International Journal of Scientific Engineering and Science
Volume 5, Issue 4, pp. 36-39, 2021. ISSN (Online): 2456-7361

Converting SVG to G-code for 3D Printers

Kuo-pao Yang[1], Ghassan Alkadi[2], Terry Parker[3]
[1, 2, 3]Computer Science Department, Southeastern Louisiana University, Hammond, Louisiana, USA

Abstract**—** *This paper describes a research project that converts Scalable Vector Graphics (SVG) directly to G-code for three-dimensional printers. An SVG file uses coordinates to create an image in XML format, and a G-code file is readable by a 3D printer. By taking an SVG file and then converting it to a G-code, an image can be printed by our custom-built 3D printer. This project is implemented by the Rust programming language, and the Turtle Graphics package is used to assist in the translation of SVG to G-code.*

Keywords**—** *G-code, Rust, SVG, Turtle Graphics.*

I. INTRODUCTION

3D printing has become very popular in recent years. It is the process of making a three-dimensional object of virtually any shape from a digital model. They are becoming more commonly used within homes as well. It also provides an opportunity to create personalized 3D objects from a picture. This project allows users to easily create 3D printable objects by just using pictures taken from their phone. There is no need to worry about how to use a specific program or learn how to use a Computer Aided Design (CAD) system.

A great diversity of CAD systems can be found in the market today. A CAD system is a versatile tool that can be used to create, manipulate, and optimize objects, pictures, or other designs [4]. These are commonly used for laser engravers, embroideries, and 3D printers. However, they need the files to be in specific formats, otherwise the user would have to create it from scratch. Even with the proper files, the CAD systems do not create files that work directly with 3D printers. There are solutions by other languages to convert Scalable Vector Graphics (SVG) [12] to G-code such as C language, but Rust programming language is preferred because it is memory safe and more efficient in accomplishing this task.

Our application converts an SVG file to G-code for three-dimensional printing within seconds. Converting SVG to G-code is completed by the implementation of the Rust programming language. This project's application takes in an SVG file and generates a G-code for a 3D printer to print objects using the Turtle Graphics package. Rust handles its memory more efficiently since it is managed by its system of ownership at compile time [3]. This makes our application process these images faster than other programming languages or CAD programs.

In the following sections, we review the related work on 3D printing. Then, this paper describes the implementation to perform the functionality for the 3D printing. Furthermore, we discuss the methods and procedures used to complete this project along with how challenges are addressed. Finally, we provide the project evaluation, conclusion, and future areas of research as well.

II. RELATED WORK

A CAD program usually manages the conversion of images to other file types. These specific programs operate off only specified items. For example, the Tinkercad [9] is a simple CAD system for 3D printing. The user can import OBJ and SVG files, but Tinkercad has a size limitation of 25MB. This is not an issue in our application because it can process any size of SVG files. The user inputs an SVG file that is converted to a Stereolithography (STL) file through the Tinkercad process. After using the Tinkercad, the user has to go through another step for 3D printing. Then, the user needs a slicer program to convert the STL file to G-code that is readable by a 3D printer [2]. Although Tinkercad is a phenomenal program to create STL files, it does not create the G-code necessary for 3D printers. A separate program is required to convert STL files to G-code for 3D printing.

A STL file is a very common 3D file type developed in 1986 by Chuck Hull of 3D System. STL is typically employed as an output model of a CAD system and describes 3D objects in mathematical terms. These descriptions store and transfer data about the file model in the form of geometric shapes, which join to form the subject's surface. For 3D printing, the STL file needs to be sliced into G-code layers. G-code files tell a 3D printer how to print a 3D object. Furthermore, G-code typically describes printer parameters, such as speed and temperature, as well as the geometry of the 3D object.

Once the design is finalized and CAD systems output the STL file, the slicer can convert STL files to G-code [1]. The most notable slicers are Cura, Slic3r, and Simplify 3D. The slicer converts STL files into a series of thin layers and produces a G-code file. Furthermore, it divides the object into a stack of flat layers and describes these layers as linear movements of the 3D printer extruder, fixation laser, or equivalent. After the G-code is generated, it can be sent to the printer to build the 3D object that suits the user's needs [10].

Our project's application is designed to resolve Tinkercad and a slicer into a single program. This allows the user to input an SVG file and retrieve a G-code that is printer-ready all within a few seconds. This cuts down the time and process significantly and simplifies it to just using one program as well.

International Journal of Scientific Engineering and Science
Volume 5, Issue 4, pp. 36-39, 2021. ISSN (Online): 2456-7361

III. IMPLEMENTATION

The software approach was done through Ray tracing [8]. This project creates images through coordinates and matrices within Rust. Fig. 1 shows the code in the Rust programming language that does a simple Ray Tracer to read coordinates to create an image.

```
pub struct Ray {
    pub origin: Point,
    pub direction: Vector3,
}

impl Ray {
    pub fn create_prime(x: u32, y: u32, scene: &Scene) ->
      Ray {
        assert!(scene.width >= scene.height);
        let fov_adjustment = (scene.fov.to_radians() /
            2.0).tan();
        let aspect_ratio = (scene.width as f64) /
            (scene.height as f64);
        let sensor_x = ((((x as f64 + 0.5) / scene.width
            as f64) * 2.0 - 1.0) * aspect_ratio) *
            fov_adjustment;
        let sensor_y = (1.0 - ((y as f64 + 0.5) /
            scene.height as f64) * 2.0) *
            fov_adjustment;

        Ray {
            origin: Point::zero(),
            direction: Vector3 {
                    x: sensor_x,
                    y: sensor_y,
                    z: -1.0,
                }
                .normalize(),
        }
    }
}
```

Fig. 1. The basis of a Ray Tracer in the Rust programming language

The path tracers can produce 3D images but take a long time. This time-consuming problem is necessary for CPU-accelerated process to handle the computing power and calculations. It can take half an hour for a simple image to complete path tracing. To resolve this problem, this project developed applications that converts SVG file to G-code by giving the object depth and volume.

With the Turtle Graphics package, this project successfully implemented the Rust programming language [7]. The developed program directly converts SVG to G-code files [5]. The basic structure of the Turtle Graphics used within our program is shown in Fig. 2.

By implementing the Turtle Graphics and machine settings codes, the created G-code can be read with our custom-built 3D printer [11]. Using vectors, the printer can process the coordinates that Turtle Graphics converted from the original SVG file. Then, these vectors break down to output the G-code file that can then be sent straight to the users' 3D printer to be printed. To implement this software, this program determines how the image can be drawn. To accomplish this task, the path segments for the image were translated and then converted to G-code.

In Fig. 3, giving specific coordinates and using a stack of 't' we can keep up with the different movements from top to bottom in an SVG file and recreate those images from top to bottom in G-code so that the 3D printer can recognize and print this file. The Cubic Bezier curve allows the image to create smooth surfaces by using four points on a coordinate to control the curve. Therefore, this program can create smooth curves that are needed for images. Alternatively, non-Bezier curves are used to generate rigid curves for objects. Different path segments are used to translate and create any image.

```
pub struct Turtle {
    curpos: F64Point,
    initpos: F64Point,
    curtran: Transform2D<f64>,
    scaling: Option<Transform2D<f64>>,
    transtack: Vec<Transform2D<f64>>,
    pub mach: Machine,
    prev_ctrl: Option<F64Point>,
}

impl Default for Turtle {
    fn default() -> Self {
        Self {
            curpos: point(0.0, 0.0),
            initpos: point(0.0, 0.0),
            curtran: Transform2D::identity(),
            scaling: None,
            transtack: vec![],
            mach: Machine::default(),
            prev_ctrl: None,
        }
    }
}
```

Fig. 2. The basic structure of the Turtle Graphics used within our project

```
PathSegment::CurveTo {
   abs, x1, y1, x2, y2, x, y,
} => t.cubic_bezier(
  *abs, *x1, *y1, *x2, *y2, *x, *y,
  opts.tolerance,
  None,
  opts.feedrate,
),
PathSegment::SmoothCurveTo {
   abs, x2, y2, x, y
} => t.smooth_cubic_bezier(
  *abs, *x2, *y2, *x, *y,
  opts.tolerance,
  None,
  opts.feedrate,
),
PathSegment::Quadratic {
   abs, x1, y1, x, y
} => t.quadratic_bezier(
  *abs, *x1, *y1, *x, *y,
  opts.tolerance,
  None,
  opts.feedrate,
),
PathSegment::SmoothQuadratic {
   abs, x, y
} => t.smooth_quadratic_bezier(
  *abs, *x, *y,
  opts.tolerance,
  None,
  opts.feedrate,
),
```

Fig. 3. Different path segments needed to translate and create any images

Finally, the application program in Fig. 4 creates the settings for the 3D printer that is stored at the beginning of the G-code for the printer to be able to execute the code properly. These are all maintained and can be edited within the code for the specific printer at use. In addition, our program can even create G-code for laser engravers.

International Journal of Scientific Engineering and Science
Volume 5, Issue 4, pp. 36-39, 2021. ISSN (Online): 2456-7361

```
G28                          ; Home extruder
G1 Z15 F240
M107              ; Turn off fan
G90                          ; Absolute positioning
M82                          ; Extruder in absolute mode
M190 S50 ; Activate all used extruder
M104 T0 S210
G92 E0            ; Reset extruder position
; Wait for all used extruders to reach temperature
M109 T0 S210
; Layer count: 3
; Layer: 0
M107
G0 F9000 X77.560 Y68.534 Z0.300
```

Fig. 4. The settings for the 3D printer

Our custom-built 3D printer is the main hardware equipment involved in this research project. The printer's extruder is run by a stepper and extrudes the material that is used to create the object. The extruder is on a three-axis pivot structure, which controls where the print is positioned on the hot bed. The components that make up the extruder are the stepper motor, quick release, heating element, heat sink, and fan shown in Fig. 5.

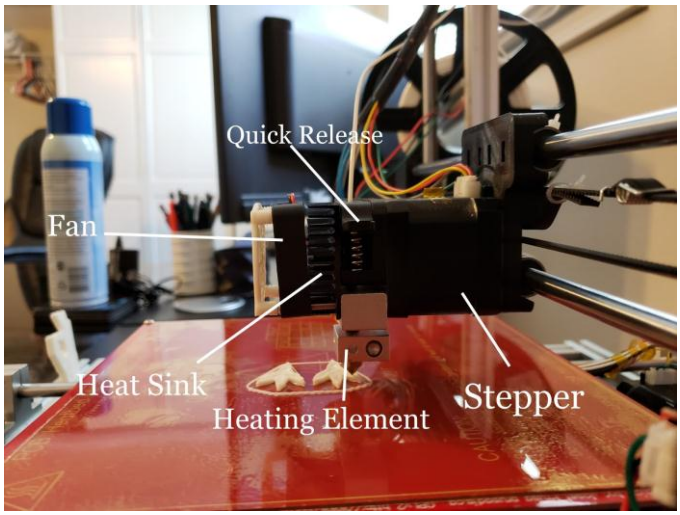

Fig. 5. Labeled extruder: stepper, quick release, heating element, heat sink, and fan

The logic components are used to power and control the parts on the 3D printer. This printer uses a Reprap Arduino Mega Pololu Shield (RAMPS) board to power the components. Stepper motors, thermistors, and heating elements are plugged into the RAMPS board. The converted G-code is stored onto an SD card, which can be read by the card reader and displayed on the LCD screen of our customized printer. Then, the 3D printer can read G-code straight from the SD card.

The settings of the 3D printer are all managed through the Repetier Server. Various printer properties can be modified using this program, including the heat of extruder, heat of the bed, extruder extrusion rate, and the tolerance level of the axis speed rate. This Repetier Server image serves as a web interface to control the printer from anywhere with an internet connection. A face printed by the 3D printer is shown in Fig. 6.

Fig. 6. A face printed by the 3D printer

IV. Evaluation

The initial test for our software and hardware was not fruitful at first. We tested the software with basic SVG files that are common for embroidery hardware and also have cleanly drawn lines. The software converted these objects to G-code with few issues. The main problem was tweaking the size and editing the machine settings for the G-code to align appropriately. Initially, the program converted SVG into a G-code file that was too large to be printed on the bed of the printer. Once the SVG files of embroidery objects worked, we moved on to images that have been converted to SVG. Within a second, the software converts SVG files to G-code, which can be stored on an SD card to be printed.

Our solution performed even faster than expected, especially because the original Ray Tracer took about fifteen minutes just to create a plane and three spheres before even converting any files to G-code. The performance of our solution is quicker than other languages, and transfers faster than CAD systems. First, CAD systems must spend a significant amount of time compiling the file to an STL file. Then, a slicer can convert the STL file to a printable G-code. These methods take much longer to convert an SVG file to a printable file for 3D printers [6].

The solutions proposed by previous methods have multiple complicated steps. Our solution simplifies this by setting up a program to convert SVG file straight to G-code, instead of converting SVG to STL and finally converting to G-code via a separate program.

The degree of success and speed make our method a better solution to converting SVG files to printable 3D objects, especially since it only takes the program around a second to create a printable file. Furthermore, the ability to be able to accomplish everything within one location for a user makes this project even more impressive. Besides user convenience, the rapid speed of completion is another great feature of this project.

V. Concusion and Future Work

This research project originated to solve the problem of printing SVG files to a 3D printer. The main challenges users

face is having to learn different software and being able to modify the files to be compatible with 3D printers. It is inconvenient and time-consuming for the user, who must figure out how to use those programs and also spend more time converting files through at least two different programs. With this project, the user only needs to input the desired SVG file, and the program will output the printable G-code. Not only does this have everything in the one project, but it is done in under a second as well. This is the main reason this solution is better than the others.

In the future, this project plans to process not only SVG but also other image files like JPG and PNG. Furthermore, our next goal is to take in a folder of files from all around an object and use Path Tracing to render a full 3D print of an object. This will be accomplished through GPU-Accelerated programming. Currently, the Nividia's CUDA software can accomplish this task. At the time of creating this project, the RustaCUDA package was still under development. We look forward to implementing this package to explore further tasks. Our solution is just the stepping-stone to incorporating the file reading conversion to G-code. Through RustaCUDA and path tracing, we will be able to combine these technologies for quick and efficient conversion of any objects. This project can be applied to laser engraving or, in fact, any other objects that uses G-code. Those can already be taken care by the solution we have now. There is also the possibility of copying organs through this project and having a 3D printer that prints with tissue. This could create great benefits for the medical field.

REFERENCES

[1] M. S. Akter and M. H. Kabir, "Temperature Optimization of RepRap (Replicating Rapid-prototyper) 3D Printer," 2018 International Conference on Computer, Communication, Chemical, Material and Electronic Engineering (IC4ME2), Rajshahi, Bangladesh, pp. 1-4, February 2018, doi: 10.1109/IC4ME2.2018.8465657.

[2] S. Battiato, G. Gallo, and G. Messina, "SVG Rendering of Real Images Using Data Dependent Triangulation," SCCG'04 Proceedings of the 20th Spring Conference on Computer Graphics, pp. 185-192, April 2004, doi: 10.1145/1037210.1037238.

[3] M. Hergarden and S. Jongmans, "Shared Memory Implementations of Protocol Programming Languages, Data-Race-Free," ICOOOLPS'18 Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, pp. 36-40, July 2018, doi: 10.1145/3242947.3242952.

[4] C. Hsia, S. Hsia and Y. Chou, "Application of CAD/CAE/3D printing to development of magnetic foldable hanger," 2016 International Conference on Advanced Materials for Science and Engineering (ICAMSE), Tainan, Taiwan, pp. 657-660, November 2016, doi: 10.1109/ICAMSE.2016.7840288.

[5] M. Joy and T. Axford, "GCODE: a revised standard for a graph representation for functional programs," ACM SIGPLAN Notices, vol. 26, issue 1, pp. 133-139, January 1991, doi: 10.1145/122203.122214.

[6] I. Liss and T. McMillan, "The Implementation of a Simple Turtle Graphics Package," ACM SIGCSE Bulletin, pp. 45-53, 1987, doi: 10.1145/39316.39326.

[7] N. Matsakis and F. Klock, "The Rust Language," HILT 2014: Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, pp. 103–104, October 2014, doi: 10.1145/2663171.2663188.

[8] S. Panghal, D. A. Bilung, N. Gupta and G. Kumar, "Enhancing Graphic Performance Curve using Ray Tracing," 2020 12th International Conference on Computational Intelligence and Communication Networks (CICN), Bhimtal, India, pp. 55-59, September 2020, doi: 10.1109/CICN49253.2020.9242622.

[9] W. Shalannanda, "Digital Logic Design Laboratory using Autodesk Tinkercad and Google Classroom," 2020 Proceedings of the 14th International Conference on Telecommunication Systems, Services, and Applications (TSSA), Bandung, Indonesia, pp. 1-5, November 2020, doi: 10.1109/TSSA51342.2020.9310842.

[10] A. Sharma, S. Madhvanath, A. Shekhawat and M. Billinghurst, "MozArt: a Multimodal Interface for Conceptual 3D Modeling," ICMI'11 Proceedings of the 13th international conference on multimodal interfaces, pp. 307-310, November 2011, doi: 10.1145/2070481.2070538.

[11] K. P. Yang, P. McDowell, R. Demourelle, T. Parker, and E. Langstonirst, "3D Printing: A Custom-Built 3D Printer with Wireless Connectivity," SSRG International Journal of Computer Science and Engineering (SSRG-IJCSE), ISSN 2348 – 8387, vol. 7, issue 10, pp. 1-5, October 2020, doi: 10.14445/23488387/IJCSE-V7I10P101.

[12] J. Ye, H. Liu, W. Xu, and J. Li, "Research and Application of Integrated System Design of Smart Substation Secondary System Based on SCD/CAD File Mapping Technology," 2019 IEEE 3rd Conference on Energy Internet and Energy System Integration (EI2), Changsha, China, pp. 2844-2848, November 2019, doi: 10.1109/EI247390.2019.9061834.