# **Detailed Description of a Vector Path and Rasterization System**

## **1. Introduction**

This report aims to provide a detailed exposition of a Python-based system designed for vector path creation and rasterization. The system's core functionality lies in its ability to translate abstract vector path descriptions into concrete pixel representations, a fundamental process in computer graphics. Vector graphics represent images through mathematical formulas defining geometric shapes, offering the advantage of scalability without any degradation in quality, making them particularly suitable for applications such as logo design and digital illustrations.1 Rasterization, conversely, is the technique of converting these mathematical descriptions into a grid of discrete pixels, which is the format necessary for display on the vast majority of digital screens. This report will thoroughly examine the algorithms and methodologies employed by the system to achieve this crucial conversion.

A key aspect of this system is its architectural design, which strategically employs several well-established design patterns. This deliberate choice facilitates extensibility, allowing for future enhancements and the integration of new features, such as advanced path types or sophisticated rendering techniques. The system currently supports enhanced control over the visual appearance of lines through customizable caps and joins, demonstrating its capability for nuanced graphical output. The distinction between vector and raster graphics is crucial to understanding the system's purpose. While vector graphics excel in scalability and editability due to their mathematical nature, raster graphics are essential for display on pixel-based devices. The system bridges this gap by providing a mechanism to convert vector paths into raster images.

## **2. Code Architecture and Design Patterns**

The system's architecture is inherently object-oriented, with its functionality decomposed into a set of interacting classes. This modular design promotes code organization, reusability, and maintainability. The system leverages several design patterns, each serving a specific purpose in enhancing its structure and flexibility.

### **2.1. Strategy Pattern**

The concept of path elements (lines, curves) and rasterization methods are abstracted through the Strategy Pattern. The PathElement abstract base class defines a contract for different types of path segments, with concrete implementations like LineTo and CubicBezierTo providing specific strategies for their representation and sampling. Similarly, the Rasterizer abstract class outlines the interface for rasterization algorithms, with SimpleRasterizer and AntiAliasedRasterizer offering distinct approaches to the rasterization process. This pattern allows for the easy addition of new path element types or rasterization algorithms without modifying the core system. For instance, if one wished to add support for elliptical arcs, a new class inheriting from PathElement would be created, implementing the sample\_points and transform methods specific to arcs. Similarly, a more advanced rasterizer with features like gradient fills could be added by creating a new class that inherits from Rasterizer.

### **2.2. Composite Pattern**

The Path class embodies the Composite Pattern. A Path object acts as a container, holding a collection of individual PathElement objects. This hierarchical structure allows complex shapes to be built by combining simpler path elements. For example, a complex outline of a character in a font might be composed of numerous line segments and Bézier curves, all managed under a single Path object. The Path class also provides methods that operate on the entire composite structure, such as sample\_points and transform, delegating the specific operations to its constituent PathElement objects. When the sample\_points method is called on a Path object, it iterates through its list of PathElement objects and calls the sample\_points method of each individual element, effectively gathering all the points that define the entire path.

### **2.3. Decorator Pattern**

The StrokeProperties class utilizes the Decorator Pattern. It encapsulates a set of properties that modify the rendering of a path's outline, such as line width, color, and styles for line ends (LineCap) and connections (LineJoin). These properties can be applied to a path to enhance its visual appearance without altering the underlying path geometry. For instance, a user might want to draw the same path with different stroke widths or colors depending on the context, and the StrokeProperties class allows for this without needing to modify the definition of the path itself. The different line cap and join styles offer further visual customization, allowing for control over how line segments end and how they connect at corners.

### **2.4. Factory Pattern**

The PathFactory class implements the Factory Pattern. It provides static methods for creating instances of commonly used path shapes, such as stars and spirals. This pattern centralizes the object creation logic and shields the client code from the complexities of object instantiation. Instead of the user needing to know the specific sequence of move\_to and line\_to calls required to create a star, they can simply call the create\_star method with the desired parameters. This simplifies the process of creating standard shapes and makes the system easier to use.

The strategic application of these design patterns suggests a well-thought-out architecture aimed at promoting flexibility and scalability. The Strategy Pattern allows for easy swapping of algorithms, the Composite Pattern facilitates the creation of complex objects from simpler ones, the Decorator Pattern enables dynamic addition of responsibilities, and the Factory Pattern simplifies object creation. This architectural foundation indicates a system designed for potential future expansion and diverse application scenarios. The separation of concerns achieved through these patterns makes the codebase more organized and easier to understand and maintain.

## **3. Geometric Primitives (Point and Vector Utilities)**

The Point class is a fundamental data structure representing a location in a two-dimensional Cartesian coordinate system. It holds two floating-point attributes: x for the horizontal coordinate and y for the vertical coordinate.2 This class serves as the basic building block for defining the geometry of paths and shapes within the system.

The distance\_to(other: 'Point') -> float method calculates the **Euclidean distance** between the current Point and another Point object. This is a standard mathematical formula derived from the Pythagorean theorem: d=(x2​−x1​)2+(y2​−y1​)2​. This method is crucial for various calculations within the system, such as determining the length of line segments or approximating the length of curves for sampling purposes.

The transform(self, sx: float, k: float, l: float, sy: float, tx: float, ty: float) -> 'Point' method applies a 2D **affine transformation** to the point. Affine transformations are linear transformations (scaling, shearing, rotation) followed by a translation. The parameters represent the elements of a 2x2 transformation matrix and a translation vector:

* sx: Scaling factor along the x-axis.
* k: Shear factor along the x-axis.
* l: Shear factor along the y-axis.
* sy: Scaling factor along the y-axis.
* tx: Translation along the x-axis.
* ty: Translation along the y-axis.

The new coordinates (x′,y′) of the transformed point are calculated as:

* x′=sx⋅x+k⋅y+tx
* y′=l⋅x+sy⋅y+ty

This method is fundamental for manipulating the position, size, and orientation of geometric objects. For example, scaling a path involves applying a scaling transformation to all its constituent points. Similarly, moving a path to a different location on the canvas is achieved by applying a translation transformation. Rotations can be achieved by combining scaling and shearing transformations. The ability to apply affine transformations is a cornerstone of vector graphics systems, allowing for flexible and powerful manipulation of shapes.1

## **4. Path Elements (Strategy Pattern)**

The abstract class PathElement serves as an interface for all types of path segments that can constitute a Path. It defines two abstract methods that must be implemented by its concrete subclasses:

* sample\_points(self, resolution: float) -> List[Point]: This method is responsible for generating a sequence of Point objects that approximate the continuous path element. The resolution parameter controls the density of these sampled points, effectively determining the level of detail captured during the discretization process. A lower resolution will result in fewer sample points and a potentially less accurate representation of the curve, while a higher resolution will yield more points and a more precise representation.
* transform(self, sx: float, k: float, l: float, sy: float, tx: float, ty: float) -> 'PathElement': This method applies a 2D affine transformation to the path element itself, returning a new PathElement object that represents the transformed segment. This ensures that when a Path object is transformed, all its constituent path elements are also transformed accordingly.

### **4.1. LineTo Class**

The LineTo class represents a straight line segment connecting a start Point to an end Point. It is a concrete implementation of the PathElement interface.

The sample\_points method in the LineTo class implements **linear interpolation** 6 to generate points along the line. First, it calculates the total Euclidean distance between the start and end points using the distance\_to method of the Point class. Then, based on the provided resolution, it determines the number of points to sample along the line, ensuring at least two points (the start and end points) are included. The number of points is calculated as the maximum of 2 and the integer part of the distance divided by the resolution. The coordinates of the intermediate points are calculated using the linear interpolation formula:

* P(t)=(1−t)⋅Pstart​+t⋅Pend​, where t ranges from 0 to 1 in discrete steps determined by the number of points to sample. This can be expanded to:
  + x=start.x+t⋅(end.x−start.x)
  + y=start.y+t⋅(end.y−start.y) Here, t is the interpolation parameter, ranging from 0 at the start point to 1 at the end point. The intermediate values of t are evenly spaced between 0 and 1 to generate the sample points. This method effectively divides the line segment into smaller, discrete segments for rasterization.

The transform method creates a new LineTo object where both the start and end points have been transformed by applying the provided affine transformation parameters to their respective coordinates using the transform method of the Point class. This ensures that the line segment is correctly positioned, scaled, sheared, and rotated when the overall path is transformed.

The sample\_points method in LineTo demonstrates a fundamental technique for representing continuous geometric shapes using a finite set of discrete points, a necessary step for rasterization. The resolution parameter directly influences the fidelity of this discrete representation. Linear interpolation 11 is a widely used algorithm in computer graphics for finding intermediate values, effectively "filling in the gaps" between two known endpoints to create the illusion of a continuous line. The formula for linear interpolation provides a straightforward way to determine the coordinates of any point along the straight line segment defined by two endpoints.

### **4.2. CubicBezierTo Class**

The CubicBezierTo class represents a cubic Bézier curve, which is defined by four control points: a start point, two intermediate control points (control1 and control2), and an end point.1 These control points determine the shape of the curve. The curve starts at the start point, moves towards control1, then is pulled towards control2, and finally arrives at the end point. The intermediate control points generally do not lie on the curve itself but influence its direction and curvature.

The \_cubic\_bezier\_point(self, t: float) -> Point method calculates a single point on the cubic Bézier curve for a given parameter value t, where 0≤t≤1. The mathematical formula for a cubic Bézier curve 1 is given by:

* B(t)=(1−t)3⋅P0​+3(1−t)2⋅t⋅P1​+3(1−t)⋅t2⋅P2​+t3⋅P3​, where P0​ is the start point, P1​ is control1, P2​ is control2, and P3​ is the end point. This formula can be expanded for the x and y coordinates separately. The terms (1−t)3, 3(1−t)2t, 3(1−t)t2, and t3 are known as the Bernstein basis polynomials for a cubic Bézier curve.

The sample\_points method approximates the length of the cubic Bézier curve by summing the Euclidean distances between the consecutive control points. This approximation is then used to determine the number of points to sample along the curve based on the provided resolution. The method iterates through a range of t values (from 0 to 1) to generate the required number of sample points by calling the \_cubic\_bezier\_point method for each t. The number of points is calculated as the maximum of 2 and the integer part of the approximate length divided by the resolution. This provides a way to generate a discrete set of points that lie along the Bézier curve, which can then be used for rasterization.

The transform method creates a new CubicBezierTo object by applying the given affine transformation to each of the four defining control points (start, control1, control2, end) using the transform method of the Point class. This ensures that the Bézier curve is correctly transformed along with the rest of the path.

Cubic Bézier curves are a fundamental tool in vector graphics for creating smooth and versatile curves. They are defined by their endpoints and two control points that influence the curve's shape and direction. The parameter t allows for parametric evaluation of the curve. The approximation of the curve's length is a simplification; more accurate methods exist but often involve more complex calculations. The **de Casteljau's algorithm** 18 provides an alternative, and often more numerically stable, method for evaluating points on a Bézier curve. This algorithm works by recursively performing linear interpolations between the control points.

## **5. Path (Composite Pattern)**

The Path class acts as a container for a sequence of PathElement objects, effectively representing a complex path composed of multiple segments. It maintains a list of elements (the PathElement objects), the current\_point (the current drawing cursor position), and a boolean flag “closed” indicating whether the path is closed. The Path class provides a way to build up complex shapes by sequentially adding different types of path elements.

The Path class provides several methods for constructing paths:

* move\_to(self, x: float, y: float) -> 'Path': Sets the current\_point to the specified coordinates without creating a new path element. This is analogous to lifting a pen from the drawing surface and moving it to a new starting position.
* line\_to(self, x: float, y: float) -> 'Path': Creates a new LineTo object connecting the current\_point to the specified coordinates and appends it to the elements list. The current\_point is then updated to the new end point. If no current\_point is set, this method implicitly calls move\_to with the given coordinates.
* cubic\_bezier\_to(self, cx1: float, cy1: float, cx2: float, cy2: float, x: float, y: float) -> 'Path': Creates a new CubicBezierTo object using the current\_point as the start point, the provided control points (cx1, cy1 and cx2, cy2), and the specified end point (x, y). This new element is appended to the elements list, and the current\_point is updated to the new end point. Similar to line\_to, if no current\_point is set, this method implicitly calls move\_to with the given end point coordinates.
* close(self) -> 'Path': If the path contains any elements and a current\_point is defined, this method adds a LineTo element connecting the current\_point back to the starting point of the first PathElement in the elements list, effectively closing the path. The closed attribute is then set to True. This creates a closed shape that can be filled.

The sample\_points(self, resolution: float) -> List[Point] method iterates through all the PathElement objects in the elements list and calls the sample\_points method of each element with the given resolution. It then aggregates all the sampled points from all the elements into a single list. This provides a unified way to obtain a discrete representation of the entire path, regardless of the types of segments it contains.

The get\_edges(self) -> List] method serves to convert the potentially curved path into a series of straight line segments. It iterates through each PathElement in the elements list and calls its sample\_points method with a fixed resolution of 0.25. It then takes these sampled points and creates a list of edges, where each edge is a tuple of two consecutive Point objects. This representation is often useful for polygon filling algorithms that operate on straight line segments.

The transform(self, sx: float, k: float, l: float, sy: float, tx: float, ty: float) -> 'Path' method creates a new Path object. It then iterates through the elements of the original path, applying the given affine transformation to each element by calling its transform method, and appends the transformed element to the new path's elements list. If the original path had a current\_point, it is also transformed and set as the current\_point of the new path. The closed status of the path is also copied to the new path. This ensures that transformations applied to the Path object propagate to all its constituent parts.

The copy(self) -> 'Path' method creates a completely independent copy of the Path object and all its constituent PathElement objects using the deepcopy function from the copy module. This is important when you need to modify a path without affecting the original.

## **6. Fill Properties and Rules**

The abstract class FillRule defines the interface for determining the interior of a path for filling purposes. It declares a single abstract method: is\_inside(self, crossings: int) -> bool. The crossings parameter represents a value that is determined based on how a ray cast from a point interacts with the path's edges, and its interpretation depends on the specific fill rule being implemented. Different fill rules can lead to different interpretations of which areas are considered "inside" a path, especially for complex or self-intersecting paths.

### **6.1. EvenOddFillRule**

This class implements the **even-odd fill rule**.23 According to this rule, a point is considered to be inside a shape if a ray drawn from that point to infinity crosses an odd number of edges of the shape. If the number of crossings is even, the point is outside. This rule is independent of the direction of the path segments.

The is\_inside(self, crossings: int) -> bool method in EvenOddFillRule returns True if the provided crossings count is odd (crossings % 2 == 1), and False otherwise. The crossings value in this context represents the number of times a scanline intersects with the edges of the path.

The even-odd rule is particularly useful for handling complex paths, including those with self-intersections and nested shapes. It effectively defines the "inside" by counting the number of times the path boundary is crossed. This rule is often used in graphics applications to create effects like holes in shapes.23 For example, consider two concentric circles. Using the even-odd rule, the area between the two circles would be considered inside, while the area inside the inner circle would be outside.

### **6.2. NonZeroWindingFillRule**

This class implements the **non-zero winding rule**.23 This rule determines whether a point is inside a path by considering the direction in which the path segments cross a ray emanating from the point. A winding number is calculated by initializing a counter to zero and then traversing the edges of the path. For each edge that crosses the ray from left to right (as viewed from the point), the counter is decremented, and for each edge that crosses from right to left, the counter is incremented. If the final winding number is non-zero, the point is considered inside; otherwise, it is outside. This rule takes into account the orientation of the path segments.

The is\_inside(self, crossings: int) -> bool method in NonZeroWindingFillRule returns True if the provided crossings value (which represents the winding number) is not equal to zero.

The non-zero winding rule is another common method for defining the interior of a path in vector graphics. It differs from the even-odd rule in its handling of path direction and is often the default fill rule in standards like SVG.30 This rule can produce different filling results for self-intersecting or overlapping paths compared to the even-odd rule.23 Using the concentric circles example again, if both circles are drawn in the same direction, the non-zero winding rule would consider both the area between the circles and the area inside the inner circle to be inside.

The FillProperties class is a simple data class that holds the properties associated with filling a path. It has two attributes: color, which is a tuple representing the RGBA color to fill with, and rule, which is an instance of a FillRule subclass specifying the algorithm to use for determining the fill region.

## **7. Stroke Properties (Decorator Pattern)**

The StrokeProperties class encapsulates the properties that define how the outline of a path is rendered (stroked). These properties control the visual appearance of the line drawn along the path.

* width: A floating-point number specifying the thickness of the stroke. A value of 1.0 represents a standard one-pixel-wide line, while larger values result in thicker lines.37
* color: A tuple representing the RGBA color of the stroke. The tuple contains four integer values ranging from 0 to 255, representing the red, green, blue, and alpha (opacity) components of the color.
* line\_cap: An enumeration (LineCap) that defines the shape to be used at the ends of open subpaths.4 The possible values are:  
  **Table 1: Line Cap Styles**

| **Enum Member** | **Description** |
| --- | --- |
| BUTT | The stroke ends with a flat edge perpendicular to the line at the endpoint. This is the default style. |
| ROUND | A semicircle with a radius of half the stroke width is added to the endpoint. This creates a rounded end. |
| SQUARE | A rectangle with a width of half the stroke width and height equal to the stroke width is added beyond the endpoint. This creates a square end that extends slightly beyond the actual endpoint. |

* line\_join: An enumeration (LineJoin) that defines the shape to be used at the corners where two line segments meet.4 The possible values are:  
  **Table 2: Line Join Styles**

| **Enum Member** | **Description** |
| --- | --- |
| MITER | The outer edges of the joining lines are extended to meet at a sharp corner. The length of this extension is limited by the miter\_limit. |
| ROUND | A circular arc with a radius of half the stroke width connects the outer edges. This creates a smooth, rounded corner. |
| BEVEL | A straight line connects the outer corners of the joining lines, creating a flat, angled corner. |

* miter\_limit: A floating-point number that specifies the maximum allowed ratio of the miter length to the stroke width for miter joins. If the miter ratio exceeds this limit, the join is typically rendered as a bevel join to prevent excessively long and sharp spikes that can occur with very acute angles.43

## **8. Rasterizer (Strategy Pattern)**

The abstract class Rasterizer defines the interface for classes that perform the rasterization of Path objects. It declares the following abstract methods:

* rasterize(...): The core method for converting a vector path into a raster image.
* fill\_path(...): Responsible for filling the interior of a path.
* stroke\_path(...): Responsible for drawing the outline of a path.
* get\_buffer(...): Returns the current rasterized image.

### **8.1. SimpleRasterizer**

The SimpleRasterizer class provides a basic implementation of a rasterizer. Its \_\_init\_\_ method initializes a NumPy array self.canvas representing the image buffer.

The rasterize method orchestrates the rasterization process by calling fill\_path and stroke\_path if the respective properties are provided.

The fill\_path method implements a **scanline polygon filling algorithm**.48 This algorithm works by iterating through each horizontal line (scanline) of the image. For each scanline, it determines which pixels lie inside the polygon defined by the path. This is typically done by finding the intersections of the scanline with the edges of the polygon, sorting these intersections by their x-coordinate, and then filling the pixels between pairs of intersections. The \_generate\_runs method is responsible for finding these spans of pixels to fill based on the chosen fill rule.

The stroke\_path method renders the outline of the path. It samples points along the path and then connects these points with line segments. It takes into account the StrokeProperties to determine the width, color, line caps, and line joins of the stroke. The draw\_cap function handles the rendering of the line ends according to the line\_cap property, while the draw\_join function manages the appearance of corners based on the line\_join property.

The \_draw\_line\_segment method draws a thick line segment. It calculates the perpendicular vectors to the line segment to determine the boundaries of the thick line and then fills the resulting polygon. It also draws the core of the line.

The \_sample\_line method uses linear interpolation to generate intermediate points along a line segment.

The \_perpendicular\_vector method calculates a vector perpendicular to a given line segment, used for drawing thick lines.

The \_draw\_circle method draws a filled circle, used for round line caps and joins.

The \_fill\_polygon method fills a polygon using a scanline approach. It finds the intersections of each scanline with the polygon's edges and fills the pixels between the intersections.

The \_angle\_between\_vectors method calculates the angle between two vectors, used for determining the miter angle in line joins.

The \_plot\_pixel method sets the color of a single pixel on the canvas, handling alpha blending for transparency.

The \_generate\_runs method is the core of the scanline filling algorithm. It identifies the horizontal spans of pixels that should be filled on each scanline based on the path's edges and the selected fill rule. It calculates intersections between the scanline and the path's edges and uses the winding number and the FillRule to determine which spans are inside the path.

### **8.2. AntiAliasedRasterizer**

The AntiAliasedRasterizer class extends SimpleRasterizer to implement **anti-aliasing** using the **supersampling** technique.55 Anti-aliasing is a technique used to reduce the jagged edges (aliasing artifacts) that can appear when rendering lines and curves on a pixel grid. Supersampling achieves this by rendering the image at a higher resolution than the target resolution and then downsampling it.

In its \_\_init\_\_ method, it creates a canvas with twice the width and height of the desired output. The rasterize method first renders the path onto this higher-resolution canvas using the parent class's rasterization methods. Then, it iterates through the pixels of the target resolution. For each target pixel, it averages the colors of the corresponding 2x2 block of pixels from the high-resolution canvas. This averaging process smooths out the edges and reduces the appearance of jaggies. The resulting averaged color is then assigned to the pixel in the final, lower-resolution canvas. This technique significantly improves the visual quality of the rendered paths, especially for diagonal lines and curves.

## **9. Path Factory (Factory Pattern)**

The PathFactory class provides static methods for creating common geometric shapes as Path objects, encapsulating the creation logic.

* create\_star(cx: float, cy: float, outer\_radius: float, inner\_radius: float, points: int) -> Path: This method generates a star shape centered at (cx, cy). It calculates the vertices of the star using the given outer and inner radii and the specified number of points. It then constructs a Path object by moving to the first vertex and drawing lines to the subsequent vertices, alternating between the outer and inner radii to create the star's points. Finally, it closes the path.
* create\_spiral(cx: float, cy: float, start\_radius: float, end\_radius: float, turns: float, segments: int) -> Path: This method creates a spiral centered at (cx, cy). It generates points along the spiral by interpolating the radius from start\_radius to end\_radius over the specified number of turns and segments. It then constructs a Path by moving to the first point and drawing lines to the subsequent points, creating the spiral effect.

## **10. Utility Functions**

The save\_to\_png(canvas: np.ndarray, filename: str, width: int = None, height: int = None) -> None function takes a NumPy array representing the rasterized image and saves it as a PNG file. It uses the Pillow (PIL) library for image manipulation. If optional width and height parameters are provided, the image is resized before saving, using a high-quality resampling filter (Image.Resampling.LANCZOS). The function ensures that the filename has the ".png" extension.

## **11. Example Usage (main function)**

The main function demonstrates how to use the vector path and rasterization system. It creates a simple path using move\_to, line\_to, and cubic\_bezier\_to methods, and also uses the PathFactory to create a star and a spiral. It then instantiates both SimpleRasterizer and AntiAliasedRasterizer to show the difference in rendering. Various StrokeProperties and FillProperties objects are created with different parameters to customize the appearance of the paths. The rasterize method of the rasterizers is called with different combinations of paths and properties to generate the raster images. Finally, the save\_to\_png function is used to save the resulting images to files. The example also specifically demonstrates the difference between the EvenOddFillRule and the NonZeroWindingFillRule by filling the same path with each rule and saving the results, highlighting how these rules can affect the filled regions of overlapping or self-intersecting paths.

## **12. Conclusion**

The provided Python code offers a robust and well-structured system for creating and rasterizing vector paths. It effectively applies fundamental computer graphics algorithms and design patterns to achieve its functionality. The system's architecture, utilizing patterns like Strategy, Composite, Decorator, and Factory, promotes flexibility, extensibility, and maintainability. Key algorithms implemented include linear interpolation for rendering lines, the cubic Bézier curve formula for drawing curves, a scanline algorithm for filling polygons (supporting both even-odd and non-zero winding rules), and supersampling for enhancing image quality through anti-aliasing. The inclusion of customizable stroke and fill properties allows for a wide range of visual styles. Further development could explore the addition of more advanced path elements, rendering techniques such as gradients and patterns, and performance optimizations for handling more complex scenes efficiently.

#### Works cited

1. Bézier curve - Wikipedia, accessed on May 10, 2025, <https://en.wikipedia.org/wiki/B%C3%A9zier_curve>
2. Bezier curve - The Modern JavaScript Tutorial, accessed on May 10, 2025, <https://javascript.info/bezier-curve>
3. Computer Graphics Curve in Computer Graphics | GeeksforGeeks, accessed on May 10, 2025, <https://www.geeksforgeeks.org/computer-graphics-curve-in-computer-graphics/>
4. COMPUTER GRAPHICS – ( MODULE – II ) Output Primitives: Line Drawing Algorithms – Loading the Frame Buffer, accessed on May 10, 2025, <https://www.aagasc.edu.in/cs/CG%20Notes%20Module2.pdf>
5. Introduction to Computer Graphics, Section 2.2 -- Shapes, accessed on May 10, 2025, <https://math.hws.edu/graphicsbook/c2/s2.html>
6. Linear Interpolation Formula - Derivation, Formulas, Examples - Cuemath, accessed on May 10, 2025, <https://www.cuemath.com/linear-interpolation-formula/>
7. Linear Interpolation Formula - BYJU'S, accessed on May 10, 2025, <https://byjus.com/linear-interpolation-formula/>
8. Linear Interpolation Formula | GeeksforGeeks, accessed on May 10, 2025, <https://www.geeksforgeeks.org/linear-interpolation-formula/>
9. Linear Interpolation Formula: Definition, Formula, Solved Examples, accessed on May 10, 2025, <https://www.toppr.com/guides/maths-formulas/linear-interpolation-formula/>
10. How to implement linear interpolation in Python? - GeeksforGeeks, accessed on May 10, 2025, <https://www.geeksforgeeks.org/how-to-implement-linear-interpolation-in-python/>
11. Linear interpolation - Wikipedia, accessed on May 10, 2025, <https://en.wikipedia.org/wiki/Linear_interpolation>
12. Cubic Bezier Curve Implementation in C - GeeksforGeeks, accessed on May 10, 2025, <https://www.geeksforgeeks.org/cubic-bezier-curve-implementation-in-c/>
13. Cubic Bezier Curve - GeoGebra, accessed on May 10, 2025, <https://www.geogebra.org/m/zc3mMQjU>
14. Cubic Bézier: from math to motion - The Blog of Maxime Heckel, accessed on May 10, 2025, <https://blog.maximeheckel.com/posts/cubic-bezier-from-math-to-motion/>
15. Understanding of cubic Bézier curves in one dimension - Math Stack Exchange, accessed on May 10, 2025, <https://math.stackexchange.com/questions/2571471/understanding-of-cubic-b%C3%A9zier-curves-in-one-dimension>
16. Finding the formula for Bezier curve ratios (hull/point : point/baseline) - MathOverflow, accessed on May 10, 2025, <https://mathoverflow.net/questions/122257/finding-the-formula-for-bezier-curve-ratios-hull-point-point-baseline>
17. 6.4. Parametric curves. We next study the approximation of curves which can not be expressed as a function of one coordinate var - rutgers math, accessed on May 10, 2025, <https://ow3.math.rutgers.edu/~falk/math373/lecture10.pdf>
18. pages.mtu.edu, accessed on May 10, 2025, <https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/de-casteljau.html#:~:text=The%20fundamental%20concept%20of%20de,A%20and%20B%20is%20u).>
19. Finding a Point on a Bézier Curve: De Casteljau's Algorithm, accessed on May 10, 2025, <https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/de-casteljau.html>
20. De Casteljau's algorithm - Wikipedia, accessed on May 10, 2025, <https://en.wikipedia.org/wiki/De_Casteljau%27s_algorithm>
21. 3. De Casteljau's algorithm | Animation | Computer animation | Khan Academy - YouTube, accessed on May 10, 2025, <https://www.youtube.com/watch?v=TlxSlwQiYgM>
22. de Casteljau's Algorithm - TOM ROCKS MATHS, accessed on May 10, 2025, <https://tomrocksmaths.com/wp-content/uploads/2021/05/de-casteljaus-algorithm.pdf>
23. How does usesEvenOddFillRule work? - Stack Overflow, accessed on May 10, 2025, <https://stackoverflow.com/questions/14840563/how-does-usesevenoddfillrule-work>
24. Even–odd rule - Wikipedia, accessed on May 10, 2025, <https://en.wikipedia.org/wiki/Even%E2%80%93odd_rule>
25. The Winding Order of the Fill Rule — Using SVG with CSS3 and HTML5, accessed on May 10, 2025, <https://oreillymedia.github.io/Using_SVG/extras/ch06-fill-rule.html>
26. odd even rule computer graphics | Inside Outside Test For Area Filling Algorithm - YouTube, accessed on May 10, 2025, <https://www.youtube.com/watch?v=gEmO2BD-Cnk>
27. Inside-outside Test Odd-Even Rule Nonzero Winding Number Rule - Raghunathpur College, accessed on May 10, 2025, <https://elearning.raghunathpurcollege.ac.in/files/8244622C16351874620.pdf>
28. Computer Graphics Inside-Outside Test - Tutorialspoint, accessed on May 10, 2025, <https://www.tutorialspoint.com/computer_graphics/computer_graphics_inside_outside_test.htm>
29. inside outside test in computer graphics | Area filling ODD-EVEN RULE, NON ZERO WINDING NUMBER RULE - YouTube, accessed on May 10, 2025, <https://www.youtube.com/watch?v=l_LbdBKEZCU>
30. Nonzero-rule - Wikipedia, accessed on May 10, 2025, <https://en.wikipedia.org/wiki/Nonzero-rule>
31. Winding | PDF - Scribd, accessed on May 10, 2025, <https://www.scribd.com/doc/100579360/Winding>
32. Defining winding rules · OpenFL Developer's Guide - Books, accessed on May 10, 2025, <https://books.openfl.org/openfl-developers-guide/using-the-drawing-api/advanced-use-of-the-drawing-api/defining-winding-rules.html>
33. non zero winding rule in computer graphics | Inside Outside Test For Area Filling Algorithm, accessed on May 10, 2025, <https://www.youtube.com/watch?v=4OLGoYnBh5o>
34. 5.5- An Inside Test Of Polygon By Winding No. Method Or Non Zero Winding No. In Computer Graphics - YouTube, accessed on May 10, 2025, <https://www.youtube.com/watch?v=ZaiAjBd9Mtc>
35. Non-zero winding rule - YouTube, accessed on May 10, 2025, <https://www.youtube.com/watch?v=y6SnS3jHPWk>
36. Non-Zero Rule - WebGlossary.info, accessed on May 10, 2025, <https://webglossary.info/terms/non-zero-rule/>
37. Unit – II Line Attributes - Wavoo Wajeeha Women's College, accessed on May 10, 2025, <https://wavoowajeehacollege.in/assets/pdf/e-content/cs/UNIT%202%20-%20LINE%20ATTRIBUTES.pdf>
38. What is the use of LineCap and LineJoin properties for line series? - Syncfusion support, accessed on May 10, 2025, <https://support.syncfusion.com/kb/article/4970/what-is-the-use-of-linecap-and-linejoin-properties-for-line-series>
39. Stroke Line Cap - Javafx - Tutorialspoint, accessed on May 10, 2025, <https://www.tutorialspoint.com/javafx/stroke_line_cap.htm>
40. www.geeksforgeeks.org, accessed on May 10, 2025, <https://www.geeksforgeeks.org/html-canvas-linecap-property/#:~:text=butt%3A%20It%20is%20the%20default,each%20end%20of%20the%20line.>
41. HTML canvas lineCap Property - GeeksforGeeks, accessed on May 10, 2025, <https://www.geeksforgeeks.org/html-canvas-linecap-property/>
42. JavaFX Stroke Line Cap Property - Tutorialspoint, accessed on May 10, 2025, <https://www.tutorialspoint.com/javafx/javafx_stroke_line_cap_property.htm>
43. CSS | stroke-linejoin Property - GeeksforGeeks, accessed on May 10, 2025, <https://www.geeksforgeeks.org/css-stroke-linejoin-property/>
44. www.tutorialspoint.com, accessed on May 10, 2025, <https://www.tutorialspoint.com/javafx/stroke_line_join.htm#:~:text=Bevel%20%E2%88%92%20The%20bevel%20join%20is,edges%20of%20the%20shape%20(StrokeLineJoin.>
45. HTML canvas lineJoin Property - GeeksforGeeks, accessed on May 10, 2025, <https://www.geeksforgeeks.org/html-canvas-linejoin-property/>
46. Stroke Line Join - Javafx - Tutorialspoint, accessed on May 10, 2025, <https://www.tutorialspoint.com/javafx/stroke_line_join.htm>
47. CS2401-COMPUTER GRAPHICS UNIT I 2D PRIMITIVES PART - A 1. Define persistence. Persistence is defined as the time it takes the em, accessed on May 10, 2025, <http://www.amsheela.org.in/dwn/cse/IV%20Year/CG2.pdf>
48. Scan-line Polygon Filling in C - Scaler Topics, accessed on May 10, 2025, <https://www.scaler.com/topics/scan-line-polygon-fill-algorithm/>
49. Scan Line Algorithm for Polygon Filling in Computer Graphics - Tutorialspoint, accessed on May 10, 2025, <https://www.tutorialspoint.com/computer_graphics/computer_graphics_scan_line_algorithm.htm>
50. Polygon Filling Algorithm in Computer Graphics - Tutorialspoint, accessed on May 10, 2025, <https://www.tutorialspoint.com/computer_graphics/polygon_filling_algorithm.htm>
51. Scan-line Polygon filling using OPENGL in C - GeeksforGeeks, accessed on May 10, 2025, <https://www.geeksforgeeks.org/scan-line-polygon-filling-using-opengl-c/>
52. Scanline Fill Algorithm - Computer Science | UC Davis Engineering, accessed on May 10, 2025, <https://web.cs.ucdavis.edu/~ma/ECS175_S00/Notes/0411_b.pdf>
53. Intro to Computer Graphics: Polygon Filling, accessed on May 10, 2025, <https://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/PolygonFilling.html>
54. Scanline Fill Algorithm, accessed on May 10, 2025, <https://scholar.cu.edu.eg/?q=ehesham/files/scanlinefill.pdf>
55. Supersampling - Cloudinary, accessed on May 10, 2025, <https://cloudinary.com/glossary/supersampling>
56. Supersampling - Wikipedia, accessed on May 10, 2025, <https://en.wikipedia.org/wiki/Supersampling>
57. Anti-Aliasing | Cycling '74 Documentation, accessed on May 10, 2025, <https://docs.cycling74.com/learn/articles/02-antialiasing/>
58. Antialiasing methods, accessed on May 10, 2025, <https://web.cs.wpi.edu/~matt/courses/cs563/talks/antialiasing/methods.html>
59. DLSS – Deep Learning Super Sampling | GeeksforGeeks, accessed on May 10, 2025, <https://www.geeksforgeeks.org/dlss-deep-learning-super-sampling/>
60. How does supersampling work? Why does rendering a bigger image and projecting on a smaller yield a better looking image than native resolution? : r/oculus - Reddit, accessed on May 10, 2025, <https://www.reddit.com/r/oculus/comments/878scp/how_does_supersampling_work_why_does_rendering_a/>