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.

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 deep 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);
}
```
Image Specification

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. If you use C++, you may convert the Pixel type into a class and create appropriate methods for it.

Image Fields

- data: pointer or double pointer to space for storing Pixels
- z-buffer: pointer or double pointer to space for storing depth values (probably floats)
- int rows: number of rows in the image
- int cols: number of columns in the image
- filename: (optional) char array to hold the filename of the image

C++ Method Specification

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);

C Function Specification

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. 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.
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. Color, which calculating shading, is represented on a [0..1] scale while the Pixel type is 0..255.

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

typedef float Color[3];

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.
**Color C++ definition**

The following is an example of how you might want to create a Color class in C++.

```cpp
#ifndef __COLOR_H
#define __COLOR_H

#include <iostream>
#include "ppmIO.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);
    }

    // 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
```

©2006 Bruce A. Maxwell
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

**C++ Method Specification**

**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 \(p\). 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 \(p\).

C Function Specification

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 \(p\). 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.

**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 (C)**

• int zBuffer; – whether to use the z-buffer; should default to true (1)

• int numVertex – Number of vertices

• Point *vertex – vertex information

**Polygon fields (C++ alternative)**

• int zBuffer; – whether to use the z-buffer; should default to true (1)

• std::vector<Point> – STL vector holding an array of vertices.

**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

**Polyline fields (C++ alternative)**

• int zBuffer; – whether to use the z-buffer; should default to true (1)

• std::vector<Point> – STL vector holding an array of vertices.
C++ 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 zBuffer(int flag) – sets the z-buffer flag to the given value.
- void drawFill(Image *src, Pixel p) – draw the filled polygon using color p. Modify so that at each pixels it checks the z-buffer and only draws the pixel if the z-value is in front of the existing z-value. Remember to interpolate \( \frac{1}{2} \) 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.

C Function Specification

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_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 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. **Modify so that at each pixels it checks the z-buffer and only draws the pixel if the z-value is in front of the existing z-value. Remember to interpolate $\frac{1}{2}$ 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**
Transform Matrices

Transform matrices should be 4x4 matrices of doubles. Use either a structure (C) or a class (C++) with a 4x4 array in the structure. Using the STL vector class for matrices would be a bit of overkill since the structures are fixed size. In C, the type would be declared as below.

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

C Matrix Specification

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 space below.
• void Matrix_clear(Matrix *m) – Set the matrix to all zeros.
• void Matrix_identity(Matrix *m) – Set the matrix to the identity matrix.
• 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.

\[ m = left \times right \]

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.
• void Matrix_xformPolygon(Matrix *m, Polygon *p) – Transform the points 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 \( s_x \) and \( s_y \).
\[ m = S(s_x, s_y) \cdot m \quad \text{where} \quad S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- `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.

\[ m = R_Z(\cos(\theta), \sin(\theta)) \cdot m \quad \text{where} \quad R_Z(\cos(\theta), \sin(\theta)) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- `void Matrix_translate2D(Matrix *m, double tx, double ty)`—Premultiply the matrix by a 2D translation matrix parameterized by \( t_x \) and \( t_y \).

\[ m = T(t_x, t_y) \cdot m \quad \text{where} \quad T(t_x, t_y) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- `void Matrix_shear2D(Matrix *m, double shx, double shy)`—Premultiply the matrix by a 2D shear matrix parameterized by \( sh_x \) and \( sh_y \).

\[ m = Sh(sh_x, sh_y) \cdot m \quad \text{where} \quad Sh(sh_x, sh_y) = \begin{bmatrix} 1 & sh_x & 0 & 0 \\ sh_y & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

3D Additions to Matrix (C Version)

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

\[ m = T(t_x, t_y, t_z) \cdot m \quad \text{where} \quad T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- `void Matrix_scale2(Matrix *m, double sx, double sy, double sz)`—Premultiply the matrix by a scale matrix parameterized by \( s_x, s_y, s_z \).

\[ m = S(s_x, s_y, s_z) \cdot m \quad \text{where} \quad S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- `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.

\[ m = R_x(\cos(\theta), \sin(\theta)) \cdot m \quad \text{where} \quad R_x(\cos(\theta), \sin(\theta)) = \begin{bmatrix} \cos(\theta) & 0 & 0 & 0 \\ 0 & -\sin(\theta) & 0 & 0 \\ 0 & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
\[ m = R_X(\cos(\theta), \sin(\theta)) \cdot m \text{ where } R((\cos(\theta), \sin(\theta))) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- 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.

\[ m = R_Y(\cos(\theta), \sin(\theta)) \cdot m \text{ where } R((\cos(\theta), \sin(\theta))) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- 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.

\[ m = R_{\vec{u}, \vec{v}, \vec{w}}(\vec{u}, \vec{v}, \vec{w}) \cdot m \text{ where } R_{\vec{u}, \vec{v}, \vec{w}}(\vec{u}, \vec{v}, \vec{w}) = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- void Matrix\_shearZ(Matrix *m, double shx, double shy) – Premultiply the matrix by a shear Z matrix parameterized by \( shx \) and \( shy \).

\[ m = S_{z}(sh_x, sh_y) \cdot m \text{ where } S_{z}(d) = \begin{bmatrix} 1 & 0 & 0 & shx \\ 0 & 1 & 0 & shy \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

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

\[ m = \text{Persp}(d) \cdot m \text{ where } \text{Persp}(d) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \]

**C++ Method Specification**

The following methods should be defined for the Matrix class.

- void clear() – Set the matrix to all zeros.
- void identity() – Set the matrix to the identity matrix.
- double get(int r, int c) – Return the element of the matrix at row r, column c.
- void set(int r, int c, double v) – Set the element of the matrix at row r, column c to v.
- void set(Matrix *src) – Copy in the src matrix data.
- void transpose() – Transpose the matrix m in place.
• void xform(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.
• void xform(Polygon *p) – Transform the points in the Polygon p by the matrix m.
• void xform(Polyline *p) – Transform the points in the Polyline p by the matrix m.
• void xform(Line *line) – Transform the points in line by the matrix m.
• void scale2D(double sx, double sy) – Premultiply the matrix by a scale matrix parameterized by $s_x$ and $s_y$.
• void rotateZ(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.
• void translate2D(double tx, double ty) – Premultiply the matrix by a 2D translation matrix parameterized by $t_x$ and $t_y$.
• void shear2D(double shx, double shy) – Premultiply the matrix by a 2D shear matrix parameterized by $sh_x$ and $sh_y$.

3D Additions to Matrix (C++ Version)

• void translate(double tx, double ty, double tz) – Premultiply the matrix by a 3D translation matrix parameterized by $t_x$, $t_y$, and $t_z$.
• void scale(double sx, double sy, double sz) – Premultiply the matrix by a scale matrix parameterized by $s_x$, $s_y$, and $s_z$.
• void rotateX(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.
• void rotateY(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.
• void rotateXYZ(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.
• void shearZ(double shx, double shy) – Premultiply the matrix by a shear Z matrix parameterized by $sh_x$ and $sh_y$.
• void perspective(double d) – Premultiply the matrix by a perspective matrix parameterized by $d$.

The following functions should also be defined on matrices, but not as part of the Matrix class.

• Matrix &operator*(Matrix &left, Matrix &right) – Multiplies left times right and returns a reference to a the result.
• void operator<<(std::ostream &os, const Matrix &right) – Print out the matrix in a nice 4x4 arrangement with a space below.
• Matrix &operator=(Matrix &left, Matrix &right) – Does a proper copy of right to left. Returns a reference to left matrix.
Hierarchical Modeling Functions (C Version)

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,
   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;

// end option 1

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

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

// end option 2

typedef struct {
   Element *head; // pointer to the head of the linked list
   Element *tail; // keep around a pointer to the last object
} Module;
C 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.
3D Additions to Module (C version)

- `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.

C++ Functions

In C++ the one option is to create a base type, Element, and let each of the individual types inherit. Then the Module class only needs to worry about working with Element pointers (except for Module pointers). Elements are not specified here, however, so the exact design of a Module class is flexible so long as it provides the functionality below.

The following set of functions create, add items to, manipulate, draw, and destroy Modules. Note that the add function is overloaded in C++.

- `void Module()`—Destructor needs to free all of the memory associated with a module.
- `void clear()`—clear the module’s list, freeing memory as appropriate.
- `void 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 insert(Module *md, Element *e)`—Generic insert of an element into the module at the tail of the list.
- `void add(Module *md)`—Adds a pointer to the Module sub to the tail of the module’s list.
- `void add(Point *p)`—Adds $p$ to the tail of the module’s list.
- `void add(Line *p)`—Adds $p$ to the tail of the module’s list.
- `void add(Polyline *p)`—Adds $p$ to the tail of the module’s list.
- `void add(Polygon *p)`—Adds $p$ to the tail of the module’s list.
- `void add(Pixel p)`—Adds a pixel to the tail of the module’s list.
- `void identity()`—Object that sets the current transform to the identity, placed at the tail of the module’s list.
• **void translate(double tx, double ty)** – Matrix operand to add a translation matrix to the tail of the module’s list.

• **void scale(md, double sx, double sy)** – Matrix operand to add a scale matrix to the tail of the module’s list.

• **void rotateZ(double cth, double sth)** – Matrix operand to add a rotation about the Z axis to the tail of the module’s list.

• **void shear2D(double shx, double shy)** – Matrix operand to add a 2D shear matrix to the tail of the module’s list.

**3D Additions to Module (C++ version)**

• **void translate(double tx, double ty, double tz)** – Matrix operand to add a 3D translation to the Module.

• **void scale(double sx, double sy, double sz)** – Matrix operand to add a 3D scale to the Module.

• **void rotateX(double cth, double sth)** – Matrix operand to add a rotation about the X-axis to the Module.

• **void rotateY(double cth, double sth)** – Matrix operand to add a rotation about the Y-axis to the Module.

• **void rotateXYZ(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 cube(void)** – Adds a unit cube, axis-aligned and centered on zero to the Module.
3D Viewing

Define a PerspectiveView structure/class as below.

```c
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.

**C++ Version:** void Matrix::set3DView(PerspectiveView &view) – Implement the 3D perspective pipeline. Inside the function, begin by initializing matrix to the identity. Do not modify any of the values in the PerspectiveView structure inside the function.