CSE 528 COMPUTER GRAPHICS (FALL 2024)
HOMEWORK ONE

For this homework, there are two separate parts: (1) Non-programming part; and (2) Programming part. Please refer to the TA Help Page for general requirements, and take note that all deadlines are **strictly enforced**. To prevent late submissions caused by network issues, **avoid** submitting at the last minute! Please also note that your credits for this homework will be scaled (i.e., from $100 + 300$ to $50 + 100$) on Brightspace to match the overall grading schemes.

Non-programming Part

1. (**30 points**) (a) Please give a brief definition on Computer Graphics; (b) Why is Computer Graphics also called Image Synthesis; and (c) Please provide at least **ten** reasons why Computer Graphics is necessary and critical to information technologies and computer science and engineering!

2. (**30 points**) (a) Based on the reading assignment from the instructor, please do a sufficient amount of readings (based on the textbook and/or the Internet resources) and document in details at least **ten** areas (fields) with concrete examples in such areas why Computer Graphics technologies are of fundamental significance to the success of these fields/areas; and (b) Please do further readings on key components of hardware and software in Computer Graphics and explain in details the key components of hardware and software critically relevant to Computer Graphics systems and software environments.

3. (**40 points**) Please define and explain the following commonly-used terms in computer graphics: (1) Raster graphics; (2) Virtual reality and augmented reality; (3) Graphical user interface; (4) Computer aided design and manufacturing; (5) Data visualization; (6) Ray casting; (7) Calligraphic display; (8) Frame buffer; (9) Spatial coherence; (10) Image resolution; (11) Image processing; (12) Linear transformation; (13) Ray casting; (14) Polygon clipping; (15) Implicit representation of a line equation; (16) Parametric representation; (17) Window and view-port; (18) Affine transformation; (19) Homogeneous coordinates; (20) Halftoning;

Programming Part

PLEASE NOTE THAT, all programming parts (including course projects, which will be discussed in class in detail later) in this course will be graded based on: (1) Correctness of program functionality; (2) Efficiency of program execution; and (3) Quality of programming code (i.e., whether they are understandable with proper comments)!

The main objective of this programming part is to learn the most fundamental component of 2D graphics techniques (i.e., the draw-pixel functionality and rasterization) and apply them to the line drawing and the curve drawing. In essence, you are only allowed to **use the draw-pixel command** to handle line geometry and extend the drawing procedure to further deal with poly-lines (including many line segments) and commonly used curves such as circles and ellipses. Please write your program (aided by OpenGL) that implements the drawing of poly-lines and commonly-used curves. In essence, you are only allowed to use the **`GL_POINTS`** mode in OpenGL drawing commands. You are also required to use OpenGL in the **core-profile** mode (modern OpenGL). Deprecated functions in immediate mode (legacy OpenGL) are **not** permitted.

Before you proceed, carefully review both the course materials and the attached **midpoint.pdf** and **scan-conversion.pdf**. If you encounter difficulties understanding the materials, please consult the original textbook, *Computer Graphics with OpenGL Fourth Edition*. It is crucial to emphasize that the attached materials serve solely as a reference. Only incorporate external content after conducting your own verification, particularly when dealing with source code. It is your responsibility to ensure your program functions in all scenarios; refrain from mere copy-pasting.

1. (**80 points**) Draw a line segment whose slope m satisfies $0 \leq m \leq 1$ by specifying the starting and ending points using the midpoint algorithm (detailed in our class lecture, course textbook, and **midpoint.pdf**). Your program should allow users to interactively enter positions (including both x coordinate and y coordinate) of two vertices that define a line geometry.

   In an ideal situation, your program should be able to handle all possible line configurations which afford different slope values on the 2D settings. However, in order for all of us to have a comfortable and excellent start, our course TA will provide a detailed document/code on how to achieve the full functionality of (1) (for your convenience towards better learning this part of Computer Graphics). Therefore, I am strongly suggesting that you should simply follow what he is suggesting all of us to do here (of course, you are welcome to design your own technical solutions). **FOR THIS PART ONLY**, you **ONLY** need to handle the simplest case here, i.e., when the line slope m is between 0 and 1. Its possible extensions and generalizations will be handled in (2). Once again, our course TA will provide you with a complete solution (only for your convenience).

   User Interface. The user will press **key 1** to enter this part. The user will left-click the mouse to specify the starting position of a line segment, then move the mouse cursor to the endpoint, then right-click the mouse to have it finalized. When the user moves the mouse, there should be a preview of the line (which will be finalized after the right click.)

2. (**60 points**) Based on what you have already accomplished above (i.e., (1)), please generalize your line drawing functionality to all other general cases, including:
   (A) m is between 0 and -1;
   (B) $m > 1$; (**note that**, including the correct handling of a vertical line, when m is in fact infinity); and
   (C) $m < -1$.

3. (**20 points**) After you have implemented the first two parts of this assignment (detailed above), you should be able to extend your line geometry drawing to handle poly-lines which consist of n points with a clear indication of a starting point and an ending point. Furthermore, you should generate a polygon which essentially is a closed poly-lines, in which the starting and the ending points are the same.

   User Interface. The user will press key **3** to enter this part. The user will use consecutive left clicks to add new vertices to a poly-line and right-click the mouse to add the last vertex and have it finalized. Moreover, if the user is holding key $\mathbf{C}$ when right-clicking the mouse, your program should further connect the last vertex with the first vertex, completing the poly-line into a closed polygon. Once again, there should be a preview of the poly-line or polygon (which will be finalized after the right-click.)

   To make the process crystal clear, let's consider the user wants to draw a triangle $\mathbf{ABD}$. The user would do the following:
   (a) Left-click the mouse, which defines $\mathbf{A}$;
   (b) Move the mouse ($\mathbf{A}$ and the mouse cursor should be connected);
   (c) Left-click the mouse again, which defines $\mathbf{B}$, and line $\mathbf{AB}$ is fixed;
   (d) Move the mouse ($\mathbf{B}$ and the mouse cursor should be connected);
   (e) Right-click the mouse, which defines $\mathbf{D}$, and line $\mathbf{BD}$ is fixed. If the user is holding key $\mathbf{C}$, line $\mathbf{DA}$ should also be connected and displayed immediately;
   (f) Left-click the mouse again, which cleans the viewport and starts drawing another poly-line or polygon.

4. (**40 points**)

   (A) Generalize the midpoint line drawing algorithm to the 2D drawing of a circle:

   $$(x - x_0)^2 + (y - y_0)^2 = r^2 \quad (r \neq 0).$$

   (B) Generalize the mid-point line drawing algorithm to the 2D drawing of an ellipse:

   $$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1 \quad (ab \neq 0).$$

   For both shapes, we expect you to handle the parameters interactively. Moreover, you are **not** permitted to draw circles/ellipsis as concatenations of multiple line segments. Please implement the circle/ellipse rasterization algorithms as detailed in the attached **midpoint.pdf**.

   User Interface. The user will press key **4** to enter this part:

   (A) Circle: The user will press and hold key **Shift** to indicate he/she's drawing a circle. The user will left-click the mouse to specify the circle's center (x_0, y_0) and right-click the mouse to specify a point on the circle (which will determine the radius of this circle.) Again, there should be a preview of the circle, and the preview should be finalized after the right-click.

   (B) Ellipse: The user will left-click the mouse to specify the center $\mathbf{C}(x_0, y_0)$ of the ellipse, and right-click the mouse to specify a point $\mathbf{P}$, where the absolute value of vector $\mathbf{CP}$'s x, y components denote a and b respectively. Again, there should be a preview of the ellipse, and the preview should be finalized after the right-click.

5. (**50 points**) (**BONUS PART-1, 25 points for (A) and (B) each**)

Generalize your circle and ellipse drawing procedures to handle the drawing of:

(A) Any quadratic curve defined by:

$$y(x) = a_2x^2 + a_1x + a_0;$$

(B) Any cubic curve defined by:

$$y(x) = a_3x^3 + a_2x^2 + a_1x + a_0,$$

where y is a function of x, and a_i $(i = 0, 1, 2, 3)$ are user-specified constants (serving as parameters). The coordinate system is that: The origin is the bottom-left corner of the viewport, the positive x-axis points toward the right, and the positive y-axis points toward the top. Once again, unless the parameters do represent a line (e.g., $a_2 = 0$ for quadratic curves), you are **not** permitted to draw polynomial curves as concatenations of line segments. Please extend the midpoint algorithm to handle curves directly.

Please note that, you can use other function plot software to help you understand some reasonable values for a_i's so that we will be able to see some interesting examples. To help you expedite this process, you could simply decide and suggest 2-4 different combinations of these parameters and read them from a file, please be specific about possible combinations of these parameters in your submission.

User Interface. The user will press key **5** to enter this part. The parameters should be read upon switching to this mode from the configuration file `sbuid_hw1/etc/config.txt`. This configuration file has exactly one line, which contains four space-separated floating-point numbers, separately denoting a_3, a_2, a_1, and a_0. For quadratic curves, we simply have $a_3 = 0$.

6. (**50 points**) (**BONUS PART-2, 25 points for (A) and (B) each**)

   (A) Draw a triangle and scan-convert all of its interior pixels so that a triangle will have a solid shading, and then extend the drawing capability to the display of a quadrangle and a pentagon. For this question, you are **ONLY** required to consider the convex cases. Please note that, to simplify the processing algorithm, we can simply limit our program to the display of a simple quadrangle and a simple pentagon (without the proper handling of edge-edge self-intersection, as explained in the lectures, i.e., you do **NOT** need to worry about cases where edge-edge self-intersection occurs, and you do **NOT** need to handle concave cases).

   (B) You should be able to handle the scan-conversion of any-sized polygons (i.e., n-gons). You **MUST** detect whether the input polygon is a simple polygon by checking whether any two line segments intersect with each other besides their endpoints. Please provide users with any meaningful visual feedback whenever and wherever such self-intersection occurs. Please note that, we have explained different ways to detect edge-edge self-intersections in our lectures extensively, and you are free to use different methods and techniques to properly provide the correct detection, you are not required to use the methods we have discussed in our class, but you want to make sure that the method you are employing is really working correctly and robustly, handling all different cases without any problem! Your program should have an option that discards any non-simple polygons and only keeps the simple polygons for the following tasks. For this part, please make sure that your program is **ONLY** required to support convex polygons with any number of edges (i.e., n-gon). Since this is a bonus part, we expect that your program should be able to handle both convex and concave cases.

User Interface. The user should be able to conduct scan conversion and complex polygon removal within part 3 (i.e., where you implemented wireframe polygon rasterization): After the user finalizes a wireframe polygon, he/she should be able to further scan-convert its interior pixels by pressing key **F**. However, if the polygon is complex, your program should not scan-convert the polygon; it should keep this polygon as a wireframe, and mark the self-intersecting edges in **red** (while other edges are in white, as in our program template). Please note that the user will not click the interior of these shapes, so a flood fill algorithm will not work here. The solution here is the scanline algorithm as detailed in our lectures and the attached **scan-conversion.pdf**.