

CS 211, Winter 2004  
Lab Assignment L5: Winamp Visualization  
Assigned: Monday, Mar. 8, 11:00 AM, Due: Monday, Mar. 15th,  
5:00 PM

Brian M.Dennis, Instructor  
Bin Lin, Tom Lechner, Rachel Goldsborough, Teaching Assistants

**Winamp Visualizations, 50 Points**

The short version of this assignment is to implement a Winamp plugin that does visualization. The plugin should demonstrate your ability to write C++ code that does the following:

- Write basic C++ code: expressions, conditionals, iterations.
- Extend a pre-existing class hierarchy using derived classes and virtual functions.
- Show initiative in learning how to use a large, complex, industry standard library to implement something.
- Design, format, and document your code with good style.

You are writing code that fits into a certain application programmer's interface (API). This API repeatedly gives your plugin a chunk of information representing an analytical snapshot of the audio data currently being played by Winamp. Most of the work involved in supporting this interface will be taken care of by C++ code that implements a class hierarchy to hide the grunge.

At the same time, your code will use a graphics imaging API to take the data and draw something interesting on the screen. Something interesting in this case is arbitrary. Code provided for this assignment, will demonstrate some effects and can be used as a starting point. However, you should feel free to exercise your creativity as you see fit.

This part of the assignment entails effort that's not directly related to programming. This includes:

- Getting and installing Winamp (be sure to get a 2.x version)
- Getting, understanding, and building the startup plugin provided.
- Installing that basic plugin where Winamp can find it

- Reading information about one of two Windows APIs, GDI or OpenGL, and understanding how they work
- Potentially making sure you have the API installed on your machine.
- Figuring out some effects you would like to try
- Applying that to the base code
- Debugging the whole kit 'n caboodle

There are two graphics APIs that should be supported on just about every version of Windows. One is the low level, builtin GDI. This is used to do basic 2D vector and bitmap graphics. The higher level OpenGL supports 3 dimensional graphics and a more sophisticated color model. The libraries to support these APIs should be available on most modern Windows machines. For this assignment we will be using the OpenGL API. However, the header files needed for development might not be on your machine. A search for `gl.h` can conform whether these headers are available. If they aren't check with BMD to see how to rectify the situation.

The two APIs are documented at:

<http://msdn.microsoft.com/library/default.asp> under **Graphics and Multimedia**.

A good place to start for OpenGL tutorials is <http://nehe.gamedev.net> along with

<http://www.opengl.org/>

A word to the wise, get your team together (if you have one) and get the base code working early. Once you have that foundation, then you can just continually tweak that code to implement various effects. Note that this part of the assignment doesn't really lend itself to straight ahead divvying up of tasks. There aren't real clear discrete chunks. You should expect to work together relatively closely with each other. Also, putting this off until the last minute is a definite recipe for disaster. This part of the assignment is of the "throw them in the deep end and see who swims" variety.

## Grading

Project teams can have one or two people. Teams are self forming and selecting for this part of the assignment. It's up to you to find a partner. Here are the varying requirements:

- One person. Implement one visualization.
- Two people. Implement two different visualizations. One should primarily use varying shapes, the other color manipulation.

Your code should fit into the class hierarchy in the startup code. The handout code has a sample extension class `sample`, which is a subclass of our special `opengl` class. `opengl` takes care of all the window management and provides a large number of member functions that abstract many OpenGL functions.

The two most obvious ways to differ in visualizations are in what data they use from Winamp. The waveform data represents the last 576 output samples sent to the audio channel. The spectrum data represents the

spectral analysis of the audio stream for 576 different frequencies. The data in both of these chunks is represented using 8 bits, leading to 256 different values. You can also vary widely in what exactly is drawn. Using different shapes can prove interesting. Also, different schemes for colors can provide a lot of interesting effects.

Grading breakdown for this part of the assignment:

- 20% Getting your plug-in to build and run within Winamp. This means being able to run for more than a minute without crashing. This also means simply extending the provided classes, even if you don't do something interesting.
- 50% Appropriately using the OpenGL API, as described above, to implement your classes.
- 30% Creativity factor. The more interesting your eye candy, the higher your score on this part. This is completely subjective and will be judged by BMD.

## Alternative Platforms

The major downside of using Winamp is that it only runs on Windows platforms. If you're using UNIX or Mac OS this can be a problem. We are open to you doing a music visualization in another music player such as iTunes on Mac OS or XMMS on free UNices. Unfortunately, we can't really provide start up code for these platforms, but if you're adventurous don't let us stop you.

## Cheezy Automata

If you're not up for a music visualization, then continue with extending the cheezo interpreter. Add the following to the interpreter:

- A `lambda` special form that is like Scheme's `lambda`, in that it returns a procedure object that captures the current environment. When the procedure is invoked, execution occurs in the captured environment.
- Add a `grid` datatype to the interpreter. The `grid` datatype represents a 1D cellular automata grid as in homework 2. Each entry of the grid holds a 0 or 1. A `grid` should support the following operations
  - `make-grid n`, creates a grid of size  $n$ , all positions set to 0
  - `get-grid g i`, returns the value at position  $i$  in grid  $g$
  - `set-grid! g i v`,  $v$  must equal 0 or 1, and position  $i$  should be set to  $v$  in grid  $g$ .
  - `transform-grid g p`,  $p$  should be a procedure of three arguments. `transform-grid` should return a new grid, where each element position  $i$  is the result of applying  $p$  to the values of positions  $i - 1, i, i + 1$  in  $g$ . The result of each call to  $p$  should be a 0 or 1. Closures should be perfectly usable as a  $p$  argument.
  - You should add a `print_form` member function for `grids`

Use your extended cheezo interpreter to reimplement the cellular automata of lab4. For example, you should be able to write a file of cheezo code similar to the following and run it through your interpreter:

```

(define runner
  (lambda (grid func cnt)
    (print (transform-grid grid func))
    (if (= cnt 0)
        0
        (runner (transform-grid grid func) (- cnt 1)))))

(define next-gen
  (let ((table (vector 0 1 0 1 1 0 1 0)))
    (lambda (l m r)
      (let ((idx (+ (* 4 l) (* 2 m) r)))
        (vector-ref table idx)))
    ))

(define grid (make-grid 9))
(set-grid! grid 5 1)
(runner grid next-gen 10)

```

Astute observers will note that the `grid` datatype bears a striking resemblance to `vectors` or even better large `bitsets`. Grading of this version of the project will be as follows:

- 50% for a working `lambda`
- 40% for a working `grid` datatype
- 10% for getting `cheezo` code similar to the above working and demonstrating with a few other 1D automata rules