Mathematics and Graphics I

Assignment 1: An implementation of L-Systems

Louis Bennette

My implementation of L-Systems can be found at https://github.com/Deahgib/LSystems
A video showcasing the program can be found at https://youtu.be/rglL_ebrodY

1. Definition

An L-System is a fractal like graphical construction. L-Systems were devised in the late 1960s by Aristid Lindenmayer. An L-System is considered as a grammar, it is an array or string of characters that each specify an instruction to construct a geometrical structure. An L-System is created from an axiom, which is an initial string of characters specifying the initial conditions of that L-System. Additionally an L-System is iterated (or evolved), by using a set of rules used to specify how the characters mutate to different new strings. Each iteration of the L-System creates a new, typically longer, string. The characters are also used as a set of instructions to create our geometry.

For example, the Pythagoras tree L-System:
The axiom is '0'
The rules are:

- 0 -> 1[0]0
- 1 -> 11

The geometry's instructions are:

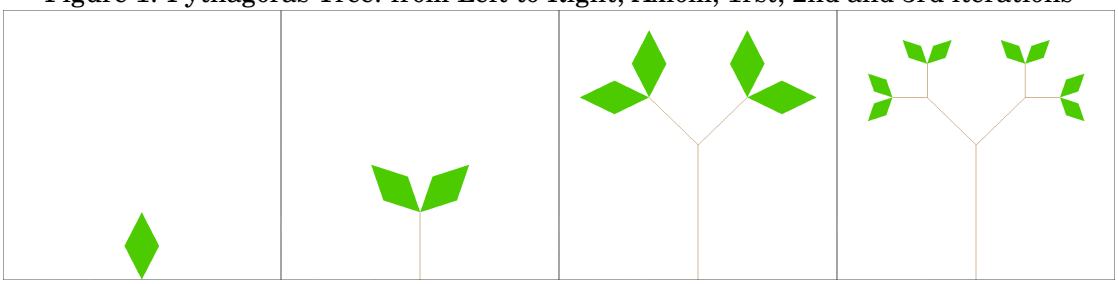
- 0 : draw a leaf line segment
- 1 : draw a branch line segment
- a [: push our current bearing onto a stack, then rotate -45 degrees
- a] : pop our last bearing from the stack, then rotate +45 degrees

As we iterate through the string starting from an axiom we create new iterations of the L-System. See Figure 1 for graphical output.

- Axiom: 0
- 1rst Step: 1[0]0
- 2nd Step: 11[1[0]0]1[0]0
- 3rd Step: 1111[11[1[0]0]1[0]0]11[1[0]0]1[0]0

When we use our geometry's instructions to build the L-System we see can see a structure emerging, these structures tend to have recursive properties due to the recursive process used to create them. The recursive structure has fractal like properties of repeating patterns.

Figure 1: Pythagoras Tree: from Left to Right, Axiom, 1rst, 2nd and 3rd iterations



2. Implementation

To implement the L-Systems tree drawing I have used the octet framework (by Andy Thomason). I will be drawing my tree's with opengl provided by octet. The meshes are generated using a turtle rendering technique.

The user needs to be able to specify the rules of an L-System, I have done this by creating a .frac file format which contains the variables necessary. Specifically, the axioms, the rules and the angle amounts for the control characters. Additionally the file allows the user to specify the x, y coordinates of the origin of the tree (between -1 and 1).

The software is divided into three classes, namely LSystems, TreeString and Turtle. LSystems is the class responsible for handling any I/O. It extends the octet::app class which in turn provides the class with a main loop used for user input and rendering. Any user input is checked for and handled here, when the user specifies a new tree to load the $load\_fractal\_file$ function is called and the new L-System parameters are loaded and set into their appropriate locations.

2.1 LSystems class

The LSystems class is also responsible for rendering the tree to the view, it does this in two passes, the first to render the branches and the second to render the leaves. Additionally, the L-Systems is a controller class, it makes the calls to the subsequent TreeString and Turtle classes and dictates the flow of the overall software.

2.2 TreeString class

The TreeString class is responsible for holding the current step of the current string. When a new file is loaded the axiom from the file is input to the class as well as the rules for that system. The only subsequent actions are to perform an iteration on the string or to reset the axiom. When the do_step() function is called the characters of the string are looped over. If the character is a rule character (control character in the software) then the output or rule conversion is looked up in a table. The output is appended to a new string. If the character is not a control character, it is simply passed on and appended to the new string as it's a geometric construction character. The current string is set to our new string and can be accessed from the LSystems class using get_string().

2.3 Turtle class

The Turtle class is responsible for generating a list of vertices which will be used later to render the structure using openGL. The structures can be made up of leaves and/or branches. This is important because our leaves are drawn using triangles and our branches are drawn using lines. For this reason the Turtle class generates two lists of vertices, one for the branches and one for the leaves. openGL will render a buffer with line data and a buffer with triangle data. This also, allows us to set different colours to the branches or leaves using different shaders for each buffer.

The turtle starts with an origin of the structure and a rotation. The default rotation is 0 degrees, a line with 0 rotation is a vertical line in positive y, 90 degrees is a horizontal line in positive x.

When the user specifies any changes in the system's rendering, i.e. angles, origin or the input string have changed, a new set of vertices are rendered. The characters of the string are looped over and based on our geometry rules the different actions take place. A point is rendered by projecting where it should be based on the position of it's connected point and the angle the turtle is facing. This is done using a technique similar to polar to cartesian conversion.

Listing 1: Turtle.h: *get_next_projected_point*()

```
//  Quad 4 | Quad 1
// --------|--------
//  Quad 3 | Quad 2
int quadrant = ( ( (int)facing_angle / 90) % 4) + 1;
float remainder_angle = fmod(facing_angle, 90);
remainder_angle = get_radians(remainder_angle);
float dx, dy;
switch (quadrant) {
  case 1:
    dx = unit_length * math::sin(remainder_angle);
    dy = unit_length * math::cos(remainder_angle);
    break;
  case 2:
    dx = unit_length * math::cos(remainder_angle);
    dy = -unit_length * math::sin(remainder_angle);
    break;
  case 3:
    dx = -unit_length * math::sin(remainder_angle);
    dy = -unit_length * math::cos(remainder_angle);
    break;
  case 4:
    dx = -unit_length * math::cos(remainder_angle);
    dy = unit_length * math::sin(remainder_angle);
    break;
```

```
}
point next_point;
next_point.x = last_point.x + dx;
next_point.y = last_point.y + dy;
```

The leaves are generated using the same next projected point however two additional points are added with a perpendicular offset from the midpoint of our line, this creates a diamond which is rendered in green for our leaves.

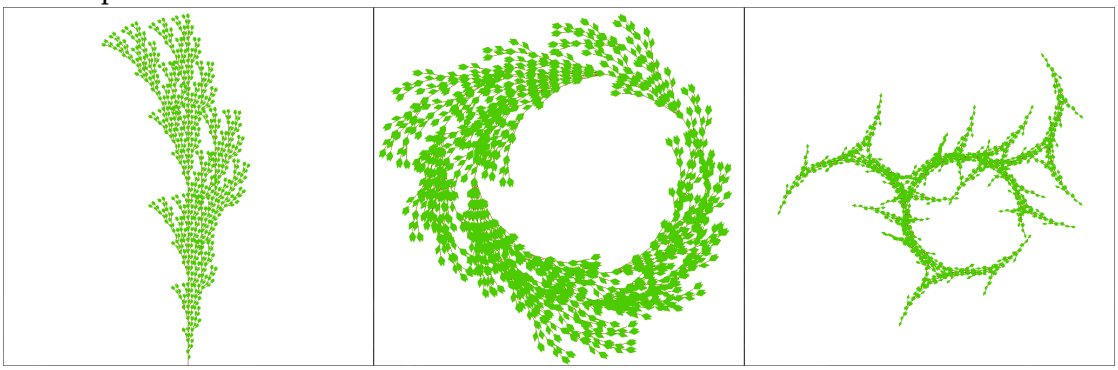
Listing 2: Descriptive Caption Text

```
float dx = end.x - start.x;
float dy = end.y - start.y;
// Mid point of our vector
point mid;
mid.x = start.x + dx * 0.5f;
mid.y = start.y + dy * 0.5f;
// Normal vector to our line vector.
// | 0 -1 | normal vector transform
// | 1  0 |
point normal;
normal.x = -dy;
normal.y = dx;
point vert1;
vert1.x = mid.x + normal.x * 0.25f;
vert1.y = mid.y + normal.y * 0.25f;
point vert2;
vert2.x = mid.x - normal.x * 0.25f;
vert2.y = mid.y - normal.y * 0.25f;
```

2.4 Rendering

Once the structure has been generated from the string we use opengl's vertex buffer objects to render our branches and leaves. The openGL rendering calls are made in the LSystems class inside the $draw_world$ function. I did not use the classical video game rendering approach by using a matrix applied to my mesh for rotation or translation for example. Instead I render directly to the -1.0, 1.0 default opengl screen space. This is because the task would not benefit directly from the ability to rotate and translate the entire structure. For now the object can simply be generated within the -1 to 1 constraints. This is beneficial for any future code reuse, the output of the turtle class could be considered as a mesh and used as a game asset.

Figure 2: Tree-b: Showcasing how different angles change the tree's appearance for the same input



2.5 The file reader

The identity for an L-System is input from a .frac file format. The .frac file format requires a strict ordering of the parameters per line. Specifically, the characters used for branches (branch:), the characters used for leaves (leaf:), the axiom (axiom:), the conversion rules (rules:), the angle specified by the '+' and '-' characters (angle:), the angle specified by the '[' and ']' characters (pp-angle:) and finally the origin of the structure to be generated. (origin:)

Listing 3: Example input file

```
branch:1
leaf:0
axiom:0
rules:1>11,0>1[0]0
angle:0
pp-angle:45
origin:0,-1
```

2.6 Input

The user can swap between the trees they would like to render by using keyboard hotkeys,

The L-Systems' parameters can be changed in real time by the user. The user can change the turtle's angles used to generate the structure. Different angles can create different emergent patterns. See Figure 2

The user can also change the location of the origin. And additionally the user can activate the animate state which will continuously change the angle every frame.

Listing 4: Input keys

```
1 -> 0 keys: Load a fractal file

Left arrow: move the origin left by 0.1
Right arrow: move the origin right by 0.1
Up arrow: move the origin up by 0.1
Down arrow: move the origin down by 0.1

SHIFT + Left arrow: change the angle amount by -1
SHIFT + Right arrow: change the angle amount by +1

CTRL + Left arrow: change the push pop angle amount by -1
CTRL + Right arrow: change the push pop angle amount by +1

a: Animate the change in angle every frame by changing the angle
amount in increments of +0.1
```