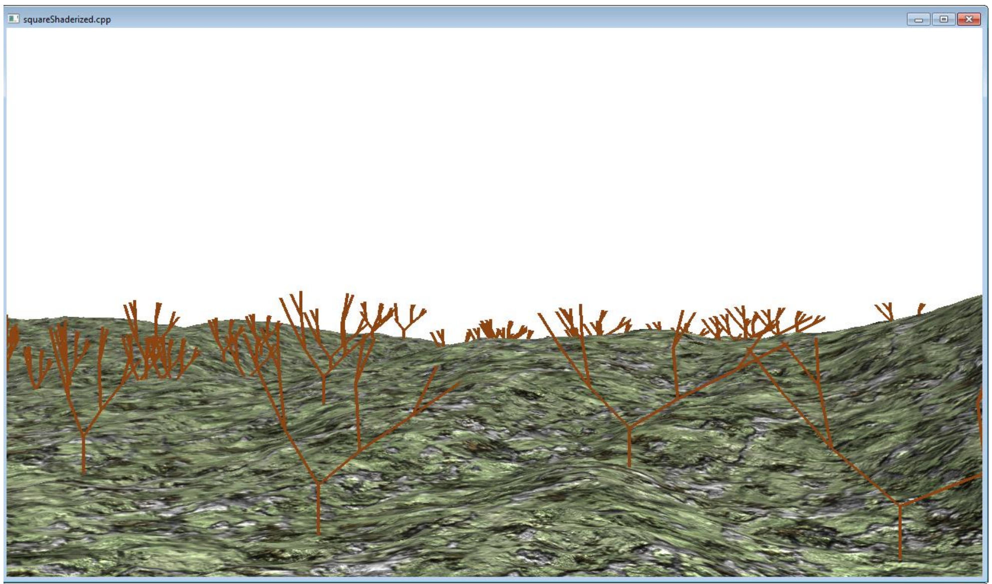
Appendices

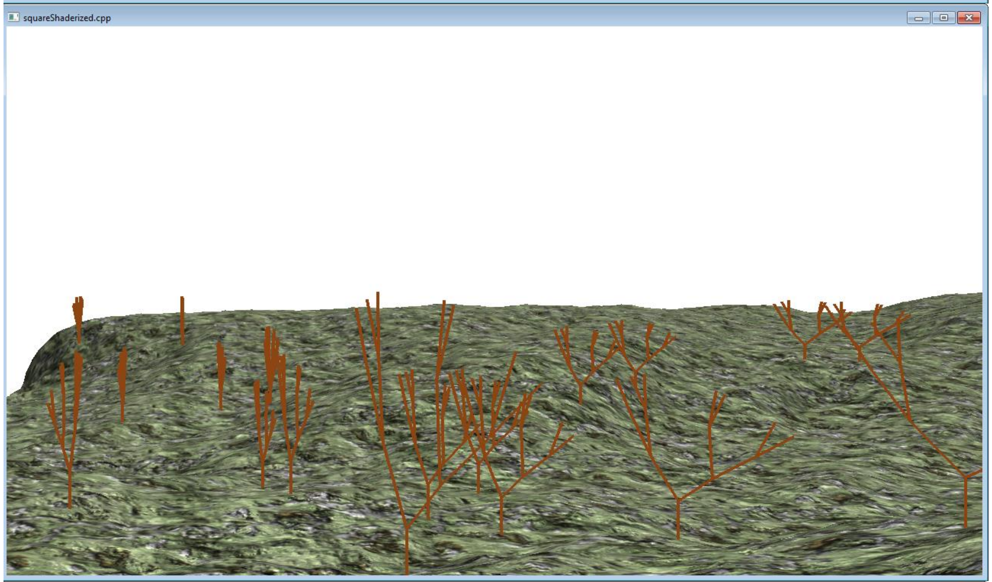
Appendix 1 – Install File
Once you have downloaded the github zip file you should then be able to run the installshield wizard to begin setup and installation. The installshield exe file can also be found here:

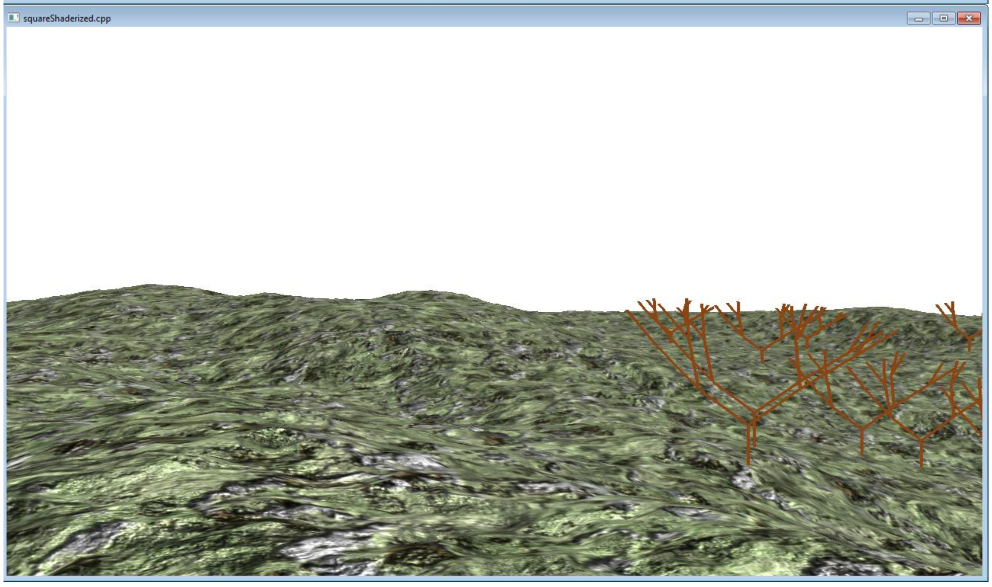
Appendix 2 - GitHub
https://github.com/sam-mckay/322_Project_Two_New

Appendix 3 - Images

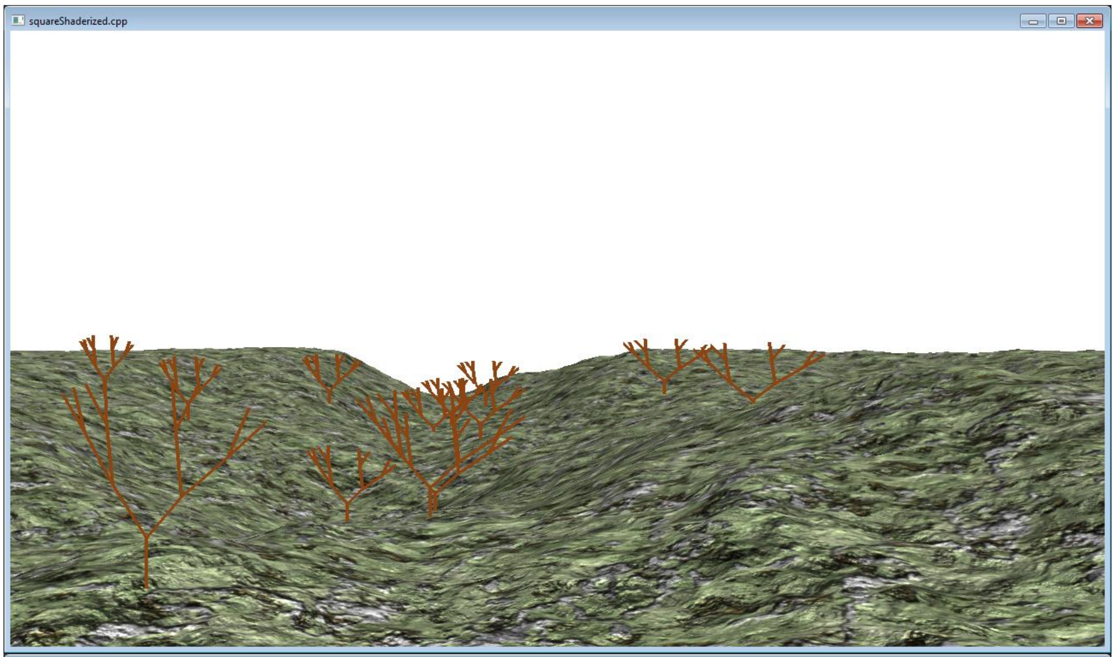
Seed 4

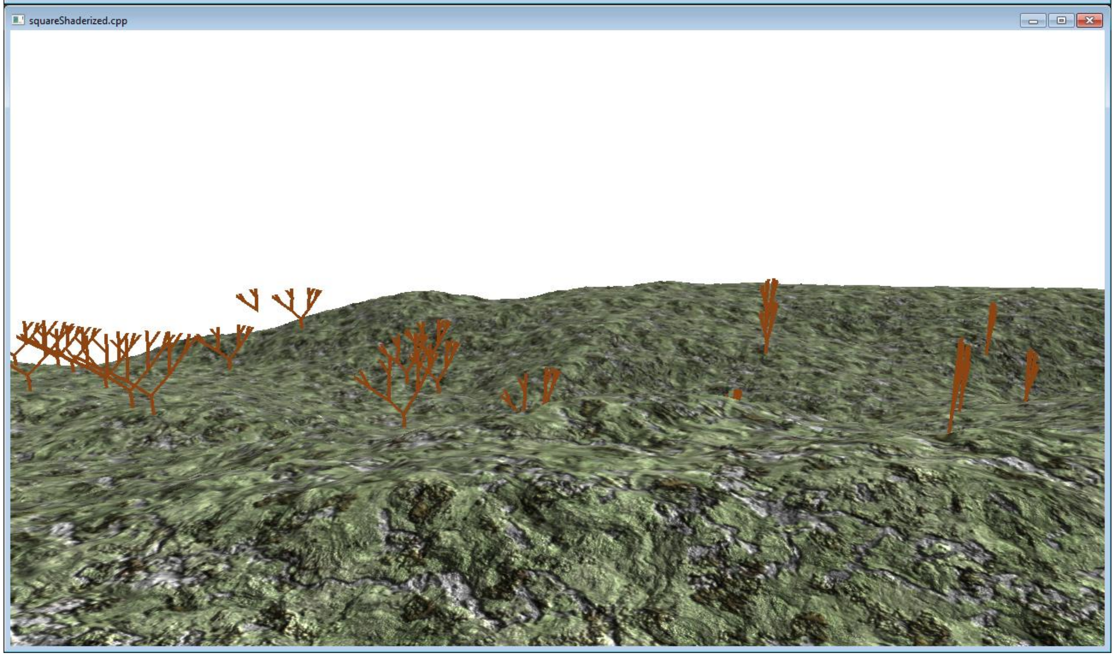




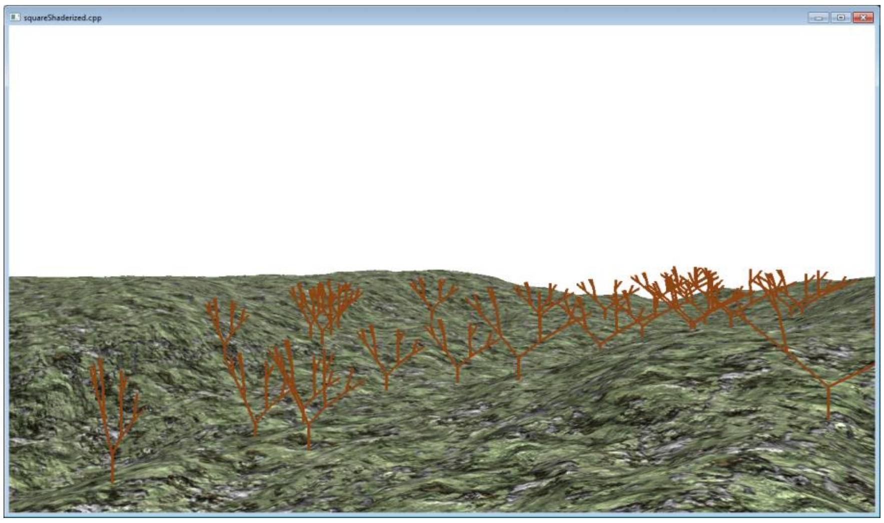


Seed 15654

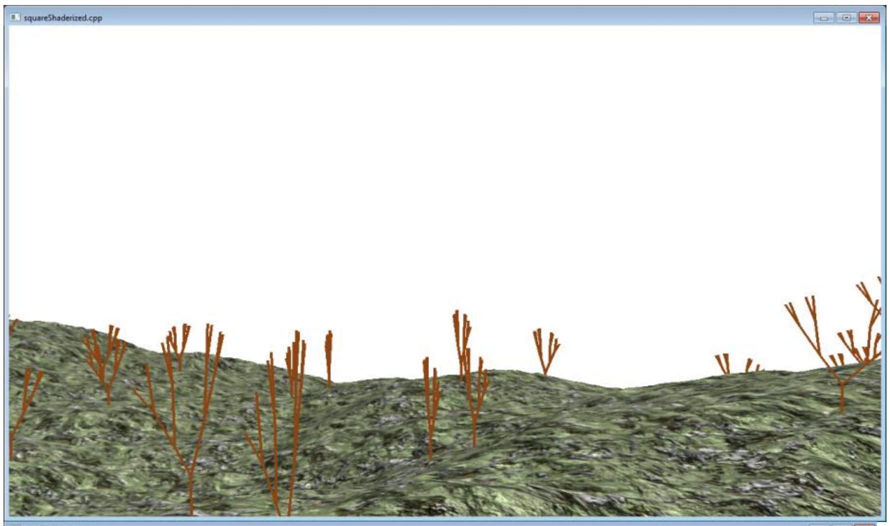


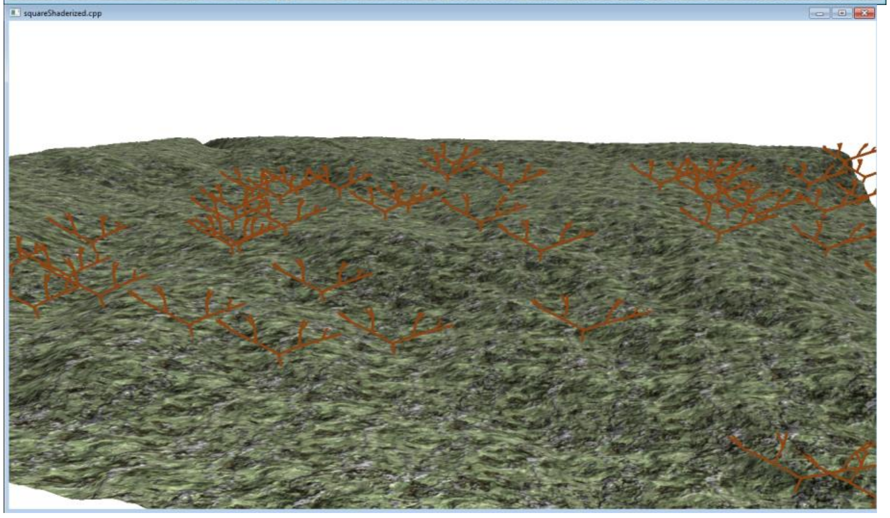


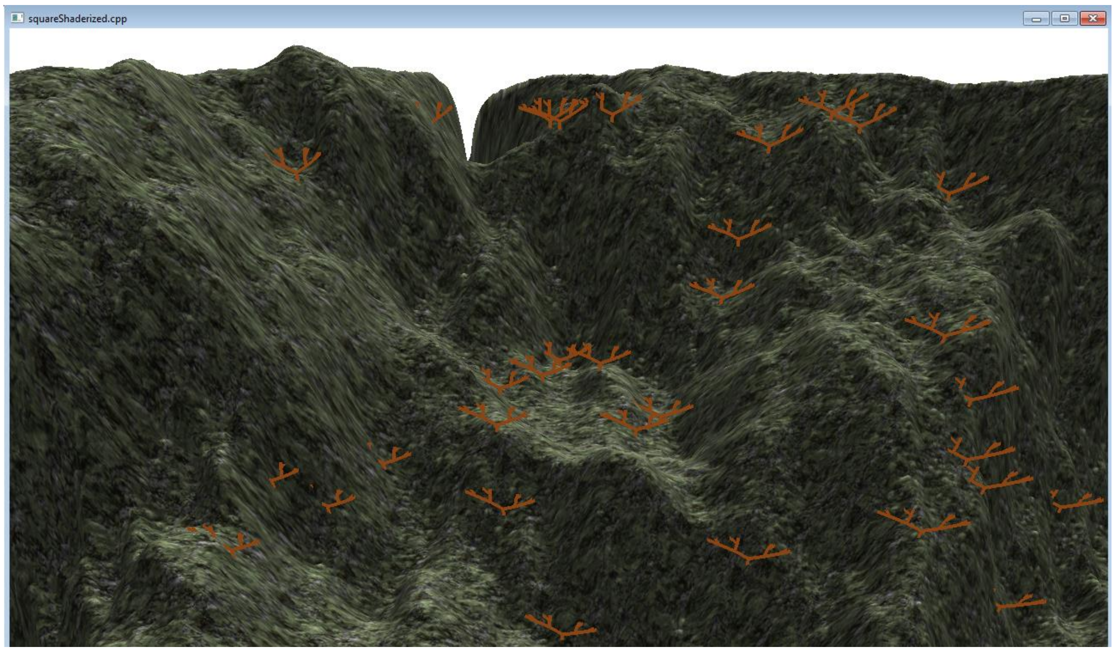
Seed 7546445646545645658



Seed 864621






squareShaderized.cpp