```cpp
/*
* File: main.cpp
* Author: Jacob Christensen
* Course: CS3210
* Date: 03/29/19
*
* D = load in STL file and draw
* Arrow Keys = rotation/orbiting
* Scroll wheel = scaling
* Numbers = change focus distance option
*/

#include "shape.h"
#include "x11context.h"
#include <unistd.h>
#include <iostream>
#include "mydrawing.h"
int main(void)
{
    GraphicsContext* gc = new X11Context(800,600,GraphicsContext::BLACK);
    gc->setColor(GraphicsContext::GREEN);
    // make a drawing
    MyDrawing md;
    // start event loop - this function will return when X is clicked
    // on window
    gc->runLoop(&md);
    delete gc;
    return 0;
}
```

```
/*
* File: viewcontext.h
* Author: Jacob Christensen
* Course: CS3210
* Date: 04/25/19
*/

#ifndef _VIEW_CONTEXT_
#define _VIEW_CONTEXT_

#include "matrix.h"
#include "gcontext.h"


class ViewContext {
private:
    matrix transformation;
    matrix inverse;
    matrix* viewPoint;
    matrix* refPoint;
    matrix* normal;
    matrix* L;
    matrix* M;
    matrix* V;
    matrix* unit_L;
    matrix* unit_M;
    matrix* unit_N;
    double ro_angle, trans_vert, trans_hor, scale_factor, focus_distance;

public:
    // Constructor
    ViewContext();
    // Destructor
    ~ViewContext();
    // Convert coordinates from model to device
    matrix modelToDevice(matrix model_coor, GraphicsContext* gc) const;
    // Convert coordinates from device to model
    matrix deviceToModel(matrix device_coor, GraphicsContext* gc) const;
    // Convert coordinates from model to view
    matrix modelToView(matrix model_coor) const;
    // Convert coordinates from view to projection
    matrix viewToProjection(matrix view_coor) const;


    // Method to add a scaling transform to the compound transformation
    void scale(const double factor);

    // Method to add a rotation to the compound transformation
    void rotate_H(const double angle);

    // Method to add a rotation to the compound transformation
    void rotate_V(const double angle);

    // Method to add a translation to the compound transformation
    void translate(const double horizontal, const double vertical);

    // Method to set the focus distance for point projection
    void setFocus(const double distance);

    // Applies the compound transformation to the provided matrix
    matrix transform(matrix mat) const;
    // Applies the inverse compound transformation to the provided matrix
    matrix invert(matrix mat) const;

    // Resets the transformation and the viewport
    void reset(GraphicsContext* gc);
};

#endif
```

```
/*
* File: viewcontext.cpp
* Author: Jacob Christensen
* Course: CS3210
* Date: 04/28/19
*/

#include "viewcontext.h"
#include "matrix.h"
#include "gcontext.h"
#include "image.h"
#include "line.h"
#include "triangle.h"

#include <cmath>
#include <iostream>

using namespace std;

ViewContext::ViewContext():transformation(matrix::identity(4)),inverse(matrix::identity(4)),
viewPoint(new matrix(4,1)), refPoint(new matrix(4,1)),normal(new matrix(3,1)),L(new matrix(3,1)),
M(new matrix(3,1)), V(new matrix(3,1)), unit_L(new matrix(3,1)),unit_M(new matrix(3,1)),unit_N(new matrix(3,1)) {
    ro_angle = trans_vert = trans_hor = scale_factor = 0.0;
    focus_distance = 100.0;
    (*V)[1][0] = 1;
    (*viewPoint)[0][0] = 2;
    (*viewPoint)[1][0] = 2;
    (*viewPoint)[2][0] = 5;
    (*viewPoint)[3][0] = 1.0;
    (*refPoint)[0][0] =  0;
    (*refPoint)[1][0] = 0;
    (*refPoint)[2][0] = 0;
    (*refPoint)[3][0] = 0;
    (*normal)[0][0] = (*viewPoint)[0][0] - (*refPoint)[0][0];
    (*normal)[1][0] = (*viewPoint)[1][0] - (*refPoint)[1][0];
    (*normal)[2][0] = (*viewPoint)[2][0] - (*refPoint)[2][0];
    (*L) = matrix::crossProduct(*V, *normal);
    (*M) = matrix::crossProduct(*normal, *L);
    (*unit_L) = (*L) * (1.0/sqrt((pow((*L)[0][0],2) + pow((*L)[1][0],2) + pow((*L)[2][0],2))));
    (*unit_M) = (*M) * (1.0/sqrt((pow((*M)[0][0],2) + pow((*M)[1][0],2) + pow((*M)[2][0],2))));
    (*unit_N) = (*normal) * (1.0/sqrt((pow((*normal)[0][0],2) + pow((*normal)[1][0],2) + pow((*normal)[2][0],2))));
}

ViewContext::~ViewContext() {
    delete viewPoint;
    delete refPoint;
    delete normal;
    delete M;
    delete L;
    delete V;
    delete unit_L;
    delete unit_M;
    delete unit_N;
}

matrix ViewContext::modelToDevice(matrix model_coor, GraphicsContext* gc) const {
    matrix ret_mat(4,1);
    matrix reflect = matrix::identity(4);
    matrix trans = matrix::identity(4);
    reflect[1][1] = -1;
    trans[1][3] = (gc->getWindowHeight());
    // Reflect the model coordinates over the x-axis
    ret_mat = reflect * model_coor;
    // Shift the coordinates down by the height of the viewport
    ret_mat = trans * ret_mat;
```

```
66      return ret_mat;
67  }
68
69  matrix ViewContext::deviceToModel(matrix device_coor, GraphicsContext* gc) const {
70      matrix ret_mat(4,1);
71      matrix reflect = matrix::identity(4);
72      matrix trans = matrix::identity(4);
73      reflect[1][1] = -1;
74      trans[1][3] = -(gc->getWindowHeight());
75      // Shift the coordinates up by the height of the viewport
76      ret_mat = trans * device_coor;
77      // Reflect the coordinates over the x-axis
78      ret_mat = reflect * ret_mat;
79      return ret_mat;
80  }
81
82  matrix ViewContext::modelToView(matrix model_coor) const {
83      matrix retVal(4,1);
84      matrix mTv = matrix::identity(4);
85      mTv[0][0] = (*unit_L)[0][0];
86      mTv[0][1] = (*unit_L)[1][0];
87      mTv[0][2] = (*unit_L)[2][0];
88      mTv[0][3] = -(((*unit_L)[0][0] * (*viewPoint)[0][0]) + ((*unit_L)[1][0] * (*viewPoint)
    [1][0]) + ((*unit_L)[2][0] * (*viewPoint)[2][0]));
89      mTv[1][0] = (*unit_M)[0][0];
90      mTv[1][1] = (*unit_M)[1][0];
91      mTv[1][2] = (*unit_M)[2][0];
92      mTv[1][3] = -(((*unit_M)[0][0] * (*viewPoint)[0][0]) + ((*unit_M)[1][0] * (*viewPoint)
    [1][0]) + ((*unit_M)[2][0] * (*viewPoint)[2][0]));
93      mTv[2][0] = (*unit_N)[0][0];
94      mTv[2][1] = (*unit_N)[1][0];
95      mTv[2][2] = (*unit_N)[2][0];
96      mTv[2][3] = -(((*unit_N)[0][0] * (*viewPoint)[0][0]) + ((*unit_N)[1][0] * (*viewPoint)
    [1][0]) + ((*unit_N)[2][0] * (*viewPoint)[2][0]));
97      retVal = mTv *  model_coor;
98      return retVal;
99  }
100
101 matrix ViewContext::viewToProjection(matrix view_coor) const {
102     matrix retVal(4,1);
103     matrix vTp = matrix::identity(4);
104     vTp[2][2] = 0;
105     vTp[3][2] = -1.0/focus_distance;
106     retVal = vTp * view_coor;
107     return retVal;
108 }
109
110
111 void ViewContext::scale(double factor) {
112     scale_factor *= factor;
113     // Create the transform
114     matrix scale_mat = matrix::identity(4);
115     matrix i_scale_mat = matrix::identity(4);
116     scale_mat[0][0] = scale_mat[1][1] = scale_mat[2][2] = factor;
117     i_scale_mat[0][0] = i_scale_mat[1][1] = i_scale_mat[2][2] = 1/factor;
118     // Add the transform to the compound
119     this->translate(-400,-300);
120     transformation = scale_mat * transformation;
121     this->translate(400,300);
122     this->translate(0,600);
123     inverse = i_scale_mat * inverse;
124     this->translate(0,-600);
125     return;
126 }
127
128 void ViewContext::rotate_H(double angle) {
129     ro_angle += angle;
130     // Create the rotation transform
131     matrix ro = matrix::identity(4);
132     matrix i_ro = matrix::identity(4);
133     ro[0][0] = cos(angle * 3 / 180);
```

```
134     ro[0][2] = sin(angle * 3 / 180);
135     ro[2][0] = -(sin(angle * 3 / 180));
136     ro[2][2] = cos(angle * 3 / 180);
137     i_ro[0][0] = cos(-angle * 3 / 180);
138     i_ro[0][1] = -(sin(-angle * 3 / 180));
139     i_ro[1][0] = sin(-angle * 3 / 180);
140     i_ro[1][1] = cos(-angle * 3 / 180);
141     // Add the rotation to the viewpoint
142     (*viewPoint) = ro * (*viewPoint);
143     (*normal)[0][0] = (*viewPoint)[0][0] - (*refPoint)[0][0];
144     (*normal)[1][0] = (*viewPoint)[1][0] - (*refPoint)[1][0];
145     (*normal)[2][0] = (*viewPoint)[2][0] - (*refPoint)[2][0];
146     (*L) = matrix::crossProduct(*V, *normal);
147     (*M) = matrix::crossProduct(*normal, *L);
148     (*unit_L) = (*L) * (1.0/sqrt((pow((*L)[0][0],2) + pow((*L)[1][0],2) + pow((*L)[2][0],2
    ))));
149     (*unit_M) = (*M) * (1.0/sqrt((pow((*M)[0][0],2) + pow((*M)[1][0],2) + pow((*M)[2][0],2
    ))));
150     (*unit_N) = (*normal) * (1.0/sqrt((pow((*normal)[0][0],2) + pow((*normal)[1][0],2) + p
    ow((*normal)[2][0],2))));
151     return;
152 }
153
154 void ViewContext::rotate_V(double angle) {
155     ro_angle += angle;
156     // Create the rotation transform
157     matrix ro_z = matrix::identity(4);
158     matrix ro_y = matrix::identity(4);
159     matrix ro_vector = matrix(3,1);
160     matrix z_vector = matrix(3,1);
161     z_vector[2][0] = 1.0;
162     ro_vector[0][0] = (*unit_L)[0][0];
163     ro_vector[2][0] = (*unit_L)[2][0];
164     double xy_angle = matrix::vectorAngle(z_vector,ro_vector);
165     // Rotate around y-axis
166     this->rotate_H(-xy_angle * (180 / 3));
167     ro_z[0][0] = cos(angle * 3 / 180);
168     ro_z[1][0] = sin(angle * 3 / 180);
169     ro_z[0][1] = -(sin(angle * 3 / 180));
170     ro_z[1][1] = cos(angle * 3 / 180);
171     // Add the rotation to the viewpoint
172     (*viewPoint) = ro_z * (*viewPoint);
173     (*normal)[0][0] = (*viewPoint)[0][0] - (*refPoint)[0][0];
174     (*normal)[1][0] = (*viewPoint)[1][0] - (*refPoint)[1][0];
175     (*normal)[2][0] = (*viewPoint)[2][0] - (*refPoint)[2][0];
176     (*L) = matrix::crossProduct(*V, *normal);
177     (*M) = matrix::crossProduct(*normal, *L);
178     (*unit_L) = (*L) * (1.0/sqrt((pow((*L)[0][0],2) + pow((*L)[1][0],2) + pow((*L)[2][0],2
    ))));
179     (*unit_M) = (*M) * (1.0/sqrt((pow((*M)[0][0],2) + pow((*M)[1][0],2) + pow((*M)[2][0],2
    ))));
180     (*unit_N) = (*normal) * (1.0/sqrt((pow((*normal)[0][0],2) + pow((*normal)[1][0],2) + p
    ow((*normal)[2][0],2))));
181     // Rotate back around the y-axis
182     this->rotate_H(xy_angle * (180 / 3));
183     return;
184 }
185
186 void ViewContext::translate(double horizontal, double vertical) {
187     // Create the translation transform
188     matrix trans = matrix::identity(4);
189     matrix i_trans = matrix::identity(4);
190     trans[0][3] += horizontal;
191     trans[1][3] += vertical;
192     i_trans[0][3] -= horizontal;
193     i_trans[1][3] -= vertical;
194     // Apply the transform to the compound
195     transformation = trans * transformation;
196     inverse = i_trans * inverse;
197     return;
198 }
```

```
199
200 void ViewContext::setFocus(const double distance) {
201     focus_distance = distance;
202 }
203
204 // Apply the transformation to the given matrix
205 matrix ViewContext::transform(matrix mat) const {
206     return transformation * mat;
207 }
208
209 // Apply the inverse transformation to the given matrix
210 matrix ViewContext::invert(matrix mat) const {
211     return inverse * mat;
212 }
213
214 void ViewContext::reset(GraphicsContext* gc) {
215     gc->clear();
216     transformation = matrix::identity(4);
217     inverse = matrix::identity(4);
218     focus_distance = 100.0;
219     (*V)[1][0] = 1;
220     (*viewPoint)[0][0] = 2;
221     (*viewPoint)[1][0] = 2;
222     (*viewPoint)[2][0] = 5;
223     (*viewPoint)[3][0] = 1.0;
224     (*refPoint)[0][0] =  0;
225     (*refPoint)[1][0] = 0;
226     (*refPoint)[2][0] = 0;
227     (*refPoint)[3][0] = 0;
228     (*normal)[0][0] = (*viewPoint)[0][0] - (*refPoint)[0][0];
229     (*normal)[1][0] = (*viewPoint)[1][0] - (*refPoint)[1][0];
230     (*normal)[2][0] = (*viewPoint)[2][0] - (*refPoint)[2][0];
231     (*L) = matrix::crossProduct(*V, *normal);
232     (*M) = matrix::crossProduct(*normal, *L);
233     (*unit_L) = (*L) * (1.0/sqrt((pow((*L)[0][0],2) + pow((*L)[1][0],2) + pow((*L)[2][0],2
    ))));
234     (*unit_M) = (*M) * (1.0/sqrt((pow((*M)[0][0],2) + pow((*M)[1][0],2) + pow((*M)[2][0],2
    ))));
235     (*unit_N) = (*normal) * (1.0/sqrt((pow((*normal)[0][0],2) + pow((*normal)[1][0],2) + p
    ow((*normal)[2][0],2))));
236 }
```

```
1  /*
2  * File: mydrawing.h
3  * Author: Jacob Christensen
4  * Course: CS3210
5  * Date: 04/20/19
6  */
7
8  #ifndef _MY_DRAWING_H
9  #define _MY_DRAWING_H
10
11 #include "drawbase.h"
12 #include "image.h"
13 #include "matrix.h"
14 #include "viewcontext.h"
15
16 // forward reference
17 class GraphicsContext;
18
19 class MyDrawing: public DrawingBase
20 {
21 public:
22     // Enum that defines possible drawing modes
23     enum shapeMode {POINT, LINE, TRIANGLE};
24
25     // Empty Constructor
26     MyDrawing();
27     // Destructor
28     ~MyDrawing();
29     virtual void paint(GraphicsContext* gc);
30     // Event handler for clicking the mouse button down
31     virtual void mouseButtonDown(GraphicsContext* gc, unsigned int button,
32                                  int x, int y);
33     // Event handler for releasing the mouse button
34     virtual void mouseButtonUp(GraphicsContext* gc, unsigned int button,
35                                int x, int y);
36     // Event handler for moving the mouse
37     virtual void mouseMove(GraphicsContext* gc, int x, int y);
38     // Event handler pressing a keyboard key down
39     virtual void keyDown(GraphicsContext* gc, unsigned int keycode);
40     // Event handler for releasing a keyboard key
41     virtual void keyUp(GraphicsContext* gc, unsigned int keycode);
42
43 private:
44     //Keycodes for switching modes, saving, and loading
45     const unsigned int P = 112;
46     const unsigned int L = 108;
47     const unsigned int T = 116;
48     const unsigned int S = 115;
49     const unsigned int D = 100;
50     const unsigned int R = 114;
51     const unsigned int LEFT_ARR = 65361;
52     const unsigned int UP_ARR = 65362;
53     const unsigned int RIGHT_ARR = 65363;
54     const unsigned int DOWN_ARR = 65364;
55     const unsigned int CCW_RO = 44;
56     const unsigned int CW_RO = 46;
57
58
59     // Possible coordinates
60     int x0, y0, x1, y1, x2, y2;
61     // Current color
62     unsigned int color;
63     // Image holding all of the shapes being drawn
64     image img;
65     // Enum to keep track of the drawing mode the program is in
66     shapeMode mode;
67     // ViewContext for displaying and transforming image
68     ViewContext vc;
69 };
70
71
```

```
72 #endif
```

```
/*
* File: mydrawing.cpp
* Author: Jacob Christensen
* Course: CS3210
* Date: 04/20/19
*/

#include "mydrawing.h"
#include "gcontext.h"
#include "viewcontext.h"
#include "matrix.h"
#include "triangle.h"
#include "line.h"

#include <fstream>
#include <iostream>

using namespace std;

// Constructor
MyDrawing::MyDrawing()
{
    color = GraphicsContext::GREEN;
    x0 = x1 = y0 = y1 = x2 = y2 = 0;
    return;
}

MyDrawing::~MyDrawing() {}

void MyDrawing::paint(GraphicsContext* gc)
{
    int middlex = gc->getWindowWidth()/2;
    int middley = gc->getWindowHeight()/2;

    gc->setColor(GraphicsContext::MAGENTA);

    for (int yi=middley-50;yi<=middley+50;yi++)
    {
        gc->drawLine(middlex-50,yi,middlex+50,yi);
    }

    gc->setColor(GraphicsContext::GREEN);
    // redraw the line if requested
    gc->drawLine(x0,y0,x1,y1);
    return;
}

void MyDrawing::mouseButtonDown(GraphicsContext* gc, unsigned int button, int x, int y)
{
    if(button == 4) {
        // Scale the viewport by a factor of 2
        vc.scale(2);
        gc->clear();
        img.draw(gc,vc);
    }
    else if(button == 5) {
        // Scale the viewport by a factor of 1/2
        vc.scale(0.5);
        gc->clear();
        img.draw(gc,vc);
    }

    return;
}

void MyDrawing::mouseButtonUp(GraphicsContext* gc, unsigned int button, int x, int y)
{
    return;
}

void MyDrawing::mouseMove(GraphicsContext* gc, int x, int y)
```

```
72  {
73
74      return;
75  }
76
77  // Do nothing on key down
78  void MyDrawing::keyDown(GraphicsContext* gc, unsigned int keycode) {}
79
80  // Respond to keypress on keyUp
81  void MyDrawing::keyUp(GraphicsContext* gc, unsigned int keycode) {
82      if(keycode == this->S) {
83          ofstream output_file;
84          output_file.open("image_out.txt", ofstream::trunc);
85          output_file << img;
86          output_file.close();
87      }
88      // Load the image described in image_in.txt
89      else if(keycode == this->D) {
90          vc.reset(gc);
91          img.erase(img);
92          gc->clear();
93          vc.translate(400,-300);
94          ifstream input_file("image_in.txt");
95          input_file >> img;
96          input_file.close();
97          img.draw(gc, vc);
98      }
99      // Reset the viewport to its original coordinates
100     else if(keycode == this->R) {
101         vc.reset(gc);
102         vc.translate(400,-300);
103         img.draw(gc,vc);
104     }
105     // Rotate the viewport counter-clockwise 10 degrees
106     else if(keycode == this->RIGHT_ARR) {
107         vc.rotate_H(10);
108         gc->clear();
109         img.draw(gc,vc);
110     }
111     // Rotate the viewport clockwise 10 degrees
112     else if(keycode == this->LEFT_ARR) {
113         vc.rotate_H(-10);
114         gc->clear();
115         img.draw(gc,vc);
116     }
117     else if(keycode == this->UP_ARR) {
118         vc.rotate_V(-10);
119         gc->clear();
120         img.draw(gc,vc);
121     }
122     else if(keycode == this->DOWN_ARR) {
123         vc.rotate_V(10);
124         gc->clear();
125         img.draw(gc,vc);
126     }
127     // Change the current focus distance
128     else if(keycode > 48 && keycode < 58){
129         switch(keycode) {
130             case 49 :
131                 vc.setFocus(15.0);
132                 gc->clear();
133                 img.draw(gc,vc);
134                 break;
135             case 50 :
136                 vc.setFocus(20.0);
137                 gc->clear();
138                 img.draw(gc,vc);
139                 break;
140             case 51 :
141                 vc.setFocus(30.0);
142                 gc->clear();
```

```
143                     img.draw(gc,vc);
144                     break;
145                 case 52 :
146                     vc.setFocus(40.0);
147                     gc->clear();
148                     img.draw(gc,vc);
149                     break;
150                 case 53 :
151                     vc.setFocus(50.0);
152                     gc->clear();
153                     img.draw(gc,vc);
154                     break;
155                 case 54 :
156                     vc.setFocus(60.0);
157                     gc->clear();
158                     img.draw(gc,vc);
159                     break;
160                 case 55 :
161                     vc.setFocus(70.0);
162                     gc->clear();
163                     img.draw(gc,vc);
164                     break;
165                 case 56 :
166                     vc.setFocus(80.0);
167                     gc->clear();
168                     img.draw(gc,vc);
169                     break;
170                 case 57 :
171                     vc.setFocus(90.0);
172                     gc->clear();
173                     img.draw(gc,vc);
174                     break;
175             }
176         }
177         return;
178     }
```

```
1  /*
2  * File: image.h
3  * Author: Jacob Christensen
4  * Course: CS3210
5  * Date: 04/07/19
6  */
7
8  #ifndef _IMAGE_H_
9  #define _IMAGE_H_
10
11 #include "x11context.h"
12 #include "shape.h"
13 #include "viewcontext.h"
14 #include <vector>
15 #include <string>
16
17 class image {
18     //Container for the shapes to be held
19     std::vector<shape *> shapes;
20
21 public:
22     // Constructor - create an empty image object
23     image();
24
25     // Copy constructor
26     image(const image& from);
27
28     // Destructor - free allocated memory
29     ~image();
30
31     // Copy Constructor - performs deep copy from rhs to this
32     image& operator=(const image& rhs);
33
34     void add(shape* s);
35
36     // Draw - interates throught the vector, drawing all of
37     //        the shapes
38     void draw(GraphicsContext* gc, ViewContext& vc) const;
39
40     // Erase - clears all the shapes from the shapes vector
41     void erase(image& img);
42
43     // Out - this method will send the properties of the image
44     //       to the output stream
45     virtual std::ostream& out(std::ostream& os) const;
46
47     // In - takes in the image properties from a text file
48     virtual std::istream& in(std::istream& in);
49
50     std::vector<shape *>& getShapes();
51 };
52
53 // Global Stream Operators
54 std::ostream& operator<<(std::ostream& os, const image& rhs);
55
56 std::istream& operator>>(std::istream& is, image& rhs);
57
58 #endif
```

```cpp
/*
* File: image.cpp
* Author: Jacob Christensen
* Course: CS3210
* Date: 04/07/19
*/

#include "x11context.h"
#include "image.h"
#include "matrix.h"
#include "shape.h"
#include "line.h"
#include "triangle.h"
#include <vector>
#include <string>
#include <iostream>

// Constructor - create an image object
image::image() {}


// Copy constructor
image::image(const image& from) {
    for(auto i = from.shapes.begin(); i != from.shapes.end(); i++) {
        shapes.push_back((*i)->clone());
    }
}

// Destructor - free allocated memory
image::~image() {
    for(auto i = shapes.begin(); i != shapes.end(); i++) {
        delete *i;
    }
}

// Assignment operator- performs deep copy from rhs to this
image& image::operator=(const image& rhs) {
    if(this != &rhs) {
        // Delete current image
        for(auto i = shapes.begin(); i != shapes.end(); i++) {
            delete *i;
        }
        // Copy shapes
        for(auto i = rhs.shapes.begin(); i != rhs.shapes.end(); i++) {
            shapes.push_back((*i)->clone());
        }
    }

    return *this;
}

// Add - add a new shape to the image
void image::add(shape* s) {
    shapes.push_back(s);
}

// Draw - draws the image. Pure virtual, functionality is
//        defined in subclasses
void image::draw(GraphicsContext* gc, ViewContext& vc) const {
    if(shapes.size() > 0) {
        for(auto i = shapes.begin(); i != shapes.end(); i++) {
            (*i)->draw(gc, vc);
        }
    }
}

// Erase - Clear the shapes from an image
void image::erase(image& img) {
    if(shapes.size() > 0) {
        for(auto i = img.shapes.begin(); i != img.shapes.end(); i++) {
            delete *i;
```

```
72          }
73          shapes.clear();
74      }
75  }
76
77  // Out - this method will send the properties of the image
78  //       to the output stream
79  std::ostream& image::out(std::ostream& os) const {
80      if(shapes.size() > 0) {
81          for(auto i = shapes.begin(); i != shapes.end(); i++) {
82              os << *(*i) << std::endl;
83          }
84      }
85      return os;
86  }
87
88  // In - takes in the image properties from a text file
89  std::istream& image::in(std::istream& in) {
90      std::string identifier, in_line;
91      bool inFacet = false;
92      matrix p0(4,1);
93      matrix p1(4,1);
94      matrix p2(4,1);
95      p0[3][0] = 1.0;
96      p1[3][0] = 1.0;
97      p2[3][0] = 1.0;
98      int next_c = in.peek();
99      while(next_c != EOF) {
100         in >> identifier;
101         getline(in,in_line);
102         // Identifying the start of a new facet in the file
103         if(identifier == "facet" && !inFacet) {
104             inFacet = true;
105         }
106
107         if(identifier == "outer" && inFacet) {
108             triangle* t = new triangle(0x00FF00, p0, p1, p2);
109             in >> *t;
110             shapes.push_back(t);
111         }
112
113         // Identifying the end of a facet
114         if(identifier == "endfacet" && inFacet) {
115             inFacet = false;
116         }
117         next_c = in.peek();
118     }
119     return in;
120 }
121
122 std::vector<shape *>& image::getShapes() {
123     return shapes;
124 }
125
126 // Stream insertion operator - not a member function
127 std::ostream& operator<<(std::ostream& os, const image& rhs) {
128     rhs.out(os);
129     return os;
130 }
131
132 // Stream Extraction operator - not a member function
133 std::istream& operator>>(std::istream& is, image& rhs) {
134     rhs.in(is);
135     return is;
136 }
```

```
/*
* File: shape.h
* Author: Jacob Christensen
* Course: CS3210
* Date: 03/27/19
*/

#ifndef _SHAPE_H_
#define _SHAPE_H_

#include "matrix.h"
#include "x11context.h"
#include "viewcontext.h"
#include <iostream>
#include <string>

class shape {
    // Int value that describes the color
    unsigned int color;
    // Matrix to hold the origin point
    matrix p0;

protected:
    // Copy Constructor - performs deep copy from rhs to this
    //                    only to be accessed by children of shape
    shape& operator=(const shape& rhs);

public:
    // Constructor - create a shape object with user specified
    //               RGB value and center point at p0 (x,y,z,1.0)
    shape(unsigned int color, matrix p0);

    // Copy constructor
    shape(const shape& from);

    // Destructor - free allocated memory
    virtual ~shape();

    // Accessors for the private properties
    unsigned int get_color() const;
    matrix get_p0() const;

    // Clone - makes copies of shapes (Pure Virtual)
    virtual shape* clone() const = 0;

    // Draw - draws the shape. Pure virtual, functionality is
    //        defined in subclasses
    //TODO
    virtual void draw(GraphicsContext* gc, ViewContext& vc) const = 0;

    // Out - this method will send the properties of the shape
    //       to the output stream
    virtual std::ostream& out(std::ostream& os) const;

    // In - takes in the shape properties from a text file
    virtual std::istream& in(std::istream& in);
};

// Global Stream Operators
std::ostream& operator<<(std::ostream& os, const shape& rhs);

std::istream& operator>>(std::istream& is, shape& rhs);

#endif
```

```cpp
/*
* File: shape.cpp
* Author: Jacob Christensen
* Course: CS3210
* Date: 03/28/19
*/

#include "shape.h"

#include <string>
#include <iomanip>
#include <iostream>

// Constructor for initializing Shape object
shape::shape(unsigned int color, matrix p0):color(color),p0(p0){}

// Copy constructor
shape::shape(const shape& from):color(from.color),p0(from.p0){}

// Destructor - free allocated memory
shape::~shape() {}

// Color accessor
unsigned int shape::get_color() const {
    return color;
}

// p0 accessor
matrix shape::get_p0() const {
    return p0;
}

// Copy Constructor - performs deep copy from rhs to this
shape& shape::operator=(const shape& rhs) {
    if(this != &rhs) {
        p0 = rhs.p0;
        color = rhs.color;
    }
    return *this;
}

// Out - this method will send the properties of the shape
//       to the output stream
std::ostream& shape::out(std::ostream& os) const{
    os << "C\n" << std::to_string(get_color()) << "\n";
    os << "P0\n" << (get_p0())[0][0] << std::endl;
    os << (get_p0())[1][0] << std::endl;
    os << (get_p0())[2][0] << std::endl;
    return os;
}

// In - takes in the shape properties from a text file
std::istream& shape::in(std::istream& in) {
    std::string identifier, point_0_str;
    double x, y, z;
    matrix* new_p0 = new matrix(4,1);
    in >> identifier;
    getline(in,point_0_str);
    if(identifier == "vertex") {
        std::stringstream(point_0_str) >> x >> y >> z;
        (*new_p0)[0][0] = x;
        (*new_p0)[1][0] = y;
        (*new_p0)[2][0] = z;
        // Set the last matrix entry to 1.0
        (*new_p0)[3][0] = 1;
        p0 = *new_p0;
    }
    delete new_p0;
    return in;
}

```

```
72  // Stream Insertion Operator - Calls out function
73  std::ostream& operator<<(std::ostream& os, const shape& rhs) {
74      rhs.out(os);
75      return os;
76  }
```

```
1  /*
2  * File: triangle.h
3  * Author: Jacob Christensen
4  * Course: CS3210
5  * Date: 03/29/19
6  */
7
8  #ifndef _TRIANGLE_H_
9  #define _TRIANGLE_H_
10
11
12 #include "shape.h"
13 #include "x11context.h"
14 #include "viewcontext.h"
15 #include "matrix.h"
16
17 class triangle : public shape {
18     matrix p1;
19     matrix p2;
20
21 public:
22
23     //  Constructor - calls shape constructor
24     triangle(unsigned int color, matrix p0, matrix p1, matrix p2);
25     // Copy Constructor - creates a new triangle that is a
26     //                    deep copy of from
27     triangle(const triangle& from);
28
29     // Accessor for P1
30     matrix get_p1() const;
31     // Accessor for P2
32     matrix get_p2() const;
33
34     // Destructor
35     ~triangle();
36
37     // Clone - returns a new triangle that is a deep copy of this
38     shape* clone() const;
39
40     // Assignment Operator - performs a deep copy from rhs to this
41     triangle& operator=(const triangle& rhs);
42
43     //Draw - draws line segments between each vertex
44     void draw(GraphicsContext* gc, ViewContext& vc) const;
45
46     // Out - this method will send the properties of the shape
47     //       to the output stream
48     std::ostream& out(std::ostream& os) const;
49
50     // In - takes in the shape properties from a text file
51     std::istream& in(std::istream& in);
52 };
53
54 // Global Stream Operators
55 std::ostream& operator<<(std::ostream& os, const triangle& rhs);
56
57 std::istream& operator>>(std::istream& is, triangle& rhs);
58
59
60 #endif
```

```cpp
/*
* File: triangle.cpp
* Author: Jacob Christensen
* Course: CS3210
* Date: 03/29/19
*/
#include "triangle.h"
#include "x11context.h"
#include "viewcontext.h"
#include <sstream>
#include <iostream>

// Constructor for initializing triangle objects
triangle::triangle(unsigned int color, matrix p0, matrix p1, matrix p2):shape(color,p0),p1(p1),p2(p2){}

// Copy Constructor
triangle::triangle(const triangle& from):triangle(from.get_color(), from.get_p0(), from.get_p1(), from.get_p2()){}

// Destructor - nothing new needs to happen
triangle::~triangle(){}

// Accessor for P1
matrix triangle::get_p1() const {
    return p1;
}

// Accesssor for P2
matrix triangle::get_p2() const {
    return p2;
}

// Assignment operator - performs deep copy from rhs to this
triangle& triangle::operator=(const triangle& rhs) {
    if(this != &rhs) {
        shape::operator=(rhs);
        this->p1 = rhs.get_p1();
        this->p2 = rhs.get_p2();
    }
    return *this;
}

// Clone - returns a deep copy of the triangle
shape* triangle::clone() const {
    return new triangle(*this);
}

//Draw - draw lines between each vertex
void triangle::draw(GraphicsContext* gc, ViewContext& vc) const{
    matrix disp_p0 = this->get_p0();
    disp_p0 = vc.modelToView(disp_p0);
    disp_p0 = vc.viewToProjection(disp_p0);
    disp_p0 = vc.modelToDevice(disp_p0, gc);
    disp_p0 = disp_p0 * (1.0/disp_p0[3][0]);
    disp_p0 = vc.transform(disp_p0);
    matrix disp_p1 = this->get_p1();
    disp_p1 = vc.modelToView(disp_p1);
    disp_p1 = vc.viewToProjection(disp_p1);
    disp_p1 = vc.modelToDevice(disp_p1, gc);
    disp_p1 = disp_p1 * (1.0/disp_p1[3][0]);
    disp_p1 = vc.transform(disp_p1);
    matrix disp_p2 = this->get_p2();
    disp_p2 = vc.modelToView(disp_p2);
    disp_p2 = vc.viewToProjection(disp_p2);
    disp_p2 = vc.modelToDevice(disp_p2, gc);
    disp_p2 = disp_p2 * (1.0/disp_p2[3][0]);
    disp_p2 = vc.transform(disp_p2);
    //std::cout << disp_p0 << std::endl;
    //std::cout << disp_p1 << std::endl;
    //std::cout << disp_p2 << std::endl;
```

```
70      gc->setColor(this->get_color());
71      gc->drawLine(disp_p0[0][0],disp_p0[1][0],disp_p1[0][0],disp_p1[1][0]);
72      gc->drawLine(disp_p1[0][0],disp_p1[1][0],disp_p2[0][0],disp_p2[1][0]);
73      gc->drawLine(disp_p2[0][0],disp_p2[1][0],disp_p0[0][0],disp_p0[1][0]);
74  }
75
76  //Output - output properties to ostream
77  std::ostream& triangle::out(std::ostream& os) const{
78      os << "T" << std::endl;
79      this->shape::out(os);
80      os << "P1\n" << (get_p1())[0][0] << std::endl;
81      os << (get_p1())[1][0] << std::endl;
82      os << (get_p1())[2][0] << std::endl;
83      os << "P2\n" << (get_p2())[0][0] << std::endl;
84      os << (get_p2())[1][0] << std::endl;
85      os << (get_p2())[2][0] << std::endl;
86      os << "END" << std::endl;
87      return os;
88  }
89
90  //Input -take in properties from istream
91  std::istream& triangle::in(std::istream& in) {
92      unsigned int vector_count = 0;
93      double x, y, z;
94      std::string identifier, point_str;
95      matrix* new_p1 = new matrix(4,1);
96      matrix* new_p2 = new matrix(4,1);
97      this->shape::in(in);
98      vector_count++;
99      in >> identifier;
100     getline(in,point_str);
101     while(identifier != "endloop" && vector_count < 3) {
102         if(identifier == "vertex" && vector_count == 1) {
103             std::stringstream(point_str) >> x >> y >> z;
104             (*new_p1)[0][0] = x;
105             (*new_p1)[1][0] = y;
106             (*new_p1)[2][0] = z;
107             // Set the last matrix entry to 1.0
108             (*new_p1)[3][0] = 1;
109             p1 = *new_p1;
110             vector_count++;
111         }
112         else if(identifier == "vertex" && vector_count == 2) {
113             std::stringstream(point_str) >> x >> y >> z;
114             (*new_p2)[0][0] = x;
115             (*new_p2)[1][0] = y;
116             (*new_p2)[2][0] = z;
117             // Set the last matrix entry to 1.0
118             (*new_p2)[3][0] = 1;
119             p2 = *new_p2;
120             vector_count++;
121         }
122         in >> identifier;
123         getline(in,point_str);
124     }
125     delete new_p1;
126     delete new_p2;
127     return in;
128 }
129
130 // Global Stream Operators
131 std::ostream& operator<<(std::ostream& os, const triangle& rhs) {
132     rhs.out(os);
133     return os;
134 }
135
136 std::istream& operator>>(std::istream& is, triangle& rhs) {
137     rhs.in(is);
138     return is;
139 }
```

```
/*
* File: line.h
* Author: Jacob Christensen
* Course: CS3210
* Date: 03/29/19
*/

#ifndef _LINE_H_
#define _LINE_H_


#include "shape.h"
#include "x11context.h"
#include "viewcontext.h"
#include "matrix.h"

class line : public shape {
    matrix p1;

public:

    //  Constructor - calls shape constructor
    line(unsigned int color, matrix p0, matrix p1);

    // Copy Constructor - Creates new line that is a deep copy
    //                    of from
    line(const line& from);

    // Accessor for P1
    matrix get_p1() const;

    // Destructor
    ~line();

    // Clone - returns a new point that is a deep copy of this
    virtual shape* clone() const;

    // Assignment Operator - performs a deep copy from rhs to this
    line& operator=(const line& rhs);

    //Draw - Draws a line segment between P0 and P1
    void draw(GraphicsContext* gc, ViewContext& vc) const;

    // Out - this method will send the properties of the shape
    //       to the output stream
    std::ostream& out(std::ostream& os) const;
    // In - takes in the shape properties from a text file
    std::istream& in(std::istream& in);
};

// Global Stream Operators
std::ostream& operator<<(std::ostream& os, const line& rhs);

std::istream& operator>>(std::istream& is, line& rhs);

#endif
```

```cpp
/*
* File: line.cpp
* Author: Jacob Christensen
* Course: CS3210
* Date: 03/29/19
*/
#include "line.h"
#include "x11context.h"
#include "viewcontext.h"
#include <sstream>

line::line(unsigned int color, matrix p0, matrix p1):shape(color,p0),p1(p1){}

line::line(const line& from):line(from.get_color(), from.get_p0(), from.get_p1()){}

line::~line(){}

matrix line::get_p1() const {
    return p1;
}

// Clone
shape* line::clone() const {
    return new line(*this);
}

line& line::operator=(const line& rhs) {
    if(this != &rhs) {
        shape::operator=(rhs);
        this->p1 = rhs.get_p1();
    }
    return *this;
}

//Draw
void line::draw(GraphicsContext* gc, ViewContext& vc) const{
    matrix disp_p0 = this->get_p0();
    disp_p0 = vc.modelToView(disp_p0);
    disp_p0 = vc.viewToProjection(disp_p0);
    disp_p0 = disp_p0 * (1.0/disp_p0[3][0]);
    disp_p0 = vc.modelToDevice(disp_p0, gc);
    disp_p0 = vc.transform(disp_p0);

    matrix disp_p1 = this->get_p1();
    disp_p1 = vc.modelToView(disp_p1);
    disp_p1 = vc.viewToProjection(disp_p1);
    disp_p1[3][0] = disp_p1[3][0] * (1.0/disp_p1[3][0]);
    disp_p1 = vc.modelToDevice(disp_p1, gc);
    disp_p1 = vc.transform(disp_p1);
    gc->setColor(this->get_color());
    gc->drawLine(disp_p0[0][0],disp_p0[1][0],disp_p1[0][0],disp_p1[1][0]);
}

//Output
std::ostream& line::out(std::ostream& os) const{
    os << "L" << std::endl;
    this->shape::out(os);
    os << "P1 \n" << (get_p1())[0][0] << std::endl;
    os << (get_p1())[1][0] << std::endl;
    os << (get_p1())[2][0] << std::endl;
    os << "END" << std::endl;
    return os;
}

std::istream& line::in(std::istream& in) {
    this->shape::in(in);
    std::string identifier, point_1_str;
    matrix* new_p1 = new matrix(4,1);
    getline(in,point_1_str);
    std::stringstream(point_1_str) >> *new_p1[0][0];
    getline(in,point_1_str);
```

```
72      std::stringstream(point_1_str) >> *new_p1[0][1];
73      getline(in,point_1_str);
74      std::stringstream(point_1_str) >> *new_p1[0][2];
75      getline(in,point_1_str);
76      p1 = *new_p1;
77      delete new_p1;
78      return in;
79  }
80
81  std::ostream& operator<<(std::ostream& os, const line& rhs) {
82      rhs.out(os);
83      return os;
84  }
85
86  std::istream& operator>>(std::istream& is, line& rhs) {
87      rhs.in(is);
88      return is;
89  }
```

```
/**
 * matrix.h - declaration of matrix class.  This class will be
 * capable of representing a Matrix as a first-class type.
 *
 * Do not change any public methods in this file unless otherwise
 * instructed.
 *
 * For CS321.  (c) 2015 Dr. Darrin Rothe
 */


// compile guard
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream> // for std::ostream
#include <stdexcept>    // for std::runtime_error
#include <string>   // used in exception

// a helper class to bundle a message with any thrown exceptions.
// To use, simply 'throw matrixException("A descriptive message about
// the problem")'.  This will throw the exception object by value.
// Recommendation is to catch by reference (to prevent slicing).
class matrixException:public std::runtime_error
{
    public:
        matrixException(std::string message):
              std::runtime_error((std::string("Matrix Exception: ") +
                       message).c_str()) {}
};


class matrix
{
    public:
        // No default (no argument) constructor.  It doesn't really make
        // sense to have one as we cannot rely on a size.  This may trip
        // us up later, but it will lead to a better implementation.
        // matrix();

        // Constructor - create Matrix and clear cells.  If rows or
        // cols is < 1, throw a matrixException.  Note, we will not use
        // "exception specifications" as multiple sources report that
        // the mechanism is not properly supported by most compilers.
        //
        // throw (matrixException)
        //
        matrix(unsigned int rows, unsigned int cols);

        // Copy constructor - make a new Matrix just like rhs
        matrix(const matrix& from);

        // Destructor.  Free allocated memory
        ~matrix();

        // Assignment operator - make this just like rhs.  Must function
            // correctly even if rhs is a different size than this.
        matrix& operator=(const matrix& rhs);

        // "Named" constructor(s).  This is not a language mechanism, rather
        // a common programming idiom.  The underlying issue is that with
        // overloaded operators, you can lose sight of what various
        // combinations of arguments means.  For our design, constructor
        // arguments set the size of the matrix.  If we want to generate
        // "special" matrices, we are better off with static methods that
        // will generate these and then we can give the methods meaningful
        // names.

        // Named Constructor - produce a square identity matrix of the
        // requested size.  Since we do not know how the object produced will
        // be used, we pretty much have to return by value.  A size of 0
```

```
72          // would not make sense and should throw an exception.
73          //
74          // throw (matrixException)
75          //
76          static matrix identity(unsigned int size);
77
78
79
80          // Matrix addition - lhs and rhs must be same size otherwise
81          // an exception shall be thrown
82          //
83          // throw (matrixException)
84          //
85          matrix operator+(const matrix& rhs) const;
86
87          // Matrix multiplication - lhs and rhs must be compatible
88          // otherwise an exception shall be thrown
89          //
90          // throw (matrixException)
91          //
92          matrix operator*(const matrix& rhs) const;
93
94          // Cross product - perform the cross product between two
95          // 3x1 matrices, or "vectors"
96          //
97          // throw (matrixException)
98          //
99          static matrix crossProduct(matrix a, matrix b);
100
101         // Vector angle - finds the angle between 2 3x1 vectors
102         //
103         // throw (matrixException)
104         //
105         static double vectorAngle(const matrix a, const matrix b);
106
107         // Scalar multiplication.  Note, this function will support
108         // someMatrixObject * 5.0, but not 5.0 * someMatrixObject.
109         matrix operator*(const double scale) const;
110
111         // Transpose of a Matrix - should always work, hence no exception
112         matrix operator~() const;
113
114         // Clear Matrix to all members 0.0
115         void clear();
116
117         // Access Operators - throw an exception if index out of range
118         //
119         // Note how these operators are to work.  Consider a Matrix
120         // object being addressed with two sets of brackets - m1[1][2],
121         // for example.  The compiler will execute this: (m1[1])[2].
122         // The first set of brackets will call this function, and this
123         // function should return a pointer to the first element of the
124         // requested row.  The second set of brackets is applied to the
125         // double*, which results in it being treated as an array, thus
126         // the requested column is indexed.  The const version is
127         // necessary if you would like to use the operator within other
128         // const methods.  Both of these operators are extremely
129         // dangerous as prototyped.  The nature of the danger and
130         // a fix are left up to you to discover and fix.  A proper
131         // fix will require a change to these function signatures
132         // and the use of an internal "helper class."
133         //
134         // throw (matrixException)
135         //
136         double* operator[](unsigned int row);
137
138         // const version of above - throws an exception if indices are out of
139         // range
140         //
141         // throw (matrixException)
142         //
```

```
143         double* operator[](unsigned int row) const;
144
145         // I/O - for convenience - this is intended to be called by the global
146         // << operator declared below.
147         std::ostream& out(std::ostream& os) const;
148
149     private:
150         // The data - note, per discussion on arrays, you can store these data
151         // as a 1-D dynamic array, thus the double* below.  Alternatively, and
152         // perhaps preferred, you could store the data as an array of arrays
153         // which would require the_Matrix to be changed to a double**.
154         double** the_matrix;
155         unsigned int rows;
156         unsigned int cols;
157
158         /** routines **/
159
160         // add any "helper" routine here, such as routines to support
161         // matrix inversion
162
163 };
164
165 /** Some Related Global Functions **/
166
167 // Overloaded global << with std::ostream as lhs, Matrix as rhs.  This method
168 // should generate output compatible with an ostream which is commonly used
169 // with console (cout) and files.  Something like:
170 // [[ r0c0, r0c1, r0c2 ]
171 //  [ r1c0, r1c1, r1c2 ]
172 //  [ r0c0, r0c1, r0c2 ]]
173 // would be appropriate.
174 //
175 // Since this is a global function, it does not have access to the private
176 // data of a Matrix object.  So, it will need to use the public interface of
177 // Matrix to do its job.  The method Matrix::out was added to Matrix
178 // specifically for this purpose.  The other option would have been to make
179 // it a "friend"
180
181 std::ostream& operator<<(std::ostream& os, const matrix& rhs);
182
183 // We would normally have a corresponding >> operator, but
184 // will defer that exercise that until a later assignment.
185
186
187 // Scalar multiplication with a global function.  Note, this function will
188 // support 5.0 * someMatrixObject, but not someMatrixObject * 5.0
189 matrix operator*(const double scale, const matrix& rhs);
190
191
192 #endif
```

```cpp
/*
* File: matrix.cpp
* Author: Jacob Christensen
* Course: CS3210
* Date: 03/14/19
*/

#include "matrix.h"
#include <string>
#include <cmath>
#include <iostream>

using namespace std;

// Parameterized constructor
matrix::matrix(unsigned int rows, unsigned int cols):rows(rows),cols(cols)
{
    if (rows < 1 || cols < 1)
    {
        throw matrixException("p-constructor bad arguments");
    }
    else
    {
        the_matrix = new double *[rows];
        for(unsigned int i=0;i<rows;i++)
        {
            the_matrix[i] = new double[cols];
        }
        //Clearing all existing values in the memory
        for(unsigned int r=0;r<rows;r++)
        {
            for(unsigned int c=0;c<cols;c++)
            {
                the_matrix[r][c] = 0.0;
            }
        }
    }
}

// Copy constructor
matrix::matrix(const matrix& from):rows(from.rows),cols(from.cols)
{
    the_matrix = new double *[rows];
    for(unsigned int x=0;x<rows;x++)
    {
        the_matrix[x] = new double[cols];
        for(unsigned int y=0;y<cols;y++) {
            the_matrix[x][y] = from.the_matrix[x][y];
        }
    }
}

// Destructor
matrix::~matrix()
{
    for(unsigned int i=0;i<rows;i++) {
        delete [] the_matrix[i];
    }
    delete [] the_matrix;
    rows = 0;
    cols = 0;
}

matrix matrix::crossProduct(matrix a, matrix b) {
    if(a.rows == 3 && a.cols == 1 && b.rows == 3 && b.cols == 1) {
        matrix retVal(3,1);
        retVal[0][0] = (a[1][0] * b[2][0]) - (a[2][0] * b[1][0]);
        retVal[1][0] = (a[2][0] * b[0][0]) - (a[0][0] * b[2][0]);
        retVal[2][0] = (a[0][0] * b[1][0]) - (a[1][0] * b[0][0]);
        return retVal;
    }
```

```
72      else {
73          throw matrixException(
74              "Cross product must be between two 3x1 matrices");
75      }
76
77  }
78
79  double matrix::vectorAngle(const matrix a, const matrix b) {
80      double mag_a = sqrt((pow(a[0][0],2) + pow(a[1][0],2) + pow(a[2][0],2)));
81      double mag_b = sqrt((pow(b[0][0],2) + pow(b[1][0],2) + pow(b[2][0],2)));
82      double dot_prod = (a[0][0] * b[0][0]) + (a[1][0] * b[1][0]) + (a[2][0] * b[2][0]);
83      return acos(dot_prod/(mag_a * mag_b));
84  }
85
86
87  // Assignment operator
88  matrix& matrix::operator=(const matrix& rhs)
89  {
90      // check for self-assignment
91      if(this != &rhs)
92      {
93          for(unsigned int i=0;i<rows;i++) {
94              delete [] the_matrix[i];
95          }
96          delete [] the_matrix;
97
98          rows = rhs.rows;
99          cols = rhs.cols;
100         the_matrix = new double *[rows];
101         for(unsigned int x=0;x<rows;x++)
102         {
103             the_matrix[x] = new double[cols];
104             for(unsigned int y=0;y<cols;y++)
105             {
106                 the_matrix[x][y] = rhs.the_matrix[x][y];
107             }
108         }
109     }
110     return *this;
111 }
112
113 // Named constructor (static)
114 matrix matrix::identity(unsigned int size)
115 {
116     if(size < 1) {
117         throw matrixException(
118             "Cannot have an identity matrix with size < 0");
119     }
120     // use p-constructor
121     matrix ident(size, size);
122     for(unsigned int x=0;x<size;x++)
123     {
124         for(unsigned int y=0;y<size;y++)
125         {
126             if(x == y)
127             {
128                 ident.the_matrix[x][y] = 1;
129             }
130             else
131             {
132                 ident.the_matrix[x][y] = 0;
133             }
134         }
135     }
136     return ident;
137 }
138
139
140 // Binary operations
141 matrix matrix::operator+(const matrix& rhs) const
142 {
```

```
143     matrix retVal(rows, cols);
144     if(rows != rhs.rows || cols != rhs.cols)
145     {
146         throw matrixException("Matrices cannot be added: Different sizes");
147     }
148     else {
149         for(unsigned int r=0;r<rows;r++) {
150             for(unsigned int c=0;c<cols;c++) {
151                 retVal[r][c] = the_matrix[r][c] + rhs[r][c];
152             }
153         }
154     }
155     return retVal;
156 }
157
158
159 matrix matrix::operator*(const matrix& rhs) const
160 {
161     matrix retVal(rows, rhs.cols);
162     if(cols != rhs.rows) {
163         throw matrixException(
164             "Matrices cannot be multiplied: Columns and rows are not compatible");
165     }
166     else {
167         for(unsigned int i=0;i<rows;i++) {
168             for(unsigned int j=0;j<rhs.cols;j++) {
169                 retVal[i][j] = 0;
170                 for(unsigned int k=0;k<rhs.rows;k++) {
171                     retVal[i][j] += the_matrix[i][k] * rhs[k][j];
172                 }
173             }
174         }
175     }
176     return retVal;
177 }
178
179 matrix matrix::operator*(const double scale) const
180 {
181     matrix retVal(*this);
182     for(unsigned int r=0;r<rows;r++) {
183         for(unsigned int c=0;c<cols;c++) {
184             retVal[r][c] = the_matrix[r][c] * scale;
185         }
186     }
187     return retVal;
188 }
189
190
191 // Unary operations
192 matrix matrix::operator~() const
193 {
194     // stub
195     matrix retVal(cols, rows);
196     for(unsigned int r=0;r<rows;r++)
197     {
198         for(unsigned int c=0;c<rows;c++)
199         {
200             retVal[r][c] = the_matrix[c][r];
201         }
202     }
203     return retVal;
204 }
205
206
207 void matrix::clear()
208 {
209     for(unsigned int r=0;r<rows;r++)
210     {
211         for(unsigned int c=0;c<rows;c++)
212         {
213             the_matrix[r][c] = 0.0;
```

```
214             }
215         }
216         return;
217     }
218
219     double* matrix::operator[](unsigned int row)
220     {
221         if(row > rows || row < 0) {
222             throw matrixException(
223                 "Cannot access index");
224         }
225         else {
226             return the_matrix[row];
227         }
228
229     }
230
231     double* matrix::operator[](unsigned int row) const
232     {
233         if(row > rows || row < 0) {
234             throw matrixException(
235                 "Cannot access index");
236         }
237         else {
238             return the_matrix[row];
239         }
240     }
241
242
243     std::ostream& matrix::out(std::ostream& os) const
244     {
245         string matrixString = "[";
246         for(unsigned int r=0;r<rows;r++)
247         {
248             matrixString += "[";
249             for(unsigned int c=0;c<cols;c++)
250             {
251                 matrixString += to_string(the_matrix[r][c]);
252                 if(c<cols-1) {
253                     matrixString += ",";
254                 }
255             }
256             matrixString += "]";
257             if(r<rows-1)
258             {
259                 matrixString += ",\n";
260             }
261         }
262         matrixString += "]";
263         os << matrixString;
264         return os;
265     }
266
267
268
269     // Global insertion and operator
270     std::ostream& operator<<(std::ostream& os, const matrix& rhs)
271     {
272         return rhs.out(os);
273     }
274
275     // Global scalar multiplication
276     matrix operator*(const double scale, const matrix& rhs)
277     {
278         matrix retVal(rhs);
279         retVal = rhs * scale;
280         return retVal;
281     }
```

```
#ifndef GCONTEXT_H
#define GCONTEXT_H

/**
 * This class is intended to be the abstract base class
 * for a graphical context for various platforms.  Any
 * concrete subclass will need to implement the pure virtual
 * methods to support setting pixels, getting pixel color,
 * setting the drawing mode, and running an event loop to
 * capture mouse and keyboard events directed to the graphics
 * context (or window).  Specific expectations for the various
 * methods are documented below.
 *
 * Note, naive implementations of a line scan-conversion and a
 * circle scan-conversion are provided here which rely on the
 * concrete setPixel of the implemnting subclass.  These
 * implementation are expected to be overridden for
 * better performance.
 *
 * */


// forward reference - needed because runLoop needs a target for events
class DrawingBase;


class GraphicsContext
{
    public:
        /*************************************************************
         * Some constants and enums
         *************************************************************/
        // This enumerated type is an argument to setMode and allows
        // us to support two different drawing modes.  MODE_NORMAL is
        // also call copy-mode and the affect pixel(s) are set to the
        // color requested.  XOR mode will XOR the new color with the
        // existing color so that the change is reversible.
        enum drawMode {MODE_NORMAL, MODE_XOR};

        // Some colors - for fun
        static const unsigned int BLACK = 0x000000;
        static const unsigned int BLUE = 0x0000FF;
        static const unsigned int GREEN = 0x00FF00;
        static const unsigned int RED = 0xFF0000;
        static const unsigned int CYAN = 0x00FFFF;
        static const unsigned int MAGENTA = 0xFF00FF;
        static const unsigned int YELLOW = 0xFFFF00;
        static const unsigned int GRAY = 0x808080;
        static const unsigned int WHITE = 0xFFFFFF;


        /*************************************************************
         * Construction / Destruction
         *************************************************************/
        // Implementations of this class should include a constructor
        // that creates the drawing canvas (window), sets a background
        // color (which may be configurable), sets a default drawing
        // color (which may be configurable), and start with normal
        // (copy) drawing mode.

        // need a virtual destructor to ensure subclasses will have
        // their destructors called properly.  Must be virtual.
        virtual ~GraphicsContext();

        /*************************************************************
         * Drawing operations
         *************************************************************/

        // Allows the drawing mode to be changed between normal (copy)
        // and xor.  The implementing context should default to normal.
        virtual void setMode(drawMode newMode) = 0;
```

```
72
73          // Set the current color.  Implementations should default to white.
74          // color is 24-bit RGB value
75          virtual void setColor(unsigned int color) = 0;
76
77          // Set pixel to the current color
78          virtual void setPixel(int x, int y) = 0;
79
80          // Get 24-bit RGB pixel color at specified location
81          // unsigned int will likely be 32-bit on 32-bit systems, and
82          // possible 64-bit on some 64-bit systems.  In either case,
83          // it is large enough to hold a 16-bit color.
84          virtual unsigned int getPixel(int x, int y) = 0;
85
86          // This should reset entire context to the current background
87          virtual void clear()=0;
88
89          // These are the naive implementations that use setPixel,
90          // but are overridable should a context have a better-
91          // performing version available.
92
93           /* This is a naive implementation that uses floating-point math
94           * and "setPixel" which will need to be provided by the concrete
95           * implementation.
96           *
97           * Parameters:
98           *  x0, y0 - origin of line
99           *  x1, y1 - end of line
100          *
101          * Returns: void
102          */
103         virtual void drawLine(int x0, int y0, int x1, int y1);
104
105         /* This is a naive implementation that uses floating-point math
106          * and "setPixel" which will need to be provided by the concrete
107          * implementation.
108          *
109          * Parameters:
110          *  x0, y0 - origin/center of circle
111          *  radius - radius of circle
112          *
113          * Returns: void
114          */
115         virtual void drawCircle(int x0, int y0, unsigned int radius);
116
117
118         /*******************************************************
119          * Event loop operations
120          *******************************************************/
121
122         // Run Event loop.  This routine will receive events from
123         // the implementation and pass them along to the drawing.  It
124         // will return when the window is closed or other implementation-
125         // specific sequence.
126         virtual void runLoop(DrawingBase* drawing) = 0;
127
128         // This method will end the current loop if one is running
129         // a default version is supplied
130         virtual void endLoop();
131
132
133         /*******************************************************
134          * Utility operations
135          *******************************************************/
136
137         // returns the width of the window
138         virtual int getWindowWidth() = 0;
139
140         // returns the height of the window
141         virtual int getWindowHeight() = 0;
142
```

```
143     protected:
144         // this flag is used to control whether the event loop
145         // continues to run.
146         bool run;
147 };
148
149 #endif
```

```
1  /* This is an abstract base class representing a generic graphics
2   * context.  Most implementation specifics will need to be provided by
3   * a concrete implementation.  See header file for specifics. */
4
5  #define _USE_MATH_DEFINES   // for M_PI
6  #include <cmath>    // for trig functions
7  #include "gcontext.h"
8
9  /*
10  * Destructor - does nothing
11  */
12 GraphicsContext::~GraphicsContext()
13 {
14     // nothing to do
15     // here to insure subclasses handle destruction properly
16 }
17
18
19 /* This is a naive implementation that uses floating-point math
20  * and "setPixel" which will need to be provided by the concrete
21  * implementation.
22  *
23  * Parameters:
24  *  x0, y0 - origin of line
25  *  x1, y1 - end of line
26  *
27  * Returns: void
28  */
29 void GraphicsContext::drawLine(int x0, int y0, int x1, int y1)
30 {
31
32     // find slope
33     int dx = x1-x0;
34     int dy = y1-y0;
35
36     // make sure we actually have a line
37     if (dx != 0 || dy !=0)
38     {
39         int xi, yi, xi1, yi1, di, di1;
40         // slope < 1?
41         if (std::abs(dx)>std::abs(dy))
42         {
43             xi = x0;
44             yi = y0;
45             di = 2*std::abs(dy) - std::abs(dx);
46             setPixel(xi,yi);
47
48             while(xi != x1) {
49                 // Always increment/decrement the x-value
50                 if(xi < x1) {
51                     xi1 = xi + 1;
52                 }
53                 else{
54                     xi1 = xi - 1;
55                 }
56
57                 // Calculate the new decision matrix
58                 if(di < 0 || di == 0) {
59                     // If di is negative or zero, keep the same y-value
60                     yi1 = yi;
61                     di1 = di + 2 * std::abs(dy);
62                 }
63                 else {
64                     // If di is non-negative, increment the y-value
65                     if(dy > 0) {
66                         yi1 = yi + 1;
67                     }
68                     // Otherwise, decrement the y-value
69                     else {
70                         yi1 = yi - 1;
71                     }
```

```
                    di1 = di + 2 * std::abs(dy) - 2 * std::abs(dx);
                }
                setPixel(xi, yi);
                xi = xi1;
                yi = yi1;
                di = di1;
            }

        } // end of if |slope| < 1
        else
        {
            xi = x0;
            yi = y0;
            di = 2*std::abs(dx) - std::abs(dy);
            setPixel(xi,yi);

            while(yi != y1) {
                // Always increment/decrement the y-value
                if(yi < y1) {
                    yi1 = yi + 1;
                }
                else{
                    yi1 = yi - 1;
                }

                // Calculate the new decision matrix
                if(di <= 0) {
                    // If di is negative, keep the same x-value
                    xi1 = xi;
                    di1 = di + 2 * std::abs(dx);
                }
                else {
                    // If di is non-negative, increment the x-value
                    if(dx > 0) {
                        xi1 = xi + 1;
                    }
                    // Otherwise, decrement the x-value
                    else {
                        xi1 = xi - 1;
                    }
                    di1 = di + 2 * std::abs(dx) - 2 * std::abs(dy);
                }
                setPixel(xi, yi);
                xi = xi1;
                yi = yi1;
                di = di1;
            }
        } // end of else |slope| >= 1
    } // end of if it is a real line (dx!=0 || dy !=0)
    return;
}


/* This is a naive implementation that uses floating-point math
 * and "setPixel" which will need to be provided by the concrete
 * implementation.
 *
 * Parameters:
 *  x0, y0 - origin/center of circle
 *  radius - radius of circle
 *
 * Returns: void
 */
void GraphicsContext::drawCircle(int x0, int y0, unsigned int radius)
{
    // Declaring variables to be useed
    int xi, xi1, yi, yi1, pi, pi1;
    // Check if there is a valid radius
    if(radius <= 0) {
        // If radius is <= 0, then only set one pixel
```

```
143             setPixel(x0,y0);
144         }
145         else {
146             // Initial values for the variables
147             xi = x0;
148             yi = radius + y0;
149             pi = 1 - (radius);
150             //Sweep the x-values
151             while((xi-x0)<(yi-y0 + 1)) {
152                 xi1 = xi + 1;
153                 // If decision matrix is negative
154                 if(pi<0) {
155                     // Y doesn't change
156                     yi1 = yi;
157                     pi1 = pi + (2 * (xi1 - x0)) + 1;
158                 }
159                 // If decision matrix is positive
160                 else {
161                     // Y increments
162                     yi1 = yi - 1;
163                     pi1 = pi + (2 * ((xi1 - x0) - (yi1 - y0))) + 1;
164                 }
165                 // Set the pixels
166                 setPixel(xi,yi);
167                 setPixel(x0-(xi-x0),yi);
168                 setPixel(xi,y0-(yi-y0));
169                 setPixel(x0-(xi-x0),y0-(yi-y0));
170                 // Move to the next values
171                 xi = xi1;
172                 yi = yi1;
173                 pi = pi1;
174             }
175             yi = y0;
176             xi = x0 +  radius;
177             pi = 1 - (radius);
178             //Sweep the y-values
179             while((xi-x0 + 1)>(yi-y0)) {
180                 yi1 = yi +1;
181                 // If decision matrix is negative
182                 if(pi<0) {
183                     // X doesn't change
184                     xi1 = xi;
185                     pi1 = pi + (2 * (yi1 - y0)) + 1;
186                 }
187                 // If decision matrix is positive
188                 else {
189                     // X decrements
190                     xi1 = xi - 1;
191                     pi1 = pi + (2 * ((yi1 - y0) - (xi1 - x0))) + 1;
192                 }
193                 // Set the pixels
194                 setPixel(xi,yi);
195                 setPixel(x0-(xi-x0),yi);
196                 setPixel(xi,y0-(yi-y0));
197                 setPixel(x0-(xi-x0),y0-(yi-y0));
198                 // Move to the next values
199                 xi = xi1;
200                 yi = yi1;
201                 pi = pi1;
202             }
203         }
204         return;
205     }
206
207     void GraphicsContext::endLoop()
208     {
209         run = false;
210     }
211
212
```

```
#ifndef X11_CONTEXT
#define X11_CONTEXT
/**
 * This class is a sample implementation of the GraphicsContext class
 * for the X11 / XWindows system.
 * */

#include <X11/Xlib.h>   // Every Xlib program must include this
#include "gcontext.h"      // base class

class X11Context : public GraphicsContext
{
    public:
        // Default Constructor
        X11Context(unsigned int sizex = 400, unsigned int sizey = 400,
                           unsigned int bg_color = GraphicsContext::BLACK);

        // Destructor
        virtual ~X11Context();

        // Drawing Operations
        void setMode(drawMode newMode);
        void setColor(unsigned int color);
        void setPixel(int x, int y);
        unsigned int getPixel(int x, int y);
        void clear();

        /*
         * These are not currently overridden, but could be as XLib
         * has much more efficient implementations available.
         */
        //void drawLine(int x1, int y1, int x2, int y2);
        //void drawCircle(int x, int y, int radius);


        // Event looop functions
        void runLoop(DrawingBase* drawing);

        // we will use endLoop provided by base class

        // Utility functions
        int getWindowWidth();
        int getWindowHeight();


    private:
        // X11 stuff - specific to this context
        Display* display;
        Window window;
        GC graphics_context;

};

#endif
```

```
1  /* Provides a simple drawing context for X11/XWindows
2   * You must have the X11 dev libraries installed.  If missing,
3   * 'sudo apt-get install libx11-dev' should help.
4   */
5
6  #include <X11/Xlib.h> // Every Xlib program must include this
7  #include <X11/Xutil.h> // needed for XGetPixel
8  #include <X11/XKBlib.h> // needed for keyboard setup
9  #include "x11context.h"
10 #include "drawbase.h"
11 #include <iostream>
12
13 /**
14  * The only constructor provided.  Allows size of window and background
15  * color be specified.
16  * */
17 X11Context::X11Context(unsigned int sizex,unsigned int sizey,
18                        unsigned int bg_color)
19 {
20     // Open the display
21     display = XOpenDisplay(NULL);
22
23     // Holding a key in gives repeated key_press commands but only
24     // one key_release
25     int supported;
26
27     XkbSetDetectableAutoRepeat(display,true,&supported);
28
29     // Create a window - we will assume the color map is in RGB mode.
30     window = XCreateSimpleWindow(display, DefaultRootWindow(display), 0, 0,
31                 sizex, sizey, 0, 0 , bg_color);
32
33     // Sign up for MapNotify events
34     XSelectInput(display, window, StructureNotifyMask);
35
36     // Put the window on the screen
37     XMapWindow(display, window);
38
39     // Create a "Graphics Context"
40     graphics_context = XCreateGC(display, window, 0, NULL);
41
42     // Default color to white
43     XSetForeground(display, graphics_context, GraphicsContext::WHITE);
44
45     // Wait for MapNotify event
46     for(;;)
47     {
48         XEvent e;
49         XNextEvent(display, &e);
50         if (e.type == MapNotify)
51         break;
52     }
53
54     // We also want exposure, mouse, and keyboard events
55     XSelectInput(display, window, ExposureMask|
56                                   ButtonPressMask|
57                                   ButtonReleaseMask|
58                                   KeyPressMask|
59                                   KeyReleaseMask|
60                                   PointerMotionMask);
61
62     // We need this to get the WM_DELETE_WINDOW message from the
63     // window manager in case user click the X icon
64     Atom atomKill = XInternAtom(display, "WM_DELETE_WINDOW", False);
65     XSetWMProtocols(display, window, &atomKill, 1);
66
67     return;
68 }
69
70 // Destructor  - shut down window and connection to server
71 X11Context::~X11Context()
```

```
72  {
73      XFreeGC(display, graphics_context);
74      XDestroyWindow(display,window);
75      XCloseDisplay(display);
76  }
77
78  // Set the drawing mode - argument is enumerated
79  void X11Context::setMode(drawMode newMode)
80  {
81      if (newMode == GraphicsContext::MODE_NORMAL)
82      {
83          XSetFunction(display,graphics_context,GXcopy);
84      }
85      else
86      {
87          XSetFunction(display,graphics_context,GXxor);
88      }
89  }
90
91  // Set drawing color - assume colormap is 24 bit RGB
92  void X11Context::setColor(unsigned int color)
93  {
94      // Go ahead and set color here - better performance than setting
95      // on every setPixel
96      XSetForeground(display, graphics_context, color);
97  }
98
99  // Set a pixel in the current color
100 void X11Context::setPixel(int x, int y)
101 {
102     XDrawPoint(display, window, graphics_context, x, y);
103     XFlush(display);
104 }
105
106 unsigned int X11Context::getPixel(int x, int y)
107 {
108     XImage *image;
109     image = XGetImage (display, window, x, y, 1, 1, AllPlanes, XYPixmap);
110     XColor color;
111     color.pixel = XGetPixel (image, 0, 0);
112     XFree (image);
113     XQueryColor (display, DefaultColormap(display, DefaultScreen (display)),
114                  &color);
115     // I now have RGB values, but, they are 16 bits each, I only want 8-bits
116     // each since I want a 24-bit RGB color value
117     unsigned int pixcolor = color.red & 0xff00;
118     pixcolor |= (color.green >> 8);
119     pixcolor <<= 8;
120     pixcolor |= (color.blue >> 8);
121     return pixcolor;
122 }
123
124 void X11Context::clear()
125 {
126     XClearWindow(display, window);
127     XFlush(display);
128 }
129
130
131
132 // Run event loop
133 void X11Context::runLoop(DrawingBase* drawing)
134 {
135     run = true;
136
137     while(run)
138     {
139         XEvent e;
140         XNextEvent(display, &e);
141
142         // Exposure event - lets not worry about region
```

```
143             if (e.type == Expose)
144                 drawing->paint(this);
145 
146             // Key Down
147             else if (e.type == KeyPress)
148                 drawing->keyDown(this,XLookupKeysym((XKeyEvent*)&e,
149                         (((e.xkey.state&0x01)&&!(e.xkey.state&0x02))||
150                         (!(e.xkey.state&0x01)&&(e.xkey.state&0x02)))?1:0));
151 
152             // Key Up
153             else if (e.type == KeyRelease){
154                 drawing->keyUp(this,XLookupKeysym((XKeyEvent*)&e,
155                         (((e.xkey.state&0x01)&&!(e.xkey.state&0x02))||
156                         (!(e.xkey.state&0x01)&&(e.xkey.state&0x02)))?1:0));
157                     }
158 
159             // Mouse Button Down
160             else if (e.type == ButtonPress)
161                 drawing->mouseButtonDown(this,
162                 e.xbutton.button,
163                 e.xbutton.x,
164                 e.xbutton.y);
165 
166             // Mouse Button Up
167             else if (e.type == ButtonRelease)
168                 drawing->mouseButtonUp(this,
169                 e.xbutton.button,
170                 e.xbutton.x,
171                 e.xbutton.y);
172 
173             // Mouse Move
174             else if (e.type == MotionNotify)
175                 drawing->mouseMove(this,
176                 e.xmotion.x,
177                 e.xmotion.y);
178 
179             // This will respond to the WM_DELETE_WINDOW from the
180             // window manager.
181             else if (e.type == ClientMessage)
182             break;
183         }
184 }
185 
186 
187 int X11Context::getWindowWidth()
188 {
189     XWindowAttributes window_attributes;
190     XGetWindowAttributes(display,window, &window_attributes);
191     return window_attributes.width;
192 }
193 
194 int X11Context::getWindowHeight()
195 {
196     XWindowAttributes window_attributes;
197     XGetWindowAttributes(display,window, &window_attributes);
198     return window_attributes.height;
199 }
200 
201 // leave these out for now
202 //void X11Context::drawLine(int x1, int y1, int x2, int y2)
203 //{
204 //  XDrawLine(display, window, graphics_context, x1, y1, x2, y2);
205 //  XFlush(display);
206 //}
207 
208 //void X11Context::drawCircle(int x, int y, int radius)
209 //{
210 //  XDrawArc(display, window, graphics_context, x-radius,
211 //               y-radius, radius*2, radius*2, 0, 360*64);
212 //  XFlush(display);
213 //}
```

214

```
1  #ifndef DRAWBASE_H
2  #define DRAWBASE_H
3
4  // forward reference
5  class GraphicsContext;
6
7  class DrawingBase
8  {
9      public:
10         // prevent warnings
11         virtual ~DrawingBase(){}
12         virtual void paint(GraphicsContext* gc){}
13         virtual void keyDown(GraphicsContext* gc, unsigned int keycode){}
14         virtual void keyUp(GraphicsContext* gc, unsigned int keycode){}
15         virtual void mouseButtonDown(GraphicsContext* gc,
16                                      unsigned int button, int x, int y){}
17         virtual void mouseButtonUp(GraphicsContext* gc,
18                                      unsigned int button, int x, int y){}
19         virtual void mouseMove(GraphicsContext* gc, int x, int y){}
20 };
21 #endif
```

```
1  CC=g++
2  CFLAGS=-c -Wall
3  LDFLAGS= -lX11
4  SOURCES=main.cpp gcontext.cpp x11context.cpp shape.cpp matrix.cpp triangle.cpp line.cpp im
   age.cpp mydrawing.cpp viewcontext.cpp
5  OBJECTS=$(SOURCES:.cpp=.o)
6  EXECUTABLE=draw
7
8  all: $(SOURCES) $(EXECUTABLE)
9
10 # pull in dependency info for *existing* .o files
11 -include $(OBJECTS:.o=.d)
12
13 $(EXECUTABLE): $(OBJECTS)
14     $(CC) $(OBJECTS) $(LDFLAGS) -o $@
15
16 .cpp.o:
17     $(CC) $(CFLAGS) $< -o $@
18     $(CC) -MM $(CFLAGS) $< > $*.d
19
20 clean:
21     rm -rf $(OBJECTS) $(EXECUTABLE) *.d
```

Table of Contents