Ellie Ansell
Computing for Animation 1
NCCA Bournemouth University, level I

4. Develop a system that provides an interface to allow l-system based modelling of plant-like structures using openGL. (The L-system should read in the grammar of the system from a file)

Aim: To develop a 3D interface allowing l-system based modelling of plants.
Primary research: Aristid Lindenmayer, a biologist studying plant growth of algae, devised l-systems to describe plant growth of simple multicellular organisms in 1968. They have recursive self-similarity, meaning that when given an initial state, multiple rules can then be applied to a string to expand it iteratively, forming a pattern. Variations of l-systems include parametric systems which fill a space given initial algorithms, and microsorium linguaeforme can replicate the formation of cells. L-systems can create natural forms, so aren't limited to plants.

The basic principle
Each symbol of the string is expanded in accordance to a collection of production rules and axioms, e.g. 'A'-> 'AB' where 'A' is the axiom, and 'A'->'AB' is the rule applied when the string 'A' is read. Multiple rules can be used, so 'B'-->'A' too. As shown below:

```
n=0:          A          Initial string
             / \
n=1:        A   B        The rule A->AB was applied, and the output string is 'AB'
           /|    \
n=2     A B     A        From 'AB', A->AB and B->A.
       /| |     |\
n=3: A B A      A B      The rules are continued to be applied to the returned string recursively.
```

With every recursion the string is expanded and so grows exponentially - this output string is the input to the next iteration where the rules are applied again, and so on. More than one rule can be applied at a time because the grammar replaces the working set in parallel, unlike a language being defined in sequence where only one rule can be applied at a time. As a result, fractals are made as well as organic plant forms by adjusting the rules for Axioms along with other parameters, such as iterations and angles. Examples:

Roast, K.[1]

The symbol 'F' may mean 'move forwards', '+' and '-' rotate, and other symbols, such as 'X' for the terdragon curve may have no effect asides being used to control the curve. In the case of plants, in order to return to a particular branch, stacks are used to store vector coordinates. Symbols such as '[' are used to push values on top of a stack, where they're stored until the symbol ']' is read, whereby the values are popped off the stack and the 'turtle' returns to that location, then continues to read the string.
Initial parameters required to be inputted by the user are the iterations, angle, starting axiom and rules. To achieve more variety, some considerations are required to vary the curve created, such

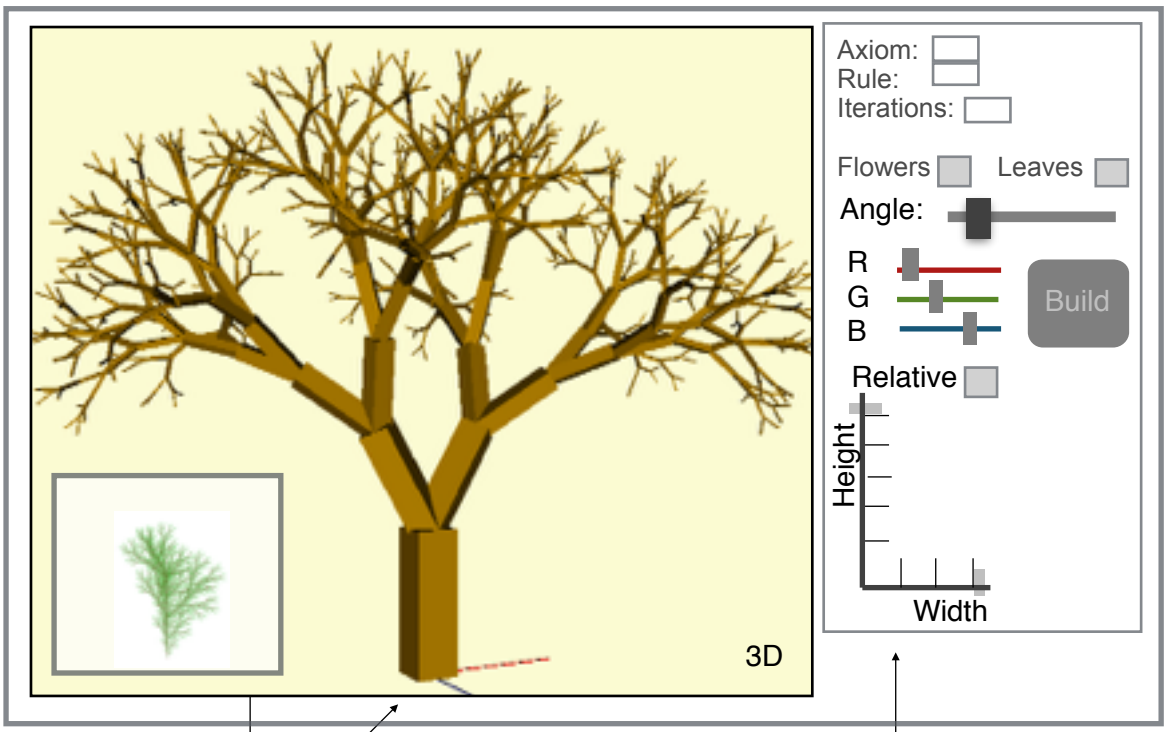
as randomisation for realism by using stochastic elements. With stochastic elements, each plant varies subtly and so a forest could be modelled with similar types of plants. When making trees, the user may want branches to scale, rotate in Y slightly, change colour or end in flowers and leaves for further aesthetic appeal with randomisation applied to each.

Usability

The user should be able to build the plant, then manipulate it in 3D space using keyboard operations, such as scrolling with the mouse to decrease/increase the number of iterations as if the tree is growing, and holding h+scrolling could change the height, and respectably for width. Clicking and dragging on the 3D window to the left/right could change the angle. They could pan around the tree using operations similar to maya (alt+click) such as mouse scrolling to zoom the camera in and out. If any of the parameters change, the program should rebuild the tree in realtime with the changes..

Adjustable parameters include: angle, colour (RGB), branch width, height, whether there are flowers or leaves, the colour of these, whether it's random so the angle varies slightly for each rotation.

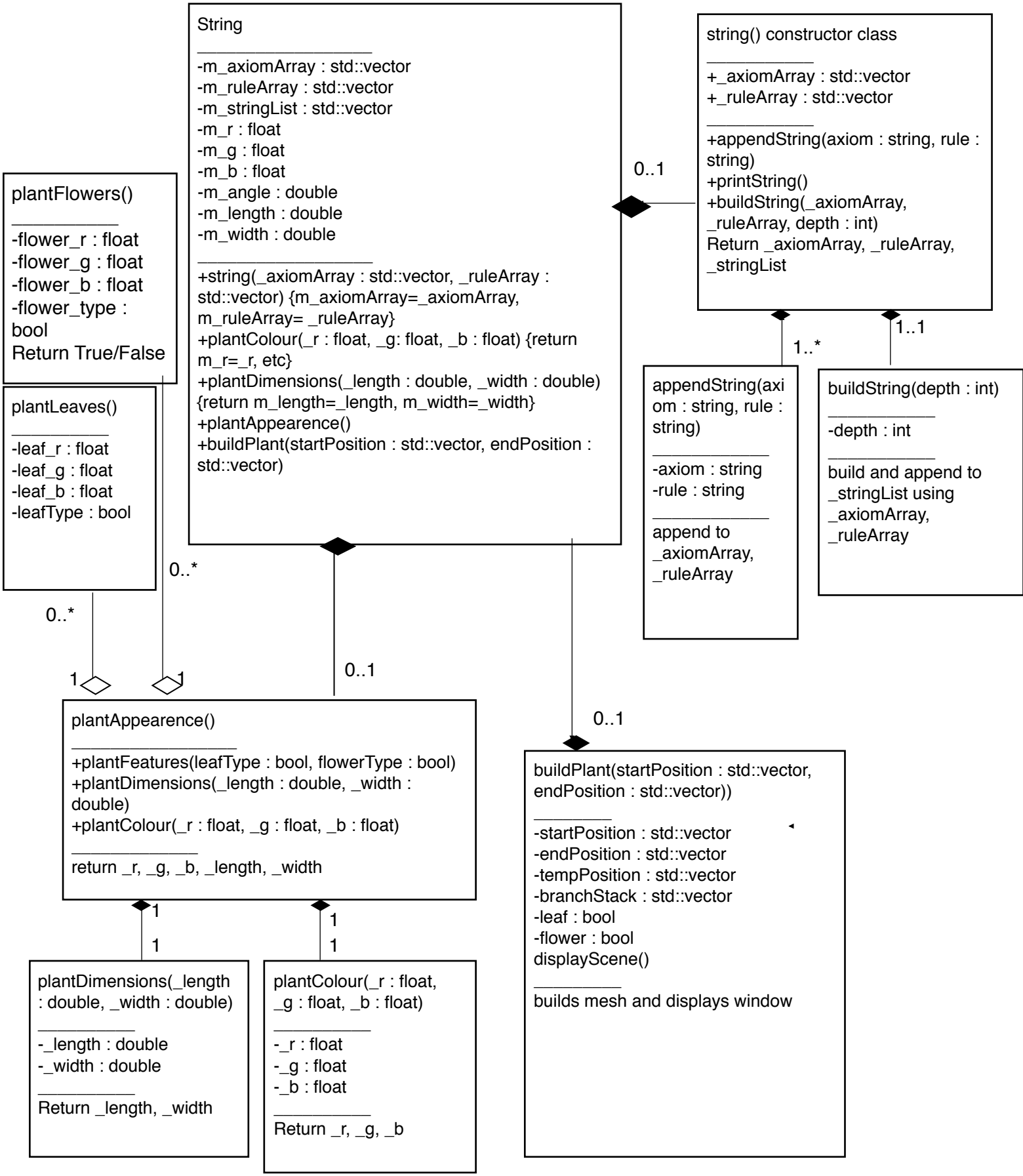
There could be an option to export the finished product as a .obj, so the user could then make another plant with stochastic attributes which have a probability of occurring so they could be used in a forest scene of plants with similar heritage.
Below shows how the interface could look, with controls on one side and a viewport on the other:



Sliders to adjust colour, angle

Builds string in 2D and 3D space. The camera rotates around the centre of the 3D plant.

Variable height and width sliders. By selecting 'relative', the base of the polygon will scale relative to the height. Otherwise, the base and height can be scaled independently.


String
-m_axiomArray : std::vector
-m_ruleArray : std::vector
-m_stringList : std::vector
-m_r : float
-m_g : float
-m_b : float
-m_angle : double
-m_length : double
-m_width : double
+string(_axiomArray : std::vector, _ruleArray : std::vector) {m_axiomArray=_axiomArray, m_ruleArray= _ruleArray}
+plantColour(_r : float, _g: float, _b : float) {return m_r=_r, etc}
+plantDimensions(_length : double, _width : double) {return m_length=_length, m_width=_width}
+plantAppearence()
+buildPlant(startPosition : std::vector, endPosition : std::vector)
string() constructor class
+_axiomArray : std::vector
+_ruleArray : std::vector
+appendString(axiom : string, rule : string)
+printString()
+buildString(_axiomArray, _ruleArray, depth : int)
Return _axiomArray, _ruleArray, _stringList
0..1
1..*
1..1
appendString(axiom : string, rule : string)
-axiom : string
-rule : string
append to _axiomArray, _ruleArray
buildString(depth : int)
-depth : int
build and append to _stringList using _axiomArray, _ruleArray
plantFlowers()
-flower_r : float
-flower_g : float
-flower_b : float
-flower_type : bool
Return True/False
plantLeaves()
-leaf_r : float
-leaf_g : float
-leaf_b : float
-leafType : bool
0..*
0..*
1
1
0..1
plantAppearence()
+plantFeatures(leafType : bool, flowerType : bool)
+plantDimensions(_length : double, _width : double)
+plantColour(_r : float, _g : float, _b : float)
return _r, _g, _b, _length, _width
0..1
buildPlant(startPosition : std::vector, endPosition : std::vector))
-startPosition : std::vector
-endPosition : std::vector
-tempPosition : std::vector
-branchStack : std::vector
-leaf : bool
-flower : bool
displayScene()
builds mesh and displays window
1
1
1
1
plantDimensions(_length : double, _width : double)
-_length : double
-_width : double
Return _length, _width
plantColour(_r : float, _g : float, _b : float)
-_r : float
-_g : float
-_b : float
Return _r, _g, _b

Psuedo code

A mother class contains all the information of the plants, and has two children: one which builds the string, and another which designs the plant.
The string() child receives axioms and rules the user inputs and these are appended to their own arrays, using vector types for dynamic arrays:

```
#include <vector>
{
std::vector<int> axiomArray;
axiomArray.push_back(6);
std::cout<<axiomArray[0];
}
```

Another alternative is to use lists as std::list<int> axiomList; , but accessing an element requires inefficient code, whereas vector arrays can append and access elements efficiently.

The derivative buildString() function then inherits these arrays and takes the user's input of "depth", and builds a string using two for loops shown below. It iterates through the elements in 'axiomArray', checking if the string is the same as its element, and if true, then replaces the string's element with the respective element from 'ruleArray' by using two "for" loops. The returned value is then recursively iterated through again until the depth is 0, and the string has been extended.

```
buildString(depth : int)
{
if (depth>0):
{ for (i==0; i++; i<(length of _axiomArray))
   { for (j==0; i++; i<(length of _stringArray))
      { if (_stringArray[j]==_axiomArray[i]) { stringList[j]=_axiomArray[i] }
   }
} return buildString(_stringArray, depth-1)
}
```

An initial problem found with this s how to convert a concatenated string into individual elements. If the user inputted 'A->AB', given the input string {'A', 'B'}, if the rule was applied it would produce {'AB', 'B'} when ideally i'd want {'A', 'B', 'B'} so that each character could be accessed separately on the next iteration. Before returning the string i'd need to separate each character in the array to individual elements.

The second child collects information about the plant such as colour, angle, length, width, whether it's randomised and if there are leaves/ flowers. When all the data is collected, buildPlant() inherits this information and iterates through the string to build the plant/ structure using "if", "else if" conditions. It contains startPosition, endPosition and "temp" is used to store information so the next branch doesn't begin where the first began.
If using vector curves to make the branches:

```
{
for (i==0; i++; i<(length of stringList))
{ if (i=='A')
   { temp=endPoint; endPoint's Yvalue +=10; startPoint=temp;
     draw branch between startPoint and endPoint; }
   else if (i=='F')
   { drawFlower(); }
   else if (i=='[')
   { append coordinates of endPosition to branchEndStack }
   else if (i==']')
   {branchEndStack = startPosition }
```

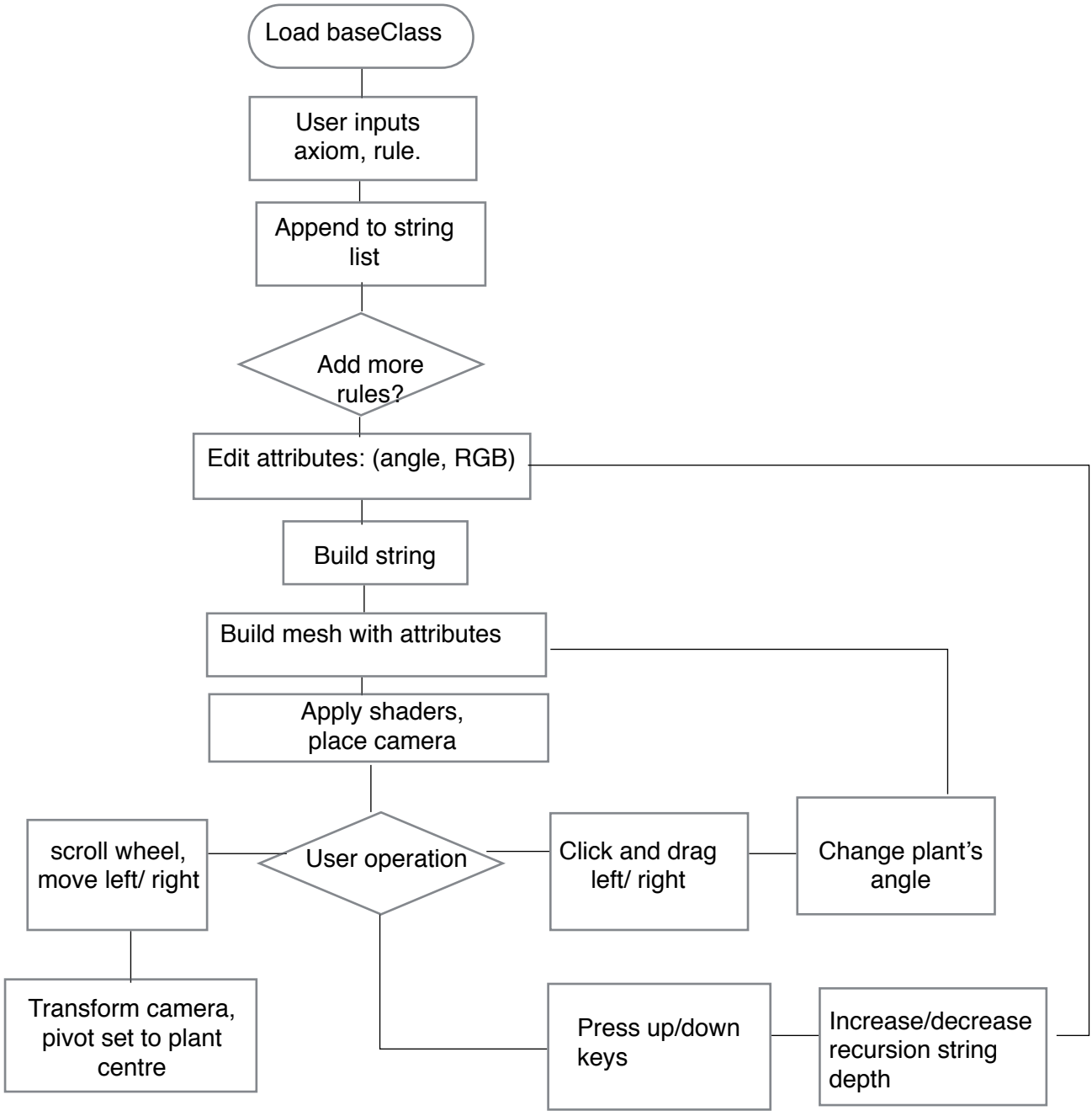
or alternatively using switch:

```
{
for (i==0; i++; i<(length of stringList))
{
switch (i) {
    case ‘A’: make a branch
}
```

The old endPosition is temporarily made into the new startPosition, increased in Y and then the startPoint is made into temp to draw a branch forwards.
In order to “branch” out, ‘[‘ stores the position on a stack, then pops it off the stack when ‘]’ is called, so another temporary vector type variable is required, “branchTemp”.

The scene is made with a camera in displayScene(), and the user can rotate the camera around the scene.

Flowchart



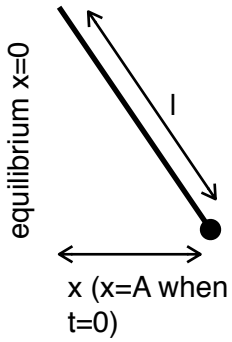
Further development: applying simple harmonic motion

"SHM: an oscillation in which the acceleration of an object is directly proportional to its displacement from the midpoint, and is directed towards the midpoint." (CGP, p44)

Any object oscillating in SHM has a restoring force, making it exchange potential and kinetic energy in the form of GPE (gravitational potential energy) or EPE (elastic potential energy) to form a sinusoidal wave. The branches would be treated as particles and contain GPE, but not EPE. The user could set a distance which to initially pull the particles, and the rate at which the particles oscillate would depend on the distance of the particles from their equilibrium positions.

The frequency and period of an oscillation are independent of amplitude, meaning a branch would swing at regular intervals even if the swing was very small. Before animating the branches swaying, i'd first find the amplitude: the maximum displacement of the branch when t=0 from its starting position, when it's affected by gravity and restoring forces aren't involved.

Given a branch and a mass at the end of the branch, to find its amplitude at a given time you first find the distance it was moved from its equilibrium when t=0.

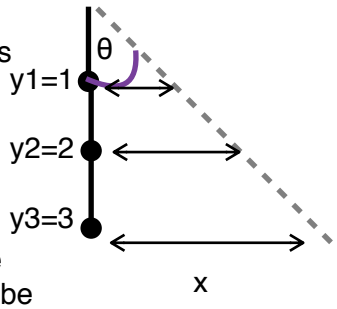


Find the period of one oscillation with T=2*pi((l/g)^1/2) where:
T = period (s)
l = length of branch (m)
g = gravity (9.81 ms^-2)

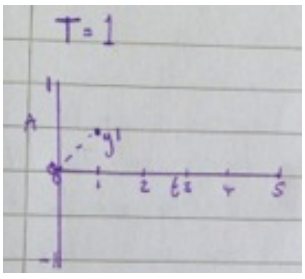
Knowing the period, I can find the frequency (F, ms^-1) of oscillations as F=1/T.

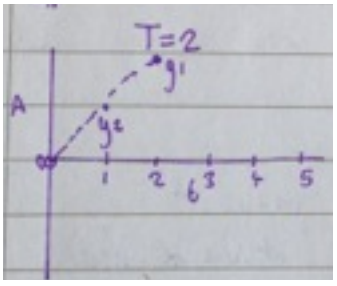
The branch end is pulled distance, x from equilibrium position. Its angular velocity, w (ms^-1), can be found with: w=2*pi*F.
When t=0, amplitude, A, is at its maximum therefore x=A as x=Acos(wt), and so I can used this to find the displacement of the branch end at time increments. When played, it should produce a sine wave.

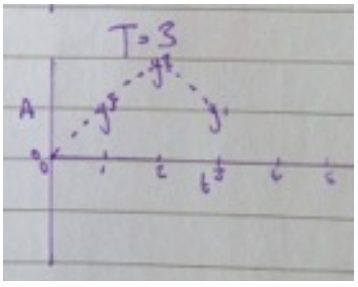
The acceleration is directly proportional to the displacement from the equilibrium position, so given particles in a line, if someone pulled the bottom particle right, this would in turn move the children right by smaller amounts. The distance these particles were moved can be found by x=y*tan(θ) (shown right).



This would make the particles animate as one object however, as each particle moves at the same time; when t=0, all of them have max amplitude. Animated individually they'd produce a wave, but together they produce the same wave. In order to make the branches look like they were swaying, i'd need to treat them like points on a graph so that the time of displacement for each particle would need to be offset by an amount.
T=animation timeline time.
When T=1, t=1 for y1 and its amplitude increases. When T=2, t=1 for y2 and its amplitude increases, etc (shown below). As a result, a sine wave can be animated from the particles.







Further considerations

Adding flowers/ leaves: To add a flower/ leaf when a certain parameter is met e.g. the branch is a certain distance from the centre.
Reacting to the seasons: The leaves could turn red in autumn or fall, using bullet's physics to simulate leaves falling. The user could use a slider to change the seasons.

References

[1] Roast, K. (n.d.). L-Systems Turtle Graphics Renderer - HTML5 Canvas - by Kevin Roast. [online] Kevs3d.co.uk. Available at: http://www.kevs3d.co.uk/dev/lsystems/ [Accessed 16 Jan. 2015].

[2] Apavlov, (2011). Cutting module in L systems [online] Available at: https://apavlov.wordpress.com/2011/06/ [Accessed 16 Jan. 2015].

Bibliography

Wikipedia (last modified 2014). L-system. [online] Available at: http://en.wikipedia.org/wiki/L-system [Accessed 16 Jan. 2015].

Boutal, A., Hilton, S., Rix, A., (2009). CGP: A2 level physics. Coordination Group Publications Ltd. (p 45-46).

Prusinkiewicz, P., Lindenmayer, A., (2004) The algorithmic beauty of plants: parametric l-systems, Springer-Verlag, New York (p47-49).

Prusinkiewicz, P., Lindenmayer, A., (2004) The algorithmic beauty of plants: Microsorium linguaeforme, Springer-Verlag, New York (p155-165).

Macey, J., (2009) Dynamic Animation and re-modelling of L-systems. [online] Available at: http://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc09/Hampshire/thesis.pdf [Accessed 16. Jan 2015]

Agni.phys.iit.edu, (n.d.). *waves and oscillations*. [online] Available at: http://agni.phys.iit.edu/~vpa/wavesosci.html [Accessed 18 Jan. 2015].