Graphics Report – 100265937 – Thomas Rogers

**Introduction:**

Computer Graphics and data visualization are two inextricably linked concepts, the process of visualizing data on a computer requires us to render out primitives, linking them to unique data points in our datum.

The benefits of this type of interactive data visualization is two-fold, first, the speed at which modern computers operate allow us to visualize data much faster than by hand, additionally, the interactions we can perform can be done much quicker and more efficiently. These interactions vary from simplistic, such as being able to pan and move the graphs to more complex interactions, such as being able to highlight and change data. The advantages of this are simple over traditional methods of data visualization, it allows us to ask questions like “What if?” and allow us to simply and effectively alter the data.

Despite this, there are some technological challenges in creating a data visualizer. There are some assumptions we have to make about our data that could be much easier to make by hand, and there are some computational requirements we must abide by – The need to store radically different primitives, for instance, for each chart.

This report intends to go over the work I have performed and the reasons for the decisions I made. I will go over the methods that I used, evaluation of these methods, and have a discussion of the strengths and weaknesses of the work I undertook.

**Methods:**

A good place to start with my implementation is the front-end, namely the menu system implemented. I created this menu system to ensure that users of the program would be able to interact, add, and organise charts. It’s dynamic based on how many charts are present, and whether data has been loaded into memory. The menu system allows users to create an OpenFileDialog.

This OpenFileDialog is abstracted out into its own class, this was to ensure that it could be used in other applications, and to open other files other than CSV. While this feature was never necessary, I still believe it to be an important principle to adhere by, as it allows us to build on top of the application later. This OpenFileDialog returns a wpath object, which is passed into the CSV parser. By returning a path we ensure that any data passed to other classes can be handled by that class, further ensuring the abstract nature of the program.

The menu system further allows you to add Charts, which are stored as a vector of pointers in the OGLWindow class. These charts derive from a base class, OGLChart, which holds a rendering Method. This method is used by all the classes, and the charts are created by an overridden method known as InitSources. This method is in charge of filling a Map of OGLShapes\* and DataCell, which is rendered out in the Base class. This object oriented approach to rendering is something that also held over to the Shape class.

Each shape is rendered out in the base class, which cycles through a list of vertexs that are created by the derived class. The Shape class allows us to pass in an Unsigned\_Int, which determines what type of rendering gl\_begin() will use. This also allows us to create polygonal shapes in the base class, which allows us to create a spiderplot with the user of generalised shapes.

By having shapes be an integral and permanent member of the Chart class we can also use them to detect mouse over and other interactions. These are performed two fold, one of which is a binding box, a simple and non-intensive way to detect if the mouse is near the object, then via a line intersections test. These interactions are the glue that binds the application together. By allowing us to detect whether a mouse is over we are able to highlight specific shapes and use them to create strings based on datacells. This forms the basis of the highlighting information in my application.

The DataCells are classes that can store any type of information, based on the <boost> any class. It Decays the information from a type that you pass in, then casts it to a pointer. It later uses the C++11 auto keyword in order to determine the type of the object. This is done by dynamically casting the object out as the type, and determining if the pointer is Not Null. This allows us to store any object. The CSVParser converts a string from a csv into a DataCell. The CSV parser also uses heuristics to determine whether or not there was a header, and if there is, we pass it as a name to the DataColumn. All of these are then stored in a datatable, which is used to fill the Data part of the Menu allowing us to pass information to the charts.

The scaling and translating aspects of my program mostly used Vector operations in order to change the shape and size of the shapes – This ensures that any function like mouse inside would still remain compliant. However, there are some uses of the glScale, Rotate and Translate functions. To perform this I created my own Matrix class which defined their operations and allowed public access to the data. These would then fit in the glMultMatrix function.

Text is created with GLFont module created by Brad Fish (Brad Fish, 2001). It creates textured primitives that use a glf file and determines the position based on texture size and location of the letter in the texture.

Additional to this, a colour struct was created that stores 4 members, Red Green Blue and Alpha. The alpha property allows me to render multiple charts out on screen at once and composite them over each other.

**Evaluation, Discussion and Result**

At first, the application used a vector of vectors to allocate the data. This proved very inefficient, requiring a lot of memory allocation and copy. To alleviate this I did a number of things, first, I ran a heuristic on the data in the csv parser to determine size. While this is initially a very costly procedure, it is much cheaper than the constant copy and memory allocation of reallocating vectors. We use this heuristic to pre-allocate the vectors. This means that there is no need for copy. Further to this I also defined the Move method in the Data Cell class, which meant that rather than copying the data each time it would merely be moved to a new area of memory rather than copied each time. While allocating a vector manually does call the default constructor, I have found that the reserve function in the vector class does allocate the space, but doesn’t actually construct the object, merely allocates the room for it (Jan Hudec, 2011).

While storing shapes in the chart class may at first seem less efficient than simply rendering shapes out each time, it is important to note that the trade-off is what allows more interactions to be possible. Additionally to this, it allows us to defer our rendering to the actual base class, storing pointers to OGLShapes in the base class. This allows us to define behaviour and avoid code reuse.

Another performance concern I had was the ability to render out multiple charts at once, while this in theory does not seem too intensive, there will be a lot of primitives on the screen at once. While this was an initial concern, it didn’t seem to slow down too much – and depending on the data shown I found that there were limited slowdowns. I considered not rendering out the information that was neither in the viewport, or overlapped by another chart. However, since I later added the ability to view graphs over each with opacity I decided against it, because it would limit the ability to view multiple charts.

While I believe most of my methods to be successful, and generally a good way to deal with the problem of data visualization, there were some issues that I encountered.   
 First, I would like to address my concerns about passing data, there is a lot of copy of vectors and other such data structures. I did address this by passing Pointers whenever possible, but found that this solution was not always viable. Additionally to this I also defined the move constructor whenever it was necessary to avoid frequent copying amongst classes.

The object oriented nature of my program also helps ensure that there it is extendable in the future; all the charts come from one OGLChart class, and all the shapes inherit from the OGLShape class. This means that creating a new shape or chart is a simple procedure, and doesn’t (often) require us to write new code for the render/move etc.

However, this provided its own problems - When moving to more complex Graph types such as the 3d Scatter plot plotting simple 2d points was no longer sufficient – This required me to abstract out my Shape class into a 2d and 3d class, a 3d class would allow for a z axis to be added. Further to this, the 3d Scatterplot required me to translate differently than the other graphs; I used a frustum to create a perspective matrix, but moving into the z axis on here meant that the reference point would remain static. To fix this I ensured that the rendering would be done slightly differently; it would instead move the frustum rather than the x and y coordinates of the shapes. This would

**Conclusion**