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.

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 deep 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 be with color c
    return(0);
}
```
Image Specification

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. If you use C++, you may convert the Pixel type into a class and create appropriate methods for it.

Image Fields

- data: pointer or double pointer to space for storing Pixels
- int rows: number of rows in the image
- int cols: number of columns in the image
- filename: (optional) char array to hold the filename of the image

C++ Method Specification

Use this set of specifications if you decide to use C++.

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.
- 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. 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.
C Function Specification

Use this set of specifications if you decide to use straight C.

Constructors and destructors:

- Image *Image_create() – Allocates an Image structure and initializes the fields to appropriate values. Returns a pointer to the allocated Image structure. Returns a NULL pointer if the operation fails.

- Image *Image_init(int rows, int cols) – allocates space for the image data given rows and columns and returns a pointer to an Image structure. Returns a NULL pointer if the operation fails.

- void Image_free(Image *src) – de-allocates image data and resets Image fields.

I/O functions:

- Image *Image_read(char *filename) – reads a PPM image from the given filename. An optional extension is to determine the image type from the filename. Returns a NULL pointer if the operation fails.

- int Image_writePPM(Image *src, char *filename) – writes a PPM image to the given filename. Returns 0 on success.

- int Image_write(Image *src, char *filename, int type) – (optional) writes an image of the specified type to the given filename. If you write this function, have the Image_write() function above call it with the PPM type. Returns 0 on success.

Access (you may want to inline these):

- Pixel Image_get1D(Image *src, int i) – returns the value of the ith Pixel.

- Pixel Image_get(Image *src, int r, int c) – returns the value of pixel (r, c).

- void Image_set1D(Image *src, Pixel p, int i) – sets the value of the ith Pixel to p.

- void Image_set(Image *src, Pixel p, int r, int c) – sets the value of Pixel (r, c) to p.

- 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.

Color

As we move into shading and 3D color calculations, it will be important to use floating point math rather than integer math to represent colors. Therefore, you will want to create a Color type that is separate from the Pixel type. You may also want to create functions that convert between the two representations. Color, which calculating shading, is represented on a [0..1] scale while the Pixel type is 0..255.

A simple way to define a Color in C is as an array of floats.

```plaintext
typedef float Color[3];
```

In C++ you may also create a class for Color, which will enable you to do things like define multiplication, addition, subtraction, and other operators. If Color is a C++ class, be sure to define the operator[] so that it is possible to access the individual color values like a simple array.
Color C++ definition

The following is an example of how you might want to create a Color class in C++.

```cpp
#ifndef __COLOR_H
#define __COLOR_H

#include <iostream>
#include "ppmIO.h"

class Color {
    public:
        float c[3];

        // constructors and destructors
        Color() {} // do nothing on construction
        Color(float r, float g, float b) {
            c[0] = r; c[1] = g; c[2] = b;
        }
        Color(const Pixel &p) {
            c[0] = float(p.r) / 255.0;
            c[1] = float(p.g) / 255.0;
            c[2] = float(p.b) / 255.0;
        }

        inline float &operator[](int i) {
            return(c[i]);
        }

        inline Color operator*(const Color &a) {
            Color q(a.c[0] * c[0], a.c[1] * c[1], a.c[2] * c[2]);
            return(q);
        }

        // pre-multiply by a constant
        inline Color operator*(const float a, const Color &c) {
            Color q(a * c.c[0], a * c.c[1], a * c.c[2]);
            return(q);
        }

        // post-multiply by a constant
        inline Color operator*(const Color &c, const float a) {
            Color q(a * c.c[0], a * c.c[1], a * c.c[2]);
            return(q);
        }

        // print a Color using the << operator
        inline std::ostream &operator<<(std::ostream &os, const Color &right) {
            os << right.c[0] << " " << right.c[1] << " " << right.c[2];
            return(os);
        }
};
#endif
```

©2006 Bruce A. Maxwell
Primitive Objects

Primitive objects like pixels, lines, circles, and polygons must hold enough information to know where and how to draw themselves in an image. The primitives Point, Line, Circle, and Ellipse are required for this assignment. The minimum fields required for each type are listed below. Note that on this assignment all z-values will be ignored. However, we’ll need them for 3D in a few weeks. Why we’re using 4-element vectors for 3D will become clear soon.

In C++ you can make the Point type a class and define an operator[] to access the values. In C, while you could use typedef double Point[4];, for the spec it is probably easier to use a struct.

Point fields

Line fields
- Point a – starting point
- Point b – ending point

Circle fields
- double r – radius,
- Point c – center

Ellipse fields
- double ra – major axis radius
- double rb – minor axis radius
- Point c – center
- double a – (optional) angle of major axis relative to the X-axis

C++ Method Specification

Point
- Point() – initialize a Point to the zero point.
- Point(double x, double y) – set the first two values of the vector to x and y. Set the third value to zero and the fourth value to 1.0.
- Point(double x, double y, double z, double h) – set the four values of the vector to x, y, z, and h, respectively.
- void draw(Image *src, Pixel p) – draw the point into the image using color p.

Line
- Line() – initialize a line to two zero points
- Line(int x0, int y0, int x1, int y1) – initialize a 2D line to \((x0,y0)\) and \((x1,y1)\).
- `Line(const Point &ta, const Point &tb)` – initialize a line to `ta` and `tb`.
- `void draw(Image *src, Pixel p)` – draw the line into `src` using color `p`.

**Circle**

- `Circle()` – initialize a circle to zero center and zero radius.
- `Circle(const Point &tc, double tr)` – initialize a circle to location `tc` and radius `tr`.
- `void draw(Image *src, Pixel p)` – draw the circle into `src` using color `p`.
- `void drawFill(Image *src, Pixel p)` – draw a filled circle into `src` using color `p`.

**Ellipse**

- `Ellipse()` – Initialize an ellipse to zero center and zero radii.
- `Ellipse(const Point &tc, double ta, double tb)` – initialize an ellipse to location `tc` and radii `ta` and `tb`.
- `void draw(Image *src, Pixel p)` – draw the ellipse into `src` using color `p`.
- `void drawFill(Image *src, Pixel p)` – draw a filled ellipse into `src` using color `p`.

**C Function Specification**

**Point**

- `void Point_set2D(Point *p, double x, double y)` – set the first two values of the vector to `x` and `y`. Set the third value to zero and the fourth value to 1.0.
- `void Point_set(Point *p, double x, double y, double z, double h)` – set the four values of the vector to `x`, `y`, `z`, and `h`, respectively.
- `void Point_draw(Point *p, Image *src, Pixel p)` – draw the point into `src` using color `p`.

**Line**

- `void Line_set2D(Line *l, int x0, int y0, int x1, int y1)` – initialize a 2D line
- `void Line_set(Line *l, Point ta, Point tb)` – initialize a line to `ta` and `tb`.
- `void Line_draw(Line *l, Image *src, Pixel p)` – draw the line into `src` using color `p`.

**Circle**

- `void Circle_set(Circle *c, Point tc, double tr)` – initialize to center `tc` and radius `tr`.
- `void Circle_draw(Circle *c, Image *src, Pixel p)` – draw the circle into `src` using `p`.
- `void Circle_drawFill(Circle *c, Image *src, Pixel p)` – draw a filled circle into `src` using `p`.

**Ellipse**

- `void Ellipse_set(Ellipse *e, Point tc, double ta, double tb)` – initialize an ellipse to location `tc` and radii `ta` and `tb`.
Polygons

Polygons require a more complex type than the other primitive objects because they are variable sized structures. If you are using C++, polygon structures are a good place to begin using the standard template library [STL]. The polygon and polyline structures are similar. However, a polyline structure cannot be filled since it does not necessarily form a closed shape. You may want to put more fields into your polygon (and you will definitely need to later on), but for now these are the minimum required fields.

**Polygon fields (C)**

- int numVertex – Number of vertices
- Point *vertex – vertex information

**Polygon fields (C++ alternative)**

- std::vector<Point> – STL vector holding an array of vertices.

**Polyline fields (C)**

- int numVertex – Number of vertices
- Point *vertex – vertex information

**Polyline fields (C++ alternative)**

- std::vector<Point> – STL vector holding an array of vertices.

**C++ Method Specification**

**Polygon**

- Polygon() – initialize numVertex to 0 and vertex to NULL.
- Polygon(std::vector<
- void set(std::vector<Point> &vlist) – initialize the vertex list to the points in vlist.

**Polyline**

- Polyline() – initialize numVertex to 0 and vertex to NULL.
- Polyline(std::vector<Point> &vlist) – initialize the vertex list to the points in vlist.
- void set(std::vector<Point> &vlist) – initialize the vertex list to the points in vlist.
- void draw(Image *src, Pixel p) – draw the lines defined by the vertex list using color p.
C Function Specification

Polygon

- Polygon *Polygon_create() – returns an allocated Polygon pointer initialized so that numVertex is 0 and vertex is NULL.

- Polygon *Polygon_init(int numV, Point *vlist) – returns an allocated Polygon pointer with the vertex list initialized to the points in vlist.

- void Polygon_set(Polygon *p, int numV, Point *vlist) – initializes the vertex list to the points in vlist.

- void Polygon_drawFrame(Polygon *p, Image *src, Pixel c) – draw the outline of the polygon using color c.

- void Polygon_drawFill(Polygon *p, Image *src, Pixel c) – draw the filled polygon using color c.

Polyline

- Polyline *Polyline_create() – returns an allocated Polyline pointer initialized so that numVertex is 0 and vertex is NULL.

- Polyline *Polyline_init(int numV, Point *vlist) – returns an allocated Polyline pointer with the vertex list initialized to the points in vlist.

- void Polyline_set(Polyline *p, int numV, Point *vlist) – initializes the vertex list to the points in vlist.

- void Polyline_drawFrame(Polyline *p, Image *src, Pixel c) – draw the outline of the polyline using color c.