```
/**
  *****************************************************************************
  * @file    : main.cpp
  * @brief   : Main program
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : May 11 2021
  * @author : Julian Singkham
  *****************************************************************************
  * @attention
  * This lab is an extension of the previous lab where the user can now interact with
  * the graphics window.
  *
  * Features:
  *   User can draw points, lines, and triangles with 9 different colors.
  *   All draw methods rubberband.
  *   User can undo the last drawn shape.
  *   User can save the image to output.txt.
  *   User can load image data from file using the console.
  *   Window can be resized without affecting the image.
  *****************************************************************************
**/

#include "shape.h"
#include "image.h"
#include "x11context.h"
#include "mydrawing.h"

using namespace std;
//====================================Methods=======================================
/**
  * @brief The program entry point. Assume tests are successful unles otherwise stated
  *
  * @param: NOT USED
  *
  * @retval NOT USED
  */
int main(){
    GraphicsContext *gc = new X11Context(800, 600, GraphicsContext::BLACK);

    cout << "This program creates an interactive graphics window using the mouse "
         << "to set the location of a point and keyboard to specify commands\n\n"
         << "Pen Color Settings\n"
         << "Key 1: Black\n"
         << "Key 2: Gray\n"
         << "Key 3: White\n"
         << "Key 4: Red\n"
         << "Key 5: Green\n"
         << "Key 6: Blue\n"
         << "Key 7: Cyan\n"
         << "Key 8: Magenta\n"
         << "Key 9: Yellow\n\n"
         << "Graphics Commands\n"
         << "Key P: Draw Point\n"
         << "Key L: Draw Line\n"
         << "Key T: Draw Triangle\n"
         << "Key S: Save image to output.txt\n"
         << "Key O: Load image from file using console\n"
         << "Key Z: Undo last drawn shape" << endl;

    MyDrawing md;//Make a drawing
    gc->runLoop(&md); //Event loop that returns when window is closed
    delete gc;
    return 0;
}
```

```cpp
/**
  ***********************************************************************************
  * @file    : x11context.h
  * @brief   : Outline for X11 context
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : APR 27 2021
  * @author : unknown (provided)
  ***********************************************************************************
**/

#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========================================
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
```

```
/**
  **********************************************************************************
  * @file    : x11context.h
  * @brief   : Outline for X11 context
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : APR 27 2021
  * @author  : unknown (provided)
  **********************************************************************************
  * @attention
  * 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.
 * */
//========================================Class========================================
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
void X11Context::setMode(drawMode newMode)
{
    if (newMode == GraphicsContext::MODE_NORMAL)
    {
        XSetFunction(display,graphics_context,GXcopy);
    }
    else
    {
        XSetFunction(display,graphics_context,GXxor);
    }
}

// Set drawing color - assume colormap is 24 bit RGB
void X11Context::setColor(unsigned int color)
{
    // Go ahead and set color here - better performance than setting
    // on every setPixel
    XSetForeground(display, graphics_context, color);
}

// Set a pixel in the current color
void X11Context::setPixel(int x, int y)
{
    XDrawPoint(display, window, graphics_context, x, y);
    XFlush(display);
}

unsigned int X11Context::getPixel(int x, int y)
{
    XImage *image;
    image = XGetImage (display, window, x, y, 1, 1, AllPlanes, XYPixmap);
    XColor color;
    color.pixel = XGetPixel (image, 0, 0);
    XFree (image);
    XQueryColor (display, DefaultColormap(display, DefaultScreen (display)),
                 &color);
    // I now have RGB values, but, they are 16 bits each, I only want 8-bits
    // each since I want a 24-bit RGB color value
    unsigned int pixcolor = color.red & 0xff00;
    pixcolor |= (color.green >> 8);
    pixcolor <<= 8;
    pixcolor |= (color.blue >> 8);
    return pixcolor;
}

void X11Context::clear()
{
    XClearWindow(display, window);
    XFlush(display);
}



```

```
// Run event loop
void X11Context::runLoop(DrawingBase* drawing)
{
    run = true;

    while(run)
    {
        XEvent e;
        XNextEvent(display, &e);

        // Exposure event - lets not worry about region
        if (e.type == Expose)
            drawing->paint(this);

        // Key Down
        else if (e.type == KeyPress)
            drawing->keyDown(this,XLookupKeysym((XKeyEvent*)&e,
                    (((e.xkey.state&0x01)&&!(e.xkey.state&0x02))||
                    (!(e.xkey.state&0x01)&&(e.xkey.state&0x02)))?1:0));

        // Key Up
        else if (e.type == KeyRelease){
            drawing->keyUp(this,XLookupKeysym((XKeyEvent*)&e,
                    (((e.xkey.state&0x01)&&!(e.xkey.state&0x02))||
                    (!(e.xkey.state&0x01)&&(e.xkey.state&0x02)))?1:0));
                }

        // Mouse Button Down
        else if (e.type == ButtonPress)
            drawing->mouseButtonDown(this,
            e.xbutton.button,
            e.xbutton.x,
            e.xbutton.y);

        // Mouse Button Up
        else if (e.type == ButtonRelease)
            drawing->mouseButtonUp(this,
            e.xbutton.button,
            e.xbutton.x,
            e.xbutton.y);

        // Mouse Move
        else if (e.type == MotionNotify)
            drawing->mouseMove(this,
            e.xmotion.x,
            e.xmotion.y);

        // This will respond to the WM_DELETE_WINDOW from the
        // window manager.
        else if (e.type == ClientMessage)
        break;
    }
}


int X11Context::getWindowWidth()
{
    XWindowAttributes window_attributes;
    XGetWindowAttributes(display,window, &window_attributes);
    return window_attributes.width;
}

int X11Context::getWindowHeight()
{
    XWindowAttributes window_attributes;
    XGetWindowAttributes(display,window, &window_attributes);
    return window_attributes.height;
}

void X11Context::drawLine(int x1, int y1, int x2, int y2)
{
    XDrawLine(display, window, graphics_context, x1, y1, x2, y2);
    XFlush(display);
}

void X11Context::drawCircle(int x, int y, unsigned int radius)
{
    XDrawArc(display, window, graphics_context, x-radius,
                     y-radius, radius*2, radius*2, 0, 360*64);
    XFlush(display);
}
```

```
/**
  *************************************************************************************
  * @file   : gcontext.h
  * @brief  : Outline for Graphics Context
  *         : Lab 6: Event Driven Drawing
  *         : CS-3210/021
  * @date   : APR 27 2021
  * @author : unknown (provided)
  *************************************************************************************
  * 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.
  *************************************************************************************
**/
#ifndef GCONTEXT_H
#define GCONTEXT_H

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

        // This should reset entire context to the current background
        virtual void clear()=0;

        // These are the naive implementations that use setPixel,
        // but are overridable should a context have a better-
        // performing version available.

         /* will need to be provided by the concrete
         * implementation.
         *
         * Parameters:
         *  x0, y0 - origin of line
         *  x1, y1 - end of line
         *
         * Returns: void
         */
        virtual void drawLine(int x0, int y0, int x1, int y1);

        /* will need to be provided by the concrete
         * implementation.
         *
         * Parameters:
         *  x0, y0 - origin/center of circle
         *  radius - radius of circle
         *
         * Returns: void
         */
        virtual void drawCircle(int x0, int y0, unsigned int radius);


        /**********************************************************
         * Event loop operations
         **********************************************************/

        // Run Event loop.  This routine will receive events from
        // the implementation and pass them along to the drawing.  It
        // will return when the window is closed or other implementation-
        // specific sequence.
        virtual void runLoop(DrawingBase* drawing) = 0;

        // This method will end the current loop if one is running
        // a default version is supplied
        virtual void endLoop();


        /**********************************************************
         * Utility operations
         **********************************************************/

        // returns the width of the window
        virtual int getWindowWidth() = 0;

        // returns the height of the window
        virtual int getWindowHeight() = 0;

    protected:
        // this flag is used to control whether the event loop
        // continues to run.
        bool run;
};
```

```

#endif
```

```
/**
  ***********************************************************************************
  * @file   : gcontext.cpp
  * @brief  : Graphics Context
  *         : Lab 6: Event Driven Drawing
  *         : CS-3210/021
  * @date   : APR 27 2021
  * @author : unknown (provided)
  ***********************************************************************************
  * 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;
}
```

```
/**
  ************************************************************************************
  * @file   : DrawBase.h
  * @brief  : Outline for Drawing base class
  *         : Lab 6: Event Driven Drawing
  *         : CS-3210/021
  * @date   : APR 27 2021
  * @author : Unknown (provided)
  ************************************************************************************
**/
#ifndef DRAWBASE_H
#define DRAWBASE_H

// forward reference
class GraphicsContext;
//========================================Class==========================================
//Refer to mydrawing.h for documentation
class DrawingBase{
    public:
        // prevent warnings
        virtual ~DrawingBase(){}
        virtual void paint(GraphicsContext* gc){}
        virtual void keyDown(GraphicsContext* gc, unsigned int keycode){}
        virtual void keyUp(GraphicsContext* gc, unsigned int keycode){}
        virtual void mouseButtonDown(GraphicsContext* gc,
                                unsigned int button, int x, int y){}
        virtual void mouseButtonUp(GraphicsContext* gc,
                                unsigned int button, int x, int y){}
        virtual void mouseMove(GraphicsContext* gc, int x, int y){}
};
#endif
```

```
/**
  **********************************************************************************
  * @file    : mydrawing.h
  * @brief   : Outline for user input drawing class, derived from drawBase
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : MAY 11 2021
  * @author : Julian Singkham
  **********************************************************************************
**/
#ifndef MYDRAWING_H
#define MYDRAWING_H
#include "drawbase.h"
#include "image.h"

// forward reference
class GraphicsContext;

//XK key values
enum drawing_state{
    //Logic
    point    = 112, //P
    line     = 108, //L
    triangle = 116, //T
    save     = 115, //S
    load     = 111, //O
    undo     = 122, //Z

    //Pen Colors
    black    = 49, //1
    gray     = 50, //2
    white    = 51, //3
    red      = 52, //4
    green    = 53, //5
    blue     = 54, //6
    cyan     = 55, //7
    magenta  = 56, //8
    yellow   = 57, //9
};

//========================================Class========================================
class MyDrawing : public DrawingBase{
    private:
        //Shape properties
        int x0;
        int y0;
        int x1;
        int y1;
        int x2;
        int y2;
        int color;

        //mydrawing properties
        bool dragging; //Flag to know if mouse is moving
        bool can_undo; //Flag to know if undo can be used
        Image image;   //Shapes container
        int points_count;        //Keeps track of what point is being drawn
        drawing_state state;     //Keeps track of what command is being run
        drawing_state old_shape; //Keeps track of last drawn shape

        /**
         * @brief Saves image information to output.txt
         *
         * @param image: Image to save
         *
         * @retval NONE
         */
        void save_to_file(const Image image) const;

        /**
         * @brief Loads image information from file specified in console
         *
         * @param gc: Where to draw image
         *
         * @retval NONE
         */
        void load_file(GraphicsContext* gc);

        /**
         * @brief Undos the last drawn shape. Only works if the user isn’t in
         * the process of drawing a shape. Shapes aren’t saved to image until
         * another shape is drawn or when saving.
         *
         * @param gc: Where to draw image
         *
         * @retval NONE
         */
        void undo_shape(GraphicsContext* gc);

        /**
         * @brief Saves the last drawn shape to image
         *
         * @param: NONE
         *
         * @retval NONE
         */
        void update_image();

    public:
        /**
         * @brief Constructor for a drawing (window). MyDrawing contains an image
         * that contains all shapes to draw and uses a graphic context to display.
         *
         * @param: None
         *
         * @retval NONE
         */
        MyDrawing();

        /**
         * @brief Handles redrawing when window is resized
         *
         * @param gc: Where to draw image
         *
         * @retval NONE
         */
        void paint(GraphicsContext* gc);

        /**
         * @brief Handles key presses from keyboard. Not currently used.
         *
         * @param gc: Where to draw image
         * @param keycode: XK value of pressed key
         *
         * @retval NONE
         */
        void keyDown(GraphicsContext *gc, unsigned int keycode);

        /**
         * @brief Handles key releases from keyboard
         *
         * @param gc: Where to draw image
         * @param keycode: XK value of released key
         *
         * @retval NONE
         */
        void keyUp(GraphicsContext *gc, unsigned int keycode);

        /**
         * @brief Handles mouse presses
         *
         * @param gc: Where to draw image
         * @param button: What mouse button was pressed
         * @param x: x-coordinate of mouse cursor
         * @param y: y-coordinate of mouse cursor
         *
         * @retval NONE
         */
        void mouseButtonDown(GraphicsContext* gc, unsigned int button, int x,
                                    int y);

        /**
         * @brief Handles mouse releases
         *
         * @param gc: Where to draw image
         * @param button: What mouse button was released
         * @param x: x-coordinate of mouse cursor
         * @param y: y-coordinate of mouse cursor
         *
         * @retval NONE
         */
        void mouseButtonUp(GraphicsContext* gc, unsigned int button, int x,
                                    int y);

        /**
         * @brief Handles mouse movement
         *
         * @param gc: Where to draw image
         * @param x: x-coordinate of mouse cursor
         * @param y: y-coordinate of mouse cursor
         *
         * @retval NONE
         */
        void mouseMove(GraphicsContext* gc, int x, int y);
};
#endif
```

```cpp
/**
  ************************************************************************************
  * @file    : mydrawing.cpp
  * @brief   : User driven drawing class, derived from drawbase
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : MAY 11 2021
  * @author  : Unknown (provided)
  ************************************************************************************
  * @attention
  * The mydrawing handles user interaction with the graphics context window.
  * A shape is saved to mydrawing, which is then added to image once another shape
  * is drawn or saving to file. This was done to support an undo last shape feature.
  ************************************************************************************
**/
#include "mydrawing.h"
#include <iostream>
#include <stdexcept>
#include <fstream>

using namespace std;
//======================================Private=======================================
/**
 * @brief Saves image information to output.txt
 *
 * @param image: Image to save
 *
 * @retval NONE
 */
void MyDrawing::save_to_file(const Image image)const{
    ofstream file;
    file.open("output.txt", ofstream::trunc);
    image.out(file);
    file.close();
    cout << "Saving to output.txt" << endl;
}
/**
 * @brief Loads image information from file specified in console
 *
 * @param gc: Where to draw image
 *
 * @retval NONE
 */
void MyDrawing::load_file(GraphicsContext* gc){
    cout << "Enter name of .txt file containing image information" << endl;
    string file_name;
    cin >> file_name;
    ifstream file(file_name);
    if (!file.is_open())
        cout << "Error: unable to open " << file_name << endl;
    else{
        image.in(file);
        cout << "Loaded " << file_name << endl;
        gc -> clear();
        image.draw(gc);
        file.close();
    }
}

/**
 * @brief Undos the last drawn shape. Only works if the user isn't in
 * the process of drawing a shape. Shapes aren't saved to image until
 * another shape is drawn or when saving.
 *
 * @param gc: Where to draw image
 *
 * @retval NONE
 */
void MyDrawing::undo_shape(GraphicsContext* gc){
    switch (old_shape){
        case point:
                gc->setMode(GraphicsContext::MODE_XOR);
                gc->drawLine(x0, y0, x0, y0);
                gc->setMode(GraphicsContext::MODE_NORMAL);
                break;
            case line:
                gc->setMode(GraphicsContext::MODE_XOR);
                gc->drawLine(x0, y0, x1, y1);
                gc->setMode(GraphicsContext::MODE_NORMAL);
                break;
            case triangle:
                gc->setMode(GraphicsContext::MODE_XOR);
                gc->drawLine(x0, y0, x1, y1);
                gc->drawLine(x1, y1, x2, y2);
                gc->drawLine(x2, y2, x0, y0);
                //Triangles create points on vertices on window
                //This only exists in the window, image is not affected
                gc->drawLine(x0, y0, x0, y0);
                gc->drawLine(x1, y1, x1, y1);
                gc->drawLine(x2, y2, x2, y2);
                gc->setMode(GraphicsContext::MODE_NORMAL);
                break;
            default:
                break;
        }
        can_undo = false;
        gc->clear();
        image.draw(gc);
        cout << "Undo" << endl;
}
/**
 * @brief Saves the last drawn shape to image
 *
 * @param: NONE
 *
 * @retval NONE
 */
void MyDrawing::update_image(){
        switch (old_shape){
            case point:
                image.add(new Line(x0, y0, x0, y0, color));
                break;
            case line:
                image.add(new Line(x0, y0, x1, y1, color));
                break;
            case triangle:
                image.add(new Triangle(x0, y0, x1, y1, x2, y2, color));
                break;
            default:
                break;
        }
        can_undo = false;
}
//=======================================Public======================================

/**
 * @brief Constructor for a drawing (window). MyDrawing contains an image
 * that contains all shapes to draw and uses a graphic context to display.
 *
 * @param: None
 *
 * @retval NONE
 */
MyDrawing::MyDrawing(){
        dragging = false;
        can_undo = false;
        x0 = x1 = x2 = y0 = y1 = y2 = 0;
        points_count = 0;
        color = GraphicsContext::WHITE; //Default pen color
        image = Image();
        return;
}
```

```

/**
 * @brief Handles redrawing when window is resized
 *
 * @param gc: Where to draw image
 *
 * @retval NONE
 */
void MyDrawing::paint(GraphicsContext* gc){
    // it will only show up after an exposure
    image.draw(gc);
    //Since the last drawn isn't saved to image, we have to redraw it to
    //preserve the undo functionality
    if(can_undo){
        switch (old_shape){
            case point:
                gc->setMode(GraphicsContext::MODE_NORMAL);
                gc->drawLine(x0, y0, x0, y0); //draw line in copy mode
                break;
            case line:
                gc->setMode(GraphicsContext::MODE_NORMAL);
                gc->drawLine(x0, y0, x1, y1); //draw line in copy mode
                break;
            case triangle:
                gc->setMode(GraphicsContext::MODE_NORMAL);
                gc->drawLine(x0, y0, x1, y1); //draw line in copy mode
                gc->drawLine(x1, y1, x2, y2);
                gc->drawLine(x2, y2, x0, y0);
                break;
            default:
                break;
            }
    }
    return;
}

/**
 * @brief Handles key presses from keyboard. Not currently used.
 *
 * @param gc: Where to draw image
 * @param keycode: XK value of pressed key
 *
 * @retval NONE
 */
void MyDrawing::keyDown(GraphicsContext *gc, unsigned int keycode){
    return;
}

/**
 * @brief Handles key releases from keyboard
 *
 * @param gc: Where to draw image
 * @param keycode: XK value of released key
 *
 * @retval NONE
 */
void MyDrawing::keyUp(GraphicsContext *gc, unsigned int keycode){
    switch (keycode){
        case point:
            state = point;
            cout << "POINT" << endl;
            break;
        case line:
            state = line;
            cout << "LINE" << endl;
            break;
        case triangle:
            state = triangle;
            points_count = 0;
            cout << "TRIANGLE" << endl;
            break;
        case save:
            if(can_undo)
                update_image();
            save_to_file(image);
            break;
        case load:
            load_file(gc);
            break;
        case undo:
            if(can_undo)
                undo_shape(gc);
            else
                cout << "Unable to undo: Either the last shape was already "
                     << "removed or no images have been added" << endl;
            break;
        case black:
            color = GraphicsContext::BLACK;
            cout << "BLACK" << endl;
            break;
        case gray:
            color = GraphicsContext::GRAY;
            cout << "GRAY" << endl;
            break;
        case white:
            color = GraphicsContext::WHITE;
            cout << "WHITE" << endl;
            break;
        case red:
            color = GraphicsContext::RED;
            cout << "RED" << endl;
            break;
        case green:
            color = GraphicsContext::GREEN;
            cout << "GREEN" << endl;
            break;
        case blue:
            color = GraphicsContext::BLUE;
            cout << "BLUE" << endl;
            break;
        case cyan:
            color = GraphicsContext::CYAN;
            cout << "CYAN" << endl;
            break;
        case magenta:
            color = GraphicsContext::MAGENTA;
            cout << "MAGENTA" << endl;
            break;
        case yellow:
            color = GraphicsContext::YELLOW;
            cout << "YELLOW" << endl;
            break;
        default:
            cout << "Key down: " << keycode << endl;
            break;
    }
    gc->setColor(color);
}

/**
 * @brief Handles mouse presses
 *
 * @param gc: Where to draw image
 * @param button: What mouse button was pressed
 * @param x: x-coordinate of mouse cursor
 * @param y: y-coordinate of mouse cursor
 *
 * @retval NONE
 */
void MyDrawing::mouseButtonDown(GraphicsContext* gc, unsigned int button, int x,
                                int y) {
    //Undow will only be available if the user isn't drawing a shape
    if(can_undo)
        update_image();

    //gc MODE_NORMAL means that shapes will be drawn ontop of the frame
    //gc MODE_XOR means that shape collision will negate eachother
    switch (state){
        case point:
            x0 = x;
            y0 = y;
            gc->setMode(GraphicsContext::MODE_NORMAL);
            gc->drawLine(x, y, x, y);
            old_shape = point;
            can_undo = true;
            break;
        case line:
            if (points_count == 0){
                points_count = 1;
                x0 = x1 = x;
                y0 = y1 = y;
                gc->setMode(GraphicsContext::MODE_XOR);
                dragging = true;
            }
            else{
                dragging = false;
                gc->drawLine(x0, y0, x1, y1); //Delete old line
                x1 = x;
                y1 = y;
                gc->setMode(GraphicsContext::MODE_NORMAL);
                gc->drawLine(x0, y0, x1, y1);

                //Set Flags
                points_count = 0;
                old_shape = line;
                can_undo = true;
            }
            break;
        case triangle:
            if (points_count == 0){
                points_count = 1;
                x0 = x1 = x2 = x;
                y0 = y1 = y2 = y;
                gc->setMode(GraphicsContext::MODE_XOR);
                dragging = true;
            }
            else if (points_count == 1){
                dragging = false;
                points_count = 2;
                gc->drawLine(x0, y0, x1, y1); //Delete old line
                x1 = x;
                y1 = y;
                gc->setMode(GraphicsContext::MODE_NORMAL);
                gc->drawLine(x0, y0, x1, y1);
                gc->setMode(GraphicsContext::MODE_XOR);

                //Set Flags
                dragging = true;
            }
            else{
                dragging = false;
                points_count = 0;
                gc->drawLine(x1, y1, x2, y2); //Delete old line
                gc->drawLine(x2, y2, x0, y0); //Delete old line
                x2 = x;
                y2 = y;
                gc->setMode(GraphicsContext::MODE_NORMAL);
                gc->drawLine(x1, y1, x2, y2);
                gc->drawLine(x2, y2, x0, y0);

                //Set Flags
                old_shape = triangle;
                can_undo = true;
                }
                break;
            default:
                break;
        }
        return;
}

/**
 * @brief Handles mouse releases. Not currently used.
 *
 * @param gc: Where to draw image
 * @param button: What mouse button was released
 * @param x: x-coordinate of mouse cursor
 * @param y: y-coordinate of mouse cursor
 *
 * @retval NONE
 */
void MyDrawing::mouseButtonUp(GraphicsContext* gc, unsigned int button, int x,
                                int y){
    return;
}

/**
 * @brief Handles mouse movement
 * Assume GC in MODE_XOR prior to function call
 *
 * @param gc: Where to draw image
 * @param x: x-coordinate of mouse cursor
 * @param y: y-coordinate of mouse cursor
 *
 * @retval NONE
 */
void MyDrawing::mouseMove(GraphicsContext* gc, int x, int y){
    switch (state){
        case line:
            if (dragging){
                gc->drawLine(x0, y0, x1, y1); //Delete old line
                //Update
                x1 = x;
                y1 = y;
                gc->drawLine(x0, y0, x1, y1); //Draw new line
            }
            break;
        case triangle:
            if (dragging){
                if(points_count == 1){
                    gc->drawLine(x0, y0, x1, y1);
                    x1 = x;
                    y1 = y;
                    gc->drawLine(x0, y0, x1, y1);
                }
                else if(points_count == 2){
                    //Since all points are inialized to the origin, we have to
                    //update the last vertex before drawing it or else the
                    //first line will be missing in the window
                    if((x2 != x0) || (y2 != y0)){

                        gc->drawLine(x1, y1, x2, y2);
                        gc->drawLine(x2, y2, x0, y0);
                    }
                    x2 = x;
                    y2 = y;
                    gc->drawLine(x1, y1, x2, y2);
                    gc->drawLine(x2, y2, x0, y0);
                }
            }
            break;
        default:
            break;
    }
```

```cpp
        return;
}
```

```
/**
  ************************************************************************************
  * @file    : matrix.h
  * @brief   : Outline for matrix
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : MAR 30 2021
  * @author : Dr. Darrin Rothe
  * @author : Julian Singkham (modified)
  ************************************************************************************
**/

#ifndef MATRIX_H
#define MATRIX_H

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

/**
 * @brief Custom exception used in the matrix class.
 */
class matrix_Exception:public std::runtime_error{
    public:
        matrix_Exception(std::string message):
              std::runtime_error((std::string("A Matrix Error has occured: ") +
                message).c_str()) {}
};
//==========================================Class==========================================
class matrix
{
    public:
        /**
     * @brief Makes the insertion operator a friend so it can access matrix.
         * Basically allows the matrix to be printed to std.
     *
     * @param os: Stream to write to.
         * @param rhs: Reference to the matrix that is being printed.
     *
     * @retval Stream containing the matrix printout.
     */
        friend std::ostream& operator<<(std::ostream& os, const matrix& rhs);

        /**
     * @brief Parameterized constructor, it creates a matrix of given dimensions
         * with clear (zeroed) cells.
     *
     * @param rows: How many rows in the matrix.
         * @param cols: How many columns in the matrix.
     *
     * @retval NONE
     */
        matrix(unsigned int rows, unsigned int cols);

        /**
     * @brief Copy constructor that makes a new matrix from a given one.
     *
     * @param from: matrix to copy into the new matrix.
     *
     * @retval A copy of the given matrix.
     */
        matrix(const matrix& from);

        /**
     * @brief Frees allocated memory from the matrix.
     *
     * @param: NONE
     *
     * @retval NONE
     */
        ~matrix();

        /**
     * @brief Named constructor, it creates an identity matrix of given size.
         *
         * Identity matrix is a matrix that is all zeros expect when
         * row#=col#, then it is 1.
         * // 1 0 0 [0][0] = 1
         * // 0 1 0 [1][1] = 1
         * // 0 0 1 [2][2] = 1
     *
     * @param size: Square dimensions of the matrix.
     *
     * @retval The identity matrix.
     */
        static matrix identity(unsigned int size);

        /**
     * @brief Assigns the matrix to the value stored in the given matrix.
     *
     * @param rhs: The given matrix to copy from.
     *
     * @retval A copy of the given matrix.
     */
        matrix& operator=(const matrix& rhs);

        /**
     * @brief Matrix addiiton. The lhs and rhs must be the same size.
     *
     * @param rhs: The right hand side matrix.
     *
     * @retval The resulting matrix after addition.
     */
        matrix operator+(const matrix& rhs) const;

        /**
     * @brief Matrix multiplication.
     * The lhs column size and rhs row size must match.
     *
     * @param rhs: The right hand side matrix.
     *
     * @retval The resulting matrix after multiplication.
     * Dimension: lhs.rows x rhs.cols.
     */
        matrix operator*(const matrix& rhs) const;


        /**
     * @brief Matrix scaler multiplication.
     * This only supports matrix * 5, not 5 * matrix.
     *
     * @param scale: Value to scale the matrix.
     *
     * @retval The scaled matrix.
     */
        matrix operator*(const double scale) const;

        /**
     * @brief This allows access of the matrix elements by using [].
     *
     * @param row: The desired row of the matrix.
     *
     * @retval A pointer to the desired element of the matrix.
     */
        double* operator[](unsigned int row);

        /**
     * @brief This allows access of the matrix elements by using [].
     * Const version
     *
     * @param row: The desired row of the matrix.
     *
```

```
       * @retval A pointer to the desired element of the matrix.
       */
          double* operator[](unsigned int row) const;

          /**
       * @brief Zeroes the elements of the matrix.
       *
       * @param: None
       *
       * @retval None
       */
          void clear();
    private:
          //The data
          double** the_matrix;
          unsigned int rows;
          unsigned int cols;
};
//=======================================Global=======================================
/**
 * @brief Matrix scaler multiplication.
 * This only supports matrix * 5, not 5 * matrix.
 *
 * @param scale: Value to scale the matrix.
 * @param rhs: The matrix to apply the scaling.
 *
 * @retval The scaled matrix.
 */
matrix operator*(const double scale, const matrix& rhs);

#endif
```

```
/**
  ***********************************************************************************
  * @file   : matrixm.cpp
  * @brief  : Matrix
  *         : Lab 6: Event Driven Drawing
  *         : CS-3210/021
  * @date   : MAR 23 2021
  * @author : Julian Singkham
  ***********************************************************************************
  * @attention
  * This API handles the creation, deletion, and =,+,* operators of a matrix as well
  * as retrieving matrix values using []. In essence the double matrix used in this
  * API is an array of arrays where the **double matrix points to rows *double arrays
  * that then point to col elements.
  ***********************************************************************************
**/
#include "matrix.h"
#include <string>
#include <cmath>

//=======================================Public=======================================
/**
 * @brief Makes the insertion operator a friend so it can access matrix.
 * Basically allows the matrix to be printed to std.
 *
 * @param os: Stream to write to.
 * @param rhs: Reference to the matrix that is being printed.
 *
 * @retval Stream containing the matrix printout.
 */
std::ostream& operator<<(std::ostream& os, const matrix& rhs){
    for(int i = 0; i < rhs.rows; i++){
        os << "|";
        for(int j = 0; j < rhs.cols; j++){
            double temp = rhs.the_matrix[i][j];
            os << temp << "|";
        }
        os << std::endl;
    }
    return os;
}

/**
 * @brief Matrix constructor, it creates a matrix of given dimensions
 * with clear (zeroed) cells. Throws error if dimensions are not possible (<1).
 *
 * @param rows: How many rows in the matrix.
 * @param cols: How many columns in the matrix.
 *
 * @retval NONE
 */
matrix::matrix(unsigned int rows, unsigned int cols) : rows(rows), cols(cols){
    if(rows < 1 || cols < 1)
        throw matrix_Exception("p-constructor has bad arguments.");

    the_matrix = new double*[rows];//Allocates memory to the # of rows

    //Allocate memory for each row of the array to the # of columns
    //Basically creates a 1-D array of 1-D arrays
    for(int i = 0; i < rows; i++)
        the_matrix[i] = new double[cols];

    clear(); //Fill matrix with zeroes
}

/**
 * @brief Copy constructor that makes a new matrix from a given one.
 *
 * @param from: matrix to copy into the new matrix.
 *
 * @retval A copy of the given matrix.
 */
matrix::matrix(const matrix& from) : rows(from.rows), cols(from.cols){
    the_matrix = new double*[rows];//Allocates memory to the # of rows

    //Allocate memory for each row of the array to the # of columns
    //Basically creates a 1-D array of 1-D arrays
    for(int i = 0; i < rows; i++)
        the_matrix[i] = new double[cols];

    //Copy values from "from" into new matrix
    for(int i = 0; i < rows; i++)
        for(int j = 0; j < cols; j++)
            the_matrix[i][j] = from[i][j];
}

/**
 * @brief Frees allocated memory form matrix
 *
 * @param: NONE
 *
 * @retval NONE
 */
matrix::~matrix(){
    for(int i = 0; i <  rows; i++)
        delete[] the_matrix[i]; //Delete each row of the matrix
    delete[] the_matrix; //Delete the matrix itself
}

/**
 * @brief Named constructor, it creates a square identity matrix
 * of given size.
 *
 * Identity matrix is a matrix that is all zeros expect when
 * row#=col#, then it is 1.
 * // 1 0 0 [0][0] = 1
 * // 0 1 0 [1][1] = 1
 * // 0 0 1 [2][2] = 1
 *
 * @param size: Square dimensions of the matrix.
 *
 * @retval The square identity matrix.
 */
matrix matrix::identity(unsigned int size){
    if(size == 0)
        throw matrix_Exception("Can not create an identity matrix of size 0.");

    matrix return_matrix(size, size);
    for(int i = 0; i < size; i++)
        return_matrix[i][i] = 1.0;
    return return_matrix;
}

/**
 * @brief Assigns the matrix to the value stored in the given matrix.
 *
 * @param rhs: The given matrix to copy from.
 *
 * @retval A copy of the given matrix.
 */
matrix& matrix::operator=(const matrix& rhs){
    //Verify matrices match in size
    if(rows != rhs.rows || cols != rhs.cols){
        for(int i = 0; i <  rows; i++)
            delete[] the_matrix[i]; //Delete each row of the matrix
        delete[] the_matrix; //Delete the matrix itself

        rows = rhs.rows;
        cols = rhs.cols;
        the_matrix = new double*[rows];//Allocates memory to the # of rows

        //Allocate memory for each row of the array to the # of columns
        //Basically creates a 1-D array of 1-D arrays
        for(int i = 0; i < rows; i++)
            the_matrix[i] = new double[cols];
    }
    //Copy values from rhs into current matrix
    for(int i = 0; i < rows; i++)
        for(int j = 0; j < cols; j++)
            the_matrix[i][j] = rhs[i][j];

    return *this;
}

/**
 * @brief Matrix addiiton. The lhs and rhs must be the same size.
 *
 * @param rhs: The right hand side matrix.
 *
 * @retval The resulting matrix after addition.
 */
matrix matrix::operator+(const matrix& rhs) const{
    //Verify matrices match in size
    if(rows != rhs.rows || cols != rhs.cols)
        throw matrix_Exception("Size mismatch - The column/row of the left matrix"
                               " does not match the column/row of the right matrix:");

    matrix return_matrix(rows, cols);

    for(int i = 0; i < rows; i++)
        for(int j = 0; j < cols; j++)
            return_matrix[i][j] = the_matrix[i][j] + rhs[i][j];

    return return_matrix;
}

/**
 * @brief Matrix multiplication.
 * The lhs column size and rhs row size must match.
 *
 * @param rhs: The right hand side matrix.
 *
 * @retval The resulting matrix after multiplication.
 * Dimension: lhs.rows x rhs.cols.
 */
matrix matrix::operator*(const matrix& rhs) const{
    //Verify matrices match in size
    if(cols != rhs.rows)
        throw matrix_Exception("Size mismatch - The column of the left matrix does"
                               " not match the row of the right matrix:");

    matrix return_matrix(rows, rhs.cols);

    for (int i = 0; i < rows; ++i)
        for (int j = 0; j < rhs.cols; ++j)
            for (int k = 0; k < rhs.cols; ++k)
                return_matrix[i][j] += the_matrix[i][k] * rhs[k][j];
    return return_matrix;
}

/**
 * @brief Matrix scaler multiplication.
 * This only supports matrix * 5, not 5 * matrix.
 *
 * @param scale: Value to scale the matrix.
 *
 * @retval The scaled matrix.
 */
matrix matrix::operator*(const double scale) const{
    matrix return_matrix(rows, cols);

    for (int i = 0; i < rows; ++i)
        for (int j = 0; j < cols; ++j)
```

```
            return_matrix[i][j] = the_matrix[i][j] * scale;
    return return_matrix;
}

/**
 * @brief This allows access of the matrix elements by using [].
 *
 * @param row: The desired row of the matrix.
 *
 * @retval A pointer to the desired element of the matrix.
 */
double* matrix::operator[](unsigned int row){
    //Verify row is within bounds
    if (row >= rows || row < 0)
        throw matrix_Exception("Size mismatch - The requested row is"
                               " out of bounds.");
    double *ret = the_matrix[row];
    return ret;
}

/**
 * @brief This allows access of the matrix elements by using [].
 * Const version
 *
 * @param row: The desired row of the matrix.
 *
 * @retval A pointer to the desired element of the matrix.
 */
double* matrix::operator[](unsigned int row) const{
    //Verify row is within bounds
    if (row >= rows || row < 0)
        throw matrix_Exception("Size mismatch - The requested row is"
                               " out of bounds.");
    double *ret = the_matrix[row];
    return ret;
}

/**
 * @brief Zeroes the elements of the matrix.
 *
 * @param: None
 *
 * @retval None
 */
void matrix::clear(){
    for(int i = 0; i < rows; i++)
        for(int j = 0; j < cols; j++)
            the_matrix[i][j] = 0.0;
    return;
}

//======================================Global======================================
/**
 * @brief Matrix scaler multiplication.
 * This only supports matrix * 5, not 5 * matrix.
 *
 * @param scale: Value to scale the matrix.
 * @param rhs: The matrix to apply the scaling.
 *
 * @retval The scaled matrix.
 */
matrix operator*(const double scale, const matrix& rhs){
    matrix return_matrix(rhs);
    return_matrix = rhs * scale;
    return return_matrix;
}
```

```
/**
  ***********************************************************************************
  * @file    : shape.h
  * @brief   : Outline for shape base class
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : APR 27 2021
  * @author : Julian Singkham
  ***********************************************************************************
**/
#ifndef SHAPE_H
#define SHAPE_H

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

//=====================================Base Class=====================================
class Shape{
    protected:
        int color;
        matrix point1;

        /**
         * @brief Assigns properties from the given shape to this shape
         *        Made protected so that the children of shape can't be set to
         *        eachother. A triangle should not converted into a line.
         *
         * @param rhs: The given shape to copy from
         *
         * @retval A copy of the given shape
         */
        virtual Shape &operator=(const Shape &rhs);

    public:

        /**
         * @brief Read shape properties from a text file (stream)
         *
         * @param is: Stream to read from
         *
         * @retval NONE
         */
        virtual std::istream &in(std::istream &is);

        /**
         * @brief Parameterized constructor, it creates a shape with a color.
         *
         * @param color_red: 3x8-bit value for red, green, blue
         *
         * @retval NONE
         */
        Shape(int color);

        /**
         * @brief Copy constructor that copies the paramters from the given shape
         *
         * @param from: shape to copy into the current shape.
         *
         * @retval NONE
         */
        Shape(const Shape &from);

        /**
         * @brief Virtual constructor thats used to copy a shape
         *
         * @param: NONE
         *
         * @retval NONE
         */
        virtual Shape *clone() = 0;

        /**
         * @brief Shape destructor, frees memory allocated to shape
         *        Not currently used due to image handling deletion
         *
         * @param: NONE
         *
         * @retval NONE
         */
        virtual ~Shape();

        /**
         * @brief Draws the given shape
         *
         * @param gc: GraphicsContext object that tells the shape where to draw
         *
         * @retval NONE
         */
        virtual void draw(GraphicsContext *gc) = 0;

        /**
         * @brief Print contents of shape into std.
         *        Method made const to prevent modifying when outputting
         *
         *        Shape_type
         *          Color: 0x......
         *          Point?: x y z
         *
         * @param os: Stream to write to
         *
         * @retval NONE
         */
        virtual std::ostream &out(std::ostream &os) const;
};

#endif
```

```cpp
/**
  ***********************************************************************************
  * @file   : shape.cpp
  * @brief  : Shape base class
  *         : Lab 6: Event Driven Drawing
  *         : CS-3210/021
  * @date   : APR 27 2021
  * @author : Julian Singkham
  ***********************************************************************************
  * @attention
  * Abstract base class for all types of shapes (currently line, triangle).
  * Shape houses the color and origin point for all its children since all shapes.
  * Shape functions are only ever called on when a child needs to modify/get
  * color or point1.
  ***********************************************************************************
**/
#include <sstream> //For String Stream

#include "shape.h"

//=====================================Protected=====================================
/**
 * @brief Assigns properties from the given shape to this shape
 *        Made protected so that the children of shape can't be set to
 *        eachother. A triangle should not converted into a line.
 *
 * @param rhs: The given shape to copy from
 *
 * @retval A copy of the given shape
 */
Shape &Shape::operator=(const Shape &rhs){
    //check if shape is being assigned it itself
    if(this != &rhs){
        color = rhs.color;
        point1 = matrix(rhs.point1);
    }
    return *this;
}
//======================================Public=======================================
/**
 * @brief Read line properties from a text file (stream)
 *
 * @param is: Stream to read from
 *
 * @retval NONE
 */
std::istream &Shape::in(std::istream &is){
    std::string Line;
    std::stringstream str_stream;

    //Copy Color
    std::getline(is, Line); //Read line
    str_stream = std::stringstream(Line);
    str_stream.ignore(32, ':');
    str_stream >> std::hex >> color;

    //Copy first point
    std::getline(is, Line); //Read line
    str_stream = std::stringstream(Line);
    str_stream.ignore(32, ':');
    str_stream >> point1[0][0];
    str_stream >> point1[1][0];
    str_stream >> point1[2][0];

    return is;
}
/**
 * @brief Parameterized constructor, it creates a shape with a color.
 *
 * @param color_red: 3x8-bit value for red, green, blue
 *
 * @retval NONE
 */
Shape::Shape(int color)
    : color(color), point1(5,5){
}

/**
 * @brief Copy constructor that copies the paramters from the given shape
 *
 * @param from: shape to copy into the current shape.
 *
 * @retval NONE
 */
Shape::Shape(const Shape &from)
    : color(from.color), point1(from.point1){
}

/**
 * @brief Line destructor, frees memory allocated to line
 *        Not currently used due to image handling deletion
 *
 * @param: NONE
 *
 * @retval NONE
 */
Shape::~Shape(){
}

/**
 * @brief Print contents of line into std.
 *        Method made const to prevent modifying when outputting
 *
 *        Shape_type
 *          Color: 0x......
 *          Point1: x y z
 *
 * @param os: Stream to write to
 *
 * @retval NONE
 */
std::ostream &Shape::out(std::ostream &os) const{
    os << "\tColor: 0x" << std::uppercase << std::hex << color << std::endl;

    os << "\tPoint 1: "
       << point1[0][0] << " "
       << point1[1][0] << " "
       << point1[2][0]
       << std::endl;

    return os;
}
```

```
/**
  ***********************************************************************************
  * @file    : line.h
  * @brief   : Outline for line shape class
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : APR 27 2021
  * @author  : Julian Singkham
  ***********************************************************************************
**/
#ifndef LINE_H
#define LINE_H

#include "shape.h"

//========================================Class========================================

class Line : public Shape{
    private:
        //points to draw to
        matrix point2;

         /**
         * @brief Constructor that makes a new line from a stream
         *        Made private so that only image can create triangles with a stream.
         *        Image will handle parsing through the file and determining what
         *        shape gets created.
         * @param is: Input stream that contains Line parameters
         *
         * @retval NONE
         */
        Line(std::istream &is);

        /**
         * @brief Read line properties from a text file (stream)
         *
         * @param is: Stream to read from
         *
         * @retval NONE
         */
        std::istream &in(std::istream &is);

    public:
        friend class Image; //Allows image access to the instream methods

        /**
         * @brief Parameterized constructor, it creates a Line with a color.
         *
         * @param color: 3x8-bit value for red, green, blue
         *
         * @retval NONE
         */
        Line(double x0, double y0, double x1, double y1, int color);

        /**
         * @brief Copy constructor that copies the paramters from the given line
         *
         * @param from: Line to copy into the current line.
         *
         * @retval None
         */
        Line(const Line &from);

        /**
         * @brief Virtual constructor thats used to copy a shape
         *
         * @param: NONE
         *
         * @retval NONE
         */
        Line *clone();

        /**
         * @brief Line destructor, frees memory allocated to line
         *        Not currently used due to image handling deletion
         *
         * @param: NONE
         *
         * @retval NONE
         */
        ~Line();

        /**
         * @brief Assigns properties from the given line to this line
         *
         * @param rhs: The given line to copy from
         *
         * @retval A copy of the given line
         */
        Line &operator=(const Line &rhs);

        /**
         * @brief Draws the given line
         *
         * @param gc: GraphicsContext object that tells the shape where to draw
         *
         * @retval NONE
         */
        void draw(GraphicsContext *gc);

        /**
         * @brief Print contents of line into std.
         *        Method made const to prevent modifying when outputting
         *
         *        Shape_type
         *          Color: 0x......
         *          Point?: x y z
         *
         * @param os: Stream to write to
         *
         * @retval NONE
         */
        std::ostream &out(std::ostream &os) const;
};


#endif
```

```cpp
/**
  ************************************************************************************
  * @file   : line.cpp
  * @brief  : line shape class
  *         : Lab 6: Event Driven Drawing
  *         : CS-3210/021
  * @date   : APR 27 2021
  * @author : Julian Singkham
  ************************************************************************************
  * @attention
  * Handles the creation of a line in 3-D space using x11 graphics.
  ************************************************************************************
**/
#include <sstream> //For String Stream

#include "line.h"
//======================================Private=======================================
/**
 * @brief Constructor that makes a new line from a stream
 *        Made private so that only image can create triangles with a stream.
 *        Image will handle parsing through the file and determining what
 *        shape gets created.
 * @param is: Input stream that contains Line parameters
 *
 * @retval NONE
 */
Line::Line(std::istream &is)
    : Shape(color), point2(5,5){

    in(is);
}

/**
 * @brief Read line properties from a text file (stream)
 *
 * @param is: Stream to read from
 *
 * @retval NONE
 */
std::istream &Line::in(std::istream &is){
    std::string str_line;
    std::stringstream str_stream;

    Shape::in(is); //Call parent first

    //Copy second point
    std::getline(is, str_line); //Read line
    str_stream = std::stringstream(str_line);
    str_stream.ignore(32, ':');
    str_stream >> point2[0][0];
    str_stream >> point2[1][0];
    str_stream >> point2[2][0];

    return is;
}

//=======================================Public=======================================

/**
 * @brief Parameterized constructor, it creates a Line with a color.
 *
 * @param color: 3x8-bit value for red, green, blue
 *
 * @retval NONE
 */
Line::Line(double x0, double y0, double x1, double y1, int color)
    : Shape(color), point2(5,5){

    //Copy origin point
    this->point1[0][0] = x0;
    this->point1[1][0] = y0;
    this->point1[2][0] = 0; //Default
    this->point1[3][0] = 1; //Default

    //Copy second point
    this->point2[0][0] = x1;
    this->point2[1][0] = y1;
    this->point2[2][0] = 0; //Default
    this->point2[3][0] = 1; //Default
}

/**
 * @brief Copy constructor that copies the paramters from the given line
 *
 * @param from: Line to copy into the current line.
 *
 * @retval None
 */
Line::Line(const Line &from)
    : Shape(from.color), point2(from.point2){

    point1 = matrix(from.point1);
}

/**
 * @brief Virtual constructor thats used to copy a shape
 *
 * @param: NONE
 *
 * @retval NONE
 */
Line *Line::clone(){
    return new Line(*this);
}

/**
 * @brief Line destructor, frees memory allocated to line
 *        Not currently used due to image handling deletion
 *
 * @param: NONE
 *
 * @retval NONE
 */
Line::~Line(){
}

/**
 * @brief Assigns properties from the given line to this line
 *
 * @param rhs: The given line to copy from
 *
 * @retval A copy of the given line
 */
Line &Line::operator=(const Line &rhs){
    //check if shape is being assigned it itself
    if(this != &rhs){
        color = rhs.color;
        point1 = matrix(rhs.point1);
        point2 = matrix(rhs.point2);
    }
    return *this;
}

/**
 * @brief Draws the given line
 *
 * @param gc: GraphicsContext object that tells the shape where to draw
 *
 * @retval NONE
 */
void Line::draw(GraphicsContext *gc){
    gc->setColor(color);
    gc->drawLine(point1[0][0], point1[1][0], point2[0][0], point2[1][0]);
}

/**
 * @brief Print contents of line into std.
 *        Method made const to prevent modifying when outputting
 *
 *        Shape type
 *          Color: 0x......
 *          Point?: x y z
 *
 * @param os: Stream to write to
 *
 * @retval NONE
 */
std::ostream &Line::out(std::ostream &os) const{
    os << "Line" << std::endl;
    Shape::out(os); //Call shape's printout first

    os << "\tPoint 2: "
       << point2[0][0] << " "
       << point2[1][0] << " "
       << point2[2][0]
       << std::endl;

    return os;
}
```

```
/**
  ***************************************************************************************
  * @file    : triangle.h
  * @brief   : Outline for triangle shape class
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : APR 27 2021
  * @author : Julian Singkham
  ***************************************************************************************
**/
#ifndef TRIANGLE_H
#define TRIANGLE_H

#include "shape.h"
//========================================Class========================================
class Triangle : public Shape{
    private:
        //points to draw to
        matrix point2, point3;

         /**
         * @brief Constructor that makes a new triangle from a stream
         *        Made private so that only image can create triangles with a stream.
         *        Image will handle parsing through the file and determining what
         *        shape gets created.
         *
         * @param is: Input stream that contains triangle parameters
         *
         * @retval NONE
         */
        Triangle(std::istream &is);

        /**
         * @brief Read triangle properties from a text file (stream)
         *
         * @param is: Stream to read from
         *
         * @retval NONE
         */
        std::istream &in(std::istream &is);

    public:
        friend class Image; //Allows image access to the instream methods

        /**
         * @brief Parameterized constructor, it creates a triangle with a color.
         *
         * @param color: 3x8-bit value for red, green, blue
         *
         * @retval NONE
         */
        Triangle(double x0, double y0, double x1, double y1, double x2, double y2, int col
or);

        /**
         * @brief Copy constructor that copies the paramters from the given triangle
         *
         * @param from: Triangle to copy into the current triangle.
         *
         * @retval None
         */
        Triangle(const Triangle &from);

        /**
         * @brief Virtual constructor thats used to copy a shape
         *
         * @param: NONE
         *
         * @retval NONE
         */
        Triangle *clone();

        /**
         * @brief Triangle destructor, frees memory allocated to triangle
         *        Not currently used due to image handling deletion
         *
         * @param: NONE
         *
         * @retval NONE
         */
        ~Triangle();

        /**
         * @brief Assigns properties from the given triangle to this triangle
         *
         * @param rhs: The given triangle to copy from
         *
         * @retval A copy of the given triangle
         */
        Triangle &operator=(const Triangle &rhs);

        /**
         * @brief Draws the given triangle
         *
         * @param gc: GraphicsContext object that tells the shape where to draw
         *
         * @retval NONE
         */
        void draw(GraphicsContext *gc);

        /**
         * @brief Print contents of triangle into std.
         *        Method made const to prevent modifying when outputting
         *
         *        Shape_type
         *          Color: 0x......
         *          Point?: x y z
         *
         * @param os: Stream to write to
         *
         * @retval NONE
         */
         std::ostream &out(std::ostream &os) const;
};

#endif
```

```
/**
  ************************************************************************************
  * @file    : triangle.cpp
  * @brief   : Triangle shape class
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : APR 27 2021
  * @author : Julian Singkham
  ************************************************************************************
  * @attention
  * Handles the creation of a triangle in 3-D space using x11 graphics.
  ************************************************************************************
**/
#include <sstream> //For String Stream

#include "triangle.h"
//======================================Private=======================================
/**
 * @brief Constructor that makes a new triangle from a stream
 *        Made private so that only image can create triangles with stream.
 *        Image will handle parsing through the file and determining what
 *        shape gets created.
 *
 * @param is: Input stream that contains triangle parameters
 *
 * @retval NONE
 */
Triangle::Triangle(std::istream &is)
    : Shape(color), point2(5,5), point3(5,5){

    in(is);
}

/**
 * @brief Read triangle properties from a text file (stream)
 *
 * @param is: Stream to read from
 *
 * @retval NONE
 */
std::istream &Triangle::in(std::istream &is){
    std::string line;
    std::stringstream str_stream;

    Shape::in(is); //Call parent first

    //Copy second point
    std::getline(is, line); //Read line
    str_stream = std::stringstream(line);
    str_stream.ignore(32, ':');
    str_stream >> point2[0][0];
    str_stream >> point2[1][0];
    str_stream >> point2[2][0];

    //Copy third point
    std::getline(is, line); //Read line
    str_stream = std::stringstream(line);
    str_stream.ignore(32, ':');
    str_stream >> point3[0][0];
    str_stream >> point3[1][0];
    str_stream >> point3[2][0];

    return is;
}
//=======================================Public=======================================

/**
 * @brief Parameterized constructor, it creates a triangle with a color.
 *
 * @param color: 3x8-bit value for red, green, blue
 *
```

```
  * @retval NONE
  */
 Triangle::Triangle(double x0, double y0, double x1, double y1, double x2, double y2,
                     int color) : Shape(color), point2(5,5), point3(5,5){

     //Copy origin point
     this->point1[0][0] = x0;
     this->point1[1][0] = y0;
     this->point1[2][0] = 0; //Default
     this->point1[3][0] = 1; //Default

     //Copy second point
     this->point2[0][0] = x1;
     this->point2[1][0] = y1;
     this->point2[2][0] = 0; //Default
     this->point2[3][0] = 1; //Default

     //Copy third point
     this->point3[0][0] = x2;
     this->point3[1][0] = y2;
     this->point3[2][0] = 0; //Default
     this->point3[3][0] = 1; //Default
 }

 /**
  * @brief Copy constructor that copies the paramters from the given triangle
  *
  * @param from: Triangle to copy into the current triangle.
  *
  * @retval None
  */
 Triangle::Triangle(const Triangle &from)
     : Shape(from.color), point2(from.point2), point3(from.point3){

     point1 = matrix(from.point1);
 }

 /**
  * @brief Virtual constructor thats used to copy a shape
  *
  * @param: NONE
  *
  * @retval NONE
  */
 Triangle *Triangle::clone(){
     return new Triangle(*this);
 }

 /**
  * @brief Triangle destructor, frees memory allocated to triangle
  *        Not currently used due to image handling deletion
  *
  * @param: NONE
  *
  * @retval NONE
  */
 Triangle::~Triangle(){
 }

 /**
  * @brief Assigns properties from the given triangle to this triangle
  *
  * @param rhs: The given triangle to copy from
  *
  * @retval A copy of the given triangle
  */
 Triangle &Triangle::operator=(const Triangle &rhs){
     //check if shape is being assigned it itself
     if(this != &rhs){
         color = rhs.color;
         point1 = matrix(rhs.point1);
```

```
        point2 = matrix(rhs.point2);
        point3 = matrix(rhs.point3);
    }
    return *this;
}

/**
 * @brief Draws the given triangle
 *
 * @param gc: GraphicsContext object that tells the shape where to draw
 *
 * @retval NONE
 */
void Triangle::draw(GraphicsContext *gc){
    gc->setColor(color);
    gc->drawLine(point1[0][0], point1[1][0], point2[0][0], point2[1][0]);
    gc->drawLine(point2[0][0], point2[1][0], point3[0][0], point3[1][0]);
    gc->drawLine(point3[0][0], point3[1][0], point1[0][0], point1[1][0]);
}

/**
 * @brief Print contents of triangle into std.
 *        Method made const to prevent modifying when outputting
 *
 *        Shape_type
 *          Color: 0x......
 *          Point?: x y z
 *
 * @param os: Stream to write to
 *
 * @retval NONE
 */
std::ostream &Triangle::out(std::ostream &os) const{
    os << "Triangle" << std::endl;
    Shape::out(os); //Call shape's printout first

    os << "\tPoint 2: "
       << point2[0][0] << " "
       << point2[1][0] << " "
       << point2[2][0]
       << std::endl;

    os << "\tPoint 3: "
       << point3[0][0] << " "
       << point3[1][0] << " "
       << point3[2][0]
       << std::endl;

    return os;
}
```

```
/**
  ***********************************************************************************
  * @file    : image.h
  * @brief   : Outline for image container class
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : APR 27 2021
  * @author  : Julian Singkham
  ***********************************************************************************
**/
#ifndef IMAGE_H
#define IMAGE_H

#include <vector> //Shape verticies are stored in a vector

#include "shape.h"
#include "triangle.h"
#include "line.h"

//=======================================Class==========================================
class Image{
    private:
        std::vector<Shape *> shapes; //List of shapes in the container

    public:
        /**
         * @brief Constructor
         *
         * @param: NONE
         *
         * @retval NONE
         */
        Image();

        /**
         * @brief Copy constructor that copies the contents from the given image
         *
         * @param from: Image to copy into the current image.
         *
         * @retval NONE
         */
        Image(const Image &from);

        /**
         * @brief Image destructor, frees memory allocated to image
         *
         * @param: NONE
         *
         * @retval NONE
         */
        ~Image();

        /**
         * @brief Delete all shapes within the image
         *
         * @param: NONE
         *
         * @retval NONE
         */
        void erase();

        /**
         * @brief Assigns the image to another image
         *
         * @param rhs: The given image to copy from
         *
         * @retval A copy of the given image
         */
        Image &operator=(const Image &rhs);

        /**
```

```
         * @brief Adds a shape to the container
         *
         * @param shape: Shape to add
         *
         * @retval NONE
         */
        void add(Shape *shape);

        /**
         * @brief Draws shapes in the image
         *        Method made const to prevent modifying when outputting
         *
         * @param gc: GraphicsContext object that tells the shape where to draw
         *
         * @retval NONE
         */
        void draw(GraphicsContext *gc) const;

        /**
         * @brief Print contents of image into std.
         *        Method made const to prevent modifying when outputting
         *
         * @param os: Stream to write to
         *
         * @retval NONE
         */
        std::ostream &out(std::ostream &os) const;

        /**
         * @brief Read shape properties from a text file (stream)
         *
         * @param is: Stream to read from
         *
         * @retval NONE
         */
        std::istream &in(std::istream &is);

        /**
         * @brief Shapes vector getter
         *
         * @param: NONE
         *
         * @retval Shapes vector
         */
        std::vector<Shape *> get_shapes();
};

#endif
```

```
/**
  ***************************************************************************************
  * @file    : Image.cpp
  * @brief   : Image container class
  *          : Lab 6: Event Driven Drawing
  *          : CS-3210/021
  * @date    : APR 27 2021
  * @author : Julian Singkham
  ***************************************************************************************
  * @attention
  * The image class is a container for shapes. Think of image as a frame and shapes
  * are added to the frame to be displayed on the monitor.
  * When creating shapes with a stream, image must be called so that it can determine
  * what shapes the parameters belong to.
  ***************************************************************************************
**/
#include <sstream> //For String Stream

#include <string>

#include "image.h"
//=======================================Public=======================================
/**
 * @brief Constructor
 *
 * @param: NONE
 *
 * @retval NONE
 */
Image::Image(){}

/**
 * @brief Copy constructor that copies the contents from the given image
 *
 * @param from: Image to copy into the current image.
 *
 * @retval NONE
 */
Image::Image(const Image &from){
    for (Shape *i : from.shapes)
        add((i)->clone());
}

/**
 * @brief Image destructor, frees memory allocated to image
 *
 * @param: NONE
 *
 * @retval NONE
 */
Image::~Image(){
    erase();
}

/**
 * @brief Delete all shapes within the image
 *
 * @param: NONE
 *
 * @retval NONE
 */
void Image::erase(){
    for (Shape *i : shapes)
        delete i;
    shapes.clear();
}

/**
 * @brief Assigns the image to another image
 *
 * @param rhs: The given image to copy from
 *
 * @retval A copy of the given image
 */
Image &Image::operator=(const Image &rhs){
    //check if image is being assigned it itself
    if(this != &rhs){
        shapes.clear();
        for (Shape *i : rhs.shapes)
            add((i)->clone());
    }
    return *this;
}

/**
 * @brief Adds a shape to the container
 *
 * @param shape: Shape to add
 *
 * @retval NONE
 */
void Image::add(Shape *shape){
    shapes.push_back(shape);
}

/**
 * @brief Draws tall shapes in the image
 *        Method made const to prevent modifying when outputting
 *
 * @param gc: GraphicsContext object that tells the shape where to draw
 *
 * @retval NONE
 */
void Image::draw(GraphicsContext *gc) const{
    for (Shape *i : shapes)
        (i)->draw(gc);
}

/**
 * @brief Print contents of image into std.
 *        Method made const to prevent modifying when outputting
 *
 * @param os: Stream to write to
 *
 * @retval NONE
 */
std::ostream &Image::out(std::ostream &os) const{
    for (Shape *i : shapes)
        i->out(os);
    return os;
}

/**
 * @brief Read shape properties from a text file (stream)
 *
 * @param is: Stream to read from
 *
 * @retval NONE
 */
std::istream &Image::in(std::istream &is){
    std::string str_line;
    while(std::getline(is, str_line)){
        if (str_line.rfind("Line", 0) == 0)
            add(new Line(is));
        else if (str_line.rfind("Triangle", 0) == 0)
            add(new Triangle(is));
        else
            std::cout << "Unable to read line, Skipping" << std::endl;
    }

    return is;
}
```

```

/**
 * @brief Shapes vector getter
 *
 * @param: NONE
 *
 * @retval Shapes vector
 */
std::vector<Shape *> Image::get_shapes(){
    return shapes;
}
```

Table of Content

Table of Contents