Diminuendo is a program that produces chords and visuals by applying a sequence of transformations to an initial generator. Using Diminuendo, a composer can create a tonal structure for music simply by specifying a few transformations and a little information about how they are connected. By supplying graphics transformations as well, the composer has the unique opportunity to show the audience a spatial representation the musical flow. The transformations provided are arranged in a self-similar pattern governed by a transition matrix and then used to produce both audio and video. Diminuendo works best when paired with other instruments, especially a high-quality MIDI (Musical Instrument Digital Interface) synthesizer and/or a drum machine.

This program is written entirely in Haskell and makes use of the HOpenGL graphics library, the GLUT library for user interface, and the Euterpea library for music. It can theoretically be compiled on any machine with these libraries installed, but has only been tested for the Windows OS (specifically Windows 7). An important note that I will make right away here is that to compile with GHC, one must use the -threaded command. Diminuendo uses two threads, one for music and one for graphics, and HOpenGL requires that -threaded be used when compiling a threaded program with GHC.

There are five individual modules in addition to the Dimin.hs file: MatrixGen.hs, Display.hs, TransEx.hs, Music.hs, and Square.hs, I will outline in this paper the purpose for each and explain any notable functions. To help in understanding it, here is a brief description of the flow of the program, along with where each step takes place:

1. The user is prompted for an eigenvalue, which is used to determine a transition matrix. (MatrixGen)
2. The transition matrix is used to generate a semi-random list of transformations, which at this point are specified by the numbers 0, 1, 2, and 3. (MatrixGen)
3. The transformation numbers are translated to a list of graphical transformations (Display, TransEx) and a list of music transformations (Music)
4. Music plays (Music) and graphics appear (Display, TransEx, Square)

Now we’ll discuss the individual modules.

MatrixGen

The goal of MatrixGen is to get a suitable transition matrix with a largest eigenvalue approximately equal to a user input. A transition matrix is simply an nxn matrix of 1s and 0s that expresses allowable transformation transitions; ie, whether after making one transformation, a certain other transformation can immediately follow. In this matrix, the entry $a_{i,j}$ indicates whether transformation $j$ can immediately follow transformation $i$, where 1 means that this is allowable and 0 means it is not. In this program, we set an equal probability of moving to each allowed transformation in the case that the transition matrix has multiple allowed next steps.

eigFinder is the centerpiece function of MatrixGen. Supplied with an Int $n$ and Float $eig$, it returns the $n x n$ transition matrix which has a largest eigenvalue closest to $eig$. The reason for generating the transition matrix by eigenvalue is that the largest eigenvalue of a transition matrix is
closely related to the “connectedness” of the matrix, and in fractal geometry, the dimension of the fractal produced by applying certain graphical transformations according to the transition matrix. I shy away from describing the images produced by Diminuendo as fractals; although they are generated by the same kind of graphical iteration that can produce fractal images, fractals put more restraints on the types of graphical transformations that can be used, namely that there be a consistent scaling system involved. In this program, all the visuals shrink gradually over time, and the transformations are more based on rotation.

Nevertheless, supplying a largest eigenvalue is a great way to generate a transition matrix that is somewhat random but is about as dense as you would like. The specified eigenvalue should be between 1 and 4, where 1 will produce the 4x4 identity matrix and 4 will produce the 4x4 all-ones matrix.

The function toProbMatrix is also important: given the transition matrix, we now want to have a matrix of probabilities of moving from $i$ to $j$, not just whether it's allowed. This function gives us just that, by dividing each entry by the total sum of its row. It also wins my “Glad I'm Using Haskell” award for this module, since map, flip, and fold made it trivially easy to write.

Finally, tList creates a list of transformation numbers from a transition matrix. These are the numbers we will need to feed to the Music and Display threads.

Display

The function is Display may at first seem complicated but are in fact very simple. Understanding the types involved is crucial to getting a grasp of what is going on. The main type is called a Transformation, which is defined in TransEx as (String, Float, (Float, Float, Float)). This is interpreted loosely as (Type of Transformation, Amount, Co-ordinates). These three pieces of information are enough to define rotations, translations, where the transformation details work as follows:

Translation → (“Translate”, garbage (usually 0.0), (change in x, change in y, change in z)
Rotation → (“Rotate”, amount (in degrees), (0,0,1) if amount is positive; (0,0,-1) if negative)
       Note: all rotations are performed around the origin. Positive rotations are counter-clockwise, negative rotations are clockwise.
Flip → (“Flip”, garbage (usually 0.0), garbage (usually (0.0, 0.0, 0.0))

The reason that the vector in the “Rotate” transformation needs to be given is that the function that is in charge of “animating” the moving square is unaware about which direction to move the square without this extra information. We'll see why this problem comes up in a moment.

To make the program more versatile, a TransformationList is defined as a list of any number of Transformations. Then, one of the transformations dictated by our transition matrix might be (in English):

“Translate by (.1, .2), then rotate 90 degrees,”

which would be written as a TransformationList as

[(“Translate”, 0.0, (0.1, 0.2, 0.0), (“Rotate”, 90, (0.0, 0.0, 1.0)))]
This may be confusing, but just remember that most times when we are talking about transformations, we are actually referring to the type TransformationList.

That of course means that what we're finally going to be applying to the images is a list of TransformationLists, or a list of lists of Transformations. Mercifully, this is not given its own type, but is seen in the code as [TransformationList].

The short story of Display is that it translates the transformation numbers generated by MatrixGen into TransformationLists, then applies each transformation, drawing a square after each. But it is not quite so simple. To begin, it's important to understand that the display function is called 30 times per second, and every square is drawn every time it is called. The changes that occur in the program are dictated by three IORefs:

1. numB (or n), which gives the number of squares currently on display
2. twist (or t), which is a timer that runs from 0.0 to the length of the current chord being played
3. counter (or c) which is identical to twist except scaled between 0 and 1

The twist variable is incremented by a small amount every time display is called, and numB is incremented by 1 every time twist hits its limit and resets. The lengths of the chords (which are the lengths of time that each square should be at the forefront) are contained in the variable durList, which is produced by a function in the Music module, and these lengths dictate how large twist should grow before resetting and adding one to numB.

display then converts the transformation numbers to TransformationLists, and takes the first numB elements of this list to draw to the screen. drawList is a key function in this process. Given a list of TransformationLists, drawList will go through one at a time, perform the transformation, and draw a square. The square's size and brightness are proportional to the square root of (1 over the length of the list of remaining transformations). This way, squares that are near the front of the list (and have been drawn earlier) are smaller and darker than squares at the end. The square corresponding to the last TransformationList is full size, and is actually processed by a different function than the rest of the squares.

The function drawTransList is used to draw all but the last square, and it simply draws a square where the transformations dictate it should go. But drawTListGrad is used for the last square, and it provides a smooth animation. This function makes use of the counter value, which, recall, grows from 0 to 1 over the course of a chord. While \( c < 0.75 \), drawTListGrad acts as if the final transformation isn't there, but when \( 0.75 < c < 1.0 \), it draws the final transformation gradually, with the important values of the transformation being scaled proportional to \( c \). This gives the impression of a square smoothly moving from one transformation to the next.

When all of the transformations have been performed and displayed, (and numB is now greater than the length of the transformation list), display branches to a finishing animation. All the squares are shown, and they are rotated at ever-increasing speed for a second or two. The squares spin around faster and faster, then vanish, and the program is over.

There is one more problem that needs to be addressed. It deals with rotations and is the main purpose for TransEx.
When HOpenGL’s *rotate* function is called, it rotates the reference axes along with all graphics drawn after its call. Because of this, making a rotation transformation and the translation transformation will result in the translation taking place in a rotated direction. For example, consider a point at (0, 0) and the following two transformations:

1. Rotate the object 90 degrees
2. Translate the object 1 unit along the positive x-axis

If we just make the transformations like this, the point will move to (0,1) instead of (1,0) because the 90 degree rotation has altered the interpretations of future translations, effectively changing the layout of the x, y, and z-axes. Instead, *TransEx* contains functions that allow *Display* to do the following:

1. Rotate the object 90 degrees
   1.1 *de-rotate the scene 90 degrees*
2. Translate the object 1 unit along the positive x-axis
   2.1 *rotate the object 90 degrees*

In essence, the scene is “de-rotated” after every TransformationList is processed. Then, each future TransformationList has all the rotations from previous lists added in after all the translations have already taken place. All this parsing is done through the *fixRotations* function, which is called before any transforming or drawing. Additionally, the *insertRotation* function adds rotations to current rotations if they exist, so we don’t get a TransformationList with an excessive number of rotations tacked onto it, we get one with its existing rotation modified to account for previous rotations.

This is why it is required that a rotation Transformation use the co-ordinate field to specify whether the rotation is positive or negative; *fixRotations* can occasionally change the sign of a rotation, and the animating function would have no idea whether to rotate clockwise or counter-clockwise. It uses the z value given in the initial Transformation (which is left untouched by *fixRotations*) to determine in which direction to animate the rotation.

*TransEx* also contains the definitions for exactly which graphical transformations are to be used, along with some functions to specify the colors that accompany the transformations. These are variables that can and should be modified by the user to create a unique composition.

**Music**

Note: in this section we will need to use the concept of functional chords in a key. In *Diminuendo*, chords are referred to as one less than their traditional name, thus a tonic chord is a 0, and 2nd is a 1, and so on.

*Music* is much like *Display* in that it converts the transformation numbers into its own transformations which are applied to an initial generator. Here we use types parallel to Transformation and TransformationList which are called CTransform and CTransList. A CTransform is equivalent to (String, Int) which is interpreted as (Type of Transformation, Argument). The different types of CTransform are as follows:
Move → (“Move”, Amount)  
Moves the chord a given number of function places within a key.  
For example, (“Move”, 3) would send ii → V or vii → iii, etc.  
while (“Move”, -2) would send IV → ii or vi → IV, etc.

Smooth Move → (“sMove”, Amount)  
Identical to Move but applies certain heuristics to make the move between  
chords sound smoother.

Pivot → (“Pivot”, Place)  
Treat the current chord as a pivot chord and reinterpret it in a new key.  
For example, an F major chord given a (“Pivot”, 5) will be interpreted as a  
six chord (remember that the functional names are all shifted one), which  
will set the new key as A minor.

Switch Mode → (“Switch Mode”, garbage (usually 0))  
Change the mode from minor to major, or major to minor.

Switch Duration → (“Switch Dur”, Type)  
Change the relative duration of the notes in the chord to one of the preset patterns.

Scale Duration → (“Scale Dur”, Amount)  
Scale the duration of a chord.

Cadence → (“Cadence”, Type)  
Insert a four-five-one progression in the current key, with the lengths of the three chords  
determined by the preset cadence type.

All of these transformations operate on a music type called MChord. MChord is a fairly  
complicated type, but each piece of it serves an expressive purpose. The definition of MChord is

MChord = (Int, Rational, String, String, (Int, String, Int))  
= (Functional place, Duration, Note pattern type, Chord inversion, Key)

where Key is a tuple consisting of (Root, Mode (Maj or Min), Octave).

In addition to the MChords, it seems to help in visualization to have a bass note that strictly  
follows the path of the transpositions. A IV chord going to a iv chord does not always sound like it is  
ascending for example, especially if Smooth Move is used. Considering this, a bass line is added to the  
chords which always descends when the transformation is a negative move and ascends when the  
transformation is a positive move. The only cases when the bass line disobeys this rule is if following it  
would lead to an octave less than 2 or greater than 4, which would be inappropriate for a bass line. The  
bass line is formed from a new type BMusic, which is just a tuple indicating (note, duration, octave).

Most of the functions in Music are relatively straightforward, mainly focusing on converting  
between MMusic, MChord, BMusic, and Music Pitch. The actual application of the CTransLists is  
handled by transChord and its auxiliary functions. Once the MChords and the MBass are formed and  
properly transformed, they are converted to Music Pitch and played with Euterpea's play function. The  
lengths of the chords are conveyed to Display by mkCDurList to keep everything in sync.
Square

Square's only purpose is to provide a function to draw a square of a certain size wherever the view has been transformed to.

To conclude, I'll offer a piece of advice for someone wanting to experiment with Diminuendo. I have found that the following transformation relationships make for the most intuitive understanding of the program:

<table>
<thead>
<tr>
<th>Music Transformation</th>
<th>Graphics Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move or sMove</td>
<td>Rotate</td>
</tr>
<tr>
<td>Pivot</td>
<td>Translate</td>
</tr>
<tr>
<td>Cadence</td>
<td>Flip</td>
</tr>
<tr>
<td>Switch Mode</td>
<td>Flip</td>
</tr>
<tr>
<td>Switch Dur, Scale Dur</td>
<td>None</td>
</tr>
</tbody>
</table>

With a little training, it becomes quite natural to follow the music visually. Stick with a certain set of transformations until you can see where the music is going as you hear it. Try interesting combinations; a transformation consisting of a Move, Pivot and Switch Mode can produce interesting results. Finally, the chords produced by Diminuendo are only a starting point. Once you can track the musical progression by eye, try playing along with your favorite instrument. Start at a slow tempo and have patience!

You may alter and distribute the code as much as you like. Email me at lewispossible@gmail.com with any questions or comments. Enjoy!