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 3D 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.
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.

```c
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.
3 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. Colors, which are used for calculating shading, use a range of [0, 1], while Pixels use a range of [0, 255].

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

typedef float Color[3];

However, this representation can also be cumbersome for some tasks (like copying). So the recommended method of making a Color type in C is as below.

typedef struct {
    float c[3];
} Color;

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.

3.1 Color Functions

Define the following function for the Color type.

void Color_calcShading(
    Color *color,
    int numLights,
    Light *light,
    Color *bodyColor,
    Color *surfaceColor,
    float surfaceCoeff,
    Point *vertex,
    Vector *normal,
    Point *viewer)

Given the set of lights and the remaining information, calculate the color of the surface at the vertex. For each of the types of lights, use the following equations to calculate the contribution of each source. Sum all of the light source contributions together and then clip the result to the range [0, 1].

Ambient lighting

\[ I_c = bodyColor_c \times light_c \quad \forall c \in \{R, G, B\} \]  \hspace{1cm} (1)

Point lighting:

\[ I = B + S \]  \hspace{1cm} (2)

\[ B_c = bodyColor_c \times light_c \times (normal \times (light_{position} - \text{vertex})) \quad \forall c \in \{R, G, B\} \]  \hspace{1cm} (3)

\[ S_c = surfaceColor_c \times light_c \times \left( \frac{(\text{viewer} - \text{vertex}) + light_{position}}{||\text{viewer} - \text{vertex} + light_{position}||} \times normal \right) \times surfaceCoeff \quad \forall c \in \{R, G, B\} \]  \hspace{1cm} (4)
3.2 Color Definition

The following is an example of how you might want to create a Color class in C++. In addition, create a calcShading member function, as defined above, with the exception of the initial color argument.

```cpp
#ifndef __COLOR_H
#define __COLOR_H
#include <iostream>
#include "ppmI0.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);
  }
  inline Color &operator=(const Color &a) {
    c[0] = a.c[0];
    c[1] = a.c[1];
    c[2] = a.c[2];
    return(*this);
  }
};

// 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
```

©2008 Bruce A. Maxwell
4 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. You are also free to just use a struct.

**Point fields**

**Line fields**
- int zBuffer; – whether to use the z-buffer, should default to true (1).
- 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

4.1 Method Specifications

**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 zBufferSet(int flag) – set the z-buffer flag to the given value.
- int zBufferGet(void) – get the z-buffer flag’s current value.
- void draw(Image *src, Pixel p) – draw the line into src using color p. If the z-buffer flag is set, the algorithm should take into account z-buffer values when drawing the line.

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.
5 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 (option 1)

- int zBuffer; – whether to use the z-buffer; should default to true (1)
- int numVertex – Number of vertices
- Point *vertex – vertex information
- Vector *normal – surface normal information

Polygon fields (option 2)

- int zBuffer; – whether to use the z-buffer; should default to true (1)
- std::vector<Point> – STL vector holding an array of vertices.
- std::vector<Vector> – STL vector holding an array of surface normals.

Polyline fields (option 2)

- int zBuffer; – whether to use the z-buffer; should default to true (1).
- int numVertex – Number of vertices
- Point *vertex – vertex information

Polyline fields (option 2)

- int zBuffer; – whether to use the z-buffer; should default to true (1)
- std::vector<Point> – STL vector holding an array of vertices.
5.1 Method Specification

Polygon

- Polygon() – initialize numVertex to 0 and vertex to NULL.
- Polygon(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 set(std::vector<Point> &vlist, std::vector<Vector> &nlist) – initializes the vertices and normals to the given values.
- void zBuffer(int flag) – sets the z-buffer flag to the given value.
- void drawFill(Image *src, Pixel p) – draw the filled polygon using color c. At each pixel the algorithm checks the z-buffer and draws the pixel only if the z-value of the polygon is in front of the existing z-value in the z-buffer. Remember to interpolate $\frac{1}{2}$, rather than $z$ when using perspective projection.
- void shadeFill(Image *src, Color *clist) – draw the filled polygon by interpolating the colors provided for each vertex in the array clist. At each pixel the function should check the z-buffer and draw the pixel only if the z-value is in front of the existing z-value. Interpolate the color as a homogeneous vector $[r\ g\ b\ 1]$ when using perspective projection.

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 zBuffer(int flag) – sets the z-buffer flag to the given value.
- void draw(Image *src, Pixel p) – draw the lines defined by the vertex list using color p. If the zBuffer flag is set, should take into account the z-buffer values when drawing lines.