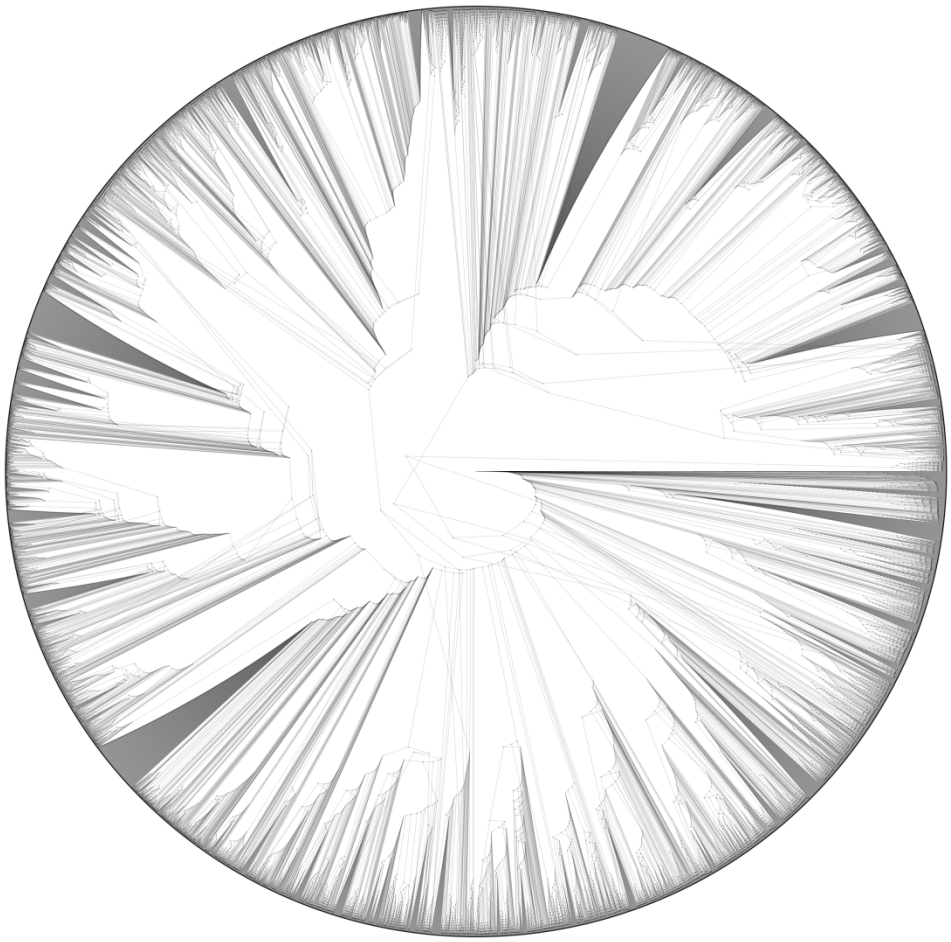
Beautiful Code, Compelling Evidence

Functional programming for information visualization and visual analytics



J.R. Heard

Contents

Introduction

Visualization programmers need a close-to-the-hardware, powerful 2D and 3D graphics toolkit to create applications that handle large amounts of data well. OpenGL is the most portable system for 3D graphics card programming, and thus is a common language for many visualization applications. Scott Dillard has said of the OpenGL graphics library bindings in Haskell:

> *The library is fantastic. I don't think it gets enough fanfare. The only other GL API that rivals it is the C API itself. Most other languages provide a shoddy and incomplete interface to that, instead of an idiomatic interpretation of the OpenGL specification. I can't think of a single language, not even Python, whose OpenGL bindings come close.*

OpenGL is powerful, but it can also be more complicated than actually necessary. For applications involving 2D graphics, a low amount of interactivity, and a smaller amount of data, it would be simpler not to bother with the video card and the rendering pipeline. Additionally, some visualizations are meant for print form. The Gnome Foundation's Cairo 2D graphics toolkit is perfect for these applications. As luck would have it, Haskell also has an excellent binding to Cairo.

Three other things make Haskell ideally suited to information visualization and visual analytics: a well-thought out and extensible library of generic data structures, lazy evaluation, and the separation of data transformation code from input/output inherent in a pure functional programming language. Visualizations written in Haskell tend naturally to break up into portions of reusable and visualization specific code. Thus, programs for visualization written in Haskell maintain readability and reusability as well or better than Python, but do not suffer the performance problems of an interpreted language.

For much the same reason as Simon Peyton-Jones put together *Tackling the Awkward Squad*, I have put together these lecture notes. I hope these will bring a programmer interested in visualization and open to the aesthetic advantages of functional programming up to speed on both topics.

Functional programming

A lazy functional programming language such as Haskell has practical effects that reduce the programmer's concern over minutia that are irrelevant to the larger task at hand. Namely, it expresses concurrency and takes advantage of parallelism without the programmer having to bother with barriers, threads, shared data, and IPC. It is capable of loading, unloading, processing, and storing complex data structures *on demand* without complicated support structures, caching, and loading schemes. It enforces separation of data transformation from data presentation. It makes debugging easier for a variety of reasons, one being type-safety and compiler-time strict type checking. Finally, recursive data structures, such as graphs and trees, can be naturally expressed and traversed in functional programming languages without loss of efficiency.

Visual analytics

Traditional scientific visualization is primarily concerned with showing structured data about events or phenomena in the physical world. Earthquakes, supernovae, ocean currents, air quality, wind-tunnel tests, hydrodynamics; the models generated by scientific simulation or studies of our environment or ourselves are the concern of scientific visualization.

Information visualization and visual analytics are a bit different. Information visualization concerns itself with representing knowledge, facts, and the structures we use to order these facts. Graphs, charts, maps, diagrams, and infographics aim to illustrate clearly found relationships in a manner more efficient and more elucidating than, or complimentary to lengthy articles or papers.

Visual analytics extends this notion to add the visual element to the process of data mining. Visual analytics applications take raw data and apply transforms iteratively (or recursively), allowing the user to see intermediate results of mining operations and direct the further application of mining techniques based on the results.

In this tutorial, we will focus on visual analytics and information visualization. Haskell's abstract data types map well to these applications. Once you master the simpler visualizations I present in this tutorial, you can, on your own, apply Haskell's wide array of prepackaged data structures to look at the World Wide Web or the Facebook social network as a directed graph, at Amazon rankings as an ordered list, or other collections of data in sets or associative lists.

Preparations

Before you start this tutorial, I recommend that you install GHC 6.8.3[1], the latest version of the Glasgow Haskell Compiler (GHC) as of this writing, and while you're at it, install GHC's extralibs package as well. This will contain all the OpenGL and GLUT bindings as well as quite a few other things you will find essential as you follow along. You will also want to get the Cairo library from *http://haskell.org/gtk2hs* or from the tutorial's website, *http://bluheron.europa.renci.org/renci-haskell*.

Other resources you may find helpful after going through this tutorial are: the OpenGL documentation for Haskell[2], *Yet Another Haskell Tutorial*[3], SPJ's excellent *Tackling the Awkward Squad*[4] essay on IO, concurrency, and exception handling, and finally the OpenGL blue or red book (known as the *OpenGL SuperBible* and the *OpenGL Reference*, respectively).

Getting Organized

Ben Fry's recent book *Visualizing Data* argues for a particular process to developing visualizations. I find it helpful in my own work, and therefore we'll be using it here to structure our tutorial. Briefly, the process is:

Acquire • Parse • Filter • Mine • Represent • Refine • Interact

Acquire
: Obtain the raw data source, or a sample of the raw data source to construct your visualization.

Parse
: Create data structure that is natural to any of: the underlying data, the way you intend to visualize the data, or the techniques you will use to mine the data.

Filter
: Slice and dice, compress, and clean data until you have only the data you need to visualize.

Mine
: Use data mining techniques or summary statistics to find patterns in the data.

---

[1] *http://www.haskell.org/ghc*
[2] *http://www.haskell.org/ghc/docs/latest/html/libraries/*
[3] *http://en.wikibooks.org/wiki/Haskell/YAHT*
[4] *http://research.microsoft.com/%7Esimonpj/Papers/marktoberdorf/mark.pdf.gz*

Represent
: Choose a visual model to represent your data. Use a bottom-up approach. Start simply and work up to more complex representations as needed.

Refine
: Improve the representation until you have something you're satisfied with.

Interact
: Add tools for the user to interact with the visualization, and if necessary, add tools for the visualization to interact with the data stream.

ACQUIRE

The internet is full of data. Life is full of data. Science is full of data. The most persistent problem for a data miner or a visualization professional is that people don't understand why or when to visualize data. The second most endemic problem is that people often don't know what data they have.

Information visualization is about presenting and supporting conclusions visually. Showing patterns that are non-obvious, or associating data in ways that elucidate aspects of the data's structure. Visual analytics is about answering questions progressively. The goal is to give the researcher a visual tool *before* s/he has come to the final conclusions about the data. Although inferences can't be directly supported by visualization (the gestalt of scientific research today is that statistics or proofs are required to back up conclusions, not pictures), the process of a scientist mining his or her data can be directed by these tools, especially when data structure gets complex.

Most people have trouble understanding what can be useful and usable to a visualization or data mining person, and so omit from their catalogue quite a lot of what they think won't be helpful to you. If you're working with people rather than just finding your data on the internet, you'll need to ask questions.

The first question to ask someone interested in applying visual analytics, is, "What kinds of questions are you interested in answering?" You'll get a general answer, and this won't be sufficient or directed enough to create a single visualization from, but hopefully from there you can ask narrower and narrower questions until you have something you can bite off.

After that, the question becomes "What's the data I have to work with?" You'll often get the answer, "We don't really have anything." This is almost never the case. Create a mental prototype of what the visualization that answers their question looks like. Think of what data you would need to come up with to create the visualization. Ask for that. It's much easier to acquire data if you can be specific in what you want. I once almost missed several gigabytes worth of useful text documents in a data mining job, because the owner of the documents didn't think I could use them because they weren't in an SQL database. It took careful directed questioning to get them to realize what they had.

Internet acquisition is somewhat easier. The web is its own enormous dataset, of course, as are the various social networks out there. The US Government website, USGS, and CIA FactFinder have interesting data to play around with to get your chops. If you're creating visual analytics tools for a client, it's often desirable to create mashups using their data as well as new data from the internet that is related in some way to what they're doing.

PARSE

The raw data you have can be as complicated or as simple as you like, but I'm going to suggest a couple of basic structures for storing data on disk that can get you by in many (if not most) situations.

First and foremost, there's the regular delimited text file, stored in either row or column major format. If you have statistical or numerical data you are going to process without the need for excessive querying, insertion, or deletion capability, this is an excellent format to put your data into, even for databases of several gigabytes in size. Yes, it's wasteful, but it's easy to read into Haskell, and usually easy to create from existing datasets. While tempting to build, a structured database won't save you anything unless you are going to use the same data repository outside of the visualization.

As an efficiency tip, it's best to store chunks of data you intend to process as a set as lines on a disc, as Haskell can read these lines lazily. This is backwards from the way you normally work on a spreadsheet:

Name	**Rank**	**Salary**	**Service Years**
Smith	Field Agent	50000	4
Jones	Field Agent	50500	5
Calhoun	Director	120000	21

Instead, you would store the data thusly:

Name	Smith	Jones	Calhoun
Rank	Field Agent	Field Agent	Director
Salary	50000	50500	120000
Service Years	4	5	21

Then the following function will read a text file stored in this format (tab delimited) and lay it out into a dictionary where each row can be accessed by the name stored in the leftmost column:

imports, aliases (1-3)

```
import Data.List (foldl')
import qualified Data.ByteString.Lazy.Char8 as BStr
import qualified Data.Map as Map

readDatafile name = do
```

Split all lines in the file. (6-7)

```
    sheet <- (map (BStr.split '\t') . BStr.lines) `fmap`
             BStr.readFile name
    return $ foldl' go Map.empty sheet
```

Insert them into the map (9)

```
  where go m (x:xs) = Map.insert (BStr.unpack x) xs m
```

Trust me – you'll want to cut and paste that somewhere. It's a useful function. What would you do if the file were, say, 16GB though? It certainly won't fit all into memory. In a language like C or Python, you'd have to rewrite the function so that it didn't read the entire file at once, but in Haskell you don't. You can use this same function for any tab-delimited file of any size, because Haskell will only load the data as it is needed, and because the garbage collector will throw out the data after it has been processed. That's the beauty of *laziness*.

What's more, we're using something called *Lazy ByteStrings*, which allow Haskell to read enormous amounts of data as quickly as it can be done in the best implemented C functions[5].

[5] See *http://cgi.cse.unsw.edu.au/~dons/blog/2008/05/16#fast* on writing reliably fast Haskell code.

We're talking about hundredths of a second difference between the two. You won't write an entire visualization in Haskell only to have to port the code to C later for performance reasons.

Because this is such a nice and versatile function, I'm going to say it makes sense to extend the notion, for now, of using a regular delimited file to deal with hierarchically structured data. Yes, you may want to use XML to store your hierarchical data instead, and you are welcome to, but for the sake of not adding APIs to the tutorial beyond OpenGL, It will do to create two files: one for the linkage relationships between nodes, and one for the data at each node.

Each row in the linkage file will contain a parent's in the left column. The names of the children will follow in the remaining columns. In the other file will be the data associated with each name, row by row, with the individual data elements appearing in columns, much like a traditional spreadsheet organization. I have included an example file on the web for this kind of dataset, as well as functions to turn it into a tree of data.

There are other formats out there, of course, such as text, rich text, images, XML, and NetCDF (for regular scientific data), as well as packed binary formats of all different sorts. These formats require special treatment, and we won't be covering them in this tutorial.

The function that I wrote earlier brings the data into Haskell's heap as a map of raw ByteStrings. We need to get that into data we can use. If a datatype is not a string, but still fairly simple (that is, an integer or floating point number, an enumerated type, or anything else Haskell's "read" function can handle), then a fairly simple function will suffice to turn a list of ByteStrings into a list of usable data:

map applies its argument to all the elements of a list.. read turns a string into data. Bstr.unpack forces the data to be read from disk.

```
import qualified Data.ByteString.Lazy.Char8 as BStr


toData :: Read a => [BStr.ByteString] -> [a]
toData = map (read . Bstr.unpack)
```

This function handles most cases, and it's so short, it's often more convenient to just write it inline. Any primitive except a string can be read by it, including any user defined types that are declared to be "readable" (we'll see how to do that in a minute). Creating Strings instead of ByteStrings is accomplished by removing the `read .` from inside the parentheses (and changing the prototype) to `toData :: [BStr.ByteString] -> String`. It's amazing how many datatypes in Haskell are just readable by default: parenthesized tuples of readables are readable, as are bracketed lists of readables, record types of readables, and even recursive types containing readables, making the simple delimited file much more powerful that it sounds on the surface:

Declare a record type like a C struct.

Declare it readable/writable.

[1,2,3,4,5,6] is readable.

(One,Two…) is also readable.

```
data RecordType =
  Record { enum :: EnumeratedType
         , value :: [Float]
         } deriving (Ord, Read, Show)

let list = [1,2,3,4,5,6]
let tuple = (One,Two,Three,Many 4,Many 5)
```

Specifying instances of these declared types in your file is as simple as writing instances of them like you would in your source code.

This line declares it readable

```
data EnumeratedType =
    One
  | Two
  | Three
  | Many Int
  deriving (Ord, Read, Show)
```

There are of course situations where this basic format doesn't suffice, and Haskell has tools for those. Parsec is a library that comes with GHC by default which allows the user to construct parsers. Happy and Alex are Haskell tools similar to *yacc* and *lex*. Also well supported are validating and non-validating XML parsing at various DOM levels and HTML parsing. The learning of these tools, however, is left as an exercise for the reader.

The structure that you end up with will likely be reflected by your visualization, and in the way you construct the visual representation. If you build a tree, you will obviously want to use visualizations appropriate to a tree, but you will notice that you can construct the visualization by walking the tree in some natural manner, such as a `fold` or a `map`.

Filter

One of the beautiful things about laziness in a programming language is how short this section of the tutorial can be. There are situations where you want to filter your data because it's easier to conceptualize, but filtering is less a question of removal than it is structure. If your data is structured properly, laziness will take care of not loading too much nor doing too much processing.

Only as much of your data structure as you actually use will ever be constructed, which can save on time, and can allow you to write code more generally than you would in another language, because you don't have to worry about unnecessary processing taking up time that could be used in your visualization. For record types, this means that only the individual fields of the record that are accessed will ever be constructed. For recursive types, such as trees and lists, only as far into the structure as you descend will ever be constructed. Even in some instances arrays can be lazy, evaluating elements only as they need to be evaluated.

Mine

Aside from the excellent implementation of OpenGL, mining is the reason to use Haskell for visualization. The functions in `Data.List`, `Data.Set`, and `Data.Map`, in fact, give you most of the tools you need to mine data for visualizations. These three together support the following types of operations on lists, unique item sets, and associative dictionaries (similar to Python or Perl dictionaries, and known in Haskell as maps):

- Transformations (mapping)
- Reductions (folding)
- Programmed Construction (scans, accumulation, infinite lists, unfolding)
- Merging (zip, union, intersection, difference, and other common set operations)
- Extraction (sublists, splits, and partitions using indices or predicates)
- Search and Lookup

Most data mining functions can be reduced to combinations of mathematical functions applied using these operators. For example, clamping a value between a high point and a low point and mapping it to a value between [0,1], linearly is:

Prototype
Clamp the value from low to hi.

```
clamp1 :: Floating a => a -> a -> a -> a
clamp1 lo hi val = (val - lo) / (hi - lo)
```

Similar prototype, but note the last parameter and return are lists. map makes clamp work on lists.

```
clamp :: Floating a => a -> a -> [a] -> [a]
clamp lo hi = map (clamp1 lo hi)
```

The same thing can be done with `Data.Map.map` and `Data.Set.map`; these functions take, instead of lists as their arguments associative maps and unique item sets.

The following function will compute the basic sample statistics: max, min, mean, variance, and sample standard deviation, often needed for performing normalization of the data, or comparing two different datasets:

Prototype

```
stats :: (Ord a, Floating a) => [a] -> (a,a,a,a,a,a)
stats (x:xs) =
```

Data.List.foldl' does a reduction with an accumulator.

```
  finish . foldl' stats' (x,x,x,x*x,1) $ xs
```

The elemental step that adds the element x to the accumulator

```
stats' (mx,mn,s,ss,n) x = ( max x mx
                          , min x mn
                          , s + x
                          , ss + x*x
                          , n+1)
```

Calculate the mean, the variance, and the standard deviation from the values stored in the accumulator

```
finish (mx,mn,s,ss,n) = (mx,mn,av,va,stdev,n)
  where av = s/n
        va = (1/(n-1))*ss - (n/(n-1))*av*av
        stdev = sqrt va
```

And finally, the next function will compute paired t-tests for two datasets, using the results of the function for computing sample statistics:

Paired t test
Max and min are never computed, because they're never used. Yay laziness.

```
pairedTTest left right =
  (x1-x2) / sqrt ( s1**2/n1 + s2**2/n2 )
  Where (_,_,x1,_,s1,n1) = stats left
        (_,_,x2,_,s2,n2) = stats right
```

Quite a lot of the time, 99% of your mining process is simply getting the data into the structures you're going to use. Much, then, can be inferred from building the data structures, and with Haskell's lazy evaluation, you can specify as much as you want to be inferred without worrying about its impact on the performance of the program. The inferences will only take place when the user somehow needs them in the display.

REPRESENT

Since I started programming visualizations in Haskell, I have developed a working pattern for building the visual elements from scratch. It involves four steps:

1. Create functions to transform your data structure into geometry.
2. Write functions that render that geometry.
3. Define the mutable state of the program.
4. Create a master render function to render the scene, which is sensitive to OpenGL's selection mode and renders only the selectable elements when called in this mode.

The first thing you want to do is create some rules for turning your data into geometry. Generally, you will create some structure of 3D or 2D vectors and possibly material and color information that can be used later in the actual rendering function, or passed through another transformer to do something interesting to the geometry (like force-directed layout algorithms) before rendering it. If your geometry's structure mirrors the structure of your data, then you can traverse them together in further computations, making it easy to transform geometry based on the data at hand. This is not as complicated as it sounds:

DatNode would be a user-defined module.

```
module DataGeometry where
import DataNode
import qualified Graphics.Rendering.OpenGL.GL as GL

black :: GL.Color4 Float
black = GL.Color4 0 0 0 1
```

OpenGL geometry for a tree containing name for GL selection and a color.
Left child
Right child
Leaf declaration has no childrent.

```
data Tree = Node { name :: GL.Name
                 , vertex :: GL.Vertex3 Float
                 , color :: GL.Color4 Float
                 , left :: Tree
                 , right :: Tree }
          | Leaf { name :: GL.Name
                 , vertex :: GL.Vertex3 Float
                 , color :: GL.Color4 Float }
```

Recursive case. Creates a tree that minimcs the DataNode and DataTree structures..

```
polarGeometry rad n_leaves
    (DataNode num height depth lt rt) =
  Node (GL.Name num) (GL.Vertex3 r t 0) black lt' rt'
  where
    h = fromIntegral height
    d = fromIntegral depth
    r = h / (depth+height) * rad
    t = (ycoord lt' + ycoord rt') / 2
```

Recurse.

```
    lt' = polarGeometry rad n_leaves left
```

Recurse.

```
    rt' = polarGeometry rad n_leaves right
```

Leaf case. No recursion in this function..

```
polarGeometry r n_leaves (DataLeaf num) =
  Leaf (GL.Name num) (GL.Vertex3 r t 0) black
  where t = 2*pi*(fromIntegral num)/(fromIntegral n_leaves)
```

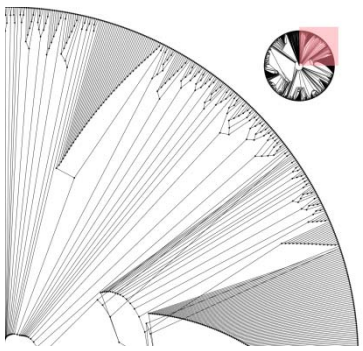
The preceding function creates a perfectly round tree with leaves spaced evenly along the rim and internal nodes positioned relative to the height and depth fields in the dataset. Note that the new DataGeometry Tree exactly mirrors the structure of the DataNode Tree. Also note that no graphics calls are actually made here.

This is a layout function, which determines ahead of time where everything should be. Now, we can use the same geometry to create lines, points, and position labels or other, simply writing a function that traverses this geometry, the original data, and any other ancillary data we want to layout with it. After this, I write a function that will traverse the data and the newly created geometry structure together and produce OpenGL calls that will render the structure the way I want it rendered:

```
import qualified DataGeometry as Tree
import qualified Graphics.Rendering.OpenGL.GL as GL

-- Prototype
treeLines :: Tree.Tree -> IO ()
treeLines (Tree.Node n srcV srcClr lt rt) = do
    -- Set drawing color,
    -- Draw head vertex,
    GL.color srcClr
    GL.vertex srcV
    -- Draw left child vertex,
    GL.color ltClr
    GL.vertex ltV
    -- Draw head vertex,
    GL.color srcClr
    GL.vertex srcV
    -- Draw right child vertex.
    GL.color rtClr
    GL.vertex rtV
    treeLines lt
    treeLines rt
    -- Define variables used above.
    -- Note that lt and rt are in
    -- the next level of the tree
    where ltClr = Tree.color lt
          rtClr = Tree.color rt
          ltV = Tree.vertex lt
          rtV = Tree.vertex rt
-- Don't draw from the leaves.
treeLines (Tree.Leaf _ _ _ _ _) = return ()

-- Nnodes are different; we
-- want to make them
-- selectable using
-- GL.withName.
treeNodes :: Tree.Tree -> IO ()
treeNodes (Tree.Node n v c lt rt) = do
  GL.withName n $ GL.renderPrimitive GL.Points $ do
                    GL.color c
                    GL.vertex v
  treeNodes lt
  treeNodes rt
-- Piecewise define leaf case
treeNodes (Tree.Leaf n v c) = do
  GL.withName n $ GL.renderPrimitive GL.Points $ do
                     GL.color c
                     GL.vertex v
```



If function after function to create new data structures sounds inefficient, keep in mind that laziness works in your favor. While conceptually, you will create a brand new tree that contains geometry for every bit of your data structure, and each new transformer you layer onto that data structure will also create a brand new tree, the effect in the program is more akin to piping parts of single data points through transformers only when they're needed. It's likely that the new data structure will never be stored in memory at all.

I generally create a separate module to encapsulate the program state code and the event handler functions. This contains a datatype called `ProgramState` (you can use anything, but this

is what I do), which is a record type containing everything that would look like a mutable global or instance variable. In your `main` program, you will create an instance of `IORef ProgramState` and pass it as the first argument to all of your callbacks. This will allow the callbacks to each do their own internal record keeping on the state of the program and preserve state between calls.

Your master render function will actually do the job of setting up OpenGL, setting up the scene parameters, and then calling the render functions for your individual visualization elements, usually creating display lists for the more complex elements. You'll be calling the render function from multiple places, notably the event handlers. You'll also be calling it for multiple purposes, such as rendering into the selection buffer. Because of this, you should make this the function that can analyze the current program state and render what ought to be displayed to screen completely, depending on nothing else. Finally, you will want to include the render function itself inside your `ProgramState` object; I use the name `renderer` for this.

Refine

The reason that the process of refining your visual presentation is listed before interaction is because you will generally cut out a few visual elements once you get your visualization on screen, and you may also decide to add a few. Refining your presentation is generally considered the hardest part of visualization in Haskell. Type safety and strict compile-time type checking, if not properly accounted for in some way, can make for a programmer having to salt and pepper their entire codebase with minor changes to fix typing.

This step is the primary reason that I've developed the pattern I have in creating visualizations. By separating program state from rendering, rendering from geometry, and geometry from data structure, as well as creating a master rendering function that takes into account the entire scene, there is a minimum of hassle in changing one part of the code versus another.

The main reason for the refine step is that you don't want to, as Ben Fry puts it, "build a cathedral." If you try to do everything at once, you'll often be disappointed; the person you're doing the visualization for won't need all of what you give them, you'll be late delivering it, and end up having to change things anyway. By starting with the mindset that you're going to develop visualizations incrementally, your code will be more flexible and you'll be able to create and deliver visualizations on-time that do everything they're supposed to.

Interact

To make any visualization useful, it has to interact with the user in some way. One of the handiest things about GLUT, the GL User's Toolkit is that it has easy to understand bindings for event handlers and some basic windowing capabilities. With the introduction of FreeGLUT[6], GLUT no longer requires that it has full control of the program's main loop, and therefore it can integrate with other user interface toolkits. In the old days, GLUT was used primarily as a teaching tool for OpenGL, but the functionality it contains is so useful that until you manage to develop an application in which you can see GLUT will not suffice, I recommend it for all OpenGL development. It is portable, reasonably small, and easy to program for. GLUT in Haskell is contained in `Graphics.UI.GLUT`.

Often, there is only one handler to write, the mouse-click/keyboard handler. The GLUT prototype for it looks like this:

[6] *http://freeglut.sourceforge.net*

```
KeyboardMouseCallback :: Key -> KeyState -> Modifiers -> Position -> IO ()
```

We will actually prepend this prototype with an `IORef ProgramState`, so we can manipulate and save the program state inside the callback. You will define this function piecewise. Another beautiful thing about Haskell is that you can often replace complicated case and nested if statements from other languages with a piecewise definition of a function. Now I'll deconstruct that prototype from above. Let's say you want to respond to all individual letter keys, left clicks, and right clicks of the mouse. We'll ignore control, alt, and shift for now:

Function prototype.

```
kmc :: IORef ProgramState -> KeyboardMouseCallback
```

The first parameter of the function is an instance of GLUT.Key. When the event is a keyboard character, the istannce of GLUT.Key is GLUT.Char, the character is bound to the variable c. You can handle pressing and releasing the key differently by repeating this instance and replacing GLUT.Down with GLUT.Up. The mouse position when the character was pressed is bound to x and y. The underscore is a "wildcard" parameter, meaning that we don't care about the value of Modifiers.

Piecewise definition 1

```
kmc state (GLUT.Char c) Down _ (Position x y) = do
```

Read the state.

```
  st <- readIORef state
```

… do something here to handle the individual keystroke …

Write the new state.

```
  state `writeIORef` st { … describe state changes here … }
```

Render, passing the new state.

```
  render st $ state
```

When a mouse button is pressed, GLUT.Key becomes GLUT.MouseButton. If it's the left button this is further specified to LeftButton, and if the button is currently Down, this function will be called. Note that if we didn't care about the mousebutton, we could replace LeftButton with _, and any mouse button being down would trigger the function.

Piecewise definition 2

```
kmc state (MouseButton LeftButton) Down _ (Position x y) = do
  st <- readIORef state
```

… do something to handle the left click …

```
  state `writeIORef` st { … describe state changes here … }
  render st $ state
```

Note the only difference here between the previous function and this one is RightButton.

Piecewise definition 3

```
kmc state (MouseButton RightButton) Down _ (Position x y) = do
  st <- readIORef state
```

… do something to handle the right click …

```
  state `writeIORef` st { … describe state changes here … }
  render st $ state
```

This is the default case, which will be called if the event doesn't match any of the other functions. This has to be last, as it will match anything. Any unrecognized event will be processed by this function.

Pricewise Definition 4

```
kmc _ _ _ _ _ = return ()
```

By placing a default definition at the end, you can define as much or as little event handling as you want. You could even create individual function definitions for all the different characters you are interested in handling. Defining these functions piecewise, however, doesn't create a ton of tiny functions internally to bog down your program's event loop; rather it is all combined into a single function in the compiler, allowing you to write code that's reasonably easy to read without sacrificing performance.

A few notes of explanation about the code are due. Generally, you will:

1. Read the state
2. Handle the event
3. Write any changes to the state necessitated by the event.
4. Render the scene changes.

In reading the state, the `<-` operator does two things (as opposed to the `=` operator, which you could use by mistake, although the compiler will catch you). First it draws the value out of the reference, creating an immutable copy of it. Second, it binds that immutable copy to `st`. This means that you can continue working on the state as it was when this click occurred: other clicks and state changes will not affect the way the function completes, and nothing this call of the event handler does will affect others running simultaneously. No locking, no semaphores, nor mutexes are needed.

Handling the event is usually a matter of figuring out which state changes need to occur based on the value of `st` and what event happened. As soon as you've figured those out, write the changes back to the state. If you need to render because of the state change, after you've changed the state object is the time.

Note that here is where we make use of having stored the master render function in the program state object. `(renderer st)` grabs the renderer element out of the state object. `$ state` causes the new state to be passed as the parameter to the master renderer.

One final note about the order in which you define a function piecewise: cases are evaluated in the order that they're defined. This means that you need to order your piecewise function definitions from the most specific to the most general. If you want to do something special on the value `'c'`, but all the other keyboard characters can be handled in the same fashion, define your case for `'c'` first, before the general case. The last piecewise definition you create should be the default case, where all other inputs possible will fall. This will mean that your program doesn't crash when the user does something you don't handle explicitly. If you don't define the final case, then bad user input will cause the program to abruptly quit without warning.

GLUT defines other input-output handlers as well: mouse-motion (passive/active), tablet, joystick, and the less-used spaceball and dial-and-button-box callbacks. You can define these incrementally, using more or less the same pattern as the mouse/keyboard handler I wrote earlier.

Reading Haskell for beginners

While I don't want to replicate the excellent work of others' more in-depth Haskell tutorials, I do want to introduce enough of the syntax in this introduction to give you a feel for how to read and write the code yourself. This tutorial won't give you the means to read every byte of code I've written for the tutorial, but it will give you an idea of what you're looking at.

First thing's first. This is a functional programming language, so we should talk about functions. I'm going to start with our statistics functions from earlier:

```
stats :: [Float] -> (Float,Float,Float,Float,Float,Float)
stats (x:xs) = finish . foldl' stats' (x,x,x,x*x,1) $ xs
```

```
stats' (mx,mn,s,ss,n) x = ( max x mx
                          , min x mn
                          , s + x
                          , ss + x*x
                          , n+1)

finish (mx,mn,s,ss,n) = (mx,mn,av,va,stdev,n)
  where av = s/n
        va = (1/(n-1))*ss - (n/(n-1))*av*av
        stdev = sqrt va
```

On the first line, we see the prototype of the function. Like a C or Java prototype, this declares the name of the function (before the `::`), the type of the parameters, and the return type. `[Float]` says that the function takes a list of `Float`s, which are regular, single-precision floating point numbers. `(Float,Float,Float,Float,Float,Float)` indicates that the function returns six `Float`s, which we can see in the code refer to the list's max, min, mean, variance, standard deviation, and the number of elements. The return type always follows the last arrow.

You don't always need a prototype for a function, much like in C. The compiler can usually (but not always) figure out one for you. It's advisable, though, to write them, both for your own understanding, and as help to the compiler. Your compile times will be shorter and the code the compiler produces will be better if you consistently write prototypes.

On the next line, we see the definition of the function itself. Once again we see the name, followed by a funny looking thing, `(x:xs)`. This is an *argument pattern*. What you actually see there is the list deconstruction operator, `:`. The `:` splits off the first element of the list from the rest of the list, and thus this pattern `(x:xs)` says that `x` is bound to the first number in the list and `xs` bound to the rest of the numbers in the list. Following the pattern is an `=` sign, which assigns the body of the function to the name stats, to be executed in the case that there is at least one element in the list, `x` (`xs` can be empty).

The body of the function, `finish . foldl' stats' (x,x,x,x,1) $ xs`, is a little harder to read. The `'` in `foldl'` and `stats'` is part of the name, first off, and is not an operator. The `.` operator refers to *function composition*, and the `$` operator refers to something called a *fixpoint,* which is really a way to shove an evaluated argument through the functions to the left of the `$`. Function composition allows you to construct a larger function out of a series of smaller functions, allowing you to pipe a single element through a created pipeline. You could write the same code as `(finish (foldl' stats' (x,x,x,x,1) xs))`, but I think you'll agree that's less clear that way. Those of you familiar with LISP will recognize the syntax. The dot and dollar-sign is how Haskell fixes it. It's actually a little more complicated than that (there is a difference between `$` and `.`, even if it doesn't look that way on the surface), but the understanding you have will suffice for now.

`foldl'` works a bit like a for loop with an accumulated result. You hand it a function, an initial value, and a list. The function should take the accumulator value on the left and a list element on the right, and incorporate that element into the accumulator value, and finally return the accumulator value. `foldl'` then uses this as the body of a for loop and builds the accumulator value up using each successive element of the list. At the end, it returns the accumulator. `stats'` is the function I pass to `foldl'`, and as you can see, it follows the rules, taking the accumulator value on the left and a single float on the right.

Finish is the final function, which takes the accumulator value that `foldl'` returns and creates our final statistics out of it. In this case, the body of the function is simply the final value,

`(mx,mn,av,va,stdev,n)`. Note that there is no return keyword in this language. This is because, conceptually, everything to the right of the equals sign is returned.

Notice the `where` keyword on line 11. In Haskell, we can normally bind all variables that are used inside a function as an afterthought. This is much more readable than littering the function with variable declarations as they are needed, or having to worry about having a variable defined before it is used. The `where` keyword accomplishes this, binding all the variables used at once in a list at the end of the function. Note you can even have the definition of one variable depend on another, as long as you don't create a circular dependency doing so. This is especially useful in more complicated functions where there are many intermediate results.

Finally, note something I have not done. I never changed the value of a variable once it's been assigned. That's because you can't. A variable's value, once assigned, is fixed. As cumbersome as this seems like it would be, once you get used to it, it saves you from all kinds of nasty errors. The fold and map family of functions generally handle the situations that arise that would cause you to change the value of a variable in other languages.

There's something else I haven't addressed. I/O. Lazy functional programming requires that you separate all input-output from the calculation of values. Unless the only thing you plan on doing is a single input/output statement, you need a `do` keyword, and you'll need to know that the function you're writing *must* have its final return value be the current program state, known, conveniently as `IO`. I/O is anything that *must* cause a write or read to memory, the screen, the network, a video card, a disc, *anything*. So a `GL.vertex` is an I/O because it writes to the video card. Assigning a new value to a reference in memory is I/O. Printing to screen is I/O.

This sounds complicated, and it does take a little while to get used to, however the savings in debugging time later are tremendous. Anything that doesn't do I/O, for example, can be executed in parallel without worrying about mutexes, semaphores, critical sections, and the like. Haskell allows you to take advantage of this in more advanced libraries. Also, you can't accidentally write over a piece of memory that is needed elsewhere, nor can you have an uninitialized value passed into a function.

The reason you have to do things this way is because Haskell is lazy, and at some point, you're going to have to *make* it do something. Lazy operations don't specify any ordering. If your print statements and IO were lazy, you would have no way to control when a vertex went to memory or a frame got rendered to the screen. Therefore these happen in the special `IO` type which is *strict* (the opposite of lazy).

The `do` keyword is rather special. For a complete example of it, go ahead and look at Appendix B's render function. First of all, `do` allows you to write successive lines of code that each do something separately from each other. These will occur in sequence, from top to bottom, because of the do keyword. This may look like we've completely changed languages to Python, but under the hood, do is just cleaning up something you could do using the Haskell we've discussed previously. Namely, it threads an implicit argument through each line, which is the result of the computation performed by the previous line. In this case, though, the result is the *entire program state,* which is also known as `IO`. Since it would be ridiculously cumbersome to account for the entire program state at each line: each heap and stack variable, the state of memory, the video card, the operating system, open files, the file system and so on, all this information is crystallized inside of `IO` so you don't have to think about it.

Inside a `do` keyword, every line is an action, explicitly sequenced in line order (and yes, indentation matters). In the render function in Appendix B, some lines assign values to variables using the `$=` operator. Yes, I said earlier you can't do that, and in fact what those lines do is look

up a reference in memory and write to that memory location. The value of the variable is the memory location, and that does not change. Some lines read the value of these variables. Some lines send vertices to the video card. All these will happen in the same order that you read them, and as long as the `main` function calls `render` at some point, all of them are guaranteed to happen.

Finally, I'll talk about custom datatypes. You see these everywhere in Haskell. Anything with an initial capital letter is either a module or a datatype. These are constructed and inspected using *pattern matching* like we mentioned before.

```
data Vertex = Vertex1 Float
            | Vertex2 Float Float
            | Vertex3 Float Float Float
```

This defines a simple vertex object that can be in one, two, or three dimensions and contains a single-precision floating point coordinate for x, (x,y), or (x,y,z). You can use it in a function prototype or a case expression like this:

```
translate (Vertex1 x) (Vertex1D by) = …
translate (Vertex2 x y) (Vertex2D byx byy) = …
translate (Vertex3 x y z) (Vertex3D byx byy byz) = …
```

Or in a `case` expression:

```
case vertex of
  Vertex1 x -> x
  Vertex2 x _ -> x
  Vertex3 x _ _ -> x
```

Note that all of the returns of the case statement return `x`. All possibilities in a case statement must return the same datatype. The next two datatypes are actually part of the Haskell 98 standard:

```
data Either a b = Right b | Left a
data Maybe a = Just a | Nothing
```

These two datatypes are *parameterized*. Either can take any two custom datatypes, `a` and `b`, and return type `a` with a `Left` and type `b` with a `Right`. This is often used in Haskell for exceptional cases, where `Left` represents some failure value and `Right` represents the proper result. Then you use a `case` statement to handle the different values appropriately (keep in mind that you still have to return the same type after handling the failure as after handling the success, because of the rules of the case statement). `Maybe` allows you to have uninitialized values – it takes the place of the null pointer that is so common in other programming languages. The advantage of `Maybe` over some general undefined value, however, is that using it makes it explicit in the code that there's a possibility of a null value, and forces you to handle the case separately:

```
case parse str of
    Left str -> "unparsable: " ++ str
  | Right tree -> show tree

case parser language of
    Nothing -> do fail "language not supported"
  | Just parse -> do putStr . parse $ str
```

The previous two case statements call functions. `parse` returns either a `Left` with the original string, which we prepend with `"unparseable: "` or a `Right` with the tree structure. We use `show` to convert the tree structure to string. In the second, we have a function that looks up parsers based on the language input. If there doesn't exist a parser, we `fail` (inside IO, this constitutes an exception). Otherwise, we bind `parse` to the parser function that is returned and print the parsed string to the screen.

```
data Tree a b = Node a [Tree]
              | Leaf b deriving (Show)
```

This datatype defines a tree. Note that the name is used on both the left and right sides of the equals sign. A `Node` contains a data value of type `a` and a list of children of type `Tree`, which is either `Node a [Tree]` or `Leaf b`. `Leaf` contains just a data value of type `b`. The best way to go through this datatype is with a recursive function:

```
mysum :: Tree Int Float -> Float
mysum (Node x children) =
  fromIntegral x + (sum . (map mysum) $ children)
mysum (Leaf x) = x
```

The next datatype we will consider defines a record type, like a C struct. There are two ways to access the members of the structure, and one way to "modify" the individual members of the structure (they actually return a partially new structure – the old value will still be accessible, but a new structure with the new values will be returned, sharing as much memory as possible).

```
data Triangle = Triangle { first :: Vertex
                         , second :: Vertex
                         , third :: Vertex }

firstx t = case first t of
             Vertex1 x -> x
             Vertex2 x _ -> x
             Vertex3 x _ _ -> x

firstx1 (Triangle (Vertex1 x) _ _) = x
firstx1 (Triangle (Vertex2 x _) _ _) = x
firstx1 (Triangle (Vertex3 x _ _) _ _) = x

translateAll n t = t{ first = translate n . first $ t
                    , second = translate n . second $ t
                    , third = translate n . third $ t }
```

Defining `Triangle` creates three functions: `first`, `second`, and `third`, which take a `Triangle` and return the element of the structure of the same name. Also, you can suffix a variable of type `Triangle` with `{ … }` to change the values of the individual elements, as in the definition of `translateAll`. Additionally, pattern matching can break down the internal structure of a record type, as in the second definition of `firstx` you see.

The final datatype we will consider is a simple typesafe enumeration. Note that we can derive instances of `Ord` and `Show`, which say that the ordering of the type is the same as the order we specified the type in, as well as defining a `show` such that `show` prints the names of the values as you see them:

```
data Frequency = Never | Sometimes | HalfAndHalf | Often | Always
               deriving (Ord,Eq,Show)
```

This is nearly everything you need to know to read most of the code in this tutorial. One other miscellaneous rule that I haven't covered is that of how to recognize infix operators (like +, -, /, and *). Any function whose name is composed of symbols rather than letters or numbers is an infix operator, and can take at most two arguments. Also, any two argument function can be made to be infix by surrounding it with backquotes: `` `fun` ``. Knowing these rules, you should now be able to read most, if not all of the Haskell code in this tutorial, and you can use the boilerplates I've supplied to write your own code.

HASKELL'S OPENGL INTERFACE

This is also not the venue for a full OpenGL Haskell tutorial; however I will give a brief introduction to OpenGL for 2D visualizations here and talk a little bit about the differences between the C API and the Haskell API, as well as how to read the reference material on the Haskell website. Later in this document is a boilerplate for building an OpenGL visualization. The best tutorial you can get is to skim through the OpenGL SuperBible, working through examples here and there as they pique your interest. It is organized in more or less a progressive manner, giving you the tools to crank out basic scenes quickly.

OpenGL is two things. First, it is a framework for specifying operations in a rendering pipeline. Second, it is a state machine that controls how these operations are processed. The general procedure for rendering a scene is:

1. Set up the windowing system and create a GL context (GLUT does this)
2. Set up state variables.
3. Set up the projection matrix (also a state variable).
4. Set up the model-view matrix.
5. Pass some rendering commands through the pipeline.
6. Flush the pipeline.
7. Swap buffers.

Steps 2-7 are repeated over and over for each frame of your scene. You only need to set up the windowing system once. Until you know what you're doing with OpenGL, I would recommend the following GLUT call to set up your window:

```
GLUT.initialWindowMode $ [GLUT.DoubleBuffer, GLUT,RGBA, GLUT.Depth]
```

This will create a window that doesn't flicker while you update it, allows you to create translucent objects, and can handle depth testing, which tells OpenGL to act intuitively when you're working in 3D and some objects you specify overlap. Now for steps 2-7, I will try to give you the calls that will be most useful to the most people, which will make your scene look fairly polished by default.

Can be the render callback

```
render = do
```

Step 2.

Clear the screen to black

```
  GL.clearColor $= GL.Color4 0 0 0 1
```

Enable blending & smoothing

```
  GL.blend $= GL.Enabled
  GL.blendFunc $= (GL.SrcAlpha, GL.OneMinusSrcAlpha)
```

Smooth lines

```
  GL.lineSmooth $= GL.Enabled
```

Smooth points

```
  GL.pointSmooth $= GL.Enabled
```

Smooth shape edges

```
  GL.polygonSmooth $= GL.Enabled
```

Clear the backbuffers

```
GL.clear [GL.ColorBuffer, GL.DepthBuffer]
```

Step 3.

Set up the projection matrix
Start with a fresh matrix
Find the window reolution
GL coords = window coords

```
GL.matrixMode $= GL.Projection
GL.loadIdentity
(_, GL.Size xres yres) <- GL.get GL.viewport
GL.ortho2D 0 0 (fromIntegral xres) (fromIntegral yres)
```

Step 4.

Set up the modelview matrix with a fresh matrix

```
GL.matrixMode $= GL.Modelview 0
GL.loadIdentity
```

Step 5.

Without affecting our matrix set up in the future, render a white square of widh 100, starting at point (10,10)

```
GL.preservingMatrix $ GL.renderPrimitive GL.Quads $ do
  GL.color (GL.Color4 1 1 1 1 :: GL.Color4 Float)
  GL.vertex (GL.Vertex2 10 10 :: GL.Vertex2 Float)
  GL.vertex (GL.Vertex2 110 10 :: GL.Vertex2 Float)
  GL.vertex (GL.Vertex2 110 110 :: GL.Vertex2 Float)
  GL.vertex (GL.Vertex2 10 110 :: GL.Vertex2 Float)
```

Step 6.

Flush the pipeline

```
GL.flush
```

Step 7.

Show the drawing on screen.

```
GLUT.swapBuffers
```

The matrices, by the way, set up your coordinate system. Here, I've done something a little unusual, which is considered bad-form by many OpenGL programmers, but I argue works in practice more often than not: the OpenGL coordinates are equal to the window coordinates. In 2D, this is not really a big deal at all, and gives the beginning programmer a crutch with which to think about where things are in the window. Note OpenGL numbers from the bottom-left to the top-right, which means that y=0 is at the bottom edge of the window, while y=1 is above it. By default, all the coordinates go from zero to one, meaning that the bottom left is (0,0), and the top right is (1,1). If your window is square, this is fine, but it's often not.

Vertices are specified for a single polygon in counter-clockwise order. That is merely an OpenGL convention, and assures that the "front" of the polygon you draw is facing the user. This becomes important later when dealing with more complicated scenes involving texturing and lighting, but for now we'll just mention that it's good practice.

Now to explain the COpenGL/HOpenGL differences. The first, and strangest thing that a veteran OpenGL programmer will notice is the lack of `glBegin()` and `glEnd()`. This is replaced in HOpenGL by `renderPrimitive`, which is used like this:

Takes an IO () as a parameter, which we can substitute with an anonymous do block

```
renderPrimitive GL.Lines $ do
  GL.vertex v1
  GL.vertex v2
  ...
```

This formalizes the practice that the OpenGL manual suggests, that you indent code between `glBegin` and `glEnd` invocations, as if it were a function. Here, it is an actual function. Note that `GL_LINES` also becomes `GL.Lines`. This is the general pattern when it comes to constants. Anything that is prefixed by `gl` or `glu` has the prefix removed (we can qualify it with the module name `GL` instead), and underscores are elided using camel case. So, for example, `GL_TRIANGLE_FAN` becomes `GL.TriangleFan`. There are a few other places where this paradigm is used, such as `glPushMatrix`, which becomes `GL.preservingMatrix`, and with

selection mode (using `GL.withName` *name*). In general, look for places in OpenGL where you push something onto a stack before a `glBegin`.

The next strange thing is that `glEnable` is gone, and most things that are considered part of the OpenGL machine's "state" are encapsulated in something called a `StateVar`. The `StateVars` work like this:

```
-- Grab just the $= symbol
-- Everything else is GL.*
import Graphics.Rendering.OpenGL.GL ($=)
import qualified Graphics.Rendering.OpenGL.GL as GL

-- Set some state variables
-- Enable smoothing.
GL.lineSmooth $= GL.Enabled
-- Enable blending.
GL.blend      $= GL.Enabled
-- Set ModelView matrix, level
GL.matrixMode $= GL.ModelView 0

-- Get the value of state variables
-- Get value of smoothing.
smoothing <- GL.get GL.lineSmooth
-- LHS pattern matching binds variables from viewport.
(GL.Size xres yres,GL.Position x y) <- GL.get GL.viewport
```

There's a bit of trial and error involved in knowing what is a `StateVar` and what isn't, but you start to get the feel for it after using HOpenGL for a little while.

The last thing to get used to, which is actually rather an improvement rather than a mere difference between the two APIs, is the way vertices and vectors are specified. Haskell introduces type safety to this common task, letting you be explicit about whether you're indicating a a Size value or a Vector value or a Vertex, etcetera, and also constrains these types so that only valid values can go in. These API constraints catch at compile time some of the "meters vs. inches" problems that are common problems in OpenGL code. Some common types are `GL.Size`, `GL.Position`, `GL.Vector[2,3,4] a`, `Gl.TexCoord[1,2,3] a`, and `GL.Vertex[2,3,4] a`, where a is a numeric type. You can find more types and more explicit documentation on where they're used and what values they take in the docs for `Graphics.Rendering.OpenGL.VertexSpec`[7] and `Graphics.Rendering.OpenGL.CoordTrans`[8].

This covers most of the OpenGL basics. OpenGL can take months to master for visualization and years to master for creating realistic scenery for games, but getting your feet wet with it isn't all that hard, and it's the least crippled toolkit for graphics in existence. You will not be working on a project in OpenGL and suddenly realize that there's something that just cannot be done.

[7] *http://haskell.org/ghc/docs/latest/html/libraries/OpenGL/Graphics-Rendering-OpenGL-GL-VertexSpec.html*
[8] *http://haskell.org/ghc/docs/latest/html/libraries/OpenGL/Graphics-Rendering-OpenGL-GL-CoordTrans.html*

HASKELL'S CAIRO INTERFACE

A simpler interface than OpenGL to use in everyday situations is Cairo. Cairo is limited to 2D graphics, and cannot handle interactivity or large volumes of data as well as OpenGL, but it is easier to export the data to the outside world, making it ideal for graphs, maps, charts, and other static or semi-static visualizations. Unlike OpenGL, Cairo does not come with GHC. I have supplied the latest version of Cairo for Windows and 32-bit Fedora Core Linux on the CD, but the Cairo distribution should generally be gotten off the Gtk2Hs project's website[9], which also serves up the full documentation[10] for Cairo and Gtk2Hs.

Cairo works on a stencil-and-paper model – the number one reason why it's more intuitive than OpenGL. It maintains an idea of where the pen is at all times, what color it is, the line width, and several other convenient things as part of its state. All drawing is done on a flat sheet of virtual paper, known as a `Surface`. A surface can be a PNG image, an off-screen image, a PDF, a PostScript file, or an SVG (Scalable Vector Graphics) file. The steps for rendering an image in Cairo are:

1. Create a surface.
2. Set up the paint color
3. Pass rendering commands through the pipeline to create a stencil
4. Transfer the stencil to the surface using the current paint color.
5. If necessary, write the surface to disk.

Steps 2-4 are repeated for every element in your scene. You don't have to enable smoothing or worry much about Cairo's state unless you are doing something really complicated. Most everything is set to sensible defaults for you. The default coordinate space is in pixels for image surfaces and points (1/72in) for vector based surfaces like PDFs and SVGs.

CCairo and HCairo are virtually the same except for the introduction of type-safe enumerations instead of C style enumerations and the ability to have Haskell temporarily allocate surfaces and automatically garbage collect them. The API is also quite small, the complete documentation fitting into a single HTML file. Let's look at an example, which renders a 100x100 pixel black square with the text "Hello World," to a PNG file:

Note that the calls to `C.fill` actually draw the objects to the surface. Until then, you are building a virtual "stencil", which will be used to `C.setOperator C.OperatorSource` is a compositing operator, the documentation for which can be found on the Cairo website[11], but basically, this determines how the current drawing state gets written onto the surface. The most common operator is `OperatorOver`, which tells Cairo that any calls that draw should draw over what is already there, like pen on paper.

[9] *http://haskell.org/gtk2hs/download*
[10] *http://haskell.org/gtk2hs/docs/gtk2hs-docs-0.9.12/Graphics-Rendering-Cairo.html*
[11] *http://www.cairographics.org/manual/cairo-cairo-t.html#cairo-operator-t*

```
Import cairo qualified
import qualified Graphics.Rendering.Cairo as C
                                                                    Step 1
withImageSurface allocates a
100x100 RGBA image called
surf.
main = C.withImageSurface
  C.FormatARGB32 100 100 $ \surf -> do
   C.renderWith surf $ do
Save cairo's state.
      C.save
Sets how composting is done.
      C.setOperator C.OperatorOver
                                                                    Step 2
Set the pen color to black.
      C.setSourceRGB 0 0 0
                                                                    Step 3
Create a rectangle.
      C.rectangle 0 0 100 100
                                                                    Step 4
Fill in the rectangle.
      C.fill
                                                                    Step 2
Set the pen to white.
      C.setSourceRGB 1 1 1
                                                                    Step 3
Select a font.
      C.selectFontFace "Trebuchet MS"
                       C.FontSlantNormal
                       C.FontWeightNormal
Set the font size in points.
      C.setFontSize 18
Draw some text.
      C.textPath "Hello world"
                                                                    Step 4
Fill it in.
      C.fill
Restore Cairo's state.
      C.restore
                                                                    Step 5.
Write the result to file.
   C.surfaceWriteToPNG surf "Text.png"
```

Cairo Example : Sparklines

A perfect example to show off how and when to use Cairo are sparklines. Sparklines embed inline with text and show a reader an informative graph at quick glance whenever statistics are cited. As more and more typesetting systems are able to embed pictures with text, they are gradually becoming more common. Our example will take a simple column from a tab-delimited file, like you might get out of Excel, and create a sparkline out of it with a given width, height, and resolution. The sparkline will output as a PNG file, and can be used as a picture inside of Word or OpenOffice. [12]

StructuredDataHandler.hs

```
module StructuredDataHandler where

import Data.List (foldl')
import qualified Data.ByteString.Lazy.Char8 as BStr
import qualified Data.Map as Map
                    Read tab delimited file and treat each column as a separate data list
readTabDelimitedFileAndTranspose name = do
Transpose before creating the
map this time.
    sheet <- (transpose . map (BStr.split '\t')) .
               BStr.lines `fmap` BStr.readFile name
return is just a function that
returns its argument + the state.
    return $ foldl' go Map.empty sheet
  where go m (x:xs) = Map.insert (BStr.unpack x) xs m
```

[12] Rendered using the example

SPARKLINE.HS

```
module Sparkline where
```
Import Cairo
```
import Graphics.Rendering.Cairo
```
Import our stats function
```
import Stats
```
Remap values from one range (mn-mx) to another. (mn'-mx')
```
clamp :: Double -> Double -> Double -> Double -> Double
      -> Double
clamp mx mn mx' mn' x =
  (x-mn) / (mx-mn) * (mx'-mn') + mn'

renderSparkline values width height = do
```
Save Cairo's state
```
  save
```
Only mx & mn are calculated. Y = nearest pixel prop. to the data range. X=every 2 pixels. Make (x,y) pairs
```
  let (mx,mn,_,_,_,_) = stats values
      yvals = map (clamp mx mn 0 height) values
      xvals = iterate (+width) 0
      (init:coords) = zip xvals yvals
```
Set the pen to black
```
  setSourceRGB 0 0 0
```
Place the pen
```
  moveto (fst init) (snd init)
```
For all pts, draw a line from the prev.to the next. Stroke draws it to the page. Restore the state
```
  mapM_ (uncurry lineto) coords
  stroke
  restore
```

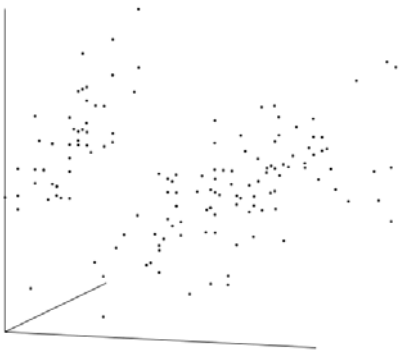
MAIN.HS

Import Cairo
```
import Graphics.Rendering.Cairo
```
Import our data handler
```
import StructuredDataHandler
```
Import the sparkline renderer
```
import Sparkline
```
Import sys.argv
```
import System.Environment (getArgs)
```
Import Map's lookup operator
```
import Data.Map (!)
```
ByteString's string conversion.
```
import Data.ByteString.Lazy.Char8 (unpack)

main = do
```
Bind args to variables, ignore rest
```
  [filename,column,ofilename,sWPV,sHeight] <- getArgs
```
Read datafile
```
  theData <- readTabDelimitedFileAndTranspose filename
```
Read data column into Floats
```
  let values = map (read . unpack) $ theData ! column
```
2 pixels for each value
```
      wpv = read sWPV
      width = wpv * length values
```
Convert commandline height and width to integer.
```
      height = read sHeight

  withImageSurface
```
Allocate a rendering surface
```
    FormatARGB32 width height $ \surf -> do
      renderWith surf
```
Render a sparkline
```
                 (renderSparkline values
                                  (fromIntegral wpv)
                                  (fromIntegral height))
```
Write to the output file.
```
      surfaceWriteToPNG surf ofilename
```

OpenGL Example : 3D Scatterplots



Scatterplots are an excellent way to display large amounts of data. 3D scatterplots can be flown through, and can contain more than just three dimensions of data: individual points can be different sizes, and different colors. We want our scatterplot to be able to handle large amounts of data, and be interactive, so that we can turn the plot around. This is a good simple program that you can do a lot to after you're done with it. You could change coordinate systems and spaces. You could change it from using display lists to using vertex and color arrays to accommodate even more data. Scatterplots are wonderful, versatile things.

StructuredDataHanlder.hs

```
module StructuredDataHandler where

import Data.List (foldl',transpose)
import qualified Data.ByteString.Lazy.Char8 as BStr
import qualified Data.Map as Map

readTabDelimitedFileAndTranspose = do
    sheet <- (transpose . map (BStr.splot '\t')) .
                BStr.lines `fmap` BStr.readFile name
    return $ foldl' go Map.empty sheet
  where go m (x:xs) = Map.insert (BStr.unpack x) xs m
```

Read In a row major formatted dataset with names in the first row.

ProgramState.hs

```
module ProgramState where

import qualified Graphics.Rendering.OpenGL.GL as GL
import qualified Graphics.Rendering.OpenGL.GLU as GL
import qualified Graphics.UI.GLUT as GLUT
import Graphics.Rendering.OpenGL.GL (($=))
import Data.IORef
import Data.Map (Map,(!))

data ProgramState = ProgramState {
    plotstats :: Map String
      (Float,Float,Float,Float,Float,Float)
  , datapoints :: [GL.Vertex3 Float]
  , axes :: (String,String,String)
  , camerarotation :: (Float,Float,Float)
  , renderer :: IORef ProgramState -> IO ()
  }
```

The statistics from our stats fn

OpenGL Vertices for the data

The names of the axes

The angle of the camera

Render function.

Event handler for up, down, left, right buttons as well as – and = for zoom in and out.

```
keyboardMouseCallback :: IORef ProgramState ->
  GLUT.KeyboardMouseCallback
keyboardMouseCallback
  state (GLUT.SpecialKey GLUT.KeyUp) GLUT.Down _ _ = do
    st <- readIORef state
    let (x,y,z) = camerarotation st
    state `writeIORef` st{ camerarotation = (x+3,y,z) }
    renderer st $ state
keyboardMouseCallback
  state (GLUT.SpecialKey GLUT.KeyDown) GLUT.Down _ _ = do
    st <- readIORef state
    let (x,y,z) = camerarotation st
    state `writeIORef` st{ camerarotation = (x-3,y,z) }
    renderer st $ state
keyboardMouseCallback
  state (GLUT.SpecialKey GLUT.KeyRight) GLUT.Down _ _= do
    st <- readIORef state
    let (x,y,z) = camerarotation st
    state `writeIORef` st{ camerarotation = (x,y+3,z) }
    renderer st $ state
keyboardMouseCallback
  state (GLUT.SpecialKey GLUT.KeyLeft) GLUT.Down _ _ = do
    st <- readIORef state
    let (x,y,z) = camerarotation st
    state `writeIORef` st{ camerarotation = (x,y-3,z) }
    renderer st $ state
keyboardMouseCallback _ _ _ _ _ = return ()
```

Handle up rotation. *Read state* *Get the current camera rotation.* *Adjust the x rotation 10 degrees.* *Render.* *Handle down rotation* *Adjust x by -10 degrees* *Handle right rotation* *Adjust y by 10 degrees* *Handle Left rotation* *Adjust y by -10 degrees*

MAIN.HS

```
import qualified Graphics.Rendering.OpenGL as GL
import qualified Graphics.UI.GLUT as GLUT
import Data.Map ((!),fromList)
import Graphics.Rendering.OpenGL.GL (($=))
import ProgramState
import StructuredDataHandler
import Data.IORef
import Stats
import Data.ByteString.Lazy.Char8 (unpack)

rotx = GL.Vector3 1 0 0 :: GL.Vector3 Float
roty = GL.Vector3 0 1 0 :: GL.Vector3 Float

white = GL.Color4 1 1 1 1 :: GL.Color4 GL.GLclampf
black = GL.Color4 0 0 0 1 :: GL.Color4 GL.GLclampf

render state = do
  st@(ProgramState plotstats
                   datapoints
                   (xaxis,yaxis,zaxis)
                   (xdeg,ydeg,zdeg)
                   displaylist _) <- readIORef state
```

X rotation vector for GL.rotate *Y rotation vector for GL.rotate* *White* *Black*

Set the clear color
Enable blending and smoothing
Almost always use this ->
Smooth lines
Smooth points
Do sane things with overlaps
Clear the buffer
Set up the projection matrix

```
  GL.clearColor $= white
  GL.blend $= GL.Enabled
  GL.blendFunc $= (GL.SrcAlpha, GL.OneMinusSrcAlpha)
  GL.lineSmooth $= GL.Enabled
  GL.pointSmooth $= GL.Enabled
  GL.depthFunc $= Just GL.Less
  GL.clear [GL.ColorBuffer, GL.DepthBuffer]
  GL.matrixMode $= GL.Projection
  GL.loadIdentity
```

We want an orthographic projection

```
  GL.ortho (-1.2) 1.2 (-1.2) 1.2 (-1.2) 1.2
```

Setup the modelview matrix

```
  GL.matrixMode $= GL.Modelview 0
  GL.loadIdentity
```

Center the plot around the origin

```
  GL.translate (GL.Vector3 (-0.5) (-0.5) (-0.5) ::
                  GL.Vector3 Float)
```

Rotate the plot for the camera

```
  GL.rotate xdeg rotx
  GL.rotate ydeg roty
```

Set the drawing color
Make our points visible
Make our lines thin
Render the datapoints

```
  GL.color black
  GL.pointSize $= 3
  GL.lineWidth $= 1
  GL.renderPrimitive GL.Points $ mapM_
    GL.vertex datapoints
```

Render origin lines for the 1st quadrant.

```
  GL.renderPrimitive GL.Lines $ do
    GL.vertex (GL.Vertex3 0 0 0 :: GL.Vertex3 Float)
    GL.vertex (GL.Vertex3 1 0 0 :: GL.Vertex3 Float)
    GL.vertex (GL.Vertex3 0 0 0 :: GL.Vertex3 Float)
    GL.vertex (GL.Vertex3 0 1 0 :: GL.Vertex3 Float)
    GL.vertex (GL.Vertex3 0 0 1 :: GL.Vertex3 Float)
    GL.vertex (GL.Vertex3 0 0 0 :: GL.Vertex3 Float)
```

Flush OpenGL calls
Show our drawing on the screen.

```
  GL.flush
  GLUT.swapBuffers
```

Define function to take three equal-length lists and make vertices.

```
vertices :: [Float] -> [Float] -> [Float]
         -> [GL.Vertex3 Float]
vertices (x:xs) (y:ys) (z:zs) =
  GL.Vertex3 x y z : vertices xs ys zs
```

Handle unequal lengths

```
vertices [] _ _ = []
```

…

```
vertices _ [] _ = []
```

…

```
vertices _ _ [] = []
```

Clamp values in [0,1]

```
clamp mn mx v = (v-mn) / (mx-mn)
```

Program starts executing here.

```
main = do
```

Set display mode to something standard.

```
  GLUT.initialDisplayMode $= [GLUT.RGBAMode
                             ,GLUT.Multisampling
                             ,GLUT.DoubleBuffered
                             ,GLUT.WithAlphaComponent]
```

Set the default window size.

```
  GLUT.initialWindowSize $= GL.Size 1000 1000
  (progname, args) <- GLUT.getArgsAndInitialize
```

Read a tab delimited file
Column name for x axis
Column name for y axis
Column name for z axis
Initial rotation is 0
Read the datafile

Make vertices from the data within [0,1] on all axes.

Get the x data from the map
Get the y data from the map
Get the z data from the map
Calculate X axis stats
Calculate Y axis stats
Calculate Z axis stats
Create map of stats

Setup the program state

Create a window
Set the display callback
Set the event handler

Go.

```
let filename = args !! 0
    xaxis = args !! 1
    yaxis = args !! 2
    zaxis = args !! 3
    rotate = (0,0,0)
dat <- readTabDelimitedFileAndTranspose filename

let points = vertices (map (clamp mnx mxx) xdata)
                      (map (clamp mny mxy) ydata)
                      (map (clamp mnz mxz) zdata)
    xdata = map (read . unpack) $ dat ! xaxis
    ydata = map (read . unpack) $ dat ! yaxis
    zdata = map (read . unpack) $ dat ! zaxis
    xstats@(mxx,mnx,_,_,_,_) = stats xdata
    ystats@(mxy,mny,_,_,_,_) = stats ydata
    zstats@(mxz,mnz,_,_,_,_) = stats zdata
    plstats = fromList [(xaxis,xstats)
                       ,(yaxis,ystats)
                       ,(zaxis,zstats)]
state <- newIORef $ ProgramState
  plstats
  points
  (xaxis,yaxis,zaxis)
  rotate
  render

GLUT.createWindow "3D scatterplot"
GLUT.displayCallback $= render state
GLUT.keyboardMouseCallback $=
   Just (keyboardMouseCallback state)
GLUT.mainLoop
```

An OpenGL visualization boilerplate

Main.hs

```
import qualified Graphics.Rendering.OpenGL.GL as GL
import qualified Graphics.Rendering.OpenGL.GLU as GL
import qualified Graphics.UI.GLUT as GLUT
import Graphics.Rendering.OpenGL.GL ($=)
import ProgramState
import StructuredDataHandler
import Data.IORef

render state = do
                                                    Read program state
  ProgramState … mouseposn <- readIORef state
                                                    Set up state variables
  GL.clearColor $= GL.Color4 0 0 0 1 :: GL.Color4 Float
  GL.blend $= GL.Enabled
  GL.lineSmooth $= GL.Enabled
  GL.pointSmooth $= GL.Enabled
  GL.polygonSmooth $= GL.Enabled
                                                    Clear screen
  GL.clear [GL.ColorBuffer, GL.DepthBuffer]
                                                    Set up matrix
  GL.matrixMode $= GL.Projection
  GL.loadIdentity

        Find out the window size.  The @ sign binds the whole pattern to vport, while the individual pieces bind to x, y, xres, and yres
  vport@(GL.Position x y, GL.Size xres yres) <- GL.get GL.viewport

                                    If render has been called to determine mouse selection, render only under the mouse
  mode <- GL.get GL.renderMode
  if rendermode == GL.Select
    then let Just (GL.Position mx my) = mouseposn
             pickx = fromIntegral mx
             picky = fromIntegral my + fromIntegral y
             area = (2,2) in
               GL.pickMatrix pickx picky area vport
    else return ()

                                                    Set up GL coordinates = window coordinates
  GL.ortho2d 0 0 (fromIntegral xres) (fromIntegral yres)

                                                    Set up the modelview matrix
  GL.matrixMode $= GL.ModelView 0
  GL.loadIdentity

                                                    Check again to see if we're using render mode or select mod
  if mode == GL.render then do
        … render stuff as if it will be displayed
  else do
        … render only stuff that is selectable
  GL.flush
  GLUT.swapBuffers
```

```
main = do
  (progname, args) <- GLUT.getArgsAndInit
  GL.Size xres yres <- GL.get GLUT.screenSize
  GLUT.createWindow "WindowName" 1000 1000
```
Set up the program state below, followed by the display and event handler callbacks. Before this, read in and create your data structure.
```
  state <- newIORef $ ProgramState …
  GLUT.displayCallback $= render state
  GLUT.keyboardMouseCallback $= Just (keyboardMouseCallback state)
  GLUT.mainLoop
```

PROGRAMSTATE.HS

Module name must be the same as the filename.
```
module ProgramState where

import qualified Graphics.Rendering.OpenGL.GL as GL
import qualified Graphics.Rendering.OpenGL.GLU as GL
import qualified Graphics.UI.GLUT as GLUT
import Graphics.Rendering.OpenGL.GL ($=)
import Data.IORef
```
Create program state record type
```
data ProgramState = ProgramState {
  …
  , renderer :: IO ()
  }
```
Create keyboard/mouse motion callback
```
keyboardMouseCallback :: IORef ProgramState -> GLUT.KeyboardMouseCallback
```
Handle keyboard chars
```
keyboardMouseCallback state (GLUT.Char c) GLUT.Down
                      (GLUT.Modifiers shift ctrl alt) (GL.Position x y) =do
  st <- readIORef state
```
… do stuff
```
  state `writeIORef` st{ … state changes … }
  renderer st $ state
```
Handle left click
```
keyboardMouseCallback state (GLUT.MouseButton GLUT.LeftButton) GLUT.Down
                      (GLUT.Modifiers shift ctrl alt) (GL.Position x y) =do
  st <- readIORef state
```
… do stuff
```
  state `writeIORef` st{ … state changes … }
  renderer st $ state
```
Handle right click
```
keyboardMouseCallback state (GLUT.MouseButton GLUT.RightButton) GLUT.Down
                      (GLUT.Modifiers shift ctrl alt) (GL.Position x y) =do
  st <- readIORef state
```
… do stuff
```
  state `writeIORef` st{ … state changes … }
  renderer st $ state
```
Default do-nothing case.
```
keyboardMouseCallback _ _ _ _ _ = return ()
```

StructuredDataHandler.hs

```
module StructuredDataHandler where

import Data.List (foldl')
import qualified Data.ByteString.Lazy.Char8 as BStr
import qualified Data.Map as Map

                                    Red in a column-major formatted dataset with names in the first column
readTabDelimitedFile name = do
    sheet <- (map (BStr.split '\t') . BStr.lines) `fmap`
             BStr.readFile name
    return $ foldl' go Map.empty sheet
  where go m (x:xs) = Map.insert (BStr.unpack x) xs m

                                    Read in a row-major format dataset with names in the first row
readTabDelimitedFileAndTranspose = do
    sheet <- (transpose . map (BStr.splot '\t') . BStr.lines `fmap`
             BStr.readFile name
    return $ foldl' go Map.empty sheet
  where go m (x:xs) = Map.insert (BStr.unpack x) xs m

                                    Create some data structures for your data here.
data DataStructure = …

                                    Some single entrypoint for taking the map of names to data lists and making the data structure you just defined.
buildDatastructure :: Map String [BStr.ByteString] -> DataStructure
buildDatastructure = …

                                    Convenience function for reading the data structure from file.
readDatastructureFromFile filename = do
  dat <- readTabDelimitedFile filename
  return . buildDatastructure $ dat
```

A Cairo Visualization Boilerplate

StructuredDataHandler.hs

```
module StructuredDataHandler where

import Data.List (foldl')
import qualified Data.ByteString.Lazy.Char8 as BStr
import qualified Data.Map as Map
```

Read tab delimited file and treat each column as a separate data list

```
readTabDelimitedFileAndTranspose name = do
    sheet <- (transpose . map (BStr.splot '\t') .
               BStr.lines `fmap` BStr.readFile name
    return $ foldl' go Map.empty sheet
  where go m (x:xs) = Map.insert (BStr.unpack x) xs m
```

Geometry.hs

```
module Geometry where
import Graphics.Rendering.Cairo
import Stats

render = do
  save
  setSourceRGBA 0 0 0 1
  moveto 0 0
  …
  stroke
  restore
```

Main.hs

```
import Graphics.Rendering.Cairo
import StructuredDataHandler
import Geometry
import System.Environment (getArgs)
import Data.Map (!)
import Data.ByteString.Lazy.Char8 (unpack)

main = do
  [filename,ofilename~] <- getArgs
  theData <- readTabDelimitedFileAndTranspose filename

  withImageSurface
    FormatARGB32 1000 1000 $ \surf -> do
      renderWith surf render
      surfaceWriteToPNG surf ofilename
```

Copyright, Credits & Acknowledgements

The image on the cover is of a hierarchical cluster of the Gene Ontology conducted as part of a research study with Dr. Xiaojun Guan of the Renaissance Computing Institute and Dr. William Kaufmann of the University of North Carolina at Chapel Hill. The original publication of the visualization is:

> Jefferson Heard, Xiaojun Guan. 2008. Functional Visualization of Large Binary Trees in Haskell. *International Conference on Functional Programming (ICFP2008)*. Vancouver, BC. To appear.

Document layout and fonts inspired by Edward Tufte's excellent books: *Beautiful Evidence*, *Visual Explanations*, *The Visual Display of Quantitative Information*, and *Envisioning Information*.

I would like to thank Paul Brown for debugging my Sparkline code, to Don Stewart for a long time ago showing me how to read data lazily, and to Ray Idaszak, Jason Coposky, Sarat Kocherlakota, Theresa-Marie Rhyne, and rest of the visualization group at the Renaissance Computing Institute for making me into a visualization programmer.