

```

1  /**
2      *****
3      * @file      : main.cpp
4      * @brief     : Main program
5      *           : Lab 6: Event Driven Drawing
6      *           : CS-3210/021
7      * @date      : May 11 2021
8      * @author    : Julian Singkham
9      *****
10     * @attention
11     * This lab is an extension of the previous lab where the user can now interact with
12     * the graphics window.
13     *
14     * Features:
15     *   User can draw points, lines, and triangles with 9 different colors.
16     *   All draw methods rubberband.
17     *   User can undo the last drawn shape.
18     *   User can save the image to output.txt.
19     *   User can load image data from file using the console.
20     *   Window can be resized without affecting the image.
21     *****
22  */
23
24  #include "shape.h"
25  #include "image.h"
26  #include "x11context.h"
27  #include "mydrawing.h"
28
29  using namespace std;
30  //=====Methods=====
31  /**
32   * @brief The program entry point. Assume tests are successful unless otherwise stated
33   *
34   * @param: NOT USED
35   *
36   * @retval NOT USED
37   */
38  int main(){
39      GraphicsContext *gc = new X11Context(800, 600, GraphicsContext::BLACK);
40
41      cout << "This program creates an interactive graphics window using the mouse "
42            << "to set the location of a point and keyboard to specify commands\n\n"
43            << "Pen Color Settings\n"
44            << "Key 1: Black\n"
45            << "Key 2: Gray\n"
46            << "Key 3: White\n"
47            << "Key 4: Red\n"
48            << "Key 5: Green\n"
49            << "Key 6: Blue\n"
50            << "Key 7: Cyan\n"
51            << "Key 8: Magenta\n"
52            << "Key 9: Yellow\n\n"
53            << "Graphics Commands\n"
54            << "Key P: Draw Point\n"
55            << "Key L: Draw Line\n"
56            << "Key T: Draw Triangle\n"
57            << "Key S: Save image to output.txt\n"
58            << "Key O: Load image from file using console\n"
59            << "Key Z: Undo last drawn shape" << endl;
60
61      MyDrawing md; //Make a drawing
62      gc->runLoop(&md); //Event loop that returns when window is closed
63      delete gc;
64      return 0;
65  }

```

```

1  /**
2      *****
3      * @file      : x11context.h
4      * @brief     : Outline for X11 context
5      *             : Lab 6: Event Driven Drawing
6      *             : CS-3210/021
7      * @date      : APR 27 2021
8      * @author    : unknown (provided)
9      *****
10 */
11
12 #ifndef X11_CONTEXT
13 #define X11_CONTEXT
14 /**
15  * This class is a sample implementation of the GraphicsContext class
16  * for the X11 / XWindows system.
17  */
18
19 #include <X11/Xlib.h>    // Every Xlib program must include this
20 #include "gcontext.h"    // base class
21 //=====Class=====
22 class X11Context : public GraphicsContext
23 {
24     public:
25         // Default Constructor
26         X11Context(unsigned int sizex,unsigned int sizey,unsigned int bg_color);
27
28         // Destructor
29         virtual ~X11Context();
30
31         // Drawing Operations
32         void setMode(drawMode newMode);
33         void setColor(unsigned int color);
34         void setPixel(int x, int y);
35         unsigned int getPixel(int x, int y);
36         void clear();
37         void drawLine(int x1, int y1, int x2, int y2);
38         void drawCircle(int x, int y, unsigned int radius);
39
40
41         // Event loop functions
42         void runLoop(DrawingBase* drawing);
43
44         // we will use endLoop provided by base class
45
46         // Utility functions
47         int getWindowWidth();
48         int getWindowHeight();
49
50     private:
51         // X11 stuff - specific to this context
52         Display* display;
53         Window window;
54         GC graphics_context;
55
56 };
57
58 #endif

```

```

1  /**
2  * *****
3  * @file    : x11context.h
4  * @brief   : Outline for X11 context
5  *          : Lab 6: Event Driven Drawing
6  *          : CS-3210/021
7  * @date    : APR 27 2021
8  * @author  : unknown (provided)
9  * *****
10 * @attention
11 * Provides a simple drawing context for X11/XWindows. You must have the X11 dev
12 * libraries installed. If missing, 'sudo apt-get install libx11-dev' should help.
13 * *****
14 */
15
16 #include <X11/Xlib.h> // Every Xlib program must include this
17 #include <X11/Xutil.h> // needed for XGetPixel
18 #include <X11/XKLib.h> // needed for keyboard setup
19 #include "x11context.h"
20 #include "drawbase.h"
21 #include <iostream>
22
23 /**
24 * The only constructor provided. Allows size of window and background
25 * color be specified.
26 * */
27 //=====Class=====
28 X11Context::X11Context(unsigned int sizex=400, unsigned int sizey=400,
29                        unsigned int bg_color=GraphicsContext::BLACK)
30 {
31     // Open the display
32     display = XOpenDisplay(NULL);
33
34     // Holding a key in gives repeated key_press commands but only
35     // one key_release
36     int supported;
37
38     XkbSetDetectableAutoRepeat(display, true, &supported);
39
40     // Create a window - we will assume the color map is in RGB mode.
41     window = XCreateSimpleWindow(display, DefaultRootWindow(display), 0, 0,
42                                sizex, sizey, 0, 0, bg_color);
43
44     // Sign up for MapNotify events
45     XSelectInput(display, window, StructureNotifyMask);
46
47     // Put the window on the screen
48     XMapWindow(display, window);
49
50     // Create a "Graphics Context"
51     graphics_context = XCreateGC(display, window, 0, NULL);
52
53     // Default color to white
54     XSetForeground(display, graphics_context, GraphicsContext::WHITE);
55
56     // Wait for MapNotify event
57     for(;;)
58     {
59         XEvent e;
60         XNextEvent(display, &e);
61         if (e.type == MapNotify)
62             break;
63     }
64
65     // We also want exposure, mouse, and keyboard events
66     XSelectInput(display, window, ExposureMask|
67                ButtonPressMask|
68                ButtonReleaseMask|
69                KeyPressMask|
70                KeyReleaseMask|
71                PointerMotionMask);

```

```

72
73     // We need this to get the WM_DELETE_WINDOW message from the
74     // window manager in case user click the X icon
75     Atom atomKill = XInternAtom(display, "WM_DELETE_WINDOW", False);
76     XSetWMPprotocols(display, window, &atomKill, 1);
77
78     return;
79 }
80
81 // Destructor - shut down window and connection to server
82 X11Context::~X11Context()
83 {
84     XFreeGC(display, graphics_context);
85     XDestroyWindow(display, window);
86     XCloseDisplay(display);
87 }
88
89 // Set the drawing mode - argument is enumerated
90 void X11Context::setMode(drawMode newMode)
91 {
92     if (newMode == GraphicsContext::MODE_NORMAL)
93     {
94         XSetFunction(display, graphics_context, GXcopy);
95     }
96     else
97     {
98         XSetFunction(display, graphics_context, GXxor);
99     }
100 }
101
102 // Set drawing color - assume colormap is 24 bit RGB
103 void X11Context::setColor(unsigned int color)
104 {
105     // Go ahead and set color here - better performance than setting
106     // on every setPixel
107     XSetForeground(display, graphics_context, color);
108 }
109
110 // Set a pixel in the current color
111 void X11Context::setPixel(int x, int y)
112 {
113     XDrawPoint(display, window, graphics_context, x, y);
114     XFlush(display);
115 }
116
117 unsigned int X11Context::getPixel(int x, int y)
118 {
119     XImage *image;
120     image = XGetImage (display, window, x, y, 1, 1, AllPlanes, YYPixmap);
121     XColor color;
122     color.pixel = XGetPixel (image, 0, 0);
123     XFree (image);
124     XQueryColor (display, DefaultColormap(display, DefaultScreen (display)),
125                 &color);
126     // I now have RGB values, but, they are 16 bits each, I only want 8-bits
127     // each since I want a 24-bit RGB color value
128     unsigned int pixcolor = color.red & 0xff00;
129     pixcolor |= (color.green >> 8);
130     pixcolor <= 8;
131     pixcolor |= (color.blue >> 8);
132     return pixcolor;
133 }
134
135 void X11Context::clear()
136 {
137     XClearWindow(display, window);
138     XFlush(display);
139 }
140
141
142

```

```

143 // Run event loop
144 void X11Context::runLoop(DrawingBase* drawing)
145 {
146     run = true;
147
148     while(run)
149     {
150         XEvent e;
151         XNextEvent(display, &e);
152
153         // Exposure event - lets not worry about region
154         if (e.type == Expose)
155             drawing->paint(this);
156
157         // Key Down
158         else if (e.type == KeyPress)
159             drawing->keyDown(this, XLookupKeysym((XKeyEvent*)&e,
160                 ((e.xkey.state&0x01)&&!(e.xkey.state&0x02)) ||
161                 (!(e.xkey.state&0x01)&&(e.xkey.state&0x02)))?1:0));
162
163         // Key Up
164         else if (e.type == KeyRelease){
165             drawing->keyUp(this, XLookupKeysym((XKeyEvent*)&e,
166                 ((e.xkey.state&0x01)&&!(e.xkey.state&0x02)) ||
167                 (!(e.xkey.state&0x01)&&(e.xkey.state&0x02)))?1:0));
168         }
169
170         // Mouse Button Down
171         else if (e.type == ButtonPress)
172             drawing->mouseButtonDown(this,
173                 e.xbutton.button,
174                 e.xbutton.x,
175                 e.xbutton.y);
176
177         // Mouse Button Up
178         else if (e.type == ButtonRelease)
179             drawing->mouseButtonUp(this,
180                 e.xbutton.button,
181                 e.xbutton.x,
182                 e.xbutton.y);
183
184         // Mouse Move
185         else if (e.type == MotionNotify)
186             drawing->mouseMove(this,
187                 e.xmotion.x,
188                 e.xmotion.y);
189
190         // This will respond to the WM_DELETE_WINDOW from the
191         // window manager.
192         else if (e.type == ClientMessage)
193             break;
194     }
195 }
196
197
198 int X11Context::getWindowWidth()
199 {
200     XWindowAttributes window_attributes;
201     XGetWindowAttributes(display, window, &window_attributes);
202     return window_attributes.width;
203 }
204
205 int X11Context::getWindowHeight()
206 {
207     XWindowAttributes window_attributes;
208     XGetWindowAttributes(display, window, &window_attributes);
209     return window_attributes.height;
210 }
211
212 void X11Context::drawLine(int x1, int y1, int x2, int y2)
213 {

```

```
214     XDrawLine(display, window, graphics_context, x1, y1, x2, y2);
215     XFlush(display);
216 }
217
218 void X11Context::drawCircle(int x, int y, unsigned int radius)
219 {
220     XDrawArc(display, window, graphics_context, x-radius,
221              y-radius, radius*2, radius*2, 0, 360*64);
222     XFlush(display);
223 }
```

```

1  /**
2   * *****
3   * @file      : gcontext.h
4   * @brief     : Outline for Graphics Context
5   *           : Lab 6: Event Driven Drawing
6   *           : CS-3210/021
7   * @date      : APR 27 2021
8   * @author    : unknown (provided)
9   * *****
10  * This class is intended to be the abstract base class for a graphical context
11  * for various platforms. Any concrete subclass will need to implement the pure
12  * virtual methods to support setting pixels, getting pixel color, setting the
13  * drawing mode, and running an event loop to capture mouse and keyboard events
14  * directed to the graphics context (or window). Specific expectations for the
15  * various methods are documented below.
16  * *****
17  */
18  #ifndef GCONTEXT_H
19  #define GCONTEXT_H
20
21  // forward reference - needed because runLoop needs a target for events
22  class DrawingBase;
23  //=====Class=====
24  class GraphicsContext
25  {
26  public:
27      /*****
28       * Some constants and enums
29       * *****/
30      // This enumerated type is an argument to setMode and allows
31      // us to support two different drawing modes. MODE_NORMAL is
32      // also call copy-mode and the affect pixel(s) are set to the
33      // color requested. XOR mode will XOR the new color with the
34      // existing color so that the change is reversible.
35      enum drawMode {MODE_NORMAL, MODE_XOR};
36
37      // Some colors - for fun
38      static const unsigned int BLACK = 0x000000;
39      static const unsigned int BLUE = 0x0000FF;
40      static const unsigned int GREEN = 0x00FF00;
41      static const unsigned int RED = 0xFF0000;
42      static const unsigned int CYAN = 0x00FFFF;
43      static const unsigned int MAGENTA = 0xFF00FF;
44      static const unsigned int YELLOW = 0xFFFF00;
45      static const unsigned int GRAY = 0x808080;
46      static const unsigned int WHITE = 0xFFFFFFFF;
47
48      /*****
49       * Construction / Destruction
50       * *****/
51      // Implementations of this class should include a constructor
52      // that creates the drawing canvas (window), sets a background
53      // color (which may be configurable), sets a default drawing
54      // color (which may be configurable), and start with normal
55      // (copy) drawing mode.
56
57      // need a virtual destructor to ensure subclasses will have
58      // their destructors called properly. Must be virtual.
59      virtual ~GraphicsContext();
60
61      /*****
62       * Drawing operations
63       * *****/
64
65      // Allows the drawing mode to be changed between normal (copy)
66      // and xor. The implementing context should default to normal.
67      virtual void setMode(drawMode newMode) = 0;
68
69      // Set the current color. Implementations should default to white.
70      // color is 24-bit RGB value
71

```

```

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

```



```
143  
144 #endif
```

```

1  /**
2      *****
3      * @file      : gcontext.cpp
4      * @brief     : Graphics Context
5      *           : Lab 6: Event Driven Drawing
6      *           : CS-3210/021
7      * @date      : APR 27 2021
8      * @author    : unknown (provided)
9      *****
10     * This is an abstract base class representing a generic graphics context.
11     * Most implementation specifics will need to be provided by a concrete
12     * implementation. See header file for specifics.
13     *****
14  */
15
16  #define _USE_MATH_DEFINES    // for M_PI
17  #include <cmath>             // for trig functions
18  #include "gcontext.h"
19
20  /**
21   * Destructor - does nothing
22   */
23  GraphicsContext::~GraphicsContext()
24  {
25      // nothing to do
26      // here to insure subclasses handle destruction properly
27  }
28
29  //does nothing
30  void GraphicsContext::drawLine(int x0, int y0, int x1, int y1){}
31  void GraphicsContext::drawCircle(int x0, int y0, unsigned int radius){}
32
33
34  void GraphicsContext::endLoop()
35  {
36      run = false;
37  }

```

```

1  /**
2      *****
3      * @file      : DrawBase.h
4      * @brief     : Outline for Drawing base class
5      *           : Lab 6: Event Driven Drawing
6      *           : CS-3210/021
7      * @date      : APR 27 2021
8      * @author    : Unknown (provided)
9      *****
10 */
11 #ifndef DRAWBASE_H
12 #define DRAWBASE_H
13
14 // forward reference
15 class GraphicsContext;
16 //=====Class=====
17 //Refer to mydrawing.h for documentation
18 class DrawingBase{
19     public:
20         // prevent warnings
21         virtual ~DrawingBase(){}
22         virtual void paint(GraphicsContext* gc){}
23         virtual void keyDown(GraphicsContext* gc, unsigned int keycode){}
24         virtual void keyUp(GraphicsContext* gc, unsigned int keycode){}
25         virtual void mouseButtonDown(GraphicsContext* gc,
26                                     unsigned int button, int x, int y){}
27         virtual void mouseButtonUp(GraphicsContext* gc,
28                                    unsigned int button, int x, int y){}
29         virtual void mouseMove(GraphicsContext* gc, int x, int y){}
30 };
31 #endif

```

```

1  /**
2   * *****
3   * @file    : mydrawing.h
4   * @brief   : Outline for user input drawing class, derived from drawBase
5   *         : Lab 6: Event Driven Drawing
6   *         : CS-3210/021
7   * @date    : MAY 11 2021
8   * @author  : Julian Singkham
9   * *****
10 */
11 #ifndef MYDRAWING_H
12 #define MYDRAWING_H
13 #include "drawbase.h"
14 #include "image.h"
15
16 // forward reference
17 class GraphicsContext;
18
19 //XK key values
20 enum drawing_state{
21     //Logic
22     point    = 112, //P
23     line     = 108, //L
24     triangle = 116, //T
25     save     = 115, //S
26     load     = 111, //O
27     undo     = 122, //Z
28
29     //Pen Colors
30     black    = 49, //1
31     gray     = 50, //2
32     white    = 51, //3
33     red      = 52, //4
34     green    = 53, //5
35     blue     = 54, //6
36     cyan     = 55, //7
37     magenta  = 56, //8
38     yellow   = 57, //9
39 };
40
41 //=====Class=====
42 class MyDrawing : public DrawingBase{
43     private:
44         //Shape properties
45         int x0;
46         int y0;
47         int x1;
48         int y1;
49         int x2;
50         int y2;
51         int color;
52
53         //mydrawing properties
54         bool dragging; //Flag to know if mouse is moving
55         bool can_undo; //Flag to know if undo can be used
56         Image image;   //Shapes container
57         int points_count; //Keeps track of what point is being drawn
58         drawing_state state; //Keeps track of what command is being run
59         drawing_state old_shape; //Keeps track of last drawn shape
60
61     /**
62      * @brief Saves image information to output.txt
63      *
64      * @param image: Image to save
65      *
66      * @retval NONE
67      */
68     void save_to_file(const Image image) const;
69
70     /**
71      * @brief Loads image information from file specified in console

```

```

72      *
73      * @param gc: Where to draw image
74      *
75      * @retval NONE
76      */
77      void load_file(GraphicsContext* gc);
78
79      /**
80      * @brief Undos the last drawn shape. Only works if the user isn't in
81      * the process of drawing a shape. Shapes aren't saved to image until
82      * another shape is drawn or when saving.
83      *
84      * @param gc: Where to draw image
85      *
86      * @retval NONE
87      */
88      void undo_shape(GraphicsContext* gc);
89
90      /**
91      * @brief Saves the last drawn shape to image
92      *
93      * @param: NONE
94      *
95      * @retval NONE
96      */
97      void update_image();
98
99      public:
100     /**
101     * @brief Constructor for a drawing (window). MyDrawing contains an image
102     * that contains all shapes to draw and uses a graphic context to display.
103     *
104     * @param: None
105     *
106     * @retval NONE
107     */
108     MyDrawing();
109
110     /**
111     * @brief Handles redrawing when window is resized
112     *
113     * @param gc: Where to draw image
114     *
115     * @retval NONE
116     */
117     void paint(GraphicsContext* gc);
118
119     /**
120     * @brief Handles key presses from keyboard. Not currently used.
121     *
122     * @param gc: Where to draw image
123     * @param keycode: XK value of pressed key
124     *
125     * @retval NONE
126     */
127     void keyDown(GraphicsContext *gc, unsigned int keycode);
128
129     /**
130     * @brief Handles key releases from keyboard
131     *
132     * @param gc: Where to draw image
133     * @param keycode: XK value of released key
134     *
135     * @retval NONE
136     */
137     void keyUp(GraphicsContext *gc, unsigned int keycode);
138
139     /**
140     * @brief Handles mouse presses
141     *
142     * @param gc: Where to draw image

```

```
143      * @param button: What mouse button was pressed
144      * @param x: x-coordinate of mouse cursor
145      * @param y: y-coordinate of mouse cursor
146      *
147      * @retval NONE
148      */
149      void mouseButtonDown(GraphicsContext* gc, unsigned int button, int x,
150                          int y);
151
152      /**
153      * @brief Handles mouse releases
154      *
155      * @param gc: Where to draw image
156      * @param button: What mouse button was released
157      * @param x: x-coordinate of mouse cursor
158      * @param y: y-coordinate of mouse cursor
159      *
160      * @retval NONE
161      */
162      void mouseButtonUp(GraphicsContext* gc, unsigned int button, int x,
163                        int y);
164
165      /**
166      * @brief Handles mouse movement
167      *
168      * @param gc: Where to draw image
169      * @param x: x-coordinate of mouse cursor
170      * @param y: y-coordinate of mouse cursor
171      *
172      * @retval NONE
173      */
174      void mouseMove(GraphicsContext* gc, int x, int y);
175  };
176  #endif
```

```

1  /**
2   * *****
3   * @file    : mydrawing.cpp
4   * @brief   : User driven drawing class, derived from drawbase
5   *          : Lab 6: Event Driven Drawing
6   *          : CS-3210/021
7   * @date    : MAY 11 2021
8   * @author  : Unknown (provided)
9   * *****
10  * @attention
11  * The mydrawing handles user interaction with the graphics context window.
12  * A shape is saved to mydrawing, which is then added to image once another shape
13  * is drawn or saving to file. This was done to support an undo last shape feature.
14  * *****
15  */
16 #include "mydrawing.h"
17 #include <iostream>
18 #include <stdexcept>
19 #include <fstream>
20
21 using namespace std;
22 //=====Private=====
23 /**
24  * @brief Saves image information to output.txt
25  *
26  * @param image: Image to save
27  *
28  * @retval NONE
29  */
30 void MyDrawing::save_to_file(const Image image) const{
31     ofstream file;
32     file.open("output.txt", ofstream::trunc);
33     image.out(file);
34     file.close();
35     cout << "Saving to output.txt" << endl;
36 }
37 /**
38  * @brief Loads image information from file specified in console
39  *
40  * @param gc: Where to draw image
41  *
42  * @retval NONE
43  */
44 void MyDrawing::load_file(GraphicsContext* gc){
45     cout << "Enter name of .txt file containing image information" << endl;
46     string file_name;
47     cin >> file_name;
48     ifstream file(file_name);
49     if (!file.is_open())
50         cout << "Error: unable to open " << file_name << endl;
51     else{
52         image.in(file);
53         cout << "Loaded " << file_name << endl;
54         gc -> clear();
55         image.draw(gc);
56         file.close();
57     }
58 }
59
60 /**
61  * @brief Undos the last drawn shape. Only works if the user isn't in
62  * the process of drawing a shape. Shapes aren't saved to image until
63  * another shape is drawn or when saving.
64  *
65  * @param gc: Where to draw image
66  *
67  * @retval NONE
68  */
69 void MyDrawing::undo_shape(GraphicsContext* gc){
70     switch (old_shape){
71         case point:

```

```

72         gc->setMode(GraphicsContext::MODE_XOR);
73         gc->drawLine(x0, y0, x0, y0);
74         gc->setMode(GraphicsContext::MODE_NORMAL);
75         break;
76     case line:
77         gc->setMode(GraphicsContext::MODE_XOR);
78         gc->drawLine(x0, y0, x1, y1);
79         gc->setMode(GraphicsContext::MODE_NORMAL);
80         break;
81     case triangle:
82         gc->setMode(GraphicsContext::MODE_XOR);
83         gc->drawLine(x0, y0, x1, y1);
84         gc->drawLine(x1, y1, x2, y2);
85         gc->drawLine(x2, y2, x0, y0);
86         //Triangles create points on vertices on window
87         //This only exists in the window, image is not affected
88         gc->drawLine(x0, y0, x0, y0);
89         gc->drawLine(x1, y1, x1, y1);
90         gc->drawLine(x2, y2, x2, y2);
91         gc->setMode(GraphicsContext::MODE_NORMAL);
92         break;
93     default:
94         break;
95     }
96     can_undo = false;
97     gc->clear();
98     image.draw(gc);
99     cout << "Undo" << endl;
100 }
101 /**
102  * @brief Saves the last drawn shape to image
103  *
104  * @param: NONE
105  *
106  * @retval NONE
107  */
108 void MyDrawing::update_image(){
109     switch (old_shape){
110     case point:
111         image.add(new Line(x0, y0, x0, y0, color));
112         break;
113     case line:
114         image.add(new Line(x0, y0, x1, y1, color));
115         break;
116     case triangle:
117         image.add(new Triangle(x0, y0, x1, y1, x2, y2, color));
118         break;
119     default:
120         break;
121     }
122     can_undo = false;
123 }
124 //=====Public=====

125 /**
126  * @brief Constructor for a drawing (window). MyDrawing contains an image
127  * that contains all shapes to draw and uses a graphic context to display.
128  *
129  * @param: None
130  *
131  * @retval NONE
132  */
133 MyDrawing::MyDrawing(){
134     dragging = false;
135     can_undo = false;
136     x0 = x1 = x2 = y0 = y1 = y2 = 0;
137     points_count = 0;
138     color = GraphicsContext::WHITE; //Default pen color
139     image = Image();
140     return;
141 }

```



```

142
143 /**
144  * @brief Handles redrawing when window is resized
145  *
146  * @param gc: Where to draw image
147  *
148  * @retval NONE
149  */
150 void MyDrawing::paint(GraphicsContext* gc){
151     // it will only show up after an exposure
152     image.draw(gc);
153     //Since the last drawn isn't saved to image, we have to redraw it to
154     //preserve the undo functionality
155     if(can_undo){
156         switch (old_shape){
157             case point:
158                 gc->setMode(GraphicsContext::MODE_NORMAL);
159                 gc->drawLine(x0, y0, x0, y0); //draw line in copy mode
160                 break;
161             case line:
162                 gc->setMode(GraphicsContext::MODE_NORMAL);
163                 gc->drawLine(x0, y0, x1, y1); //draw line in copy mode
164                 break;
165             case triangle:
166                 gc->setMode(GraphicsContext::MODE_NORMAL);
167                 gc->drawLine(x0, y0, x1, y1); //draw line in copy mode
168                 gc->drawLine(x1, y1, x2, y2);
169                 gc->drawLine(x2, y2, x0, y0);
170                 break;
171             default:
172                 break;
173         }
174     }
175     return;
176 }
177
178 /**
179  * @brief Handles key presses from keyboard. Not currently used.
180  *
181  * @param gc: Where to draw image
182  * @param keycode: XK value of pressed key
183  *
184  * @retval NONE
185  */
186 void MyDrawing::keyDown(GraphicsContext *gc, unsigned int keycode){
187     return;
188 }
189
190 /**
191  * @brief Handles key releases from keyboard
192  *
193  * @param gc: Where to draw image
194  * @param keycode: XK value of released key
195  *
196  * @retval NONE
197  */
198 void MyDrawing::keyUp(GraphicsContext *gc, unsigned int keycode){
199     switch (keycode){
200         case point:
201             state = point;
202             cout << "POINT" << endl;
203             break;
204         case line:
205             state = line;
206             cout << "LINE" << endl;
207             break;
208         case triangle:
209             state = triangle;
210             points_count = 0;
211             cout << "TRIANGLE" << endl;
212             break;

```

```

213     case save:
214         if (can_undo)
215             update_image();
216             save_to_file(image);
217         break;
218     case load:
219         load_file(gc);
220         break;
221     case undo:
222         if (can_undo)
223             undo_shape(gc);
224         else
225             cout << "Unable to undo: Either the last shape was already "
226                  << "removed or no images have been added" << endl;
227         break;
228     case black:
229         color = GraphicsContext::BLACK;
230         cout << "BLACK" << endl;
231         break;
232     case gray:
233         color = GraphicsContext::GRAY;
234         cout << "GRAY" << endl;
235         break;
236     case white:
237         color = GraphicsContext::WHITE;
238         cout << "WHITE" << endl;
239         break;
240     case red:
241         color = GraphicsContext::RED;
242         cout << "RED" << endl;
243         break;
244     case green:
245         color = GraphicsContext::GREEN;
246         cout << "GREEN" << endl;
247         break;
248     case blue:
249         color = GraphicsContext::BLUE;
250         cout << "BLUE" << endl;
251         break;
252     case cyan:
253         color = GraphicsContext::CYAN;
254         cout << "CYAN" << endl;
255         break;
256     case magenta:
257         color = GraphicsContext::MAGENTA;
258         cout << "MAGENTA" << endl;
259         break;
260     case yellow:
261         color = GraphicsContext::YELLOW;
262         cout << "YELLOW" << endl;
263         break;
264     default:
265         cout << "Key down: " << keycode << endl;
266         break;
267     }
268     gc->setColor(color);
269 }
270
271 /**
272  * @brief Handles mouse presses
273  *
274  * @param gc: Where to draw image
275  * @param button: What mouse button was pressed
276  * @param x: x-coordinate of mouse cursor
277  * @param y: y-coordinate of mouse cursor
278  *
279  * @retval NONE
280  */
281 void MyDrawing::mouseButtonDown(GraphicsContext* gc, unsigned int button, int x,
282                                 int y) {
283     //Undow will only be available if the user isn't drawing a shape

```

```

284     if(can_undo)
285         update_image();
286
287     //gc MODE_NORMAL means that shapes will be drawn on top of the frame
288     //gc MODE_XOR means that shape collision will negate each other
289     switch (state){
290         case point:
291             x0 = x;
292             y0 = y;
293             gc->setMode(GraphicsContext::MODE_NORMAL);
294             gc->drawLine(x, y, x, y);
295             old_shape = point;
296             can_undo = true;
297             break;
298         case line:
299             if (points_count == 0){
300                 points_count = 1;
301                 x0 = x1 = x;
302                 y0 = y1 = y;
303                 gc->setMode(GraphicsContext::MODE_XOR);
304                 dragging = true;
305             }
306             else{
307                 dragging = false;
308                 gc->drawLine(x0, y0, x1, y1); //Delete old line
309                 x1 = x;
310                 y1 = y;
311                 gc->setMode(GraphicsContext::MODE_NORMAL);
312                 gc->drawLine(x0, y0, x1, y1);
313
314                 //Set Flags
315                 points_count = 0;
316                 old_shape = line;
317                 can_undo = true;
318             }
319             break;
320         case triangle:
321             if (points_count == 0){
322                 points_count = 1;
323                 x0 = x1 = x2 = x;
324                 y0 = y1 = y2 = y;
325                 gc->setMode(GraphicsContext::MODE_XOR);
326                 dragging = true;
327             }
328             else if (points_count == 1){
329                 dragging = false;
330                 points_count = 2;
331                 gc->drawLine(x0, y0, x1, y1); //Delete old line
332                 x1 = x;
333                 y1 = y;
334                 gc->setMode(GraphicsContext::MODE_NORMAL);
335                 gc->drawLine(x0, y0, x1, y1);
336                 gc->setMode(GraphicsContext::MODE_XOR);
337
338                 //Set Flags
339                 dragging = true;
340             }
341             else{
342                 dragging = false;
343                 points_count = 0;
344                 gc->drawLine(x1, y1, x2, y2); //Delete old line
345                 gc->drawLine(x2, y2, x0, y0); //Delete old line
346                 x2 = x;
347                 y2 = y;
348                 gc->setMode(GraphicsContext::MODE_NORMAL);
349                 gc->drawLine(x1, y1, x2, y2);
350                 gc->drawLine(x2, y2, x0, y0);
351
352                 //Set Flags
353                 old_shape = triangle;
354                 can_undo = true;

```

```

355         }
356         break;
357     default:
358         break;
359     }
360     return;
361 }
362
363 /**
364  * @brief Handles mouse releases. Not currently used.
365  *
366  * @param gc: Where to draw image
367  * @param button: What mouse button was released
368  * @param x: x-coordinate of mouse cursor
369  * @param y: y-coordinate of mouse cursor
370  *
371  * @retval NONE
372  */
373 void MyDrawing::mouseButtonUp(GraphicsContext* gc, unsigned int button, int x,
374                               int y){
375     return;
376 }
377
378 /**
379  * @brief Handles mouse movement
380  * Assume GC in MODE_XOR prior to function call
381  *
382  * @param gc: Where to draw image
383  * @param x: x-coordinate of mouse cursor
384  * @param y: y-coordinate of mouse cursor
385  *
386  * @retval NONE
387  */
388 void MyDrawing::mouseMove(GraphicsContext* gc, int x, int y){
389     switch (state){
390     case line:
391         if (dragging){
392             gc->drawLine(x0, y0, x1, y1); //Delete old line
393             //Update
394             x1 = x;
395             y1 = y;
396             gc->drawLine(x0, y0, x1, y1); //Draw new line
397         }
398         break;
399     case triangle:
400         if (dragging){
401             if(points_count == 1){
402                 gc->drawLine(x0, y0, x1, y1);
403                 x1 = x;
404                 y1 = y;
405                 gc->drawLine(x0, y0, x1, y1);
406             }
407             else if(points_count == 2){
408                 //Since all points are inialized to the origin, we have to
409                 //update the last vertex before drawing it or else the
410                 //first line will be missing in the window
411                 if((x2 != x0) || (y2 != y0)){
412                     gc->drawLine(x1, y1, x2, y2);
413                     gc->drawLine(x2, y2, x0, y0);
414                 }
415                 x2 = x;
416                 y2 = y;
417                 gc->drawLine(x1, y1, x2, y2);
418                 gc->drawLine(x2, y2, x0, y0);
419             }
420         }
421         break;
422     default:
423         break;
424     }
425 }

```

```
426     return;  
427 }
```

```

1  /**
2  ****
3  * @file      : matrix.h
4  * @brief     : Outline for matrix
5  *           : Lab 6: Event Driven Drawing
6  *           : CS-3210/021
7  * @date      : MAR 30 2021
8  * @author    : Dr. Darrin Rothe
9  * @author    : Julian Singkham (modified)
10 ****
11 **/
12
13 #ifndef MATRIX_H
14 #define MATRIX_H
15
16 #include <iostream>      // for std::ostream
17 #include <stdexcept>    // for std::runtime_error
18 #include <string>       // used in matrix_Exception
19
20 /**
21 * @brief Custom exception used in the matrix class.
22 */
23 class matrix_Exception:public std::runtime_error{
24 public:
25     matrix_Exception(std::string message):
26         std::runtime_error((std::string("A Matrix Error has occured: ") +
27             message).c_str()) {}
28 };
29 //=====Class=====
30 class matrix
31 {
32 public:
33     /**
34     * @brief Makes the insertion operator a friend so it can access matrix.
35     * Basically allows the matrix to be printed to std.
36     *
37     * @param os: Stream to write to.
38     * @param rhs: Reference to the matrix that is being printed.
39     *
40     * @retval Stream containing the matrix printout.
41     */
42     friend std::ostream& operator<<(std::ostream& os, const matrix& rhs);
43
44     /**
45     * @brief Parameterized constructor, it creates a matrix of given dimensions
46     * with clear (zeroed) cells.
47     *
48     * @param rows: How many rows in the matrix.
49     * @param cols: How many columns in the matrix.
50     *
51     * @retval NONE
52     */
53     matrix(unsigned int rows, unsigned int cols);
54
55     /**
56     * @brief Copy constructor that makes a new matrix from a given one.
57     *
58     * @param from: matrix to copy into the new matrix.
59     *
60     * @retval A copy of the given matrix.
61     */
62     matrix(const matrix& from);
63
64     /**
65     * @brief Frees allocated memory from the matrix.
66     *
67     * @param: NONE
68     *
69     * @retval NONE
70     */
71     ~matrix();

```

```

72
73  /**
74  * @brief Named constructor, it creates an identity matrix of given size.
75  *
76  * Identity matrix is a matrix that is all zeros except when
77  * row#=col#, then it is 1.
78  * // 1 0 0 [0][0] = 1
79  * // 0 1 0 [1][1] = 1
80  * // 0 0 1 [2][2] = 1
81  *
82  * @param size: Square dimensions of the matrix.
83  *
84  * @retval The identity matrix.
85  */
86  static matrix identity(unsigned int size);
87
88  /**
89  * @brief Assigns the matrix to the value stored in the given matrix.
90  *
91  * @param rhs: The given matrix to copy from.
92  *
93  * @retval A copy of the given matrix.
94  */
95  matrix& operator=(const matrix& rhs);
96
97  /**
98  * @brief Matrix addition. The lhs and rhs must be the same size.
99  *
100  * @param rhs: The right hand side matrix.
101  *
102  * @retval The resulting matrix after addition.
103  */
104  matrix operator+(const matrix& rhs) const;
105
106  /**
107  * @brief Matrix multiplication.
108  * The lhs column size and rhs row size must match.
109  *
110  * @param rhs: The right hand side matrix.
111  *
112  * @retval The resulting matrix after multiplication.
113  * Dimension: lhs.rows x rhs.cols.
114  */
115  matrix operator*(const matrix& rhs) const;
116
117  /**
118  * @brief Matrix scalar multiplication.
119  * This only supports matrix * 5, not 5 * matrix.
120  *
121  * @param scale: Value to scale the matrix.
122  *
123  * @retval The scaled matrix.
124  */
125  matrix operator*(const double scale) const;
126
127  /**
128  * @brief This allows access of the matrix elements by using [].
129  *
130  * @param row: The desired row of the matrix.
131  *
132  * @retval A pointer to the desired element of the matrix.
133  */
134  double* operator[](unsigned int row);
135
136  /**
137  * @brief This allows access of the matrix elements by using [].
138  * Const version
139  *
140  * @param row: The desired row of the matrix.
141  *
142  */

```

```
143      * @retval A pointer to the desired element of the matrix.
144      */
145      double* operator[](unsigned int row) const;
146
147      /**
148      * @brief Zeroes the elements of the matrix.
149      *
150      * @param: None
151      *
152      * @retval None
153      */
154      void clear();
155 private:
156      //The data
157      double** the_matrix;
158      unsigned int rows;
159      unsigned int cols;
160 };
161 //=====Global=====
162 /**
163  * @brief Matrix scaler multiplication.
164  * This only supports matrix * 5, not 5 * matrix.
165  *
166  * @param scale: Value to scale the matrix.
167  * @param rhs: The matrix to apply the scaling.
168  *
169  * @retval The scaled matrix.
170  */
171 matrix operator*(const double scale, const matrix& rhs);
172
173 #endif
```



```

1  /**
2  ****
3  * @file      : matrixm.cpp
4  * @brief     : Matrix
5  *           : Lab 6: Event Driven Drawing
6  *           : CS-3210/021
7  * @date      : MAR 23 2021
8  * @author    : Julian Singkham
9  ****
10 * @attention
11 * This API handles the creation, deletion, and =,+,* operators of a matrix as well
12 * as retrieving matrix values using []. In essence the double matrix used in this
13 * API is an array of arrays where the **double matrix points to rows *double arrays
14 * that then point to col elements.
15 ****
16 */
17 #include "matrix.h"
18 #include <string>
19 #include <cmath>
20
21 //=====Public=====
22 /**
23 * @brief Makes the insertion operator a friend so it can access matrix.
24 * Basically allows the matrix to be printed to std.
25 *
26 * @param os: Stream to write to.
27 * @param rhs: Reference to the matrix that is being printed.
28 *
29 * @retval Stream containing the matrix printout.
30 */
31 std::ostream& operator<<(std::ostream& os, const matrix& rhs){
32     for(int i = 0; i < rhs.rows; i++){
33         os << "|";
34         for(int j = 0; j < rhs.cols; j++){
35             double temp = rhs.the_matrix[i][j];
36             os << temp << "|";
37         }
38         os << std::endl;
39     }
40     return os;
41 }
42
43 /**
44 * @brief Matrix constructor, it creates a matrix of given dimensions
45 * with clear (zeroed) cells. Throws error if dimensions are not possible (<1).
46 *
47 * @param rows: How many rows in the matrix.
48 * @param cols: How many columns in the matrix.
49 *
50 * @retval NONE
51 */
52 matrix::matrix(unsigned int rows, unsigned int cols) : rows(rows), cols(cols){
53     if(rows < 1 || cols < 1)
54         throw matrix_Exception("p-constructor has bad arguments.");
55
56     the_matrix = new double*[rows]; //Allocates memory to the # of rows
57
58     //Allocate memory for each row of the array to the # of columns
59     //Basically creates a 1-D array of 1-D arrays
60     for(int i = 0; i < rows; i++)
61         the_matrix[i] = new double[cols];
62
63     clear(); //Fill matrix with zeroes
64 }
65
66 /**
67 * @brief Copy constructor that makes a new matrix from a given one.
68 *
69 * @param from: matrix to copy into the new matrix.
70 *
71 * @retval A copy of the given matrix.

```

```

72  */
73  matrix::matrix(const matrix& from) : rows(from.rows), cols(from.cols){
74      the_matrix = new double*[rows]; //Allocates memory to the # of rows
75
76      //Allocate memory for each row of the array to the # of columns
77      //Basically creates a 1-D array of 1-D arrays
78      for(int i = 0; i < rows; i++)
79          the_matrix[i] = new double[cols];
80
81      //Copy values from "from" into new matrix
82      for(int i = 0; i < rows; i++)
83          for(int j = 0; j < cols; j++)
84              the_matrix[i][j] = from[i][j];
85  }
86
87  /**
88   * @brief Frees allocated memory form matrix
89   *
90   * @param: NONE
91   *
92   * @retval NONE
93   */
94  matrix::~matrix(){
95      for(int i = 0; i < rows; i++)
96          delete[] the_matrix[i]; //Delete each row of the matrix
97      delete[] the_matrix; //Delete the matrix itself
98  }
99
100  /**
101   * @brief Named constructor, it creates a square identity matrix
102   * of given size.
103   *
104   * Identity matrix is a matrix that is all zeros expect when
105   * row#=col#, then it is 1.
106   * // 1 0 0 [0][0] = 1
107   * // 0 1 0 [1][1] = 1
108   * // 0 0 1 [2][2] = 1
109   *
110   * @param size: Square dimensions of the matrix.
111   *
112   * @retval The square identity matrix.
113   */
114  matrix matrix::identity(unsigned int size){
115      if(size == 0)
116          throw matrix_Exception("Can not create an identity matrix of size 0.");
117
118      matrix return_matrix(size, size);
119      for(int i = 0; i < size; i++)
120          return_matrix[i][i] = 1.0;
121      return return_matrix;
122  }
123
124  /**
125   * @brief Assigns the matrix to the value stored in the given matrix.
126   *
127   * @param rhs: The given matrix to copy from.
128   *
129   * @retval A copy of the given matrix.
130   */
131  matrix& matrix::operator=(const matrix& rhs){
132      //Verify matrices match in size
133      if(rows != rhs.rows || cols != rhs.cols){
134          for(int i = 0; i < rows; i++)
135              delete[] the_matrix[i]; //Delete each row of the matrix
136          delete[] the_matrix; //Delete the matrix itself
137
138          rows = rhs.rows;
139          cols = rhs.cols;
140          the_matrix = new double*[rows]; //Allocates memory to the # of rows
141
142          //Allocate memory for each row of the array to the # of columns

```

```

143         //Basically creates a 1-D array of 1-D arrays
144         for(int i = 0; i < rows; i++)
145             the_matrix[i] = new double[cols];
146     }
147     //Copy values from rhs into current matrix
148     for(int i = 0; i < rows; i++)
149         for(int j = 0; j < cols; j++)
150             the_matrix[i][j] = rhs[i][j];
151
152     return *this;
153 }
154
155 /**
156  * @brief Matrix addition. The lhs and rhs must be the same size.
157  *
158  * @param rhs: The right hand side matrix.
159  *
160  * @retval The resulting matrix after addition.
161  */
162 matrix matrix::operator+(const matrix& rhs) const{
163     //Verify matrices match in size
164     if(rows != rhs.rows || cols != rhs.cols)
165         throw matrix_Exception("Size mismatch – The column/row of the left matrix "
166                                "does not match the column/row of the right matrix:");
167
168     matrix return_matrix(rows, cols);
169
170     for(int i = 0; i < rows; i++)
171         for(int j = 0; j < cols; j++)
172             return_matrix[i][j] = the_matrix[i][j] + rhs[i][j];
173
174     return return_matrix;
175 }
176
177 /**
178  * @brief Matrix multiplication.
179  * The lhs column size and rhs row size must match.
180  *
181  * @param rhs: The right hand side matrix.
182  *
183  * @retval The resulting matrix after multiplication.
184  * Dimension: lhs.rows x rhs.cols.
185  */
186 matrix matrix::operator*(const matrix& rhs) const{
187     //Verify matrices match in size
188     if(cols != rhs.rows)
189         throw matrix_Exception("Size mismatch – The column of the left matrix does "
190                                "not match the row of the right matrix:");
191
192     matrix return_matrix(rows, rhs.cols);
193
194     for (int i = 0; i < rows; ++i)
195         for (int j = 0; j < rhs.cols; ++j)
196             for (int k = 0; k < rhs.cols; ++k)
197                 return_matrix[i][j] += the_matrix[i][k] * rhs[k][j];
198     return return_matrix;
199 }
200
201 /**
202  * @brief Matrix scalar multiplication.
203  * This only supports matrix * 5, not 5 * matrix.
204  *
205  * @param scale: Value to scale the matrix.
206  *
207  * @retval The scaled matrix.
208  */
209 matrix matrix::operator*(const double scale) const{
210     matrix return_matrix(rows, cols);
211
212     for (int i = 0; i < rows; ++i)
213         for (int j = 0; j < cols; ++j)

```

```

214         return_matrix[i][j] = the_matrix[i][j] * scale;
215     return return_matrix;
216 }
217
218 /**
219  * @brief This allows access of the matrix elements by using [].
220  *
221  * @param row: The desired row of the matrix.
222  *
223  * @retval A pointer to the desired element of the matrix.
224  */
225 double* matrix::operator[](unsigned int row){
226     //Verify row is within bounds
227     if (row >= rows || row < 0)
228         throw matrix_Exception("Size mismatch – The requested row is"
229                                " out of bounds.");
230     double *ret = the_matrix[row];
231     return ret;
232 }
233
234 /**
235  * @brief This allows access of the matrix elements by using [].
236  * Const version
237  *
238  * @param row: The desired row of the matrix.
239  *
240  * @retval A pointer to the desired element of the matrix.
241  */
242 double* matrix::operator[](unsigned int row) const{
243     //Verify row is within bounds
244     if (row >= rows || row < 0)
245         throw matrix_Exception("Size mismatch – The requested row is"
246                                " out of bounds.");
247     double *ret = the_matrix[row];
248     return ret;
249 }
250
251 /**
252  * @brief Zeroes the elements of the matrix.
253  *
254  * @param: None
255  *
256  * @retval None
257  */
258 void matrix::clear(){
259     for(int i = 0; i < rows; i++)
260         for(int j = 0; j < cols; j++)
261             the_matrix[i][j] = 0.0;
262     return;
263 }
264
265 //=====Global=====
266 /**
267  * @brief Matrix scaler multiplication.
268  * This only supports matrix * 5, not 5 * matrix.
269  *
270  * @param scale: Value to scale the matrix.
271  * @param rhs: The matrix to apply the scaling.
272  *
273  * @retval The scaled matrix.
274  */
275 matrix operator*(const double scale, const matrix& rhs){
276     matrix return_matrix(rhs);
277     return_matrix = rhs * scale;
278     return return_matrix;
279 }

```

```

1  /**
2  ****
3  * @file    : shape.h
4  * @brief   : Outline for shape base class
5  *          : Lab 6: Event Driven Drawing
6  *          : CS-3210/021
7  * @date    : APR 27 2021
8  * @author  : Julian Singkham
9  ****
10 /**/
11 #ifndef SHAPE_H
12 #define SHAPE_H
13
14 #include "matrix.h"
15 #include "gcontext.h"
16
17 //=====Base Class=====
18 class Shape{
19     protected:
20         int color;
21         matrix point1;
22
23         /**
24          * @brief Assigns properties from the given shape to this shape
25          *        Made protected so that the children of shape can't be set to
26          *        eachother. A triangle should not converted into a line.
27          *
28          * @param rhs: The given shape to copy from
29          *
30          * @retval A copy of the given shape
31          */
32         virtual Shape &operator=(const Shape &rhs);
33
34     public:
35
36         /**
37          * @brief Read shape properties from a text file (stream)
38          *
39          * @param is: Stream to read from
40          *
41          * @retval NONE
42          */
43         virtual std::istream &in(std::istream &is);
44
45         /**
46          * @brief Parameterized constructor, it creates a shape with a color.
47          *
48          * @param color_red: 3x8-bit value for red, green, blue
49          *
50          * @retval NONE
51          */
52         Shape(int color);
53
54         /**
55          * @brief Copy constructor that copies the paramters from the given shape
56          *
57          * @param from: shape to copy into the current shape.
58          *
59          * @retval NONE
60          */
61         Shape(const Shape &from);
62
63         /**
64          * @brief Virtual constructor thats used to copy a shape
65          *
66          * @param: NONE
67          *
68          * @retval NONE
69          */
70         virtual Shape *clone() = 0;
71

```

```
72     /**
73      * @brief Shape destructor, frees memory allocated to shape
74      *      Not currently used due to image handling deletion
75      *
76      * @param: NONE
77      *
78      * @retval NONE
79      */
80     virtual ~Shape();
81
82     /**
83      * @brief Draws the given shape
84      *
85      * @param gc: GraphicsContext object that tells the shape where to draw
86      *
87      * @retval NONE
88      */
89     virtual void draw(GraphicsContext *gc) = 0;
90
91     /**
92      * @brief Print contents of shape into std.
93      *      Method made const to prevent modifying when outputting
94      *
95      *      Shape_type
96      *      Color: 0x.....
97      *      Point?: x y z
98      *
99      * @param os: Stream to write to
100     *
101     * @retval NONE
102     */
103     virtual std::ostream &out(std::ostream &os) const;
104 };
105
106 #endif
```

```

1  /**
2   * *****
3   * @file    : shape.cpp
4   * @brief   : Shape base class
5   *         : Lab 6: Event Driven Drawing
6   *         : CS-3210/021
7   * @date    : APR 27 2021
8   * @author  : Julian Singkham
9   * *****
10  * @attention
11  * Abstract base class for all types of shapes (currently line, triangle).
12  * Shape houses the color and origin point for all its children since all shapes.
13  * Shape functions are only ever called on when a child needs to modify/get
14  * color or point1.
15  * *****
16  */
17 #include <sstream> //For String Stream
18
19 #include "shape.h"
20
21 //=====Protected=====
22 /**
23  * @brief Assigns properties from the given shape to this shape
24  *         Made protected so that the children of shape can't be set to
25  *         eachother. A triangle should not converted into a line.
26  *
27  * @param rhs: The given shape to copy from
28  *
29  * @retval A copy of the given shape
30  */
31 Shape &Shape::operator=(const Shape &rhs){
32     //check if shape is being assigned it itself
33     if(this != &rhs){
34         color = rhs.color;
35         point1 = matrix(rhs.point1);
36     }
37     return *this;
38 }
39 //=====Public=====
40 /**
41  * @brief Read line properties from a text file (stream)
42  *
43  * @param is: Stream to read from
44  *
45  * @retval NONE
46  */
47 std::istream &Shape::in(std::istream &is){
48     std::string Line;
49     std::stringstream str_stream;
50
51     //Copy Color
52     std::getline(is, Line); //Read line
53     str_stream = std::stringstream(Line);
54     str_stream.ignore(32, ':');
55     str_stream >> std::hex >> color;
56
57     //Copy first point
58     std::getline(is, Line); //Read line
59     str_stream = std::stringstream(Line);
60     str_stream.ignore(32, ':');
61     str_stream >> point1[0][0];
62     str_stream >> point1[1][0];
63     str_stream >> point1[2][0];
64
65     return is;
66 }
67 /**
68  * @brief Parameterized constructor, it creates a shape with a color.
69  *
70  * @param color_red: 3x8-bit value for red, green, blue
71  *

```

```

72  * @retval NONE
73  */
74  Shape::Shape(int color)
75      : color(color), point1(5,5){
76  }
77
78  /**
79   * @brief Copy constructor that copies the paramters from the given shape
80   *
81   * @param from: shape to copy into the current shape.
82   *
83   * @retval NONE
84   */
85  Shape::Shape(const Shape &from)
86      : color(from.color), point1(from.point1){
87  }
88
89  /**
90   * @brief Line destructor, frees memory allocated to line
91   *         Not currently used due to image handling deletion
92   *
93   * @param: NONE
94   *
95   * @retval NONE
96   */
97  Shape::~Shape() {
98  }
99
100 /**
101  * @brief Print contents of line into std.
102  *         Method made const to prevent modifying when outputting
103  *
104  *         Shape_type
105  *         Color: 0x.....
106  *         Point1: x y z
107  *
108  * @param os: Stream to write to
109  *
110  * @retval NONE
111  */
112  std::ostream &Shape::out(std::ostream &os) const{
113      os << "\tColor: 0x" << std::uppercase << std::hex << color << std::endl;
114
115      os << "\tPoint 1: "
116          << point1[0][0] << " "
117          << point1[1][0] << " "
118          << point1[2][0]
119          << std::endl;
120
121      return os;
122  }

```



```

1  /**
2  ****
3  * @file    : line.h
4  * @brief   : Outline for line shape class
5  *          : Lab 6: Event Driven Drawing
6  *          : CS-3210/021
7  * @date    : APR 27 2021
8  * @author  : Julian Singkham
9  ****
10 */
11 #ifndef LINE_H
12 #define LINE_H
13
14 #include "shape.h"
15
16 //=====Class=====
17
18 class Line : public Shape{
19     private:
20         //points to draw to
21         matrix point2;
22
23         /**
24          * @brief Constructor that makes a new line from a stream
25          *       Made private so that only image can create triangles with a stream.
26          *       Image will handle parsing through the file and determining what
27          *       shape gets created.
28          * @param is: Input stream that contains Line parameters
29          * @retval NONE
30          */
31         Line(std::istream &is);
32
33         /**
34          * @brief Read line properties from a text file (stream)
35          *
36          * @param is: Stream to read from
37          *
38          * @retval NONE
39          */
40         std::istream &in(std::istream &is);
41
42     public:
43         friend class Image; //Allows image access to the instream methods
44
45         /**
46          * @brief Parameterized constructor, it creates a Line with a color.
47          *
48          * @param color: 3x8-bit value for red, green, blue
49          *
50          * @retval NONE
51          */
52         Line(double x0, double y0, double x1, double y1, int color);
53
54         /**
55          * @brief Copy constructor that copies the paramters from the given line
56          *
57          * @param from: Line to copy into the current line.
58          *
59          * @retval None
60          */
61         Line(const Line &from);
62
63         /**
64          * @brief Virtual constructor thats used to copy a shape
65          *
66          * @param: NONE
67          *
68          * @retval NONE
69          */
70         Line *clone();

```

```
71
72     /**
73      * @brief Line destructor, frees memory allocated to line
74      *      Not currently used due to image handling deletion
75      *
76      * @param: NONE
77      *
78      * @retval NONE
79      */
80     ~Line();
81
82     /**
83      * @brief Assigns properties from the given line to this line
84      *
85      * @param rhs: The given line to copy from
86      *
87      * @retval A copy of the given line
88      */
89     Line &operator=(const Line &rhs);
90
91     /**
92      * @brief Draws the given line
93      *
94      * @param gc: GraphicsContext object that tells the shape where to draw
95      *
96      * @retval NONE
97      */
98     void draw(GraphicsContext *gc);
99
100    /**
101     * @brief Print contents of line into std.
102     *      Method made const to prevent modifying when outputting
103     *
104     *      Shape_type
105     *      Color: 0x.....
106     *      Point?: x y z
107     *
108     * @param os: Stream to write to
109     *
110     * @retval NONE
111     */
112    std::ostream &out(std::ostream &os) const;
113 };
114
115
116 #endif
```

```

1  /**
2   * *****
3   * @file    : line.cpp
4   * @brief   : line shape class
5   *         : Lab 6: Event Driven Drawing
6   *         : CS-3210/021
7   * @date    : APR 27 2021
8   * @author  : Julian Singkham
9   * *****
10  * @attention
11  * Handles the creation of a line in 3-D space using x11 graphics.
12  * *****
13  */
14  #include <sstream> //For String Stream
15
16  #include "line.h"
17  //=====Private=====
18  /**
19   * @brief Constructor that makes a new line from a stream
20   *         Made private so that only image can create triangles with a stream.
21   *         Image will handle parsing through the file and determining what
22   *         shape gets created.
23   * @param is: Input stream that contains Line parameters
24   *
25   * @retval NONE
26   */
27  Line::Line(std::istream &is)
28      : Shape(color), point2(5,5){
29
30      in(is);
31  }
32
33  /**
34   * @brief Read line properties from a text file (stream)
35   *
36   * @param is: Stream to read from
37   *
38   * @retval NONE
39   */
40  std::istream &Line::in(std::istream &is){
41      std::string str_line;
42      std::stringstream str_stream;
43
44      Shape::in(is); //Call parent first
45
46      //Copy second point
47      std::getline(is, str_line); //Read line
48      str_stream = std::stringstream(str_line);
49      str_stream.ignore(32, ':');
50      str_stream >> point2[0][0];
51      str_stream >> point2[1][0];
52      str_stream >> point2[2][0];
53
54      return is;
55  }
56
57  //=====Public=====
58
59  /**
60   * @brief Parameterized constructor, it creates a Line with a color.
61   *
62   * @param color: 3x8-bit value for red, green, blue
63   *
64   * @retval NONE
65   */
66  Line::Line(double x0, double y0, double x1, double y1, int color)
67      : Shape(color), point2(5,5){
68
69      //Copy origin point
70      this->point1[0][0] = x0;
71      this->point1[1][0] = y0;

```

```

71     this->point1[2][0] = 0; //Default
72     this->point1[3][0] = 1; //Default
73
74     //Copy second point
75     this->point2[0][0] = x1;
76     this->point2[1][0] = y1;
77     this->point2[2][0] = 0; //Default
78     this->point2[3][0] = 1; //Default
79 }
80
81 /**
82  * @brief Copy constructor that copies the paramters from the given line
83  *
84  * @param from: Line to copy into the current line.
85  *
86  * @retval None
87  */
88 Line::Line(const Line &from)
89     : Shape(from.color), point2(from.point2){
90
91     point1 = matrix(from.point1);
92 }
93
94 /**
95  * @brief Virtual constructor thats used to copy a shape
96  *
97  * @param: NONE
98  *
99  * @retval NONE
100 */
101 Line *Line::clone(){
102     return new Line(*this);
103 }
104
105 /**
106  * @brief Line destructor, frees memory allocated to line
107  *         Not currently used due to image handling deletion
108  *
109  * @param: NONE
110  *
111  * @retval NONE
112 */
113 Line::~Line(){
114 }
115
116 /**
117  * @brief Assigns properties from the given line to this line
118  *
119  * @param rhs: The given line to copy from
120  *
121  * @retval A copy of the given line
122 */
123 Line &Line::operator=(const Line &rhs){
124     //check if shape is being assigned it itself
125     if(this != &rhs){
126         color = rhs.color;
127         point1 = matrix(rhs.point1);
128         point2 = matrix(rhs.point2);
129     }
130     return *this;
131 }
132
133 /**
134  * @brief Draws the given line
135  *
136  * @param gc: GraphicsContext object that tells the shape where to draw
137  *
138  * @retval NONE
139 */
140 void Line::draw(GraphicsContext *gc){
141     gc->setColor(color);

```

```
142     gc->drawLine(point1[0][0], point1[1][0], point2[0][0], point2[1][0]);
143 }
144
145 /**
146  * @brief Print contents of line into std.
147  *         Method made const to prevent modifying when outputting
148  *
149  *         Shape type
150  *         Color: 0x.....
151  *         Point?: x y z
152  *
153  * @param os: Stream to write to
154  *
155  * @retval NONE
156  */
157 std::ostream &Line::out(std::ostream &os) const{
158     os << "Line" << std::endl;
159     Shape::out(os); //Call shape's printout first
160
161     os << "\tPoint 2: "
162         << point2[0][0] << " "
163         << point2[1][0] << " "
164         << point2[2][0]
165         << std::endl;
166
167     return os;
168 }
```

```

1  /**
2  * *****
3  * @file    : triangle.h
4  * @brief   : Outline for triangle shape class
5  *          : Lab 6: Event Driven Drawing
6  *          : CS-3210/021
7  * @date    : APR 27 2021
8  * @author  : Julian Singkham
9  * *****
10 */
11 #ifndef TRIANGLE_H
12 #define TRIANGLE_H
13
14 #include "shape.h"
15 //=====Class=====
16
17 class Triangle : public Shape{
18     private:
19         //points to draw to
20         matrix point2, point3;
21
22         /**
23          * @brief Constructor that makes a new triangle from a stream
24          *          Made private so that only image can create triangles with a stream.
25          *          Image will handle parsing through the file and determining what
26          *          shape gets created.
27          * @param is: Input stream that contains triangle parameters
28          * @retval NONE
29          */
30         Triangle(std::istream &is);
31
32         /**
33          * @brief Read triangle properties from a text file (stream)
34          * @param is: Stream to read from
35          * @retval NONE
36          */
37         std::istream &in(std::istream &is);
38
39     public:
40         friend class Image; //Allows image access to the instream methods
41
42         /**
43          * @brief Parameterized constructor, it creates a triangle with a color.
44          * @param color: 3x8-bit value for red, green, blue
45          * @retval NONE
46          */
47         Triangle(double x0, double y0, double x1, double y1, double x2, double y2, int col
48 or);
49
50         /**
51          * @brief Copy constructor that copies the paramters from the given triangle
52          * @param from: Triangle to copy into the current triangle.
53          * @retval None
54          */
55         Triangle(const Triangle &from);
56
57         /**
58          * @brief Virtual constructor thats used to copy a shape
59          * @param: NONE
60          * @retval NONE
61          */
62
63
64
65
66
67
68
69

```

```
70     Triangle *clone();
71
72     /**
73      * @brief Triangle destructor, frees memory allocated to triangle
74      *         Not currently used due to image handling deletion
75      *
76      * @param: NONE
77      *
78      * @retval NONE
79      */
80     ~Triangle();
81
82     /**
83      * @brief Assigns properties from the given triangle to this triangle
84      *
85      * @param rhs: The given triangle to copy from
86      *
87      * @retval A copy of the given triangle
88      */
89     Triangle &operator=(const Triangle &rhs);
90
91     /**
92      * @brief Draws the given triangle
93      *
94      * @param gc: GraphicsContext object that tells the shape where to draw
95      *
96      * @retval NONE
97      */
98     void draw(GraphicsContext *gc);
99
100    /**
101     * @brief Print contents of triangle into std.
102     *         Method made const to prevent modifying when outputting
103     *
104     *         Shape_type
105     *         Color: 0x.....
106     *         Point?: x y z
107     *
108     * @param os: Stream to write to
109     *
110     * @retval NONE
111     */
112     std::ostream &out(std::ostream &os) const;
113 };
114
115 #endif
```

```

1  /**
2      *****
3      * @file    : triangle.cpp
4      * @brief   : Triangle shape class
5      *         : Lab 6: Event Driven Drawing
6      *         : CS-3210/021
7      * @date    : APR 27 2021
8      * @author  : Julian Singkham
9      *****
10     * @attention
11     * Handles the creation of a triangle in 3-D space using x11 graphics.
12     *****
13  */
14  #include <sstream> //For String Stream
15
16  #include "triangle.h"
17  //=====Private=====
18  /**
19   * @brief Constructor that makes a new triangle from a stream
20   *         Made private so that only image can create triangles with stream.
21   *         Image will handle parsing through the file and determining what
22   *         shape gets created.
23   *
24   * @param is: Input stream that contains triangle parameters
25   *
26   * @retval NONE
27   */
28  Triangle::Triangle(std::istream &is)
29      : Shape(color), point2(5,5), point3(5,5){
30
31      in(is);
32  }
33
34  /**
35   * @brief Read triangle properties from a text file (stream)
36   *
37   * @param is: Stream to read from
38   *
39   * @retval NONE
40   */
41  std::istream &Triangle::in(std::istream &is){
42      std::string line;
43      std::stringstream str_stream;
44
45      Shape::in(is); //Call parent first
46
47      //Copy second point
48      std::getline(is, line); //Read line
49      str_stream = std::stringstream(line);
50      str_stream.ignore(32, ':');
51      str_stream >> point2[0][0];
52      str_stream >> point2[1][0];
53      str_stream >> point2[2][0];
54
55      //Copy third point
56      std::getline(is, line); //Read line
57      str_stream = std::stringstream(line);
58      str_stream.ignore(32, ':');
59      str_stream >> point3[0][0];
60      str_stream >> point3[1][0];
61      str_stream >> point3[2][0];
62
63      return is;
64  }
65  //=====Public=====
66
67  /**
68   * @brief Parameterized constructor, it creates a triangle with a color.
69   *
70   * @param color: 3x8-bit value for red, green, blue

```



```

71  * @retval NONE
72  */
73  Triangle::Triangle(double x0, double y0, double x1, double y1, double x2, double y2,
74                    int color) : Shape(color), point2(5,5), point3(5,5){
75
76      //Copy origin point
77      this->point1[0][0] = x0;
78      this->point1[1][0] = y0;
79      this->point1[2][0] = 0; //Default
80      this->point1[3][0] = 1; //Default
81
82      //Copy second point
83      this->point2[0][0] = x1;
84      this->point2[1][0] = y1;
85      this->point2[2][0] = 0; //Default
86      this->point2[3][0] = 1; //Default
87
88      //Copy third point
89      this->point3[0][0] = x2;
90      this->point3[1][0] = y2;
91      this->point3[2][0] = 0; //Default
92      this->point3[3][0] = 1; //Default
93  }
94
95  /**
96   * @brief Copy constructor that copies the paramters from the given triangle
97   *
98   * @param from: Triangle to copy into the current triangle.
99   *
100  * @retval None
101  */
102  Triangle::Triangle(const Triangle &from)
103      : Shape(from.color), point2(from.point2), point3(from.point3){
104
105      point1 = matrixx(from.point1);
106  }
107
108  /**
109   * @brief Virtual constructor thats used to copy a shape
110   *
111   * @param: NONE
112   *
113   * @retval NONE
114   */
115  Triangle *Triangle::clone(){
116      return new Triangle(*this);
117  }
118
119  /**
120   * @brief Triangle destructor, frees memory allocated to triangle
121   *      Not currently used due to image handling deletion
122   *
123   * @param: NONE
124   *
125   * @retval NONE
126   */
127  Triangle::~Triangle(){
128  }
129
130  /**
131   * @brief Assigns properties from the given triangle to this triangle
132   *
133   * @param rhs: The given triangle to copy from
134   *
135   * @retval A copy of the given triangle
136   */
137  Triangle &Triangle::operator=(const Triangle &rhs){
138      //check if shape is being assigned it itself
139      if(this != &rhs){
140          color = rhs.color;
141          point1 = matrixx(rhs.point1);

```

```

142         point2 = matrix(rhs.point2);
143         point3 = matrix(rhs.point3);
144     }
145     return *this;
146 }
147
148 /**
149  * @brief Draws the given triangle
150  *
151  * @param gc: GraphicsContext object that tells the shape where to draw
152  *
153  * @retval NONE
154  */
155 void Triangle::draw(GraphicsContext *gc){
156     gc->setColor(color);
157     gc->drawLine(point1[0][0], point1[1][0], point2[0][0], point2[1][0]);
158     gc->drawLine(point2[0][0], point2[1][0], point3[0][0], point3[1][0]);
159     gc->drawLine(point3[0][0], point3[1][0], point1[0][0], point1[1][0]);
160 }
161
162 /**
163  * @brief Print contents of triangle into std.
164  *         Method made const to prevent modifying when outputting
165  *
166  *         Shape_type
167  *         Color: 0x.....
168  *         Point?: x y z
169  *
170  * @param os: Stream to write to
171  *
172  * @retval NONE
173  */
174 std::ostream &Triangle::out(std::ostream &os) const{
175     os << "Triangle" << std::endl;
176     Shape::out(os); //Call shape's printout first
177
178     os << "\tPoint 2: "
179         << point2[0][0] << " "
180         << point2[1][0] << " "
181         << point2[2][0]
182         << std::endl;
183
184     os << "\tPoint 3: "
185         << point3[0][0] << " "
186         << point3[1][0] << " "
187         << point3[2][0]
188         << std::endl;
189
190     return os;
191 }

```

```

1  /**
2      *****
3      * @file      : image.h
4      * @brief     : Outline for image container class
5      *           : Lab 6: Event Driven Drawing
6      *           : CS-3210/021
7      * @date      : APR 27 2021
8      * @author    : Julian Singkham
9      *****
10 */
11 #ifndef IMAGE_H
12 #define IMAGE_H
13
14 #include <vector> //Shape verticies are stored in a vector
15
16 #include "shape.h"
17 #include "triangle.h"
18 #include "line.h"
19
20 //=====Class=====
21 class Image{
22     private:
23         std::vector<Shape *> shapes; //List of shapes in the container
24
25     public:
26         /**
27          * @brief Constructor
28          *
29          * @param: NONE
30          *
31          * @retval NONE
32          */
33         Image();
34
35         /**
36          * @brief Copy constructor that copies the contents from the given image
37          *
38          * @param from: Image to copy into the current image.
39          *
40          * @retval NONE
41          */
42         Image(const Image &from);
43
44         /**
45          * @brief Image destructor, frees memory allocated to image
46          *
47          * @param: NONE
48          *
49          * @retval NONE
50          */
51         ~Image();
52
53         /**
54          * @brief Delete all shapes within the image
55          *
56          * @param: NONE
57          *
58          * @retval NONE
59          */
60         void erase();
61
62         /**
63          * @brief Assigns the image to another image
64          *
65          * @param rhs: The given image to copy from
66          *
67          * @retval A copy of the given image
68          */
69         Image &operator=(const Image &rhs);
70
71         /**

```

```
72      * @brief Adds a shape to the container
73      *
74      * @param shape: Shape to add
75      *
76      * @retval NONE
77      */
78      void add(Shape *shape);
79
80      /**
81      * @brief Draws shapes in the image
82      *         Method made const to prevent modifying when outputting
83      *
84      * @param gc: GraphicsContext object that tells the shape where to draw
85      *
86      * @retval NONE
87      */
88      void draw(GraphicsContext *gc) const;
89
90      /**
91      * @brief Print contents of image into std.
92      *         Method made const to prevent modifying when outputting
93      *
94      * @param os: Stream to write to
95      *
96      * @retval NONE
97      */
98      std::ostream &out(std::ostream &os) const;
99
100     /**
101     * @brief Read shape properties from a text file (stream)
102     *
103     * @param is: Stream to read from
104     *
105     * @retval NONE
106     */
107     std::istream &in(std::istream &is);
108
109     /**
110     * @brief Shapes vector getter
111     *
112     * @param: NONE
113     *
114     * @retval Shapes vector
115     */
116     std::vector<Shape *> get_shapes();
117 };
118
119 #endif
```

```

1  /**
2   * *****
3   * @file    : Image.cpp
4   * @brief   : Image container class
5   *         : Lab 6: Event Driven Drawing
6   *         : CS-3210/021
7   * @date    : APR 27 2021
8   * @author  : Julian Singkham
9   * *****
10  * @attention
11  * The image class is a container for shapes. Think of image as a frame and shapes
12  * are added to the frame to be displayed on the monitor.
13  * When creating shapes with a stream, image must be called so that it can determine
14  * what shapes the parameters belong to.
15  * *****
16  */
17 #include <sstream> //For String Stream
18
19 #include <string>
20
21 #include "image.h"
22 //=====Public=====
23 /**
24  * @brief Constructor
25  *
26  * @param: NONE
27  *
28  * @retval NONE
29  */
30 Image::Image() {}
31
32 /**
33  * @brief Copy constructor that copies the contents from the given image
34  *
35  * @param from: Image to copy into the current image.
36  *
37  * @retval NONE
38  */
39 Image::Image(const Image &from) {
40     for (Shape *i : from.shapes)
41         add((i)->clone());
42 }
43
44 /**
45  * @brief Image destructor, frees memory allocated to image
46  *
47  * @param: NONE
48  *
49  * @retval NONE
50  */
51 Image::~Image() {
52     erase();
53 }
54
55 /**
56  * @brief Delete all shapes within the image
57  *
58  * @param: NONE
59  *
60  * @retval NONE
61  */
62 void Image::erase() {
63     for (Shape *i : shapes)
64         delete i;
65     shapes.clear();
66 }
67
68 /**
69  * @brief Assigns the image to another image
70  *
71  * @param rhs: The given image to copy from

```

```

72  *
73  * @retval A copy of the given image
74  */
75  Image &Image::operator=(const Image &rhs){
76      //check if image is being assigned it itself
77      if(this != &rhs){
78          shapes.clear();
79          for (Shape *i : rhs.shapes)
80              add((i)->clone());
81      }
82      return *this;
83  }
84
85  /**
86   * @brief Adds a shape to the container
87   *
88   * @param shape: Shape to add
89   *
90   * @retval NONE
91   */
92  void Image::add(Shape *shape){
93      shapes.push_back(shape);
94  }
95
96  /**
97   * @brief Draws tall shapes in the image
98   *          Method made const to prevent modifying when outputting
99   *
100   * @param gc: GraphicsContext object that tells the shape where to draw
101   *
102   * @retval NONE
103   */
104  void Image::draw(GraphicsContext *gc) const{
105      for (Shape *i : shapes)
106          (i)->draw(gc);
107  }
108
109  /**
110   * @brief Print contents of image into std.
111   *          Method made const to prevent modifying when outputting
112   *
113   * @param os: Stream to write to
114   *
115   * @retval NONE
116   */
117  std::ostream &Image::out(std::ostream &os) const{
118      for (Shape *i : shapes)
119          i->out(os);
120      return os;
121  }
122
123  /**
124   * @brief Read shape properties from a text file (stream)
125   *
126   * @param is: Stream to read from
127   *
128   * @retval NONE
129   */
130  std::istream &Image::in(std::istream &is){
131      std::string str_line;
132      while(std::getline(is, str_line)){
133          if (str_line.rfind("Line", 0) == 0)
134              add(new Line(is));
135          else if (str_line.rfind("Triangle", 0) == 0)
136              add(new Triangle(is));
137          else
138              std::cout << "Unable to read line, Skipping" << std::endl;
139      }
140
141      return is;
142  }

```

```
143
144 /**
145  * @brief Shapes vector getter
146  *
147  * @param: NONE
148  *
149  * @retval Shapes vector
150  */
151 std::vector<Shape *> Image::get_shapes() {
152     return shapes;
153 }
```

1 **Table of Contents**

2	1	main.cpp.....	sheets	1 to	1 (1)	pages	1-	1	66	lines
3	2	xllcontext.h.....	sheets	2 to	2 (1)	pages	2-	2	60	lines
4	3	xllcontext.cpp.....	sheets	3 to	6 (4)	pages	3-	6	224	lines
5	4	gcontext.h.....	sheets	7 to	9 (3)	pages	7-	9	145	lines
6	5	gcontext.cpp.....	sheets	10 to	10 (1)	pages	10-	10	38	lines
7	6	drawbase.h.....	sheets	11 to	11 (1)	pages	11-	11	32	lines
8	7	mydrawing.h.....	sheets	12 to	14 (3)	pages	12-	14	177	lines
9	8	mydrawing.cpp.....	sheets	15 to	21 (7)	pages	15-	21	428	lines
10	9	matrix.h.....	sheets	22 to	24 (3)	pages	22-	24	174	lines
11	10	matrix.cpp.....	sheets	25 to	28 (4)	pages	25-	28	280	lines
12	11	shape.h.....	sheets	29 to	30 (2)	pages	29-	30	107	lines
13	12	shape.cpp.....	sheets	31 to	32 (2)	pages	31-	32	123	lines
14	13	line.h.....	sheets	33 to	34 (2)	pages	33-	34	117	lines
15	14	line.cpp.....	sheets	35 to	37 (3)	pages	35-	37	169	lines
16	15	triangle.h.....	sheets	38 to	39 (2)	pages	38-	39	116	lines
17	16	triangle.cpp.....	sheets	40 to	42 (3)	pages	40-	42	192	lines
18	17	image.h.....	sheets	43 to	44 (2)	pages	43-	44	120	lines
19	18	image.cpp.....	sheets	45 to	47 (3)	pages	45-	47	154	lines