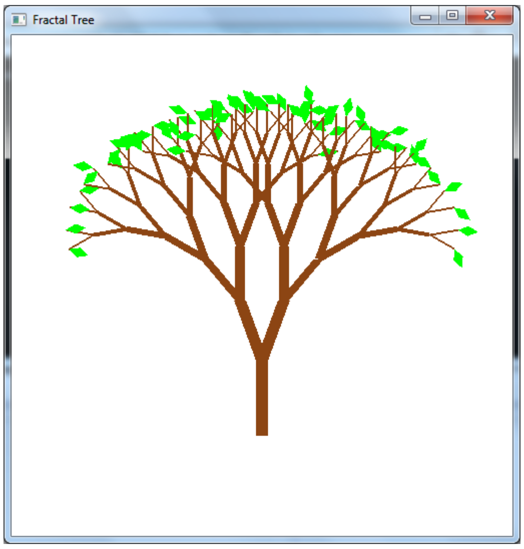
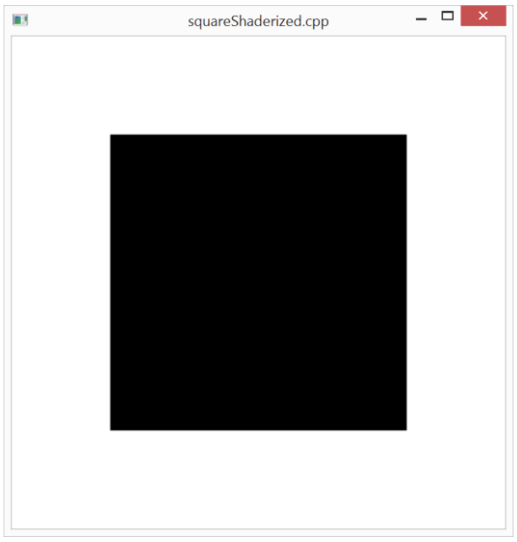
Advanced Graphics Programming

Workshop Eight – Fractal Tree Generation (2D)

In this workshop you will create a fractal tree using the principles of self-similarity and recursion. To start with you will work in 2D (as shown below), this algorithm can be extended to 3D by using 3D rotations.



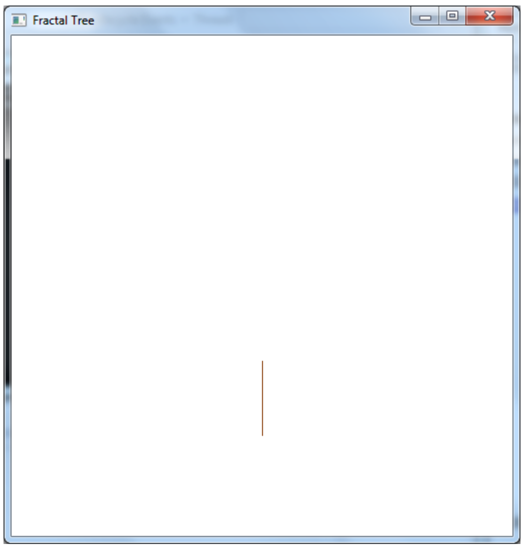
Step 1: Start with SquareShaderizedPlus Visual Studio solution on Moodle which is similar to SquareShaderized from Chapter 20 with additional error checking and using the GLM library to view the scene in 2D using orthographic projection. Run the project and it should display a black square on a white background.



Step 2: Draw your tree trunk, which can simply be a vertical line segment. The location and orientation of the trunk can be specified by you, it can be placed freely anywhere on a plane (2D) or space (3D). The tree trunk in the example below is a line from (0,-30) to (0,-15), the line width is 1 and the colour is brown (0.55,0.27,0.075). Here is a handy table of RGB colour values: http://www.rapidtables.com/web/color/RGB_Color.htm

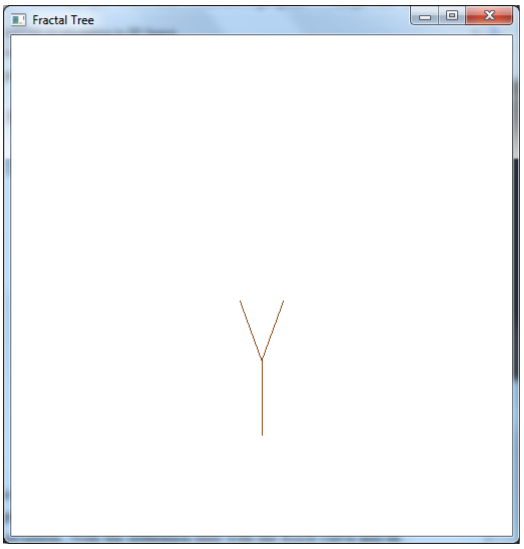
Hint: You will need to replace the vertices of the square with the vertices for the tree trunk and in the drawScene function change the GL_TRIANGLE_STRIP for the appropriate primitive.

Hint: You will also need to update the parameters of the ortho function so that you can see your tree. e.g. `projMat = ortho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);`



Step 3: The next step is to specify a rule to generate branches. The branches are rigidly associated with the trunk, they have the same initial location and orientation. The branches for a tree in this workshop are a V-shaped two-segment polyline located atop the trunk, the length of each segment is a specified fraction (RATIO) of the length of the trunk, with a specified angle (ANGLE) between them. There are four sub-steps to calculate the new verticies, which are described on the next page.

Hint: You can add the vertices for the branches to the same VBO and VBA as the trunk.



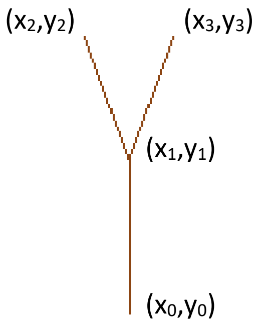


Figure 1 In the example scene RATIO (r) = 0.85 and ANGLE (α) = 40.

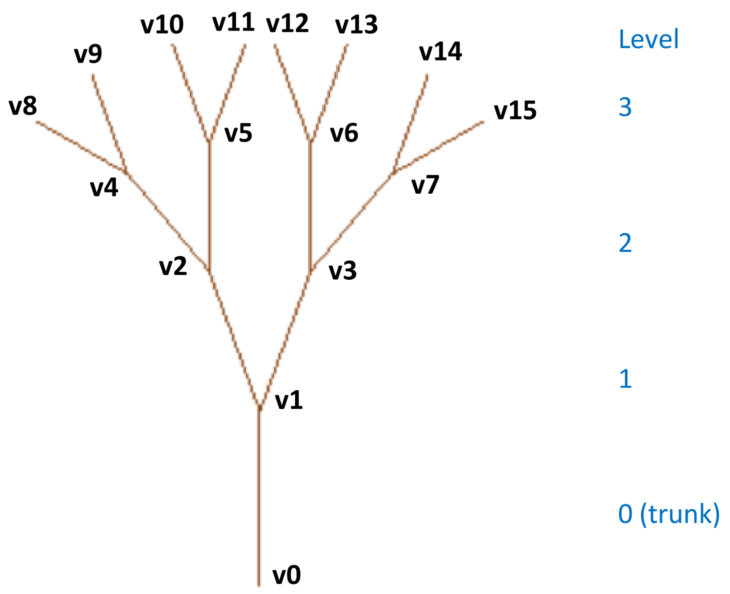
1) Scale the trunk (or previous branch) by ratio, r so that the next branch is smaller than the previous:
    a. $x_s = (x_1 - x_0) * r$
    b. $y_s = (y_1 - y_0) * r$
2) Rotate anticlockwise (default) by **half** of the angle to get the left branch:
    a. $x_l = \cos\left(\frac{\alpha}{2}\right) * x_s - \sin\left(\frac{\alpha}{2}\right) * y_s$
    b. $y_l = \sin\left(\frac{\alpha}{2}\right) * x_s + \cos\left(\frac{\alpha}{2}\right) * y_s$

    Hint: In C++ cos() and sin() are calcuated in radians so you will need to convert degrees to radians before calling it, glm has a function to do this!

3) Move the left branch to the end of the previous branch:
    a. $x_2 = x_1 + x_l$
    b. $y_2 = y_1 + y_l$

4) Repeat the above steps but rotate clockwise $(-\alpha/2)$ to get the right branch.

Step 4: To draw the tree correctly you will need to use index buffer data to instruct OpenGL how to draw the lines between each vertex (v). The following is an example of ordering the vertex data which allows branches further up the tree to be drawn thinner.



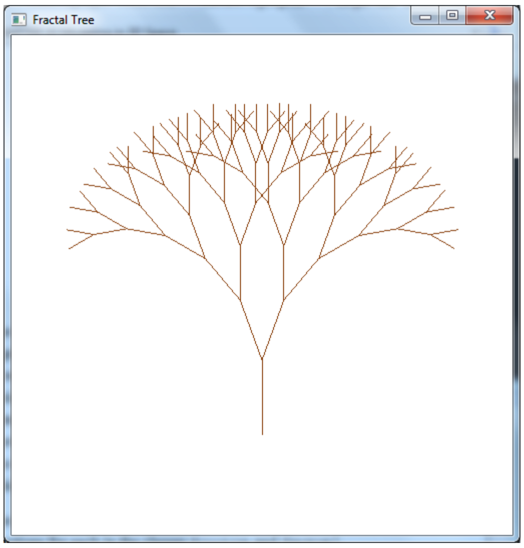
If you are going to generate trees of a fixed number of levels (e.g. 1) you could hard code the index buffer data as follows:

```
const int BRANCH_INDEX_COUNT = 3;
uint branchIndexData[3] = {0,1, 1,2, 1,3};
```

And then draw the tree with the following command:

```
glDrawElements(GL_LINES, BRANCH_INDEX_COUNT, GL_UNSIGNED_INT, branchIndexData);
```

Step 5: You are now ready to recursively produce the fractal tree to any desired level by adding succeeding branches at each level. Level 0 is the trunk and each level adds a layer of branches, using the same equations from step 3. The example below is level 6.



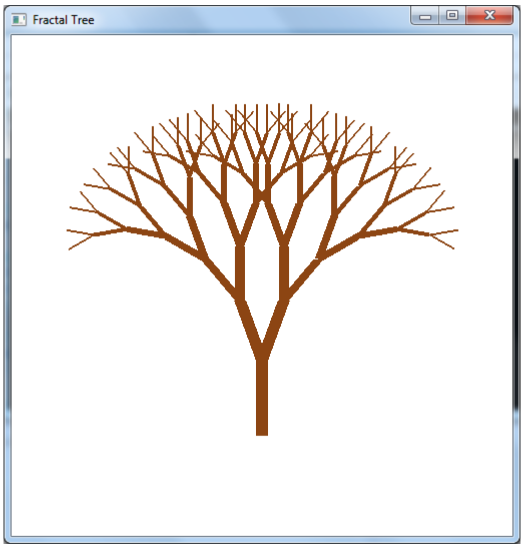
Hint: you will need to update your index buffer data to include the new branches. This is straightforward to hardcode up to level 3 but after that it will become tedious. The alternative is to generate the index buffer data which can be done with the following formula:

```
newLeftVertexIndex = previousVertexIndex x 2;
newRightVertexIndex = previousVertexIndex x 2 + 1;
```

These formulas will work for all cases except the trunk where the vertex data indices for v0 and v1 can be explicitly set (0 and 1).

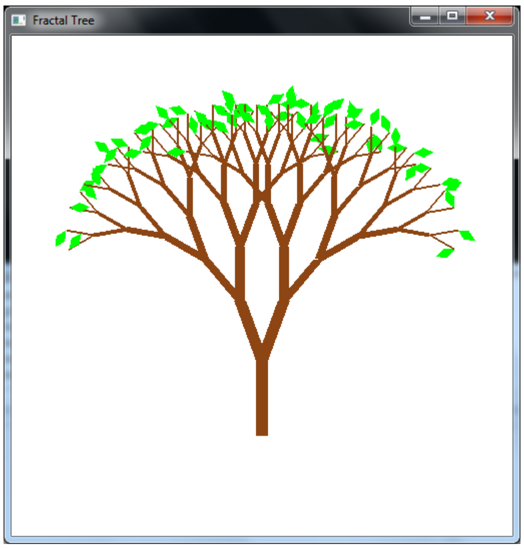
Step 6: Now you have completed the fractal algorithm you can improve the appearance by altering the line thickness depending on the level, with branches at successive levels drawn thinner.

WARNING: Increasing the line width only works if you are NOT using forward compatibility mode!



Step 7: The final step in creating the 2D tree is to embellish with leaves, which are pairs of triangles at random angles at the ends of the top-level branches.

Hint: You will need to add a new VBO and VBA for the leaves. See example below for how to create multiple VBOs/ VBAs, you will also need to update the draw commands and the vertex shader: D:\OpenGL\ExperimenterSource\Chapter20\ballAndTorusShaderized



Step 8: Create variations of the tree by adjusting the RATIO and ANGLE parameters as well as changing the original trunk length. Create a non-uniform tree by adding randomisation so that not all branches are drawn.

Gobal variables:

```
const int NUMPOINTS = 1000;
const int MAXLEVEL = 6; //maximum tree level
const float R = 0.85;
const float alfa = 40.0;
static Vertex TrunkVertices[NUMPOINTS] = {};
static Vertex LeafVertices[NUMPOINTS] = {};
```

Function 1: Create single branch of tree

```
void computeSingleBranch(float angle,float x0,float y0,float x1, float y1, float &x2,float &y2)
{
        float xs, ys, xl, yl;
        xs = (x1 - x0)*R;
        ys = (y1 - y0)*R;
        xl = cos(angle / 2.0)*xs - sin(angle / 2.0)*ys;
        yl = sin(angle / 2.0)*xs + cos(angle / 2.0)*ys;
        x2 = x1 + xl;
        y2 = y1 + yl;
}
```

Step 2: Recursively compute branches of tree

```
void recurComputeBranch(int depth, int index,float angle, vector<vec2> BasePts, vector<vec2> BrPts)
{
        int i,size;
        float x2,y2;
        vec2 ttPt;
        vector<vec2> NewBasePts, NewBrPts;

        if (depth > MAXLEVEL) return;

        size = BasePts.size();
        if (size == 0) return;

        for (i = 0; i < size; i++)
        {
                computeSingleBranch(angle, BasePts[i].x, BasePts[i].y, BrPts[i].x, BrPts[i].y, x2, y2);
                TrunkVertices[index].coords[0] = x2;
                TrunkVertices[index].coords[1] = y2;
                TrunkVertices[index].coords[2] = 0.0;
                TrunkVertices[index].coords[3] = 1.0;
                index++;
                NewBasePts.push_back(BrPts[i]);
                ttPt.x = x2; ttPt.y = y2;
                NewBrPts.push_back(ttPt);

                computeSingleBranch(-angle, BasePts[i].x, BasePts[i].y, BrPts[i].x, BrPts[i].y, x2, y2);
                TrunkVertices[index].coords[0] = x2;
                TrunkVertices[index].coords[1] = y2;
                TrunkVertices[index].coords[2] = 0.0;
                TrunkVertices[index].coords[3] = 1.0;
                index++;
                NewBasePts.push_back(BrPts[i]);
                ttPt.x = x2; ttPt.y = y2;
                NewBrPts.push_back(ttPt);
        }

        depth++;
        recurComputeBranch(depth, index, angle, NewBasePts, NewBrPts);
}
```

Step 3: Create whole tree

```
void createTree(void)
{
        int i,count;
        float x0,y0,x1,y1,x2,y2,xs, ys, xl, yl,angle;
        angle = 40.0*3.1415926 / 180.0;
        vec2 ttPt;
        vector<vec2> BasePts, BranchPts;

        for (i = 0; i < NUMPOINTS; i++)
        {
                TrunkVertices[i].colors[0] = 0.55;
                TrunkVertices[i].colors[1] = 0.27;
                TrunkVertices[i].colors[2] = 0.075;
                TrunkVertices[i].colors[3] = 1.0;
        }
        //major trunk
        TrunkVertices[0].coords[0] = 0.0;
        TrunkVertices[0].coords[1] = -30.0;
        TrunkVertices[0].coords[2] = 0.0;
        TrunkVertices[0].coords[3] = 1.0;
        ttPt.x = TrunkVertices[0].coords[0]; //x0
        ttPt.y = TrunkVertices[0].coords[1]; //y0
        BasePts.push_back(ttPt);

        TrunkVertices[1].coords[0] = 0.0;
        TrunkVertices[1].coords[1] = -15.0;
        TrunkVertices[1].coords[2] = 0.0;
        TrunkVertices[1].coords[3] = 1.0;
        ttPt.x = TrunkVertices[1].coords[0]; //x1
        ttPt.y = TrunkVertices[1].coords[1]; //y2
        BranchPts.push_back(ttPt);

        i = 2;
        recurComputeBranch(0,i, angle, BasePts, BranchPts);
        //create leaf vertices
```