CO-EVOLVING NICHES IN VIRTUAL PLANT SPECIES
Exploring the niche forming capabilities of coevolving plants in a virtual environment.
Summary

The aim of this thesis is to examine the viability of using evolutionary algorithms to create species of virtual plants for use as models in video games. By using evolution the intent is to create a cohesive ecosystem where the plants have developed distinct strategies or niches. The genes in the simulation are the rules of a Lindenmayer-system and dictate the morphology of the plant. The fitness of a plant is calculated from the amount of sunlight it can get minus the cost of its parts. Plants can overshadow each other which means they must compete for sunlight. Many plants generated demonstrated signs of niching. Some plants appeared more artificial than others, which may cause game developers to use other methods to create plant models. Genes with the power over the shape of leaves and branches may be advantageous when creating realistic plants; this would be the next step in development.

**Keywords:** Coevolution, Virtual Plant, Lindenmayer-system
Index

1 Introduction ............................................................................................................. 1
2 Background ............................................................................................................. 2
  2.1 Evolution ............................................................................................................ 2
    2.1.1 Coevolution .................................................................................................. 2
    2.1.2 Competitive exclusion principle .................................................................. 2
  2.2 Speciation in plants .......................................................................................... 3
    2.2.1 Prezygotic barriers ..................................................................................... 3
    2.2.2 Postzygotic barriers ................................................................................... 3
  2.3 Lindenmayer-System ......................................................................................... 3
    2.3.1 OL-System .................................................................................................. 4
    2.3.2 Stochastic L-System ................................................................................... 4
    2.3.3 Context-Sensitive L-System ....................................................................... 5
  2.4 Coevolution and the Red Queen effect shape virtual plants ......................... 5
    2.4.1 The L-system ............................................................................................ 5
    2.4.2 Selection method ...................................................................................... 6
    2.4.3 Fitness ....................................................................................................... 6
    2.4.4 Environment ............................................................................................ 7
3 Problem Description ............................................................................................. 8
  3.1 Evolutionary process ......................................................................................... 8
4 Implementation .................................................................................................... 9
  4.1 Research ........................................................................................................... 9
    4.4.4 Ebners genetic operators .......................................................................... 10
  4.2 Possible complications ..................................................................................... 12
  4.3 Development process ....................................................................................... 13
    4.3.1 L-system implementation ......................................................................... 13
    4.3.2 Sunlight with OpenGL ............................................................................. 13
    4.3.3 Model generation with OpenGL ................................................................ 13
  4.4 Scientific parallels ........................................................................................... 14
  4.5 Ascertainment .................................................................................................. 15
5 Analysis ................................................................................................................ 16
  5.1 Emergence of niches ....................................................................................... 16
  5.2 Explaining rapid table-turning ...................................................................... 17
  5.3 The effects of random placement of offspring ............................................... 18
  5.4 The effects of smaller populations of co-evolution ....................................... 19
  5.5 Isolating co-evolutionary properties ............................................................... 21
  5.6 The effects of larger numbers of co-evolving species .................................... 23
  5.7 Common adaptation ....................................................................................... 25
6 Conclusions .......................................................................................................... 26
  6.1 Conclusion summary ....................................................................................... 26
  6.2 Discussion ......................................................................................................... 26
  6.3 Future work ...................................................................................................... 27
1 Introduction

The thesis presents knowledge from evolutionary biology and Lindenmayer-systems (shortened to L-system) as a foundation on which to build a computer simulation for evolving multiple species of virtual plants. Within evolutionary biology topics such as co-evolution, speciation in plants and the red queen effect are addressed. Within the topic of representation of plant growth patterns three different versions of L-systems are presented.

The creation of content for video-games can be of a substantial quantity, to minimize the amount of work one alternative to manual production is procedural generation. This thesis explores if it is possible to use evolution to generate plants that are viable as graphical resources for videogames. By using an evolutionary algorithm to create the plants they adapt in response to each other competing for sunlight, which means they may form a coherent ecology by adapting different strategies in response to each other.

The simulation is based on the simulation presented by Marc Ebner (2006) in his article “Coevolution and the Red Queen effect shape virtual Plants”. However in contrast this simulation requires that species are separated into groups where each group is sexually isolated from other groups. In each generation a fixed number of plants are created for each group. The plants use the rule-set of an L-system as the genes of the plants. This is how the genes define the position of each branch and leaf. Each new generation is selected by means of tournament selection and new plants are placed either in the shadow of their parent or randomly depending on the settings of the specific run. Data is then gathered during a specified number of generations and the resulting plants are compared and analyzed.
2 Background

In many video games plants and trees make up a large part of the environment. Since these must be created by an artist these are usually few and are reused throughout the game environment. This can be disruptive to the immersion of the player, and in response software like SpeedTree has been made to generate trees, bushes and other plants. However, though this software rapidly speeds of the creation of foliage this kind of application still requires an artist to create the models. By using a system that creates plants by means of evolution the plants generated not only have adapted to the environment that they evolved in, but also to compete with each other and form niches. A niche here being defined as: “a method of surviving used by a species that is not shared by all other species who occupies the same environment”. This gives the environment a more cohesive atmosphere and deepens immersion as the plants have formed tangible relationships. There are several different ways of studying nature as an ecological system. Peter Room, Jim Hanan, and Przemyslaw Prusinkiewicz (1996) describe in “Virtual plants: new perspectives for ecologists, pathologists and agricultural scientists” how virtual systems can be used to study how plants interact with their physical and biotic environment.

2.1 Evolution

The theory of evolution was first described by Charles Darwin (1859) in “The origin of the species”. In it he describes how organisms change over time by means of speciation and extinction. There are multiple mechanisms for evolution, one of which is artificial selection. Artificial selection is where you select individuals in population with specific attributes and breed them. For example, milk farmers chose the cows that produce the most milk in order to get calves that will inherit that trait. The mechanism for which adaptation happens in nature is natural selection. This is where individuals that survive in nature are the ones that get to breed, and those with bad mutations are more likely to die and those with positive more likely to survive. This means that positive traits are accumulated over generations and the species evolve.

2.1.1 Co-evolution

Co-evolution is described by The American Heritage Dictionary of the English Language (2009) as “The evolution of two or more interdependent species, each adapting to changes in the other. It occurs, for example, between predators and prey and between insects and the flowers that they pollinate”. Examples of coevolution are briefly described in Encyclopædia Britannica (2012). One of the examples presented is Batesian mimicry, which is when a species mimic the appearance of poisonous or otherwise dangerous species to gain protection due to being mistaken for the other species. An example of this is the scarlet king snake which mimics the poisonous coral snake.

2.1.2 Competitive exclusion principle

Encyclopædia Britannica (2012) states that “in competition between species that seek the same ecological niche, one species survives while the other expires under a given set of environmental conditions. The result is that each species occupies a distinct niche”. What this means is that when two species compete for the same resources within an environment, one of them will eventually outcompete and either displace (forcing it to adapt another niche) or drive it to extinction. The displaced species may become locally extinct, by either migration or death, or it may adapt to a sufficiently distinct niche within the environment so that it
continues to coexist noncompetitively with the displacing species. This principle is also called "Gause's principle".

2.2 Speciation in plants

In the article “Plant Speciation” Loren Rieseberg and John Willis (2007) writes “Like the formation of animal species, plant speciation is characterized by the evolution of barriers to genetic exchange between previously interbreeding populations. Pre-zygotic barriers, which impede mating or fertilization between species, typically contribute more to total reproductive isolation in plants than do postzygotic barriers, in which hybrid offspring are selected against”. In this citation the word zygotic is used which means the phase when the genetic material from each parents mix, i.e. the moment of fertilization. This means that pre-zygotic is before fertilization and post-zygotic after fertilization.

2.2.1 Pre-zygotic barriers

An example of a pre-zygotic barrier is that of pollinating insects. Quite often insects can specialize by only using the flowers of a single species, this means that cross pollination between species aren’t as common. To artificially create a pre-zygotic barrier one can divide the population in to two reproductive groups. This forces them to evolve independently, but this doesn’t guarantee that they evolve different solutions due to the possibility of parallel evolution. Parallel evolution means that one species divided into two isolated groups may evolve the same trait independent of each other by being under similar selective pressures. Diane Dodd (1989) writes that “According to the biological species concept, speciation is basically a problem of reproductive isolation”. She did an experiment with fruit flies as “an attempt to gain insight into the process of the development of reproductive isolation”. The flies were separated and fed with different diets (starch-based and melanin-based) and after eight generations the diet had changed their color. When reintroduced they preferred flies of the same color to reproduce with, which is a form of assortative mating.

2.2.2 Post-zygotic barriers

When interspecific pollination occurs there may be a post-zygotic barrier in the form of sterile hybrids or reduced fitness. Infertile hybrids could be caused by a diploid (2 sets of chromosomes) plant reproducing with a tetraploid (4 sets of chromosomes) plant. The offspring of a diploid and a tetraploid would be a triploid (3 sets of chromosomes) which are mainly infertile. The Russian plant geneticist Georgii Karpechenko (1927) made a new species by crossing a cabbage with a radish. Even though these species have different genus both have a diploid number of 18. The hybrids were mainly infertile but a few fertile plants were formed. These plants had roots of a cabbage, leaves of a radish and could breed with each other and radish, but not with cabbage. To comment this result he writes “it seems that we here approach nearer than we ever did the experimental reproduction of one of the processes in species-formation”.

2.3 Lindenmayer-System

The L-system, originally created by Hungarian biologist Aristid Lindenmayer, was originally used to describe the development of simple algae. L-systems are parallel rewriting system and are of a recursive nature. This means it generates self-similar patterns, similar to those produced with fractals. With this system plantlike forms are easy to define, and by increasing the number of recursions the model can be made to look like it grows and increases in
complexity. The system uses an alphabet, a starting string of letters from the alphabet and some substitution rules. A substitution rule defines what new string of letters a letter is substituted by. Table 1 shows the first five generations of a system with the alphabet “AB”, the start string “A” and the rules ‘A becomes AB’ and ‘B becomes A’.

### Table 1  Development example of an L-system

<table>
<thead>
<tr>
<th>#</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>AB</td>
</tr>
<tr>
<td>2</td>
<td>ABA</td>
</tr>
<tr>
<td>3</td>
<td>ABAAB</td>
</tr>
<tr>
<td>4</td>
<td>ABAABABA</td>
</tr>
<tr>
<td>5</td>
<td>ABAABABAABAAB</td>
</tr>
</tbody>
</table>

In Przemyslaw Prusinkiewicz and Lindenmayers (1991) article “The Algorithmic Beauty of Plants (The Virtual Laboratory)” they write about the properties of L-systems and how they can be applied to model the morphology of plants. In the article they show multiple versions, which are OL-systems, stochastic L-systems and context-sensitive L-systems.

#### 2.3.1 OL-system

An OL-system, the system used by Ebner (2006), is characterized by the brackets used to push and pop the current state (position and rotation) of the structure. This is useful because it makes it possible to create branches, for example the string “[+F]-F” can be used to create two branches in a V-shape. In the string the characters [ and ] push and pop the current state, + and – rotates the current grow angle and F creates a branch. Figure 1 shows a simple example of a tree structure generated from a string using this type of interpretation.

![Figure 1 From Przemyslaw Prusinkiewicz and Lindenmayers (1991) “The Algorithmic Beauty of Plants (The Virtual Laboratory)” Example of a tree made from a bracketed string representation](image)

#### 2.3.2 Stochastic L-system

Stochastic L-systems also uses brackets but differ in that they use multiple rules for the same symbol and select one of the rules randomly each time the symbol is transformed. The usefulness of this system lies in that if one uses the same system to create all plants in a single scene all plants will look identical and because of that feel artificial. By using a stochastic system by introducing an element of randomness into the plants morphology generation identical plants will be very unlikely to be generated.
2.3.3 Context-sensitive L-system
A context-sensitive L-system like the two previous examples uses brackets but differs in that it tries to emulate irregularities in the plants morphology due to the flow of nutrients or hormones. It does this by making some rules context-sensitive. This means that for the same symbol multiple rules may exist, but instead of selecting one of these variants at random like the stochastic version each of these rules are selected based on the preceding symbols in the string.

2.4 Coevolution and the Red Queen effect shape virtual Plants
In the article “Coevolution and the Red Queen effect shape virtual Plants” by Marc Ebner (2006) the Red Queen effect (also known as the Red Queen hypothesis) was tested. The Red Queen effect is an evolutionary hypothesis by Leigh Van Valen (1973). It states that continuous development is required for a species in order to maintain the relative fitness it has to the species it is co-evolving with. It is named after a character in “Through the looking glass” by Lewis Carroll (1871). In this book the Red Queen says "It takes all the running you can do, to keep in the same place". The article has the focus of seeing if the fitness value is stable over time while evolving a population of a single plant species, where each individual is competing with the others. The simulation uses a fixed population size and both asexual and sexual reproduction. Positive traits rapidly spread inside the population and no speciation occurs. For there to emerge two species in this system two plants have to develop two different advantageous traits, whose hybrid is inferior so the species doesn’t merge. These new species must then form a balance to prevent that one outcompete the other. More common is that one plant in the population evolves an improvement which is quick to spread to the entire population.

2.4.1 The L-system
To represent the plants morphology and development he used an L-system, the alphabet of which is written in table 2. In the system the letters A to Z do nothing but as the rules evolves for a species they may prove useful for defining the growth pattern of the plant. The rules may be changed using the genetic operators defined by table 3. These are the mutations that happen when offspring is produced. It is always one mutation that occurs and each mutation has an equal chance of occurring, which in this case is 10%. When using the evolved L-system it is always iterated 5 times for each individual plant.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>draw a branch segment (cylinder) and move forward</td>
</tr>
<tr>
<td>l</td>
<td>draw a leaf</td>
</tr>
<tr>
<td>[</td>
<td>push the current state (transformation matrix) onto the stack</td>
</tr>
<tr>
<td>]</td>
<td>pop state from stack</td>
</tr>
<tr>
<td>&gt;</td>
<td>+22.5° rotation around x axis</td>
</tr>
<tr>
<td>&lt;</td>
<td>-22.5° rotation around x axis</td>
</tr>
<tr>
<td>\</td>
<td>+22.5° rotation around y axis</td>
</tr>
<tr>
<td>/</td>
<td>-22.5° rotation around y axis</td>
</tr>
<tr>
<td>+</td>
<td>+22.5° rotation around z axis</td>
</tr>
<tr>
<td>-</td>
<td>-22.5° rotation around z axis</td>
</tr>
<tr>
<td>A to Z</td>
<td>no operation</td>
</tr>
</tbody>
</table>
Table 3 The genetics operators for mutation

<table>
<thead>
<tr>
<th>Genetic operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permutation</td>
<td>Two neighboring symbols are exchanged.</td>
</tr>
<tr>
<td>Mutation</td>
<td>A randomly selected symbol is replaced with a new symbol.</td>
</tr>
<tr>
<td>Insertion</td>
<td>A new symbol is inserted at a random locus.</td>
</tr>
<tr>
<td>Deletion</td>
<td>A symbol is deleted at a random locus.</td>
</tr>
<tr>
<td>One-Point-Crossover</td>
<td>Crossover is performed by selecting a rule and exchanging all rules between the two individuals which follow the selected rule.</td>
</tr>
<tr>
<td>Sub-Tree-Crossover</td>
<td>A randomly selected bracketed sub-tree is exchanged between two individuals.</td>
</tr>
<tr>
<td>Add-Branch</td>
<td>An empty branch is added to an individual.</td>
</tr>
<tr>
<td>Delete-Branch</td>
<td>A possibly non-empty bracketed sub-tree is deleted.</td>
</tr>
<tr>
<td>Add-Rule</td>
<td>A new rule is appended to the individual, i.e. if the last rule is $C \rightarrow \chi$ then $D \rightarrow D$ is added.</td>
</tr>
<tr>
<td>Delete-Rule</td>
<td>The last rule of an individual is deleted.</td>
</tr>
</tbody>
</table>

### 2.4.2 Selection method

To select the parents for the next generation tournament selection is used. The first step in the tournament selection method is to pick out $K$ individuals at random from the population. Then one of these individuals is selected for reproduction where their chance to be selected is proportional to their relative fitness. This means that if $K$ is 1 the selection would be entirely random while if $K$ is the same size as the population it is very unlikely that individuals with low fitness being selected. By changing the size of $K$ the randomness of the simulation can be regulated. After an individual has been selected and the genetic operator has been randomly selected and implemented the new plants location is selected based on the footprint of the parent. The footprint is the shadow the plant forms when rendered from the sun’s perspective. This is done in a similar way to when calculating the sunlight points for the plants, but without rendering the other plants. This is done until the population limit has been filled.

### 2.4.3 Fitness

The method for calculating fitness is made up by two parts. A plant is awarded points by collecting sunlight, and points are deducted due to the complexity of the plant. The complexity is meant to approximate the costs of creating or growing the plant’s branches and leaves.

\[
\text{Fitness} = \text{Sunlight} - \text{Complexity}
\]

The sunlight is calculated by assigning a unique color to each plant and render using an orthographic view from above to a texture 512 by 512 pixels. The texture shows the leaves which are closest to the sun and gathering light, which means that plants overshadow each other (see figure 2). Then by counting the number of occurrences of the different colors in the texture the score for each plant can be calculated. The score is then scaled by a value to increase the value of each occurrence in the texture.

\[
\text{Sunlight} = \text{Score} \cdot \text{Gain}
\]
The method for calculating structural complexity is by counting each branch and each leaf and scaling the cost by a factor based on how high it is of the ground. Because of the height factor of the complexity algorithm and the fact that there is a limited amount of sunlight a maximum height for the plants is created. In the algorithm below b is a branch contained in the set of the plants branches B and l is a leaf contained in the set of the plants leaves L. Factor regulates the rate at which cost increases per height unit and is set to 1.1. Height is a function that extracts a component’s height of the ground. Cost\textsubscript{branch} is the cost of a branch and Cost\textsubscript{leaf} is the cost of a leaf.

\[
\text{Complexity} = \sum_{b \in B} \text{Cost}_{\text{branch}} \cdot \text{Factor}^{\text{Height}(b)} + \sum_{l \in L} \text{Cost}_{\text{leaf}} \cdot \text{Factor}^{\text{Height}(l)}
\]

![Figure 2](image.png) Example of plants overshadowing each other

### 2.4.4 Environment

The plants fitness is evaluated both individually and all together competing for sunlight. When Ebner (2006) tests them individually they are all positioned in the center of the environment. When all plants are tested at the same time the plants are distributed in a circle near the center, but as the generations go by the plants spread out and occupy the entire field. The position of a plant's offspring is always in the shadow of the parent. He uses three different environments of which one is flat, one is sloped and one that is flat but has two levels (see figure 3).

![Figure 3](image.png) Ebners (2006) different environments
3 Problem description

There are several methods of creating plants for use as environmental graphics in video games. These give the artist the job of imagining how each plant interacts with the environment and how they collectively form a stable ecology, which can be quite difficult if the number of species is great. If the plants in the environment are made in such a way that each plant form a relationship with another, be it competitive or collaborative, then the believability should be higher due to a feeling of coherence. The aim of this thesis is to examine if multiple species of virtual plant can evolve different strategies or niches in the same population. By using evolution as the method of creating plants it is easy to create a coherent ecology of plants without any imaginative process. These evolved plants have adapted to the environment and the competition therein so they form a natural relationship with it. As in nature different species have different niches and strategies, so can they also evolve in a computer simulation.

3.1 Evaluation process

The purpose is to explore the viability of evolutionary algorithms for producing a cohesive ecosystem of plant models for video games. To find the answer to this two aspects have to be answered. First the plants have to develop niches that are apparent in order for them to feel cohesive. Secondly the plants generated have to be visually appealing as plants that are apparently artificial could destroy the aesthetic of the game. To gather necessary data a computer simulation is used for evolving the different species, whilst they compete against each other in the same environment.

Instead of designing the experiment from scratch Ebners (2006) implementation will be replicated (described in part 2). The implementation he made is simple and is used to test how the stability of the fitness of a single plant species is affected as it evolves by competing with itself. In this thesis focus lies on seeing if multiple species of plants can develop different strategies in response to each other, and therefore multiple species must be used.

To force a population to form multiple species a barrier can be used to divide the population into groups. This would cause reproductive isolation and prevent traits evolved by one group to spread to the other, which forces each group to evolve independently but still in response to each other. The morphology and growth patterns of plants use an L-system where the OL-system rule set are the genes of the plants. As the rules change the plants growth pattern is altered, and as a result the morphology too. An arms race will ensue forcing each species to develop strategies in response to the other; however there is no guarantee that these strategies will be different or dissimilar.

By comparing the shape, height and shadow cast by the plants to plants of the other species evidence of niching may be discovered. One possible niche that the plants may adapt is one of height, where the plant stretches straight up in a single or a few branches. This has the advantage of being cost effective as they can very cheaply stretch higher than competing plants. Another possible niche is one of reproduction. As the plants shadow becomes bigger when the leaves are spread the area of the plants reproduction is bigger allowing it to spread around faster decreasing the intra-species competition.
4 Implementation

In this simulation the plants need to adapt to the environment and to their competition. This they do by changing their morphology, which means the way their branches and leaves are positioned. In order for the plants to do so they must evolve, and to do that they need genes to represent their morphology that can be passed on and mutated. An L-system is an ideal choice for this as they were originally designed to represent plant development. An L-system is made up of three parts: An alphabet, a start string and a set of rules that describe how each letter in the alphabet is substituted. In this simulation the genes define the set of rules, and as the genes are individual to each plant the rules are different for each plant as well. An example of plants modeled from L-systems can be seen in appendix A. Not only genes are needed to evolve the plants, a selection method is also required. The chance each plant has of being selected is in proportion to its fitness in relation to the total amount of fitness the species has. Fitness is determined by an algorithm designed to promote sought after traits such as sun exposure and discourage unwanted ones such as useless branches and leaves. Then by examining the morphology of the different populations after the specified number of generations has been simulated evidence of niching should be visible. To compare the plants their shape and shadow can be studied and from that conclusions can be drawn.

4.1 Research

By replicating the system used in another article one can simplify the process of designing this simulation. The article by Ebner (2006), while having its focus to the Red Queen effect, uses a very simple system that is described in great detail (see chapter 2.4). The article establishes how a single plant population develops, which can be used to compared to how multiple species are affected morphologically and how the stability of their fitness change. By replicating the implementation that he used I hope of being able to compare the results found here to the results of his experiment.

In the simulation different runs will use different parameters in order to compare them with each other and draw conclusions from their differences; these parameters are presented in table 4. The first run is the one most like Ebner’s (2006) runs, with the exception that the population is divided into two different species. The similarities are not a coincidence since this allows the results found by Ebner (2006) to be compared to the results found in this run. The second run is the same as the first with the exception that the placement of offspring is random instead of being placed in the shadow of the parent, and because this is the only difference the effect of random placement should be detectable by comparing this run to the first. The third run has half the total population of the first run; this means that the effect of population size and available space should be detectable by comparing the first and the third run. The fourth run simulates only one species, this is done to see what behavior is due to co-evolution and what is caused by the Red Queen effect. The fifth run uses three species; this is done in order to determine if some of the behavior that has been attributed to co-evolution becomes more or less pronounced when the number of species increases.
Table 4 Simulation parameters

<table>
<thead>
<tr>
<th>Experiment number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of generations</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Population size per species</td>
<td>100</td>
<td>100</td>
<td>50</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Number of species</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Method of selection</td>
<td>Tournament</td>
<td>Tournament</td>
<td>Tournament</td>
<td>Tournament</td>
<td>Tournament</td>
</tr>
<tr>
<td>factor</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>cost\textsubscript{branch}</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>cost\textsubscript{Leaf}</td>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>gain</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>child placement</td>
<td>Shadow</td>
<td>Random</td>
<td>Shadow</td>
<td>Shadow</td>
<td>Shadow</td>
</tr>
</tbody>
</table>

The size of the environment will be 100 by 100 units, this number has been chosen by trying to approximate the variables used by Ebner (2006). It was chosen by testing different parameters and comparing the result with the pictures in Ebners (2006) paper. The respective size of branches and leaves are shown in figure 4. Ebner (2006) used multiple environments (flat, sloped, etc.), however only the simple flat one will be used in this implementation. The initial distribution of plants will be in a circle alternating between the species. After 500 generations the original distribution will have had very little influence over the final result.

![Figure 4 Measurements of the different components](image)

4.1.1 Ebners genetic operators

The genetic operators presented in table 3 from Ebners (2006) article will be used as they are to minimize the gap between his test data and the data presented in this thesis. A minimal set of operators is generally preferred; all operators should add some functionality that the other operators don’t. Because of this the relevance of all operators included should be made clear. They will, however, still all be used regardless in order to stay as similar to Ebners (2006) paper as possible.
The first of the operators is permutation, what this operator does is change the place of two neighboring genes effectively migrating individual genes up and down the genome. The effect this has is minimal since it does neither add nor remove any genes. It could therefore be excluded from the simulation without limiting the adaptability of the plants.

Mutation, insertion and deletion are the next operators. Insertion adds one gene to the genome and deletion removes a single gene. What mutation does is that it replaces a gene, which may be described as a deletion followed by an insertion at the same location in the genome. This means that mutation is superfluous and is simply a faster way to potentially remove a bad gene and add a good one.

One-point and sub-tree crossovers are the two and only means of sexual reproduction. This means that this is the only way for different good genes produced in different plants to be able to become part of the same merge into a superior hybrid. This means that while it is not needed for the plants to adapt and evolve there are scenarios where it could be advantageous.

Add-branch and delete-branch regulates the number of potential branches that a genome contains. Add-branch adds a set of empty brackets while delete-branch removes a set of branches including the genes between the left-bracket and the right-bracket. Add-branch is vital in order for plants to develop branches, without it plants would only be able to grow in a vine-like manner. The motivation for keeping the delete-branch operator needs some more in depth explaining, as its effects are not self-evident.

Since empty brackets don’t affect the shape of a plant it may appear as though there is little need for the delete-branches operator. But it is very important as it regulates the amount of rotation genes added that have an effect and the chance of adding genes to new branches or adding genes on a branch that already has genetic information. For example let’s assume we have two genomes, one with a few empty brackets and another with many. If both plants were to be subject to an insertion and the type of gene inserted was a rotation then the chance of this insertion going into an empty bracket is greater with the genome that has more of them. A rotation gene is the only gene inside a bracket it will have no effect as the popping of the matrix removes its effect. If the insertion is of another type it will have a direct effect on the shape of the plant, however it will become less and less likely that genes will be inserted into a non-empty bracket. This means that there will be more and shorter branches without there being a selective pressure for it. This results in that when an insertion occurs then it has a greater effect on plants with fewer empty brackets, which in turn means that there will be less advantageous insertions and therefore the resulting rate of adaptation is lower. This means that if there weren’t an operator for removing empty brackets the rate of adaptation would get slower and slower.

The Add-rule and delete-rules add and subtract the number of extra letters and associated rules a genome uses. More rules mean that a genome can have a more complex growth pattern. Complex growth patterns may not be required but adds more possible forms a plant may take. Since insertions are made in a randomly selected rule a gnome with a large number of rules would have the insertions spread among more rules, and as a result a smaller number of letters in each substitution string.
4.2 Possible complications

To accurately analyze how the plants have developed niches one has to examine them from the point of view of their fitness since that is what drives their evolution. This is done by both comparing their fitness value and examining how they have developed morphologically compared to each other. One possible outcome is that because the environmental conditions are the same for all species they may adapt the same, or similar, strategies. This would mean that the species haven’t developed niches as they are all doing the same thing. The amount of information gained in this case is limited, but one can speculate how the parameters and environment have affected the development. Another concern is that of how much the plants can adapt. Ebners (2006) method doesn’t have genes for the shape and size of leaves and the manipulation of branch thickness. It also doesn’t use any flowers or any other real world form of reproduction. These things limit the number of possible strategies a plant can adapt. Prusinkiewicz and Lindenmayers (1991) describe ways of modeling representing these things, but these representations are not used in this experiment.

The simulation tries to produce as realistic plants as possible by approximating reality, but there are many things that the simulation doesn’t take into account. One of these is that sunlight only comes directly from above. In the real world the sun moves across the sky and plants have evolved in response to this. Because this isn’t represented in the simulation it means that there is no evolutionary pressure for developing leaves on the sides of plants. Another is that there isn’t any form of collision between the plants, which means that different branches can intersect. Because of this lack of realism the plants in the simulation may have less of an impact on one another.

The simulation is divided into discrete steps where each step contains a generation of plants. All plants are created fully formed and continuous growth isn’t simulated. This means that earlier stages of a plants growth isn’t represented in the simulation, which means that plants can’t develop strategies for overshadowing their competition during earlier stages of their development. Because of the plants aren’t growing continuously it means that the plants can’t react to the environment. Real plants will grow towards the light in a reactive way; in this simulation the exact form of a plant is predetermined.

And because each branch segment is a cylinder a gap can opens up between branch segments if they aren’t parallel to each other, as shown in figure 5. This means that there may be some pixels where leaves would have been overshadowed but aren’t. This is very specific set of circumstances which rarely occur so this problem can be safely ignored.

![Figure 5 Branch joint](image-url)
4.3 Development process

During the process of implementing the simulation numerous design choices has been made, some of which were made to improve runtime speed and some just because alternative solutions would require too much development time. There are three parts required to successfully implement this simulation. They are the L-system, the evolution algorithm and the model generation, if one of these parts were missing the simulation wouldn’t work. Here follows an account of some of the difficulties that arose and the solutions used so solve them.

4.3.1 L-system implementation

Implementing an L-system is a very simple task in any programming language, but is in this case complicated due to the rules needing mutating capabilities. The L-system is implemented as a class and for a mutation to function it needs access to the data stored inside that class. Generally speaking there are a few ways to do this, for example giving the L-system an interface that provides everything that the mutations need to operate. The method selected was to implement the mutations as member functions of the L-system, which limits the class’ reusability. There were early attempts at achieving reusability through templates, but they were unsuccessful and were quickly abandoned.

4.3.2 Sunlight with OpenGL

When rendering the sunlight texture a framebuffer object (FBO) is used, for which a class was created. The class had methods for making it the active render target, reverting back to the original and retrieving the texture data. Before the FBO method was implemented a different method was used where the models were rendered to the default framebuffer, after which the relevant pixels were copied to a texture and then loaded into primary memory. This method was abandoned since the FBO method appeared faster. Figure 6 shows what one of these textures can look like.

![Figure 6 Sunlight texture used in calculating sunlight exposure](image)

4.3.3 Model generation with OpenGL

The models for the plant used in the simulation are composed of simple six-sided cylinders for the branch segments and hexagons for the leaves. They were implemented as display lists that later were called from the plants display lists. Attempts were made to make the plants look more visually appealing and realistic. To do this a substitute for the cylinders was required. When the cylinders were used at angles from each other they could have gaps...
between them. In order to fix this problem each segment has to be connected to the previous in a seamless manner. Attempts were made to try and fix this, but due to time constraints they were abandoned early in development.

The first method used to generate the plant models was a display list created using a switch case set. In the switch case set there was a case for each symbol (excluding the letter A-Z) with the appropriate computer instructions. The switch case set was fed each letter of the resulting string of the L-system for the plant, and all instructions were compiled into the display list. During the generation the height of each branch or leaf is extracted from the model-transformation matrix and is then used to calculate the complexity value for the plant. At first this method was planned to be scrapped later in development because there was no apparent method of calculating the height of each branch segment and leaf. After such a method was discovered a change would have been superfluous.

The complexity calculation was one of the things that proved difficult due to the L-systems having a tendency to explode in height. The explosion leads to the float used for summing up the plants complexity to overflow, giving the plant negative complexity. This lead to very high fitness in these individuals and they quickly locked up the system by being made up by a huge amount of triangles, sometimes overflowing the std::string character capacity. The quick fix to this problem was to change the data type to a double. The reason for the plants to explode in complexity was because the model matrix wasn’t reset before generating the plants. This made the fitness cost scale along the wrong axis, leading to plants costing nothing at certain position in the environment. After this bug was caught float could be used again.

4.4 Scientific parallels

To strengthen the simulation scientifically it is made as a slightly modified version of Ebner’s (2006) simulation, but there are others that have similar simulations and it is interesting to see how their method relates to this method. Prusinkiewicz and Lindenmayers (1991) describe several alternative methods of modeling a plant from various types or l-systems and alphabets, many of which model plants in much more detail. For example the plants in this simulation doesn’t have flowers, the sunlight is always shining from above and is using neither stochastic nor context sensitive l-systems. Because of this vast variety of ways to implement these kinds of simulations and no common framework to work with the scientific comparisons can only be limited to papers using the same method, which in this case is the paper by Ebner (2006).

Since the simulation doesn’t allow a species to die out Gause’s law isn’t in effect, which means that the simulation doesn’t accurately mirror the real-world. The results of this simulation are therefore meaningless in a real-world context. However, since the purpose of this paper isn’t to make real plants but plants that look real, the trouble this absence creates is negligible. However, there may form multiple groups within a species that are so genetically different that they cannot produce viable offspring which could be called different species. These groups are short lived as the population size isn’t large enough, this leads to one species driving the other to extinction very quickly. Because the simulation is using fixed population sizes, if one species prevents the other from gaining any sunlight there will be no evolutionary pressure selecting for or against any genes for that species. This means that total chaos ensues and that any shape or form is viable. This scenario could be interpreted as a form of evolutionary brainstorming.
4.5 Ascertainability

When examining the results to confirm or deny the emergence of niches within the simulation the results have to be compared to the definition of a niche given in the background; “a method of surviving used by a species that is not shared by all other species who occupies the same environment”. There is nothing to suggest that there is anything preventing niches with the algorithms used, though it may not happen every time. There is one problem which is that these kinds of simulations can be implemented in a variety of ways, this being one of the simpler ones. The multitude of implementation choices makes it very difficult to draw generalized conclusions on the benefits of niched plants, as it may vary greatly between different implementations.
5 Analysis

5.1 Emergence of niches

When the simulation starts there is a period of time before randomly coming across a form capable of collecting sunlight, all plants have 0 as their fitness value and therefore cannot evolve. This is because no genes are selected for or against as all plants have an equal value in fitness. In figure 7 we see that the blue species is tall like trees while the other plants are much shorter. This is due to the fact that the blue species was the first to develop into ‘gliders’. A glider is a plant that has a single leaf tilted to gather sunlight, and since the children are born in their parents shadow it gives off the appearance that the plant is gliding to one side as the generations’ progress. This is the simplest form a plant can take in this simulation, and because of blue found this form quickly it had a head start for many generations. This head start forced the other species to live in the blue species shadow, preferring to gather the limited sunlight it could below the canopies. Being short also made the complexity cost of the plant very small, making the sunlight gathered have a greater impact on its overall fitness.

Figure 7 Blue trees and cyan and green shrubs.

It is also plain to see how they group together, which is caused by being placed in the shadow of their parent(s). This is very common in the runs where children are placed in the shadow of their parents. Another possibility is that the plants are distributed as they are because of a preference to a specific side of the environment. As orientation is always the same the plants morphology may have adapted to stave off competing plants of different species from a specific direction. However there is little to suggest that this is actually happening. Even though individuals within a species are very likely to group together they are often moving around as a group around the environment. This suggests that no such preference exists, though it may be in fact because the species migrate around the environment that prevents it from developing such angle dependent preferences in the first place.
5.2 Explaining rapid table-turning

![Fitness Graph](image)

**Figure 8** Average fitness with placement in parents shadow, 2 species and 100 in population size.

In figure 8 between generation 65 and 100 there is a major shift in fitness where the two species very rapidly have a change in overall fitness. A simple explanation to this is that at a generation around 65 species 1 develops a superior plant that is vastly superior to the other plants in the population. This plants genome is in the following generations spread throughout the population. As the superior genes become more and more prevalent the average fitness rapidly changes. If this is true then the best plant of each generation should be very similar throughout the duration of the rapid increase. It should also show that species 2 evolves slowly and show little change as any major adaptation would slow the rise or descent.

**Figure 9** Best individual plant from generation 65, 66, 77 and 100 (shown left to right) of species 1 with placement in parents shadow, 2 species and 100 in population size.

Figure 9 shows that at generation 65 the best individual is a vortex-like plant that spiral upward. Smaller versions of this has been around before but they didn’t have as many spirals and wasn’t as tall. Over the next generations the best individual alternates between this version and the smaller version. This can be explained by siblings produced by this plant competing with each other brings down the fitness of that group, while simpler plants have less complexity and therefore more stable fitness. Even though the taller version isn’t recorded as the best each generation it doesn’t mean that the form isn’t most fit overall. The group of plants may represent a higher share of the average fitness. At generation 77 a much taller and straighter version is the best individual, and this form is the most prevalent among the best plants in the generations up to 100.
In comparison to figure 9 the plants in figure 10 shows smaller change in morphology. The shape from generation generations 65 is most prevalent between generations 65 and 72. The shape shown in generation 78 is most prevalent in generations 78 to 84. In the generations between 87 and 100 the most prevalent shape is the one shown in generation 100. These most prevalent forms should show the overall trend and the evolution within the population. What is seen is that species 2 overall grows a little taller and not much else.

5.3 The effects of random placement on offspring

In figure 8 we see a very clear correlation between growth in one species and reduction in the other, which is especially clear early on as the two species switch places. As the simulation progresses the plants get more complex in order to remain competitive and the overall fitness decreases. Comparing this to figure 11 where plants are placed randomly the effect that species have on each other in regards to fitness is much less prominent, and is only slightly visible between generations 60 and 300. This could be because of how placing offspring in the shadow of the parent tend to make species form a big cluster. These clusters then tend to grow, shrink and migrate. The interaction between the average fitness of species could be affected by this through one cluster growing bigger resulting in the other growing smaller. When offspring is placed randomly this wouldn't happen and responses in average fitness would be less pronounced. Towards the end of this run the two species also reduce both in overall fitness and difference in fitness, this is also seen in figure 8.
Figure 11. Average fitness with random placement, 2 species and 100 population size.

5.4 The effects of smaller populations and more space

In this run of the simulation half as many plants are used as in the first run, this means that the amount of space available per plant increases. This also means that the overall fitness is higher since the amount of sunlight per plant also increases, and that populations are smaller which may have an effect on how the plants evolve. Figure 10 shows less signs of a direct relationship between the gains in one species concurrently with the reduction of the other species fitness.

Figure 12. Average fitness with placement in parents shadow, 2 species and 50 in population size.

Comparing figure 12 to figure 8 it is clear that figure 12 is a lot more chaotic, and a rise in one species resulting in the decline of another isn’t as pronounced. A cause of this could be that since plants have more space they tend to grow wider, this causes the shadow of parents to be bigger. A bigger shadow means that the next generation is placed more chaotically, and would increase the occurrences of when multiple plants are placed in positions near each other. Smaller footprints would, inversely, have less overlap with the shadow of other plants and are therefore less likely to place their offspring near the offspring of another plant.
In figure 13 we can see that species 1 (on the left) is very complex while species 2 (on the right) is relatively simple. Species 2 is forced to stay simple because species 1 is blocking sunlight from reaching species 2, as branches and leaves gives negative fitness simpler plants are able to retain more of the fitness gained by collecting sunlight. The advantage of species 1 is clearly visible in figure 12 as species 1 has an average fitness of 16000 while species 2 has an average fitness of about 1500. This early advantage is however gradually lost as the generations go by, and around generation 230 species 2 is overtaken by species 1. In figure 14 you can see very clearly that species 2 has adapted a very simple and tall shape. This is a direct consequence of the morphology of species 1. One species overtaking another like this is happened in the first run as well and shouldn’t be attributed to the increase in area per plant.

![Figure 13 Best plants of the 51st generation of either species](image)

*(population size: 50 placement in parents shadow)*

![Figure 14 Best plants of the 230th generation of either species](image)

*(population size: 50 placement in parents shadow)*

In figure 12 generation 230 is point where species 1 is overtaken by species 2 in average fitness. When comparing the best plants of the generations 230 and 500 (seen in figures 14 and 15) there is a clear resemblance. As species 2 was overtaken it had to simplify and grow taller in order to stay competitive, but it still retains its bush-like esthetic. Species 1 on the other hand gaining the upper hand and adapted more branches with more leaves in order to maximize the sunlight gathered.
By comparing the species seen in figure 15 and figure 16 we can see that that the plants have grown wider, this may be due to the halved total population resulting in more space available per plant. The plants in figure 15 also appear more organic and chaotic while the plants in figure 16 seem more artificial. But there is nothing to suggest that this is due to the difference in population size. This is just one of the things that change each time the program is run, but may be of some concern when trying to create procedural plants in an unsupervised manner.

5.5 Isolating co-evolutionary properties

By only using a single species in the simulation the results should help to figure out which properties are results of co-evolution and those that are not. Similar to Ebners (2006) paper the average fitness shown in figure 17 shoots up and dives down again at around generation 100. Subsequent generations are then hovering around 1000 in average fitness, with occasional spikes bringing it up to around 3000-4000. It is possible that when a species adapts in response to itself complexity rises, and adapting in response to other species limits this rise in complexity. It could also just be caused by the multiple genes that can be selected for and the ones that increase complexity happen to be the ones that develop first. A larger number of tests are needed to accurately determine if this faster dip is caused by the use of only one single species or simply by chance.
Figure 17 Average fitness with placement in parents shadow, 1 species and 100 in population size.

Figure 18 is a graph Ebner (2006) presented in his paper showing his results on a flat environment. It shows both the maximum and average fitness of the species evolved in the flat environment. In this graph only the average fitness is of interest as this is what has been measured in the simulation used here. The graphs average fitness peaks early and is very low after generation 100, which is similar to the pattern shown in figure 17.

Figure 18 From Ebners (2006) article showing the result with placement in parents shadow, 1 species and 200 in population size.

Ebner looks at the generations where fitness is at most stable and compare it to how the morphology and complexity of the species changes over time. The Red Queen effect (Van Valen, 1973) says that a species may continue to adapt yet remain relatively stable in overall fitness. This pattern is not limited to adaptations in response to the same species but is predominately attributed to predator and prey interactions. Ebner (2006) writes that “It may also be of interest if an environment can be found which leads to speciation of the plant, i.e. to have one set of plants located in one area and a different set of plants located in another area.” Allowing speciation may cause the other species to continue evolving by forcing evolutionary responses in the other. If a species evolves adaptations resulting in selective pressure in another and if that species doesn’t adapt in return it would eventually be overtaken. However, co-evolution sometimes runs into a barrier of physical limitations. A rabbit cannot evolve faster and faster indefinitely, therefore in this case the selective pressure generated by the predator can’t be adapted in response to indefinitely.
5.6 The effects of larger numbers of co-evolving species

Figure 19 shows a simulation run done with 3 species, here the average fitness shown is a lot more chaotic compared to figure 8. Because of this any rise or fall in average fitness is difficult to associate to a specific species or even determine if it was caused by co-evolution. One of the many possible explanations for this chaos is that some of the noise in the graphs may have been caused by mutated plants. These plants would have mutated into being so large that they block out a large amount of light, but are so complex that their resulting fitness is zero. Another possibility is that when very successful plants produce many children they have to compete for sunlight with a large amount of siblings, which brings down the average fitness of those individuals.

Figure 19 Average fitness with placement in parents shadow, 3 species and 50 in population size.

It is clear in figure 19 that species 3 is heavily suppressed in the generations after 150, but appears to make a comeback after generation 450. This comeback may be caused by species 1 and 2 evolving in direct response to each other with little to no selective pressure in response to adaptations made by species 3; this would allow species 3 to sneak back in. It can therefore be reasoned that a greater number of species allows for species to adapt with less counter-adaptation if they are not the biggest rival at the time.

The reason for its suppression is simple, the other species in the environment are better at collecting sunlight. Species 3 would have been outcompeted and driven to extinction if it had been able to. Because species in the simulation are of a fixed size and can’t die out species 3 enters a state of where reduced complexity is most profitable. Low average fitness can therefore either be caused by too complex a plant or failure to collect sunlight. Reducing complexity would be beneficial in both cases as collected sunlight accounts for more with less cost associated with maintaining it. This reduction in complexity makes it very hard for species 3 to make a comeback. It needs to stretch as high or higher as the other species in order to collect sunlight that will pay for the cost of the parts needed to build that high. Since the sunlight needed to pay for the stages in between is blocked by the other species this becomes increasingly unlikely.
Figure 20 Best plants of the 500th generation of each species
(population size: 50 placement in parents shadow)

In figure 20 we can see the very different shapes that each species have employed. Species 1 (left) has adapted a very simple straight and tall morphology, species 2 (middle) has adapted a branching tree-like morphology and species 3 has adapted a more bush-like shape. Whether this biodiversity is caused by the plants developing niches is hard to tell as chance dictates which adaptations are available to a plant. It would therefore be inadvisable to suggest that each plant having a distinct morphology is directly caused by co-evolution. There is no guarantee that all three species will remain distinct in all runs of the simulation. Since the numbers of optimal forms are limited more species would at some point guarantee that there are species with similar adaptations.

In figure 19 at generation 390 species 2 has a large spike. Since only the best individual is recorded it may be impossible to find the cause of this spike. One explanation would be that an individual that is superior in regards to the current population emerges; this plants superior gene is then spread within the species. But as this gene spreads there are more plants with this adaptation which results in more competition. The increase in complexity that was well worth the cost when there were less plants with the adaptation is now costing more than it provides. The rise is therefore due to the spread of this gene when there were fewer plants with it, and the fall is due to there being too many plants with the adaptation. To see if this is actually the case data showing how the plants are overlapping during the spike is required, since this data isn’t available the question remains unanswered.
5.7 Common adaptations

The difference between a good trait and a bad one depends heavily on the environment that the trait is developd in. A bigger plant may be able to collect more sunlight, but it also costs more to maintain. If the amount of nourishment is insufficient to sustain the size of a plant being smaller is beneficial. This means that what separates a good trait from a bad one varies depending on the current state of the environment that it is evaluated against. Some shapes are however more common than others. In figure 20 three different plants are shown, these three shapes are commonly produced by the simulation. The shapes have some similarities to real-world plants. All these shapes try to maximize the amount of gathered sunlight, and employ different strategies for doing so that each has extra effects.

5.7.1 Nettle

Nettles grow very straight with leaves growing out along the stem, some plants demonstrated this structure. These plants have a disadvantage when it comes to spreading out because of their small footprint. A small footprint means that offspring will more likely be placed closer to the parent. A smaller footprint also means that they can survive with very little space; this means that there can be more plants inhabiting the same area. Therefore this shape is very advantageous when the population is large in relation to the space it inhabits.

5.7.2 Tree

Some plants grew long stems that had long branches that grew out of them; this has the advantage of creating many leaves high off the ground. The tree also has a larger footprint and a higher cost of maintenance than the nettle. The higher cost comes from the branches need to spread out. In order to benefit from the branches trees usually need more space in between themselves. If two trees are places next to each other they would be overshadowing each other which would result in much less sunlight exposure. Less sunlight exposure means that the cost of the branches will consume more fitness than it generates.

5.7.3 Bush

In the simulation there are spiraling plants that form a very wide and hollow structure, these shapes resemble bushes. These bushes produce more leaves lower down in the structure in comparison to trees; which is less cost effective. In reality these leaves would gather sunlight coming in from lower angles but can here only hope to gather the small amount of sunlight that manages to slip by the canopies.
6 Conclusions

6.1 Conclusion summary

The simulation used is a reproduction of the simulation used by Ebner (2006) but with the added capability of using multiple species. The reason for basing the simulation on Ebners (2006) simulation being that the very limited capabilities of the genes makes it easier to implement. This being a simplified method of representing plants the results found may not be applicable to other simulations, other simulations with more powerful genes may get very different results. The simulation once finished was run 5 times, each time using different settings, in order to detect if there were different trends based on the number of species, number of plants in each species and placement selection of offspring.

During the different runs the simulation produced plant species that are morphologically distinct from other species evolved in the same environment. Being distinct they fulfill the requirement posed on them for being classed as niches. The produced plants do however vary in their distinctness, visual appeal and realism. This means that it may not be viable to generate plants on the fly on a client computer using evolutionary algorithms. Alternatively they can instead be evolved beforehand and delivered with the end product. The dependence on a supervisor devalues this type of approach and could cause game developers to adopt other more predictable and/or controllable algorithms.

The purpose of this thesis was to see if it was viable to use evolutionary algorithms to produce procedural plants for video games. Due to the unpredictability and lack of control one has over the outcome the conclusion has to be made that it can’t be used for games where the intent is for each player to have uniquely generated plants. In cases where this is not the case plants can be evolved, and a supervisor can then cull those that are visually unappealing.

The reason to use an evolutionary algorithm was to create a diverse yet coherent fauna for a game environment, where plants had an apparent relationship with the other species of plants inhabiting the same environment. Though the plants developed niches since the plants weren’t given the means to change the shape and size of leaves and branches it was impossible for the plants to make themselves truly unique and visually distinct. This is however something that can be added to the simulation, but as it could dramatically alter the dynamic of the program results found here it would be of little use.

6.2 Discussion

The data shown in chapter 5 was sampled from randomly selected runs of the program. No effort was made to select specific runs that more fairly represents the general behavior of the program, but this also means that the selected data couldn’t have been selected because it was more in line with any preconceptions held.

It is not very likely that anyone would consider this paper a contribution to science, but just as Ebners (2006) paper presented a basis for the simulation used in this thesis other future prospectors of this field may use this thesis in a similar manner. For game developers it may act as something to start with and work from when developing software for evolved procedural plants.
While there are no apparent social implications this thesis has on everyday life there may be some peripheral effects that aren’t a directly result of the conclusions found. It might be beneficial for schools to use software used in scientific research, this could help in teaching different aspects of science and the scientific method. All code for the simulation used is released with the paper, which may be useful for explaining how coevolution and niches are formed in plants. Perhaps if schools and scientists shared software used in their research like this then the understanding of scientific research would be better. This could impact the general sharing of software typically not available to schools. Other planets have different temperature, atmospheric composition and gravitational pull. If the simulation took into account things like these the simulation could be used for exploring how plants could evolve on other planets.

6.3 Future work

While there are a lot of interesting data one can sample and analyze from this simulation there are many features that would make it more realistic and complete. In this simulation the effect genes could have on a plant is very limited and would produce more diverse and realistic plants if they either were randomly given different types of leaves or letting genes determine their shape and size. In this simulation it is very improbable for speciation to occur within a species, therefore it would be interesting to implement mechanisms for assortative mating. This would mean allowing plants to evolve preferences for mating and thereby allowing the resulting species of speciation to stay genetically isolated from each other. If more time had been available it would have been of great interest to implement these features.

As it stands now this simulation is of little use, but there is no open source platform for this kind of research. The code for this program is therefore released in the hope that it makes it easier for interested parties to make their own simulations. If a game company would use the resulting simulation as a starting point for their own procedurally generated plants there would be much to fix. But if the work done here is of any use to anyone I would be greatly honored to have been useful. Someday there might be a software platform for everyone in the field to base their work on. A common platform would result in fewer bugs and a less biased result as more people are examining the code. It would also reduce time needed to implement the simulation as code from the common base can be reused.
References


Appendix A - L-system plant

Here there are two plants presented from the simulation using 2 species, 100 in population size and placement in the shadow of the parent.

Species 1 of generation 155

<table>
<thead>
<tr>
<th>Letter</th>
<th>Substitution string</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td><a href="/"></a>[&gt;/f+-[&lt;&gt;]]-f[]<a href=""></a>&gt;\</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>[]/\</td>
</tr>
<tr>
<td>C</td>
<td>-&lt;[]</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>E[]</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Species 2 of generation 155

<table>
<thead>
<tr>
<th>Letter</th>
<th>Substitution string</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>B[][]/[]][f]&gt;[]][[]][&lt;[&gt;]+[]</td>
</tr>
<tr>
<td>A</td>
<td>E[]</td>
</tr>
<tr>
<td>B</td>
<td>[]][A</td>
</tr>
<tr>
<td>C</td>
<td>AE[&lt;[]]\</td>
</tr>
<tr>
<td>D</td>
<td>[]][EFA</td>
</tr>
<tr>
<td>E</td>
<td>[]][EA</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Appendix B - Plant simulation source code

common.h

#ifndef COMMON_H
#define COMMON_H

const float PI = 3.14159265f;

#endif

csv.h

#ifndef CVS_H
#define CVS_H

#include <fstream>
#include <string>

class CSV
{
private:
    char* file_path;
    std::ofstream file;
    std::string current_row;

public:
    CSV(char* file_path);
    ~CSV();
    void AddElement(unsigned int element);
    void NextRow();
    void Reset();
};

#endif

csv.cpp

#include "csv.h"
#include <sstream>

CSV::CSV(char* file_path) : file_path(file_path), file(file_path, std::ios::trunc) {} 

CSV::~CSV()
{
    if(current_row != "")
        NextRow();
    file.close();
}

void CSV::AddElement(unsigned int element)
{
    if(current_row != "")
        current_row += ";

    std::stringstream converter;
    converter<<element;
...
```
current_row += converter.str();
}
void CSV::NextRow()
{
    file<<current_row<<std::endl;
    current_row.clear();
}
void CSV::Reset()
{
    file.close();
    file.open(file_path, std::ios::trunc);
}

environment.h
#ifndef ENVIRONMENT_H
#define ENVIRONMENT_H
#include "plantmodel.h"
#include "plant.h"
#include "fbo.h"
namespace Environment
{
    void SetParameters(unsigned int pop_size, unsigned int num_species, bool save = false, bool random = false);
    void Update(FBO* fbo);
    void Draw();
    void RenderShadow(FBO* fbo);
    int NumGenerations();
    void UpdateFitness(FBO* fbo);
    void SelectNextGen(FBO* fbo);
    void MutatePopulation();
    void UpdateModels();
};
#endif

environment.cpp
#include "environment.h"
#include "window.h"
#include "floor.h"
#include "common.h"
#include <vector>
#include <cmath>
#include <fstream>
#include <sstream>
#include "csv.h"
static const int MAX_POP_SIZE = 100;
static const int MAX_NUM_SPECIES = 3;
static Plant plants_pop1[MAX_POP_SIZE * MAX_NUM_SPECIES];
static Plant plants_pop2[MAX_POP_SIZE * MAX_NUM_SPECIES];
```
static Plant *parent;
static Plant *child;
static PlantModel plant_models[MAX_POP_SIZE * MAX_NUM_SPECIES];

static int number_of_generations = 0;
static unsigned int number_of_species = 0;
static unsigned int population_size = 0;
static unsigned int total_plants = 0;

unsigned int fitness[MAX_POP_SIZE * MAX_NUM_SPECIES];
unsigned int complexity[MAX_POP_SIZE * MAX_NUM_SPECIES];
static unsigned char pixels[3*512*512];
static int SelectRandom(Plant *plants, unsigned int *fitness, int pop_size);
static bool save_data = false;
static bool random_placement = false;
static CSV csv("excel_data.csv");

unsigned int Average(unsigned int elements[], unsigned int size)
{
    long int sum = 0;
    for(int i=0; i<size; i++)
        sum += elements[i];
    return sum / size;
}

void SaveGeneration();

int Environment::NumGenerations()
{ return number_of_generations; }

void Environment::SetParameters(unsigned int pop_size, unsigned int num_species, bool save, bool random)
{
    save_data = save;
    random_placement = random;
    number_of_generations = 0;
    number_of_species = num_species;
    population_size = pop_size;
    total_plants = population_size * number_of_species;

    const float INCREMENT = (PI * 2.0f) / float(pop_size);
    const float OFFSET = (PI * 2.0f) / float(pop_size * num_species);

    for(unsigned int i=0; i < number_of_species; i++)
    {
        for(unsigned int j=0; j < population_size; j++)
        {
            int index = i * population_size + j;
            plants_pop1[index].l_sys.HardReset();
            plants_pop1[index].pos.x = floor_halfsize + sinf(INCREMENT * j + OFFSET * i) * (floor_halfsize / 2);
            plants_pop1[index].pos.y = -floor_halfsize + cosf(INCREMENT * j + OFFSET * i) * (floor_halfsize / 2);
            complexity[index] = plant_models[index].Generate("f", plants_pop1[index].pos.x, plants_pop1[index].pos.y);
        }
    }

    parent = &plants_pop1[0];
    child = &plants_pop2[0];
void Environment::Update(FBO *fbo)
{
    fbo->SetActive();
    Window::SetProjOrtho();

    UpdateFitness(fbo);
    SelectNextGen(fbo);
    MutatePopulation();
    UpdateModels();

    Plant* temp = parent;
    parent = child;
    child = temp;

    if(save_data)
        SaveGeneration();

    number_of_generations++;
    fbo->SetToDefault();
}

void Environment::UpdateFitness(FBO* fbo)
{
    //Draw to sun_texture
    Window::BeginDrawing();
    glRotatef(90, 1.0f, 0.0f, 0.0f);
    glTranslatef(0.0f, -1000.0f, 0.0f);
    Draw();
    Floor::DrawColored();

    glFinish(); //Wait until finished drawing
    fbo->GetTextureData(&pixels); //Get pixels from renderTarget

    //Calculate Fitness ---
    ZeroMemory(&fitness, sizeof(unsigned int) * total_plants);
    for(unsigned int i=0; i < 3*512*512; i += 3)
        if(pixels[i] != 255 && !(pixels[i+0] == 0 && pixels[i+1] == 0 &&
            pixels[i+2] == 0))
            fitness[255 - pixels[i+1]]++;
    for(unsigned int i=0; i < total_plants; i++)
        fitness[i] *= 10;

    //Subtract complexity from fitness
    for(unsigned int i=0; i < total_plants; i++)
    {
        if(complexity[i] >= fitness[i])
            fitness[i] = 0;
        else
            fitness[i] -= complexity[i];
    }

    if(save_data)
    {
        static CSV csv("excel_data.csv");
        int sum = 0;
        for(unsigned int i = 0; i < number_of_species; i++)
        {
int average = Average(&fitness[i * population_size], population_size);
csv.AddElement(average);
sum += average;
}
csv.AddElement(sum / number_of_species);
csv.NextRow();
}

} //Select next generation ---
if(!random_placement)
{
static std::vector<Point> my_points[MAX_POP_SIZE * MAX_NUM_SPECIES];
for(int i=0; i<MAX_POP_SIZE * MAX_NUM_SPECIES; i++)
    my_points[i].clear();
for(unsigned int i=0; i < number_of_species; i++)
{
    for(unsigned int j=0; j < population_size; j++)
    {
        int offset = i * population_size;
        int parent_id = offset + SelectRandom(&parent[offset], &fitness[offset], population_size);
        if(my_points[parent_id].size() == 0)
        {
            //Render shadow of parent
            Window::BeginDrawing();
            glRotatef(90, 1.0f, 0.0f, 0.0f);
            glTranslatef(0.0f, -1000.0f, 0.0f);
            glColor3ub(0, 0, 0);
            plant_models[parent_id].DrawBranches();
            plant_models[parent_id].DrawLeaves();
            glFinish();
            fbo->GetTextureData(&pixels);
            for(unsigned int u=0; u<512; u++)
            {
                for(unsigned int k=0; k<512; k++)
                {
                    if(pixels[(u * 512 + k) * 3] == 0)
                    {
                        Point point = { float(k) / 512 * floor_halfsize * 2, -float(u) / 512 * floor_halfsize * 2 };
                        my_points[parent_id].push_back(point);
                    }
                }
            }
        }
        int index = offset + j;
        child[index] = parent[parent_id];
        if(my_points[parent_id].size() > 0)
            child[index].pos = my_points[parent_id].at(rand() % my_points[parent_id].size());
        else
            child[index].pos = parent[parent_id].pos;
    }
}
else
{
    for(unsigned int i=0; i < number_of_species; i++)
    {
        for(unsigned int j=0; j < population_size; j++)
        {
            int offset = i * population_size;
            int parent_id = offset + SelectRandom(&parent[offset], &fitness[offset], population_size);
            int index = offset + j;
            child[index] = parent[parent_id];
            child[index].pos.x = float(rand() % 512) / 512 * floor_halfsize * 2;
            child[index].pos.y = -float(rand() % 512) / 512 * floor_halfsize * 2;
        }
    }
}
void Environment::MutatePopulation()
{
    //Mutate new generation ---
    for(unsigned int i=0; i < number_of_species; i++)
    {
        for(unsigned int j=0; j < population_size; j++)
        {
            int offset = i * population_size;
            unsigned int index = i * population_size + j;
            switch(rand()%10)
            {
            case 0:
                child[index].l_sys.Permutate();
                break;
            case 1:
                child[index].l_sys.Mutate();
                break;
            case 2:
                child[index].l_sys.Insertion();
                break;
            case 3:
                child[index].l_sys.Deletion();
                break;
            case 4:
                child[index].l_sys.OnePointCrossover( parent[offset + SelectRandom(&parent[offset], &fitness[offset], population_size)].l_sys );
                break;
            case 5:
                child[index].l_sys.SubTreeCrossover( parent[offset + SelectRandom(&parent[offset], &fitness[offset], population_size)].l_sys );
                break;
            case 6:
                child[index].l_sys.AddBranch();
                break;
            case 7:
                child[index].l_sys.DeleteBranch();
                break;
            case 8:
                child[index].l_sys.AddRule();
                break;
            case 9:
                child[index].l_sys.DeleteRule();
                break;
            }
void Environment::UpdateModels()
{
    for(unsigned int i=0; i < total_plants; i++)
    {
        child[i].l_sys.Reset();
        child[i].l_sys.Update(5);
        complexity[i] = plant_models[i].Generate(child[i].l_sys.GetResult(),
                child[i].pos.x, child[i].pos.y);
    }
}

void Environment::RenderShadow(FBO *fbo)
{
    glPushMatrix();
    fbo->SetActive();
    Window::SetProjOrtho();
    Window::BeginDrawing();
    glRotatef(90, 1.0f, 0.0f, 0.0f);
    glTranslatef(0.0f, -1000.0f, 0.0f);
    glColor3ub(0, 0, 0);
    for(unsigned int i=0; i < total_plants; i++)
    {
        plant_models[i].DrawLeaves();
        plant_models[i].DrawBranches();
    }
    Floor::DrawColored();
    glFinish();
    //Set back to default fbo
    fbo->SetToDefault();
    Window::SetProjPers();
    glPopMatrix();
}

void Environment::Draw()
{
    glColor3ub(0, 0, 0);
    for(unsigned int i=0; i < total_plants; i++)
        plant_models[i].DrawBranches();

    for(unsigned int i=0; i < total_plants; i++)
    {
        unsigned char blue = number_of_species > 1 ? (i / population_size) * 255 / (number_of_species-1) : 0;
        glColor3ub(0, 255 - static_cast<unsigned char>(i), blue);
        plant_models[i].DrawLeaves();
    }
}

int SelectRandom(Plant *plants, unsigned int *fitness, int pop_size)
{
    int selection[7];
    for(int i=0; i<7; i++)
        selection[i] = rand() % pop_size;
unsigned int total_fit = 0;
for(int i=0; i<7; i++)
{
    if(total_fit > UINT_MAX - fitness[selection[i]])
    {
        //Would have overflowed
        static bool overflowed_before = false;
        if(!overflowed_before)
            MessageBox(NULL, L"SelectRandom has overflowed", L"Overflow Error", MB_OK | MB_ICONERROR);
        overflowed_before = true;
        total_fit = UINT_MAX;
        break;
    }
    else
        total_fit += fitness[selection[i]];
}
if(total_fit == 0)
    return selection[0];

unsigned int random = static_cast<unsigned int>(rand());
random %= total_fit;
for(int i=0; i<7; i++)
{
    if(random > fitness[selection[i]])
        random -= fitness[selection[i]];
    else
        return selection[i];
}
return selection[0];

void SaveGeneration()
{
    for(unsigned int i=0; i < number_of_species; i++)
    {
        stringstream dir;
        dir<<"PlantData/G"<<number_of_generations<<"S"<<i<<".txt";
        string temp = dir.str();
        ofstream out(temp.c_str(), std::ios::trunc);
        if(!out.is_open())
            continue;

        unsigned int best = i * population_size;
        // Find best in generation
        for(unsigned j = 0; j < population_size; j++)
        {
            //Save stuff
            int index = i * population_size + j;
            if(fitness[index] > fitness[best])
                best = index;
        }

        out.write(reinterpret_cast<char*>(&parent[best].pos), sizeof(Point));
        out.write(reinterpret_cast<char*>(&parent[best].l_sys.num_active_alphabet),
        sizeof(int));
```cpp
out<<(parent[best].l_sys.substitutions_prim)<<std::endl;
for (int k = 0; k < parent[best].l_sys.num_active_alphabet; k++)
    out<<(parent[best].l_sys.substitutions[k])<<std::endl;
}
}

fbo.h

#ifndef FBO_H
#define FBO_H

#include <cassert>
#include "window.h"

class FBO
{
private:
    GLuint fbo_id;
    GLuint depthbuffer;
    GLuint img;
    unsigned int width, height;
public:
    FBO(unsigned int width, unsigned int height) : width(width), height(height)
    {
        glGenFramebuffersEXT(1, &fbo_id);
        SetActive();

        // Create a depthbuffer
        glGenRenderbuffersEXT(1, &depthbuffer);
        glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depthbuffer);
        glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT, width, height);
        glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT, GL_RENDERBUFFER_EXT, depthbuffer);

        // Create a colorbuffer
        glGenTextures(1, &img);
        glBindTexture(GL_TEXTURE_2D, img);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, img, 0);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        GLenum status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
        if(status != GL_FRAMEBUFFER_COMPLETE_EXT)
            assert(!"Framebuffer couldn't be created");
        SetToDefault();
    }

    ~FBO()
    {
        glDeleteFramebuffersEXT(1, &fbo_id);
        glDeleteRenderbuffersEXT(1, &depthbuffer);
    }

};
```
inline void SetActive()
{
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo_id);
    glPushAttrib(GL_VIEWPORT_BIT);
    glViewport(0,0,width, height);
}

static inline void SetToDefault()
{
    glPopAttrib();
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
}

void GetTextureData(void* buffer)
{
    glGetTexImage(
        GL_TEXTURE_2D, //Target
        0, //Level
        GL_RGB, //
        GL_UNSIGNED_BYTE, //
        buffer //
    );
}

GLuint GetTextureId()
{
    return img;
}

#endif

floor.h

#ifndef FLOOR_H
#define FLOOR_H

#include "window.h"

const float floor_halfsize = 50.0f;

namespace Floor
{
    void Initialize();
    void DrawColored();
    void DrawTextured(GLuint tex_id);
};
#endif

floor.cpp

#include "floor.h"
#include "window.h"

static GLuint floor_id;

void Floor::Initialize()
{
    floor_id = glGenLists(1);
    glNewList(floor_id, GL_COMPILE);
    glBegin(GL_QUADS);

```cpp
void Floor::DrawColored()
{
    glDisable(GL_TEXTURE_2D);
    glColor3ub(255, 255, 0);
    glCallList(floor_id);
}
void Floor::DrawTextured(GLuint tex_id)
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, tex_id);
    glColor3ub(255, 255, 255);
    glCallList(floor_id);
    glDisable(GL_TEXTURE_2D);
}
```

```cpp
lsystem.h
```

```cpp
#ifndef LSYSTEM_H
#define LSYSTEM_H
#include <string>

class Lsystem
{
    friend void SaveGeneration();
private:
    int num_active_alphabet;
    std::string substitutions_prim;
    std::string substitutions[26];
    std::string result;
public:
    Lsystem();
    void Reset()
    { result = "f"; }
    void HardReset();
    void Update(int times = 1);
    //Accessors  --------------------------
    std::string GetResult();
};
```

40
void SetResult(std::string r)
{ result = r; }

std::string GetRule(char l)
{
    if(l == 'f')
        return substitutions_prim;
    else
        return substitutions[l - 'A'];
}

void SetRule(char symbol, std::string substitute);
void SetRuleSize(int num);

// Mutations -----------------------------
void Permutate();
void Mutate();
void Insertion();
void Deletion();
void OnePointCrossover(const Lsystem& parent);
void SubTreeCrossover(const Lsystem& parent);
void AddBranch();
void DeleteBranch();
void AddRule();
void DeleteRule();
};

#endif

lsystem.cpp

#include "lsystem.h"
#include <cmath>
#include <vector>

static const char symbol_alph[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
static const char random_symbol[] = "fl<>+/+-ABCDEFGHIJKLMNOPQRSTUVWXYZ";

// CONSTRUCTOR ------------------------------------

Lsystem::Lsystem() :
    num_active_alphabet(0)
{
    Reset();
    substitutions_prim = "f";
    for(int i=0; i<26; i++)
        substitutions[i] = symbol_alph[i];
}

void Lsystem::HardReset()
{
    Reset();
    substitutions_prim = "f";
    for(int i=0; i<26; i++)
        substitutions[i] = symbol_alph[i];
}

// STUFF ----------------------------------------

std::string Lsystem::GetResult()
{ return result; }

//
void Lsystem::Update(int times)
{
    std::string new_string;

    for(int i=0; i<times; i++)
    {
        for(unsigned int j = 0; j < result.length(); j++)
        {
            if(result[j] == 'f')
                new_string += substitutions_prim;
            else if(result[j] >= 'A' && result[j] <= 'Z' && result[j] - 'A' < num_active_alphabet)
                new_string += substitutions[result[j] - 'A'];
            else
                new_string += result[j];
        }
        result = new_string;
        new_string.clear();
    }
}

void Lsystem::SetRule(char symbol, std::string substitute)
{
    if(symbol == 'f')
        substitutions_prim = substitute;
    else
    {
        int index = symbol - 'A';
        if(index >= 0 && index < 26)
            substitutions[index] = substitute;
    }
}

void Lsystem::SetRuleSize(int num)
{
    for(int i = num_active_alphabet; i < num && i < 26; i++)
        substitutions[i] = symbol_alph[i];
    num_active_alphabet = num;
}

// MUTATIONS --------------------------------------

void Lsystem::Permutate()
{
    int index = rand() % (num_active_alphabet+1);
    std::string* subs_ptr = 0;
    if(index == 0)
        subs_ptr = &substitutions_prim;
    else
        subs_ptr = &substitutions[index-1];

    if(subs_ptr->size() < 2)
        return;

    int letter = rand() % (subs_ptr->size()-1);
    if((*subs_ptr)[letter] != '[' && (*subs_ptr)[letter] != ']' &&
    {
        char temp = (*subs_ptr)[letter];
        (*subs_ptr)[letter] = (*subs_ptr)[letter+1];
        (*subs_ptr)[letter+1] = temp;
    }
}
void Lsystem::Mutate()
{
    int index = rand() % (num_active_alphabet + 1);
    std::string* subs_ptr = 0;

    if(index == 0)
        subs_ptr = &substitutions_prim;
    else
        subs_ptr = &substitutions[index - 1];

    if(subs_ptr->size() <= 0)
        return;

    int letter = rand() % (subs_ptr->size());
    if((*subs_ptr)[letter] != '[' && (*subs_ptr)[letter] != ']
    (*subs_ptr)[letter] = random_symbol[rand() % (8 + num_active_alphabet)];
}

void Lsystem::Insertion()
{
    int index = rand() % (num_active_alphabet + 1);
    std::string* subs_ptr = 0;

    if(index == 0)
        subs_ptr = &substitutions_prim;
    else
        subs_ptr = &substitutions[index - 1];

    if(subs_ptr->size() == 0)
        (*subs_ptr) += random_symbol[rand() % (8 + num_active_alphabet)];
    else
    {
        int letter = rand() % (subs_ptr->size() + 1);
        std::string temp = (*subs_ptr).substr(0, letter);
        temp += random_symbol[rand() % (8 + num_active_alphabet)];
        if(subs_ptr->size() - letter > 0)
            temp += (*subs_ptr).substr(letter, subs_ptr->size() - letter);
        *subs_ptr = temp;
    }
}

void Lsystem::Deletion()
{
    int index = rand() % (num_active_alphabet + 1);
    std::string* subs_ptr = 0;

    if(index == 0)
        subs_ptr = &substitutions_prim;
    else
        subs_ptr = &substitutions[index - 1];

    if(subs_ptr->size() <= 1)
        *subs_ptr = "";
    else

std::string temp = (*subs_ptr).substr(0, letter);
letter++;
if(subs_ptr->size() - letter > 0)
    temp += (*subs_ptr).substr(letter, subs_ptr->size() - letter);

*subs_ptr = temp;
}

void Lsystem::OnePointCrossover(const Lsystem& parent)
{
    int max_num_rules = std::min(num_active_alphabet, parent.num_active_alphabet);
    int random_rule = rand() % (max_num_rules + 1);
    if(random_rule == 0)
        substitutions_prim = parent.substitutions_prim;
    else
        substitutions[random_rule] = parent.substitutions[random_rule - 1];
}

inline int SelectRandomBegining(std::string rule)
{
    std::vector<int> beginings;
    for(int i=0; i<rule.size(); i++)
        if(rule[i] == '[')
            beginings.push_back(i);
    if(beginings.size() == 0)
        return -1;
    else
        return beginings[rand()%beginings.size()];
}

inline std::string GetBranch(std::string rule)
{
    int begining = SelectRandomBegining(rule);
    if(begining == -1)
        return "";
    int bracket_cntr = 0;
    for(unsigned int i = begining; i < rule.size(); i++)
        if(rule[i] == '[')
            bracket_cntr++;
        else if(rule[i] == ']
        {
            if(bracket_cntr > 0)
                bracket_cntr--;
            else
                return rule.substr(begining, i-begining);
        }
    return "";
}

void Lsystem::SubTreeCrossover(const Lsystem& parent)
int rule_index = rand() % (num_active_alphabet + 1);
std::string rule = rule_index == 0 ? substitutions_prim : substitutions[rule_index - 1];

//Add the beginning
int beginning = SelectRandomBeginning(rule);
if(begining == -1)
    return;
std::string result = rule.substr(0, beginning);

//Add the middle
int other_rule_index = rand() % (num_active_alphabet + 1);
std::string other_rule = other_rule_index == 0 ? parent.substitutions_prim : parent.substitutions[other_rule_index - 1];
result += GetBranch(other_rule);

//Add the end
int bracket_cntr = 0;
for(unsigned int i = beginning; i < rule.size(); i++)
{
    if(rule[i] == '[')
        bracket_cntr++;
    else if(rule[i] == ']
    {
        if(bracket_cntr > 0)
            bracket_cntr--;
        else
            
                result += rule.substr(i, std::string::npos);
                break;
    }
}

void Lsystem::AddBranch()
{
    int index = rand() % (num_active_alphabet + 1);
    std::string* subs_ptr = 0;
    if(index == 0)
        subs_ptr = &substitutions_prim;
    else
        subs_ptr = &substitutions[index - 1];

    if(subs_ptr->size() == 0)
        *subs_ptr += "[");
    else
    {
        int letter = rand() % (subs_ptr->size() + 1);
        std::string temp = (*subs_ptr).substr(0, letter);
        temp += "[");
        if(subs_ptr->size() - letter > 0)
            temp += (*subs_ptr).substr(letter, subs_ptr->size() - letter);
        *subs_ptr = temp;
    }
}
void Lsystem::DeleteBranch()
{
    int rule_index = rand() % (num_active_alphabet + 1);
    std::string rule = rule_index == 0 ? substitutions_prim : substitutions[rule_index - 1];

    // Add the beginning
    int beginning = SelectRandomBeginning(rule);
    if(beginning == -1)
    {
        return;
    }
    std::string result = rule.substr(0, beginning);

    // Add the end
    int bracket_cntr = 0;
    for(unsigned int i = beginning; i < rule.size(); i++)
    {
        if(rule[i] == '[')
        {
            bracket_cntr++;
        }
        else if(rule[i] == ']
        {
            if(bracket_cntr > 0)
            {
                bracket_cntr--;
            }
            else
            {
                result += rule.substr(i, std::string::npos);
                break;
            }
        }
    }
}

void Lsystem::AddRule()
{
    if(num_active_alphabet < 26)
    {
        substitutions[num_active_alphabet] = symbol_alph[num_active_alphabet];
        num_active_alphabet++;
    }
}

void Lsystem::DeleteRule()
{
    if(num_active_alphabet > 0)
    {
        num_active_alphabet--;
        if(result.find(symbol_alph[num_active_alphabet]) != std::string::npos)
        {
            std::string temp = "";
            for(unsigned int j=0; j < result.size(); j++)
            {
                if(result.at(j) != symbol_alph[num_active_alphabet])
                {
                    temp += result.at(j);
                }
            }
            result = temp;
        }
        // MUST remove letters in substitutions
        for(int i=0; i < num_active_alphabet; i++)
        {
        }
}
if(substitutions[i].find(symbol_alph[num_active_alphabet]) ==
std::string::npos)
    continue;

std::string temp = "";
for(unsigned int j=0; j < substitutions[i].size(); j++)
{
    if(substitutions[i].at(j) != symbol_alph[num_active_alphabet])
        temp += substitutions[i].at(j);
}
substitutions[i] = temp;

plant.h

#ifndef PLANT_H
#define PLANT_H

#include <string>
#include "lsystem.h"
#include "plantmodel.h"

struct Point
{ float x, y; };

struct Plant
{
    Lsystem l_sys;
    Point pos;
};
#endif

plantmodel.h

#ifndef PLANTMODEL_H
#define PLANTMODEL_H

#include <string>

class PlantModel
{
    private:
        unsigned int display_list_id;
    public:
        PlantModel();
        PlantModel(std::string data, float position_x, float position_y);
        void DrawBranches();
        void DrawLeaves();
        unsigned int Generate(std::string data, float position_x, float position_y);
};
#endif

plantmodel.cpp
```cpp
#include "plantmodel.h"
#include "common.h"
#include <cmath>
#include <vector>
#include <stack>
#include "GL/glew.h"
#include <Windows.h>

static const float SCALAR = 2.0f;
static const float BRANCH_LENGTH = 1.0f * SCALAR;
static const float BRANCH_WIDTH = 0.25f * SCALAR;
static const float LEAF_LENGTH = 2.5f * SCALAR;
static const float LEAF_WIDTH = 1.0f * SCALAR;
static const int SIDES = 6;

static int leaf_list;
static int branch_list;
static int knot_list;
static bool initiated = false;

struct PlantModelVertex
{
    GLfloat vx, vy, vz;
    GLfloat nx, ny, nz;
};

static void Initiate();
static unsigned int CreateBranch(const std::string& data, GLint id, float position_x, float position_y);
static void CreateLeaves(const std::string& data, GLint id, float position_x, float position_y);

PlantModel::PlantModel() :
    display_list_id(-1)
{};

PlantModel::PlantModel(std::string data, float position_x, float position_y) :
    display_list_id(-1)
{
    Generate(data, position_x, position_y);
}

void PlantModel::DrawBranches()
{
    glCallList(display_list_id);
}

void PlantModel::DrawLeaves()
{
    glCallList(display_list_id + 1);
}

unsigned int PlantModel::Generate(std::string data, float position_x, float position_y)
{
    if(!initiated){ Initiate(); initiated = true; }

    if(display_list_id == -1)
    {
```
display_list_id = glGenLists(2);
if(display_list_id == GL_INVALID_VALUE || display_list_id ==
GL_INVALID_OPERATION)
    {/* delete plantmodel; return (PlantModel*)0xABADBABE; */
}
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// Generate list for branches
CreateLeaves(data, display_list_id+1, position_x, position_y);
return CreateBranch(data, display_list_id, position_x, position_y);
}
static unsigned int CreateBranch(const std::string& data, GLint id, float position_x,
float position_y)
{
    //bool straight = true;
double complexity = 0;
unsigned int long_comp = 0;

    glNewList(id, GL_COMPILE_AND_EXECUTE);
    glPushMatrix();
    glTranslatef(position_x, 0.0f, position_y);
    static float matrix[16];
    for(unsigned int i=0; i<data.size(); i++)
    {
        switch(data[i])
        {
        case 'f':
            glCallList(branch_list); //Draw branch
            glTranslatef(0, BRANCH_LENGTH, 0); //Move forward
            glGetFloatv(GL_MODELVIEW_MATRIX, &matrix[0]);
            if(matrix[13] >= 0)
                complexity += 1.0 * pow(1.1,
double(matrix[13]));
            else
                complexity += 2.0 * pow(1.1,
double(-matrix[13]));
            break;
        case 'l':
            glGetFloatv(GL_MODELVIEW_MATRIX, &matrix[0]);
            if(matrix[13] >= 0)
                complexity += 3.0 * pow(1.1,
double(matrix[13]));
            else
                complexity += 6.0 * pow(1.1,
double(-matrix[13]));
            break;
        case '[':
            glPushMatrix();
            break;
        case ']':
            glPopMatrix();
            break;
        case '>':
            glRotatef(22.5f, 1, 0, 0);
//straight = false;
break;
case '<':	glRotatef(-22.5f, 1, 0, 0);
//straight = false;
break;

//Rotation Y
case '\\':	glRotatef(-22.5f, 0, 1, 0);
//straight = false;
break;
case '/':	glRotatef(22.5f, 0, 1, 0);
//straight = false;
break;

//Rotation Z
case '+':	glRotatef(22.5f, 0, 0, 1);
//straight = false;
break;
case '-':	glRotatef(-22.5f, 0, 0, 1);
//straight = false;
break;

default:
    continue;
}

if(complexity >= 10000.0f && long_comp < UINT_MAX - static_cast<unsigned int>(complexity))
{
    long_comp += 10000;
    complexity-= 10000.0f;
}
else
    long_comp = UINT_MAX;

glPopMatrix();
glEndList(); //Branches End
return long_comp + static_cast<unsigned int>(complexity);
}

static void CreateLeaves(const std::string& data, GLint id, float position_x, float position_y)
{
    // Generate list for leaves
    glNewList(id, GL_COMPILE);
    glPushMatrix();
    //glLoadIdentity();
    glTranslatef(position_x, 0.0f, position_y);
    for(unsigned int i=0; i<data.size(); i++)
    {
        switch(data[i])
        {
        case 'f':	glTranslatef(0, BRANCH_LENGTH, 0); //Move forward
            break;
        case 'l':
            break;

        default:
            continue;
        }

        if(complexity >= 10000.0f && long_comp < UINT_MAX - static_cast<unsigned int>(complexity))
        {
            long_comp += 10000;
            complexity-= 10000.0f;
        }
        else
            long_comp = UINT_MAX;
    }

glPopMatrix();
glEndList(); //Branches End
return long_comp + static_cast<unsigned int>(complexity);
}
glCallList(leaf_list); //Draw leaf
break;

//Push orientation and position
case '[':
glPushMatrix();
break;
case ']':
glPopMatrix();
break;

//Rotation X
case '>':
glRotatef(22.5f, 1, 0, 0);
break;
case '<':
glRotatef(-22.5f, 1, 0, 0);
break;

//Rotation Y
case '\\':
glRotatef(-22.5f, 0, 1, 0);
break;
case '/':
glRotatef(22.5f, 0, 1, 0);
break;

//Rotation Z
case '+':
glRotatef(22.5f, 0, 0, 1);
break;
case '-':
glRotatef(-22.5f, 0, 0, 1);
break;

default:
    continue;
    }
glPopMatrix();
glEndList(); // End Leaves
}
}

static void Initiate()
{
    //Leaf
    leaf_list = glGenLists(1);
gNewList(leaf_list, GL_COMPILE);

    glBegin(GL_TRIANGLES_FAN);
    glVertex3f(0.0f, 0.0f, 1.0f); // NORMAL
    glVertex3f(LEAF_WIDTH/2, 1.0f * LEAF_LENGTH/3, 0.0f); //5
    glVertex3f(LEAF_WIDTH/2, 2.0f * LEAF_LENGTH/3, 0.0f); //4
    glVertex3f(LEAF_WIDTH/2, 2.0f * LEAF_LENGTH/3, -1.0f); //3
    glVertex3f(LEAF_WIDTH/2, 1.0f * LEAF_LENGTH/3, 0.0f); //2
    glVertex3f(0.0f, 0.0f * LEAF_LENGTH/3, 0.0f); //1
    glEnd();

    glBegin(GL_TRIANGLES_FAN);
    glVertex3f(0.0f, 0.0f, -1.0f); // NORMAL
    glVertex3f(0.0f, 0.0f, 0.0f); //0
    glEnd();
}
glVertex3f(-LEAF_WIDTH/2, 1.0f * LEAF_LENGTH/3, 0.0f); //1
glVertex3f(-LEAF_WIDTH/2, 2.0f * LEAF_LENGTH/3, 0.0f); //2
glVertex3f( 0.0f, 3.0f * LEAF_LENGTH/3, 0.0f); //3
glVertex3f( LEAF_WIDTH/2, 2.0f * LEAF_LENGTH/3, 0.0f); //4
glVertex3f( LEAF_WIDTH/2, 1.0f * LEAF_LENGTH/3, 0.0f); //5
glEnd();

//Branch
branch_list = glGenLists(1);
glNewList(branch_list, GL_COMPILE);
glBegin(GL_QUAD_STRIP);
for(int i=0; i < SIDES+1; i++)
{
    //glNormal3f(float(sin(i/6.0 * 2*PI - PI/6)), 0.0f, float(cos(i/6.0 * 2*PI - PI/6))); // For flat shading
    glNormal3f(float(sin(i/6.0 * 2*PI)), 0.0f, float(cos(i/6.0 * 2*PI))); // For smooth shading
    glVertex3f(float(sin(i/6.0 * 2*PI)) * BRANCH_WIDTH, BRANCH_LENGTH, float(cos(i/6.0 * 2*PI)) * BRANCH_WIDTH);
    glVertex3f(float(sin(i/6.0 * 2*PI)) * BRANCH_WIDTH, 0.0f, float(cos(i/6.0 * 2*PI)) * BRANCH_WIDTH);
}
glEnd();
glEndList();

//Knot
knot_list = glGenLists(1);
glNewList(knot_list, GL_COMPILE);
glBegin(GL_TRIANGLE_FAN);
for(int i=0; i < SIDES+1; i++)
{
    //glNormal3f(0.0f,0.0f,0.0f);
    glVertex3f(float(sin(i/6.0 * 2*PI)) * BRANCH_WIDTH / 2, 0.2f, float(cos(i/6.0 * 2*PI)) * BRANCH_WIDTH / 2);
}
glEnd();
glBegin(GL_QUAD_STRIP);
for(int i=0; i < SIDES+1; i++)
{
    glVertex3f(float(sin(i/6.0 * 2*PI)) * BRANCH_WIDTH / 2, 0.2f, float(cos(i/6.0 * 2*PI)) * BRANCH_WIDTH / 2);
    glVertex3f(float(sin(i/6.0 * 2*PI)) * BRANCH_WIDTH, 0.0f, float(cos(i/6.0 * 2*PI)) * BRANCH_WIDTH);
}
glEnd();
glEndList();

point.h
#ifndef POINT_H
#define POINT_H

template<typename T>
struct Point
{
    T x, y;
};

typedef Point<int> Pointi;
typedef Point<float> Pointf;
#endif

simulation.h

#ifndef SIMULATION_H
#define SIMULATION_H

namespace Simulation
{
    void Initialize(bool save, bool random = false);
    void Update();
    void Toggle();
    void Draw();
};
#endif

simulation.cpp

#include "simulation.h"
#include "environment.h"
#include "window.h"
#include "floor.h"
#include "fbo.h"
#include <ctime>

static const int NUM_PRESETS = 4;
static struct{
    int num_species;
    int pop_size;
}presets[NUM_PRESETS] =
{
    {2, 100},
    {3, 50},
    {2, 50},
    {1, 100}
};

static int toggle = 0;
static FBO *fbo;

static float count = 0.0f;
static float closeness = 1.0f;
static bool update_toggle = false;
static bool draw_shadow = true;

void DrawOrtho();
void DrawPersp();

static bool save_m;
static bool random_m;

void Simulation::Initialize(bool save, bool random)
{
    srand(unsigned int(time(0)));
    //srand(1337);

    save_m = save;
    random_m = random;

    fbo = new FBO(512, 512);
    Environment::SetParameters(presets[toggle].pop_size, presets[toggle].num_species, save, random);
    Floor::Initialize();
}

void Simulation::Update()
{
    //Input
    if(GetAsyncKeyState(VK_LEFT) < 0)
        count -= 1.5f;
    if(GetAsyncKeyState(VK_RIGHT) < 0)
        count += 1.5f;

    if(GetAsyncKeyState(VK_UP) < 0 && closeness > 0.1)
        closeness -= 0.01f;
    if(GetAsyncKeyState(VK_DOWN) < 0 && closeness < 1.2)
        closeness += 0.01f;

    //Toggle evolution
    static bool prev_space = false;
    if(GetAsyncKeyState(VK_SPACE) < 0 && !prev_space)
        update_toggle = !update_toggle;
    prev_space = GetAsyncKeyState(VK_SPACE) < 0;

    //Toggle evolution
    static bool prev_shade = false;
    if(GetAsyncKeyState('S') < 0 && !prev_shade)
        draw_shadow = !draw_shadow;
    prev_shade = GetAsyncKeyState('S') < 0;

    if(Environment::NumGenerations() < 500 && update_toggle)
        Environment::Update(fbo);
    else if(draw_shadow)
        Environment::RenderShadow(fbo);
}

void Simulation::Toggle()
{
    toggle = (toggle + 1) % NUM_PRESETS;
    Environment::SetParameters(presets[toggle].pop_size, presets[toggle].num_species, save_m, random_m);
    update_toggle = false;
}

void Simulation::Draw()
{
    if(GetAsyncKeyState('O') < 0)
        DrawOrtho();
    else
DrawPersp();
}

void DrawOrtho()
{
    Window::SetProjOrho();
    glRotatef(90, 1.0f, 0.0f, 0.0f);
    glTranslatef(0.0f, -1000.0f, 0.0f);
    Environment::Draw();
    Floor::DrawColored();
}

void DrawPersp()
{
    Window::SetProjPers();
    glTranslatef(0.0f, -50.0f * closeness, -150.0f * closeness);
    glRotatef(count, 0.0f, 1.0f, 0.0f);
    glTranslatef(-floor_halfsize, 0.0f, floor_halfsize);
    Environment::Draw();

    if(draw_shadow && !update_toggle)
        Floor::DrawTextured(fbo->GetTextureId());
    else
        Floor::DrawColored();
}

window.h

#ifndef WINDOW_H
#define WINDOW_H

#include <Windows.h>
#include <GL/glew.h>
#include <SDL.h>

namespace Window
{
    bool Initialize(wchar_t *title, unsigned int width, unsigned int height);
    void Finalize();

    void SetProjOrho();
    void SetProjPers();
    bool IsOrtho();

    void BeginDrawing();
    void EndDrawing();
};
#endif

window.cpp

#include "window.h"
#include "floor.h"

static SDL_Surface *Surf_Display;
static unsigned int scrn_width = 0;
static unsigned int scrn_height = 0;
static GLuint tex = 0;

bool Window::Initialize(wchar_t *title, unsigned int width, unsigned int height)
{
    scrn_width = width;
    scrn_height = height;

    if(SDL_Init(SDL_INIT_EVERYTHING) < 0)
        return false;

    SDL_GL_SetAttribute(SDL_GL_RED_SIZE, 8);
    SDL_GL_SetAttribute(SDL_GL_GREEN_SIZE, 8);
    SDL_GL_SetAttribute(SDL_GL_BLUE_SIZE, 8);
    SDL_GL_SetAttribute(SDL_GL_ALPHA_SIZE, 8);
    SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 16);
    SDL_GL_SetAttribute(SDL_GL_BUFFER_SIZE, 32);
    SDL_GL_SetAttribute(SDL_GL_ACCUM_RED_SIZE, 8);
    SDL_GL_SetAttribute(SDL_GL_ACCUM_GREEN_SIZE, 8);
    SDL_GL_SetAttribute(SDL_GL_ACCUM_BLUE_SIZE, 8);
    SDL_GL_SetAttribute(SDL_GL_ACCUM_ALPHA_SIZE, 8);

    Surf_Display = SDL_SetVideoMode(width, height, 32, SDL_HWSURFACE | SDL_OPENGL);
    //Surf_Display = SDL_SetVideoMode(width, height, 32, SDL_HWSURFACE | SDL_GL_DOUBLEBUFFER | SDL_OPENGL);
    if(Surf_Display == NULL)
        return false;

    //Glew stuff
    GLenum err = glewInit();
    //if (GLEW_OK != err)
    //{ // failed to initialize GLEW!
    //}
    //}

    //OpenGL stuff
    glClearColor(1, 1, 1, 0);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0f, 4.0f/3.0f, 1.0f, 4000.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    //glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL);
    GLfloat light_ambient[] = { 0.3f, 0.3f, 0.3f, 1.0f };
    GLfloat light_diffuse[] = { 0.7f, 0.7f, 0.7f, 1.0f };
    GLfloat light_position[] = { 1.0f, 1.0f, 1.0f, 0.0f };
    //GLfloat light_position[] = { 0.0f, 0.0f, 1.0f, 0.0f };
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
return true;
}

void Window::Finalize()
{
    SDL_FreeSurface(Surf_Display);
    SDL_Quit();
}

static bool projection_toggle = false;
void Window::SetProjOrtho()
{
    if(!projection_toggle)
    {
        projection_toggle = true;
        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        glOrtho(0, floor_halfsize*2, 0, floor_halfsize*2, 0.1f, 1000.1f);
        glMatrixMode(GL_MODELVIEW);
        //glDisable(GL_LIGHTING);
        glViewport(0, 0, 512, 512);
    }
}

bool Window::IsOrtho()
{ return projection_toggle; }

void Window::SetProjPers()
{
    if(projection_toggle)
    {
        projection_toggle = false;
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        //glEnable(GL_LIGHTING);
        glViewport(0, 0, scrn_width, scrn_height);
    }
}

void Window::BeginDrawing()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void Window::EndDrawing()
{
    glFlush();
    SDL_GL_SwapBuffers();
}
Appendix C - Plant viewer source code

common.h

#ifndef COMMON_H
#define COMMON_H

const float PI = 3.14159265f;

#endif

fbo.h

#ifndef FBO_H
#define FBO_H

#include <cassert>
#include "window.h"

class FBO
{
private:
    GLuint fbo_id;
    GLuint depthbuffer;
    GLuint img;
    unsigned int width, height;

public:
    FBO(unsigned int width, unsigned int height) : width(width), height(height)
    {
        glGenFramebuffersEXT(1, &fbo_id);
        SetActive();

        //Create a depthbuffer
        glGenRenderbuffersEXT(1, &depthbuffer);
        glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depthbuffer);
        glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT, width, height);
        glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT, GL_RENDERBUFFER_EXT, depthbuffer);

        //Create a colorbuffer
        glGenTextures(1, &img);
        glBindTexture(GL_TEXTURE_2D, img);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, img, 0);

        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

        GLenum status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
        if(status != GL_FRAMEBUFFER_COMPLETE_EXT)
            assert(!"Framebuffer couldn't be created");

        SetToDefault();
    }
};
~FBO()
{
    glDeleteFramebuffersEXT(1, &fbo_id);
    glDeleteRenderbuffersEXT(1, &depthbuffer);
}

inline void SetActive()
{
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo_id);
    glPushAttrib(GL_VIEWPORT_BIT);
    glViewport(0,0,width, height);
}

static inline void SetToDefault()
{
    glPopAttrib();
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
}

void GetTextureData(void* buffer)
{
    glGetTexImage(
        GL_TEXTURE_2D,  //Target
        0,              //Level
        GL_RGB,         //
        GL_UNSIGNED_BYTE,   //
        buffer
    );
}

GLuint GetTextureId()
    { return img; }
};

#endif

floor.h

#ifndef FLOOR_H
#define FLOOR_H

#include "window.h"

const float floor_halfsize = 50.0f;

namespace Floor
{
    void Initialize();
    void DrawColored();
    void DrawTextured(GLuint tex_id);
};

#endif

floor.cpp

#include "floor.h"
#include "window.h"

static GLuint floor_id;

59
void Floor::Initialize()
{
    floor_id = glGenLists(1);
    glNewList(floor_id, GL_COMPILE);
    glBegin(GL_QUADS);
    glNormal3f(0.0f, 1.0f, 0.0f);
    /*glTexCoord2f(0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f(512.0f, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(512.0f, 0.0f, -512.0f);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f(0.0f, 0.0f, -512.0f);*/
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f(floor_halfsize*2, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(floor_halfsize*2, 0.0f, -floor_halfsize*2);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f(0.0f, 0.0f, -floor_halfsize*2);
    glEnd();
    glEndList();
}

void Floor::DrawColored()
{
    glDisable(GL_TEXTURE_2D);
    glColor3ub(255, 255, 0);
    glCallList(floor_id);
}

void Floor::DrawTextured(GLuint tex_id)
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, tex_id);
    glColor3ub(255, 255, 255);
    glCallList(floor_id);
    glDisable(GL_TEXTURE_2D);
}

lsystem.h

#ifndef LSYSTEM_H
#define LSYSTEM_H

#include <string>
#include "simulation.h"

class Lsystem
{
friend void Simulation::Initialize(char* dir);
friend void SaveGeneration();

private:
int num_active_alphabet;
std::string substitutions_prim;
std::string substitutions[26];
std::string result;

public:
Lsystem();

void Reset()
{ result = "f"; }

void HardReset();
void Update(int times = 1);

//Accessors ------------------------------
std::string GetResult();
void SetResult(std::string r)
{ result = r; }

std::string GetRule(char l)
{
  if(l == 'f')
    return substitutions_prim;
  else
    return substitutions[l - 'A'];
}

void SetRule(char symbol, std::string substitute);
void SetRuleSize(int num);

//Mutations ------------------------------
void Permutate();
void Mutate();
void Insertion();
void Deletion();
void OnePointCrossover(const Lsystem& parent);
void SubTreeCrossover(const Lsystem& parent);
void AddBranch();
void DeleteBranch();
void AddRule();
void DeleteRule();
};

#endif

lsystem.cpp

#include "lsystem.h"
#include <cmath>
#include <vector>

static const char symbol_alph[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
static const char random_symbol[35] = "fl<>\+/+-ABCDEFGHIJKLMNOPQRSTUVWXYZ";

// CONSTRUCTOR ------------------------------
Lsystem::Lsystem():
  num_active_alphabet(0)
```cpp
{
    Reset();
    substitutions_prim = "f";
    for(int i=0; i<26; i++)
        substitutions[i] = symbol_alph[i];
}

void Lsystem::HardReset()
{
    Reset();
    substitutions_prim = "f";
    for(int i=0; i<26; i++)
        substitutions[i] = symbol_alph[i];
}

// STUFF ------------------------------------------

std::string Lsystem::GetResult()
{ return result; }

void Lsystem::Update(int times)
{
    std::string new_string;
    for(int i=0; i<times; i++)
    {
        for(unsigned int j = 0; j < result.length(); j++)
        {
            if(result[j] == 'f')
            {
                new_string += substitutions_prim;
            }
            else if(result[j] >= 'A' && result[j] - 'A' < num_active_alphabet)
            {
                new_string += substitutions[result[j] - 'A'];
            }
            else
            {
                new_string += result[j];
            }
        }
        result = new_string;
        new_string.clear();
    }
}

void Lsystem::SetRule(char symbol, std::string substitute)
{
    if(symbol == 'f')
    {
        substitutions_prim = substitute;
    }
    else
    {
        int index = symbol - 'A';
        if(index >= 0 && index < 26)
        {
            substitutions[index] = substitute;
        }
    }
}

void Lsystem::SetRuleSize(int num)
{
    for(int i = num_active_alphabet; i < num && i < 26; i++)
        substitutions[i] = symbol_alph[i];
    num_active_alphabet = num;
}

// MUTATIONS --------------------------------------
```
void Lsystem::Permutate()
{
    int index = rand() % (num_active_alphabet+1);
    std::string* subs_ptr = 0;
    if(index == 0)
        subs_ptr = &substitutions_prim;
    else
        subs_ptr = &substitutions[index-1];

    if(subs_ptr->size() < 2)
        return;

    int letter = rand() % (subs_ptr->size()-1);
    if((*subs_ptr)[letter] != '[' && (*subs_ptr)[letter] != ']' &&
    {
        char temp = (*subs_ptr)[letter];
        (*subs_ptr)[letter] = (*subs_ptr)[letter+1];
        (*subs_ptr)[letter+1] = temp;
    }
}

void Lsystem::Mutate()
{
    int index = rand() % (num_active_alphabet+1);
    std::string* subs_ptr = 0;
    if(index == 0)
        subs_ptr = &substitutions_prim;
    else
        subs_ptr = &substitutions[index-1];

    if(subs_ptr->size() <= 0)
        return;

    int letter = rand() % (subs_ptr->size());
    if((*subs_ptr)[letter] != '[' && (*subs_ptr)[letter] != ']')
        (*subs_ptr)[letter] = random_symbol[rand() % (8+num_active_alphabet)];
}

void Lsystem::Insertion()
{
    int index = rand() % (num_active_alphabet + 1);  
    std::string* subs_ptr = 0;

    if(index == 0)
        subs_ptr = &substitutions_prim;
    else
        subs_ptr = &substitutions[index-1];

    if(subs_ptr->size() == 0)
        *subs_ptr += random_symbol[rand() % (8+num_active_alphabet)];
    else
    {
        int letter = rand() % (subs_ptr->size() + 1);
        std::string temp = (*subs_ptr).substr(0, letter);
        temp += random_symbol[rand() % (8+num_active_alphabet)];
        if(subs_ptr->size() - letter > 0)
temp += (*subs_ptr).substr(letter, subs_ptr->size() - letter);

*subs_ptr = temp;
}
}

void Lsystem::Deletion()
{
    int index = rand() % (num_active_alphabet + 1);
    std::string* subs_ptr = 0;
    if(index == 0)
        subs_ptr = &substitutions_prim;
    else
        subs_ptr = &substitutions[index-1];
    if(subs_ptr->size() <= 1)
        *subs_ptr = "";
    else
    {
        int letter = rand() % (subs_ptr->size());
        if((*subs_ptr)[letter] != '[' && (*subs_ptr)[letter] != '']
        {
            std::string temp = (*subs_ptr).substr(0, letter);
            letter++;
            if(subs_ptr->size() - letter > 0)
            temp += (*subs_ptr).substr(letter, subs_ptr->size() - letter);
            *subs_ptr = temp;
        }
    }
}

void Lsystem::OnePointCrossover(const Lsystem& parent)
{
    int max_num_rules = std::min(num_active_alphabet, parent.num_active_alphabet);
    int random_rule = rand() % (max_num_rules + 1);
    if(random_rule == 0)
        substitutions_prim = parent.substitutions_prim;
    else
        substitutions[random_rule] = parent.substitutions[random_rule-1];
}

inline int SelectRandomBegining(std::string rule)
{
    std::vector<int> beginings;
    for(int i=0; i<rule.size(); i++)
        if(rule[i] == '[')
            beginings.push_back(i);
    if(beginings.size() == 0)
        return -1;
    else
        return beginings[rand()%beginings.size()];
}

inline std::string GetBranch(std::string rule)
{
    int beginning = SelectRandomBegining(rule);
    if(begining == -1)
        return "";
}
# C++ Code Snippet

```cpp
int bracket_cntr = 0;
for(unsigned int i = begining; i < rule.size(); i++)
{
    if(rule[i] == '[')
        bracket_cntr++;
    else if(rule[i] == ']
        {
            if(bracket_cntr > 0)
                bracket_cntr--;
            else
                return rule.substr(begining, i-begining);
        }
}
return "";
}

void Lsystem::SubTreeCrossover(const Lsystem& parent)
{
    int rule_index = rand() % (num_active_alphabet + 1);
    std::string rule = rule_index == 0 ? substitutions_prim : substitutions[rule_index-1];

    //Add the beginning
    int begining = SelectRandomBegining(rule);
    if(begining == -1)
        return;
    std::string result = rule.substr(0, begining);

    //Add the middle
    int other_rule_index = rand() % (num_active_alphabet + 1);
    std::string other_rule = other_rule_index == 0 ? parent.substitutions_prim : parent.substitutions[other_rule_index-1];
    result += GetBranch(other_rule);

    //Add the end
    int bracket_cntr = 0;
    for(unsigned int i = begining; i < rule.size(); i++)
    {
        if(rule[i] == '[
            bracket_cntr++;
        else if(rule[i] == ']
            {
                if(bracket_cntr > 0)
                    bracket_cntr--;
                else
                    result += rule.substr(i, std::string::npos);
                    break;
            }
    }
}

void Lsystem::AddBranch()
{
    int index = rand() % (num_active_alphabet + 1);
    std::string* subs_ptr = 0;
    if(index == 0)
        subs_ptr = &substitutions_prim;
```
else
    subs_ptr = &substitutions[index-1];

if(subs_ptr->size() == 0)
    *subs_ptr += "[]";
else
    {
        int letter = rand() % (subs_ptr->size() + 1);
        std::string temp = (*subs_ptr).substr(0, letter);
        temp += "[]";
        if(subs_ptr->size() - letter > 0)
            temp += (*subs_ptr).substr(letter, subs_ptr->size() - letter);
        *subs_ptr = temp;
    }
}

void Lsystem::DeleteBranch()
{
    int rule_index = rand() % (num_active_alphabet + 1);
    std::string rule = rule_index == 0 ? substitutions_prim : substitutions[rule_index-1];
    //Add the begining
    int beginning = SelectRandomBegining(rule);
    if(begining == -1)
        return;
    std::string result = rule.substr(0, beginning);
    //Add the end
    int bracket_cntr = 0;
    for(unsigned int i = beginning; i < rule.size(); i++)
        {
            if(rule[i] == '[')
                bracket_cntr++;
            else if(rule[i] == ']')
                {
                    if(bracket_cntr > 0)
                        bracket_cntr--;
                    else
                        {
                            result += rule.substr(i, std::string::npos);
                            break;
                        }
                }
        }
}

void Lsystem::AddRule()
{
    if(num_active_alphabet < 26)
    {
        substitutions[num_active_alphabet] = symbol_alph[num_active_alphabet];
        num_active_alphabet ++;
    }
}

void Lsystem::DeleteRule()
if(num_active_alphabet > 0)
{
    num_active_alphabet--;

    if(result.find(symbol_alph[num_active_alphabet]) != std::string::npos)
    {
        std::string temp = "";
        for(unsigned int j=0; j < result.size(); j++)
        {
            if(result.at(j) != symbol_alph[num_active_alphabet])
                temp += result.at(j);
        }
        result = temp;
    }

    //MUST remove letters in substitutions
    for(int i=0; i < num_active_alphabet; i++)
    {
        if(substitutions[i].find(symbol_alph[num_active_alphabet]) ==
            std::string::npos)
            continue;

        std::string temp = "";
        for(unsigned int j=0; j < substitutions[i].size(); j++)
        {
            if(substitutions[i].at(j) != symbol_alph[num_active_alphabet])
                temp += substitutions[i].at(j);
        }
        substitutions[i] = temp;
    }
}

main.cpp

#include "window.h"
#include "simulation.h"

int main(int argc, char* argv[])
{
    if(argc <= 1)
        return 0;

    if(Window::Initialize(L"PlantModel Evolution", 1024, 800) == false)
        return 1;

    Simulation::Initialize(argv[1]);

    SDL_Event Event;
    bool running = true;
    while(running)
    {
        //Input ---------------------
        while(SDL_PollEvent(&Event))
        {
            if(Event.type == SDL_QUIT)
                running = false;
        }

        if(GetAsyncKeyState(VK_ESCAPE) < 0)
            break;

    }
Simulation::Update();

static bool prev_toggle = false;
if(GetAsyncKeyState('t') < 0 && !prev_toggle)
    Simulation::Toggle();
prev_toggle = GetAsyncKeyState('t') < 0;

//Render ---------------------
    Window::BeginDrawing();
    Simulation::Draw();
    Window::EndDrawing();

//FPS -----------------------
static const UINT32 FPS_LIMIT = 1000 / 60;
static UINT32 timer = 0;
static UINT32 delta_time;
delta_time = SDL_GetTicks() - timer;
if(delta_time < FPS_LIMIT)
    SDL_Delay(FPS_LIMIT - delta_time);
timer = SDL_GetTicks();
}

Window::Finalize();
return 0;
}

plant.h

#ifndef PLANT_H
#define PLANT_H

#include <string>
#include "lsystem.h"
#include "plantmodel.h"

struct Point
{ float x, y; };

struct Plant
{
    Lsystem l_sys;
    //PlantModel p_model;
    Point pos;
};

#endif

plantmodel.h

#ifndef PLANTMODEL_H
#define PLANTMODEL_H

#include <string>

class PlantModel
{
private:
    unsigned int display_list_id;
public:
PlantModel();
PlantModel(std::string data, float position_x, float position_y);
void DrawBranches();
void DrawLeaves();
unsigned int Generate(std::string data, float position_x, float position_y);
};
#endif

plantmodel.cpp

#include "plantmodel.h"
#include "common.h"
#include <cmath>
#include <vector>
#include <stack>
//#include <Windows.h>
//#include <GL\GL.h>
//#include <GL\GLU.h>
#include "GL\glew.h"
#include <Windows.h>

static const float SCALAR = 2.0f;
static const float BRANCH_LENGTH = 1.0f * SCALAR;
static const float BRANCH_WIDTH  = 0.25f * SCALAR;
static const float LEAF_LENGTH   = 2.5f * SCALAR;
static const float LEAF_WIDTH    = 1.0f * SCALAR;
static const int SIDES = 6;

static int leaf_list;
static int branch_list;
static int knot_list;
static bool initiated = false;

struct PlantModelVertex
{
    GLfloat vx, vy, vz;
    GLfloat nx, ny, nz;
};

static void Initiate();
static unsigned int CreateBranch(const std::string& data, GLint id, float position_x, float position_y);
static void CreateLeaves(const std::string& data, GLint id, float position_x, float position_y);

PlantModel::PlantModel() :
display_list_id(-1)
{}

PlantModel::PlantModel(std::string data, float position_x, float position_y) :
display_list_id(-1)
{
    Generate(data, position_x, position_y);
}

void PlantModel::DrawBranches()
{
    glCallList(display_list_id);
}
void PlantModel::DrawLeaves()
{
    glCallList(display_list_id + 1);
}

unsigned int PlantModel::Generate(std::string data, float position_x, float position_y)
{
    if(!initiated){ Initiate(); initiated = true; }

    if(display_list_id == -1)
    {
        display_list_id = glGenLists(2);
        if(display_list_id == GL_INVALID_VALUE || display_list_id ==
GL_INVALID_OPERATION)
        { /* delete plantmodel; return (PlantModel*)0xABADBABE; */ }
    }

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // Generate list for branches
    CreateLeaves(data, display_list_id+1, position_x, position_y);
    return CreateBranch(data, display_list_id, position_x, position_y);
}

static unsigned int CreateBranch(const std::string& data, GLint id, float position_x,
float position_y)
{
    // bool straight = true;
    double complexity = 0;
    unsigned int long_comp = 0;

    glNewList(id, GL_COMPILE_AND_EXECUTE);
    glPushMatrix();
    glTranslatef(position_x, 0.0f, position_y);
    static float matrix[16];
    for(unsigned int i=0; i<data.size(); i++)
    {
        switch(data[i])
        {
        case 'f':
            glCallList(branch_list); //Draw branch
            glTranslatef(0, BRANCH_LENGTH, 0); //Move forward
            glGetFloatv(GL_MODELVIEW_MATRIX, &matrix[0]);

            if(matrix[13] >= 0)
                complexity += 1.0 * pow(1.1, double(matrix[13]));
            else
                complexity += 2.0 * pow(1.1, double(-matrix[13]));
            break;

        case 'l':
            glGetFloatv(GL_MODELVIEW_MATRIX, &matrix[0]);
            if(matrix[13] >= 0)
                complexity += 3.0 * pow(1.1, double(matrix[13]));
            else
                complexity += 6.0 * pow(1.1, double(-matrix[13]));
            break;

        //Push and pop matrix
case '[':
    glPushMatrix();
    break;

case ']':
    glPopMatrix();
    // glCallList(knot_list);
    // straight = false;
    break;

// Rotation X
case '>':
    glRotatef(22.5f, 1, 0, 0);
    // straight = false;
    break;

case '<':
    glRotatef(-22.5f, 1, 0, 0);
    // straight = false;
    break;

// Rotation Y
case '\':
    glRotatef(-22.5f, 0, 1, 0);
    // straight = false;
    break;

case '/':
    glRotatef(22.5f, 0, 1, 0);
    // straight = false;
    break;

// Rotation Z
case '+':
    glRotatef(22.5f, 0, 0, 1);
    // straight = false;
    break;

    case '-':
    glRotatef(-22.5f, 0, 0, 1);
    // straight = false;
    break;

default:
    continue;
}

if(complexity >= 10000.0f && long_comp < UINT_MAX - static_cast<unsigned int>(complexity))
{
    long_comp += 10000;
    complexity = 10000.0f;
}
else
    long_comp = UINT_MAX;

glPopMatrix();
glEndList(); // Branches End
return long_comp + static_cast<unsigned int>(complexity);

static void CreateLeaves(const std::string& data, GLint id, float position_x, float position_y)
{
    // Generate list for leaves
    glNewList(id, GL_COMPILE);
glPushMatrix();
//glLoadIdentity();
glTranslatef(position_x, 0.0f, position_y);
for(unsigned int i=0; i<data.size(); i++)
{
    switch(data[i])
    {
    case 'f':
        glTranslatef(0, BRANCH_LENGTH, 0); //Move forward
        break;
    case 'l':
        glCallList(leaf_list); //Draw leaf
        break;
    //Push orientation and position
    case '[':
        glPushMatrix();
        break;
    case ']':
        glPopMatrix();
        break;
    //Rotation X
    case '>':
        glRotatef( 22.5f, 1, 0, 0);
        break;
    case '<':
        glRotatef(-22.5f, 1, 0, 0);
        break;
    //Rotation Y
    case '\':
        glRotatef(-22.5f, 0, 1, 0);
        break;
    case '/':
        glRotatef( 22.5f, 0, 1, 0);
        break;
    //Rotation Z
    case '+':
        glRotatef( 22.5f, 0, 0, 1);
        break;
    case '-':
        glRotatef(-22.5f, 0, 0, 1);
        break;
    default:
        continue;  
    }
}
glPopMatrix();
glEndList(); // End Leaves
}
static void Initiate()
{
    //Leaf
    leaf_list = glGenLists(1);
    glNewList(leaf_list, GL_COMPILE);
    glBegin(GL_TRIANGLE_FAN);
glNormal3f( 0.0f, 0.0f, 1.0f); // NORMAL
glVertex3f( LEAF_WIDTH/2, 1.0f * LEAF_LENGTH/3, 0.0f); //5
glVertex3f( LEAF_WIDTH/2, 2.0f * LEAF_LENGTH/3, 0.0f); //4
glVertex3f( 0.0f, 3.0f * LEAF_LENGTH/3, 0.0f); //3
glVertex3f(-LEAF_WIDTH/2, 2.0f * LEAF_LENGTH/3, 0.0f); //2
glVertex3f(-LEAF_WIDTH/2, 1.0f * LEAF_LENGTH/3, 0.0f); //1
glVertex3f( 0.0f, 0.0f * LEAF_LENGTH/3, 0.0f); //0
glEnd();

glBegin(GL_TRIANGLE_FAN);
glNormal3f( 0.0f, 0.0f, -1.0f); // NORMAL
glVertex3f( 0.0f, 0.0f * LEAF_LENGTH/3, 0.0f); //0
glVertex3f(-LEAF_WIDTH/2, 1.0f * LEAF_LENGTH/3, 0.0f); //1
glVertex3f(-LEAF_WIDTH/2, 2.0f * LEAF_LENGTH/3, 0.0f); //2
glVertex3f( 0.0f, 3.0f * LEAF_LENGTH/3, 0.0f); //3
glVertex3f( LEAF_WIDTH/2, 2.0f * LEAF_LENGTH/3, 0.0f); //4
glVertex3f( LEAF_WIDTH/2, 1.0f * LEAF_LENGTH/3, 0.0f); //5
gLend();

gLendList();

// Branch
branch_list = glGenLists(1);
glNewList(branch_list, GL_COMPILE);
glBegin(GL_QUAD_STRIP);
for(int i=0; i < SIDES+1; i++)
{
  glNormal3f(float(sin(i/6.0 * 2*PI - PI/6)), 0.0f, float(cos(i/6.0 * 2*PI - PI/6))); // For flat shading
  glVertex3f(float(sin(i/6.0 * 2*PI)) * BRANCH_WIDTH, 0.2f, float(cos(i/6.0 * 2*PI)) * BRANCH_WIDTH);
}
glEnd();
gLendList();

// Knot
knot_list = glGenLists(1);
glNewList(knot_list, GL_COMPILE);

gLbegin(GL_TRIANGLES);
for(int i=0; i < SIDES+1; i++)
{
  glNormal3f(0.0f,0.0f,0.0f);
  glVertex3f(float(sin(i/6.0 * 2*PI)) * BRANCH_WIDTH / 2,
             0.2f,
             float(cos(i/6.0 * 2*PI)) * BRANCH_WIDTH / 2);
}
gEnd();

gLbegin(GL_QUAD_STRIP);
for(int i=0; i < SIDES+1; i++)
{

73
//glNormal3f(float(sin(i/6.0 * 2*PI - PI/6)), 0.0f, float(cos(i/6.0 * 2*PI - PI/6))); // For flat shading
//glNormal3f(float(sin(i/6.0 * 2*PI)), 0.0f, float(cos(i/6.0 * 2*PI))); // For smooth shading

glVertex3f(float(sin(i/6.0 * 2*PI)) * BRANCH_WIDTH / 2, 0.2f, float(cos(i/6.0 * 2*PI)) * BRANCH_WIDTH / 2);
glVertex3f(float(sin(i/6.0 * 2*PI)) * BRANCH_WIDTH, 0.0f, float(cos(i/6.0 * 2*PI)) * BRANCH_WIDTH);
}
gEnd();
gEndList();
}

point.h

#ifndef POINT_H
#define POINT_H

template<typename T>
struct Point
{ T x, y; }

typedef Point<int> Pointi;
typedef Point<float> Pointf;
#endif

simulation.h

#ifndef SIMULATION_H
#define SIMULATION_H

namespace Simulation
{
    void Initialize(char* dir);
    void Update();
    void Toggle();
    void Draw();
};
#endif

simulation.cpp

#include "simulation.h"
#include "window.h"
#include "floor.h"
#include "fbo.h"
#include <ctime>
#include <fstream>
#include <sstream>
#include "plant.h"

static struct
    int num_species;
    int pop_size;
}presets[3] =
static int toggle = 0;
static FBO *fbo;

static float count = 0.0f;
static float closeness = 1.0f;
static bool update_toggle = false;

static PlantModel *plant_models;
static unsigned int pop_size;

void DrawOrtho();
void DrawPersp();

void RenderShadow(FBO *fbo)
{
    glPushMatrix();
    fbo->SetActive();
    Window::SetProjOrtho();
    Window::BeginDrawing();
    glRotatef(90, 1.0f, 0.0f, 0.0f);
    glTranslatef(0.0f, -1000.0f, 0.0f);
    glColor3ub(0, 0, 0);
    glPushMatrix();
    glTranslatef(floor_halfsize, 0.0f, -floor_halfsize);
    plant_models->DrawLeaves();
    plant_models->DrawBranches();
    glPopMatrix();
    Floor::DrawColored();
    glFinish();
    
    //Set back to default fbo
    fbo->SetToDefault();
    Window::SetProjPers();
    glPopMatrix();
}

void Simulation::Initialize(char* dir)
{
    srand(unsigned int(time(0)));
    //srand(1337);
    std::ifstream in(dir);
    if(in.is_open())
    {
        Plant *plant;
        plant = new Plant();
        plant_models = new PlantModel();
        in.read(reinterpret_cast<char*>(&plant->pos), sizeof(Point));
        in.read(reinterpret_cast<char*>(&plant->l_sys.num_active_alphabet),
                sizeof(int));
// in>>(plant[i].l_sys.substitutions_prim);
std::getline(in, plant->l_sys.substitutions_prim);

for(int j = 0; j < plant->l_sys.num_active_alphabet; j++)
  std::getline(in, plant->l_sys.substitutions[j]);
// in>>(plant[i].l_sys.substitutions[j]);

plant->l_sys.Update(5);
plant_models->Generate(plant->l_sys.GetResult(), 0, 0);
// delete plant;

fbo = new FBO(512, 512);
Floor::Initialize();

void Simulation::Update()
{
  // Input
  if(GetAsyncKeyState(VK_LEFT) < 0)
    count -= 1.5f;
  if(GetAsyncKeyState(VK_RIGHT) < 0)
    count += 1.5f;
  if(GetAsyncKeyState(VK_UP) < 0 && closeness > 0.1)
    closeness -= 0.01f;
  if(GetAsyncKeyState(VK_DOWN) < 0 && closeness < 1.2)
    closeness += 0.01f;

  RenderShadow(fbo);
}

void Simulation::Toggle()
{
  toggle = (toggle + 1)%3;
  update_toggle = false;
}

void Simulation::Draw()
{
  if(GetAsyncKeyState('O') < 0)
    DrawOrtho();
  else
    DrawPersp();
}

void DrawOrtho()
{
  Window::SetProjOrho();
glRotatef(90, 1.0f, 0.0f, 0.0f);
glTranslatef(0.0f, -1000.0f, 0.0f);

  glPushMatrix();
glTranslatef(floor_halfsize, 0.0f, -floor_halfsize);

  glColor3ub(0, 255, 0);
  plant_models->DrawLeaves();

  glColor3ub(128, 64, 0);
  plant_models->DrawBranches();

  glPopMatrix();
  Floor::DrawColored();
}
void DrawPersp()
{
    Window::SetProjPers();
    glTranslatef(0.0f, -50.0f * closeness, -150.0f * closeness);
    glRotatef(count, 0.0f, 1.0f, 0.0f);
    
    glColor3ub(0, 255, 0);
    plant_models->DrawLeaves();
    
    glColor3ub(128, 64, 0);
    plant_models->DrawBranches();
    
    //Floor::DrawTextured(fbo->GetTextureId());
    glTranslatef(-floor_halfsize, 0.0f, floor_halfsize);
    Floor::DrawTextured(fbo->GetTextureId());
}

window.h

#ifndef WINDOW_H
#define WINDOW_H

#include <Windows.h>
#include "GL/glew.h"
#include <SDL.h>

namespace Window
{
    bool Initialize(wchar_t *title, unsigned int width, unsigned int height); 
    void Finalize();
    
    void SetProjOrho();
    void SetProjPers();
    bool IsOrtho();
    
    void BeginDrawing();
    void EndDrawing();
};
#endif

window.cpp

#include "window.h"
#include "floor.h"

static SDL_Surface *Surf_Display;

//Opengl32.lib
//glu32.lib

static unsigned int scrn_width = 0;
static unsigned int scrn_height = 0;
static GLuint tex = 0;

bool Window::Initialize(wchar_t *title, unsigned int width, unsigned int height)
{
    scrn_width = width;
scrn_height = height;

if(SDL_Init(SDL_INIT_EVERYTHING) < 0)
    return false;

SDL_GL_SetAttribute(SDL_GL_RED_SIZE, 8);
SDL_GL_SetAttribute(SDL_GL_GREEN_SIZE, 8);
SDL_GL_SetAttribute(SDL_GL_BLUE_SIZE, 8);
SDL_GL_SetAttribute(SDL_GL_ALPHA_SIZE, 8);
SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 16);
SDL_GL_SetAttribute(SDL_GL_BUFFER_SIZE, 32);
SDL_GL_SetAttribute(SDL_GL_ACCUM_RED_SIZE, 8);
SDL_GL_SetAttribute(SDL_GL_ACCUM_GREEN_SIZE, 8);
SDL_GL_SetAttribute(SDL_GL_ACCUM_BLUE_SIZE, 8);
SDL_GL_SetAttribute(SDL_GL_ACCUM_ALPHA_SIZE, 8);

SDL_GL_SetAttribute(SDL_GL_MULTISAMPLEBUFFERS, 1);
SDL_GL_SetAttribute(SDL_GL_MULTISAMPLESAMPLES, 2);
SDL_GL_SetAttribute(SDL_GL_SWAP_CONTROL, 1);

Surf_Display = SDL_SetVideoMode(width, height, 32, SDL_HWSURFACE | SDL_OPENGL);
if(Surf_Display == NULL)
    return false;

//Glew stuff
GLenum err = glewInit();
if (GLEW_OK != err)
    // failed to initialize GLEW!
    return false;

//OpenGL stuff
glClearColor(1, 1, 1, 0);
glViewport(0, 0, width, height);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0f, 4.0f/3.0f, 1.0f, 4000.0f);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_COLOR_MATERIAL);
GLfloat light_ambient[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat light_diffuse[] = { 0.7f, 0.7f, 0.7f, 1.0f };
GLfloat light_position[] = { 0.0f, 1000.0f, 0.0f, 1.0f };
//GLfloat light_position[] = { 0.0f, 0.0f, 1.0f, 0.0f };
gllightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
gllightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
gllightfv(GL_LIGHT0, GL_POSITION, light_position);

return true;
void Window::Finalize()
{
    SDL_FreeSurface(Surf_Display);
    SDL_Quit();
}

static bool projection_toggle = false;
void Window::SetProjOrtho()
{
    if(!projection_toggle)
    {
        projection_toggle = true;
        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        //glOrtho(-floor_halfsize-1, floor_halfsize, -floor_halfsize-1, floor_halfsize,
        0.1f, 1000.1f);
        glOrtho(0, floor_halfsize*2, 0, floor_halfsize*2, 0.1f, 1000.1f);
        glMatrixMode(GL_MODELVIEW);
        //glDisable(GL_LIGHTING);
        glViewport(0, 0, 512, 512);
    }
}

bool Window::IsOrtho()
{ return projection_toggle; }

void Window::SetProjPers()
{
    if(projection_toggle)
    {
        projection_toggle = false;
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        //glEnable(GL_LIGHTING);
        glViewport(0, 0, scrn_width, scrn_height);
    }
}

void Window::BeginDrawing()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void Window::EndDrawing()
{
    glFlush();
    SDL_GL_SwapBuffers();
}