Recreate the photo evolution thingy (https://www.karlsims.com/papers/siggraph91.html)

Project Proposal

Description of theme:

- Scene outside?
- Scene with trees in pots/vases
  Floor is quad generated with shader

Technical features:

- Tree-like representation of arbitrary traits for generation of stuff
- Algorithms for sexual combination and random mutation to create variance
- UI to allow users to choose their favourite offspring per generation

  Specific Artificial Evolution Features
- Artificial evolution to generate fractal plants
- Artificial evolution to generate shaders

- Potentially demo on GitHub Pages? (since Qt can compile to Webassembly)

Github repo link: https://github.com/herewegoblueno/cs1230-final

Useful papers

https://www.computer.org/csdl/magazine/cg/1984/05/04055766/13rRUxYINan
https://dl.acm.org/doi/10.1145/15886.15892
https://www.red3d.com/cwr/iec/2011ReynoldsALifeJournal.pdf

Anderson's Ponderings
I just thought of this cool idea of maybe just an art gallery of paintings and trees

Karl sims: x y z based
Raynolds: texture based
https://github.com/chgagne/beagle

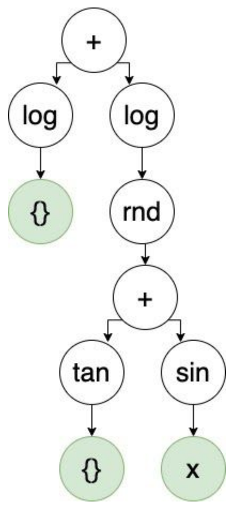
Maybe one person focuses on making the AST -> shader workflow work
Then someone else works on the basic user facing elements etc

Design Notes

Steps we gotta follow

1. Generate the AST (Abstract Syntax Tree) (more on that too)
2. Populate the shader string with any node dependencies (explained later on)
3. Then recursively translate the AST to non-recursive GLSL code (GLSL doesn't support any type of recursion)
4. Push that string into the GPU as the shader program

Here's an example of an AST



So there are two types of nodes in the AST: operations and leaves.

Leaves are the base inputs of the calculator, they can either be:

- A vector of random scalars (denoted by {})
- X
- Y
- Z
- The time variable

Operations, obviously, perform operations based on the outputs of their children leaves/nodes.

So what do the actual classes look like? I'll show their structures and explain how they work together.

Here's the Node superclass:

```
Node{
String transcribe()
Int numberOfChildrenNeeded
vector<node> children
}
```

Here's the Leaf subclass

```
Leaf{
Override int numberOfChildrenNeeded = 0
}
```

And here's the operation subclass

```
Operation {
String functionDefinitions
}
```

So we'd have subclasses of the leaf and operation nodes to make unique operations and base values.

All "values" returned by nodes (either leaves or operations) take in and return vector values. If the argument actually represents an int (like maybe a seed for a random generation operation), then the vector will be converted into an integer by the operation themselves. This does pose a bit of a limitation, but it makes implementation easier. Otherwise, we'd have to worry about polymorphic operation nodes (eg: addition of 2 vectors vs addition of vector and int vs addition of 2 ints), which is a headache. It looks like Karl Sims might have used polymorphic operation nodes though. Regardless, we can still have pretty complicated trees if we use our imagination; we could decide to make operations that only affect y values of inputted vectors, or maybe do dot products or other types of operations, so there's still a lot of versatility.

Ok so how do all these tie together? Let's explain how we'll do step 1 of the pipeline: Making an initial AST.
Here are the steps for that:

1. Pick a random operation node
2. Repeat <node.numberOfChildrenNeeded> times:
    a. Pick a random node (either a leaf or an operation node, to prevent ASTs from becoming tooo large we can make this biased towards leaves)
    b. Push that node onto this node's children vector
3. For every child in that vector, repeat step 2.

Hopefully with these steps it can become clear why we override numberOfChildrenNeeded to 0 for all leaf nodes - they're acting as the base case for this recursion.

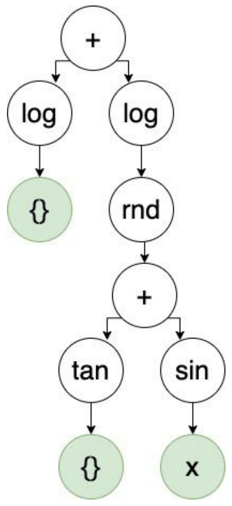
So now we have our AST. On to step 2: making that into shader code.
We're going to have to convert that to shader code.
We're going to start out with an empty shader string. We're going to build up this string recursively, then pass that string as a shader program into OpenGL (link of like how a compiler works).

Some operation nodes (imagine a perlin noise operation) would be a loooot cleaner if there was something like a noise(x,y,seed) function definition in the shader. That way they wouldn't be repeating themselves all the time when they're being translated from the AST to the GLSL.

So! If such a definition is needed, we'd type it out as GLSL code (as a string) in that operation node's functionDefinitions field.

This means the first step when making out final shader string is going though all the operation node definitions we have and adding those definitions to the final string.

Then, we're going to traverse the AST to make the main portion of the code (post order). We'll make use of the transcribe function that each node has. Let's take a look at our example again:



Assume that the transcribe for the plus operation looks like this:

```
String transcribe(){
"(" +children[0].transcribe() + " + " + children[1].transcribe + ")"
}
```

Other operators would have similar recursive transcription functions. Leaves, on the other hand, would just have transcription functions that look something like:

```
String transcribe(){
"vec(x,x,x)"
 }
```

Where x will be an `in` variable for the frag shader.
Or it could be something like

```
String transcribe(){
"vec(1,3,4)"
 }
```

At the end of this recursion, we'll have our AST turned into a string that represents a functional GLSL shader what we can push into OpenGL as a shader program.

When it comes to evolution, of course we have two options: sexual recombination and random mutation.
Random mutation just involves taking a random node (biased towards lower end nodes) and replacing it with anre random node (and recursively giving it children).

For sexual recombination, the point is doing something similar, but by making new operation nodes by combining similar operation nodes from parents (similar could be based on classifications like arity, trigonometric vs arithmetic nodes, etc).

Alana's thoughts

Looking more into L-systems and representations of tree branching structures, I think we can still simulate evolution using different factors in generating tree/branch structures (e.g. depth of recursion, angle between branches, branching patterns), however, one notable part of the evolution in the paper is mutations that "add new parameters and extend the space." It would be very difficult to do this with L-systems, as it is challenging to add new parameters in a way that doesn't make the plants resemble plants less. However, when I discuss L-system design a bit later, there is the potential to have some of this in mutating different mappings within the strings. Since this is a bit more complex, to start, this means I will probably end up using the evolution model with the same fixed parameter types. If our AST node structure is very general, this could work, and if I put parameters in the same leaf locations when building trees, flattening them and then generating corresponding L-system plants should be pretty straightforward.

In terms of L-systems, this chapter lays out generating L-systems and generating trees from them pretty well:
http://algorithmicbotany.org/papers/abop/abop-ch1.pdf

The 4 parameters for stochastic L-systems are *V*, the alphabet of letters (or words) used in the string representations of the L-systems, ω, the starting word, or string representation, *P*, a set of mappings from words in *V* to other words, and π, which consists of probabilities of different mappings from *P* occurring.Introducing stochasticity into L-systems adds more flexibility for mutations, as a stochastic L-system has a parameter for the probability distribution of the particular branching structures.

When drawing trees, we also need to take into account several additional factors that will affect actually drawing the tree, including the depth of the recursion (or number of times we will transform words in our tree using mappings from *P*), the specific angles the branches turn at, and the number of recursive branches the tree will have. This will come in to play with the class used to draw L-systems.

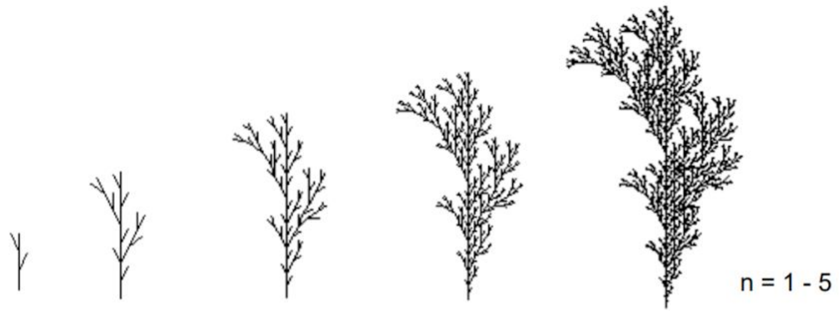
An Example

root: B

p: B → F

B → F[-B]F[+B][B]

F: move forward
+: turn left
-: turn right
[: Store the current position
]: restore the previous position



Here’s a visible example of a mapping *P* and the tree it produces

In order to represent a specific L-system, I will use a struct that contains the 4 aforementioned parameters, as well as the additional factors (depth of recursion, branch thickness decay, angles of branches, length of a branch). Once we have a particular L-system, we need a way to use that and generate a tree.

To do this, I plan to make an L-system class that generates a String given a struct with the parameters and the attributes of the tree. To make a new L-system, we will start with ω, and then use the rules in *P* to replace ω accordingly. Then, using the new string, we will repeat this process according to the rules of *P* the depth of the recursion amount of times. This will leave us with a string that we can use to build a tree.

Once we have a string generated using an L-system, we must convert it to 3D space. This guide provides a vocabulary for string instructions in 3D to draw an L-system that I plan to use: https://morphocode.com/3d-branching-structures-with-rabbit/

Making a class analogous to the Turtle in Python would be the most straightforward way to interpret these instructions. However, instead of drawing a line, each time the turtle “draws”, it should make a cylinder whose bottom face is centered at the starting position of the turtle and whose top face is centered at the ending position of the turtle. I think the most efficient way to draw a tree would be to use a flyweight pattern. Each time the turtle “draws” a branch, we could store the thickness of the branch/radius of the circle, the starting position of the branch, and the ending position of a branch. Then, once we have parsed the whole string, we use this information to generate transformation matrices for each of the branches. Then, using a cylinder, we draw one cylinder transformed that many times.

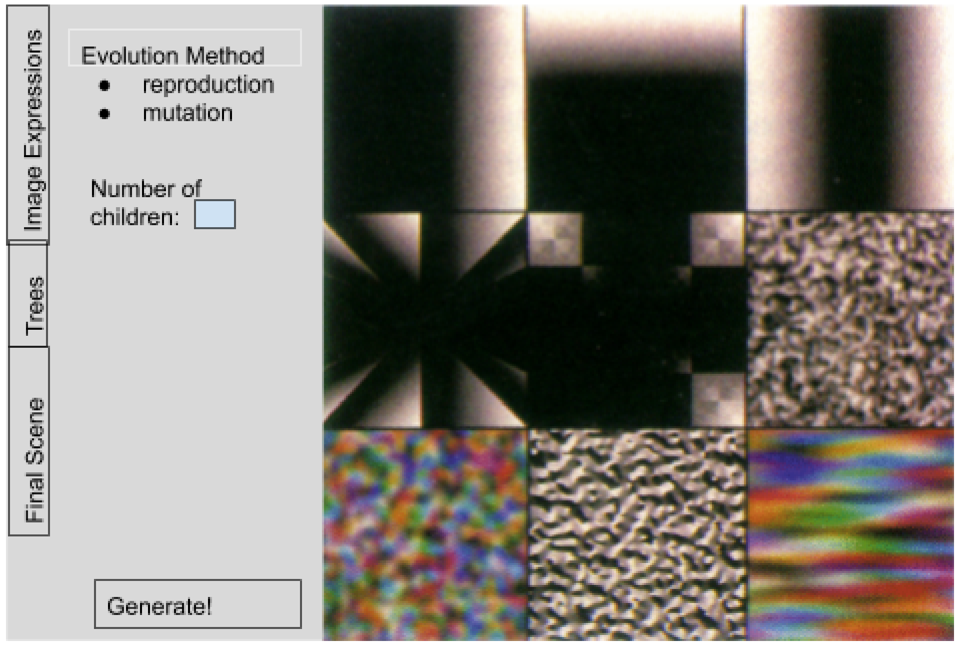
It might be a little expensive to draw that many trees during the artificial evolution process, so I could just use the x and y coordinates to draw the trees in 2D for our interface.

In terms of applying the artificial evolution process, including some parameters as leaves that will remain fixed each time seems relatively straightforward. This makes sense for factors like depth of recursion, length of branches, angle to turn by in the x, y, z, dimensions, rate the thickness of branches decreases at. However, the most interesting variation is in the modification of functions in *P*. If we make it so each starting tree has the same *V*, we could potentially represent an equation in *P* as an AST, so long as we make sure each mutation and added node follows the rules. I feel like this would be the last functionality I might implement though because ensuring mutating each expression in *P* follows the rules of the L-system might be tricky. However, breeding 2 trees still has potential for interesting crossover so long as we start with the same *V* and an ω that will map to something in *P*.

UI

For the UI, during the evolution process, it would make sense to display multiple images to the user at once and then allow them to choose from a number of actions on the sidebar. The user should also have an option to switch between image evolution and tree evolution. So, for example, in image evolution, we could start with a set of images generated using image expressions. From there, the user could choose whether they want to mutate one image or sexually combine two images. Then, the user could select the image/images they want to use and the program would generate a certain amount of those images. These could be rendered into different quads(?) on the screen.

So, to start, one might see something like this (images pulled from the Karl Sims paper)



As a stretch goal/final demo, we would like to display symbolic images next to trees in a gallery setting!

Division of Labor:

Anderson:
General AST stuff
Images/shaders with symbolic expressions
Symbolic image expressions evolution

Alana:
UI
L-system trees
L-system trees evolution