UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

CSCE604029 • Computer Graphics

Semester Gasal 2025/2026
Fakultas Ilmu Komputer, Universitas Indonesia

Assignment 1: Raterization Pipeline

Deadline: 2 October 2025 pukul 23.59 WIB

Alignment with CPMK

- Menguasai konsep-konsep dasar grafika komputer secara teoritis seperti model kamera, properti warna dan cahaya, rendering, iluminasi, dan geometri.
- Dapat mendesain solusi grafika komputer untuk pembentukan citra digital menggunakan teknik pemodelan dasar, terutama menggunakan metode berbasis rasterization dan ray tracing.

1 Introduction

This assignment focuses on implementing a **2D SVG software rasterizer**, which is a fundamental computer graphics system that converts vector graphics into pixel images. Unlike hardware-accelerated rendering (GPU-based), software rasterization is performed entirely on the CPU, giving you direct control over every pixel and a deep understanding of the rendering pipeline.

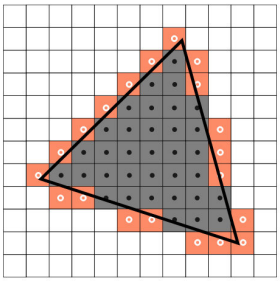


Figure 1: Rasterized Vector Triangle

1.1 What is Rasterization?

Rasterization is the process of converting geometric primitives (lines, triangles, curves) defined in continuous coordinate spaces into discrete pixels on a screen or image buffer. This process involves several key challenges:

1.1.1 Coordinate Transformation

In 2D computer graphics several coordinate systems are used in sequence. Object (local) space is where a shape is defined (for example, a unit circle at the origin). World (canvas) space places all shapes together by translation, rotation, and scaling. Normalized Device Coordinates (NDC) are a resolution-independent square $[-1,1]\times[-1,1]$ obtained after a viewing/normalization step so geometry no longer depends on pixel dimensions. Screen space consists of integer pixel coordinates $[0,W)\times[0,H)$ on the final image.

All stages are affine transforms in homogeneous 2D, represented by 3×3 matrices. If $\mathbf{M}$ maps object→world, $\mathbf{V}$ maps world→NDC, and $\mathbf{S}$ maps NDC→screen, then for a point $\mathbf{p}=(x,y,1)^{\mathsf{T}}$:

$$\mathbf{p}_{\text{screen}} = \mathbf{S}\,\mathbf{V}\,\mathbf{M}\,\mathbf{p}.$$

A convenient 2D normalization is to translate the window center (x,y) to the origin and scale by $1/\text{span}$ so the square $[x-\text{span},x+\text{span}]\times[y-\text{span},y+\text{span}]$ maps to $[-1,1]^2$:

$$\mathbf{V}_{2D} = \begin{bmatrix} \frac{1}{\text{span}} & 0 & -\frac{x}{\text{span}} \\ 0 & \frac{1}{\text{span}} & -\frac{y}{\text{span}} \\ 0 & 0 & 1 \end{bmatrix}.$$

The viewport matrix $\mathbf{S}$ then maps NDC $(u,v)\in[-1,1]^2$ to pixels:

$$x_{\text{pix}} = \tfrac{W}{2}(u+1), \qquad y_{\text{pix}} = \tfrac{H}{2}(v+1).$$

Matrix multiplication order and the chosen row/column convention must be consistent to avoid unintended flips or offsets.

Textbook/OpenGL diagrams often show *local* → *world* → *view* → *clip* → *screen*. In 3D, a view matrix re-expresses coordinates in camera space and a projection matrix sends them to clip space; NDC are obtained after the perspective divide. In our 2D orthographic setting there is no perspective, so clip = NDC and the "camera" reduces to a world-window normalization. Consequently we fold view+projection into $\mathbf{V}_{2D}$, yielding the shorter chain object → world → NDC → screen used here.

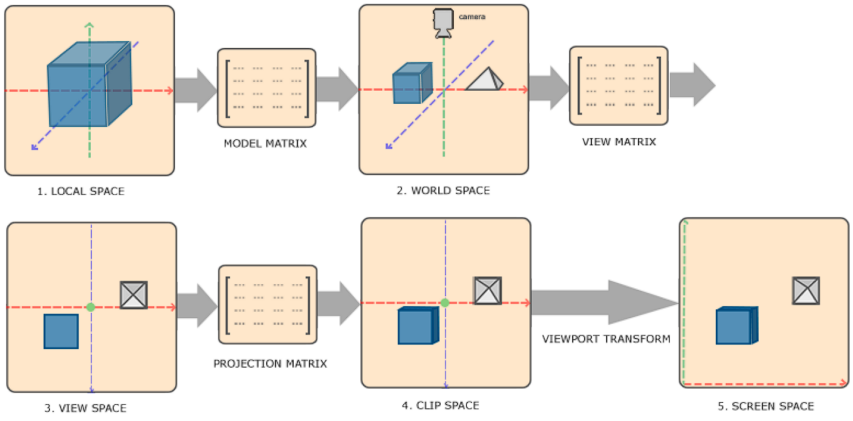


Figure 2: Common 3D pipeline (local → world → view → clip → screen). In 2D orthographic rendering we fold view+projection into a single normalization $\mathbf{V}_{2D}$, so clip = NDC and the chain simplifies to object → world → NDC → screen.

1.1.2 Primitive Decomposition

Rendering is most robust when complex 2D shapes are reduced to triangles. Any convex quadrilateral becomes two triangles; a simple (non self-intersecting) polygon can be tessellated with ear clipping. Triangles are always convex and determined by three points, so inside tests are straightforward and attribute interpolation (for colors or texture coordinates) is well behaved. Once shapes are represented as a triangle list, a single triangle routine can handle all filled geometry.

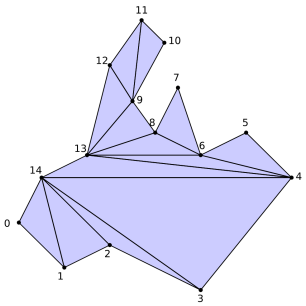


Figure 3: Ear clipping triangulation

1.1.3 Pixel Coverage

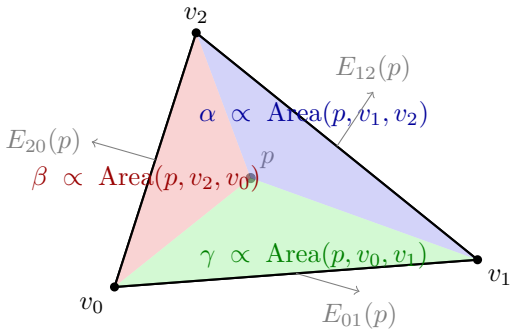
Determining whether a discrete sample lies inside a continuous triangle can be done efficiently with edge functions. For vertices v_0, v_1, v_2 and any point p,

$$E_{ab}(p) = (p - a) \times (b - a) = (p_x - a_x)(b_y - a_y) - (p_y - a_y)(b_x - a_x).$$

The signed twice-area is $A = E_{01}(v_2)$. For each sample center $p = (x + 0.5, y + 0.5)$ inside a conservative integer bounding box, evaluate $E_{01}(p), E_{12}(p), E_{20}(p)$; if they share the sign of A, the sample is inside. Barycentric coordinates follow directly,

$$(\alpha, \beta, \gamma) = \tfrac{1}{A}\big(E_{12}(p), E_{20}(p), E_{01}(p)\big), \qquad \alpha + \beta + \gamma = 1,$$

and are useful for interpolating per-vertex attributes. Degenerate cases with $|A| \approx 0$ are treated as empty.



$$\alpha = \frac{\text{Area}(p, v_1, v_2)}{\text{Area}(v_0, v_1, v_2)}, \quad \beta = \frac{\text{Area}(p, v_2, v_0)}{\text{Area}(v_0, v_1, v_2)}, \quad \gamma = \frac{\text{Area}(p, v_0, v_1)}{\text{Area}(v_0, v_1, v_2)}, \quad \alpha + \beta + \gamma = 1.$$

1.1.4 Anti-aliasing

Aliasing occurs when continuous geometry is sampled too coarsely, producing stair-steps (jaggies) and moiré. Supersampling anti-aliasing (SSAA) addresses this by subdividing each pixel into an $N \times N$ grid of sub-pixel samples. Rasterization is evaluated at these sample positions; a resolve step then averages the N^2 samples back into one pixel using a box reconstruction filter:

$$C_{\text{pixel}} = \frac{1}{N^2} \sum_{i=1}^{N^2} C_{\text{sample},i}.$$

Accurate results depend on sampling at well-defined centers, carefully converting between pixel indices and sub-sample indices ($x_s = x \cdot N + s_x$), and clamping indices to valid ranges to avoid out-of-bounds writes. SSAA trades additional work and memory for smoother edges and reduced artifacts.

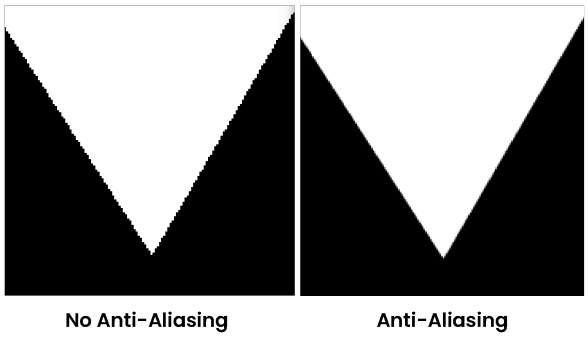


Figure 4: Comparison between Aliasing and using Anti-Aliasing

1.1.5 Color Blending

Transparency is modeled by storing an opacity value $\alpha \in [0, 1]$ alongside color C. With straight (non-premultiplied) alpha, the standard source-over composition for foreground (C_f, α_f) over background (C_b, α_b) is

$$C_{\text{out}} = \alpha_f \, C_f + (1 - \alpha_f) \, C_b, \qquad \alpha_{\text{out}} = \alpha_f + (1 - \alpha_f) \, \alpha_b.$$

This operation is applied per sample, and draw order determines which layers appear on top. Blending should be performed in a linear color space; any gamma correction is applied later when converting final resolved pixels for display.

1.2 Core Rasterization Concepts

Rasterization converts a scene described by continuous geometry and styles into a finite grid of colored pixels. The core ideas span (i) how coordinates move through spaces via linear/affine transforms, (ii) how we decide which discrete samples belong to a primitive (coverage), (iii) how we reduce aliasing by sampling more densely than one sample per pixel, (iv) how colors combine when layers overlap (alpha compositing), and (v) how images are sampled and filtered when used as textures. This subsection develops the first piece—coordinate systems and transformations—because every later stage consumes geometry that has been coherently mapped to a common image space.

1.2.1 Coordinate Systems and Transformations

A coordinate system specifies an origin and axes against which points are measured. In 2D graphics we routinely pass through several systems so that each operation happens where it is most natural. Shapes are authored in object (local) space, placed into a scene in world (canvas) space, normalized to a resolution-independent square called Normalized Device Coordinates (NDC), and finally mapped to integer screen-space pixels. For 2D orthographic rendering there is no perspective foreshortening, so clip space coincides with NDC.

All of these steps are expressed as affine transforms in homogeneous coordinates using 3×3 matrices. Writing a 2D point as $\mathbf{p} = (x, y, 1)^{\mathsf{T}}$,

$$\mathbf{p}_{\text{screen}} = \mathbf{S}\,\mathbf{V}\,\mathbf{M}\,\mathbf{p}_{\text{object}},$$

where $\mathbf{M}$ positions the object in world space, $\mathbf{V}$ normalizes a chosen world window into NDC, and $\mathbf{S}$ maps NDC to pixel coordinates. With column vectors the order matters: the rightmost transform applies first. Being consistent about this convention avoids mirrored or shifted results.

Translation, rotation, and anisotropic scaling are the building blocks:

$$\mathbf{T}(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad \mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad \mathbf{S}(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Any affine map is a product of these (and optionally a shear). Composition is associative but not commutative, so $\mathbf{R}\,\mathbf{T} \neq \mathbf{T}\,\mathbf{R}$ in general. The model matrix $\mathbf{M}$ places an object authored near $(0, 0)$ into the scene, e.g., $\mathbf{M} = \mathbf{T}\,\mathbf{R}\,\mathbf{S}$.

World coordinates are normalized to NDC by mapping a world window $[x - \text{span},\, x + \text{span}] \times [y - \text{span},\, y + \text{span}]$ to the fixed square $[-1, 1] \times [-1, 1]$. One convenient matrix is

$$\mathbf{V} = \begin{bmatrix} \frac{1}{\text{span}} & 0 & -\frac{x}{\text{span}} \\ 0 & \frac{1}{\text{span}} & -\frac{y}{\text{span}} \\ 0 & 0 & 1 \end{bmatrix},$$

which first translates the window center (x, y) to the origin and then scales by $1/\text{span}$ equally in x and y. If the image's aspect ratio is not square, either choose different spans per axis, or apply an additional scale to preserve circles as circles in NDC.

The viewport then converts $(u, v) \in [-1, 1]^2$ to integer pixel coordinates for an image of width W and height H:

$$x_{\text{pix}} = \frac{W}{2}(u + 1), \qquad y_{\text{pix}} = \frac{H}{2}(v + 1).$$

Many 2D systems take the pixel origin at the top-left with y increasing downward; if your mathematical $+y$ points up, include a flip $v \mapsto -v$ or bake it into $\mathbf{S}$. Pixel coordinates are then discretized when addressing the frame buffer; sub-pixel sample locations used for anti-aliasing (e.g., a regular $N \times N$ grid per pixel) live on a finer lattice derived from these same mappings.

Some operations (e.g., mapping a sample position back into an image's UV space) require the inverse of a composed transform. An affine matrix is invertible iff its upper-left 2×2 linear part has nonzero determinant. Guard against singular transforms (zero scale, collapsed shear), and when inverting, precompute and reuse the inverse rather than recomputing it at every sample. Keeping a consistent handedness and a single convention for where the origin sits (center vs. corner of a pixel) avoids half-pixel shifts that otherwise reveal themselves as "off-by-one" seams.

This coordinated chain—object $\to$ world $\to$ NDC $\to$ screen—produces a common image-space description of all primitives. The following subsections (coverage tests, anti-aliasing, blending, and texture filtering) assume geometry has been brought into this space and focus on how to turn those transformed shapes into high-quality pixels.

1.2.2 Line Rasterization

A line segment lives on a continuous plane, but the image is a discrete grid. Rasterization therefore decides which grid samples best approximate the continuous segment between (x_0, y_0) and (x_1, y_1). Classic approaches are the Digital Differential Analyzer (DDA) and Bresenham's algorithm; Bresenham is preferred because it uses only integer addition and comparisons, producing a coherent, gap-free path through the grid.

Let $\Delta x = x_1 - x_0$, $\Delta y = y_1 - y_0$, and define $\mathrm{sx} = \mathrm{sign}(\Delta x)$, $\mathrm{sy} = \mathrm{sign}(\Delta y)$. The *major axis* is the axis with the larger absolute delta. The idea is to advance one cell at a time along the major axis while maintaining an integer error term that determines when to also step once along the minor axis. A symmetric version that works for all eight octants is:

1. Initialize:
$$dx = |\Delta x|, \quad dy = |\Delta y|, \quad x = x_0, \quad y = y_0.$$
2. If $dx \geq dy$ (shallow line):
   - Set $e = 2dy - dx$.
   - Repeat for $dx + 1$ steps:
     (a) Plot (x, y).
     (b) If $e \geq 0$: update $y \leftarrow y + \mathrm{sy}$, $e \leftarrow e - 2dx$.
     (c) Update $x \leftarrow x + \mathrm{sx}$, $e \leftarrow e + 2dy$.
3. Else (steep line):
   - Set $e = 2dx - dy$.
   - Repeat for $dy + 1$ steps:
     (a) Plot (x, y).
     (b) If $e \geq 0$: update $x \leftarrow x + \mathrm{sx}$, $e \leftarrow e - 2dy$.
     (c) Update $y \leftarrow y + \mathrm{sy}$, $e \leftarrow e + 2dx$.

This algorithm visits exactly one grid point per major-axis step, creating a 4-connected chain that visually approximates the continuous segment. Practical details improve robustness and quality:

- Endpoints and inclusivity: to avoid overdrawing shared vertices when multiple segments meet, many systems adopt a half-open convention (e.g., include the first endpoint, exclude the last).
- Clipping: clip the segment to the screen rectangle before stepping (e.g., Cohen–Sutherland or Liang–Barsky) to avoid unnecessary iterations outside the frame.
- Sub-pixel sampling: in an oversampled renderer, treat the algorithm as operating on the sample grid $(W{\cdot}N) \times (H{\cdot}N)$; endpoints should be mapped to sample coordinates, typically centered at half-integers.
- Thickness: a thickness t can be approximated by splatting a small kernel around each visited sample, or by testing the shortest distance from a sample center to the infinite line and filling those within $t/2$. The distance of point p to the line through p_0, p_1 is $d(p) = \dfrac{|(p - p_0) \times (p_1 - p_0)|}{\|p_1 - p_0\|}$.
- Anti-aliased lines: for smoother appearance without supersampling, Xiaolin Wu's method weights neighboring pixels by their sub-pixel coverage; with supersampling already in place, Bresenham on the sample grid is sufficient.

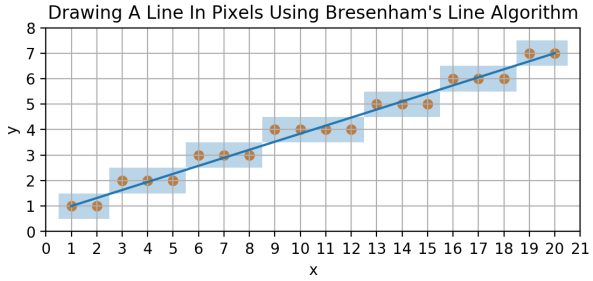


Figure 5: Bresenham's Line Algorithm

1.2.3 Triangle Rasterization

Triangles serve as a universal primitive because any polygon can be tessellated into triangles and a triangle is always convex and planar. Coverage testing based on edge functions gives a robust, incremental method that aligns well with the discrete grid.

For vertices $v_0 = (x_0, y_0)$, $v_1 = (x_1, y_1)$, $v_2 = (x_2, y_2)$, define the oriented edge function

$$E_{ab}(p) \;=\; (p-a)\times(b-a) \;=\; (p_x - a_x)(b_y - a_y) - (p_y - a_y)(b_x - a_x).$$

Its sign tells on which side of the directed edge $a \to b$ the point p lies. The signed twice-area of the triangle is $A = E_{01}(v_2)$; its sign encodes the winding (counter-clockwise if $A > 0$). A sample center $p = (x + 0.5, y + 0.5)$ is inside when the three edge functions have the same sign as A.

A fast implementation evaluates edge functions only on a conservative integer bounding box and updates them incrementally:

1. Compute the bounding box:

   $$x_{\min} = \lceil \min(x_0, x_1, x_2) \rceil, \quad x_{\max} = \lfloor \max(x_0, x_1, x_2) \rfloor,$$

   and similarly for $y_{\min}, y_{\max}$.

2. Precompute edge coefficients. For an edge from point a to b, define

   $$E_{ab}(x, y) = A_{ab}x + B_{ab}y + C_{ab},$$

   where

   $$A_{ab} = b_y - a_y, \quad B_{ab} = -(b_x - a_x), \quad C_{ab} = a_x b_y - a_y b_x.$$

   This allows constant-time incremental updates:

   $$E(x+1, y) = E(x, y) + A_{ab}, \quad E(x, y+1) = E(x, y) + B_{ab}.$$

3. Initialize the three edge functions E_{01}, E_{12}, E_{20} at the first sample center $(x_{\min}+0.5,\ y_{\min}+0.5)$. For each scanline y from $y_{\min}$ to $y_{\max}$:

   (a) Sweep x from $x_{\min}$ to $x_{\max}$.

   (b) Accept a sample if all three edge values share the sign of the triangle area.

   (c) Update the edge functions incrementally by adding the corresponding A_{ab} as x advances.

   (d) At the end of the row, update all edge values by adding the corresponding B_{ab} to move to the next scanline.

To avoid cracks between adjacent triangles that share an edge, adopt a consistent tie-break rule (the "top-left rule"): treat edges that are strictly top or strictly left as inclusive, and treat bottom/right edges as exclusive. With this half-open convention, a sample on a shared edge is owned by exactly one of the two triangles.

Barycentric coordinates fall out of the edge functions:

$$(\alpha, \beta, \gamma) = \frac{1}{A}\big(E_{12}(p),\, E_{20}(p),\, E_{01}(p)\big), \qquad \alpha + \beta + \gamma = 1.$$

They express the point p as $p = \alpha v_0 + \beta v_1 + \gamma v_2$ and provide weights for interpolating per-vertex attributes such as color or texture coordinates. In a 2D orthographic pipeline, this interpolation is affine; in a perspective 3D setting, one would use perspective-correct interpolation by dividing attributes by vertex w, interpolating, then un-dividing.

Degenerate triangles with $|A| \approx 0$ should be rejected early. Numerical robustness benefits from evaluating on sample centers, keeping all calculations in integer or fixed-point where possible, and clamping the scan region to the frame bounds so array accesses remain in range.

1.2.4 Supersampling and Anti-aliasing

Aliasing is the mismatch between a continuous signal (ideal geometric edges, fine textures) and a discrete sampling grid (pixels). When the signal contains frequencies above half the sampling rate (the Nyquist limit), those components fold back as visible artifacts: stair-steps along diagonals, moiré when patterns interfere, and flicker during motion. Supersampling Anti-aliasing (SSAA) reduces these artifacts by evaluating coverage and shading at multiple sub-pixel locations per pixel and then reconstructing one pixel value by low-pass filtering those samples.

In a regular $N \times N$ scheme, each pixel (x, y) is subdivided into sample coordinates

$$(x_s,\, y_s) = \big(x \cdot N + s_x,\; y \cdot N + s_y\big), \qquad s_x, s_y \in \{0, \ldots, N-1\},$$

with sub-pixel centers at

$$\Big(x + \frac{s_x + 0.5}{N},\; y + \frac{s_y + 0.5}{N}\Big).$$

Rasterization tests (edge functions for triangles, integer stepping for lines) are evaluated at these centers, and each covered sample stores a linear RGBA value in a floating-point sample buffer of size $(W \cdot N) \times (H \cdot N)$.

After all geometry has been processed, reconstruction produces one color per pixel. The simplest choice is the box filter, which averages the N^2 samples inside a pixel:

$$C_{\text{pixel}}(x, y) = \frac{1}{N^2} \sum_{s_x=0}^{N-1} \sum_{s_y=0}^{N-1} C_{\text{sample}}\big(x \cdot N + s_x,\; y \cdot N + s_y\big).$$

More elaborate kernels (e.g., tent, Mitchell–Netravali) can trade sharpness for ringing control, but the box filter is straightforward and effective in 2D rasterizers.

Blending belongs at the sample level and should operate in a linear color space. Each incoming sample color is composited over the existing sample using straight-alpha "source-over"; only after all primitives are composited are the samples averaged to obtain the pixel. Any gamma conversion is performed once on the resolved pixels. This ordering lets sub-pixel coverage (for example, a very thin line) contribute proportionally rather than degenerating to a binary on/off result.

SSAA improves edge smoothness and stabilizes thin features at the cost of increased memory ($\times N^2$) and additional shading work. Practical implementations clamp sample indices to valid ranges, compute conservative integer bounding boxes directly in sample space, and skip work for fully transparent primitives or regions outside the viewport. In many 2D scenarios, $N = 2$ or $N = 3$ provides a good balance between quality and cost.

1.2.5 Transform Hierarchies

Complex scenes are easier to manage when objects are organized into a hierarchy so that motion and deformation propagate naturally. Each node carries geometry in its own object space together with a local affine transform relative to its parent. The world transform of a node is the product of its ancestors' transforms with its own local transform,

$$\mathbf{M}_{\text{world}}(\text{node}) = \mathbf{M}_{\text{world}}(\text{parent})\,\mathbf{M}_{\text{local}}(\text{node}), \qquad \mathbf{M}_{\text{world}}(\text{root}) = \mathbf{I},$$

so translating the parent (a car body) automatically translates its children (wheels). The usual internal order for a local transform when using column vectors and right-multiplication is scale $\rightarrow$ rotate $\rightarrow$ translate, but the exact order should match the intended pivot and motion.

A depth-first traversal with a transform stack provides a clear and robust implementation:

```
push M_world
M_world <- M_world * M_local(node)
draw geometry of node using M_world
for each child:
    recurse on child
pop M_world
```

Pushing before descending and popping after returning prevents state from leaking between siblings. Group-level style attributes (opacity, fill, stroke) can be handled similarly by inheriting defaults downward and overriding where specified.

Many operations require an inverse mapping—for example, converting a screen/sample position back to an image's UV space. An affine matrix is invertible exactly when its 2×2 linear block has nonzero determinant; compute and cache inverses per node rather than per sample to reduce cost. Consistency about multiplication order, handedness, and the pixel-origin convention (center vs. corner) avoids subtle half-pixel shifts and edge cracks that can otherwise appear in hierarchical scenes.

1.2.6 Texture Sampling and Filtering

A texture is a discrete 2D array of texels $T[i, j]$ with width W_t and height H_t, while the renderer conceptually needs values of a continuous function $T(u, v)$ at arbitrary real coordinates. Sampling therefore has two parts: mapping a screen/sample position back to texture coordinates (inverse mapping), and reconstructing a color from nearby texels according to a chosen filter.

Inverse mapping starts from a sample location in screen space, transforms it into the texture's local space using the inverse of the object's affine transform, and then converts to normalized texture coordinates (u, v). A common convention is $u, v \in [0, 1]$ spanning the texture's left–right and top–bottom extents. Boundary handling must be defined: clamp-to-edge (u, v clamped to $[0, 1]$), repeat (u, v wrapped modulo 1), or mirror. The v-axis orientation is also a convention (images often store row 0 at the top); whichever you choose, remain consistent across mapping and sampling.

It is convenient to work in continuous texel space (s, t), where

$$s \;=\; u\,(W_t - 1), \qquad t \;=\; v\,(H_t - 1).$$

Texel centers are at integer lattice points (i, j). With this convention, $s \in [0, W_t - 1]$ and $t \in [0, H_t - 1]$; different APIs use slightly different offsets, but the formulas below are internally consistent.

Nearest-neighbor filtering chooses the texel whose center is closest to (s, t):

$$i = \text{round}(s), \quad j = \text{round}(t), \quad C = T[i, j].$$

This is fast and preserves hard edges but produces blocky magnification and sparkly minification.

Bilinear filtering linearly reconstructs from the four neighboring texels. Let $i = \lfloor s \rfloor$, $j = \lfloor t \rfloor$, and the fractional parts $a = s - i$, $b = t - j$ with $a, b \in [0, 1)$. The four samples are

$$C_{00} = T[i,\, j],\ C_{10} = T[i{+}1,\, j],\ C_{01} = T[i,\, j{+}1],\ C_{11} = T[i{+}1,\, j{+}1],$$

after applying the chosen boundary mode when indices exceed $[0, W_t{-}1] \times [0, H_t{-}1]$. Interpolate horizontally, then vertically:

$$C_0 = (1-a)\, C_{00} + a\, C_{10}, \qquad C_1 = (1-a)\, C_{01} + a\, C_{11}, \qquad C = (1-b)\, C_0 + b\, C_1.$$

The operation is done per color channel in a linear color space. If texels carry straight alpha, it is often more accurate to convert to premultiplied form for filtering (multiply RGB by α), filter, then un-premultiply (divide by α if $\alpha > 0$) to avoid edge darkening around partially transparent content.

Mipmapping addresses aliasing during *minification* (when many texels fall under one screen pixel). A mip chain is a sequence of progressively smaller images: level 0 is the original ($W_t \times H_t$), level 1 is roughly half in each dimension, and so on until 1×1. A simple generator uses a $2{\times}2$ box filter:

$$T^{(\ell+1)}[i, j] \;=\; \tfrac{1}{4}\left(T^{(\ell)}[2i, 2j] + T^{(\ell)}[2i+1, 2j] + T^{(\ell)}[2i, 2j+1] + T^{(\ell)}[2i+1, 2j+1]\right),$$

handling odd sizes by clamping the missing taps. Generate and store colors in linear space; with alpha, averaging premultiplied colors and alphas gives better results than averaging straight RGBA.

Level-of-detail (LOD) selects which mip levels to use at a given screen sample. The goal is to match the *footprint* of one screen pixel in texture space. A common estimate uses derivatives of (u, v) with respect to screen coordinates:

$$\rho^2 \;=\; \max\!\big(\, \|\partial_x \mathbf{u}\|^2,\ \|\partial_y \mathbf{u}\|^2 \,\big) \cdot \max(W_t, H_t)^2, \quad \text{with } \ \mathbf{u} = (u, v).$$

The desired mip level is then $\ell = \frac{1}{2} \log_2(\rho^2) = \log_2 \rho$. Trilinear filtering reduces popping between discrete levels by linearly interpolating bilinear results from the two nearest integer levels:

$$\ell_0 = \lfloor \ell \rfloor, \quad \ell_1 = \lceil \ell \rceil, \quad \tau = \ell - \ell_0, \qquad C = (1-\tau)\, C^{(\ell_0)}_{\text{bilinear}} + \tau\, C^{(\ell_1)}_{\text{bilinear}}.$$

This yields smooth transitions as geometry moves or scales. For extreme anisotropy (highly stretched footprints), isotropic filters like bilinear/trilinear are suboptimal; anisotropic filters align the kernel with the footprint's ellipse, but that lies beyond the present 2D scope.

Putting it together for each screen/sample position: compute (u, v) by inverse mapping; apply boundary mode; if magnifying, nearest or bilinear may suffice; if minifying, compute ℓ and perform trilinear sampling from the mip chain. Perform all computations in linear color, and ensure consistent axis conventions so that the sampled image is not flipped or shifted.

1.2.7 Alpha Compositing

Alpha models opacity as a real value $\alpha \in [0, 1]$ stored alongside color C. With $\alpha = 1$ the sample is fully opaque; with $\alpha = 0$ it contributes nothing. The standard "source-over" rule composites a new (foreground) sample (C_f, α_f) over the existing (background) sample (C_b, α_b) as

$$C_{\text{out}} = \alpha_f\, C_f + (1 - \alpha_f)\, C_b, \qquad \alpha_{\text{out}} = \alpha_f + (1 - \alpha_f)\, \alpha_b,$$

applied per channel in a linear (not gamma-encoded) color space. Draw order matters because each new fragment is placed over what is already in the buffer.

Two equivalent parameterizations are widely used. In *straight alpha* the stored color is C and the equations above apply directly. In *premultiplied alpha* the stored color is $\tilde{C} = \alpha C$; compositing becomes

$$\tilde{C}_{\text{out}} = \tilde{C}_f + (1 - \alpha_f)\,\tilde{C}_b, \qquad \alpha_{\text{out}} = \alpha_f + (1 - \alpha_f)\,\alpha_b,$$

and the straight color can be recovered (when $\alpha_{\text{out}} > 0$) by $C_{\text{out}} = \tilde{C}_{\text{out}}/\alpha_{\text{out}}$. Premultiplication is numerically robust at translucent edges and makes repeated "over" operations associative, which is helpful when combining many layers or filtering images that already contain transparency.

Coverage from rasterization and opacity from material both attenuate contribution. If a sub-pixel coverage estimate $c \in [0, 1]$ is available (e.g., via supersampling), it is best folded into alpha as $\alpha' = c\,\alpha$ so that tiny features contribute proportionally. Compositing should occur per sample; the resolve step of supersampling then averages already-composited samples into the final pixel. Any gamma conversion is deferred until the very end, after all blending and averaging in linear space.

1.2.8 Viewport Transformations

The viewport maps continuous world coordinates into the discrete image grid and exposes intuitive controls for panning, zooming, and aspect handling. Let the world window be the rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. A linear normalization to Normalized Device Coordinates (NDC) $(u, v) \in [-1, 1]^2$ is

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{2}{x_{\max} - x_{\min}} & 0 & -\frac{x_{\max} + x_{\min}}{x_{\max} - x_{\min}} \\ 0 & \frac{2}{y_{\max} - y_{\min}} & -\frac{y_{\max} + y_{\min}}{y_{\max} - y_{\min}} \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{V}_{\text{win}\to\text{NDC}}} \begin{bmatrix} x_{\text{world}} \\ y_{\text{world}} \\ 1 \end{bmatrix}.$$

Using a center–span parameterization with center (x_0, y_0) and spans $s_x, s_y > 0$ one obtains the same map by substituting $x_{\min} = x_0 - s_x$, $x_{\max} = x_0 + s_x$ and likewise for y. Panning updates (x_0, y_0); zooming scales (s_x, s_y). Keeping $s_x : s_y$ proportional to the image aspect ratio preserves shapes (a circle stays circular in NDC).

The NDC-to-pixel map for an image of width W and height H is

$$x_{\text{pix}} = \frac{W}{2}(u + 1), \qquad y_{\text{pix}} = \frac{H}{2}(v + 1),$$

which places $(-1, -1)$ at $(0, 0)$ and $(+1, +1)$ at (W, H). This choice makes the pixel origin the top-left and increases y downward; if a mathematical convention with $+y$ upward is desired, include a sign flip in the v row of the matrix.

It is often convenient to combine normalization and viewport into a single world-to-screen matrix. With the center–span form and equal spans $s_x = s_y = \text{span}$, the product reduces to

$$\begin{bmatrix} x_{\text{pix}} \\ y_{\text{pix}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{W}{2\,\text{span}} & 0 & \frac{W}{2} - \frac{x_0\,W}{2\,\text{span}} \\ 0 & \frac{H}{2\,\text{span}} & \frac{H}{2} - \frac{y_0\,H}{2\,\text{span}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{world}} \\ y_{\text{world}} \\ 1 \end{bmatrix}.$$

Clipping in world or NDC space limits work to the visible region. When supersampling, pixel coordinates convert to sample indices by $(x_s, y_s) = (\lfloor x_{\text{pix}} \rfloor \cdot N + s_x,\ \lfloor y_{\text{pix}} \rfloor \cdot N + s_y)$ for $s_x, s_y \in \{0, \ldots, N - 1\}$. A consistent convention for pixel centers (e.g., at half-integers) and careful rounding prevent half-pixel shifts and ensure that neighboring tiles meet without gaps.

1.3 Integration with SVG

Scalable Vector Graphics (SVG) provides an ideal testbed for rasterization algorithms because it contains all the fundamental primitives:

- **Basic shapes**: lines, rectangles, circles, polygons
- **Complex paths**: Bézier curves, arcs (simplified to line segments)
- **Styling**: stroke, fill, opacity, colors
- **Transformations**: translate, rotate, scale applied hierarchically
- **Images**: raster images embedded within vector graphics

The SVG coordinate system uses floating-point coordinates with (0,0) at the top-left, positive X rightward, and positive Y downward, which maps naturally to screen coordinates.

This foundational understanding of rasterization concepts will enable you to implement a complete 2D rendering pipeline, gaining deep insights into how graphics hardware and software systems convert mathematical descriptions of shapes into the pixels you see on screen.

2 Code Flow and Implementation Tasks

2.1 System Architecture Overview

The SVG rasterizer consists of several interconnected components that work together to convert vector graphics into pixel images:

1. **SVG Parser** (`svg_parser.py`): Reads SVG files and converts them into internal data structures
2. **Rendering Engine** (`software_renderer.py`): Main rasterization system that converts primitives to pixels
3. **Texture System** (`texture.py`): Handles image sampling and filtering
4. **Mathematical Utilities** (`math_utils.py`): Vector, matrix, and color operations
5. **GUI Application** (`draw_svg.py`): User interface for viewing and interacting with SVG files

2.2 Rendering Pipeline Flow

The complete rendering process follows this sequence:

`SVG File` → `Parse Elements` → `Apply Transforms` → `Rasterize Primitives` → `Anti-alias` → `Display`

2.2.1 Step 1: SVG Loading and Parsing

The system loads an SVG file and parses it into a hierarchy of elements (lines, triangles, rectangles, images, groups).

Implementation Status: **[COMPLETE]** *No student work required*

2.2.2 Step 2: Transform Hierarchy Processing

The renderer traverses the SVG element tree, applying nested transformations and drawing each element.

Student Task: [TODO] **Implement** `draw_element()` - Apply hierarchical transformations and render elements based on their type.

2.2.3 Step 3: Viewport and Coordinate Transformation

Convert from SVG world coordinates to screen pixel coordinates using viewport transformations.

Student Task: [TODO] **Implement** `get_matrix()` - Handle pan/zoom operations and coordinate space conversions.

2.2.4 Step 4: Primitive Rasterization

Convert geometric primitives (lines, triangles) into discrete pixels.

Student Tasks:

- [TODO] **Implement** `rasterize_line()` - Bresenham's algorithm for line drawing
- [TODO] **Implement** `rasterize_triangle()` - Edge function-based triangle filling

2.2.5 Step 5: Image and Texture Processing

Rasterize image elements by mapping screen pixels to texture coordinates and sampling colors.

Student Tasks:

- [TODO] **Implement** `rasterize_image()` - UV mapping and inverse transformation
- [TODO] **Implement texture sampling functions**:
  - `sample_nearest()` - Nearest neighbor filtering
  - `sample_bilinear()` - Bilinear interpolation
  - `sample_trilinear()` - Trilinear filtering with mipmaps
  - `generate_mips()` - Mipmap generation using box filtering

2.2.6 Step 6: Anti-aliasing Pipeline

Use supersampling to reduce aliasing artifacts by rendering at higher resolution and downsampling.

Student Tasks:

- [TODO] **Implement** `fill_sample()` - Write individual samples to high-resolution buffer
- [TODO] **Implement** `fill_pixel()` - Fill all samples belonging to a pixel
- [TODO] **Implement** `resolve()` - Downsample from sample buffer to pixel buffer using box filter

2.2.7 Step 7: Alpha Compositing

Blend transparent and semi-transparent colors using alpha channel information.

Student Task: [TODO] **Implement** `alpha_blend()` - Standard alpha blending formula for transparency effects.

2.2.8 Step 8: Display

Convert the final pixel buffer to a format suitable for display in the GUI.
Implementation Status: [COMPLETE] *No student work required*

2.3 Detailed Code Execution Flow

2.3.1 Main Rendering Loop

When a user opens an SVG file, the system follows this execution path:

1. **File Loading**: `SVGParser.load(filename)` reads and parses the SVG
2. **Renderer Setup**: Create sample and pixel buffers based on screen size and sample rate
3. **Clear Buffers**: Initialize buffers to background color
4. **Element Traversal**: Call `draw_element()` on root SVG element
5. **Recursive Rendering**: Each element processes its children recursively
6. **Final Resolve**: Call `resolve()` to convert sample buffer to displayable pixels
7. **GUI Update**: Display the rendered image in the interface

2.3.2 Transform Stack Management

The transform hierarchy works as follows:

1. Push current transformation matrix onto stack
2. Multiply current matrix with element's local transform
3. Render element with accumulated transformation
4. For group elements, recursively process children
5. Pop transformation matrix to restore previous state

2.3.3 Coordinate Space Conversions

The rendering pipeline uses multiple coordinate systems:

1. **SVG Coordinates**: Original coordinates from SVG file
2. **World Coordinates**: After applying modeling transformations
3. **Screen Coordinates**: After applying viewport transformation (integer pixels)
4. **Sample Coordinates**: High-resolution coordinates for anti-aliasing (screen $\times$ sample_rate)

2.4 Buffer Management System

The anti-aliasing system uses a two-buffer approach:

2.4.1 Sample Buffer

- **Size**: `width × sample_rate × height × sample_rate × 4`
- **Format**: Float RGBA values in range [0,1]
- **Purpose**: High-resolution rendering target
- **Access**: Via `fill_sample(x, y, color)`

2.4.2 Pixel Buffer

- **Size**: `width × height × 4`
- **Format**: Uint8 RGBA values in range [0,255]
- **Purpose**: Final display output
- **Access**: Written by `resolve()` function

2.4.3 Resolution Process

The `resolve()` function performs box filter averaging:

1. For each pixel in the output image
2. Collect all samples within that pixel's area
3. Average the RGBA values across all samples
4. Convert from float to uint8 format
5. Write to pixel buffer for display

2.5 Task Breakdown

The assignment is divided into 6 specific tasks that must be completed sequentially. Each task focuses on a particular aspect of the rasterization pipeline.

2.5.1 Task 0: Line Rasterization

Implement basic line drawing using Bresenham's algorithm.

Files to modify:

- `software_renderer.py`

Functions to implement:

- `rasterize_line(start: Vector2D, end: Vector2D, color: Color, width: float)`
  - Transform start and end points to screen space
  - Convert to sample coordinates
  - Implement Bresenham's algorithm with integer arithmetic
  - Handle line thickness by drawing multiple pixels
  - Use `fill_sample()` to write pixels

2.5.2 Task 1: Triangle Rasterization

Implement triangle filling using edge functions and barycentric coordinates.

Files to modify:

- `software_renderer.py`

Functions to implement:

- `rasterize_triangle(vertices: List[Vector2D], color: Color)`
  - Transform triangle vertices to screen/sample space
  - Compute bounding box optimization
  - Calculate triangle area for orientation detection
  - Use edge functions to test point-in-triangle
  - Handle both clockwise and counter-clockwise winding
  - Fill pixels using `fill_sample()`

2.5.3 Task 2: Supersampling & Anti-aliasing

Implement the anti-aliasing pipeline using supersampling and box filter reconstruction.

Files to modify:

- `software_renderer.py`

Functions to implement:

- `fill_sample(x: int, y: int, color: Color)`
  - Bounds checking for sample buffer coordinates
  - Alpha blending with existing sample color
  - Write to high-resolution sample buffer
- `fill_pixel(x: int, y: int, color: Color)`
  - Fill all NxN samples belonging to a pixel
  - Calculate correct sample coordinates
  - Call `fill_sample()` for each sample
- `resolve()`
  - Box filter averaging from sample buffer to pixel buffer
  - Handle RGBA channels correctly
  - Convert from float to uint8 format
  - Write final colors to pixel buffer

2.5.4 Task 3: Transform Hierarchy

Implement hierarchical transformations and viewport management.

Files to modify:

- `software_renderer.py`

Functions to implement:

- `draw_element(element: SVGElement)`
  - Push current transform onto stack
  - Apply element's local transformation
  - Render element based on type (point, line, triangle, polygon, rect, image, group)
  - Handle recursive rendering for groups
  - Pop transform to restore previous state
- `get_matrix() -> Matrix3x3`
  - Create transformation matrix from viewport to normalized space
  - Handle scale and translation calculations
  - Map viewport region to [-1,1] × [-1,1]

2.5.5 Task 4: Image Rasterization)

Implement complete texture sampling pipeline with multiple filtering modes.

Files to modify:

- `software_renderer.py`
- `texture.py`

Functions to implement:

In `software_renderer.py`:

- `rasterize_image(img: SVGImage)` (
  - Define image corners in local space
  - Transform corners to screen space
  - Compute bounding box
  - Create inverse transform for UV calculation
  - Map screen pixels back to texture coordinates
  - Sample texture using trilinear filtering

In `texture.py`:

- `sample_nearest(u: float, v: float, level: int) -> Color`
  - Clamp UV coordinates to [0,1] range
  - Convert UV to pixel coordinates
  - Round to nearest integer pixel
  - Handle texture bounds clamping
- `sample_bilinear(u: float, v: float, level: int) -> Color`
  - Convert UV to pixel coordinates with 0.5 offset
  - Get integer and fractional parts
  - Sample four surrounding pixels
  - Perform bilinear interpolation (horizontal then vertical)

- `sample_trilinear(u: float, v: float, mip_level: float) -> Color`
  - Determine appropriate mipmap level
  - Sample from two adjacent mipmap levels using bilinear
  - Linearly interpolate between results
  - Handle edge cases (single mip level, fractional near 0)
- `generate_mips()`
  - Start with original texture as level 0
  - For each level, average 2×2 blocks from previous level
  - Continue until reaching 1×1 resolution
  - Store all levels in mipmaps array

2.5.6 Task 5: Alpha Compositing

Implement standard alpha blending for transparency effects.

Files to modify:

- `math_utils.py`

Functions to implement:

- `alpha_blend(self, background: Color) -> Color`
  - Extract alpha from foreground color
  - Calculate inverse alpha (1 - alpha)
  - Blend RGB channels: `result = alpha * fg + (1-alpha) * bg`
  - Blend alpha channel: `result_alpha = alpha + (1-alpha) * bg_alpha`
  - Return new Color object with blended values

2.6 Task Dependencies and Recommended Order

The tasks should be completed in this order due to dependencies:

1. **Task 5** (Alpha Compositing) - Required by `fill_sample()`
2. **Task 2** (Supersampling) - Provides `fill_sample()` used by all rasterization
3. **Task 0** (Line Rasterization) - Simplest primitive to test anti-aliasing
4. **Task 1** (Triangle Rasterization) - Core primitive for complex shapes
5. **Task 3** (Transform Hierarchy) - System integration
6. **Task 4** (Image Rasterization) - Most complex, requires all previous components

3 Specifications

3.1 Application Usage

Students are expected to implement a 2D SVG software rasterizer with interactive GUI capabilities. The application should provide the following functionality:

Command Line Interface:

- Run with single SVG file: `python draw_svg.py filename.svg`
- Run with directory: `python draw_svg.py svg/directory_name`
- Run without arguments to use GUI file dialogs

Interactive Controls:

- File loading via GUI dialogs (single file or entire directory)
- Navigation between multiple SVG files using Previous/Next buttons
- Sample rate adjustment (1x, 2x, 3x anti-aliasing)
- Viewport manipulation (pan with mouse drag, zoom with mouse wheel)

3.2 Testing and Validation

A set of SVG test files is provided in the `svg` folder. You can open these files with any standard SVG viewer and compare the output against your own rasterizer to verify correctness.

4 Submission Format

Submit your source code as a single ZIP file on SCeLE and be prepared for a **live demo** where your team explains and demonstrates your implementation. This assignment is done in **pairs**.

ZIP Archive

- **Filename:** `<NPM1>-<NPM2>-A1-rasterizer.zip`

Live Demo (pair)

- **Schedule:** TBA by the TAs on SCeLE.

Happy Rendering!