Today’s class

- Viewing transformation
- Menus
- Mandelbrot set and pixel drawing
The world & the window

- World coordinates describe the coordinate system used to model our scene
- Let the world coordinates obey the following relationships:
  - \( wx_{\text{min}} \leq wx \leq wx_{\text{max}} \)
  - \( wy_{\text{min}} \leq wy \leq wy_{\text{max}} \)
- The OpenGL code to set the world coordinates is:
  - \( \text{glMatrixMode (GL_PROJECTION);} \)
  - \( \text{glLoadIdentity ();} \)
  - \( \text{gluOrtho2D (wx_{\text{min}}, wx_{\text{max}}, wy_{\text{min}}, wy_{\text{max}});} \)
- This code sets a window through which we view the scene
The viewport

- The viewport defines the physical part of the display window that the image will appear in.
- Let the viewport coordinates obey the following relationships:
  - $v_x_{\min} \leq v_x \leq v_x_{\max}$
  - $v_y_{\min} \leq v_y \leq v_y_{\max}$
- The OpenGL code to set the viewport is:
  - `glViewport (v_x_{\min}, v_y_{\min}, v_x_{\max}-v_x_{\min}, v_y_{\max}-v_y_{\min});`
- The default viewport is the entire window.
Window to viewport mapping

- Observe that going from world coordinates to viewport coordinates is nothing more than a change of coordinate systems.
- This is accomplished by a mapping.
- We want this mapping to be proportional, to preserve relative positions of objects in the scene.
Derivation of mapping

- To preserve proportionality we have (in the x direction)

\[
\frac{v_x - v_x \text{ min}}{v_x \text{ max} - v_x \text{ min}} = \frac{w_x - w_x \text{ min}}{w_x \text{ max} - w_x \text{ min}}
\]

- Solving for \(v_x\) gives

\[
v_x = \frac{v_x \text{ max} - v_x \text{ min}}{w_x \text{ max} - w_x \text{ min}} w_x + \left( v_x \text{ min} - \frac{v_x \text{ max} - v_x \text{ min}}{w_x \text{ max} - w_x \text{ min}} w_x \text{ min} \right)
\]

- Note that this involves a scaling and a shifting (or translation)
Viewing transformation

- We now have the necessary skills to understand the 2-D viewing transformation - how a 2-D image is positioned on the screen.

- First, we set the viewing window, the range of coordinate values in $x$ and $y$ that we want displayed.

- Next, we set the viewport, the range of physical pixels that we want the viewing window displayed in.
Viewing transformation steps

- Translate the window to the origin

\[
W = \begin{bmatrix}
1 & 0 & -wx_{\text{min}} \\
0 & 1 & -wy_{\text{min}} \\
0 & 0 & 1
\end{bmatrix}
\]

- Scale by the ratio of the viewport dimensions to the window dimensions

\[
S = \begin{bmatrix}
\frac{vx_{\text{max}} - vx_{\text{min}}}{wx_{\text{max}} - wx_{\text{min}}} & 0 & 0 \\
0 & \frac{vy_{\text{max}} - vy_{\text{min}}}{wy_{\text{max}} - wy_{\text{min}}} & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
Viewing transformation steps (cont.)

- Translate the viewport to its correct spot
- The composite transformation is then given by the product of the three matrices

\[ V = \begin{bmatrix} 1 & 0 & v_x_{\text{min}} \\ 0 & 1 & v_y_{\text{min}} \\ 0 & 0 & 1 \end{bmatrix} \]

\[ M = VSW \]
Three dimensions

- The real world is 3-dimensional, that is it takes 3 coordinates \((x, y, z)\) to position a point in space.
- In homogeneous coordinates we will use the vector \([x \ y \ z \ 1]^T\) to represent a point in 3 dimensions.
- Transformation matrices will now be \(4 \times 4\).
3-D scaling

- The $4 \times 4$ matrix to accomplish 3-D scaling is

$$
\begin{bmatrix}
S_x & 0 & 0 & 0 \\
0 & S_y & 0 & 0 \\
0 & 0 & S_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$
3-D translation

- The 4×4 matrix to accomplish 3-D translation is

\[
\begin{bmatrix}
1 & 0 & 0 & t_x \\
0 & 1 & 0 & t_y \\
0 & 0 & 1 & t_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
3-D reflections

- The 4×4 matrix to accomplish reflection in the y-z plane

\[
\begin{bmatrix}
-1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

- The 4×4 matrix to accomplish reflection in the x-z plane

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

- The 4×4 matrix to accomplish reflection in the x-y plane

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
3-D rotations

- The $4 \times 4$ matrix to accomplish rotation about the x-axis

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- The $4 \times 4$ matrix to accomplish rotation about the y-axis

\[
\begin{bmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- The $4 \times 4$ matrix to accomplish rotation about the z-axis

\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Geometry for rotation about an arbitrary line in 3-D
Steps for arbitrary rotation in 3-D

- Rotate $-\varphi$ around z-axis to bring L to x-z plane
- Rotate $-\nu$ around y-axis to bring L to z-axis
- Rotate $\theta$ around z-axis
- Rotate $\nu$ around y-axis
- Rotate $\varphi$ around z-axis
OpenGL representation of points

- OpenGL uses a column vector to represent points:

\[
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]
OpenGL transformation matrices

- Model-view matrix (GL_MODELVIEW)
- Projection matrix (GL_PROJECTION)
- The current transformation matrix is the product of the above two matrices
Computation of transformed point

- OpenGL computes the displayed point according to the equation
  \[ P' = MP \]
- Note that the transformation matrix (M) premultiplies the point
OpenGL order of transformations

- When another transformation is applied, the new transformation matrix \textbf{postmultiplies} the current transformation matrix.

- This has the effect of applying the last transformation specified first and applying the first transformation specified last.
OpenGL transformations

- `glTranslate{d,f} (x, y, z)`
- `glRotate{d,f} (angle, x, y, z)`
  - `angle` is in degrees
  - rotation occurs around the vector `(x, y, z)`
  - follows right-hand rule (counter clockwise if vector points toward the viewer)
- `glScale{d,f} (x, y, z)`
- `glMultMatrix{d,f} (*m)`
  - `m` is a 16 element matrix
  - column major ordering
Affine transformations

- The transformations we have been discussing are called **affine transformations**
- They have the form $P' = MP$
- Affine transformation properties:
  - preserve lines
  - parallel lines remain parallel
  - proportional distances are preserved
Window resizing

- When the window is resized, a default reshape callback function is executed to reset the viewport to the new window size.
- This will distort your image, as the projection mapping is not changed.
Reshape callback function

- You may register your own reshape callback function using `glutReshapeFunc()`

- If you do, be sure to reset the viewport in it using `glViewport()`
Example reshape callback function

```c
void reshape (GLsizei w, GLsizei h)
/* This function gets called whenever the window is resized. */
{
    /* Adjust the clipping rectangle. */
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D (0.0, w - 1, 0.0, h - 1);
    /* Adjust the viewport. */
    glViewport (0, 0, w, h);
}
```
Pop-up menus

- Attached to a mouse button
- `glutCreateMenu()` used to create a menu and to register the menu’s callback function
- `glutAddMenuEntry()` used to add an entry string and its code to the menu
- Code is used in menu’s callback function to decide which menu entry was selected
Pop-up menus (cont.)

- `glutAddSubMenu()` adds a menu as a sub-menu to the indicated menu
- `glutAttachMenu()` attaches a mouse button to the current menu
Example program

- MenuDemo program is an example of pop-up menus and window resizing
The Mandelbrot set

- Given two complex numbers, \(c\) and \(z\), and the complex plane, \(\mathbb{C}\)
- The Mandelbrot set is defined by
  \[
  \{ c \in \mathbb{C} \mid \lim_{i \to \infty} z_i \neq \infty \ \text{under} \ z_{i+1} = z_i^2 + c, \ z_0 = (0, 0) \}
  \]
- Choose \(c\). After each iteration see if \(|z| > 2\). If yes, stop, \(z \to \infty\). If not, \(c\) is in Mandelbrot set.
- Use a reasonable number of iterations.
Example

- $c = 0.5 + 0.5i$ (note this is also $z_1$)
- $z_2 = c^2 + c = 0.5 + 1.0i$
- $z_3 = -0.25 + 1.5i$
- $z_4 = -1.6875 - 0.25i$
- $z_5 = 3.28515625 + 1.34375i$
- $|z_5| > 2$, so $0.5 + 0.5i$ is not in the Mandelbrot set
Fixed points

- $c = -0.2 + 0.5i$
- $z_2 = -0.41 + 0.3i$
- $z_3 = -0.1219 + 0.254i$
- $z_4 = -0.2497 + 0.4381i$
- After 80 iterations the value of $z$ is $-0.249227 + 0.333677i$
- From this point on, $z^2 + c = z$
- Therefore, $c$ is called a **fixed point** of the function
- Since $|z| = 0.416479$, $c$ is in the Mandelbrot set
Generating the picture

- Pick a region of the complex plane
- For each point in the region look at the result of the \( z = z^2 + c \) transformation after a given number of iterations
- If \( |z| < 2 \) then \( c \) is in the Mandelbrot set - color it black
- If \( |z| > 2 \) then \( c \) is not in the Mandelbrot set - choose color based on the number of iterations it took to exceed 2 in magnitude
Boundary area

- The boundary of the Mandelbrot set is a fractal
- Chaotic behavior is exhibited there
- You can zoom in on a point and see new details
- Image at left is centered at -0.013 + 0.715i with a width of 0.01
Bitblts

- A rectangular block of pixels (bits) is known as a **bit block**
- Operations that work on bit blocks are known as **bit-block transfer (bitblt)** operations
Logic operations

- Applied bit by bit between the source image and the destination buffer

- Enable by
  - `glEnable (GL_COLOR_LOGIC_OP);`
  - `glEnable (GL_INDEX_LOGIC_OP);`

- Specify by `glLogicOp (op);`
  - Some possible ops are:
    - `GL_COPY`
    - `GL_OR`
    - `GL_XOR`
Exclusive or

- The most interesting logical operation
- If applied twice in succession, you get back to the starting state: $x = x \oplus y \oplus y$
Let M be an array of bits representing a menu that is stored off screen (in what is known as a \textit{backing store})

Let S be the part of the screen that contains an image over which the menu will appear.

Consider the sequence of operations:

\begin{itemize}
  \item S ← S ⊕ M
  \item M ← S ⊕ M (original contents of screen now stored where menu was)
  \item S ← S ⊕ M (menu now on screen)
\end{itemize}
Example program

- MandelbrotZoom.cpp draws the Mandelbrot set using pixel drawing
- glDrawPixels (width, height, format, type, pixel_array)
  - draws at current raster position
  - format indicates kind of data
  - type indicates data type
  - see the OpenGL Reference Manual for symbolic constants that can be used
- glRasterPos{234}{sifd}[v]() sets the current raster position
- glReadPixels (x, y, width, height, format, type, pixel_array) reads pixel array beginning at lower left corner (x, y)