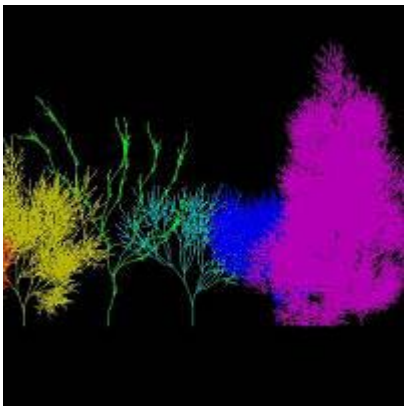

Music-based Procedural Generation of Plants

Jarrett Farnitano

Chris Jaeger

April 23rd, 2009

Introduction

This paper presents a model which generates plant imagery. However, unlike other methods which are primarily based on the simple process of creating a plant through procedural generation, our goal is to generate plants in accordance with the amplitude of sound. This growth forms a visualization that represents the flow of the music, in relatively real-time.

Related Work

This paper was inspired by a previous project created by Jarrett, which simply uses input from a music file to create a real-time image visualization.

The original plan was to work with fractals, much in the vein of the popular fern fractal. However, by having a mutable growth, one cannot guarantee that the small subset is representative of the whole shape, defeating the definition of a fractal.

We turned to procedural generation as an alternative, much in the same fashion as Prusinkiewicz et al [1, 2] and Smith [3]. Their papers were an inspiration for our method, rather than an actual source.

Motivation

We have covered a ton of reading on procedural model generation, but the scope of the class prohibits having normal class assignments of such magnitude. We wanted to explore this avenue, then, since we wouldn't really get the experience otherwise. With the musical backdrop, and the usage of the plant growth system, we are able to reduce the workload of the project to something which can be managed within the time allotted. We felt that as a project that touched upon concepts that we didn't get to reach normally, yet was small enough to achieve with true understanding of the material, it was a perfect project. That was probably our greatest motivation.

Another aspect of the project is the ability to watch it constructed real-time. Whereas traditional procedural modelling is done to hasten the process and allow interactivity, the implementation with music files is designed not only to serve as a backdrop for the drawing, but also to allow the method to run over the course of an observable time. A four minute song, for example, would show the growth of the plants over the course of four minutes.

Technique

Audio Capture

The initial conditions for the audio capture, was quick capture of the audio information, quick transformation to frequency domain, and capture of a microphone or master output of the computer. This allows the program to be standalone and not be concerned with current programs installed, such as other similar plug-ins. OpenAL was used to communicate the pulse code modulated data from the sound card to the program, and a fast Fourier transformation is next used to transform this data from the time spectrum to the frequency spectrum. After the FFT, an array of data containing amplitude information of the complete spectrum of frequencies is produced to be used by any function in the program.

OpenAL has several context management functions to access and output data. These functions provide a way to directly interact with the sound card. In this case the default device name is loaded, and then the device opened to capture data into a c++ vector. The following code illustrates finding the default input device and assigning a vector to capture the data.

```
// set up audio
const ALCchar *defaultInput = alcGetString(0,
    ALC_CAPTURE_DEFAULT_DEVICE_SPECIFIER);//find out name of default device
device = alcCaptureOpenDevice(defaultInput, 44100, AL_FORMAT_STEREO16,
    samples);//open default device.
```

To actually capture the data, the commands alcCaptureStart(device) and alcCaptureStop(device) are used. When start is called, PCM data is loaded into OpenAL internal ring buffer until stop is called. I programmed a very simple sleep function that insures enough data is in the buffer. However this may be improved by rendering the current scene while capturing the next audio samples and getting rid of the wasted time delay. The way OpenAL loads stereo data is interleaving. The interleaving data is immediately split up into the left and right channel to be compatible with the frequency spectrum transformation. The library FFTW was used for the Fast Fourier Transformations. It is one of the fastest libraries available for Fourier transformations, and was chosen for this property. Using this library, the following four lines of code produce the frequency spectrum arrays.

```
pleft = fftw_plan_r2r_1d(samples,  inl, outleft, FFTW_DHT, FFTW_ESTIMATE);
pright = fftw_plan_r2r_1d(samples,  inr, outright, FFTW_DHT, FFTW_ESTIMATE);
fftw_execute(pleft);
fftw_execute(pright);
```

The function fftw_plan_r2r_1d(), creates the plan for a real to real one dimensional transformation. These parameters were chosen, real to real, as well as the discrete Harley transformation which produces real results, preventing further calculation that would occur with finding the magnitude of

complex numbers. The function fftw_execute() actually performs the plan, and then the data found in outleft and outright can be used for any scene generation. Repeating this process several times a second allows for an animation to be produces based on the data coming straight of the sound card.

Alphabet

The Alphabet is a set of symbols used by both the Graphic Display and the Plant Grammar. However, both use the Alphabet in different methods, and so the meanings of each symbol will be defined within those sections. The Alphabet consists of the following eight symbols: X, Y, F, E, +, -, [,].

Plant Grammar

In our method, we define the Plant Grammar as the set of rules which regulate the growth of a plant. In the Plant Grammar, a string in the Alphabet describes the appearance of a plant. This string is later fed through the Graphic Display in order to create the visual image of the plant. The string is modified based on a number of production rules, in a similar fashion to an L-System.

However, unlike a traditional L-System, there exist multiple sets of production rules in the Plant Grammar. Each iteration of growth, the Plant Grammar first obtains the Audio Capture information, and selects from one of five sets of production rules. These five sets were arbitrarily assigned to five equally sized fractions of the amplitude scale, with no real method. Then, it reads the string of a plant, and applies the selected set of production rules. X, Y, and F are variables which are transformed into substrings by the production rules. The production rules for X and Y are more complex to model entire plant segments, while F usually is only adding an E or F to simulate a plant segment growing longer. E, +, -, [, and] are constants, and are skipped over by the Plant Grammar. Once it finishes parsing through the string, it will send the string to the Graphic Display.

Graphic Display

In order to procedurally model growth, it is necessary to maintain information regarding the position and orientation of each and every segment, mutable or not, within the plant. We decided to use an implementation based on Turtle Graphics in order to satisfy this need. At each iteration which changes the drawing in OpenGL, a 'seed' is planted at the root coordinate of the plant. A direction Vec3f defines both the direction of growth, and the magnitude, and is initialized as a vertical amount of 10. The string corresponding to the plant is read, and an action is performed according to the following table.

X, Y	Perform no action. X and Y are variables strictly for the Plant Grammar.
F, E	Draw a line from the seed's current position to its position plus the direction vector. Move the seed to this new position. F and E differ only for the Plant Grammar.
+	Turn the direction vector 25 degrees left, using linear algebra.
-	Turn the direction vector 25 degrees right, using linear algebra.
[	Pushes the current seed position, colour, and direction into a stack. Randomize a new colour.
]	Resets the seed position, colour, and direction to the values at the top of the stack, and pop the stack.

Stacks in C++ do not support arrays. As a result, three separate stacks of Vec3f are maintained, one each for seed position, colour, and direction. The usage of stacks simulate the creation of new branches from

the main plant. Returning to the positions on a stack then grows from the previous pattern on the plant. In order to spice the simulation, we added a colour change algorithm which occurs at the stack step, designed to colour each branch.

Testing

OpenGL issues were observed on one machine, so all simulation and testing was done on the other machine, Jarrett's.

The Plant Grammar production rules were tested with pure string manipulation. We used a singular set of production rules, which would include all 8 possible symbols, and applied it to a single string in multiple iterations. The results were correct with what should have been produced.

Rather than test the Graphic Display alone, we immediately jumped into testing the communication between the Audio Capture, the Plant Grammar, and the Graphic Display at once. An extremely simple set of production rules was assigned to each amplitude benchmark, and colour was not yet implemented. We fed a short sound segment and observed the production with both a single bucket and with six. While the results were not exactly plants, we observed that the communication was working properly and the rules of the Graphic Display were performed mostly accurately.

Further testing was resolved to simple modifications of the production rules, and each time a new algorithm or concept was proposed (adding colour, refining the stack structure, restarting the plant growth after a set number of levels, etc). Since the program was functional at this point, testing simply consisted of running the program and seeing if the output was desirable.

Outstanding Issues

Once we got the communication working and the program itself basically finished, the next step was to turn it into the beautiful display of plants that we were looking for. On the way, we ran into a number of outstanding issues, either problems that were resolved or ideas that we decided not to implement. Note that amidst these issues, we also made some subtle changes to the production rules out of flavour concern, rather than any actual mechanical issue.

Ridiculously Poor Framerate

The first issue noted was a horrendous frame rate. At the start, with small strings, it was fast to update the simulation. However, once it hit a few iterations in, the simulation would slow drastically, almost to a complete halt. This occurs within the span of a couple of seconds, if even that long.

The first "real" set of production rules we wrote were somewhat long, and after only 6 iterations, would become strings of over 480 characters in length. This grows exponentially, and is a consequence of using procedural modelling in a fashion similar to an L-System. Those normally generate a product after a small number of iterations, but our method would take songs of incredible length and produce one iteration per second. We mitigated this problem by drastically shortening the production rules. The highest amplitude remained as a long string, in order to preserve the image of complexity. We also played around with increasing the time between iterations, and also only advancing one plant at a time rather than all six simultaneously.

The framerate remains "questionable", in that it may slow in perceiveable amounts near the end of each cycle. However, we found that this slowdown is negligible in the long term and left it in.

Left-sided Trees

Once we were finally able to properly observe the plants, we noticed that no matter how many right turns were suggested in the strings, the plants would always lean primarily to the left. This turned out to be because the direction of the plant was not pushed onto a stack. This was when we implemented the third stack. Upon doing so, our trees were properly upright.

Bland Symmetric Trees

Another problem we were having was all the trees were quite uniform. After debugging some of the code, it was realized that the rule selection currently was not very well distributed, especially in the plants that corresponded to high frequencies. For the most part only the first or last rule was being called during tree growth. This was due to fact that the intensities of the high frequencies were generally much lower than the values calculated for the low frequencies. This comes from the property that within the audio spectrum high frequency sounds are clearer at the same output power for human perception. The amplitudes of the higher frequencies were less than that of the low frequencies but perceived about the same. After adding several constants to apply a gain to the high frequency amplitudes, more rules started to be called. These were tweaked to find a good range where the amplitudes were distributing to all rules, and instantly the trees became much more interesting and intricate.

Tree Refreshing

Despite having this new reduced grammar, after 6 iterations, the refresh rate was pitifully slow, almost one second for the next level to be drawn. At this point it was obvious that waiting for this exponentially growing function was not going to work. Clearing the trees every once in a while solved this problem, but still a balanced needed to be found between detail and time. Creating the plants down to 5 levels and refreshing them, gave enough time for unique plants to be generated, output, and observed before moving onto the next completely new plant. This also was more in line with the visualizer idea, because it could be set to run and would just act only what was playing currently, constantly updating and adapting to new content like most visualizers.

Spinning Trees

Once we finally had some interesting growths, we observed that sometimes trees would rotate. Sometimes they would go as far as to grow downwards, completely off the perspective of the camera. We discovered that the source of this issue was that the starting string, which was simply X, allowed the plant to begin with turning commands. And with each subsequent iteration, the plant would rotate further. By changing the initial string to E[X], it always grew upwards.

The Angle Question

The turning command in the Graphic Display is fixed at 25 degrees. This is performed with pre-defined sine and cosine floats that are multiplied into the direction vector. The implementation was designed to save computation time rather than waste time calculating the trigonometry. A function, setAngle(theta)

was written with the intent that additional angles could be predefined, and the sine and cosine constants updated.

We originally implemented a randomized angle system that was somewhat influenced by the audio capture. It would select a different angle at each turn. When we tested it, however, the results were beyond chaotic. The reason for this was the same as the directional bias of earlier; the angles were not memorized so the plants would twitch with new angles at every single juncture. Saving the angle at every juncture would be much more complicated, in addition to requiring an additional stack. We agreed to abandon the implementation of different angles and remained with the 25 degree angle.

The Multithread Question

Since 6 trees were being computed simultaneously on the same audio data, it was originally thought that multithreading these computations and the drawing would provide an instant speed up. A significant amount of the time is spent in drawing loop. However after much research, it was discovered that OpenGL and multithreading are very difficult to work together. Due to the way the Windows implementation of OpenGL works, each active OpenGL window has a context that can be drawn into. This context is locked to be accessible by only one thread at any one time. In order to multithread the drawing, the context of the open window would need to be switched constantly to different threads and restored, which is a very slow operation. In addition, because the context is locked, the draw operations would need to be performed sequentially although in an interleaving format anyway. This is basically the same as sequentially drawing the plants, only with context switching in between, providing no real advantage to multithreading. The decision to use the current non-threaded design was then made with slight improvements to the draw loops.

Finalized Settings

All of the issues and implementations resulted in the following settings.

- 6 buckets listen to 6 separate frequencies, receiving an amplitude value that is a refined into a new value for rule selection.
- 5 different sets of production rules, from low amplitude to high amplitude.

Conclusion

This project was completely successful in linking music to something new like procedural generation of plants. This technique provides a new twist on procedural modeling, using a new type of input data to generate representative plants. This type of data input could be used for generating other types of procedural modeling, especially useful if the models are to be stored due to the time taken to draw the models. Very realistic looking plants were generated and the selection of rules only enhanced their appearance. The only concern that remains in a real-time situation like this is frame-rate but considering the aesthetic quality of the plants, the achieved rates are perfectly acceptable for the application.

Future Potential

While for our purposes, the program is completed, there are a lot of additional possibilities which can be added onto the program. Some of these, such as randomized angles, were considered and simply abandoned. Here are some of the other ideas that can be expanded upon.

Singular Plant Display

As mentioned, we changed our original intent from modelling a single, large plant path to refreshing the plant after it reaches a certain level of growth. However, the code is still perfectly capable of ignoring this refresh step, and simply producing complex plants. Implementing this would change the program from an interactive-time visualization to a single-time model.

Three Dimensional Plants

The alphabet currently only operates in a 2D plane. By adding new symbols, new rules for the Graphic Display can dictate a third dimension, allowing a full three dimensional plant. From this step, additional function such as shadows and lighting can easily be added as if the plant model was any standard model.

Work Assignment

JARRETT

- Refining the Audio Capture program
- Running the simulations on a better machine
- Refining rule selection create more interesting plants.

CHRIS

- Writing the Plant Grammar
- Writing the Graphic Display

BOTH

- Incorporating all of the elements into a single program
- Refining the production rules of the Plant Grammar

References

[1] Prusinkiewicz P., Hammel M., Mjolsness E. "Animation of Plant Development" August, 1993. Computer Graphics (Proceeding of SIGGRAPH 93)

[2] Prusinkiewicz P., Hammel M., Měch R. "The Artifical Life of Plants" August, 1995. *Artificial life for graphics, animation, and virtual reality* (Volume 7 of SIGGRAPH 95)

[3] Smith A. "Plants, Fractals, And Formal Languages" July, 1984. Computer Graphics (Proceeding of SIGGRAPH 84)

Screenshots

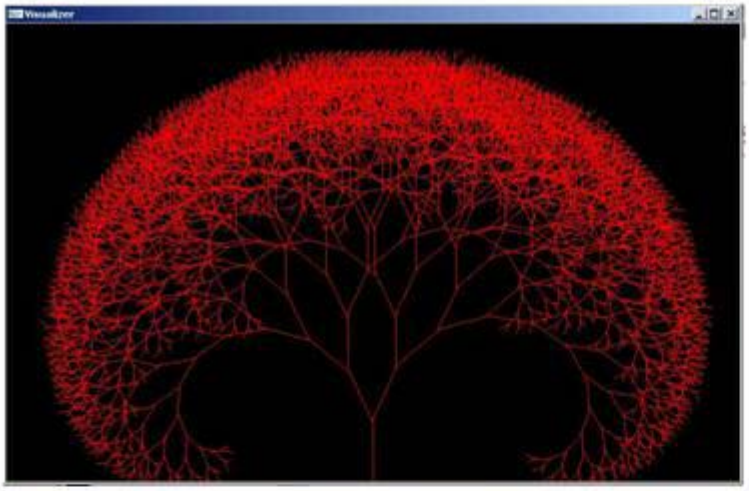

Fig 1. Single rule growth.

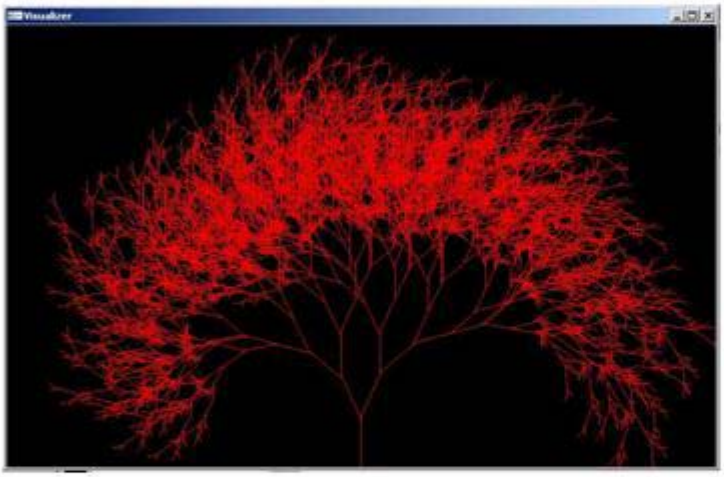

Fig 2. Two rule plant.

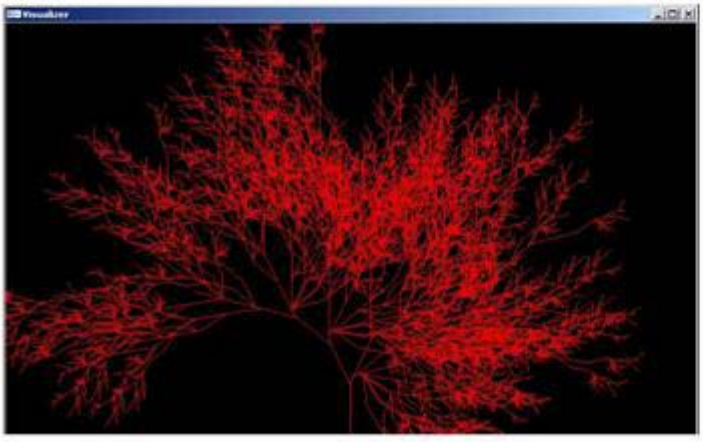

Fig 3. Some experimentation with rules.

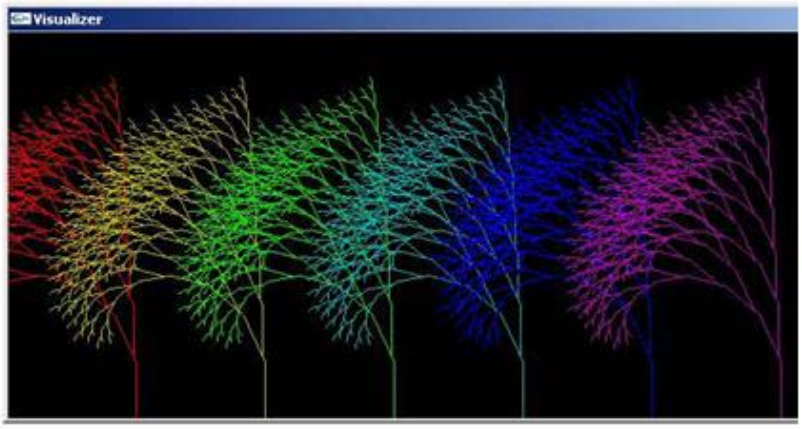

Fig 4. Left handed tree problem

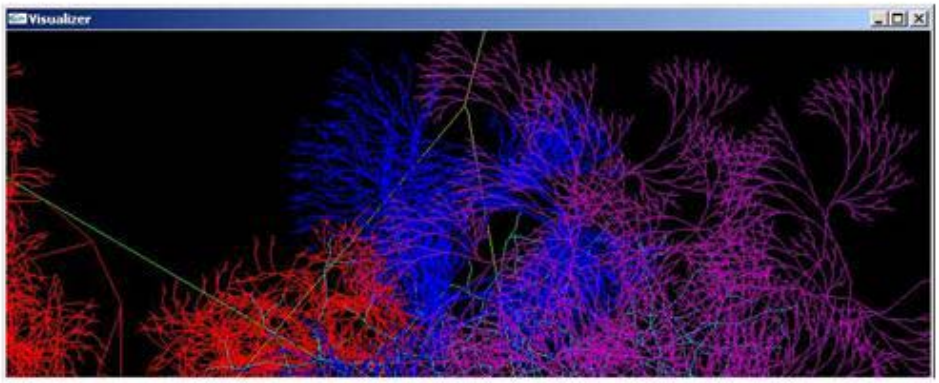

Fig 5. Grammer gone haywire.

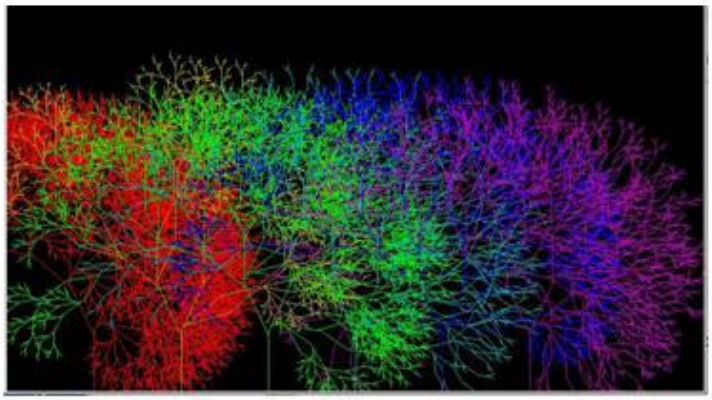

Fig 6. Some more experimentation with grammer

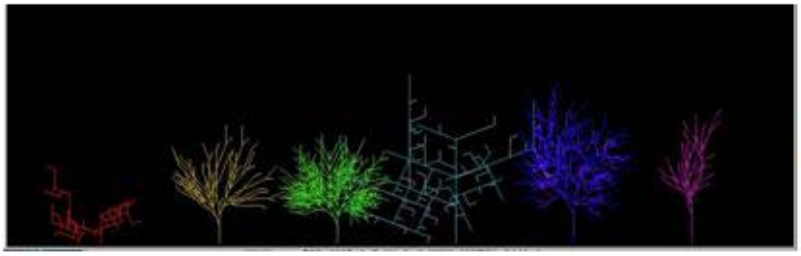

Fig 7. An illustration of the angle issues.

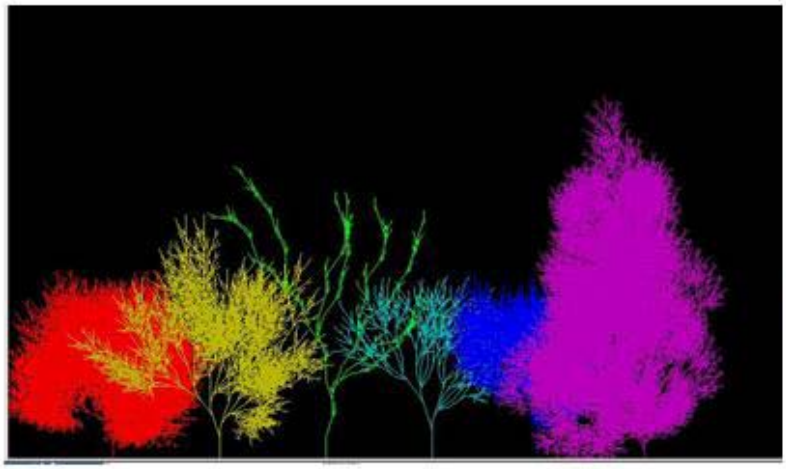

Fig 8. Final grammer, approximately 9 levels deep.

Fig 9. Final grammer, 5 levels deep.

Fig 10. Final Grammer, 5 levels deep.