ECE560 Computer Graphics Final Project Report

Michael Hanley

***Abstract*—This project aims to create a powerful tool for creating images and graphics, "drawings," using a large library of 2D and 3D graphical primitives. Drawings created with this tool can be saved to a "DRW" (draw) file, a new, binary file format that stores data in a de-interleaved fashion to optimize compression. DRW files can be loaded into a program that renders the stored drawings via OpenGL (Open Graphics Library). OpenGL ES (Open Graphics Library for Embedded Systems) is used, allowing portability across embedded systems such as Raspberry Pi 5 computers, and possibly Android devices. All of the initially desired 2D graphical primitives have been implemented; these primitives include Bézier and cubic spline curves, which can effectively create any desired 2D shape. All but one initially desired 3D graphical primitive have been implemented, and all 3D graphical primitives support lighting. SVG (Scalable Vector Graphics) files can be automatically converted to DRW files. Supported SVG elements include: <line>, <circle>, <ellipse>, <rect>, <polyline>, <polygon>, <path>, and <g>. So long as an SVG file only contains supported elements and attributes, the DRW file created from conversion generally results in an identical picture when rendered in OpenGL, with some noted exceptions. With future development, support for other SVG elements can be added, full consistency between SVG and DRW can be ensured, and compression can be optimized.**

I. Introduction

A. Goal

a) Main Features: The ultimate goal of this project is to create a library of 2D and 3D graphical primitives that can be used to create a wide array of images and graphics. From this point forward, these images and graphics will be referred to as "drawings." Drawings can be saved to and loaded from a "DRW" (draw) file. The full list of desired 2D primitives is as follows: point, line, rectangle, square, rounded rectangle, triangle, circle, ellipse, polygon, polyline, polygon/polyline with markers at the control points, grid, Bézier curve, cubic spline curve, and cubic spline curve with markers at the control points. Any 2D primitives that are completed shapes should also have a filled in form, as well as an outline form. The full list of desired 3D primitives is as follows: rectangular prism, cube, rectangular pyramid, wedge, cylinder, cone, frustum, tube (i.e. a cylinder minus a cylinder, or a frustum minus a frustum), oblique cone, torus, helix, ellipsoid, UV sphere, icosphere, linearly extruded 2D shape, and 2D shape extruded through a (cubic spline) curve. All 3D primitives should have a wireframe form, a filled in form, and a filled in form with defined normals, such that lighting can be applied. The types of supported lighting should include point lights, directional lights, and spot lights.

b) SVG Conversion: An additional major goal of this project is to be able to convert an SVG (Scalable Vector Graphics) file to a DRW file. While it is infeasible to implement support for every SVG element within the allotted development time, the following elements are priority: <line>, <circle>, <ellipse>, <rect>, <polyline>, and <polygon>. Unsupported elements or attributes may simply be ignored during conversion. So long as the SVG file does not contain any of said unsupported features, the resultant DRW file from conversion should ideally create an identical picture to that of the SVG file when loaded and rendered. By converting an SVG file to a DRW file, one gains the ability to render SVG files in OpenGL (Open Graphics Library), and the DRW files will ideally offer better compression and thus smaller file sizes than SVG.

B. Implementation

a) Graphics: As mentioned, the rendering of DRW files is performed via OpenGL. When a DRW file is loaded, a window is created via SDL2 (Simple DirectMedia Layer 2) that is the size and dimensions of the drawing (unless the drawing is too large to fit on the screen, in which case the window is sized to fit the screen). The window may be resized, and the drawing will scale appropriately with the window.

b) File Structure: DRW is a binary file format. The information in DRW files is stored de-interleaved (e.g. all x-coordinates are stored together, in sequence, and all y-coordinates are stored together, in sequence). Storing information in this manner should optimize LZMA's (the Lempel–Ziv–Markov chain algorithm) ability to compress the files [1]. At the beginning of each DRW file is a header, which includes a magic number for identifying the file type. Following the header is a sequence of binary numbers that represent the drawing instructions (e.g. DRAW_CIRCLE = 00000000, DRAW_TRIANGLE == 00000001, etc.). The remaining, de-interleaved sections of the file, in order, are: miscellaneous integer values, x-coordinates, y-coordinates, z-coordinates, miscellaneous floating point values, RGB (red, green, blue) color values, and transformation matrix values. The header also stores the sizes of each of these sections, allowing each of them to be correctly accessed. Data is also delta encoded, meaning the file is storing the differences (deltas) between values rather than the actual values themselves, to further improve compression.

c) Parsing: The parsing of SVG files for conversion to DRW files is accomplished using ANTLR v4 (ANother Tool for Language Recognition Version 4). ANTLR v4 allows one to define a grammar and then automatically convert that grammar into a series of .cpp files (or other language files) that can be used to parse and traverse text. Because SVG is an XML-based (Extensible Markup Language) format, an XML parser is more than sufficient for parsing SVG files.

II. Prior Work

A. Graphics

This project heavily uses the "MultiShape" concept from Professor Dov Kruger's "GrailGUI" project [2]. The idea behind the MultiShape concept is to render a large quantity of shapes with only a few OpenGL draw calls by storing all of the shapes' vertex and index information together. Limiting the number of draw calls limits the number of data transfers between the CPU and the GPU, theoretically greatly improving overall rendering speed. The "multishape" class in the DRW project, like the "MultiShape" class in GrailGUI, inherits from a base "shape" class, which is the parent class of anything that is drawn to the screen. A multishape object contains five dynamic arrays, one for vertices, one for line indices, two for solid indices, and one for point indices. When a multishape is rendered, all of the solid shapes can be rendered with a single call to glDrawElements using GL_TRIANGLES mode, all of the lines can be rendered with one call using GL_LINES mode, and all of the individual points can be rendered with one call using GL_POINTS mode. If solid shapes then need to be drawn on top of the lines, an additional call using GL_TRIANGLES mode can be made, hence why multishape needs two arrays of solid indices (one for solids that are drawn below lines and one for solids that are drawn on top of lines). With this structure, even millions of shapes can be rendered with a maximum of only four calls to glDrawElements. The multishape class in the DRW project has four child classes, multishape_2d, styled_multishape_2d, multishape_3d, and lit_multishape_3d. The multishape_2d and styled_multishape_2d classes are expanded versions of the MultiShape2D and StyledMultiShape2D classes from GrailGUI, whereas the multishape_3d and lit_multishape_3d classes merely borrow the concept of the multishape with entirely new code. The multishape_2d class is used for drawing 2D graphical primitives, where each vertex consists only of an x-coordinate and a y-coordinate. The color of the shapes is supplied as a uniform, meaning every shape in a single multishape_2d object must be drawn in the same color. The styled_multishape_2d class stores an x-coordinate, a y-coordinate, and RGBA (red, green, blue, alpha) values within every vertex, allowing different shapes to have different colors within the same multishape. The multishape_3d class is used for drawing 3D graphical primitives, where each vertex consists of an x-coordinate, a y-coordinate, and a z-coordinate. The lit_multishape_3d class is used for drawing 3D graphical primitives with lighting, and adds a normal vector (nx, ny, nz) to every vertex of x, y, and z.

B. Parser

To parse SVG files, an existing XML parser, which was derived from an example in the ANTLR v4 reference guide book, was used [3]. The existing .g4 lexer and parser files were used, via ANTLR v4, to automatically create .cpp and .h files that allow for easy traversal of an SVG file's elements and attributes.

C. Block Loader

This project utilizes Professor Dov Kruger's "block_loader" class, which streamlines saving and loading blocks of binary data to and from files.

III. Results

A. Accomplishments

a) Graphics: The final list of implemented, 2D graphical primitives is as follows: point, line, rectangle, square, rounded rectangle, triangle, circle, ellipse, polygon, polyline, polygon/polyline with markers at the control points, grid, Bézier curve, cubic spline curve, and cubic spline curve with markers at the control points. The final list of implemented, 3D graphical primitives is: rectangular prism, cube, rectangular pyramid, wedge, cylinder, cone, frustum, tube (i.e. a cylinder minus a cylinder, or a frustum minus a frustum), oblique cone, oblique cylinder, oblique frustum, torus, helix, ellipsoid, UV sphere, linearly extruded 2D shape, and 2D shape extruded through a (cubic spline) curve. The oblique cylinder and oblique frustum were added despite not being in the initial list, and the icosphere is the only primitive from the initial list that was not implemented. Fig. 1. displays the majority of the 3D primitives in wireframe form. All of the 3D graphical primitives have lighting support, with point lights, directional lights, and spot lights, as partially demonstrated in Fig. 2.

b) SVG Conversion: The full list of SVG elements that are at least partially supported when converted to DRW is: <line>, <circle>, <ellipse>, <rect>, <polyline>, <polygon>, <path>, and <g>. In addition to their required attributes, each of these elements also supports the following: fill, stroke, fill-opacity, stroke-opacity, stroke-width, and transform. The major exceptions are that the <path> and <polygon> elements do not currently support the fill and fill-opacity attributes, as filling the inside of an arbitrary path in OpenGL requires tessellation, which I have not yet learned how to implement. The <rect> element has additional optional attributes, rx and ry, which are supported, and allow for the drawing of rounded rectangles. Fig. 3 displays a side-by-side comparison of an example SVG file and its corresponding DRW file after conversion.

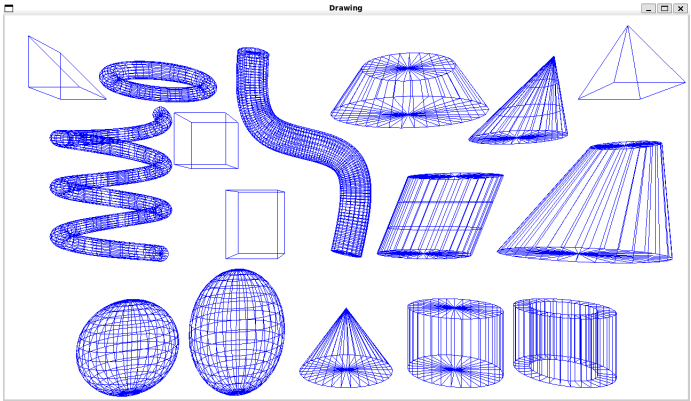

Fig. 1. 3D Graphical Primitives As Wireframes

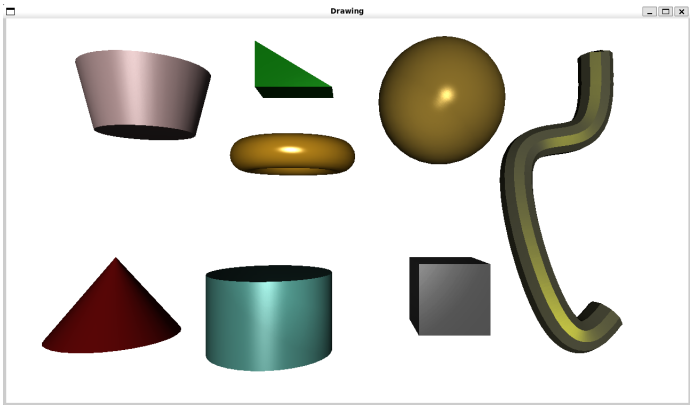

Fig. 2. 3D Graphical Primitives With Lighting

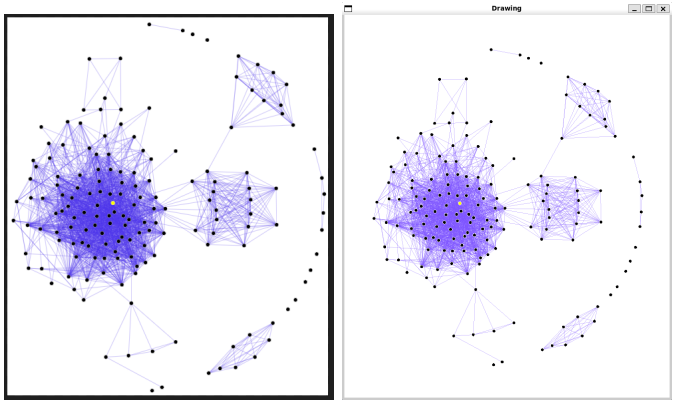

Fig. 3. Side-by-side comparison of a diagram of a social network (SVG left, DRW right) [4]

B. Limitations and Challenges

a) Inconsistencies: There are some cases where an SVG file that only contains supported elements is not drawn identically when converted and rendered as a DRW file. For example, the exact colors often do not match when dealing with semitransparent shapes, as the alpha blending function that the project currently uses with OpenGL is not identical to the function used by SVG viewers. This inconsistency is partially noticeable in Fig. 3, as the purple lines, which are semitransparent, appear darker in the SVG case. It is likely possible to utilize the correct blending function in OpenGL with the right glBlendFunc parameters, but this will take some additional time and research to achieve. Another inconsistency may arise when shapes have a stroke-width greater than 1, as OpenGL does not specify how to connect lines of width greater than 1. An example of this issue can be seen in Fig. 4, where the corners of a rectangle with a thick outline are inconsistent. Rectifying this issue would require replacing any thick line with an appropriately sized rectangle (or, more accurately, replacing any thick line with two triangles that form an appropriately sized rectangle).

b) Compression: In their current state, DRW files are not consistently smaller than SVG files. There are several ways

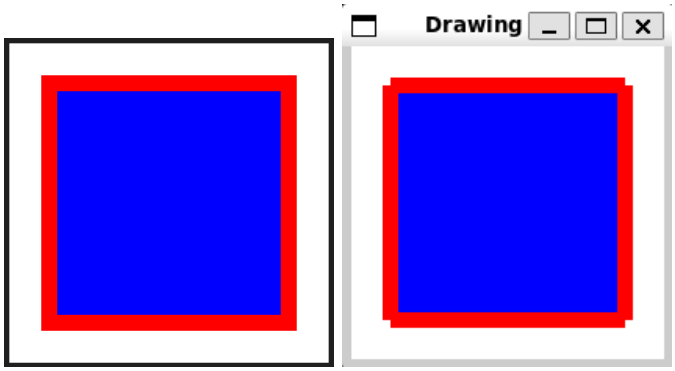

Fig. 4. Side-by-side comparison of a rectangle with a thick outline (SVG left, DRW right)

that, with future development, DRW files can be made significantly smaller, such that they will be more compressed than their SVG counterparts, consistently. For instance, multishapes that only consist of a singular shape require three instructions (START_MULTISHAPE, the instruction for the shape itself, and END_MULTISHAPE), whereas an equivalent shape in SVG only requires one instruction. Removing the necessity of these extra instructions could dramatically decrease the size of DRW files that contain many such multishapes. Additional techniques, such as discretization, may also make DRW files smaller, if implemented.

IV. Conclusion

The DRW file and its accompanying rendering program are capable of storing and rendering a large array of graphics with ease, using its large library of built-in primitives. DRW supports some of SVG's most important drawing elements, meaning many SVG files may be converted to DRW without loss of information. With additional development, DRW can become just as powerful as SVG with respect to 2D graphics, and have the added benefits of 3D graphics support and generally smaller file sizes.

References

[1] D. Kruger et al. unpublished.
[2] D. Kruger et al. GrailGUI, 2020. [Online; accessed 11-May-2025].
[3] Terence Parr. XML Parser, 2013. [Online; accessed 11-May-2025].
[4] Wikimedia Commons. File:social network diagram (large).svg — wikimedia commons, the free media repository, 2024. [Online; accessed 11-May-2025].