

# 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 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 rows or cols is 0, in which case it should set both rows and cols to 0 and the data pointers to NULL. Returns a pointer to the allocated Image structure. Returns a NULL pointer only if a malloc 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.
```

## 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, while you could use `typedef double Point[4];`, it is probably easier to use a struct.

### Point fields

- `double val[4]` – four element vector of doubles.

### 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

### 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

## 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 0.0 and the fourth value to 1.0.
- `void point_set3D(Point *p, double x, double y, double z)`  
– set the point's values to `x` and `y` and `z`. Set the homogeneous coordinate 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_normalize(Point *p)`  
– normalize the `x` and `y` values of a point by its homogeneous coordinate:  $x = x/h$ ,  $y = y/h$ .
- `void point_copy(Point *to, Point *from)`  
– copy the point data structure.
- `void point_draw(Point *p, Image *src, Color c)`  
– draw the point into `src` using color `c`.
- `void point_drawf(Point *p, Image *src, FPixel c)`  
– draw the `p` into `src` using `FPixel c`.
- `void point_print(Point *p, FILE *fp)`  
– use `fprintf` to print the contents of the `Point` to the stream `fp`. The `FILE` pointer `fp` can be an actual file opened using `fopen` or a standard output stream like `stdout` or `stderr`, both defined in `stdio.h`. The `point_print` function doesn't care which it is, it just uses `fprintf`, which takes a `FILE` pointer as the first argument.

### Line

- `void line_set2D(Line *l, double x0, double y0, double x1, double y1)`  
– initialize a 2D line.
- `void line_set(Line *l, Point ta, Point tb)`  
– initialize a line to `ta` and `tb`.
- `void line_zBuffer(Line *l, int flag)`  
– set the z-buffer flag to the given value.
- `void line_normalize(Line *l)`  
– normalize the `x` and `y` values of the endpoints by their homogeneous coordinate.
- `void line_copy(Line *to, Line *from)`  
– copy the line data structure.
- `void line_draw(Line *l, Image *src, Color c)`  
– draw the line into `src` using color `c` and the z-buffer, if appropriate.



**Circle**

- `void circle_set(Circle *c, Point tc, double tr)`  
– initialize to center tc and radius tr.
- `void circle_draw(Circle *c, Image *src, Color p)`  
– draw the circle into src using color p.
- `void circle_drawFill(Circle *c, Image *src, Color p)`  
– draw a filled circle into src using color 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_draw(Ellipse *e, Image *src, Color p)`  
– draw into src using color p.
- `void ellipse_drawFill(Ellipse *e, Image *src, Color p)`  
– draw a filled ellipse into src using color p.

## Polyline

The functions `polyline_create` and `polyline_free` manage both the Polyline data structure and the memory required for the vertex list.

- `Polyline *polyline_create()`  
– returns an allocated Polyline pointer initialized so that `numVertex` is 0 and `vertex` is NULL.
- `Polyline *polyline_createp(int numV, Point *vlist)`  
– returns an allocated Polyline pointer with the vertex list initialized to the points in `vlist`.
- `void polyline_free(Polyline *p)`  
– frees the internal data and the Polyline pointer.

The functions `polyline_init`, `polyline_set`, and `polyline_clear` work on a pre-existing Polyline data structure and manage only the memory required for the vertex list.

- `void polyline_init(Polyline *p)`  
– initializes the pre-existing Polyline to an empty Polyline.
- `void polyline_set(Polyline *p, int numV, Point *vlist)`  
– initializes the vertex list to the points in `vlist`. De-allocates/allocates the vertex list for `p`, as necessary.
- `void polyline_clear(Polyline *p)`  
– frees the internal data for a Polyline, if necessary, and sets `numVertex` to 0 and `vertex` to NULL.
- `void polyline_zBuffer(Polyline *p, int flag)`  
– sets the z-buffer flag to the given value.
- `void polyline_copy(Polyline *to, Polyline *from)`  
– De-allocates/allocates space as necessary in the destination Polyline data structure and copies the vertex data from the source polyline (from) to the destination (to).
- `void polyline_print(Polyline *p, FILE *fp)`  
– prints Polyline data to the stream designated by the FILE pointer.
- `void polyline_normalize(Polyline *p)`  
– normalize the x and y values of each vertex by the homogeneous coordinate.
- `void polyline_draw(Polyline *p, Image *src, Color c)`  
– draw the polyline using color `c` and the z-buffer, if appropriate.

## 5 Polygon

Polygons require a more complex type than the other primitive objects because they are variable sized structures. Polygons are very similar to Polylines, but they can also be filled with a color. It is assumed that the last vertex of a polygon connects back to the first. 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 oneSided` – whether to consider the polygon one-sided (1) or two-sided (0) for shading
- `int nVertex` – Number of vertices
- `Point *vertex` – vertex information
- `Color *color` – color information for each vertex.
- `Vector *normal` – surface normal information for each vertex.
- `int zBuffer;` – whether to use the z-buffer; should default to true (1)

### 5.1 Polygon Functions

#### Polygon

The functions `polygon_create` and `polygon_free` manage both the Polygon data structure and the memory required for the vertex list.

- `Polygon *polygon_create()`  
– returns an allocated Polygon pointer initialized so that `numVertex` is 0 and `vertex` is NULL.
- `Polygon *polygon_createp(int numV, Point *vlist)`  
– returns an allocated Polygon pointer with the vertex list initialized to a copy of the points in `vlist`.
- `void polygon_free(Polygon *p)`  
– frees the internal data for a Polygon and the Polygon pointer.

The functions `polygon_init`, `polygon_set`, and `polygon_clear` work on a pre-existing Polygon data structure and manage only the memory required for the vertex list.

- `void polygon_init(Polygon *p)`  
– initializes the existing Polygon to an empty Polygon.
- `void polygon_set(Polygon *p, int numV, Point *vlist)`  
– initializes the vertex array to the points in `vlist`.
- `void polygon_clear(Polygon *p)`  
– frees the internal data and resets the fields.
- `void polygon_setSided(Polygon *p, int oneSided)`  
– sets the `oneSided` field to the value.
- `void polygon_setColors(Polygon *p, int numV, Color *clist)`  
– initializes the color array to the colors in `clist`.

- `void polygon_setNormals(Polygon *p, int numV, Vector *nlist)`  
– initializes the normal array to the vectors in nlist.
- `void polygon_setAll(Polygon *p, int numV, Point *vlist, Color *clist, Vector *nlist, int zBuffer, int oneSided)`  
– initializes the vertex list to the points in vlist, the colors to the colors in clist, the normals to the vectors in nlist, and the zBuffer and oneSided flags to their respective values.
- `void polygon_zBuffer(Polygon *p, int flag)`  
– sets the z-buffer flag to the given value.
- `void polygon_copy(Polygon *to, Polygon *from)`  
– De-allocates/allocates space and copies the vertex and color data from one polygon to the other.
- `void polygon_print(Polygon *p, FILE *fp)`  
– prints polygon data to the stream designated by the FILE pointer.
- `void polygon_normalize( Polygon *p )`  
– normalize the x and y values of each vertex by the homogeneous coord.
- `void polygon_draw(Polygon *p, Image *src, Color c)`  
– draw the outline of the polygon using color c
- `void polygon_drawFill(Polygon *p, Image *src, Color c)`  
– draw the filled polygon using color c with the scanline z-buffer rendering algorithm.
- `void polygon_drawFillB(Polygon *p, Image *src, Color c)`  
– draw the filled polygon using color c with the Barycentric coordinates algorithm.
- `void polygon_drawShade(Polygon *p, Image *src, DrawState *ds, Lighting *light)`  
– draw the filled polygon using the given DrawState. The shade field of the DrawState determines how the polygon should be rendered. The Lighting parameter should be NULL unless you are doing Phong shading.

The following are the various different types of drawing modes. ShadePhong is optional.

- ShadeFrame: draw only the outline of the polygon using the DrawState color field (call drawFrame)
- ShadeConstant: fill the polygon with the DrawState color field (call drawFill).
- ShadeDepth: fill the polygon based on the depth value, which should be in the range [0, 1] as 1 is the back clip plane in the canonical view space.
- ShadeFlat: fill the polygon with a single color that is the result of a shading calculation.
- ShadeGouraud: fill the polygon using a Gouraud shading method and the colors in the polygon data structure.
- ShadePhong: fill the polygon using a Phong shading method based on the normals in the polygon data structure, the parameters in the DrawState, and the lighting in the Lighting data structure.
- `void polygon_shade(Polygon *p, Lighting *lighting, DrawState *ds)` – For the ShadeFlat and ShadeGouraud cases of the shade field of DrawState, calculate colors at each vertex and create and fill out the color array of the Polygon data structure. For ShadeFlat, use the average surface normal and average polygon location to calculate one color and set the color at each vertex to the calculated value. For ShadeGouraud use the surface normals and locations of each vertex.