Review of Lecture 2

- first program - putting it together (main, myInit, myDisplay, event driven programming)
- primitives - points, lines, polylines, polygons
- event-driven programming - simple mouse and keyboard events
- GLUT functions, GL functions

- Case study - pseudorandom clouds of dots.
Iterative functions

Sierpinski gasket: repetitive action.

Plotting hailstone sequence.

\[
\begin{align*}
  d_0 &= x_0 \\
  d_1 &= f(d_0) \\
  d_2 &= f(d_1) \\
  d_3 &= f(f(d_1)) \\
  &\vdots
\end{align*}
\]

This sequence of values \(d_0, d_1, d_2, d_3, \ldots\) is called "the orbit of \(d_0\)" for the system.

\[
f(x) = \begin{cases} 
  \frac{x}{2} & \text{if } x \text{ is even} \\
  3x + 1 & \text{if } x \text{ is odd}
\end{cases}
\]
So far we work with:

- positive values only;
- the values must extend over a large range.

We may not want to think in terms of pixels, but in terms of varying x or y from a negative value to a positive value.

World coordinates, world window

Viewport, automatic change of coordinates.

```c
void myDisplay(void)
{
    glBegin(GL_LINE_STRIP);
    for(GLfloat x = -4.0; x < 4.0; x += 0.1)
    {
        GLfloat y = sin(3.14159 * x) / (3.14159 * x);
        glVertex2f(x, y);
    }
    glEnd();
    glFlush();
}
```

```c
void myInit(void)
{
    glClearColor(1.0,1.0,1.0,0.0);
    glColor3f(0.0,0.0f,1.0f);
    glPointSize(2.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-4.0,4.0,-1.0,1.0); //sets the window
    glViewport(0,0,600,400); //sets the viewport
}
```
Orthographic view

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)
equivalent to
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble -1.0, GLdouble 1.0)

Tiling

If we lay a number of copies of a figure side by side to cover the entire screen window it’s called tiling the screen window. The picture that is copied at different positions is often called a motif. Tiling a screen window is easily achieved by using a different viewport for each instance of the motif.

```cpp
for(int i = 0; i < 5; i++)
for(int j = 0; j < 5; j++)
{
    glViewport(i * 64, j * 44, 64, 44);
drawPolyLineFile("dino.dat");
}
```

Clipping, zooming, and roaming

See the two boxes that indicate different choices of a window. Parts b and c show what is drawn if these boxes are used for the world windows. It is important to keep in mind that the same entire object is drawn in each case.

Making the window larger is similar to zooming out. When making the window smaller, whatever is in the window must be stretched to fit in the fixed viewport, similar to zooming in. A camera can also roam around a scene, taking in different parts of it at different times. This is easily accomplished by shifting the window to a new position.

```cpp
for(int i = 0; i < 5; i++)
for(int j = 0; j < 5; j++)
{
    if((i + j) % 2 == 0)
    {
        setWindow(0.0, 640.0, 0.0, 480.0);
    }
    else
    {
        setWindow(0.0, 640.0, 480.0, 0.0);
    }
    glViewport(i * 64, j * 44, 64, 44);
drawPolyLineFile("dino.dat");
}
```
Consider putting together an animation where the camera zooms in on some portion of a figure. We make a series of pictures, often called frames, using a slightly smaller window for each one. When the frames are displayed in rapid succession the visual effect is of the camera zooming in on the object.

The previous approach isn’t completely satisfying, because of the time it takes to draw each new figure. What the user sees is a repetitive cycle of:

a). Instantaneous erasure of the current figure;

b). A (possibly) slow redrawing of the new figure.

The problem is that the user sees the line-by-line creation of the new frame, which can be distracting. What the user would like to see is a repetitive cycle of:

a). A steady display of the current figure;

b). Instantaneous replacement of the current figure by the finished new figure.

The trick is to draw the new figure "somewhere else" while the user stares at the current figure, and then to move the completed new figure instantaneously onto the user’s display.

OpenGl offers double-buffering to accomplish this. Memory is set aside for an extra screen window which is not visible on the actual display, and all drawing is done to this buffer.

The command glutSwapBuffers() then causes the image in this buffer to be transferred onto the screen window visible to the user.

To make OpenGL reserve a separate buffer for this, use GLUT_DOUBLE rather than GLUT_SINGLE in the routine used in main() to initialize the display mode:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

The command glutSwapBuffers() would be placed directly after drawPolylineFile() in the code. Then, even if it takes a substantial period for the polyline to be drawn, at least the image will change abruptly from one figure to the next in the animation, producing a much smoother and visually comfortable effect.
Setting window and viewport automatically

In some cases the programmer (or possibly the user at run-time) can input the window and viewport specifications to achieve a certain effect; in other cases one or both of them are set up automatically, according to some requirement for the picture.

Setting the window

The extent (or bounding box) of an object is the aligned rectangle that just covers it.

Pass 1: Execute the drawing routine, but do no actual drawing; just compute the extent. Then set the window.

Pass 2: Execute the drawing routine again. Do the actual drawing.

Setting the viewport

Suppose you want to draw the largest undistorted version of a figure that will fit in the screen window. For this you need to specify a viewport that has the same aspect ratio as the world window. A common wish is to find the largest such viewport that will fit inside the screen window on the display.

Case a): \( R > W/H \). Here the world window is short and stout relative to the screen window, so the viewport with a matching aspect ratio \( R \) will extend fully across the screen window, but will leave some unused space above or below. At its largest, therefore, it will have width \( W \) and height \( W/R \), so the viewport is set using (check that this viewport does indeed have aspect ratio \( R \)):

\[
\text{setViewport}(0, W, 0, W/R);
\]

Case b): \( R < W/H \). Here the world window is tall and narrow relative to the screen window, so the viewport of matching aspect ratio \( R \) will reach from the top to the bottom of the screen window, but will leave some unused space to the left or right. At its largest it will have height \( H \) but width \( HR \), so the viewport is set using:

\[
\text{setViewport}(0, H * R, 0, H);
\]

Example 3.2.7: A tall window. Suppose the window has aspect ratio \( R = 1.6 \) and the screen window has \( H = 200 \) and \( W = 360 \), and hence \( W/H = 1.8 \). Therefore Case b) applies, and the viewport is set to have a height of 200 pixels and a width of 320 pixels.

Example 3.2.8: A short window. Suppose \( R = 2 \) and the screen window is the same as in the example above. Then case a) applies, and the viewport is set to have a height of 180 pixels and a width of 360 pixels.

Resizing the screen window, and the resize event.

In a windows-based system the user can resize the screen window at run-time, typically by dragging one of its corners with the mouse. This action generates a resize event that the system can respond to.

\[
\text{glutReshape(myReshape)};
\]

\[
\text{void myReshape(GLsizei W, GLsizei H)};
\]

When it is executed the system automatically passes it the new width and height of the screen window, which it can use in its calculations.

Matching the viewport

\[
\text{void myReshape(GLsizei W, GLsizei H)}
\]

\[
\text{if}(R > W/H) // use (global) window aspect ratio
\]

\[
\text{setViewport}(0, W, 0, W/R);
\]

\[
\text{else}
\]

Find the matching viewport for a window with aspect ratio .75 when the screen window has width 640 and height 480.
Clipping lines

Clipping algorithm that clips off outlying parts of each line segment presented to it.

(Note: A must be computed. Its x-coordinate is clearly its right edge position. Its y-coordinate requires adjusting g1.y by the amount d shown in the figure. But by similar triangles

\[
delx = \frac{d}{dy}
\]

where \( pl.x \rightarrow \) right edge

\[
delx = g2.x - pl.x
\]

\[
dely = g2.y - pl.y
\]

are the differences between the coordinates of the two endpoints. Thus d is easily determined, and the new g1.y is found by adding an increment to the old as

\[
g1.y = (dy.right - pl.y) * delx / delx
\]