Tran Tran – Geometric Modelling Spring 2025 Project

Generation and Rendering of Fractal Terrain

Paper Reference: https://worldcomp-proceedings.com/proc/p2013/CGV4061.pdf

From a height map, the algorithm will read the elevation data at each pixels and generate a vertex at the height data. However, how can we generate a terrain without a heightmap that look realistic.

Landscapes are self-similar in that looking at the landscape at different scales exhibit same basic characteristics. For example, a big mountain contains several peaks and each peak when zoomed out contains smaller peaks. Fractals are similar in this nature. Therefore, this characteristic allows us to represent terrain as fractals

Fractals are often used in landscapes to make it realistic. In terrain generation, close-by points are close to each other so in generating vertex, we will take into account the proximity of their neighboring vertex.

Diamond-Square algorithm to have each point's height depending in four directions of the grid. The algorithm is iterative process including:

1. Assign random altitudes to the four corners of a grid.
2. Diamond step: Average the four corners and add a random perturbation evenly distributed between –ri and ri. Assign this to the midpoint of the four corners.
3. Square step: For each diamond produced, average the four corners and add a random perturbation with the same distribution.
4. Repeat this step

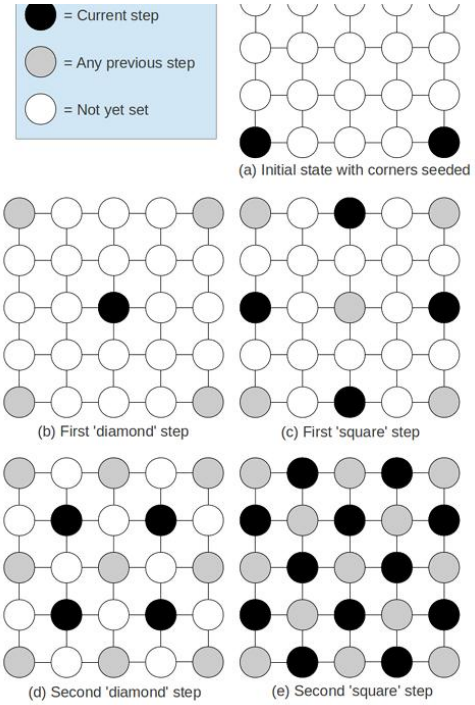
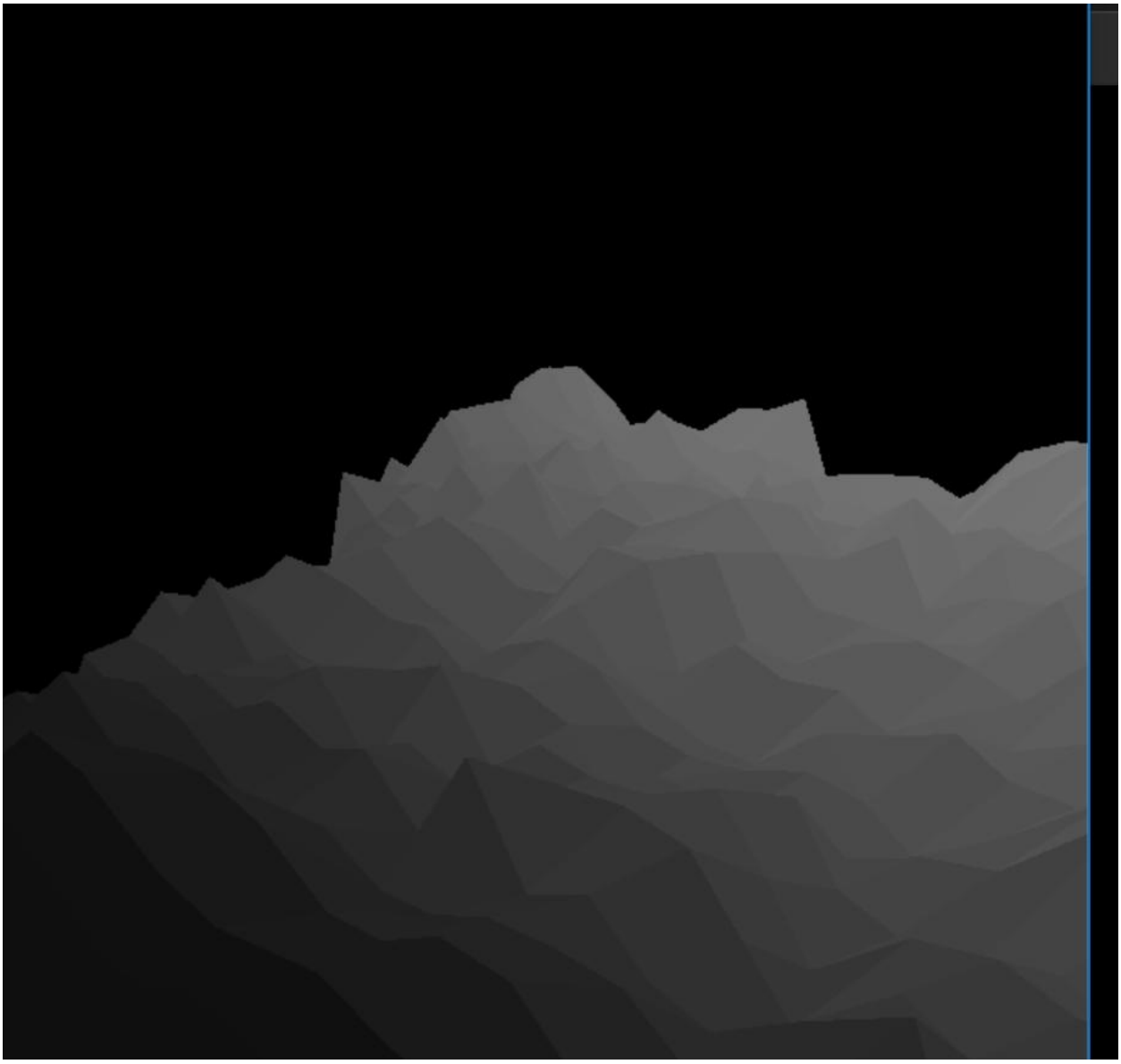



Fig. 3: Steps of the diamond-square algorithm

Code:

```
int matrix[grid_size][grid_size] = { 0 };
void generate_terrain() {
    int scale = 0.1f;
    glColor3f(red: 0.3f, green: 0.8f, blue: 0.3f);  // green terrain
    int n = grid_size - 1;

    matrix[0][0] = rand() % max_height;
    matrix[0][n] = rand() % max_height;
    matrix[n][0] = rand() % max_height;
    matrix[n][n] = rand() % max_height;
    srand(_Seed: time(_Time: NULL));
    for (int l = n; l > 0; l = l/2) {
        //Diamond step
        int r = 0;
        int c = 0;

        while ((r < n) || (c  < n)) {
            int m_r = r + l / 2;
            int m_c = c + l / 2;
            matrix[m_r][m_c] = ((matrix[r][c] + matrix[r + l][c] + matrix[r][c + l] + matrix[r + l][c + l]) / 4) + rand() % 20;
            //square step
            matrix[r][m_c] = (matrix[r][c] + matrix[m_r][m_c] + matrix[r][c + l]) / 3;//top
            matrix[m_r][c] = (matrix[r][c] + matrix[m_r][m_c] + matrix[r + l][c]) / 3;//left
            matrix[m_r][c + l] = (matrix[r][c + l] + matrix[m_r][m_c] + matrix[r + l][c + l]) / 3; //right
            matrix[r + l][m_c] = (matrix[r][c] + matrix[m_r][m_c] + matrix[r + l][c + l]) / 3;
            if (c + l < n) {
                c += l;
            }
            else if (r + l < n) {
                c = 0;
                r += l;
            }
            else {
                r += l;
                c += l;
            }
        }
    }
    const int tile_size = 5;


    for (int r = 0; r < n; r++) {
        glBegin(modo: GL_TRIANGLE_STRIP);
        for (int c = 0; c <= n; c++) {
            //printf("%d %d ; %d %d ; ", r, c, r+1, c);
            glVertex3f(x: r*tile_size, y: matrix[r][c], z: c*tile_size);
            glVertex3f(x: (r + 1) * tile_size, y: matrix[r + 1][c], z: c * tile_size);
        }
        glEnd();
    }

}
```

**Result:**



Rendering Fractal Terrains on Approximated Spherical Surfaces:

The algorithm combines spherical approximation by tetrahedron subdivision and midpoint displacement.

1. Tetrahedron subdivision: The more we subdivide the tetrahedron, the better approximated the sphere is
2. As we subdivide the tetrahedron, we assign the check if the edge already exists. If it doesn't, we assign random displacement offset to the midpoint. to generate another face of the triangle
3. Checking if the edge already exists:

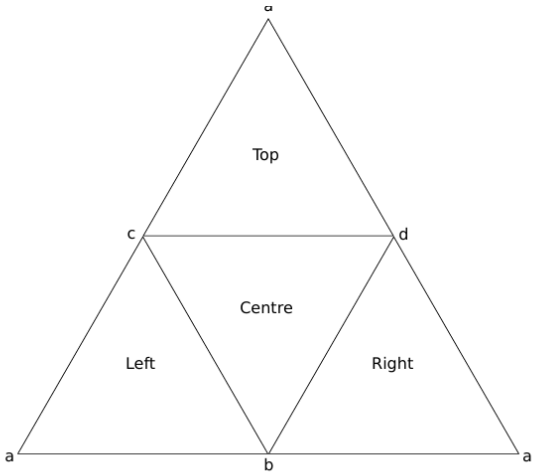


Fig. 6: The net of a regular tetrahedron

**Implementation:**

Vertex structure contains position x, y, z

Node structure: reference each triangle face. Each node will store reference to its left, top, center and right child node, its parent node, and the three vertices defining the triangle, the three midpoint vertices the define the subdivision process, its node type

QuadTree structure: defines the whole QuadTree structure with a root node.

**Step:**

1. Initialize a new tetrahedron with two parameters: height variance and roughness constant. The height variance of a child node is equal with its parent multiplying by the roughness constant:

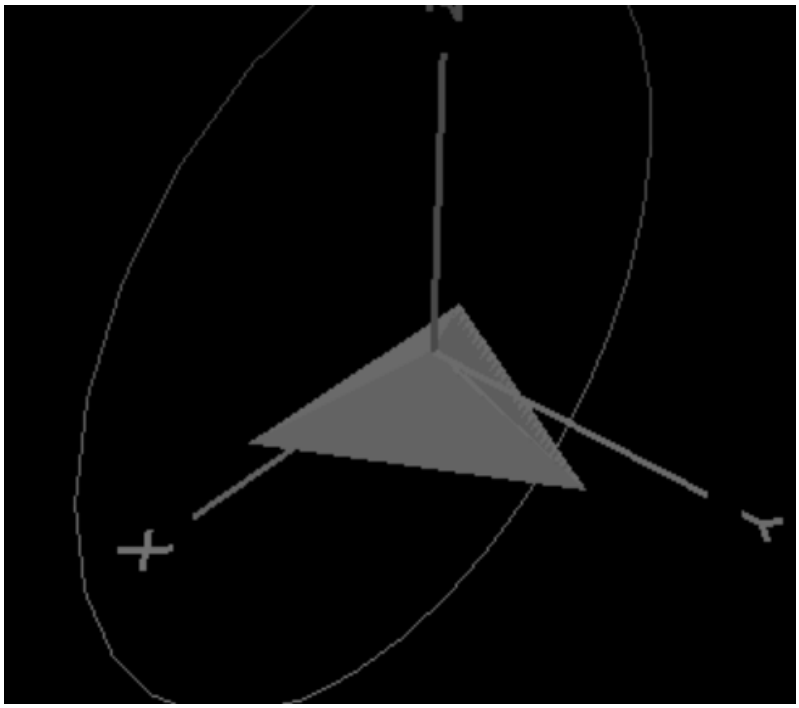
```
QuadTree::QuadTree(const double variance, const double roughness_) {
    Vertex *vA = new Vertex(a);
    Vertex* vB = new Vertex(b);
    Vertex* vC = new Vertex(c);
    Vertex* vD = new Vertex(d);
    roughness = roughness_;
    root = new Node(nullptr, NULL, vA, vA, vA);
    root->height_range = variance;

    root->ab = vD;
    root->bc = vB;
    root->ca = vC;
    root->left = new Node(root, 'L', vC, vB, vA);
    root->top = new Node(root, 'T', vA, vD, vC);
    root->center = new Node(root, 'C', vB, vC, vD);
    root->right = new Node(root, 'R', vD, vA, vB);
    subdivideTree();

}
```

2. Subdivide the tree: initialize a queue with the list of nodes in order to perform bread first traversal. Once we reach the leaf node, we will subdivide the node (triangle)

```
void QuadTree::subdivideTree() {
    int i = 0;
    while (i < 5) {
        queue<Node*> q;
        q.push(_Val: root);
        Node* t = nullptr;
        while (!q.empty()) {
            t = q.front();
            q.pop();
            if (t->left != nullptr) {
                q.push(_Val: t->left);
                q.push(_Val: t->center);
                q.push(_Val: t->top);
                q.push(_Val: t->right);
            }
            else {
                subdivideNode(curr: t);
            }
        }
        i++;
    }
}
```

3. Subdivide the node: This function check to see its neighbor triangles have already been assigned a midpoint displacement. If not, it will assign a midpoint with displacement variance. Then the function will create four new children nodes left, right, top, center.

**Result:** The current code doesn't work well to generate the terrain on the sphere. Its current result just generated a regular tetrahedron Debugging: I am printing out the list of node vertice and realize that the vertice getting smaller and smaller. I think it is related to normalization issue. But I ran out



of time to be able to fix it.