Assignment 1: Introduction to 2D Rasterization & SVG

CMSC 23700 Winter 2026: University of Chicago
Due: Fri January 16th at 11:59PM

Textbook Reference

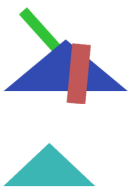
Fundamentals of Computer Graphics 5th ed.: Ch. 9 (The Graphics Pipeline)

Tip

In Computer Graphics, it is often the case that there are *many* different ways which can be used to solve the same problem. You will find that in this assignment as well. Our goal is to give you the high-level algorithmic question, and let you use different resources (such as the above textbook!) to figure out exactly *how* to try to solve it!

Introduction

In this assignment we will build a simple 2D rasterizer for triangles, line segments, and circles. This is to get yourself acquainted with the data structures we use to represent shapes and building image arrays, before tackling the trickier issues we'll encounter in 3D. You will implement the function `rasterize(svg,h,w,background)` which will take as input a svg file, image height (H) and width (W) and background color and output a *raster image* as a numpy array of dimension $(H, W, 3)$. The svg file will contain shapes that are either triangles, line segments, or circles. Below is an example of what your code will produce:



The inputs are in the folder `tests` and the corresponding expected outputs for each of these sections are in `output`. The main file is `raster.py`. In the main block of `raster.py`, you will see an example call of the `rasterize` function that could be used for the first SVG file (and first part) of this assignment.

Submission Instructions

All of your submission code should be contained in `raster.py`. Do not submit any other python files. Do not zip/tar your submission. Do not put the python file within a folder. Do not import external packages. In addition, please upload your (custom) design image (one PNG) that was rasterized with your code. Your submission should contain **only** three files: **`raster.py`**, **`custom.png`**, and **`documentation.pdf`**. You may write any helper functions you wish, so long as they remain within `raster.py`. Try to remove print statements before submitting (as this may cause the autograder to time out). If you are planning to use your late bank but are checking your assignment output using the autograder - put the following text in your **`documentation.pdf`** during submission testing: *Please do not grade this submission yet, I am using my late bank.*

Documentation

You should thoroughly and thoughtfully document your work in this project, **both in code** and in a **`documentation.pdf`** file you submit alongside the code and image.

- In **`documentation.pdf`**, document your high-level progress through the assignment: your initial planned approaches, any issues you ran into during implementation, and how you solved them.
- In **`documentation.pdf`**, draw and include an annotated diagram or sketch you use to derive your solution. Graphics algorithms are especially suited for the use of diagrams to aid understanding and debugging. **Please create and include at least one such diagram in `documentation.pdf` as you go through the assignment. Drawing by hand is fine. Diagrams from conversations with a TA/instructor are fine, but must be your own work.** Here are some examples of things you may want to visualize/sketch out for this assignment:
  - a pixel and its corresponding point in SVG (viewbox) space, and the relations between pixel indices and viewbox coordinates
  - the method you use to check if a point is inside a triangle
  - the method you use to rasterize a line segment using other primitives

  There is no hard standard for how detailed such a diagram must be, but they should be sufficiently illuminating in order to have helped you implement the subproblem in question.
- In **`documentation.pdf`**, cite all resources, online articles, Q&A threads, etc. you use in your code. In line with the Generative AI policy on the course website, you must also cite any use of Generative AI involved:
  - You must include the questions you used and GenAI answers (e.g. a link to the saved conversation if applicable; a text paste of the conversation otherwise)
  - You may **not** use GenAI to describe your code or write any part of `documentation.pdf`
  - **You must only use GenAI for the permissible uses described in the Generative AI policy on the course website.**

- In **code comments**, thoroughly document nontrivial lines and leaps of logic. We encourage documenting the shapes of numpy arrays, even if they may be obvious in the moment. Comment thoughtfully so that you may recall what you did upon re-reading your code in six months; consider the thought process of a hypothetical reader who is in the course but has not done this assignment.

Read SVG (`utils.py`)

Your program will need to read an svg file. You can use the already implemented function `read_svg` in `utils.py`. The function takes as input a string of the file path of the SVG, and will return a list of shapes to be rasterized.

Visualize Results (`utils.py`)

The starter code in the `rasterize` function saves your result image array using the function `save_img`, defined in `utils.py`. It accepts as input a numpy array of dimension $H \times W \times 3$ with values between 0 and 1, and saves both a `.png` image and an `.npy.gz` file containing the raw numpy array (with gzip compression).

In grading, we check the correctness of the **array returned by your `rasterize` function**, not the image or array saved with `save_img`. However, these images are helpful for visualizing what your implementation is currently doing; you might also wish to save and visualize intermediate images using `save_img` for debugging purposes.

In `utils.py` we also provide basic image diffing abilities: you can run the `utils.py` script directly on two images as arguments (either `.png` or `.npy.gz` is fine) which will open your image viewer with three images concatenated: (image 1, image 2, difference in red). For reliable results, only diff `.png` against `.png`, or `.npy.gz` against `.npy.gz`.

```
python utils.py test_image.png reference_image.png
```

Debugging tip

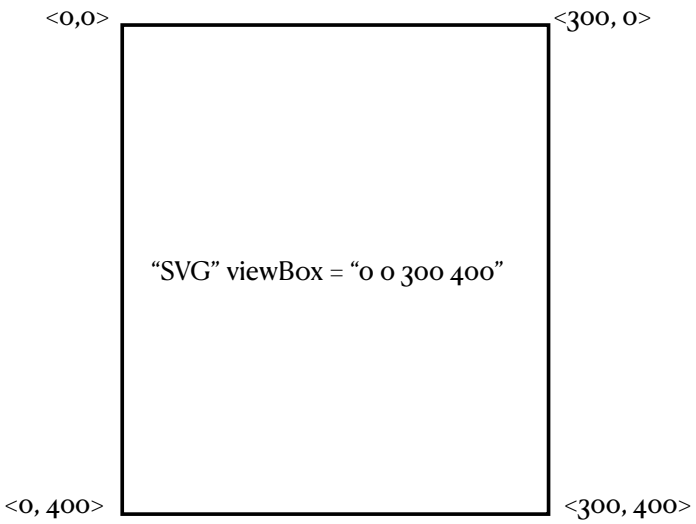
If your solution is only different from the expected images at **edge pixels** of the rendered primitives, try adding a `1e-15` tolerance to both sides of your $\leq$ or $\geq$ comparisons involved in bounding box or inside-outside calculations. For instance, for a test like `a <= x <= b`, try `(a - 1e-15) <= x <= (b + 1e-15)`. This can compensate for any floating point differences that have accumulated in your solution that aren't actually mathematically significant, only appearing because of floating point precision.

SVG Primitives (`shapes.py`)

The shape class defined in `shapes.py` contains information about the SVG shapes (triangles, lines, circles) and viewbox. The first shape in the list will always be of type `SVG` (and with the attribute `shape.type=="svg"`) and will contain the viewbox dimensions. The figure below illustrates the viewbox coordinate frame of the SVG, which is seperate from the resolution you will render your image at.

Remember that the only Python file you edit and submit is `raster.py`. If you need something to the effect of *methods* on `Shape` objects, rather than writing them as methods

in `shapes.py`, write them as functions inside `raster.py` that branch on the type of the `Shape` object.



*not the same as the rasterized pixel grid!

The viewbox information, along with the image width and height will determine transformations from viewbox to image coordinates and vice versa. **You will need to be able to render an image at any resolution, and in many cases the aspect ratio of the image will be different than the viewbox aspect ratio.** The viewbox will always have $(0, 0)$ as the origin. You will frequently need to change back and forth from image to viewbox coordinates, so it might be a good idea to write separate functions for these operations.

Note that in the numpy array of shape $(H, W, 3)$, which represents the image, y is the first axis, and x is the second axis. You may want to handle axis swaps by a numpy function or a transformation matrix. Each shape should be processed in the order that is given by `read_svg`. Following this order will be necessary to deal with ambiguity in cases of overlapping shapes. To make sure you deal with overlap correctly, your code should follow the outline given below. (*This is pseudocode, not exact Python.*)

```
def rasterize(svg):
    image[:,:] = background
    shapes = read_svg(svg)
    svg = shape[0]
    for shape in shapes[1:]:
        for i,j in bounding box:
            x,y = viewbox_coordinates_for_pixel(svg,i,j)
            a = get_coverage_for_point(shape,x,y)
            image[i,j] = (1—a)*image[i,j]+shape.color*a
    return image
```

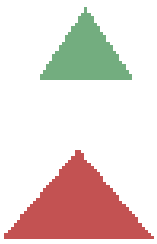
Where i and j are the 2D image coordinates inside the bounding box. Most of the work in this assignment will be in determining how much, if any, of a given pixel lies inside a particular shape. How your code is organized is entirely up to you. The only packages that your code should use is what is already imported in the provided starter code. With the exception of importing extra types from the `typing` module for type annotations (which are encouraged), **do not import any additional packages**.

Required packages

python >= 3.8, numpy >= 1.21, Pillow >= 10

1 Rasterize a triangle

In this first test, your code will rasterize a triangle using `test1.svg`, and if implemented correctly the output will look as follows:



In this part of the assignment, you should be able to output anything that starts with `test1_` and ends with `_noaa.png`.

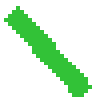
Recall that in rasterization, we need to fix a given triangle primitive, and then loop over each of the pixels to determine whether they are inside or outside the triangle. Specifically, we will check whether the *center point* of each pixel is inside or outside the triangle. You must find a way to determine whether a point p lies inside the triangle. In this assignment, we consider an edge point to lie inside a triangle.

Hint: Perform the inside-triangle checks in SVG viewbox coordinates (not image coordinates), i.e. given a pixel, find its corresponding viewbox point and use this point (representing this pixel) for the check.

`documentation.pdf` diagram: It may be helpful to draw a diagram of this corresponding point in relation to the pixel, and/or the method you use for the inside-triangle test. Include it in **`documentation.pdf`** (see the documentation requirements.)

2 Rasterize a line

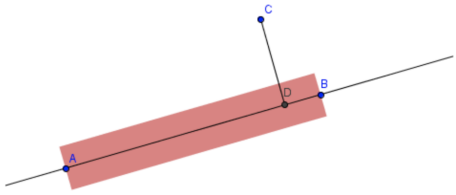
In the second part, your code will rasterize a line using `test2.svg`, and if implemented correctly the output will look as follows:



In this part of the assignment, you should be able to output anything that starts with `test2_` and ends with `_noaa.png`.

A line is determined by two endpoints and a width. The endpoints and width will be given to you in the viewbox coordinate system.

In your code, you will need to be able to determine whether a point lies inside the line segement. As with triangles, we consider an edge point to lie inside a line. See the figure:



Hint: For our purposes, a line segment is just a rectangle. You should rasterize this by rasterizing a composition of other primitives. (You do not need to implement a line-drawing algorithm such as Bresenham's.)

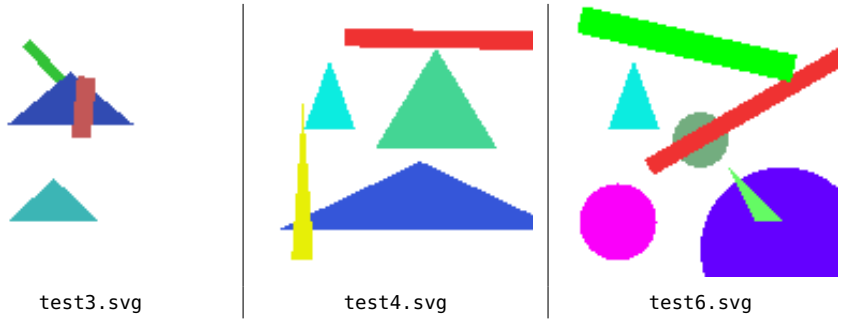
`documentation.pdf` diagram: It may be helpful to draw a diagram of how you decompose and rasterize a line in this manner. Include it in **`documentation.pdf`** (see the documentation requirements.)

3 Rasterize a circle

In the third part of this assignment, your code will rasterize circles using test5.svg. Again, we consider an edge point to lie inside a circle. If correctly implemented, your output will look as follows:

Rasterizing triangles, lines, and circles

After implementing the above, you should be able to check `test3.svg`, `test4.svg`, and `test6.svg`, which at this point should look like:

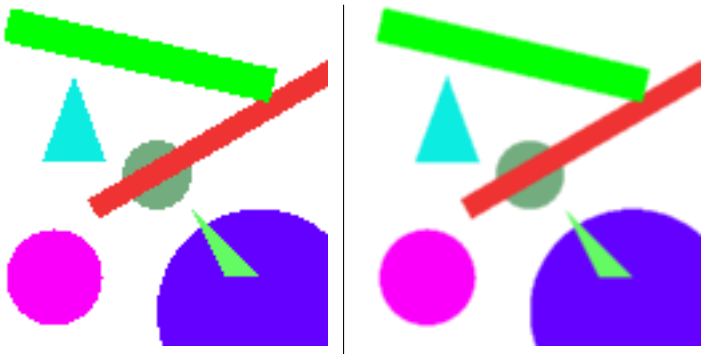


`test3.svg` `test4.svg` `test6.svg`

By this point, your code should be able to reproduce everything that ends with `_noaa.png`.

4 Antialiasing / Supersampling

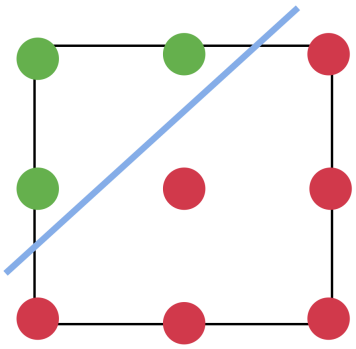
In this part of the assignment, you will implement antialiasing, and sample multiple points per pixel. This will improve the image quality, and look like the following:



without anti-alias | with anti-alias

After implementing this part of the assignment, you should be able to reproduce all the files that end with `_aa.png`.

Remember from lecture, we are assigning a *coverage* (or inside) value that will be not only 0 or 1, but also values in between. To do this, sample a 3×3 grid in each pixel and compute the fraction of grid samples that lie inside the shape to get the *coverage* value for that pixel. Multiply the coverage value by the shape's color vector and this will determine the color at that pixel. Blend this color value with whatever lies in the image at that pixel (see the code sample above). Sample the points in your grid using the same spacing as shown below.



The square represents a single pixel in image coordinates, and the points represent the locations relative to the pixel that are to be sampled. Note that in this figure, the pixel's integer coordinates, when read as a point in continuous image space, would be the **upper left corner** of the square. Each pixel needs to be **sampled deterministically at evenly-spaced points** as shown above. **You must sample the grid in this way**, otherwise your solution will not pass our checks.

5 Bounding box acceleration

To speed up your code, you should only compute color for points that lie inside a bounding box of a shape. This step should not affect your output at all, only the runtime. Note that when computing the bounding box of the line, you will have to take into account the line's width. Computing the tightest possible bounding box is not necessary; it is fine to give an overestimate. (**Hint:** if you see errors around the tips of shapes, try extending your bounding box a little; your bounding box may have been an *underestimate*.)

6 Testing your code

We provide several test svg files for you to use in the `tests` directory. The `output` directory contains rasterized images of these svgs at various resolutions, which you can use to check your code. The main loop of `raster.py` will run your code on the test inputs, but feel free to modify it to do whatever you want.

7 Create your own vector graphic!

Create and then render your own custom SVG file. Get creative! Note that the current SVG parser may not support all types of SVG files: The only polygons supported are triangles; transform matrices are not applied; strokes are only supported in the context of line segments, not as borders of polygons.

Hint: draw rectangles using triangles or "lines" with large width. Look at the SVG source of the test SVGs, copy over the rough shapes you need (to make sure your SVG uses only supported elements), and then use an editor (such as Inkscape) to move them around more conveniently than manually editing coordinates.

8 Extra credit: Vectorization

The code skeleton above computes one pixel at a time, in Python loops. This is fine for full credit (and is pedagogically useful, as you can tell exactly what happens to each pixel in isolation.)

But calling Python functions on a per pixel basis in Python loops is slow, and the cost adds up when you're computing tens of thousands of pixels! See if you can **vectorize** your implementation, **including bounding box acceleration** (i.e. for each shape, its bounding box defines the region of pixels to process in a batched way). Vectorizing means using numpy's functions on arrays to process elements in batch. A fully-vectorized implementation of this project should use **no Python loops** whose iteration count depends on the raster image size in some way, e.g. loops over pixels/rows/columns/subarrays of any array (i.e. no `for`, `while`, `map`, `reduce`, iterables, etc functionality in Python, only numpy for batch processing). The only explicitly written loop should be the loop over the `shapes` list.

(Note that just throwing your Python loop body into the `np.vectorize` function does **not** count as truly vectorizing. As its documentation says, this function is "for convenience, not for performance" and is basically a for loop using the Python function you give as the body.)