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 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 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. Allocates space for the z-buffer and initialize it to an approprate 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.

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;

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_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]. Each of the following equations should be executed for each color channel. $C_b$ is the body color of the surface, and $C_s$ is the surface color. The latter is generally a dark grey like (0.2, 0.2, 0.2). $C_{L_d}$ is the direct illuminant color, $\vec{L}$ is the light vector, $\vec{N}$ is the surface normal, $\vec{V}$ is the view vector, and $\vec{H}$ is the halfway vector.

Ambient lighting

$$I_a = C_{L_d} C_b$$  \hspace{1cm} (1)

Point lighting:

$$I = \text{Body} + \text{Surface} = C_b C_{L_d} (\vec{L} \cdot \vec{N}) + C_{L_d} C_s (\vec{H} \cdot \vec{N})^n$$  \hspace{1cm} (2)

$$\vec{H} = \frac{\vec{L} + \vec{V}}{||\vec{L} + \vec{V}||} \approx \frac{\vec{L} + \vec{V}}{2}$$  \hspace{1cm} (3)
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(Line *l, 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_draw(Ellipse *e, Image *src, Pixel p)` — draw into src using color p.
- `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_print(Polygon *p, FILE *fp)` - prints polygon data to the stream designated by the FILE pointer.
- `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 $[r\ b\ g\ z\ 1]$ 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::print(Polyline *p, FILE *fp)` — prints polyline data to the stream designated by the FILE pointer.

- `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.
6 Transformation Matrices

Transform matrices should be 4x4 matrices of doubles. Use a structure (C) with a 4x4 array in the structure. In C, the type could be declared as below.

typedef struct {
    double m[4][4];
} Matrix;

You will also need a Vector type, as below.

typedef struct {
    double v[4];
} Vector;

Vectors should have a zero as their homogeneous coordinate so they do not undergo translations. You can get away with only three values as part of the Vector class (because $h = 0$). The implementation details are up to you. Note that the Vector_set function only takes three arguments $(x,y,z)$. If you wish, you can just define Vector and Point to be the same thing.

typedef Point Vector;

6.1 Vector Functions

- void Vector_set(Vector *v, double x, double y, double z) – Set the Vector to $(x,y,z,0.0)$.
- void Vector_print(Vector *v, FILE *fp) – Print out the Vector.
- void Vector_copy(Vector *dest, Vector *src) – Copy the src Vector into the dest Vector.
- double Vector_length(Vector *v) – Returns the length of the vector.

$$L = \|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

- void Vector_normalize(Vector *v) – Normalize the Vector to unit length.

$$\hat{v} = \frac{\vec{v}}{L}$$

- double Vector_dot(Vector *a, Vector *b) – Returns the scalar product of $\vec{a}$ and $\vec{b}$.

$$d(a,b) = a_xb_x + a_yb_y + a_zb_z$$

- void Vector_cross(Vector *a, Vector *b, Vector *c) – Calculates the the cross product of $\vec{a}$ and $\vec{b}$ and puts the result in $\vec{c}$.

$$\begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} = \vec{a} \times \vec{b} = \begin{bmatrix} a_yb_z - a_zb_y \\ a_zb_x - a_xb_z \\ a_xb_y - a_yb_x \end{bmatrix}$$

©2008 Bruce A. Maxwell
6.2 2D and Generic Matrix Functions

The following functions should be defined for matrices.

- `void Matrix_print(Matrix *m, FILE *fp)` — Print out the matrix in a nice 4x4 arrangement with a blank line below.
- `void Matrix_clear(Matrix *m)` — Set the matrix to all zeros.
- `void Matrix_identity(Matrix *m)` — Set the matrix to the identity matrix.
  \[
  M = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1 \\
  \end{bmatrix}
  \]  

- `double Matrix_get(Matrix *m, int r, int c)` — Return the element of the matrix at row r, column c.
- `void Matrix_set(Matrix *m, int r, int c, double v)` — Set the element of the matrix at row r, column c to v.
- `void Matrix_copy(Matrix *dest, Matrix *src)` — Copy the src matrix into the dest matrix.
- `void Matrix_transpose(Matrix *m)` — Transpose the matrix m in place.
- `void Matrix_multiply(Matrix *left, Matrix *right, Matrix *m)` — Multiply left and right and put the result in m.
  \[
  [M] = [\text{left}] [\text{right}]
  \]  

Make sure that the function is written so that the result matrix can also be the left or right matrices.
- `void Matrix_xformPoint(Matrix *m, Point *p, Point *q)` — Transform the point p by the matrix m and put the result in q. For this function, p and q need to be different variables.
  \[
  \vec{q} = M\vec{p}
  \]  

- `void Matrix_xformVector(Matrix *m, Vector *p, Vector *q)` — Transform the vector p by the matrix m and put the result in q. For this function, p and q need to be different variables.
- `void Matrix_xformPolygon(Matrix *m, Polygon *p)` — Transform the points and surface normals (if they exist) in the Polygon p by the matrix m.
- `void Matrix_xformPolyline(Matrix *m, Polyline *p)` — Transform the points in the Polyline p by the matrix m.
- `void Matrix_xformLine(Matrix *m, Line *line)` — Transform the points in line by the matrix m.
- `void Matrix_scale2D(Matrix *m, double sx, double sy)` — Premultiply the matrix by a scale matrix parameterized by sx and sy.
\[
M = S(s_x, s_y)M = \begin{bmatrix}
s_x & 0 & 0 \\
0 & s_y & 0 \\
0 & 0 & 1 \\
0 & 0 & 1
\end{bmatrix} M \tag{11}
\]

- void Matrix\_rotateZ(Matrix *m, double cth, double sth) – Premultiply the matrix by a Z-axis rotation matrix parameterized by \(\cos(\theta)\) and \(\sin(\theta)\), where \(\theta\) is the angle of rotation about the Z-axis.

\[
M = R_Z(\cos(\theta), \sin(\theta))M = \begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 & 0 \\
\sin(\theta) & \cos(\theta) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} M \tag{12}
\]

- void Matrix\_translate2D(Matrix *m, double tx, double ty) – Premultiply the matrix by a 2D translation matrix parameterized by \(t_x\) and \(t_y\).

\[
M = T(t_x, t_y)M = \begin{bmatrix}
1 & 0 & 0 & t_x \\
0 & 1 & 0 & t_y \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} M \tag{13}
\]

- void Matrix\_shear2D(Matrix *m, double shx, double shy) – Premultiply the matrix by a 2D shear matrix parameterized by \(sh_x\) and \(sh_y\).

\[
M = Sh(sh_x, sh_y)M = \begin{bmatrix}
1 & sh_x & 0 & 0 \\
sh_y & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} M \tag{14}
\]
6.3 3D Matrix Functions

- `void Matrix_translate(Matrix *m, double tx, double ty, double tz)`—Premultiply the matrix by a translation matrix parameterized by \(t_x, t_y, \) and \(t_z\).

\[
M = T(t_x, t_y, t_z)M = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} M \tag{15}
\]

- `void Matrix_scale(Matrix *m, double sx, double sy, double sz)`—Premultiply the matrix by a scale matrix parameterized by \(s_x, s_y, s_z\).

\[
M = S(s_x, s_y, s_z)M = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M \tag{16}
\]

- `void Matrix_rotateX(Matrix *m, double cth, double sth)`—Premultiply the matrix by a X-axis rotation matrix parameterized by \(\cos(\theta)\) and \(\sin(\theta)\), where \(\theta\) is the angle of rotation about the X-axis.

\[
M = R_X(\cos(\theta), \sin(\theta))M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M \tag{17}
\]

- `void Matrix_rotateY(Matrix *m, double cth, double sth)`—Premultiply the matrix by a Y-axis rotation matrix parameterized by \(\cos(\theta)\) and \(\sin(\theta)\), where \(\theta\) is the angle of rotation about the Y-axis.

\[
M = R_Y(\cos(\theta), \sin(\theta))M = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M \tag{18}
\]

- `void Matrix_rotateXYZ(Matrix *m, Vector *u, Vector *v, Vector *w)`—Premultiply the matrix by an XYZ-axis rotation matrix parameterized by the vectors \(\vec{u}, \vec{v}, \) and \(\vec{w}\), where the three vectors represent an orthonormal 3D basis.

\[
M = R_{XYZ}(\vec{u}, \vec{v}, \vec{w})M = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M \tag{19}
\]
- void Matrix\_shearZ(Matrix *m, double shx, double shy) – Premultiply the matrix by a shear Z matrix parameterized by \(shx\) and \(shy\).

\[
M = Sh_z(sh_x, sh_y)M = \begin{bmatrix}
1 & 0 & shx & 0 \\
0 & 1 & shy & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} M
\]  

(20)

- void Matrix\_perspective(Matrix *m, double d) – Premultiply the matrix by a perspective matrix parameterized by \(d\).

\[
M = Persp(d)M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/d & 0
\end{bmatrix} M
\]  

(21)
7 Hierarchical Modeling

A module is primarily a linked list of elements. Example definitions of a Module and an Element type in C are given below. If you wish, you can create a union that can be any of the basic primitive types {Point, Line, Polygon, Polyline, Matrix, Pixel, and void * (for a Module *)} and use it instead of a void pointer in the Element structure.

typedef enum { // example of an enumerated type
    ObjLine,
    ObjPoint,
    ObjPolyline,
    ObjPolygon,
    ObjIdentity,
    ObjMatrix,
    ObjColor,
    ObjBodyColor,
    ObjSurfaceColor,
    ObjSurfaceCoeff,
    ObjModule
} ObjectType;

// option 1 for Element structure (void * option)
typedef struct {
    ObjectType type; // Type of object stored in the obj pointer
    void *obj; // pointer to the object
    void *next; // next pointer
} Element;

// option 2 for Element structure (union option)
typedef union {
    Point point;
    Line line;
    Polyline polyline;
    Polygon polygon;
    Matrix matrix;
    Pixel pixel;
    Color color;
    float coeff;
    void *module;
} Object;

// Module structure
typedef struct {
    ObjectType type;
    Object obj;
    void *next;
} Element;

typedef struct {
    Element *head; // pointer to the head of the linked list
    Element *tail; // keep around a pointer to the last object
} Module;
7.1 2D and Generic Module Functions

The following set of functions create, add items to, manipulate, draw, and destroy Modules and Elements. Note that most of the Module_add functions will be similar.

- **Element *Element_create()** – Allocate an empty Element.
- **Element *Element_init(ObjectType type, void *obj)** – Allocate an Element and store a duplicate of the data pointed to by obj in the Element. The function needs to handle each type of object separately in a case statement.
- **void Element_delete(Element *e)** – free the element and the object it contains.
- **Module *Module_create()** – Allocate an empty module.
- **void Module_delete(Module *md)** – Free all of the memory associated with a module, including the memory pointed to by md.
- **void Module_clear(Module *md)** – clear the module’s list, freeing memory as appropriate.
- **void Module_draw(Module *md, Matrix *VTM, Matrix *GTM, Pixel color, Image *src)** – Draw the module into the image using the given view transformation matrix [VTM] by traversing the list of Elements.
- **void Module_insert(Module *md, Element *e)** – Generic insert of an element into the module at the tail of the list.
- **void Module_addModule(Module *md, Module *sub)** – Adds a pointer to the Module sub to the tail of the module’s list.
- **void Module_addPoint(Module *md, Point *p)** – Adds p to the tail of the module’s list.
- **void Module_addLine(Module *md, Line *p)** – Adds p to the tail of the module’s list.
- **void Module_addPolyline(Module *md, Polyline *p)** – Adds p to the tail of the module’s list.
- **void Module_addPolygon(Module *md, Polygon *p)** – Adds p to the tail of the module’s list.
- **void Module_identity(Module *md)** – Object that sets the current transform to the identity, placed at the tail of the module’s list.
- **void Module_translate2D(Module *md, double tx, double ty)** – Matrix operand to add a translation matrix to the tail of the module’s list.
- **void Module_scale2D(Module *md, double sx, double sy)** – Matrix operand to add a scale matrix to the tail of the module’s list.
- **void Module_rotateZ(Module *md, double cth, double sth)** – Matrix operand to add a rotation about the Z axis to the tail of the module’s list.
- **void Module_shear2D(Module *md, double shx, double shy)** – Matrix operand to add a 2D shear matrix to the tail of the module’s list.
- **void Module_addColor(Module *md, Pixel p)** – Adds a pixel to the tail of the module’s list.
7.2 3D Module functions

- void Module_translate(Module *md, double tx, double ty, double tz)—Matrix operand to add a 3D translation to the Module.
- void Module_scale(Module *md, double sx, double sy, double sz)—Matrix operand to add a 3D scale to the Module.
- void Module_rotateX(Module *md, double cth, double sth)—Matrix operand to add a rotation about the X-axis to the Module.
- void Module_rotateY(Module *md, double cth, double sth)—Matrix operand to add a rotation about the Y-axis to the Module.
- void Module_rotateXYZ(Module *md, Vector *u, Vector *v, Vector *w)—Matrix operand to add a rotation that orients to the orthonormal axes $\vec{u}, \vec{v}, \vec{w}$.
- void Module_cube(Module *md)— Adds a unit cube, axis-aligned and centered on zero to the Module.

7.3 Shading Module Functions

- void Module_bodyColor(Module *md, Color c)— Adds the body color value to the tail of the module’s list.
- void Module_surfaceColor(Module *md, Color c)— Adds the surface color value to the tail of the module’s list.
- void Module_surfaceCoeff(Module *md, float coeff)— Adds the specular coefficient to the tail of the module’s list.
- void Module_shade(Module *md, Matrix *VTM, Matrix *GTM, Color bodyColor, Color surfaceColor, float surfaceCoeff, int numLights, Light *light, Point *viewer, Image *src)— Draw the module into the image using the given view transformation matrix [VTM] by traversing the list of Elements. Take into account the light sources and execute shading as appropriate.
8 Viewing

8.1 2D Viewing

Define a 2D View structure `2DView` that converts world Cartesian coordinates (y-axis up, x-axis right) into screen coordinates. The structure should have fields for the following data.

- The center of the view rectangle $V_0$ in world coordinates
- The width of the view rectangle $w_v$ in world coordinates
- The orientation angle $\theta_v$ of the view rectangle in world coordinates
- The number of columns $C$ in the output image
- The number of rows $R$ in the output image

Define a function to generate a view transformation matrix VTM. The function should set the VTM to the view specified by the structure.

```c
void Matrix_set2DView(Matrix *vtm, 2DView *view) - Sets vtm to be the view transformation defined by the 2DView structure.
```

The height of the view rectangle in world coordinates is given by:

$$h_v = \frac{w_v C}{R} \quad (22)$$

The VTM is the multiplication of four matrices. The process involves translating the origin of the view window to the origin, orienting the view window with the x-axis, scaling the view window and flipping the y-axis, then shifting the range of y-values back to $[0, R]$.

$$VTM = T(0, \frac{R}{2})S\left(\frac{C}{h_v}, -\frac{R}{w_v}\right)R_z(-\theta_v)T(-V_{0x}, -V_{0y}) \quad (23)$$

![Figure 1: 2D view parameters shown in a world coordinate system.](image)
8.2 Perspective Viewing

Define a PerspectiveView structure/class as below.

```
typedef struct {
    Point vrp;
    Vector vpn;
    Vector vup;
    double d;
    double du;
    double dv;
    double f;
    double b;
    int    screenx;
    int    screeny;
} PerspectiveView;
```

- View Reference Point [VRP]: 3-D vector indicating the origin of the view reference coordinates.
- View Plane Normal [VPN]: 3-D vector indicating the direction in which the viewer is looking.
- View Up Vector [VUP]: 3-D vector indicating the UP direction on the view plane. The only restriction is that it cannot be parallel to the view plane normal.
- Projection Distance [d]: distance in the negative VPN direction at which the center of projection is located.
- View Window Extent [du, dv]: extent of view plane around the VRP, expressed in world coordinate distances.
- Front and Back Clip Planes [F, B]: front and back clip planes expressed as distances along the positive VPN. \( F > 0 \) and \( F < B \).
- Screen Size [screenX, screenY]: Size of the desired image in pixels.

**C Version**: void Matrix_set3DView(Matrix *vtm, PerspectiveView *view) - Implement the 3D perspective pipeline. Inside the function, begin by initializing VTM to the identity. Do not modify any of the values in the PerspectiveView structure inside the function.
9 Lighting

Define a light source and an enumerated type to indicate light source type.

```c
typedef enum {
    LightAmbient,
    LightDirect,
    LightPoint,
    LightSpot,
} LightType;

typedef struct {
    LightType type;
    Color color;
    Vector direction;
    Point position;
    float alpha;
} Light;
```

You may also wish to create a series of functions for initializing and manipulating light sources, including modifying the Matrix functions to transform light source positions and surface normals. In the current implementation, however, they are simply defined as fixed objects in world coordinates and not integrated into the modeling system. Instead, the array of light sources gets passed into the modeling system so that all lights are available for all objects without doing an extra pass through the module array.