HO CHI MINH CITY UNIVERSITY OF SCIENCE

Project Report (Milestone 2)

SVG RENDER

Instructor:	Đỗ Nguyên Kha
Member list:	(22127006) Nguyễn Duy Ân (22127091) Phạm Mai Duyên (22127104) Phạm Thị Mỹ Hạnh (22127259) Nguyễn Đức Mạnh

Ho Chi Minh City, 12/2023

TABLE OF CONTENT

1. Introduction

The project centers around the rendering of Scalable Vector Graphics (SVG) images using various reading and drawing tools. SVG, a widely used XML-based vector image format, necessitates efficient rendering mechanisms for visualizing its content. In this report, the focus is on rendering SVG images using tools for reading and drawing from files. The report will delve into the code implementation, deployment methods, output visualization, results obtained, and critical observations. The objective is to provide a comprehensive understanding of how SVG images are processed and presented within the project.

The project unfolds through a strategic roadmap, encompassing three distinct milestones.

- **Milestone 1 - Rendering Basic Shapes:** The initial milestone focuses on rendering fundamental shapes. Emphasis will be placed on how the code handles basic shapes and the subsequent visual outcomes.
- **Milestone 2 - Transformation, Paths, and Grouping:** Moving beyond basic shapes, the second milestone explores the implementation of transformations, paths, and grouping elements within SVG.
- **Milestone 3 - Implementation of LinearGradient, RadialGradient, and viewBox:** The final milestone involves incorporating advanced features such as linear and radial gradients, as well as the viewBox attribute. Insights into how these features enhance the overall visual appeal and flexibility of the rendering system will be thoroughly explored.

By structuring the introduction in this manner, the reader gains a clear understanding of the project's objectives, the technologies employed, and the phased approach taken to achieve key milestones. In the pursuit of an effective SVG rendering solution, the project leverages two essential libraries: RapidXML for parsing SVG files and GDI+ for rendering images.

a. RapidXML - Lightweight XML Parsing

RapidXML stands out as a lightweight and high-speed XML parsing library integral to our SVG rendering project. Its efficiency lies in its minimalist design and focused functionality. RapidXML excels in swiftly extracting vital information from SVG files, ensuring a streamlined parsing process. The library's simplicity enhances its speed, making it an ideal choice for handling XML data within our project.

RapidXML's key features include efficient memory usage, a user-friendly interface, and a parsing mechanism optimized for performance, making it an indispensable component in our endeavor to decode SVG files accurately.

b. GDI+ - Robust Graphics Rendering

Graphics Device Interface Plus (GDI+) assumes a pivotal role in our project, serving as a robust graphics library to convert parsed data into visually compelling images. GDI+ is renowned for its versatility in rendering graphics, offering a plethora of features to enhance the visual appeal of the final output.

Among its notable features are advanced drawing capabilities, support for various image formats, and a comprehensive set of functions for transforming and manipulating graphical elements. GDI+ empowers our project to seamlessly translate parsed SVG data into captivating images, ensuring a visually rich and aesthetically pleasing rendering outcome.

2. Class diagram

In the conceptualization of our SVG image rendering project, we designed a hierarchical class structure aimed at encapsulating common attributes and methods shared among various shapes. At the core of our design is the creation of a parent class, "Shape," which serves as a blueprint for the properties and actions common to all shapes. Additionally, we introduced a method within the "Shape" class to streamline the setting of shared attributes. Subsequent subclasses inherit these characteristics, but we have made efforts to implement specific features:

- **Shape Class:**
    + The central class acting as the parent to other shape classes.
    + Contains common attributes and methods such as drawing and setting properties.
    + Implements a method for efficiently configuring shared attributes across shapes.
- **Derived Classes:**
    + Various derived classes inherit from the "Shape" class, benefiting from its shared functionalities.
    + Each derived class includes a constructor to initialize attributes to their default states, promoting a consistent starting point for different shapes.
- **Path Class:**
    + Due to the distinctive construction and reading methods of paths, additional functions are implemented to facilitate the updating of points during the drawing process.
    + Recognizing the unique characteristics of paths, we tailored specific functions to handle their complexities effectively.
- **Group:**
    + Instead of utilizing class inheritance for the "Group" class, we have opted for a direct storage and update strategy within individual child elements. The primary goal is to optimize memory usage within the group structure.

+ Attributes are stored and updated directly within each child element, eliminating the need for additional memory overhead associated with class inheritance.

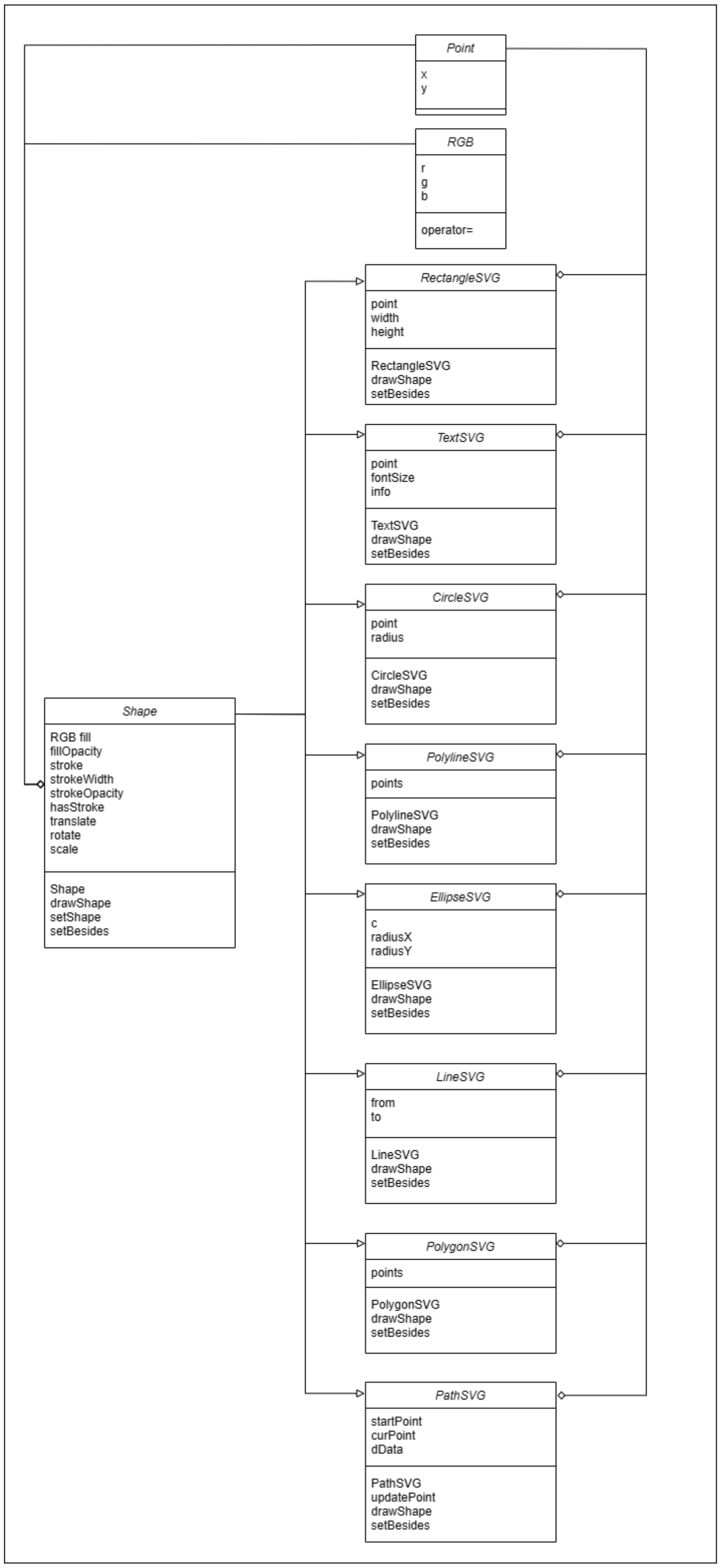


Fig 1: Class diagram

This class hierarchy and methodology ensure a modular and extensible design, allowing for easy integration of new shapes while providing flexibility for specialized implementations where needed. The UML diagram visually captures the relationships and structures, providing a comprehensive guide to the architecture of our SVG rendering classes.

3. Coding

a. Utility Functions

In our SVG image rendering project, the diversity in data types for reading and setting inputs in the drawing functions prompted the creation of various utility functions. These utility functions play a crucial role in seamlessly handling the nuances of different data types. Here's an overview of the essential utility functions:

- **Function *opacity2alpha***
    + Convert opacity value to alpha value (part of RGBA).
    + Use conditional checks to ensure opacity is within the range of 0 to 1, then convert the value and return it as an unsigned char.
- **Function *parseRGB***
    + Convert RGB color description string to a color object.
    + Parse the string to extract the values of the three color components (Red, Green, Blue) and create the corresponding color object.
- **Function *parsePoints***
    + Convert a string describing points to a vector of Points objects.
    + Parse the string to extract pairs of x, y values for each point and create a vector of Points objects.
- **Function *parsePath***
    + Convert a string describing SVG path data to a vector of pairs, each containing a command character and a vector of Points objects.
    + Preprocess the input string for consistent parsing by handling spaces, commas, and numeric values.
    + Utilize conditional checks to interpret different SVG path commands and extract corresponding Points.
    + Organize the parsed information into a vector of pairs, capturing the command and associated Points.

b. **Main Functions (Drawing and Set Properties)**

In our SVG image rendering project, the backbone of our functionality relies on two key functions: *setProperties* and *drawShape*. These functions serve as the primary orchestrators, seamlessly coordinating the execution of auxiliary read and draw functions. Below is a detailed description of each function:

- **Function setProperties**

  + **Purpose**

    setProperties is designed to organize and send the properties of SVG objects from your source code to actual objects created for drawing on a Graphics object.

  + **Setting Properties**

    This function takes the name of the SVG object (e.g., "rect," "text," "circle," etc.) along with a vector of property-value pairs (vector<pair<string, string>> a).

    Depending on the name of the object, an object of the corresponding class (such as RectangleSVG, TextSVG, etc.) is created.

  + **Calling the setBesides Method**

    After creating the object, the function calls the setBesides method of the newly created object.

    The setBesides method is responsible for reading from the vector a and setting the properties of the SVG object based on the property-value pairs.

  + **Calling the drawShape Method**

    After the properties are set, the function proceeds to call the drawShape method to draw the object onto the Graphics object passed to the function.

  + **Flexibility**

    This function helps create an intermediary layer between reading and organizing properties and the process of drawing objects, making the source code more extensible and maintainable.

- **Function drawShape**

  + **Draw geometry on a Graphics object**

    Use the set properties (color, opacity, transformations) and call specific drawing functions for each type of geometry.

    The drawShape functions in your source code use the GDI+ (Graphics Device Interface) library to draw SVG images onto a Graphics object.

  + **Graphics Class**

    The drawShape functions take a reference to a Graphics object, which is a key object for drawing images.

    Graphics is part of GDI+, used for drawing 2D images on a surface, often a window or a bitmap image.

  + **Transformation**

    Before drawing the image, the functions perform transformations such as translation (TranslateTransform), scaling (ScaleTransform), and rotation (RotateTransform). This helps draw SVG images at the desired position and shape.

  + **Image Objects**

    The functions create and use image objects like Rect, PointF, Pen, SolidBrush, GraphicsContainer, and GraphicsPath.

    Pen is used to draw the outlines of shapes.

    SolidBrush is used to fill colors for shapes.

  + **Drawing Shapes**

    The functions use methods of the Graphics object like DrawRectangle, FillRectangle, DrawLine, DrawEllipse, FillEllipse, DrawPolygon, FillPolygon, and DrawPath to draw images.

    These methods take objects like Pen and SolidBrush to determine color and transparency.

+ **Opacity**

  Opacity values for fill and stroke are converted to alpha values using the opacity2alpha function, and then used to set the transparency of the Color object in GDI+.

+ **State Preservation**

  The functions use GraphicsContainer to preserve the state of the Graphics object. This approach ensures that transformations do not affect other drawings.

c. Process path

- **Reading path - parsePath(string pathData)**

  + **In the first loop, iterate pathData string to normalize the data.**

    1. Ignore whitespace.

    2. Replace commas ',' with whitespace and increment the index by 1.

    3. Add whitespace before each non-digit or '.' character.

    4. Add whitespace before a number if the character before it is not a digit, '-', or '.'.

  + **Print the normalized string:** Afterward, the function prints the normalized string to the screen.

  + **Processing the normalized string using stringstream:** The function uses a stringstream to process the normalized string.

  + **Iterating through each command and its corresponding points:** The function employs a while loop to iterate through each character in ss (stringstream). Depending on the value of command, the function performs different actions:

  + **For commands 'M', 'm', 'L', 'l', 'C', 'c'**

    Use a while loop to read consecutive pairs of numbers (x, y) from the stringstream and add them to the points vector.

    Each pair of numbers is added to the points vector.

    The points vector is then added to the commands vector along with the corresponding command.

+ **For commands 'H', 'h'**

  Read a number (x) from the stringstream and add it to the points vector.

  The points vector is then added to the commands vector along with the corresponding command.

+ **For commands 'V', 'v'**

  Read a number (y) from the stringstream and add it to the points vector.

  The points vector is then added to the commands vector along with the corresponding command.

+ **For commands 'Z', 'z':** No numerical value is read, and the points vector is added to the commands vector with the corresponding command.

+ **Return the result:** Finally, the function returns the commands vector, which contains pairs of commands and corresponding point vectors. This data structure is useful for further processing or rendering graphics based on SVG path data.

- **Drawing path**

  + **Initialize GraphicsPath and Pen/Brush:** GraphicsPath is an object to store the drawn path, and Pen and SolidBrush are initialized to determine the color and width of the stroke.

  + **Loop through path commands:** The function uses a loop to iterate over each path command in dData.

  + **Handle path commands:** Based on the type of path command, the function performs different operations:

  + **'M' or 'm' (Move To):** Moves to a new point, and if there are multiple points, adds line segments between them.

  + **'L' or 'l' (Line To):** Draws straight line segments from the current point to new points.

  + **'H' or 'h' (Horizontal Line To):** Draws straight line segments along the x-axis.

+ **'V' or 'v' (Vertical Line To):** Draws straight line segments along the y-axis.
+ **'C' or 'c' (Cubic Bezier Curve To):** Draws cubic Bezier curves.
+ **'Z' or 'z' (Close Path):** Closes the path by drawing a line from the current point to the first point.
+ **Draw and fill the path:** After constructing the path, high-quality drawing mode and smoothing mode are set, and the path is drawn using graphics.DrawPath (if there is a stroke) and filled using graphics.FillPath (if there is a fill).
+ **End container:** Finally, the function ends the drawing container to conclude its drawing.

d. Process transform

We use 2 functions of the library, including: Begin Container and Transformations.

It uses graphics.BeginContainer() to start a new drawing container and then applies transformations such as translation (TranslateTransform), scaling (ScaleTransform), and rotation (RotateTransform). These transformations are applied to position and shape the path correctly on the canvas.

e. Process group

- **Iterative Node Processing**
    + The function iterates through the children nodes of a <g> node, systematically examining each element within the group.
    + It employs a recursive approach, determining the type of geometry or nested group for each child node.
- **Recursive Handling of Child Groups**
    + For each child node encountered, the rendering process involves recursive calls to the same function, allowing for the hierarchical exploration of nested groups.
    + This recursive strategy ensures that the rendering pipeline can effectively navigate through complex SVG structures, accommodating a diverse range of group formations.

- **Attribute Handling**
    + As the function processes each child node, it captures and stores relevant attributes.
    + Attribute values are converted and assigned sequentially to properties of the child elements, ensuring that the rendering environment is appropriately configured for each shape or group.

4. Implementation

a. **Template, environment, and tool**

- **DemoProject Template**

The provided DemoProject template serves as the architectural blueprint, offering a structured skeleton for our SVG rendering codebase. It includes essential components, such as main application files, header files, and a modular directory hierarchy. Leveraging this template ensures consistency and adherence to a standardized coding style throughout the project.

In addition, we also reorganized the files to match the team's goals and mindset (see section 4.b).

- **Development Environment**

Our development environment is centered around the use of C++ as the primary programming language. C++ provides the necessary flexibility and performance required for handling the intricate operations involved in SVG rendering.

The combination of RapidXML and GDI+ libraries enriches the development environment by bringing XML parsing and graphics rendering capabilities, respectively. This symbiotic relationship between the programming language and libraries creates a robust foundation for the project.

- **Integrated Development Tool**

To streamline the development process, we employ Visual Studio, Microsoft's integrated development environment (IDE). Visual Studio facilitates seamless code editing, debugging, and project management. Its user-friendly interface and powerful features enhance the efficiency of our development workflow. The tool's compatibility with C++ and its ability to integrate with external libraries make it an ideal choice for our SVG rendering project.

b. **Code structure**

Our SVG rendering project adopts a meticulously organized code structure, distributed across three key folders and structured into three main files:

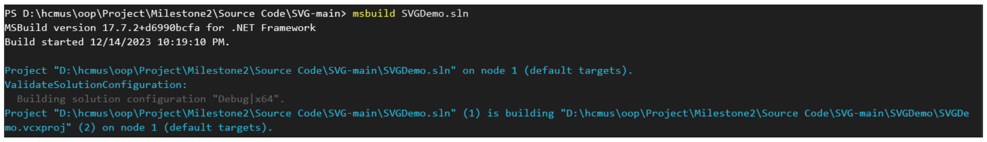
- **Folder "const":** This directory houses crucial files that define constants, ensuring consistency and ease of maintenance throughout the project. Storing constants in a dedicated folder enhances code readability and simplifies updates or modifications to shared values. This centralized approach in the "const" folder promotes a unified coding standard for our SVG rendering application.

- **Folder "public":** The "public" folder plays host to essential ICO files, used for application icons, and sample test files. This segregation ensures that external resources, such as icons and sample tests, are kept separate from the core codebase. This arrangement not only streamlines the code directory but also makes it more accessible for future modifications or additions to graphical assets.
- **Folder "src":** The heart of our codebase resides in the "src" folder, containing multiple CPP files that contribute to the project's functionality. These files are intelligently segmented to complement the "stdafx.h" file, promoting modularization and maintainability. The strategic division into smaller files aids in reducing complexity and enhances the code's comprehensibility.
- **File "stdafx.h":** The "stdafx.h" file serves as a centralized hub for including necessary libraries, utility functions, and key class declarations. It encapsulates the project's essential components, such as data type conversion utilities and foundational classes like "Shape" and its inheritors. This file acts as a precompiled header, optimizing the build process and promoting a clean separation of concerns within the codebase.
- **File "stdafx.cpp":** "stdafx.cpp" complements the "stdafx.h" file by implementing the utility functions declared in the header. This separation of declaration and implementation maintains code clarity and adheres to the best practices of modularization. The utility functions within this file handle data type conversions and other generic operations critical to the SVG rendering process.
- **File "SVGDemo.cpp":** The core functionality of our SVG rendering application is housed in the "SVGDemo.cpp" file. Acting as the main entry point, this file contains the primary function for reading SVG files, rendering images, and orchestrating the overall application flow. The "SVGDemo.cpp" file essentially operates as the main driver for the SVG rendering project, executing the core logic and ensuring the seamless integration of the various components.

By adopting this well-structured approach to code organization, we aim to enhance readability, maintainability, and collaboration among team members while facilitating the scalability of our SVG rendering project.

c. Execution

- **Build the Solution:** Open the command prompt and navigate to the directory containing the solution file (SVGDemo.sln). Use the msbuild command to build the solution. This command compiles the code and generates the necessary executable files. For example: `msbuild SVGDemo.sln`



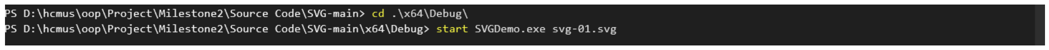
```
PS D:\hcmus\oop\Project\Milestone2\Source Code\SVG-main> msbuild SVGDemo.sln
MSBuild version 17.7.2+d6990bcfa for .NET Framework
Build started 12/14/2023 10:19:10 PM.

Project "D:\hcmus\oop\Project\Milestone2\Source Code\SVG-main\SVGDemo.sln" on node 1 (default targets).
ValidateSolutionConfiguration:
  Building solution configuration "Debug|x64".
Project "D:\hcmus\oop\Project\Milestone2\Source Code\SVG-main\SVGDemo.sln" (1) is building "D:\hcmus\oop\Project\Milestone2\Source Code\SVG-main\SVGDemo\SVGDe
mo.vcxproj" (2) on node 1 (default targets).
```

Fig 2: Build Solution

- **Navigate to the Debug Directory:** Change the current directory to the x64 Debug directory where the compiled executable is located. This is typically done using the cd (Change Directory) command. For instance: `cd .\x64\Debug\`

- **Run the Executable:** Initiate the execution of the SVGDemo.exe program by using the start command, followed by the executable name and the SVG file you want to render. For example: `start SVGDemo.exe <svg_file_name.svg>`



```
PS D:\hcmus\oop\Project\Milestone2\Source Code\SVG-main> cd .\x64\Debug\
PS D:\hcmus\oop\Project\Milestone2\Source Code\SVG-main\x64\Debug> start SVGDemo.exe svg-01.svg
```

Fig 3: Directory and execute

d. Result

In the pursuit of evaluating the practical application of our SVG image rendering project, we conducted thorough experimentation and testing. To witness the program in action, please refer to the following link for the implementation: SVG Milestone 2 - Group 13 - Demo (youtube.com).

For comprehensive testing, we utilized a set of 18 images provided by our instructor. These images serve as a diverse test suite, covering a range of SVG complexities and features. While the full set of images is available for scrutiny, we present below the results of testing using a representative sample of 3 images from the aforementioned collection.

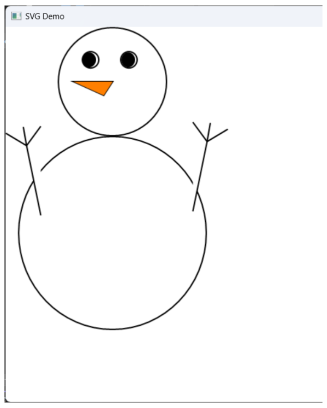


Fig 4: Sample 17 render from our code

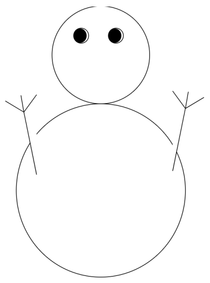


Fig 5: Sample 17 render from computer

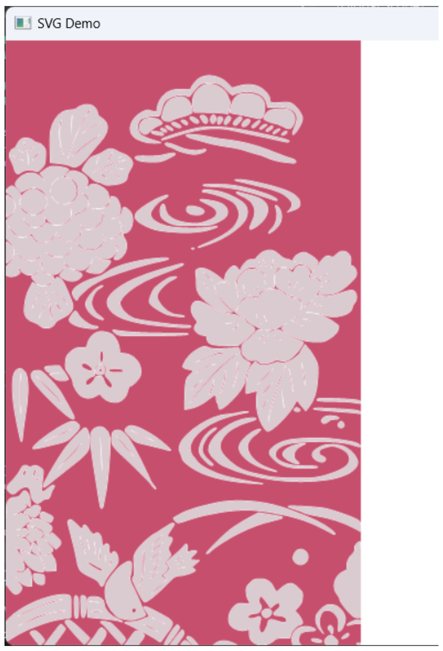


Fig 6: Sample 13 render from our code

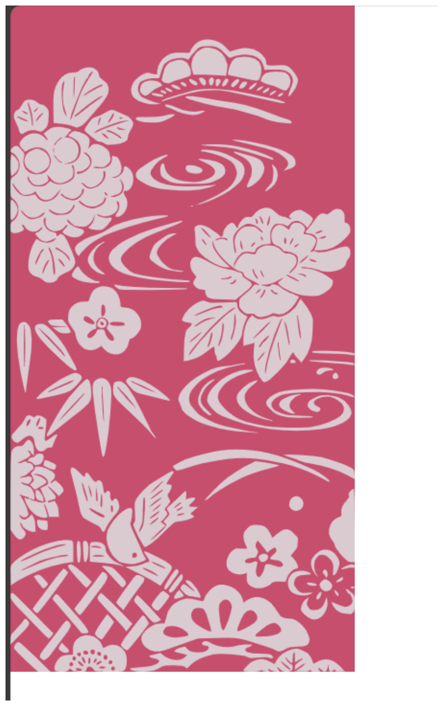


Fig 7: Sample 13 render from computer

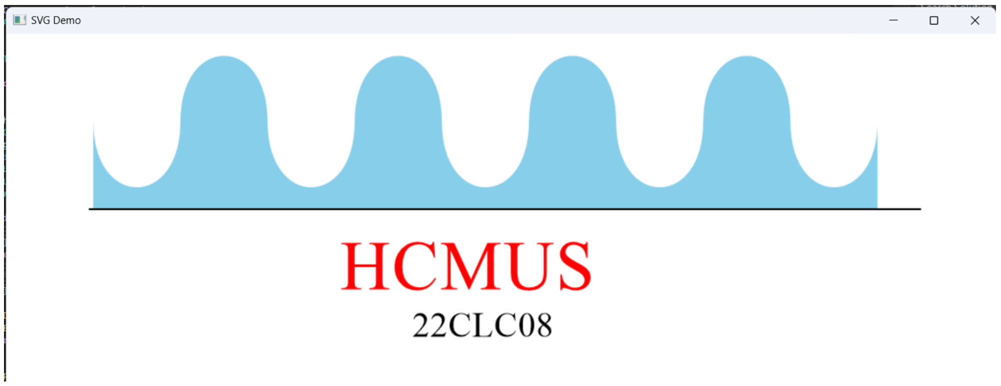


Fig 8: Sample 10 render from our code

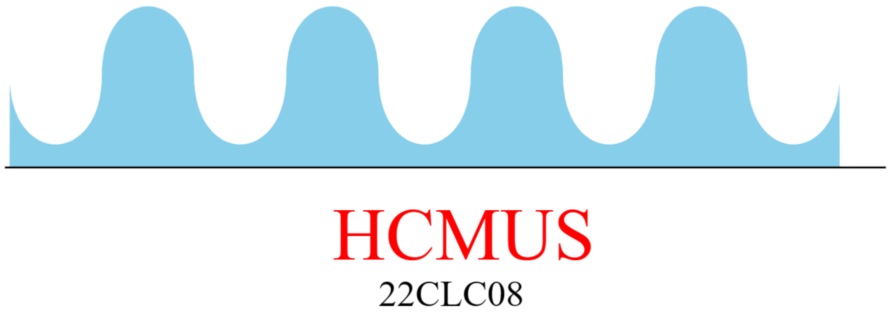


Fig 9: Sample 10 render from computer

The rendered images from the SVG files showcase a reasonably commendable level of completion in the overall samples. However, a closer examination reveals a couple of noteworthy observations. Sample 13, while adequately executed, lacks the desired level of sharpness. This deficiency in clarity could potentially detract from the visual appeal of the rendering. It is crucial to address this aspect to ensure that the final output meets the expected standards of precision and crispness.

Moreover, in the case of sample 17, there is a noticeable dearth of details. The rendering falls short in capturing intricacies that could enhance the overall quality of the image. Detailing plays a pivotal role in the visual fidelity of the render, and its absence, as seen in sample 17, diminishes the overall impact of the final output. A more comprehensive approach to incorporating finer details in the rendering process is imperative to elevate the quality of the visual representation.

In addition to the successful execution of our SVG image rendering project, a notable highlight is the remarkable speed at which it operates—nearly real-time. This swift processing capability positions our solution as highly efficient, providing a seamless rendering experience. However, despite this commendable speed, the focus shifts towards enhancing the system's adaptability to a broader spectrum of potential scenarios.

5. Conclusion

Upon completion of our SVG image rendering project, we are pleased to announce the successful achievement of both Milestone 1 and 2, attaining approximately 90% of our set goals. Reflecting on the project's strengths, we note the following:

- **Programmatic Achievements:** The program has demonstrated a high degree of accuracy, aligning closely with our initial expectations. Notably, the execution time is commendably swift, meeting the criteria for real-time performance.
- **Team Collaboration:** The collaborative efforts within the team have been harmonious and effective. There has been a positive atmosphere of mutual assistance, contributing to the overall success of the project.

However, there are some points we need to improve, including:

- **Programmatic Challenges:** Despite the substantial progress made, there remain a few outstanding challenges in handling specific test cases. Identifying and resolving these cases will be a priority for refining the robustness of our SVG rendering solution.
- **Team Collaboration Enhancement:** While teamwork has generally been successful, there is room for improvement in the utilization of collaborative tools, particularly GitHub. The team should transition from manual file exchanges to a more systematic use of version control for code commits, fostering a more efficient and organized development process.

By addressing these identified areas for improvement and consistently refining our codebase, we envision a further streamlined and optimized SVG image rendering project that not only meets but exceeds the envisioned standards of functionality and teamwork.