LAB. 4

WINDOWS AND VIEWPORTS

1. Learning goals
2. Getting started
3. 2D viewing and world-to-screen mapping
4. Example
5. Programming exercises
6. Interesting web links

Lab. 4

WINDOWS AND VIEWPORTS

This lecture continues with rendering of 2D scenes. But, now we use two or more viewports to render one or more scenes.

1. Learning Goals

At the end of this chapter **you should be able to**:

1. Explain how a scene within a domain window is mapped onto a screen viewport. This process involves 1 translation in the domain, 1 scaling between the domain window and the screen viewport, and 1 translation in the screen.
2. Understand how this window-viewport mapping is done in an OpenGL program.
3. Write a graphics program to map a 2D scene onto two or more viewports.
4. Program a reshape callback to automatically keep the aspect ratios in the window-viewport mapping.

2. Getting started

As known, OpenGL (Open Graphics Library) is a cross-platform, hardware-accelerated, language-independent, standard API that we use to produce 2D and 3D scenes, i.e., OpenGL is the software interface to graphics hardware.

3.1. Graphics libraries and functions

Usually, we use three libraries in our OpenGL programs:

1. **Core OpenGL (GL)**: consists of hundreds of functions, which start with the prefix "gl" (e.g., glColor, glVertex, glTranslate, glRotate). It is this core that allows us to model objects and build up scenes using a set of geometric primitives — point (GL_POINTS), line (GL_LINES), and polygon (GL_POLYGON).
2. **OpenGL Utility Library (GLU)**: is built on the top of the core OpenGL and provides a number of important utilities, including geometric models such as, for example, quadric surfaces. GLU functions begin with the prefix "glu" (e.g., gluLookAt, gluPerspective)
3. **OpenGL Utilities Toolkit (GLUT)**: provides a number of support facilities to interact with the operating system (e.g., creating a window, handling key and mouse inputs), as well as more geometric models such as, for example, spheres and tori. GLUT functions begin with the prefix "glut" (e.g., glutCreatewindow, glutMouseFunc).

3.2. OpenGL "cameras"

- OpenGL provides both orthographic and perspective projections
- projections include FOV (perspective only), and distances to near/far planes, and left/right/top/bottom edges of the near plane
- projections are not true "cameras" as they only describe the projection from 3D into 2D, not position and orientation of the camera.

3.3. OpenGL orthographic projection functions

- **`glMatrixMode`**`(GL_PROJECTION)`
  - projections can be expressed as a 4x4 matrix (and are expressed that way in OpenGL)
  - in order to set the projection, you must place OpenGL in GL_PROJECTION matrix mode. (Other matrix modes include GL_MODELVIEW and GL_TEXTURE.)
- **`glOrtho`**`(left, right, bottom, top, near, far)`
  - where values are GLdoubles specified in world space
- **`gluOrtho2D`**`(left, right, bottom, top)`
  - values specified in world space
  - depth (far->near) is assumed to be in the range [-1, 1]
  - 2D vertices have z = 0

3.4. Viewport mapping

Let us recall that a **viewport** is defined as a region of the window that you wish to map the OpenGL image to. Interestingly, we may define multiple viewports for a single window, so that we can display different parts of a scene into different viewports.

- **`glViewport`**`(x, y, width, height)`
  - where (x, y) specify the lower left corner of the viewport in window space (GLint)
  - where (width, height) are the size of the viewport in pixels (GLsizei)
  - default values specify the entire window, i.e. (0, 0, windowWidth, windowHeight)
- To avoid distortion of an image when the window is resized, modify the aspect ratio of the projection to match the viewport, e.g., assuming we want to maintain a 1:1 aspect ratio in our image...
  glViewport(0, 0, 400, 400) -> gluOrtho(-1.0, 1.0, -1.0, 1.0) //or any other ranges in a 1:1 ratio
  glViewport(0, 0, 400, 200) -> gluOrtho(-2.0, 2.0, -1.0, 1.0) or gluOrtho(-1.0, 1.0, -0.5, 0.5)

3.5. Window sizing and resizing throught GLUT

Setting the Window Size

- **`glutInitWindowSize`**`(int width, int height)` - size in pixels

- **`glutReshapeWindow`**`(int width, int height)` - use this after initial creation

Reading the size of the window

- `int` **`glutGet`**`(<state>);`
  - returns requested value
  - GLUT_WINDOW_WIDTH, GLUT_WINDOW_HEIGHT - size of window
  - GLUT_WINDOW_X, GLUT_WINDOW_Y - location of window

Reacting to User resizing of window

- **`glutReshapeFunc`**`(void (*func) (int width, int height) )`
  - sets the callback function to be invoked when the window gets resized
  - callback function is sent new size of the window in width & height
  - callback function should reset the viewport, and possibly the orthographic projection

3.6. Putting it all together

Depending on the application, you may need to work in 2D screen space (pixel space) or 2D world space.

Screen space

- manage viewport to match window size
- set orthographic projection to be equivalent to viewport size
  - e.g. if glViewport(0,0,640,480) then use gluOrtho2D(0,0,640,480)
- you now have one-to-one control over pixels, e.g., glVertex2i(10, 20)

World space

- manage viewport to match window size
- set orthographic projection to match aspect ratio of viewport
- establish policy for modifying projection aspect ratio in x or y direction
- you don't have precise pixel control, but you can specify values in their appropriate world coordinates. e.g., glVertex2f(0.25, -1.375);

3. 2D Viewing and World-to-Screen Mapping

Let us consider a domain window in $\mathbf{R}^2$ defined by $(x_{min},y_{min},x_{max},y_{max})$ and a point (x,y) such that $x \in [x_{min},x_{max}]$ and $y \in [y_{min},y_{max}]$. Let us also consider a viewport in a screen window given by $(X_{min},Y_{min},X_{max},Y_{max})$.

Now, the problem is to determine the pixel (X,Y) within the viewport that is located at the same relative position as (x,y) within $(x_{min},y_{min},x_{max},y_{max})$. That is, mapping a 2D point (x,y) of the domain window onto a pixel (X,Y) of a screen window (or viewport) requires to maintain the relative position of such a pixel within the viewport.

To make sure that the relative positions of (x,y) and (X,Y) are maintained, the following relationships must be satisfied:

$$\frac{x - x_{\min}}{x_{\max} - x_{\min}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

and

$$\frac{y - y_{\min}}{y_{\max} - y_{\min}} = \frac{Y - Y_{\min}}{Y_{\max} - Y_{\min}}$$

from where we can obtain the expressions of the pixel (X,Y) from the point (x,y) as follows:

$$X = X_{\min} + \frac{X_{\max} - X_{\min}}{x_{\max} - x_{\min}} \times \left(x - x_{\min}\right)$$

and

$$Y = Y_{\min} + \frac{Y_{\max} - Y_{\min}}{y_{\max} - y_{\min}} \times \left(y - y_{\min}\right)$$

Thus, the window-viewport mapping involves a translation $(-x_{min},-y_{min})$ in R^2, a scaling (S_x,S_y), and a translation (X_{min},Y_{min}) in the screen. It is clear that $S_x=(X_{max}-X_{min})/(x_{max}-x_{min})$ and $S_y=(Y_{max}-Y_{min})/(y_{max}-y_{min})$ are the scaling factors.

Note that keeping the relative positions does NOT mean to keep aspect ratios (or proportions). If the domain window and the viewport do not have identical aspect ratios, the image appears deformed on screen.

4. Example

The following program draws two rectangles in distinct viewports. This program has been borrowed from the theoretical module T03 ("Windows and Viewports").

```
1  /* Using two viewports */
2  #include <GLUT/glut.h>
3  void draw(){
4          // make background colour yellow
5      glClearColor( 100, 100, 0, 0 );
6      glClear ( GL_COLOR_BUFFER_BIT );
7         // set up FIRST viewport
8      glViewport(0,0,250,250);
9         // Sets up the PROJECTION matrix
10     glMatrixMode(GL_PROJECTION);
11     glLoadIdentity();
12         // set up world window
13     gluOrtho2D(0.0,50.0,-10.0,40.0);
14        // Draw BLUE rectangle
15     glColor3f( 0, 0, 1 );
16     glRectf(0.0,0.0,10.0,30.0);
17        // sets up SECOND viewport
18     glViewport(250,250,250,250);
```

```
19          // set up the PROJECTION matrix
20      glMatrixMode(GL_PROJECTION);
21      glLoadIdentity();
22          // set up world window
23      gluOrtho2D(0.0,50.0,-10.0,40.0);
24          // draw RED rectangle
25      glColor3f( 1, 0, 0 );
26      glRectf(0.0,0.0,10.0,30.0);
27          // display rectangles
28      glutSwapBuffers();
29  }
30
31  // Keyboard method to allow ESC key to quit
32  void keyboard(unsigned char key,int x,int y)
33  {
34      if(key==27) exit(0);
35  }
36  int main(int argc,  char ** argv)
37  {
38     glutInit(&argc, argv);
39     // Double Buffered RGB display
40     glutInitDisplayMode(GLUT_RGB |GLUT_DOUBLE);
41     // Set window size
42     glutInitWindowSize( 500,500 );
43     // two viewports
44     glutCreateWindow(”Two viewports");
45     // the display callback
46     glutDisplayFunc(draw);
47     // the keyboard callback
48     glutKeyboardFunc(keyboard);
49     // Start the Main Loop
50     glutMainLoop();
51     return 0;
52  }
```

5. Programming Exercises

1. Write a program to draw the *cos* function on a viewport and the *sin* function on another viewport.
2. Re-write the previous program in order to add a third viewport displaying a bouncing ball.
3. Re-write the house-building program implemented in the previous classes in order to simulate the panning operation through the arrow keys (GLUT_KEY_UP, GLUT_KEY_DOWN, GLUT_KEY_LEFT, and GLUT_KEY_RIGHT) of the keyboard. For that purpose, we have to move the domain window around the house. The movement step in each direction is 5 percent of the domain window's width.
4. Re-write the house-building program above to add the zooming operation. The zoom-in operation will be controlled by the '+' key, while the zoom-out operation will use the '-' key.
5. Add the reshaping facility to the previous program.
6. Add the full-screen facility to the previous program as follows:

```
// Handler for special-key event
void specialKey(int key, int x, int y)
{
  switch (key)
  {
   case GLUT_KEY_F1:    // F1: Toggle between full-screen and windowed mode
     fullScreenMode = !fullScreenMode;         // Toggle state
     if (fullScreenMode) {                     // Full-screen mode
        windowPosX   = glutGet(GLUT_WINDOW_X); // Save parameters
        windowPosY   = glutGet(GLUT_WINDOW_Y);
        windowWidth  = glutGet(GLUT_WINDOW_WIDTH);
        windowHeight = glutGet(GLUT_WINDOW_HEIGHT);
        glutFullScreen();                      // Switch into full screen
     }
     else
     {                                         // Windowed mode
        glutReshapeWindow(windowWidth,windowHeight);//Switch into windowed mode
        glutPositionWindow(windowPosX, windowPosX);//Postion top-left corner
     }
     break;
  default:
     break;
 }
}
```

This requires the registration of this callback in the main function as follows:

```
glutSpecialFunc(specialKey); // Register handler for special-key event
```

as well as the declaration of the following global variables:

```
int windowWidth  = 640;     // Windowed mode's width
int windowHeight = 480;     // Windowed mode's height
int windowPosX   = 50;      // Windowed mode's top-left corner x
int windowPosY   = 50;      // Windowed mode's top-left corner y
bool fullScreenMode = true; // Full-screen or windowed mode?
```

The first two variables are used to set up the width and height of the initial window, and the next two variables are used to set up the position of such a window on screen, as usual in the main function:

```
glutInitWindowSize(windowWidth,windowHeight);//Initial window width and height
glutInitWindowPosition(windowPosX,windowPosY);//Initial window top-left corner
```

6. Interesting web links

OpenGL Programming Guide: Chapter 1
http://glprogramming.com/red/chapter01.html

An Introduction with 2D Graphics:
http://www.ntu.edu.sg/home/ehchua/programming/opengl/GL1_Introduction.html