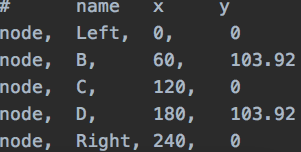
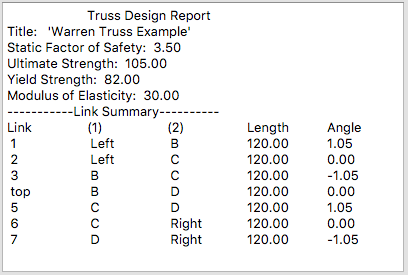
Last week we discussed remedial English in a way. This week, we are discussing remedial art class. We want to draw a figure on our GUI using lines and shapes. We imagine having a set of **(colored) pencils or pens**, **paint brushes**, **paint**, and a **piece of paper or canvas** on which to create our art. Some other tools in our engineering art kit might include a straight edge/ruler for drawing lines of particular length, graph paper with a grid of fixed size for easily locating points on the 2-D paper, a protractor for measuring angles, and a compass for drawing arcs. Animation on the computer is essentially like a flip cartoon book you might have made as a child, where each frame drawn with a slight difference from the previous frame. When you flip the pages, you see the animation happen. This is effectively what a computer does, but with much more intricate images and very fast flipping of the pages.

Let’s do a quick example of sketching a static truss:

***Example***: A Truss (from HW8)



From out previous homework, I might want to sketch this truss so I can see what it looks like. On engineering graph paper, I would sketch it as follows:

Left (0,0)

B (60,103.92)

Right (240,0)

C (120,0)

D (180,103.92)

My steps to create this graphic in Microsoft Word were to:

1. From ribbon, Select - *View→Gridlines* (checked on Show panel) This helped me draw things to scale. I just decided that each grid line represented 20 units.
2. From ribbon, Select – *Insert→Shapes→Drawing Canvas* to create a rectangular region on my document where I can draw the truss. I set the outline to a blue, thin line.
3. From ribbon, Select – *Insert→Shapes→Oval* (draw circles at node points assuming an origin at node ‘Left’ with *y*-axis pointed up, *x*-axis pointed right.) Note: the circles are drawn with a darkish blue outline with a lighter blue fill. The width of the outline can be specified as can the colors of the line and the fill.
4. From the ribbon, *Select – Insert→Shapes→Line* (draw the line segments connecting the nodes according to the truss report). The line segments are blue in color and I can select the width.
5. Finally, I labeled my nodes by *Insert→Shapes→Text Box*.

You would probably not be surprised that in OpenGL (described later), the nodes are called glVertex(0,0), glVertex(120,0), etc. No doubt, if you were to draw the same truss by hand using a box of colored pencils, you could reproduce a reasonable sketch with the nodes represented by shaded blue circles and blue lines and black text. If your colored pencils were mechanical, you might want a range of lead sizes to draw lines of different width to make your art look better.

**LED displays**:

*pixels*

When a computer draws graphics (we will restrict our attention to 2-D drawings on your screen), it is really illuminating (or dimming) a pattern of small light patches (pixels) arranged in a 2-D grid. Granted, the pixels are very small and our human eyes often can’t see this underlying structure of a LED (light emitting diode) display. It turns out that there are actually 3, side-by-side grids of LED’s, one is Red, one is Green, and one is Blue. By controlling brightness of three adjacent Red, Green and Blue LED’s, one may obtain any color for the pixel (remember the color wheel from art class). Our eyes and brain actually do the color blending and we see a continuous spectrum of colors. So, to draw a line on a computer screen, we need to tell the graphics card of the computer exactly which pixels to illuminate and how much.

*coordinates*

Since the array of pixels is inherently 2-dimensional and the physical monitor has a certain number of pixels wide × pixels tall, we can specify a pixel location with two coordinates (*nx*, *ny*). Our intuition (and previous experience) as engineers and students of mathematics is that a logical reference pixel (*origin*) would be the bottom left of the display screen and that we should use a right-handed coordinate system. Of course, we could have a global origin/coordinate system (associated with the physical computer screen) and a floating/local coordinate system (associated with a window, button, group box, etc.) to make it easier to specify points.

**A HUGE POINT OF CONFUSION** for most of us is that the origin for drawing with computers is actually at the top-left of the screen/canvas and the orientation of the *y*-axis is pointed downward while the *x*-axis still points to the right. The reason for this is historical and related to the way original displays (called cathode ray tubes, or CRTs) scanned an electron from top to bottom of a screen to create an image. Like much in computer science, this original coordinate system has persisted and can be a source of frustration for beginners.

The FACT that the display is inherently digital (coordinates are integers) with an upside-down *y*-axis is not insurmountable, because we can estimate floating point coordinates with integers and get good results if the pixels are small. Likewise, a simple transformation of coordinates can be done to ‘flip’ the *y*-axis to specify points on the screen after you have done your calculations.

**OpenGL:**

A good online reference for OpenGL is found here:

<https://docs.microsoft.com/en-us/windows/win32/opengl/opengl>

While we have learned GUI design (manually or with the aid of QTDesigner), we don’t yet have ability to draw things like trusses on our widget. Even if we could pre-draw a sketch of a truss, if we wanted to create a truss design program for exploring the engineering of a variety of trusses, the pre-drawn truss would be inadequate. Rather, we want some way to draw lines, circles, etc. to the screen, be able to interact with them, perhaps animate them, and have great control over exactly how they are drawn.

In comes the Open Graphics Library (OpenGL), which is a library of classes, functions, etc. (a so-called application programming interface, or API) that we can use to draw on our widgets. OpenGL is a vast and complicated API and is capable of the impressive 3-D graphics you see on Playstation video games, Pixar and Marvel movies, etc. We will be far less ambitious and limit our attention to simple line drawings (e.g, truss schematics and pipe networks) to get our engineering points across. Also, OpenGL uses a coordinate system that is more intuitive to us with *y* axis pointed in the intuitive direction of up.

*OpenGL as a movie studio*

OpenGL models the artwork like a Hollywood film studio. The actors and props are at specific locations in the scene. The camera(s) can film the scene from various directions and pan, tilt, rotate and zoom for artistic effect. The scene can be lighted with spotlights, ambient light, flood lights, and candles (point lights). The audience (or director) sees the movie or scene on a screen (or viewport) and the scene depends on the location of the props and actors, the camera and the lights.

This may seem very complicated for the simple line drawings we want to do (and it is). It’s like having a full movie studio filming a 1st grade doodle. Still, we (the director) have to give explicit instructions about where to position the camera, how to light the scene, and the details of the drawing.

***General approach for using OpenGL in Python/PyQT***:

So, here is how we will use OpenGL:

1. Create a GUI (QTDesigner or manual) that includes a QOpenGLWidget
2. In Python code, create the application class. Use the following imports for enabling all the OpenGL stuff:

# standard OpenGL imports

import OpenGL.GL as GL

import OpenGL.GLU as GLU

import OpenGL.GLUT as GLUT

from OpenGL\_2D\_class import gl2D, gl2DText, gl2DCircle

#note this last import is a custom class created to greatly simplify opengl.

1. Create a window object from the gl2D class.
2. Create a DrawingCallback function that contains the geometry of the things you want to draw.

The *gl2D* class handles the setup of the camera (i.e., where to focus the camera and how much to zoom in on the artwork), the setup of the viewport (a rectangle defined by the QOpenGLWidget), and handles the refreshing of the viewport when necessary (such as when we add a prop or interact with a prop using the mouse). The inner workings of the *gl2D* class are open for inspection by you, but you won’t be held responsible for understanding them.

The *gl2DText* and *gl2DCircle* are functions that make drawing lines, text and circles simple. Again, our goal is to just use them rather than understand their inner workings.

***The DrawingCallback*** *- What you are responsible to know*

The work in MAE 3403 is to understand the *DrawingCallback* function. First of all, the term *callback*, refers to a function that is passed to another function. We have used callbacks quite a bit so far (e.g., in using fsolve and odeint), but we just called the passed function a function rather than a callback function.

*vertices* -

Continuing our example of a truss, the node locations are specific points in our scene and these are known to OpenGL as vertices. Vertices can be in 3-D space or 2-D space. We are working only on a 2-D plane, so we will use *glVertex2f* to specify *x,y* coordinates with floating point numbers.

*primitives –* <https://docs.microsoft.com/en-us/windows/win32/opengl/glbegin>

Primitives are what we would call lines (segments) and circles in our truss drawing. By hand, we draw the lines by connecting the nodes of the truss. In a similar way, OpenGL draws primitives (OpenGL jargon) by issuing the command: glBegin, followed by a list of vertices, followed by glEnd. For example:

glColor(0,1,0) #specify that the lines will be green (Red, Green, Blue)

glLineWidth(2) #specify that the lines have width=2

glBegin(GL\_LINES) #begin drawing lines between successive pairs of points

glVertex2f(0.0,0.0)

glVertex2f(60.0,103.92)

glEnd() #end of drawing lines

So, this specifies all the instructions that OpenGL needs to know to calculate exactly which pixels on your screen to color green and make a line.

Now, we will do a more advance picture in Python. (see video)