

VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY



---

# PROJECT REPORT

SVG READER

---

Course: OBJECT-ORIENTED PROGRAMMING

March 15, 2025

VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY  
UNIVERSITY OF SCIENCE  
FACULTY OF INFORMATION TECHNOLOGY

# PROJECT REPORT

## SVG READER

Course: OBJECT-ORIENTED PROGRAMMING

### *Instructors' names*

*Do Nguyen Kha  
Truong Toan Thinh*

*Mai Anh Tuan  
Nguyen Le Hoang Dung*

### *Students' names*

Lam Tien Huy  
Trinh Quoc Hiep  
Le Thanh Loi  
Nguyen The Hien

### *ID*

22127151  
22127108  
22127238  
22127107

### *Class*

22CLC08  
22CLC08  
22CLC08  
22CLC08

March 15, 2025



## Acknowledgments

It has been a pleasure to be able to learn more in CSC10003 - Object-Oriented Programming as a crucial part of our major, Information Technology. This SVG Reader project is also a part of the route to developing more skills in coding, specifically in C++ language. It is also an opportunity for us to be able to create our first-ever image reader in an application, which we greatly appreciate.

For this, we want to express our sincere gratitude towards our Instructor, Mr. Do Nguyen Kha for delivering us a chance to gain more experience in the coding field through this assignment, as well as supporting us in the past few weeks throughout this course. Aside from that, we also want to send our uttermost appreciation to our Lab Instructors, Mr. Nguyen Le Hoang Dung, Mr. Mai Anh Tuan, and Mr. Truong Toan Thinh for offering us a useful share of knowledge.

Giving everything into this coding task, we want to say thank you to our wonderful classmates of 22CLC08 who had discussions with us and helped us in the journey of completing this challenging mission. It is quite an amazing thing to know that many students are here not only to explore, to compete but also willing to learn and help others.

We hope that you would like the SVG Reader and our report on this product accordingly. We also hope to receive unprejudiced comments and reasonable evaluations from teachers to make certain improvements in the future.

## Abstract

We created SVG Reader - an application using C++ programming language and Object-Oriented Programming principles. The application will parse an SVG file, render the vector graphics, and handle user interactions if applicable.

In this particular report, the program will be broken down into fundamentals and analyzed as clearly as possible. We will be introducing the SVG Reader, explaining all the features, functions, tools, and libraries. Subtle remarks as well as discussions may be seen in the report. The program is written in the C++ language and should operate well after being compiled by Visual Studio.

One thing to notice is that we will not include or write our source code directly in the report to avoid confusions and lengthiness. Nonetheless, it is up to us to provide extremely precise descriptions of our program, our code organization, and our idea of distributing classes. We are also providing direct links for easy access from this page.

Milestone 1 Demonstration Video: <https://youtu.be/H8SbhDDw9MA>

Milestone 2 Demonstration Video: <https://youtu.be/m15kdhqLONE>

Final SVG Reader Demonstration Video: <https://youtu.be/4J1kprnxFFM>

GitHub Repository: [https://github.com/LeThanhLoi19012004/CSC10003\\_SVG\\_READER](https://github.com/LeThanhLoi19012004/CSC10003_SVG_READER)

After reading this report, the instructors will have more of an idea of how this program works with all the coding involved.

# Table of Contents

<b>Acknowledgments</b>	<b>1</b>
<b>Abstract</b>	<b>2</b>
<b>Table of Contents</b>	<b>3</b>
<b>List of Figures</b>	<b>5</b>
<b>1 Brief Introduction</b>	<b>6</b>
1.1 SVG File	6
1.2 SVG Reader	7
<b>2 Basic Organization</b>	<b>8</b>
2.1 UML Class Diagram	8
2.1.1 Definition	8
2.1.2 UML Class Diagram for SVG Reader	9
2.1.3 Brief Description	10
2.2 Class Breakdown	11
2.2.1 Simplified UML Class Diagram	11
2.2.2 Color and Stroke	11
2.2.3 Figures	11
2.2.4 Point	12
2.2.5 Ellipse and Circle	13
2.2.6 Rectangle	13
2.2.7 Text	13
2.2.8 Line	14
2.2.9 Polyline and Polygon	14
2.2.10 Path	14
2.2.11 Group	15
2.2.12 FactoryFigure	15
2.2.13 Parser and Renderer	15
2.2.14 Image	16
2.2.15 Stop	16
2.2.16 Gradient	16
2.2.17 Linear Gradient and Radial Gradient	17
2.2.18 Viewbox	17

<b>3</b>	<b>Program Analysis</b>	<b>18</b>
3.1	Project Initialization	18
3.1.1	Concept	18
3.1.2	GDI+	18
3.1.3	Run SVG Reader using Command-line	19
3.2	Code Distribution	20
3.2.1	Source and Header Files	20
3.2.2	Libraries and Other Files	22
<b>4</b>	<b>Conclusive Statements</b>	<b>23</b>
4.1	Final Remarks	23
4.2	Work Description	24
	<b>References</b>	<b>25</b>

## List of Figures

1	SVG Image Representation . . . . .	6
2	Sample SVG Image to Render . . . . .	7
3	UML Formatting Rules . . . . .	8
4	UML Class Diagram for SVG Reader . . . . .	9
5	Simplified UML Class Diagram . . . . .	11
6	Command-line for SVG Reader . . . . .	19
7	Code Distribution for Core Classes . . . . .	20
8	Code Distribution for Figure Classes . . . . .	21
9	Libraries used in SVG Reader . . . . .	22
10	Work Organization . . . . .	24

# *Chapter 1*

## Brief Introduction

### 1.1 SVG File

According to Adobe, Scalable Vector Graphics (SVG) is a web-friendly vector file format. In contrast to pixel-based raster files like JPEGs, vector files store images via mathematical formulas based on points and lines on a grid. Consequently, SVGs can be significantly resized without losing quality, which makes them ideal for logos and complex online graphics.



Figure 1: SVG Image Representation

SVGs are hugely popular with web designers, because they are written in XML code, meaning they store any text information as literal text rather than shapes. This allows search engines like Google to read SVG graphics for their keywords, which can potentially help a website move up in search rankings.

Regarding formatting, SVG files are written in XML, a markup language used for storing and transferring digital information. The XML code in an SVG file specifies all the shapes, colors, and text that comprise the image. Usually starting with the tag of `<shapetype>` and the attributes behind that shape type. Since the reading of this file is straightforward, it is a perfect choice to make an Object-oriented Programming style.



## 1.2 SVG Reader

We will have to design and implement an SVG file rendering application using C++ programming language and Object-Oriented Programming principles. The application will parse an SVG file, render the vector graphics, and handle user interactions if applicable. The program must have the capability to read SVG files and parse their content, handling various SVG elements, attributes, and their hierarchical structure. Classes representing SVG elements are handled using object-oriented definition.



Figure 2: Sample SVG Image to Render

For the main part, we should implement a rendering engine that utilizes the parsed SVG data to draw vector graphics on the screen using object-oriented design patterns to represent different SVG elements as objects and render them accordingly. Zooming and rotation are optional, but including them are nice as well. Basically, it is about implementing a well-organized class hierarchy representing different SVG elements, utilizing inheritance, encapsulation, abstraction, and polymorphism to create a robust and flexible design. Implementing design patterns for object creation and traversing SVG elements are considered as well.

The image above is going to be our starting SVG File for this project, which will be interesting because there are lots of components and attributes going on. However, after the first milestone, tougher figures are introduced, which are paths and groups. All the groups will bring their own samples into demonstration for the whole class to work on the project.

# Chapter 2

## Basic Organization

### 2.1 UML Class Diagram

#### 2.1.1 Definition

A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations or methods, and the relationships among objects.

In the below image are basic formatting rules that we are going to use in our diagram. From the definition of classes, attributes, and methods, to the relationship arrow types between classes, all are present in the image to give you a brief understanding of where we are going.

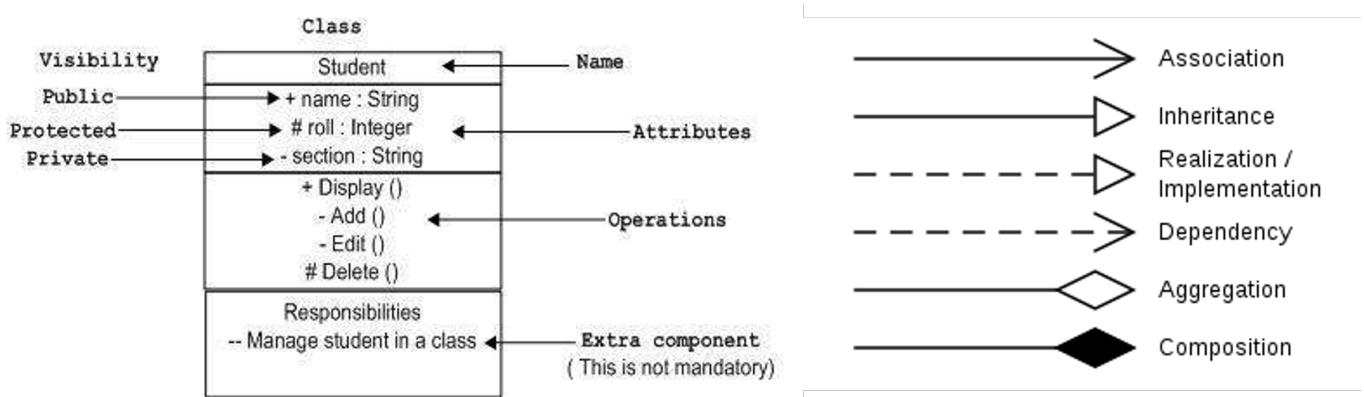


Figure 3: UML Formatting Rules

There are certain things that have not been mentioned in the image above, which we will clearly state here. For **virtual methods**, the formatting would be in *italic*. For **static methods**, the formatting would be underlined. If constructors and destructors in a class are essential, they will appear in our UML diagram. Or else, if they are just usual constructors and destructors with nothing special in particular, we might discard it in our diagram for readability.

[illegible]

Figure 4: UML Class Diagram for SVG Reader

This is our full-sized UML Class Diagram for our SVG Reader. If you find this distracting or hard to read, we also have a second simplified version to be shown in the upcoming sections.

### 2.1.3 Brief Description

Above is our **UML Class Diagram** for this project of SVG Reader. We included every class that are used as well as their attributes and methods. Since we are doing Object-Oriented Programming style, this diagram is to show the relationship between classes and how they all work together. In all, we believe that this can provide you an overview on how our program is organized.

Going from the lowest category, the **Point** class is a starting point to our figures. Most of the classes that are considered figures would at least have one point as an attribute, some might contain many. It is a very basic concept to have, and it gets destroyed if a figure gets destructed, so the relative aspect will be composition, or aggregation in certain circumstances of figures where the point vectors are there.

All the different types of figures: **Ellipse**, **Line**, **Polygon**, **Polyline**, **Rectangle**, **Text**, **Path**, and **Group**, are the next classes to be mentioned. The **Circle** class is inherited from the **Ellipse**, since a circle is a special type of an ellipse. Each and every one of them are inherited from the class **Figures**, which is a very flexible way to handle shapes.

Class **Figures** is a crucial part in our program, providing the concept of Polymorphism to the figures that have been brought up. It includes default and virtual functions to handle shaping figures. **Color** and **Stroke** classes are made to be indispensable attributes for a figure, hence the composition relationship. **FactoryFigure**, after that, plays the role of creating a series of figures according to their type, so a dependency arrow is drawn.

The **Group** class is one of the most special classes in our program. They can contain themselves as well as **Figures**, so it usually involves aggregation relationship with other classes, even itself. Nevertheless, a group is still a figure, so it is still inherited from **Figures**.

Finally, going to the **Image** class, which is the final product after calling the works of parsing and rendering figures from the SVG file. This is why **Render** and **Parser** classes, which plays the exact roles of their names, are dependent to **Group** and **Figures**. In contrast, **Image** is dependent to **Renderer** and **Parser**. An image can have multiple figures, therefore, we are going to classify the relationship as aggregation.

## 2.2 Class Breakdown

### 2.2.1 Simplified UML Class Diagram

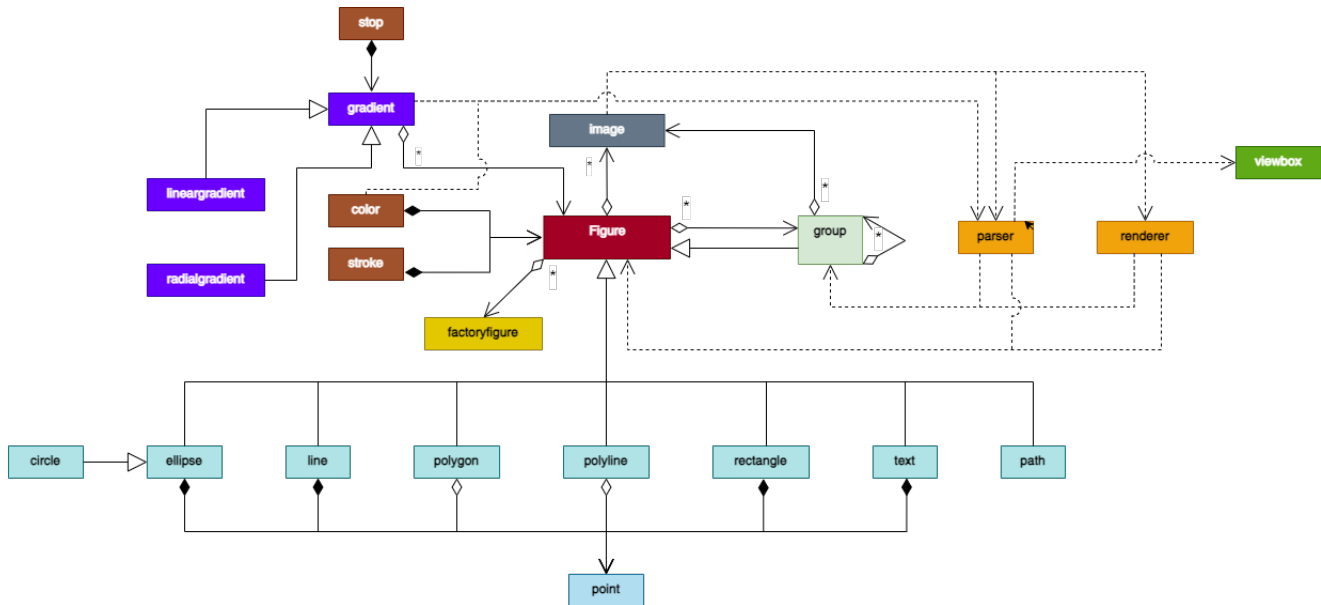


Figure 5: Simplified UML Class Diagram

This is our simplified UML Class Diagram for our SVG Reader. This version is more simple and straightforward, allowing you to look more directly of the concept of this program.

### 2.2.2 Color and Stroke

Exactly like their names, Color and Stroke include common definitions. Color follows the RGB color mode, so it contains float numbers of **r**, **g**, **b** as the number indications of red, green, and blue, and **opacity** for the opacity of the color object. Stroke contains a width **strokeWidth** float and a color named **strokeColor**. Without these two classes, we can not form any figures or any types of images. So Color and Stroke are primary attributes of a figure.

### 2.2.3 Figures

Figures are defined as graphical representations of objects' form or their external boundaries, outlines, or external surfaces. This class is inherited by all the figures, containing strings **fig** for name definition of the figure, and **text\_name** specifically made for text figure. We also have **fill**, as a color and **strk**, as a stroke are used.

The `vector<pair<string, vector<float>>>` **transVct** stores data that correspond to transformation in figures, specifically around rotating, translating or scaling.

Most of the figures' constructor, destructor, getter, and setter methods are obvious and straightforward, so we will not dig exclusively deep into them here or in the next sections that mention individual figures. Nonetheless, we will provide you short descriptions if there are any changes.

There contain *virtual* functions that we do need to care about. The destructor **Figure()** is used to properly destruct our figures, since it is virtual, it allows polymorphism which the destructor works on specific figure types. The **void updateProperty()** is one of the most essential functions, which parses the information from the string which contains information about that particular figure and updates the properties accordingly. These two are basically empty, but each figure would have their own definitions of these functions. They appear in every figure, so we will only mention them once here.

Lastly, the function **void updateTransformVct(string)** is designed to parse and extract transformation information from a string, specifically for translation, rotation, and scaling transformations, and then store this information in a vector. The transformations and their parameters are expected to be formatted within the input string.

#### 2.2.4 Point

A point is used to initialize coordinates in 2D coordinate axes. Floats of **x** and **y** visualize x-axis and y-axis coordinates. Boolean value of **intersect** is used to determine if a point is an intersection between two lines. This value is only turned to true if the mentioned condition is satisfied, and false otherwise. The **intersect** boolean is mostly used for polylines.

Apart from constructor, destructor, getter, and setter methods which are clear as daylight, a **bool operator == (const point& p1, const point& p2)** is included to check if two points are the same. And if they are, this returns true.

It is safe to say that Point is the starting point of every figure. In the next classes, we will further examine the role of this class in every instance.

### 2.2.5 Ellipse and Circle

As simple as it gets, Ellipse defines a "round-like" shape where the sum of the distances to two fixed points, called foci, is constant. Circle is a special case of Ellipse where both foci are at the center. Using inheritance, we can easily implement circles and ellipses and fully utilize their difference.

**Center** places the center point of an ellipse or circle. Floats of **rx** and **ry** illustrate the distance from the center to the x-axis and y-axis in an ellipse, respectively. As for a circle, there is just one radius throughout, so those two contains the same value.

### 2.2.6 Rectangle

A rectangle is a four sided-polygon with all four angles equal to 90 degrees. As for the attributes, the **root** point is stated as the top left corner of our rectangle, with **width** and **height** playing the same roles as their names.

A rectangle is a special type of polygon, however, we are not using inheritance but rather creating an independent class. A square, additionally, is a special type of rectangle, but we are also using the same class to illustrate them, with the width and height being the same value.

### 2.2.7 Text

The text element is used to define a text. Firstly, we have **textPos** point is used to define the top left corner as the starting point of our text. We have strings of **content** which represents the actual content of our text, **fontFamily** which represents the font family or typeface used for rendering the text, **textAnchor** which represents the text anchor property, which defines the point within the text box to align the text to, and **fontStyle** which represents the font style (bold, italic, underline, etc.) of the text.

Going to the remaining attributes, **fontSize** is a float that illustrates the font size of our text. Floats of **dx** and **dy** represents the horizontal and vertical displacement or shift of the text. Overall, text is one of the most different from the other figures, but it is still classified as one, keeping the same attributes inherited from the Figures class.

### 2.2.8 Line

Line segments are based on the connection between two points. Rather effortlessly defined, our Line class is made up of points **p1** and **p2**. Between the two points, a straight line will be drawn, connecting the two and making a line.

### 2.2.9 Polyline and Polygon

Polylines are made up of two or more vertices forming a connected line. Polygons are made up of at least four vertices forming an enclosed area. The first and last vertices are always in the same place. Both use the same attributes, a vector<point> **Vers** that stores a sequence of points that define the vertices of the polyline or the polygon.

These two are practically similar since they contain the same attributes, but the way to construct and draw them might contain differences. Polygons are more straightforward as a full shape and are filled all the way between all corners, but polylines are dependent on how the points are laid out to be filled inside.

### 2.2.10 Path

A path is a fundamental element used to define shapes such as lines, curves, and polygons. It is defined by a sequence of commands and parameters that describe the geometry of the shape. Paths are practically most basic ideas behind each and every figure, so they allow maximum flexibility. Consequently, paths are the most difficult figures to parse and render.

For the sake of simplicity, paths' properties in our SVG Reader will be limited to certain cases. To start up, **Moveto (M)** moves the pen to a specified point without drawing. **Lineto (L, H, V)**, which can be a straight, horizontal, or vertical line, draws a line from the input to the current point. **Cubic Bezier Curve To (C)** draws a cubic Bezier curve through the current point and 3 input points. Lastly, **Closepath (Z)** illustrates a line from the last to the first point of our path, successfully creating one.

These attributes might be a bit hard to understand. We created a vector<pair<char, vector<float>>> **vct** represents the properties of the path, storing commands mentioned earlier, and a vector of point objects representing the coordinates. Strings of **strokeLineJoin** and **strokeLineCap** represents the stroke line join and the stroke line cap property of the path.



### 2.2.11 Group

A group element is used to define a group of related graphics elements. It allows you to group multiple figures together as a single unit, even themselves. This grouping is helpful for applying transformations, styling, or other attributes to a set of elements collectively. It is probably the most special type of figure.

Going to the attributes, we have `vector<figure*> figureArray`, a private member variable representing a vector of pointers to figure objects, which stores the graphical elements contained within the group, and a `group* parent` which is a pointer to the parent group, allowing for the creation of a hierarchy of groups, where a group can be nested within another group. The nested idea is the reason why groups are very exceptional since they contain layers and layers of figures and groups inside them.

Multiple constructors are defined in our group class. We have a default constructor and a copy constructor of `group(const group&)`. A copy assignment operator of `group& operator=(const group&)` is also used alongside these methods. Other methods stay the same, as getters, setters, and a destructor as well.

### 2.2.12 FactoryFigure

FactoryFigure contains an `unordered_map<string, int> figureId`. It is shaped as a "factory" to form new figures. Based on the figure type, FactoryFigure has a method called `figure* getFigure(string name)` which parses the string and perform the task of creating new figure pointers, allowing polymorphism to each type of figure according to the cases specified.

### 2.2.13 Parser and Renderer

Parser is used to parse and process the original SVG file and preserve the data. Renderer is used to control the export window and render items into the screen. These two harmoniously work alongside each other to get figures to become an image.

Parser contains one attribute of `unordered_map<string, color> colorMap` used to store color names and their corresponding RGB values, and `unordered_map<string, gradient*> idMap` for gradients. There are both public and private methods. Private includes `void processColor(string, string, color&)` to process color information and populate the color structure, `void loadColorMap()` to load color mappings from an external file `color.txt` into the colorMap unordered map, and `void processProperty(string, string, string, figure*&)` to process properties and update the corresponding figure object. The only public method is `void par-`

**setItem(vector<figure\*>&, group\*, string)** which is the main public method responsible for parsing SVG content from a file, creating figure objects, and organizing them into a structure.

Renderer contains no attributes, and only contains public methods. We have **void renderFigure(Graphics&, group\*)** as the main method responsible for iterating through a vector of figure objects and rendering them on a device context (HDC) using GDI+, **void renderItem(vector<figure\*>, group\*, float, string, float, float, HDC)** as a wrapper for drawing figures calling the previous method mentioned. Other than that, we have separate functions for rendering specific types of figures such as rectangles, ellipses, lines, polygons, polylines, text, paths, and groups. Each function of the syntax **void draw <typeofFigure> (Graphics&, rectangle\*)** takes a GDI+ Graphics object and a specific figure type and renders it accordingly.

#### 2.2.14 Image

Image shows what is going to be printed on the screen. We have the strings **fileName**, **ImageName**, integers **width**, **height**, the float **antialiasingLevel**, a vector<Figures> **figures** used to store all the figures that will be parsed and rendered to the screen, and a group\* **root** which is a group pointer serving as the root of the figure hierarchy. This class is the answer to our program.

Our constructor **image(string)** reads the file name to start our process. Skipping over the getters and setters, our class uses **void parseImage(parser)** and **void renderImage(renderer, HDC)** to create a figure hierarchy, parsing and rendering our image, ready to be printed into the screen.

#### 2.2.15 Stop

A stop is a point along the gradient where a specific color is defined, which marks the location and color of a transition in the gradient. In gradients, you can define multiple stops to create a smooth transition between different colors. We have the color **stopColor**, a float **offset**, and two constructors for initialization

#### 2.2.16 Gradient

A gradient specifies a range of position-dependent colors, usually used to fill a region. We have a string **strLine**, a name keeping integer **gradId**, a place to store stops of vector<stop> **stopVct**, and a transformation keeper vector<pair<string,

vector<float>> **gradientTrans** as attributes of a gradient.

Except for getters and setters, there is a *virtual void updateElement()* for gradient types overriding. Default and copy constructors, a destructor and copy operator **gradient& operator=(const gradient&)** are also present.

### 2.2.17 Linear Gradient and Radial Gradient

A gradient specifies a range of position-dependent colors, usually used to fill a region. A linear gradient is a gradual transition between two or more colors along a straight line, while a radial gradient radiates outward from a central point. Both are inherited from class Gradient since they are types of gradients.

In a linear gradient, we are going to have two points of **A, B**. **void updateElement()** overrides the virtual function of Gradient, used to update new elements into our linear gradient. Similar to radial gradient, it contains multiple constructors, a destructor, and an assignment operator.

Differently, a radial gradient will have floats of **cx, cy** for center coordinates, **r** for radius, and **fx, fy** for focal points. A boolean of **isLink** is also added to check if the radial gradient is a link or not. But similarly to the previous one, **void updateElement()** overrides the virtual function of Gradient, and it contains multiple constructors, a destructor, and an assignment operator.

### 2.2.18 Viewbox

Viewbox is used to define the coordinate system and aspect ratio of the content within an SVG element. It has four coordinate float values of **viewX, viewY, viewWidth, viewHeight**, two port dimensionally related float values of **viewPort** and **portHeight**, and two strings for scaling and positioning of **preservedForm** and **preservedMode**. Most of the functions are getters and setters, and the viewbox is used as part of the parsing and rendering process.

# *Chapter 3*

## Program Analysis

### 3.1 Project Initialization

#### 3.1.1 Concept

Considering all the C++ compilers and IDEs, we decided to choose Visual Studio Community 2022 for many reasons. Compared to other similar products, Visual Studio has simply been a powerful application with a modern interface that is exquisite for C++ code development. The Community version of Visual Studio 2022 is a free version, which is perfect for us students and users as well, so it is the best choice financially and compatibly.

Regarding file handling, Visual Studio does this task very well since it gives us a straightforward code distribution to every header or cpp file. Additionally, this application links all the project files very efficiently, and there are lots of plugins and libraries that are useful as well. There are certain drawing choices to choose from, such as **SFML**, which we originally used. However, we decided that **GDI+** would be our final choice for simplicity, which that we will further mention in our report.

#### 3.1.2 GDI+

According to Stevewhims (2021), GDI+ is a class-based API for C/C++ programmers. It enables applications to use graphics and formatted text on both the video display and the printer. Applications based on the Microsoft Win32 API do not access graphics hardware directly. Instead, GDI+ interacts with device drivers on behalf of applications. GDI+ is also supported by Microsoft Win64.

GDI+ acts as an intermediate layer between applications and device drivers for rendering two-dimensional graphics, images and text. GDI+ has an internal structure that consists of about 40 extensible managed classes, 50 enumerations and six structures. We are going to implement that into our SVG Reader, using the drawing tools to render out the SVG files.

### 3.1.3 Run SVG Reader using Command-line

After downloading our code from our GitHub Repository, go to **CSC10003\_SVG\_READER\SVG\x64\Debug**, we can observe a ready-made Windows Application called **SVG\_Reader.exe**. Right-click and open the Terminal in Windows 11, or manually run Command Prompt and use the syntax **cd <drive:><path>** with the path as our folder mentioned earlier.

Be sure that your SVG files are not in inaccessible directories like OneDrive on Windows, or any hidden folders whatsoever. In Windows 11, we have very useful new features, like choosing the SVG file you want to render and choose **Copy as path**. In others, we can copy the file directories, add **\filename.filetype** and put them in **quotation marks " "** to get ready for our next step.

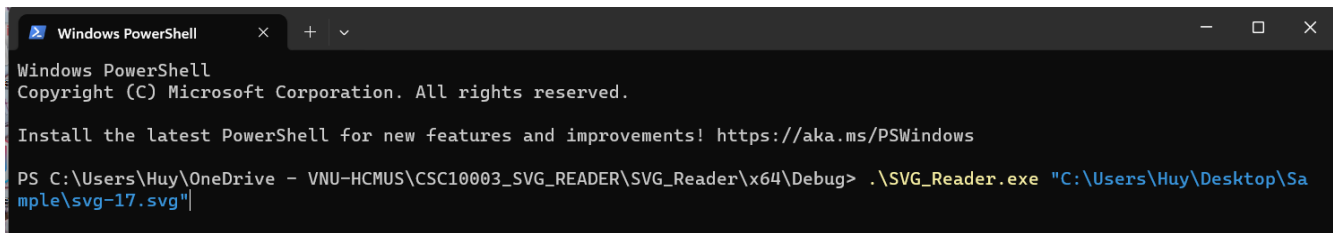
A screenshot of a Windows PowerShell terminal window. The title bar says "Windows PowerShell". The text inside shows the copyright notice for Microsoft Corporation, a link to install PowerShell, and a command prompt. The command entered is: `PS C:\Users\Huy\OneDrive - VNU-HCMUS\CSC10003_SVG_READER\SVG_Reader\x64\Debug> .\SVG_Reader.exe "C:\Users\Huy\Desktop\SAMPLE\svg-17.svg"`

Figure 6: Command-line for SVG Reader

Go to the Command Prompt window that we opened earlier like I mentioned. Input from keyboard the name of our program, **.\SVG\_Reader.exe** and then after that, paste our file directory that we copied to the space besides. Basically, our syntax will be like below.

**.\SVG\_Reader.exe "File Directory"**

After that, our program will parse, render and show the file in a new window. If the file is not accessible or the path is wrong, the program will show a full white screen indicating failed rendering.

## 3.2 Code Distribution

### 3.2.1 Source and Header Files

Header (.h)	Source Code (.cpp)	Usage
SVG_Reader.h	SVG_Reader.cpp	Main source file of the SVG Reader Windows application written in C++, following the Win32 API for creating a Windows GUI application.
Image.h	Image.cpp	Contain the class Image, which reads and draws from the SVG file to the final image that will appear on the screen.
Figure.h	Figure.cpp	Contain the class Figure, which acts as a base class for all the figures to inherit.
FactoryFigure.h	FactoryFigure.cpp	Contain the class FactoryFigure, which creates a figure from the specified type.
Color.h	Color.cpp	Contain the classes Color and Stroke, resembling the color and stroke formatting used for the Figures.
Gradient.h	Gradient.cpp	Contain the classes Gradient and Stop, which resembles a gradient color style.
LinearGradient.h	LinearGradient.cpp	Contain the class LinearGradient, which resembles a linear gradient.
RadialGradient.h	RadialGradient.cpp	Contain the class RadialGradient, which resembles a radial gradient.
Viewbox.h	Viewbox.cpp	Contain the class Viewbox, which resembles the viewbox system that is mapped to the visible area.
Parser.h	Parser.cpp	Contain the class Parser, which processes the SVG file and get their attributes.
Renderer.h	Renderer.cpp	Contain the class Renderer, which renders, prints and open a window to interact with the rendered SVG file.

Figure 7: Code Distribution for Core Classes

Header (.h)	Source Code (.cpp)	Usage
Ellipse.h	Ellipse.cpp	Contain the class Ellipse, which illustrates an Ellipse figure, inherited from Figure.
Circle.h	Circle.cpp	Contain the class Circle, which illustrates a Circle figure inherited from Ellipse, also inherited from Figure.
Line.h	Line.cpp	Contain the class Line, which illustrates a Line figure, inherited from Figure.
Point.h	Point.cpp	Contain the class Point, which illustrates a Point figure.
Polygon.h	Polygon.cpp	Contain the class Polygon, which illustrates a Polygon figure, inherited from Figure.
Polyline.h	Polyline.cpp	Contain the class Polyline, which illustrates a Polyline figure, inherited from Figure.
Rectangle.h	Rectangle.cpp	Contain the class Rectangle, which illustrates a Rectangle figure, inherited from Figure.
Text.h	Text.cpp	Contain the class Text, which illustrates a Text figure, inherited from Figure.
Path.h	Path.cpp	Contain the class Path, which illustrates a Path figure, inherited from Figure.
Group.h	Group.cpp	Contain the class Group, which illustrates a Group figure, inherited from Figure.

Figure 8: Code Distribution for Figure Classes

A figure can appear as different shapes, so each one of these classes resemble each type of shape. We have the Point, which is a standard class for every figure used to present coordinates, but used differently in each. Text is probably the most different from others because it relies on text input, font used, and other characteristics. A rectangle and an ellipse are simple figures, using points respectively as the left corner and center. A circle is a special type of an ellipse. A line, a polyline, and a polygon relies on the concept of a series of points. A path is the fundamental of everything and a group is a special type of figure that contains others, even itself.

### 3.2.2 Libraries and Other Files

Libraries and Others	Usage
<code>iostream</code>	Used for reading from and writing to the standard input and output streams.
<code>vector</code>	Provides <code>std::vector</code> for dynamic arrays, allowing resizable arrays with dynamic memory management.
<code>string</code>	C++ string handling, made easier compared to the traditional C char array.
<code>fstream</code>	Used for reading from and writing to files.
<code>sstream</code>	Allows us to treat strings as streams, convenient for converting, reading and manipulating them.
<code>cmath</code>	Includes various mathematical functions for handling more advanced math.
<code>unordered_map</code>	Provides <code>unordered_map</code> , allowing you to create hash maps for fast key-value pair lookups with constant time complexity.
<code>Lib.h</code>	Include all the libraries and files.
<code>Color.txt</code>	Contains color names and their respective Hex codes.
<code>rapidxml.hpp</code>	Main header file for the RapidXML library, includes essential declarations, configurations, and utilities needed for the implementation of a fast XML parser and Document Object Model for C++.
<code>Resource.h</code> <code>targetver.h</code> <code>framework.h</code>	Contain definitions for resources used in the graphical user interface (GUI) of the application.

Figure 9: Libraries used in SVG Reader

There might be a **sample.svg** as the original SVG file to parse and render. Moreover, icon **.ico** files can appear as the logo for our Windows application. We also included multiple sample SVG files for you to try out our application.



# *Chapter 4*

## Conclusive Statements

### 4.1 Final Remarks

Majoring in Information Technology, everyone understands that continuously learning and evolving is essential in this field where we see rocketing changes in short periods. One of the starting points of our profession is working with applications, and this project has exposed us to a brand new concept, making an image reader.

The SVG Reader project is the result of our combined efforts to create an application that can parse SVG files, produce vector graphics, and maybe manage user interactions using C++ programming and Object-Oriented Programming concepts. We have dissected the software into its constituent parts throughout this study, providing an in-depth examination of the features, functionalities, tools, and libraries utilized.

To keep things clear and succinct, we chose to leave out pure source code and opted for easier descriptions in the report, so we have included detailed explanations of the program's architecture, code structure, and class distribution strategy. Written in C++ and compiled with Visual Studio, the project is evidence of our dedication to developing a reliable and effective SVG Reader application.

## 4.2 Work Description

We are very certain and thankful that every member of the group have good senses of responsibility. With the work that each member has contributed into the group, we can proudly say that every member has definitely worked hard enough to produce the best results possible. Here is the full table of how the work is assigned.

ID	Student	Work	Completion
22127151	Lâm Tiến Huy	Leader. Design UML Class Diagrams. Handle Polygon and code convention. Create 3 demonstration videos. Main writer for the report.	25%
22127108	Trịnh Quốc Hiệp	Initialize the whole project. Read SVG files and process data. Main programmer. Handle Point, Line, Polyline, Parser, Color, Figures, FactoryFigure, Group, Gradient, LinearGradient, and RadialGradient.	27%
22127238	Lê Thành Lợi	Main host in the GitHub repository. Read SVG files and process data. Main programmer. Handle Ellipse, Circle, Text, Renderer, Color, Path, Figures, Image, Gradient, LinearGradient, and RadialGradient.	27%
22127107	Nguyễn Thế Hiển	Co-writer for the report. Handle Rectangle, Viewbox, and interactive features (Zoom, Move, Rotate) Create command-lines for running SVG files.	21%

Figure 10: Work Organization

## References

SVG files: How to create, edit and open them | Adobe. (n.d.). <https://www.adobe.com/creativecloud/file-types/image/vector/svg-file.html>.

What is Class Diagram? (n.d.). Visual Paradigm. Retrieved November 15, 2023, from <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/>

SVG text. (n.d.). [https://www.w3schools.com/graphics/svg\\_text.asp](https://www.w3schools.com/graphics/svg_text.asp)

Vector Data. (n.d.). [https://docs.qgis.org/2.18/en/docs/gentle\\_gis\\_introduction/vector\\_data.html#:~:text=Polyline%20geometries%20are%20made%20up,always%20in%20the%20same%20place.](https://docs.qgis.org/2.18/en/docs/gentle_gis_introduction/vector_data.html#:~:text=Polyline%20geometries%20are%20made%20up,always%20in%20the%20same%20place.)

Stevewhims. (2021, January 7). GDI+ - Win32 apps. Microsoft Learn. <https://learn.microsoft.com/en-us/windows/win32/gdiplus/-gdiplus-gdi-start>