Chapter 1

The 2D Image domain

In computer graphics, our finished product is a discrete image.

$$\texttt{I[i][j]}$$

`I` is a two dimensional w by h (width by height) array of values, And so the valid range the indecies is $0 \leq \texttt{i} \leq w-1$ and $0 \leq \texttt{j} \leq h-1$. For a black and white image, each pixel value is a single number. For a color image, each pixel value is three numbers representing red green and blue. In typical image formats, 8 bits are allocated for each of the red green and blue channels per pixel. These can be fed to a monitor and displayed as an image on a screen.

A discrete image is more than an array of values, it represents a regularly spaced set of samples over a continuous two dimensional domain. One can think of `I[i][j]` as a discrete realization of some image function $I(x_p, y_p)$ over the continuous two dimensional domain. We will use the subscript "p" to denote that these are two dimensional pixel coordinates. To be precice then, we can think of pixel `[i][j]` as associated with the continuous two dimensional coordinates $x_p = i$, and $x_p = j$.

We often think of each sample as having a buffer of length 1/2 in each direction. If we let the exterior boundary pixels have this buffer as well, then we can concider the valid range of the continuous image domain to be $-.5 \leq x_p \leq w-.5$ and $-.5 \leq y_p \leq h-.5$.

1.1 World Coordinates

In computer graphics, one typically describes images by describing the locations of various geometric shapes, for example a red triangle. The particular shapes are described using coordinates. For example, a triangle is described by the positions of its three vertices.

Let us suppose we wanted to specify the location of a triangle in some image. We could specify the postions of the vertices using $[x_p, y_p]^t$ coordinages. This notation is simply short for the column coordinate vector

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix}$$

It is not typically convenient to be thinking of locations in terms of pixels; if the user decides that he wants an image of a different size, then he will need to explicitly recalculate the new coordinates of the same triangle.

To avoid this inconvenience, we would like to speak of postions in the image rectangle without knowledge of the pixel dimensions. To do this, we use a new coordinate system, which we call the world coordinate system. We describe all of our coordinates with repsect to this pixel independent coordinate system. We denote these coordinates as $[x_w, y_w]^t$.

Of course we also must specify the mapping between world and pixel coordinates. The way we can do this is by explicity saying which range of world coordinates gets mapped onto the valid pixel coordinate range $-.5 \leq x_p \leq w - .5$ and $-.5 \leq y_p \leq h - .5$. This range of world coordinates is specified by the fourtuple [world left, world right, world bottom, world top]. In this case, the range of valid world coordinates is

$$\begin{aligned} \text{left} \leq \quad & x_w \quad \leq \text{right} \\ \text{bottom} \leq \quad & y_w \quad \leq \text{top} \end{aligned}$$

This range specification is sufficient to uniquely specify the (affine) mapping between these coordinate systems. With this specfication, any geometric points that have their world coordinates betweeen left and right, bottom and top, get mapped to the valid region of the image. Points outside of this range get “clipped” away, and ignored.

If we do not want any shape deformation of the object to occur, then we must make sure that this range of world coordinates has the same aspect ratio as the range of pixel coordinates.

One reasonable choice we recommend is

$$\begin{aligned} \text{left} &= -w/h \\ \text{right} &= w/h \\ \text{bottom} &= -1 \\ \text{top} &= 1 \end{aligned}$$

1.2 Image Coordinates

As far as you the user is concerned, all that is important is world coordinates, and maybe pixel coordinates. It turns out that it is useful for the rendering system to

use an intermediate coordinate system to describe points on their journey from world coorindates to pixel coordinates. We call these intermediate coordinates image coordinates $[x_i, y_i]^t$.

Image coordinates give us a generic coordinate system in which to do some geometric processing In this coordinate system, we concider the valid range of the image to be

$$\begin{aligned} -1 \leq & \quad x_i \quad \leq 1 \\ -1 \leq & \quad y_i \quad \leq 1 \end{aligned}$$

1.3 Coordinate changes

Given these three cooridinate systems, world, image, and pixel, we will need two transformations, one to change world coordinates into image coordinates, and one to change image coordinates into pixel coordinates. We want each of these transforms to have a very specific form: a scale and shift in x coorindates, and a scale and shift in y coordinates. Thus, the transform is governed by four parameters c, d, e, and f.

$$\begin{aligned} x_i &= c\, x_w + d \\ y_i &= e\, y_w + f \end{aligned}$$

The four parameters of the transform can be deterermined using the the four numbers that specify the valid range of coordinates. We consider each of the valid range numbers as a constraint. Using these four constraints, we set up a system of four linear equations and solve for c, d, e, f.

$$\begin{aligned} -1 &= cl + d \\ 1 &= cr + d \\ -1 &= eb + f \\ 1 &= et + f \end{aligned}$$

where l,r,t,b is left, right, bottom, top.

Using simple algera, we can solve for a, b, c, d obtaining:

$$\begin{aligned} x_c &= \frac{2}{r-l} x_w + \frac{-r-l}{r-l} \\ y_c &= \frac{2}{t-b} y_w + \frac{-t-b}{t-b} \end{aligned}$$

The coordinate transformation between image and pixel coordinates can be computed in the exact same way. One starts with the knowledge that the transformation is a scale and shift

$$\begin{aligned} x_p &= g\, x_i + h \\ y_p &= i\, y_i + j \end{aligned}$$

Sets up the constraints

$$\begin{aligned} -.5 &= g(-1) + h \\ w - .5 &= g1 + h \\ -.5 &= i(-1) + j \\ h - .5 &= i1 + j \end{aligned}$$

And solves the equations obtaining

$$\begin{aligned} x_p &= \frac{w}{2} x_i + \frac{w-1}{2} \\ y_p &= \frac{h}{2} y_i + \frac{h-1}{2} \end{aligned}$$

1.4 openGL commands

In openGL, the transformation between world and image coordinates can be controlled using the command
`gluOrtho2D(left,right,bottom,top)`.

In openGL, the transformation between image and pixel coordinates is controlled using the command `glViewport`. If one wants to obtain the standard transformation described in the previous section, one uses the arguments
`glViewport(0,0,w,h)`. Other arguments can be used if one wants to do fun things like fit a bunch of images into a single window.

1.4.1 Simple program

Putting this together, we get the simple openGL program.

```
#include <stdio.h>
#include <stdlib.h>
#include "GL/glut.h"

int GW;
int GH;


void display(void){
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_POLYGON);
    glColor3f(1,1,1);
```

```
    glVertex2f(-.5, -.5);
    glVertex2f(-.5,  .5);
    glColor3f(1,.5 ,.5);
    glVertex2f( .5,  .5);
    glVertex2f(.5, -.5);
  glEnd();
  glFlush();
  glutSwapBuffers();
}


void
reshape(int w, int h)
{
  GW = w;
  GH = h;
  glViewport(0, 0, w, h);               /* Set Viewport */
  glLoadIdentity();
  gluOrtho2D( -(float)w/h, (float)w/h, -1, 1);
}
```

```
main(int argc, char **argv) {

  /* glut stuff */
  glutInit(&argc, argv);                /* Initialize GLUT */
  glutInitDisplayMode(GLUT_RGB| GLUT_DOUBLE ); /* Set Dis-
play mode  */
  glutInitWindowSize(200, 100);
  glutCreateWindow("my window");           /* Create window with given ti
tle */
  glViewport(0,0,200,100 );

  glutDisplayFunc(display);             /* Set-up callback func-
tions */
  glutReshapeFunc(reshape);
  glutMouseFunc(mouse);
  glutMotionFunc(mouseMove);

  glClearColor(0,0,0,0);
  gluOrtho2D( -2,2, -1, 1);
  glShadeModel(GL_SMOOTH);

  glutMainLoop();                       /* Start GLUT event-
processing loop */

}


void changeSquareColor(){}

void
mouse(int button, int state, int x, int y) {
  if (button == GLUT_LEFT_BUTTON)
    if (state == GLUT_DOWN) {           /* If the left but-
ton is clicked */
      printf("mouse clicked at %d %d\n",x,GH-y-1);
    }
  changeSquareColor();
  glutPostRedisplay();
}

void mouseMove(int x, int y){
  printf("mouse moved to at %d %d\n",x,GH-y-1);
}
```

```
CC  test.c \
-lglut175 -lGLU -lGL -lX11 -lXmu
```