1 Overview

The purpose of a computer graphics system is to enable a user to construct scenes and views to achieve a desired result. Often, speed or real-time performance is also a major concern. Building complex systems requires careful software design in order to minimize complexity, unexpected effects of changes, readability, and expandability. Modularity in computer graphics system design is an important component of achieving these goals.

The following document specifies the data structures, functions, and functionality for a basic 2D rendering system using C. The data types within the system include the following.

- Pixel - data structure for holding image data for a single point
- Image - data structure for holding the data for a single image
- Point - a 2D or 3D location in model space
- Line - a line segment in 2D or 3D model space
- Circle - a 2D circle
- Ellipse - a 2D ellipse
- Polygon - a closed shape consisting of line segments
- Polyline - a sequence of connected line segments
- Color - a representation of light energy
- Light - a light source
- Vector - a direction in 2D or 3D model space
- Matrix - a 2D or 3D transformation matrix
- Element - a element of a model of a scene
- Module - a collection of elements
- View2D - information required for a 2D view of a scene
- ViewPerspective - information required for a perspective view of a scene
- ViewOrthographic - information required for an orthographic view of a scene

Over the course of the semester, you will implement each of the above data structures using a C struct data type. The required fields and their purposes are given in the specification below. For each data type, you will also implement a set of functions. The function prototypes and a description of the purpose of each required function are also included in the specification, and you are free to add additional functionality. If your code supports the required data types and functionality, then you will be able to run a series of test programs to evaluate and debug your system.
1.1 C versus C++

Many aspects of computer graphics are appropriate for an object-based approach to software design. Primitives such as lines, circles, points and polygons naturally exist as objects that need to be created, manipulated, drawn into an image, and destroyed. We may also want to store a symbolic representation of a scene by saving lists of primitives to a file, which is a natural part of an object-oriented approach.

The ideas of modularity and object-based design are possible to implement in either C or C++. The features of C++ give more structure and flexibility to the object-based approach; C gives the programmer lower-level control of information and forces a deeper understanding of how the information flow occurs. A continuum of possible software system structures are possible between the two extremes of a pure object-based C++ design and a modular, but strict C implementation.

As an example, consider the action of drawing a line into an image. Using C++, we might have a class and method prototyped as below. Creating a Line object and calling `line.draw(src, color)` would draw a line into the image of the specified color.

```cpp
class Line {
public:
  Point a;
  Point b;

  Line(const Point &a, const Point &b);
  int draw(Image &src, const Color &c);
};

int Line::draw(Image &src, const Color &c) {
  // all of the required information is in the Line class or Image class
  // draw the line from a to b with color c
  return(0);
}
```

The straight C code below has identical functionality and about the same level of modularity. In the main program, calling `drawLine(line, src, color)` with a Line structure, an Image and a Color will draw the line in the image.

```c
typedef struct {
  Point a;
  Point b;
} Line;

int drawLine(Line *line, Image *src, Color *b) {
  // all of the required information is in the Line or Image structures
  // draw the line from a to b with color c
  return(0);
}
```

The difference between the two is that in C all of the information required by a function must be explicitly passed through the parameters. In C++, the object on which a method is called can provide some, if not all of the information. Note also that in C passing by reference is not an option: all C parameters are passed by value, which means copies of the parameters get passed to the function. In C++, passing by reference is possible. Note that in both C and C++ we want to avoid passing whole data structures.

©2006 Bruce A. Maxwell
2 Image

The image is a basic object in computer graphics. Conceptually, it is a canvas on which object primitives can draw themselves. A useful way of thinking about the image is to treat it as a storage device that holds pixel data. Other objects can write to or read from the image as necessary, modifying the values stored in the image. An image needs to know how to read from and write itself to a file.

Use the Pixel definition from ppmIO.h as the basis for the Image type. You can just include ppmIO.h into your Image.h file. The Pixel type is defined as below.

typedef struct {
  unsigned char r;
  unsigned char g;
  unsigned char b;
} Pixel;

Image Fields

- data: pointer or double pointer to space for storing Pixels
- zbuffer: pointer or double pointer to space for storing depth values (use floats)
- rows: number of rows in the image
- cols: number of columns in the image
- maxval: (optional) maximum value for a pixel
- filename: (optional) char array to hold the filename of the image

2.1 Image Methods

Constructors and destructors:

- **Image()** – a NULL constructor that initializes the fields to appropriate values.
- **Image(int rows, int cols)** – allocates space for the image data given rows and columns. Allocate space for the z-buffer and initialize it to an appropriate value, such as 1.0.
- **Image(char *filename)** – reads a PPM image from the given filename. An optional extension is to determine the image type from the last three characters of the filename.
- **~Image()** – deletes allocated image data (if any).

I/O functions:

- **int read(char *filename)** – reads a PPM image from the given filename. Needs to properly handle case where the Image structure has already been allocated. As above, an optional extension is to determine the image type from the filename. Allocates space for and initializes the zbuffer. Returns 0 on success.
- **int write(char *filename)** – writes a PPM image to the given filename. Returns 0 on success.
- **int write(char *filename, int type)** – (optional) writes an image of the specified type to the given filename. The two functions could be the same actual function with type being an optional argument with the default value of a PPM image type. Returns 0 on success.
Access (you may want to inline these):

- `Pixel &pix(int i)` – treating the image as a 1-D array, returns a reference to the ith Pixel.
- `Pixel &pix(int r, int c)` – returns a reference to the Pixel at `(row, column) = (r, c)`;
- You may also give the programmer access to the image data directly. You may choose whether to organize the image data as a 1-D single pointer or a 2-D double-pointer.
- `float &z(int i)` – treating the z-buffer as a 1-D array, returns a reference to the ith element.
- `float &z(int r, int c)` – returns a reference to the z-buffer at `(row, column) = (r, c)`;

Fractal Imagery methods

- `int mandelbrot(float x0, float y0, float x1, float y1, int rows)` – Creates a Mandelbrot set out of the rectangle specified on the complex plane. This function should destroy any existing data in the image structure and build the new Mandelbrot set image. Returns 0 on success.
- `int julia(float x0, float y0, float x1, float y1, float cx, float cy, int rows)` – Creates a Julia set out of the rectangle specified on the complex plane using the specified values for `c`. This function should destroy any existing data in the image structure and build the new Julia set image. Returns 0 on success.