What is OpenGL?

- **OpenGL (Open Graphics Library)** is a cross-platform industry standard for graphics (including 3D graphics).
- It is built using C, but has been translated to work in most any language, and also on both Apple and Windows.
The Problem...

• 3D graphics are not a native part of Visual Studio 2005

• Later versions of Visual Studio (2008 +) support 3D graphics natively in the WPF framework, but for the purposes of this class we will explore a method compatible with VS 2005

The Solution...Use **OpenGL**

• OpenGL is open source, and many different libraries have emerged with **bindings** that hook into different languages, including C#

• **Tao** is one of the most comprehensive library frameworks that provides these OpenGL bindings to C#
Starting your project

1. We suggest that to use our template code uploaded in the guild bank.
2. Everything should be set up and ready to draw.
3. Make sure Tao.OpenGL.dll, Tao.Platform.Windows.dll and Tao.FreeGlut.dll are in same folder as your executable (If not add it manually - to references)
4. Even though the references are there, we will need to remove it and add the reference again. The files are located with the executable
5. Write codes in the draw section that we commented out for you!

Caution: You might need to copy the 4 .dll files into your system 32 folder. C:\Windows\System32

The 3D Coordinate System

- Recall that in 2D graphics, the origin is ALWAYS the upper left corner
- Unlike 2D Graphics, 3D graphics revolve around the origin of the coordinate system.
Camera System

• In 3D coordinate system, the 3D scene looks different depending on the **point of view**. We use the **camera system** to specify our point of view.
• By default, the camera lies on the **positive Z axis** looking down at the origin (towards the negative Z axis).

Points and Vectors in 3D Graphics

• In 3D graphics, Vectors and Points are represented as column matrices. The fourth (bottom) element of the matrix corresponds to whether it is a Point or a Vector.

\[
P = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ 1 \end{bmatrix} \quad V = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ 0 \end{bmatrix}
\]

Point at origin \((0,0,0)\) = \[
\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}
\]
Transformations

- When an object is drawn on the screen, it is drawn at the origin (0,0,0) unless we specify differently. To move/draw the object to another location, we need to transform it.
- There are three ways to transform an object:
  - Translation
  - Scaling
  - Rotation

Transformations

- Translation
  - `gl.glTranslatef(float x, float y, float z);`
- Scaling
  - `gl.glScalef(float x, float y, float z);`
- Rotation
  - `gl.glRotatef(float angle, float x, float y, float z);`

Note:

⚠️ You must do all transformations before drawing the object!!
Translation in OpenGL

In homogeneous coordinates: \( P' = P + d \), where \( d \) is the translational displacement.

In matrix form: \( P' = T(d) \cdot P \), where \( P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \), \( d = \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \), \( T(d) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \).

\[ P' = T(d) \cdot P = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{bmatrix}. \]

The inverse of \( T(d) \) is \( T(-d) \).

Each transformation is simply one matrix multiplication operation.
Thus, it is fast and efficient.

Transformations - Translation

- \( \text{Gl.glTranslatef}(1.0f, -1.0f, 2.0f); \)
  - translates \(+1\) unit on the \( x \)-axis
  - translates \(-1\) unit on the \( y \)-axis
  - translates \(2\) units in the \( z \)-axis
Scaling in OpenGL

- Scaling an object means changing the size of the object.
- From the last slide, we have an equation to find the new location of the object, we can replace the following matrix with the translation matrix.

\[
S(\alpha_x, \alpha_y, \alpha_z) = \begin{bmatrix}
\alpha_x & 0 & 0 & 0 \\
0 & \alpha_y & 0 & 0 \\
0 & 0 & \alpha_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- If alpha is between 0 and 1, we are shrinking the object.
- If alpha is bigger than 1, we are enlarging the object.

Transformations - Scaling

- `Gl.glScalef(2.0F, 1.0f, 1.0f);`
  // double the length of the object in the X direction while keeping the Y and Z length the same.

- `Gl.glScalef(1.0f, -2.0f, 1.0f);`
  //What would this do?
  //This keeps the X and Z axis the same length, and flip the image around an doubling the size in the Y direction

BE CAREFUL! Having a scale of 0.0f would shrink the size to 0 in the corresponding direction. In another word, it would disappear in the direction of the 0's.
Rotation in OpenGL

• The concept of rotation is pretty much same as the previous transformation, except:
  o To rotate in the Z direction, OpenGL uses the Rz matrix.
  o To rotate in the Y direction, OpenGL uses the Ry matrix.
  o To rotate in the X direction, OpenGL uses the Rx matrix.

$$R_x(\theta) = \begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 
\end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1 
\end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 
\end{bmatrix}$$

$$R = R_z R_y R_x$$

Transformations - Rotation

• `Gl.glRotatef(45.0f, 1.0f, 0.0f, 0.0f);`
  // this would rotate the graphic 45 degrees around the x axis
• `Gl.glRotatef(90.0f, 1.0f, 0.0f, 1.0f);`
  // this would rotate the graphic 90 degrees around the x and z axis

• The first parameter is the degree of rotation, and the next three parameters specify the direction of the axis to rotate around (using the x, y, and z axises respectively).
• A 1.0f in an x, y, or z parameter means you will rotate around at least that axis

• **Note:** you must apply all transformations before drawing the object.
Demonstration

- We found and modified some code online that allows you to rotate an object with the J, K, and L keys on your keyboard and translate with the arrow keys:

- Refer to the Guild Bank for our template code and try it yourself!

- As you can see, it is difficult to orient the object exactly in the way meant. However, remember that you are NOT moving the object, but the point at which you are viewing from.

using GLUT library to draw 3D shapes

- There are 2 different ways to draw shapes using the template program

- The first way is to use the GLUT library's predefined shape functions. Call the function and it draws to the screen in one step.

  ```c
  Glut.glutSolidCube(double size);
  Glut.glutTeaPot(double size);
  Glut.glutSolidSphere(double radius, )
  //and many more
  ```

  This is usually a very quick way to draw a shape, but very limiting in terms of customization (unless you have an affinity to teapots)
Drawing 3D shapes - manually (most common)

Guidelines for drawing your own shapes
• Every face of the 3D shape you draw is drawn separately
• For each face:
  o 1. Specify the color of the shape
  o 2. Specify the points \((x,y,z)\) for all vertices of that shape

Draw one face of a 3d object

• Every face of the 3d object is drawn separately include a begin (glBegin) and an end (glEnd) statement.
  
  \[
  \text{Gl.glBegin}(\text{int mode});
  \]
  // define vertices of the shape
  
  \[
  \text{Gl.glEnd}();
  \]
• the mode is an integer value that specifies the type of object you will draw
• intellisense will give you a list of mode options.
• The most common values for mode are
  
  \[
  \text{GL.GL_POINTS}
  \]
  
  \[
  \text{GL.GL_LINES}
  \]
  
  \[
  \text{GL.GL_TRIANGLES}
  \]
  
  \[
  \text{GL.GL_POLYGON}
  \]
How to specify a vertex

\texttt{Gl.glVertex3f(float x, float y, float z)}

- \textit{f} means you are using floating point coordinates
- this is a standard for plotting these shapes

-the 3 means it takes 3 parameters in the coordinate system
(it's a 3d shape)

Coloring

\texttt{Gl.glColor3f(float red, float green float blue);}

- each float is a value between 0 and 1.
- A normal RGB scale is ranged from 0 to 255.
- If you find a RGB in the 255-RGB scale, simply divide that number by 255 and you can apply it to OpenGL.
Example of Coloring:

```cpp
Gl.glColor3f(float red, float green, float blue);

Gl.glColor3f(1.0f, 1.0f, 0.0f);
// this is yellow

Gl.glColor3f(1.0f, 0.5f, 0.0f);
// this is orange ish
// RGB is (255,165,0)

Gl.glColor3f(0.0f, 0.5f, 1.0f);
// light blue ish
```

General format for drawing a polygon

```cpp
// face 1
Gl.glBegin(Gl.GL_POLYGON)
    // define all color and the vertices of that face
    Gl.glEnd();

// face 2
Gl.glBegin(Gl.GL_POLYGON)
    // define all color and the vertices of that face
    Gl.glEnd();

// face 3........... continue for all faces of the shape
```
Example: Drawing a pyramid

```c
Gl.glBegin(Gl.GL_POLYGON); //base
    Gl.glColor3f(0.0f, 0.0f, 1.0f);
    //give it color (blue)
    //Tell OpenGL the vertices of your polygon
    Gl.glVertex3f(1.0f, 1.0f, 0.0f);
    Gl.glVertex3f(1.0f, -1.0f, 0.0f);
    Gl.glVertex3f(-1.0f, -1.0f, 0.0f);
    Gl.glVertex3f(-1.0f, 1.0f, 0.0f);
Gl.glEnd();
Gl.glBegin(Gl.GL_POLYGON); //one of the sides
    Gl.glColor3f(0.0f, 1.0f, 1.0f);
    Gl.glVertex3f(1.0f, 1.0f, 0.0f);
    Gl.glVertex3f(-1.0f, 1.0f, 0.0f);
    Gl.glVertex3f(0.0f, 0.0f, 5.0f);
Gl.glEnd();
```

The order of vertices matters! It would draw from the top to bottom the way you specify it.

And........Here is the result!
What happens when the vertices are in the wrong order?

```cpp
Gl.glBegin(Gl.GL_POLYGON); //base
Gl.glColor3f(0.0f, 0.0f, 1.0f);
Gl.glVertex3f(1.0f, 1.0f, 0.0f);
Gl.glVertex3f(-1.0f, -1.0f, 0.0f);
Gl.glVertex3f(1.0f, -1.0f, 0.0f);
Gl.glVertex3f(-1.0f, 1.0f, 0.0f);
Gl.glEnd();
```

OpenGL draws the vertices from top to bottom and connects them in the same order they were drawn.

---

Drawing multiple objects

**Example 1**
```cpp
Gl.glBegin(Gl.GL_POLYGON)
Gl.glTranslatef(x,y,z);
//code for vertices
Gl.glEnd();
Gl.glBegin(Gl.GL_POLYGON)
//code for vertices
Gl.glEnd();
```

**Example 2**
```cpp
Gl.glPushMatrix();
Gl.glBegin(Gl.GL_POLYGON)
Gl.glTranslatef(x,y,z);
//code for vertices
Gl.glEnd();
Gl.glPopMatrix();
Gl.glBegin(Gl.GL_POLYGON)
//code for vertices
Gl.glEnd();
```

- What is the difference between the two codes? Assuming they are drawing the same polygons.
- Any idea on what PushMatrix and PopMatrix do?
Example 1's outcome

Example 2....Much better
Matrix Stack:

`Gl.glPushMatrix()` and `Gl.glPopMatrix()` isolate the transformations so whatever we call after the PopMatrix will not be affected by the transformation inside the Push and Pop Matrix.

This will be helpful in most any transformation, particularly in projects which contain many 3-D objects.

Once the PopMatrix method is called, any further operation will change your entire project. If you call the Push operations again, you will be able to create a new set of transformations for a whole new set of objects.

Visual Interpretation

- Think of each Push operation as creating a new, temporary coordinate system.
- We perform Transformation1, Object1 would be drawn at the origin of the translated coordinate system.

```
Gl.glPushMatrix();
Gl.glBegin(mode);
Gl.translatef(2.0f, 1.0f, 0.0f);
//draw something
Gl.glEnd();
Gl.glPopMatrix();
Gl.glBegin(mode);
//draw something
Gl.glEnd();
```

```
\begin{bmatrix}
1 & 0 & 0 & T_x \\
0 & 1 & 0 & T_y \\
0 & 0 & 1 & T_z \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
```

Translation Matrix

Translated Coordinate System

```c
Gl.glPushMatrix();
Gl.glBegin(mode);
Gl.translatef(2.0f, 1.0f, 0.0f);
//draw something
Gl.glEnd();
Gl.glPopMatrix();
Gl.glBegin(mode);
//draw something
Gl.glEnd();
```
With all of that, we can draw simple 3D objects, move them around, scale and rotate them any way we want them!

Now let's look at a few demos of 3D objects!

Snowman!

Created by using combinations of sphere, cylinders and cones.
More advanced topics with OpenGL: (will not cover)

- Ray-Tracing
  - technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects.

- Texture Mapping
  - method for adding detail, surface texture, or color to a computer-generated graphic or 3D model.

- And much more...

References

- Tao.Framework Tutorial for 2D Objects
- OpenGL 3D Navigation
- OpenGL Tutorial
- OpenGL Website
- OpenTK (Alternative to Tao Framework)

Link to download Tao Framework:
- http://sourceforge.net/projects/taoframework/

Zipped file of Tao Framework should be in the guild bank.