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.

```c
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 Functions

**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. Allocate space for the z-buffer and initialize it to an appropriate value, such as 1.0. 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 and permit the use of different file types. Allocates space for and initializes the zbuffer. 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.

• `float Image_zget1D(Image *src, int i)` — returns the value of the ith z-buffer element.

• `float Image_zget(Image *src, int r, int c)` — returns the value of z-buffer (r, c).

• `void Image_zset1D(Image *src, float z, int i)` — sets the value of the ith z-buffer element to z.

• `void Image_zset(Image *src, float z, int r, int c)` — sets the value of z-buffer (r, c) to z.

**Fractal Imagery**

• `Image *Image_mandelbrot(float x0, float y0, float x1, float y1, int rows)` — Creates a Mandelbrot set out of the rectangle specified on the complex plane.

• `Image *Image_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.
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.

```c
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.

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

3.1 Color Functions

Define the following functions for the Color type.

```c
void Color_copy(Color *to, Color *from) – copies the Color data.

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**

\[
I_c = \text{bodyColor}_c \times \text{light}_c \quad \forall c \in \{R, G, B\} \tag{1}
\]

**Point lighting:**

\[
I = B + S \tag{2}
\]

\[
B_c = \text{bodyColor}_c \times \text{light}_c \times (\text{normal} \times (\text{light}_\text{position} \text{vertex})) \quad \forall c \in \{R, G, B\} \tag{3}
\]

\[
S_c = \text{surfaceColor}_c \times \text{light}_c \times \left(\frac{(\text{viewer} - \text{vertex} + \text{light}_\text{position}) \times \text{normal}}{\|\text{viewer} - \text{vertex} + \text{light}_\text{position}\|^2}\right)^{\text{surfaceCoeff}} \quad \forall c \in \{R, G, B\} \tag{4}
\]
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. 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

• 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 Primitive Functions

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 c) – 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_zBuffer(int flag) – set the z-buffer flag to the given value.
- void Line_draw(Line *l, 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

- 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.
- void Ellipse_drawFill(Ellipse *e, Image *src, Pixel p) – draw a filled ellipse into src using color p.
5 Poly Objects

Polygons require a more complex type than the other primitive objects because they are variable sized structures. The polygon and polyline structures are similar. However, a polyline structure cannot be filled with a color since it does not necessarily form a closed shape. As you develop your system, you may want to add more fields into your polygon than the required fields listed below.

Polygon fields (C)

- `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

Polyline fields (C)

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

5.1 Poly Functions

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_delete(Polygon *p)` - frees the internal data and the Polygon pointer.
- `void Polygon_setNULL(Polygon *p)` - initializes the existing Polygon to an empty Polygon.
- `void Polygon_set(Polygon *p, int numV, Point *vlist)` - initializes the vertex list to the points in vlist.
- `void Polygon_setAll(Polygon *p, int numV, Point *vlist, Vector *nlist)` - initializes the vertices and normals to the given values.
- `void Polygon_zBuffer(Polygon *p, int flag)` - sets the z-buffer flag to the given value.
- `void Polygon_copy(Polygon *to, Polygon *from)` - Allocates space and copies the vertex and normal data from one polygon to the other.
- `void Polygon_free(Polygon *p)` - frees the internal data for a Polygon.
- `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. 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 \( \frac{1}{z} \), rather than \( z \) when using perspective projection.

void Polygon_shadeFill(Polygon *p, 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 \( \begin{bmatrix} r \\ g \\ b \\ 1 \end{bmatrix} \) when using perspective projection.

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_delete(Polyline *p) – frees the data and the polyline pointer.

void Polyline_setNULL(Polygon *p) – initializes the existing Polyline to an empty Polyline.

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

void Polyline_zBuffer(Polyline *p, int flag) – sets the z-buffer flag to the given value.

void Polyline_copy(Polyline *to, Polyline *from) – Allocates space and copies the vertex data from one polygon to the others.

void Polyline_free(Polyline *p) – frees the internal data for a polyline.

void Polyline_drawFrame(Polyline *p, Image *src, Pixel c) – draw the outline of the polyline using color c. If the zBuffer flag is set, should take into account the z-buffer values when drawing lines.