```
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
```

```cpp
/* This is an abstract base class representing a generic graphics
 * context.  Most implementation specifics will need to be provided by
 * a concrete implementation.  See header file for specifics. */

#define _USE_MATH_DEFINES   // for M_PI
#include <cmath>    // for trig functions
#include "gcontext.h"

/*
 * Destructor - does nothing
 */
GraphicsContext::~GraphicsContext()
{
    // nothing to do
    // here to insure subclasses handle destruction properly
}

//does nothing
void GraphicsContext::drawLine(int x0, int y0, int x1, int y1){}
void GraphicsContext::drawCircle(int x0, int y0, unsigned int radius){}


void GraphicsContext::endLoop()
{
    run = false;
}
```

```
#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.
 *
 * */


// 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;

        // Set the current color.  Implementations should default to white.
        // color is 24-bit RGB value
        virtual void setColor(unsigned int color) = 0;

        // Set pixel to the current color
        virtual void setPixel(int x, int y) = 0;

        // Get 24-bit RGB pixel color at specified location
        // unsigned int will likely be 32-bit on 32-bit systems, and
        // possible 64-bit on some 64-bit systems.  In either case,
        // it is large enough to hold a 16-bit color.
        virtual unsigned int getPixel(int x, int y) = 0;
```

```
79
80          // This should reset entire context to the current background
81          virtual void clear()=0;
82
83          // These are the naive implementations that use setPixel,
84          // but are overridable should a context have a better-
85          // performing version available.
86
87           /* will need to be provided by the concrete
88           * implementation.
89           *
90           * Parameters:
91           *  x0, y0 - origin of line
92           *  x1, y1 - end of line
93           *
94           * Returns: void
95           */
96          virtual void drawLine(int x0, int y0, int x1, int y1);
97
98          /* will need to be provided by the concrete
99           * implementation.
100          *
101          * Parameters:
102          *  x0, y0 - origin/center of circle
103          *  radius - radius of circle
104          *
105          * Returns: void
106          */
107         virtual void drawCircle(int x0, int y0, unsigned int radius);
108
109
110         /************************************************************
111          * Event loop operations
112          ************************************************************/
113
114         // Run Event loop.  This routine will receive events from
115         // the implementation and pass them along to the drawing.  It
116         // will return when the window is closed or other implementation-
117         // specific sequence.
118         virtual void runLoop(DrawingBase* drawing) = 0;
119
120         // This method will end the current loop if one is running
121         // a default version is supplied
122         virtual void endLoop();
123
124
125         /************************************************************
126          * Utility operations
127          ************************************************************/
128
129         // returns the width of the window
130         virtual int getWindowWidth() = 0;
131
132         // returns the height of the window
133         virtual int getWindowHeight() = 0;
134
135     protected:
136         // this flag is used to control whether the event loop
137         // continues to run.
138         bool run;
139 };
140
141 #endif
```

```cpp
#include <iostream>
#include <vector>
#include "triangle.h"
#include "line.h"
#include "shape.h"
#include "x11context.h"
#include "drawbase.h"
#include "gcontext.h"
#include "matrix.h"
#include "image.h"
#include "viewcontext.h"
using namespace std;

// Constructor
Image::Image()
{
}

// Copy Constructor
Image::Image(const Image &from)
{
    for (int i = 0; i < from.shapes.size(); i++)
    {
        shapes.push_back(from.shapes[i]->clone());
    }
}

// Destructor
Image::~Image()
{
    erase();
}

void Image::operator=(const Image &rhs)
{
    erase();
    for (int i = 0; i < rhs.shapes.size(); i++)
    {
        shapes.push_back(rhs.shapes[i]->clone());
    }
}

// Add a line to the shapes container
void Image::addLine(int x0, int y0, int x1, int y1, unsigned int color)
{
    shapes.push_back(new Line(x0, y0, x1, y1, color));
}

// Add a triangle to the shapes container
void Image::addTriangle(double x0, double y0, double x1, double y1, double x2, double y2,
unsigned int color)
{
    shapes.push_back(new Triangle(x0, y0, x1, y1, x2, y2, color));
}

// Draw all lines/triangles in the shapes container
void Image::draw(GraphicsContext *gc, ViewContext *vc)
{
    for (int i = 0; i < shapes.size(); i++)
    {
        shapes[i]->draw(gc, vc);
    }
}

// Erase all shapes and return all dynamic memory
void Image::erase()
{
    for (int i = 0; i < shapes.size(); i++)
    {
        delete shapes[i];
    }
    shapes.clear();
}

Image Image::undoShape(Image im)
{
    im.shapes.pop_back();
    return im;
```

```
78  }
```

```
#ifndef image_h
#define image_h

#include <iostream>
#include <vector>
#include "shape.h"
#include "matrix.h"
#include "line.h"
#include "triangle.h"
#include "viewcontext.h"
using namespace std;

class Image
{
public:
    Image( );
    Image(const Image &from);
    ~Image();
    void operator=(const Image &rhs);
    void addLine(int x0, int y0, int x1, int y1, unsigned int color);
    void addTriangle(double x0, double y0, double x1, double y1, double x2, double y2, unsigned int color);
    void draw(GraphicsContext *gc, ViewContext *vc);
    void erase();
    Image undoShape(Image im);

private:
    vector<Shape *> shapes;
    vector<Shape *> originalShapes;
    GraphicsContext *gc;
};


#endif
```

```cpp
#include <iostream>
#include "line.h"
#include "shape.h"
#include "x11context.h"
#include "drawbase.h"
#include "gcontext.h"
#include "matrix.h"
#include "viewcontext.h"
using namespace std;

// Line constructor
Line::Line(int x0, int y0, int x1, int y1, unsigned int color)
{
    this->coord0[0][0] = x0;
    this->coord0[1][0] = y0;
    this->coord0[2][0] = 0;
    this->coord0[3][0] = 1;

    this->coord1[0][0] = x1;
    this->coord1[1][0] = y1;
    this->coord1[2][0] = 0;
    this->coord1[3][0] = 1;

    this->color = color;
}

// Clone a line
Shape *Line::clone()
{
    return new Line(*this);
}

// Draw the line
void Line::draw(GraphicsContext *gc, ViewContext *vc)
{
    gc->setColor(color);
    Matrix point0 = vc->ModelToDevice(coord0);
    Matrix point1 = vc->ModelToDevice(coord1);
    gc->drawLine(point0[0][0], point0[1][0], point1[0][0], point1[1][0]);
}
```

```
#ifndef line_h
#define line_h

#include <iostream>
#include "shape.h"
#include "matrix.h"
#include "viewcontext.h"
using namespace std;

class Line : public Shape
{
public:
    Line(int x0, int y0, int x1, int y1, unsigned int color);
    Shape *clone();
    void draw(GraphicsContext *gc, ViewContext *vc);

private:
    Matrix coord0 = Matrix(4, 1);
    Matrix coord1 = Matrix(4, 1);
};

#endif
```

```cpp
#include "x11context.h"
#include <unistd.h>
#include <iostream>
#include "mydrawing.h"
#include <cstring>
int main(void)
{
    // GraphicsContext *gc = new X11Context(1910, 1200, GraphicsContext::BLACK);
    GraphicsContext *gc = new X11Context(1000, 800, GraphicsContext::BLACK);
    gc->setColor(GraphicsContext::GREEN);
    // make a drawing
    MyDrawing md(gc->getWindowWidth(), gc->getWindowHeight());
    // start event loop - this function will return when X is clicked
    // on window
    gc->runLoop(&md);
    delete gc;
    return 0;
}
```

```cpp
#include "matrix.h"
#include <iomanip>
using namespace std;

// constructor
Matrix::Matrix(int rows, int cols)
{
    if (rows <= 0 || cols <= 0)
    {
        throw std::out_of_range("The rows and columns must be greater than 0");
    }
    this->rows = rows;
    this->cols = cols;
    the_matrix = new Row *[rows];
    for (int i = 0; i < rows; i++)
    {
        the_matrix[i] = new Row(cols);
    }
}

// Copy constructor
Matrix::Matrix(const Matrix &from)
{
    rows = from.rows;
    cols = from.cols;

    the_matrix = new Row *[rows];
    for (int i = 0; i < rows; i++)
    {
        the_matrix[i] = new Row(cols);
        for (int j = 0; j < cols; j++)
        {
            (*the_matrix[i])[j] = from[i][j];
        }
    }
}

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

// Assignment operator. Check row.cpp from Lab 2 to see more accurately how to do this.
Matrix &Matrix::operator=(const Matrix &rhs)
{
    for (int i = 0; i < rows; i++)
    {
        delete the_matrix[i];
    }
    delete[] the_matrix;

    rows = rhs.rows;
    cols = rhs.cols;
    the_matrix = new Row *[rows];
    for (int i = 0; i < rows; i++)
    {
        the_matrix[i] = new Row(cols);
        for (int j = 0; j < cols; j++)
        {
            (*the_matrix[i])[j] = rhs[i][j];
        }
    }
    return (*this);
}

// Named Constructor
Matrix Matrix::identity(unsigned int size)
{
    Matrix result(size, size);
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
```

```
79              if (i == j)
80              {
81                  result[i][j] = 1;
82              }
83              else
84              {
85                  result[i][j] = 0;
86              }
87          }
88      }
89      return result;
90  }
91
92  // Matrix addition.
93  Matrix Matrix::operator+(const Matrix &rhs) const
94  {
95      // Check size is correct
96      if (rows != rhs.rows && cols != rhs.cols)
97      {
98          throw logic_error("Rows of both matrices and cols "
99                            "of both matrices must be equal");
100     }
101     Matrix result(rows, cols);
102     for (int i = 0; i < rows; i++)
103     {
104         for (int j = 0; j < cols; j++)
105         {
106             result[i][j] = (*this)[i][j] + rhs[i][j]; // not the_matrix[i][j]
107         }
108     }
109     return result;
110 }
111
112 // Matrix multiplication
113 Matrix Matrix::operator*(const Matrix &rhs) const
114 {
115     if (cols != rhs.rows)
116     {
117         throw logic_error("The cols of the first matrix "
118                           "must be equal to the rows of the second matrix.");
119     }
120     Matrix result(rows, rhs.cols);
121     for (int i = 0; i < result.rows; i++)
122     {
123         for (int j = 0; j < rhs.cols; j++)
124         {
125             for (int k = 0; k < cols; k++)
126             {
127                 result[i][j] += (*this)[i][k] * rhs[k][j];
128             }
129         }
130     }
131     return result;
132 }
133
134 // Scalar multiplication
135 Matrix Matrix::operator*(const double scale) const
136 {
137     Matrix result(this->rows, this->cols);
138     for (int i = 0; i < rows; i++)
139     {
140         for (int j = 0; j < cols; j++)
141         {
142             result[i][j] = ((*this)[i][j]) * scale;
143         }
144     }
145     return result;
146 }
147
148 // global scalar multiplication
149 Matrix operator*(const double scale, const Matrix &rhs)
150 {
151     Matrix result(rhs.rows, rhs.cols);
152     for (int i = 0; i < result.rows; i++)
153     {
154         for (int j = 0; j < result.cols; j++)
155         {
156             result[i][j] = scale * rhs[i][j];
```

```
157             }
158         }
159         return result;
160     }
161
162     // Transpose of a Matrix
163     Matrix Matrix::operator~() const
164     {
165         Matrix result(this->cols, this->rows);
166         for (int i = 0; i < this->rows; i++)
167         {
168             for (int j = 0; j < this->cols; j++)
169             {
170                 result[j][i] = (*this)[i][j];
171             }
172         }
173         return result;
174     }
175
176     // Clear Matrix
177     void Matrix::clear()
178     {
179         for (int i = 0; i < rows; i++)
180         {
181             for (int j = 0; j < cols; j++)
182             {
183                 (*this)[i][j] = 0;
184             }
185         }
186     }
187
188     // Access Operators - non-const
189     Row &Matrix::operator[](unsigned int row)
190     {
191         if (row < 0 || row >= rows)
192         {
193             throw out_of_range("Row cannot be less than 0 or "
194                                "greater than the amount of rows in matrix");
195         }
196         return *(the_matrix[row]);
197     }
198
199     // Access Operators - const
200     Row Matrix::operator[](unsigned int row) const
201     {
202         if (row < 0 || row >= rows)
203         {
204             throw out_of_range("Row cannot be less than 0 or "
205                                "greater than the amount of rows in matrix");
206         }
207         return *(the_matrix[row]);
208     }
209
210     // global insertion operator... ios_base
211     std::ostream &operator<<(std::ostream &os, const Matrix &rhs)
212     {
213         os.precision(6);
214         for (int i = 0; i < rhs.rows; i++)
215         {
216             cout << "[";
217             for (int j = 0; j < rhs.cols; j++)
218             {
219                 os << setw(6);
220                 os << rhs[i][j];
221                 os << setw(6);
222             }
223             os << "]" << endl;
224         }
225
226         return os;
227     }
```

```
#ifndef matrix_h
#define matrix_h

#include <iostream>
#include "row.h"
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.

    // Constructor - create Matrix and clear cells.  If rows or
    // cols is < 1, throw an exception
    Matrix(int rows, 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 - 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
    // would not make sense and should throw an exception.
    static Matrix identity(unsigned int size);

    // Matrix addition - lhs and rhs must be same size otherwise
    // an exception shall be thrown
    Matrix operator+(const Matrix &rhs) const;

    // Matrix multiplication - lhs and rhs must be compatible
    // otherwise an exception shall be thrown
    Matrix operator*(const Matrix &rhs) const;

    // Scalar multiplication.  Note, this function will support
    // someMatrixObject * 5.0, but not 5.0 * someMatrixObject.
    Matrix operator*(const double scale) const;

    // Matrix scalar multiplication when the scalar is first
    // 5.0 * someMatrixObject;
    friend Matrix operator*(const double scale, const Matrix &rhs);

    // Transpose of a Matrix - should always work, hence no exception
    Matrix operator~() const;

    // Clear Matrix to all members 0.0
    void clear();

    // Access Operators - throw an exception if index out of range
    Row &operator[](unsigned int row);

    // const version of above - throws an exception if indices are out of
    // range
    Row operator[](unsigned int row) const;

    friend std::ostream &operator<<(std::ostream &os, const Matrix &rhs);

private:
    // An array of Row pointers size "rows" that each point to a double array
    // of size "cols"
    Row **the_matrix;
    unsigned int rows;
    unsigned int cols;

    /** routines **/

    // add any "helper" routine here, such as routines to support
    // matrix inversion
};

/** Some Related Global Functions **/

```

```
79  // Overloaded global << with std::ostream as lhs, Matrix as rhs.  This method
80  // should generate output compatible with an ostream which is commonly used
81  // with console (cout) and files.  Something like:
82  // [[ r0c0, r0c1, r0c2 ]
83  //  [ r1c0, r1c1, r1c2 ]
84  //  [ r0c0, r0c1, r0c2 ]]
85  // would be appropriate.
86  //
87  // You should make this function a "friend" of the Matrix class so it can acess
88  // private data members
89  std::ostream &operator<<(std::ostream &os, const Matrix &rhs);
90
91  // We would normally have a corresponding >> operator, but
92  // will defer that exercise that until a later assignment.
93
94  // Scalar multiplication with a global function.  Note, this function will
95  // support 5.0 * someMatrixObject, but not someMatrixObject * 5.0
96  Matrix operator*(const double scale, const Matrix &rhs);
97
98  #endif
99  // Based on lab by Dr. Darrin Rothe ((c) 2015 Dr. Darrin Rothe)
```

```
#include "mydrawing.h"
#include "gcontext.h"
#include "viewcontext.h"
#include "matrix.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <limits>
#include <cstring>
using namespace std;

// Constructor
MyDrawing::MyDrawing(int width, int height)
{
    cout << "COLORS:" << endl;
    cout << "1: White" << endl;
    cout << "2: Black" << endl;
    cout << "3: Red" << endl;
    cout << "4: Yellow" << endl;
    cout << "5: Blue" << endl;
    cout << "6: Green" << endl;
    cout << "Press T to draw a triangle." << endl;
    cout << "Press L to draw a line." << endl;
    cout << "To undo previous shape, press backspace." << endl;
    cout << endl;
    cout << "To translate the image, use the arrow keys respectively." << endl;
    cout << "To rotate: Q-Counter Clockwise; E-Clockwise." << endl;
    cout << "To scale: W-Scale up; S-Scale down." << endl;
    cout << "To return back to normal: Enter Key." << endl;
    cout << "To insert an image from stl file: Z" << endl;
    numClicks = 0;                    // Track the number of clicks
    mode = 0;                         // Default mode is line
    color = GraphicsContext::GREEN; // Default color is green
    vc = new ViewContext(width, height);
}
// Destructor
MyDrawing::~MyDrawing()
{
    delete vc;
}
void MyDrawing::paint(GraphicsContext *gc)
{
    im.draw(gc, vc);
}
void MyDrawing::mouseButtonDown(GraphicsContext *gc, unsigned int button, int x, int y)
{
    if (mode == 0) // Line
    {
        if (numClicks == 0) // 1st click
        {
            x0 = x;
            y0 = y;
            numClicks++;
        }
        else // 2nd click. Draw line
        {
            gc->drawLine(x0, y0, x, y);
            coord0[0][0] = x0;
            coord0[1][0] = y0;
            coord0[3][0] = 1;
            coord1[0][0] = x;
            coord1[1][0] = y;
            coord1[3][0] = 1;
            Matrix point0 = vc->DeviceToModel(coord0);
            Matrix point1 = vc->DeviceToModel(coord1);
            im.addLine(point0[0][0], point0[1][0], point1[0][0], point1[1][0], color);
            numClicks = 0;
        }
    }
    else if (mode == 1) // Triangle
    {
        if (numClicks == 0) // 1st click
        {
            x0 = x;
            y0 = y;
            numClicks++;
        }
        else if (numClicks == 1) // 2nd click
```

```
79          {
80              x1 = x;
81              y1 = y;
82              numClicks++;
83          }
84          else // 3rd click. Draw triangle
85          {
86              gc->drawLine(x0, y0, x1, y1);
87              gc->drawLine(x0, y0, x, y);
88              gc->drawLine(x1, y1, x, y);
89              coord0[0][0] = x0;
90              coord0[1][0] = y0;
91              coord0[3][0] = 1;
92              coord1[0][0] = x1;
93              coord1[1][0] = y1;
94              coord1[3][0] = 1;
95              coord2[0][0] = x;
96              coord2[1][0] = y;
97              coord2[3][0] = 1;
98              Matrix point0 = vc->DeviceToModel(coord0);
99              Matrix point1 = vc->DeviceToModel(coord1);
100             Matrix point2 = vc->DeviceToModel(coord2);
101             im.addTriangle(point0[0][0], point0[1][0], point1[0][0], point1[1][0], point2[
    0][0], point2[1][0], color);
102             numClicks = 0;
103         }
104     }
105 }
106 void MyDrawing::undoShape(GraphicsContext *gc)
107 {
108     gc->clear();
109     im = im.undoShape(im);
110     paint(gc);
111 }
112 void MyDrawing::rotateClockwise(GraphicsContext *gc)
113 {
114     vc->rotateClockwise();
115     gc->clear();
116     paint(gc);
117 }
118 void MyDrawing::rotateCounterclockwise(GraphicsContext *gc)
119 {
120     vc->rotateCounterclockwise();
121     gc->clear();
122     paint(gc);
123 }
124 void MyDrawing::scaleUp(GraphicsContext *gc)
125 {
126     vc->scaleUp();
127     gc->clear();
128     paint(gc);
129 }
130 void MyDrawing::scaleDown(GraphicsContext *gc)
131 {
132     vc->scaleDown();
133     gc->clear();
134     paint(gc);
135 }
136 void MyDrawing::translateUp(GraphicsContext *gc)
137 {
138     vc->translateUp();
139     gc->clear();
140     paint(gc);
141 }
142 void MyDrawing::translateRight(GraphicsContext *gc)
143 {
144     vc->translateRight();
145     gc->clear();
146     paint(gc);
147 }
148 void MyDrawing::translateDown(GraphicsContext *gc)
149 {
150     vc->translateDown();
151     gc->clear();
152     paint(gc);
153 }
154 void MyDrawing::translateLeft(GraphicsContext *gc)
155 {
```

```
156     vc->translateLeft();
157     gc->clear();
158     paint(gc);
159 }
160 void MyDrawing::undoAll(GraphicsContext *gc, ViewContext *vc)
161 {
162     vc->undoAll();
163     gc->clear();
164     paint(gc);
165 }
166 void MyDrawing::readFromFile(string filename)
167 {
168     ifstream ifile(filename);
169     // Empty string to store line from stl file
170     string line;
171     // Variables to store x,y,z file data in
172     double x0;
173     double y0;
174     double z0;
175     double x1;
176     double y1;
177     double z1;
178     double x2;
179     double y2;
180     double z2;
181     string type;
182     int count = 0;
183     // Read lines of the stl file until the last one is reached
184     while (!ifile.eof())
185     {
186         // Store next line of file
187         getline(ifile, line);
188         // Create input string stream connected to line string
189         istringstream iss(line);
190         // Extract data from file
191         iss >> type;
192         int vertexR = type.compare("vertex");
193         if (vertexR == 0 && count == 0)
194         {
195             iss >> x0;
196             iss >> y0;
197             iss >> z0;
198             count++;
199         }
200         else if (vertexR == 0 && count == 1)
201         {
202             iss >> x1;
203             iss >> y1;
204             iss >> z1;
205             count++;
206         }
207         else if (vertexR == 0 && count == 2)
208         {
209             iss >> x2;
210             iss >> y2;
211             iss >> z1;
212             count = 0;
213             im.addTriangle(x0, y0, x1, y1, x2, y2, color);
214         }
215     }
216 }
217 void MyDrawing::keyDown(GraphicsContext *gc, unsigned int keycode)
218 {
219     // cout << keycode << endl;
220     switch (keycode)
221     {
222     case 0x31:
223         gc->setColor(GraphicsContext::WHITE);
224         color = GraphicsContext::WHITE;
225         break;
226     case 0x32:
227         gc->setColor(GraphicsContext::BLACK);
228         color = GraphicsContext::BLACK;
229         break;
230     case 0x33:
231         gc->setColor(GraphicsContext::RED);
232         color = GraphicsContext::RED;
233         break;
```

```
234         case 0x34:
235             gc->setColor(GraphicsContext::YELLOW);
236             color = GraphicsContext::YELLOW;
237             break;
238         case 0x35:
239             gc->setColor(GraphicsContext::BLUE);
240             color = GraphicsContext::BLUE;
241             break;
242         case 0x36:
243             gc->setColor(GraphicsContext::GREEN);
244             color = GraphicsContext::GREEN;
245             break;
246         case 0x6C:    // L key
247             mode = 0; // Line mode
248             break;
249         case 0x74:    // T key
250             mode = 1; // Triangle mode
251             break;
252         case 0xFF08: // Backspace key
253             undoShape(gc);
254             break;
255         case 0x65: // E (Rotate clockwise)
256             rotateClockwise(gc);
257             break;
258         case 0x71: // Q (Rotate counter clockwise)
259             rotateCounterclockwise(gc);
260             break;
261         case 0x77: // W Scale up
262             scaleUp(gc);
263             break;
264         case 0x73: // S Scale down
265             scaleDown(gc);
266             break;
267         case 0xFF52: // Up arrow translate up
268             translateUp(gc);
269             break;
270         case 0xFF53: // Right arrow translate right
271             translateRight(gc);
272             break;
273         case 0xFF54: // Down arrow translate down
274             translateDown(gc);
275             break;
276         case 0xFF51: // Left arrow translate left
277             translateLeft(gc);
278             break;
279         case 0x75: // Return back to normal, U key
280             undoAll(gc, vc);
281             break;
282         case 0x7A: // Insert stl file, Z key
283             cout << "Enter file name: " << endl;
284             string fileinput;
285             cin >> fileinput;
286             gc->clear();
287             im.erase();
288             readFromFile(fileinput);
289             paint(gc);
290             break;
291         }
292 }
```

```
#ifndef MYDRAWING_H
#define MYDRAWING_H
#include "drawbase.h"
#include "image.h"
#include "viewcontext.h"
#include "matrix.h"

// forward reference
class GraphicsContext;
class MyDrawing : public DrawingBase
{
public:
    MyDrawing(int width, int height);
    // we will override just these
    virtual void paint(GraphicsContext *gc);
    virtual void mouseButtonDown(GraphicsContext *gc, unsigned int button, int x, int y);
    virtual void keyDown(GraphicsContext *gc, unsigned int keycode);
    ~MyDrawing();
    void readFromFile(string filename);

private:
    Image im;
    Image copyIm;
    // We will only support one "remembered" line
    int x0;
    int y0;
    int x1;
    int y1;
    int numClicks;
    int mode;       // 0 == line, 1 == triangle
    unsigned int color;
    void undoShape(GraphicsContext *gc);
    ViewContext *vc;
    void rotateClockwise(GraphicsContext *gc);
    void rotateCounterclockwise(GraphicsContext *gc);
    void scaleUp(GraphicsContext *gc);
    void scaleDown(GraphicsContext *gc);
    void translateUp(GraphicsContext *gc);
    void translateRight(GraphicsContext *gc);
    void translateDown(GraphicsContext *gc);
    void translateLeft(GraphicsContext *gc);
    void undoAll(GraphicsContext *gc, ViewContext *vc);

    Matrix coord0 = Matrix(4, 1);
    Matrix coord1 = Matrix(4, 1);
    Matrix coord2 = Matrix(4, 1);
};
#endif
```

```cpp
#include <iostream>
#include "row.h"
using namespace std;

// parameterized constructor
Row::Row(int length)
{
    if (length <= 0)
    {
        throw std::out_of_range("The length of the row has to be greater than 0");
    }
    this->length = length; // this->length is making the length for the Row, while length is the length that is input
    row_data = new double[length];
    clear();
}

// copy constructor
Row::Row(const Row &from)
{
    length = from.length;
    row_data = new double[length];
    for (int i = 0; i < length; i++)
    {
        row_data[i] = from.row_data[i];
    }
}

// destructor
Row::~Row()
{
    delete[] row_data;
}

// access operator (const)
double Row::operator[](int column) const
{
    if (column < 0 || column >= length)
    {
        throw out_of_range("Column must be >= 0 and < length");
    }
    return row_data[column];
}

// access operator (non-const)
double &Row::operator[](int column)
{
    if (column < 0 || column >= length)
    {
        throw out_of_range("Column must be >= 0 and < length");
    }
    return row_data[column];
}

// assignment operator
Row &Row::operator=(const Row &rhs)
{
    if (this != &rhs)
    {
        length = rhs.length;
        delete[] row_data;
        row_data = new double[length];
        for (int i = 0; i < length; i++)
        {
            this->row_data[i] = rhs.row_data[i];
        }
    }
    return *this;
}

// clear row data
void Row::clear()
{
    for (int i = 0; i < length; i++)
    {
        row_data[i] = 0;
    }
}
```

```
#ifndef row_h
#define row_h
class Row{
    public:
        /* Parameterized constructor
         * Takes in length and creates a row matrix with values cleared
         * to zero
         * Should verify length > 0
         */
        Row(int length);

        /* Copy constructor
         * Create a new row matrix with the same size and values as the
         * from matrix
         */
        Row(const Row& from);

        /* Destructor
         * Correctly delete any heap memory
         */
        ~Row();

        /* Access operator (const version)
         * Allow access to row matrix data
         * Should return an exception if column is too large
        */
        double operator[](int column) const;

        /* Access operator (non const version)
         * Allow access to row matrix data
         * Should return an exception if column is too large
         */
        double& operator[] (int column);

        /* Assignment operator
         * 1. Check if two sides are the same object
         * 2. Delete the current row matrix
         * 3. Create a new row matrix with the same size and values as
         *    the rhs matrix
         */
        Row& operator= (const Row& rhs);

        /* Clear all data values to zero
         */
        void clear();
    private:
        // Row matrix data
        double * row_data;
        // Size of row matrix
        unsigned int length;
};
#endif
```

```
1  #ifndef shape_h
2  #define shape_h
3
4  #include <iostream>
5  #include "x11context.h"
6  #include "gcontext.h"
7  #include "viewcontext.h"
8  using namespace std;
9
10 class Shape
11 {
12 public:
13     virtual ~Shape(){};
14     virtual void draw(GraphicsContext *, ViewContext *) = 0;
15     virtual Shape *clone() = 0;
16
17 protected:
18     unsigned int color;
19 };
20
21 #endif
```

```cpp
#include <iostream>
#include "triangle.h"
#include "shape.h"
#include "x11context.h"
#include "drawbase.h"
#include "gcontext.h"
#include "matrix.h"
#include "viewcontext.h"
using namespace std;

// Triangle constructor
Triangle::Triangle(double x0, double y0, double x1, double y1, double x2, double y2, unsigned int color)
{
    this->coord0[0][0] = x0;
    this->coord0[1][0] = y0;
    this->coord0[2][0] = 0;
    this->coord0[3][0] = 1;

    this->coord1[0][0] = x1;
    this->coord1[1][0] = y1;
    this->coord1[2][0] = 0;
    this->coord1[3][0] = 1;

    this->coord2[0][0] = x2;
    this->coord2[1][0] = y2;
    this->coord2[2][0] = 0;
    this->coord2[3][0] = 1;

    this->color = color;
}

// Clone a triangle
Shape *Triangle::clone()
{
    return new Triangle(*this);
}

// Draw the triangle
void Triangle::draw(GraphicsContext *gc, ViewContext *vc)
{
    gc->setColor(color);
    Matrix point0 = vc->ModelToDevice(coord0);
    Matrix point1 = vc->ModelToDevice(coord1);
    Matrix point2 = vc->ModelToDevice(coord2);
    gc->drawLine(point0[0][0], point0[1][0], point1[0][0], point1[1][0]);
    gc->drawLine(point0[0][0], point0[1][0], point2[0][0], point2[1][0]);
    gc->drawLine(point1[0][0], point1[1][0], point2[0][0], point2[1][0]);
}
```

```
#ifndef triangle_h
#define triangle_h

#include <iostream>
#include "shape.h"
#include "matrix.h"
#include "viewcontext.h"
using namespace std;

class Triangle : public Shape
{
public:
    Triangle(double x0, double y0, double x1, double y1, double x2, double y2, unsigned int color);
    Shape *clone();
    void draw(GraphicsContext *gc, ViewContext *vc);

private:
    Matrix coord0 = Matrix(4, 1);
    Matrix coord1 = Matrix(4, 1);
    Matrix coord2 = Matrix(4, 1);
};


#endif
```

```cpp
#include <iostream>
#include <cmath>
#include "viewcontext.h"
#include "matrix.h"
using namespace std;

// Constructor
ViewContext::ViewContext(int width, int height)
{
    this->width = width;
    this->height = height;
    modelToDevice[0][0] = 1;
    modelToDevice[0][3] = width / 2;
    modelToDevice[1][1] = -1;
    modelToDevice[1][3] = height / 2;
    modelToDevice[2][2] = 1;
    modelToDevice[3][3] = 1;

    deviceToModel[0][0] = 1;
    deviceToModel[0][3] = width / -2;
    deviceToModel[1][1] = -1;
    deviceToModel[1][3] = height / 2;
    deviceToModel[2][2] = 1;
    deviceToModel[3][3] = 1;

    // Translate to origin
    originTranslate[0][3] = width / -2;
    originTranslate[1][3] = height / -2;
    inverseOriginTranslate[0][3] = width / 2;
    inverseOriginTranslate[1][3] = height / 2;

    // Translate to center of screen
    centerTranslate[0][3] = width / 2;
    centerTranslate[1][3] = height / 2;
    inverseCenterTranslate[0][3] = width / -2;
    inverseCenterTranslate[1][3] = height / -2;
}

// Model To Device
Matrix ViewContext::ModelToDevice(Matrix point)
{
    return modelToDevice * point;
}

// Device to model
Matrix ViewContext::DeviceToModel(Matrix point)
{
    return deviceToModel * point;
}

// Translate up by 10px
void ViewContext::translateUp()
{
    Matrix inverseTransform = Matrix::identity(4);
    Matrix transform = Matrix::identity(4);
    inverseTransform[1][3] = 10;
    transform[1][3] = -10;
    modelToDevice = transform * modelToDevice;
    deviceToModel = deviceToModel * inverseTransform;
}

// Translate right by 10px
void ViewContext::translateRight()
{
    Matrix inverseTransform = Matrix::identity(4);
    Matrix transform = Matrix::identity(4);
    inverseTransform[0][3] = -10;
    transform[0][3] = 10;
    modelToDevice = transform * modelToDevice;
    deviceToModel = deviceToModel * inverseTransform;
}

// Translate down by 10px
void ViewContext::translateDown()
{
    Matrix inverseTransform = Matrix::identity(4);
    Matrix transform = Matrix::identity(4);
    inverseTransform[1][3] = -10;
```

```
79      transform[1][3] = 10;
80      modelToDevice = transform * modelToDevice;
81      deviceToModel = deviceToModel * inverseTransform;
82  }
83
84  // Translate left by 10px
85  void ViewContext::translateLeft()
86  {
87      Matrix inverseTransform = Matrix::identity(4);
88      Matrix transform = Matrix::identity(4);
89      inverseTransform[0][3] = 10;
90      transform[0][3] = -10;
91      modelToDevice = transform * modelToDevice;
92      deviceToModel = deviceToModel * inverseTransform;
93  }
94
95  // Translate to the origin
96  void ViewContext::translateOrigin()
97  {
98      modelToDevice = originTranslate * modelToDevice;
99      deviceToModel = deviceToModel * inverseOriginTranslate;
100 }
101
102 // Translate to the center of the screen
103 void ViewContext::translateCenter()
104 {
105     modelToDevice = centerTranslate * modelToDevice;
106     deviceToModel = deviceToModel * inverseCenterTranslate;
107 }
108 // Scale the image up by 2
109 void ViewContext::scaleUp()
110 {
111     // Translate to the origin
112     translateOrigin();
113     Matrix inverseTransform = Matrix::identity(4);
114     Matrix transform = Matrix::identity(4);
115     inverseTransform[0][0] = 1 / 2;
116     inverseTransform[1][1] = 1 / 2;
117     transform[0][0] = 2;
118     transform[1][1] = 2;
119     modelToDevice = transform * modelToDevice;
120     deviceToModel = deviceToModel * inverseTransform;
121     translateCenter();
122 }
123
124 void ViewContext::scaleDown()
125 {
126     // Translate to the origin
127     translateOrigin();
128     Matrix inverseTransform = Matrix::identity(4);
129     Matrix transform = Matrix::identity(4);
130     inverseTransform[0][0] = 1 / 0.5;
131     inverseTransform[1][1] = 1 / 0.5;
132     transform[0][0] = 0.5;
133     transform[1][1] = 0.5;
134     modelToDevice = transform * modelToDevice;
135     deviceToModel = deviceToModel * inverseTransform;
136     translateCenter();
137 }
138
139 void ViewContext::rotateClockwise()
140 {
141     translateOrigin();
142     Matrix inverseTransform = Matrix::identity(4);
143     Matrix transform = Matrix::identity(4);
144     transform[0][0] = cos(-10 * M_PI / 180);
145     transform[0][1] = sin(-10 * M_PI / 180);
146     transform[1][0] = -sin(-10 * M_PI / 180);
147     transform[1][1] = cos(-10 * M_PI / 180);
148     inverseTransform[0][0] = cos(10 * M_PI / 180);
149     inverseTransform[0][1] = sin(10 * M_PI / 180);
150     inverseTransform[1][0] = -sin(10 * M_PI / 180);
151     inverseTransform[1][1] = cos(10 * M_PI / 180);
152     modelToDevice = transform * modelToDevice;
153     deviceToModel = deviceToModel * inverseTransform;
154     translateCenter();
155 }
156
```

```
157 void ViewContext::rotateCounterclockwise()
158 {
159     translateOrigin();
160     Matrix inverseTransform = Matrix::identity(4);
161     Matrix transform = Matrix::identity(4);
162     transform[0][0] = cos(10 * M_PI / 180);
163     transform[0][1] = sin(10 * M_PI / 180);
164     transform[1][0] = -sin(10 * M_PI / 180);
165     transform[1][1] = cos(10 * M_PI / 180);
166     inverseTransform[0][0] = cos(-10 * M_PI / 180);
167     inverseTransform[0][1] = sin(-10 * M_PI / 180);
168     inverseTransform[1][0] = -sin(-10 * M_PI / 180);
169     inverseTransform[1][1] = cos(-10 * M_PI / 180);
170     modelToDevice = transform * modelToDevice;
171     deviceToModel = deviceToModel * inverseTransform;
172     translateCenter();
173 }
174
175 void ViewContext::undoAll()
176 {
177     for (int i = 0; i < 4; i++)
178     {
179         for (int j = 0; j < 4; j++)
180         {
181             modelToDevice[i][j] = 0;
182             deviceToModel[i][j] = 0;
183         }
184     }
185     modelToDevice[0][0] = 1;
186     modelToDevice[0][3] = width / 2;
187     modelToDevice[1][1] = -1;
188     modelToDevice[1][3] = height / 2;
189     modelToDevice[2][2] = 1;
190     modelToDevice[3][3] = 1;
191
192     deviceToModel[0][0] = 1;
193     deviceToModel[0][3] = width / -2;
194     deviceToModel[1][1] = -1;
195     deviceToModel[1][3] = height / 2;
196     deviceToModel[2][2] = 1;
197     deviceToModel[3][3] = 1;
198 }
```

```
#ifndef viewcontext_h
#define viewcontext_h

#include <iostream>
#include "matrix.h"
using namespace std;

class ViewContext
{
public:
    ViewContext(int width, int height);
    Matrix ModelToDevice(Matrix point);
    Matrix DeviceToModel(Matrix point);
    void translateUp();
    void translateRight();
    void translateDown();
    void translateLeft();
    void scaleUp();
    void scaleDown();
    void rotateCounterclockwise();
    void rotateClockwise();
    void undoAll();

private:
    Matrix modelToDevice = Matrix(4, 4);
    Matrix deviceToModel = Matrix(4, 4);
    Matrix originTranslate = Matrix::identity(4);
    Matrix centerTranslate = Matrix::identity(4);
    Matrix inverseOriginTranslate = Matrix::identity(4);
    Matrix inverseCenterTranslate = Matrix::identity(4);
    void translateOrigin();
    void translateCenter();
    int width;
    int height;
};
#endif
```

```
/* Provides a simple drawing context for X11/XWindows
 * You must have the X11 dev libraries installed.  If missing,
 * 'sudo apt-get install libx11-dev' should help.
 */

#include <X11/Xlib.h> // Every Xlib program must include this
#include <X11/Xutil.h> // needed for XGetPixel
#include <X11/XKBlib.h> // needed for keyboard setup
#include "x11context.h"
#include "drawbase.h"
#include <iostream>

/**
 * The only constructor provided.  Allows size of window and background
 * color be specified.
 * */
X11Context::X11Context(unsigned int sizex=400,unsigned int sizey=400,
                       unsigned int bg_color=GraphicsContext::BLACK)
{
    // Open the display
    display = XOpenDisplay(NULL);

    // Holding a key in gives repeated key_press commands but only
    // one key_release
    int supported;

    XkbSetDetectableAutoRepeat(display,true,&supported);

    // Create a window - we will assume the color map is in RGB mode.
    window = XCreateSimpleWindow(display, DefaultRootWindow(display), 0, 0,
                 sizex, sizey, 0, 0 , bg_color);

    // Sign up for MapNotify events
    XSelectInput(display, window, StructureNotifyMask);

    // Put the window on the screen
    XMapWindow(display, window);

    // Create a "Graphics Context"
    graphics_context = XCreateGC(display, window, 0, NULL);

    // Default color to white
    XSetForeground(display, graphics_context, GraphicsContext::WHITE);

    // Wait for MapNotify event
    for(;;)
    {
        XEvent e;
        XNextEvent(display, &e);
        if (e.type == MapNotify)
        break;
    }

    // We also want exposure, mouse, and keyboard events
    XSelectInput(display, window, ExposureMask|
                                  ButtonPressMask|
                                  ButtonReleaseMask|
                                  KeyPressMask|
                                  KeyReleaseMask|
                                  PointerMotionMask);

    // We need this to get the WM_DELETE_WINDOW message from the
    // window manager in case user click the X icon
    Atom atomKill = XInternAtom(display, "WM_DELETE_WINDOW", False);
    XSetWMProtocols(display, window, &atomKill, 1);

    return;
}

// Destructor  - shut down window and connection to server
X11Context::~X11Context()
{
    XFreeGC(display, graphics_context);
    XDestroyWindow(display,window);
    XCloseDisplay(display);
}

// 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     void X11Context::drawLine(int x1, int y1, int x2, int y2)
202     {
203         XDrawLine(display, window, graphics_context, x1, y1, x2, y2);
204         XFlush(display);
205     }
206
207     void X11Context::drawCircle(int x, int y, unsigned int radius)
208     {
209         XDrawArc(display, window, graphics_context, x-radius,
210                         y-radius, radius*2, radius*2, 0, 360*64);
211         XFlush(display);
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,unsigned int sizey,unsigned int bg_color);

        // 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();
        void drawLine(int x1, int y1, int x2, int y2);
        void drawCircle(int x, int y, unsigned 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
```

Table of Contents