# 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
- Z-buffer - a depth image, usually part of the Image data structure
- 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 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.

```
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 `lineDraw(line, src, color)` with a Line structure, an Image and a Color will draw the line in the image.

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

int lineDraw(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 color data. In graphics we may want our image to also hold depth data–distance from the camera. 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. Note that the data type of the file does not need to match the internal data type of the image.

To represent a single pixel, we need at least three numbers for color images: R, G, and B, corresponding to the red, green, and blue values of the pixel. In graphics, we are also often mixing images together through the use of an alpha channel. The alpha value of a pixel indicates its transparency. We are also often dealing with objects that have an associated depth, so we know what surfaces are in front of what other surfaces. You will want to use a float type to represent the RGB, alpha, and depth values in your image. You can organize the data as you see fit.

One possible organization method is to create an FPixel type like the following with all five values interleaved through the image data structure. An alternative approach would be to have an FPixel data structure that has just the RGB values and then create separate alpha and z fields in your Image data structure (preferred).

**Option 1**

```
typedef struct {
  float rgb[3];
  float a;
  float z;
} FPixel;
```

**Option 2**

```
typedef struct {
  float rgb[3];
 } FPixel;
```

Then you can use a single or double pointer of FPixel type to represent all of the necessary image data.

**Image Fields**

- data: pointer or double pointer to space for storing Pixels (e.g. FPixel type)
- rows: number of rows in the image
- cols: number of columns in the image
- depth: (optional) store your z values here, this would be a float pointer (array)
- alpha: (optional) store your alpha values here, this would be a float pointer (array)
- maxval: (optional) maximum value for a pixel
- filename: (optional) char array to hold the filename of the image

 

## 2.1 Image Functions

All of the Image functions either take in an Image pointer (Image *) as an argument or return a new Image pointer. You never want to pass an Image into a function, always pass in an Image pointer. In general, if you're passing around any data structure bigger than a float or a double, use a pointer to avoid copying the contents of the struct.

**Constructors and destructors:**

- `Image *image_create(int rows, int cols)` – Allocates an Image structure and initializes the top level fields to appropriate values. Allocates space for an image of the specified size, unless either rows or cols is 0. Returns a pointer to the allocated Image structure. Returns a NULL pointer if the operation fails.
- `void image_free(Image *src)` – de-allocates image data and frees the Image structure.
- `void image_init(Image *src)` – given an uninitialized Image structure, sets the rows and cols fields to zero and the data field to NULL.
- `int image_alloc(Image *src, int rows, int cols)` – allocates space for the image data given rows and columns and initializes the image data to appropriate values, such as 0.0 for RGB and 1.0 for A and Z. Returns 0 if the operation is successful. Returns a non-zero value if the operation fails. This function should free existing memory if rows and cols are both non-zero.
- `void image_dealloc(Image *src)` – de-allocates image data and resets the Image structure fields. The function does not free the Image structure.

**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. Initializes the alpha channel to 1.0 and the z channel to 1.0. Returns a NULL pointer if the operation fails.
- `int image_write(Image *src, char *filename)` – writes a PPM image to the given filename. Returns 0 on success. Optionally, you can look at the filename extension and write different file types.

 

**Access**

- `FPixel image_getf(Image *src, int r, int c)` – returns the FPixel at (r, c).
- `float image_getc(Image *src, int r, int c, int b)` – returns the value of band b at pixel (r, c).
- `float image_geta(Image *src, int r, int c)` – returns the alpha value at pixel (r, c).
- `float image_getz(Image *src, int r, int c)` – returns the depth value at pixel (r, c).
- `void image_setf(Image *src, int r, int c, FPixel val)` – sets the values of pixel (r, c) to the FPixel val. If your FPixel contains just r, g, b values then this function should not modify the corresponding alpha or z values.
- `void image_setc(Image *src, int r, int c, int b, float val)` – sets the value of pixel (r, c) band b to val.
- `void image_seta(Image *src, int r, int c, float val)` – sets the alpha value of pixel (r, c) to val.
- `void image_setz(Image *src, int r, int c, float val)` – sets the depth value of pixel (r, c) to val.
- The programmer can also 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.
- You are welcome to define macros for access functions instead of true functions. You can also try using the *inline* keyword, which tells the compiler to substitute the code in the function for the function call (eliminating the function call) at compile time.

**Utility**

- `void image_reset(Image *src)` – resets every pixel to a default value (e.g. Black, alpha value of 1.0, z value of 1.0).
- `void image_fill(Image *src, FPixel val)` – sets every FPixel to the given value.
- `void image_fillrgb(Image *src, float r, float g, float b)` – sets the (r, g, b) values of each pixel to the given color.
- `void image_filla(Image *src, float a)` – sets the alpha value of each pixel to the given value.
- `void image_fillz(Image *src, float z)` – sets the z value of each pixel to the given value.

 

# 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. The Color type is the same as the RGB component of an FPixel. You may also want to create functions that convert between an integer and a float representation, doing appropriate scaling. Colors, which are used for calculating shading, use a range of [0, 1], while Pixels for writing to an integer-based file format generally use a range of [0, 255], although 16-bits per pixel is becoming more common.

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;
```

A simple assignment will copy a color if you use the struct definition.

## 3.1 Color Functions

Define the following functions for the Color type.

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

`void color_set(Color *to, float r, float g, float b)`
– sets the Color data.

In addition, add the following functions for an Image.

`void image_setColor( Image *src, int r, int c, Color val )`
– copies the Color data to the proper pixel.

`Color image_getColor( Image *src, int r, int c )`
– returns a Color structure built from the pixel values.

 