Computación Gráfica - TP: Algoritmos de Rasterizacion

1. Resumen de tareas

1. Implementar un algortimo de rasterización de segmentos (completar la función `drawSegment` en *RasterAlgs.cpp*).
2. Implementar un algortimo de rasterización de curvas (completar la función `drawCurve` en *RasterAlgs.cpp*).

- Las implementaciones que desarrolle (puede utilizar cualquiera de los algoritmos estudiados) deben satisfacer el requisito de contigüidad, y además asegurarse de no pintar dos veces un mismo pixel.

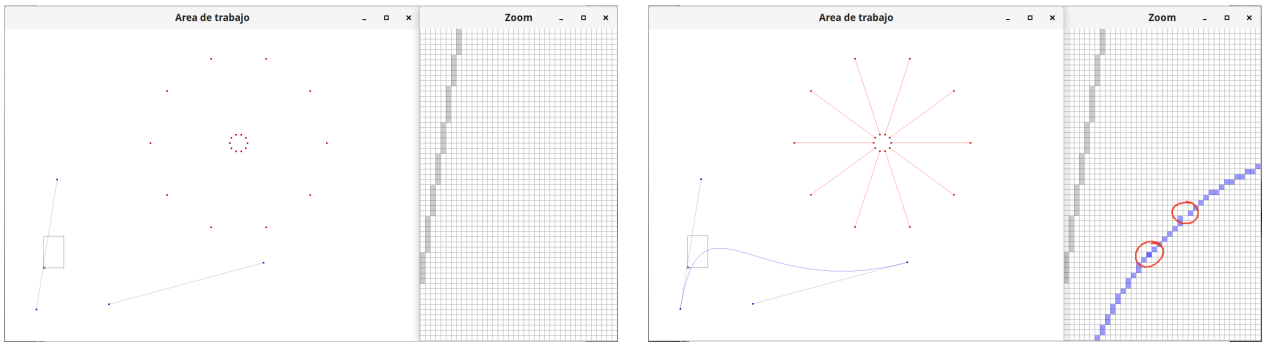
2. Consigna detallada

El objetivo es **implementar dos algoritmos de rasterización, uno para segmentos de recta y otro para curvas**. Para ello puede basarse en cualquiera de las opciones que se discutieron en la clase de teoría: puede utilizar DDA de linea o Bresenham para el segmento de recta, y puede utilizar DDA de curvas o subdivisiones para la curva.

Ambos se deben implementar en el archivo *RasterAlgs.cpp*, y dado que en este tp el objetivo es evaluar los algoritmos teóricos, y no alguna técnica específica de OpenGL, no hace falta que el alumno analice el resto del programa[1].

Al ejecutar el programa el alumno verá una ventana donde se pueden modificar los extremos de 10 segmentos de rectas (los vértices rojos). Las posiciones iniciales de estos segmentos son tales que hay segmentos con todas las combinaciones de tendencias (horizontales y verticales) y sentidos de avance (+x,+y,-x,-y), para testear la función de rasterizacion en todos los escenarios posibles.

Además, hay cuatro puntos de control adicionales (los vértices azules) que sirven para definir una curva[2], que se utilizará para probar el algoritmo de rasterización de curvas que el alumno proponga. La siguiente captura muestra el estado inicial del programa antes de resolver el TP, y en la segunda con una primera resolución parcial.



Además de la ventana principal, el programa muestra una segunda ventana con un zoom de la imagen de la principal. Esta sirve para analizar con más detalle el resultado. Por ejemplo, en la segunda captura se han señalado 2 errores

[1] Por ejemplo, no hace falta analizar *main.cpp*, que en este caso será un poco diferente al de otros tps por ser 2D, gestionar dos ventanas en simultáneo, y porque la segunda ventana muestra en realidad un especie de captura de pantalla ampliada de la primera en lugar de dibujar algo nuevo/propio, por lo que ambas ventanas comparten parte del contexto OpenGL.

[2] Es específicamente una curva *de Bezier*. No importa ahora conocer los detalles de la misma, se estudiarán más adelante en la unidad de *Curvas y Superficies*. Pero dado que es el tipo de curva más usual en programas de diseño vectorial en 2D, si ha dibujado alguna vez curvas arbitrarias en programas como *Inkscape*, *Corel Draw*, *Adobe Ilustrator*, o hasta en *Microsoft Office* o *LibreOffice* probablemente ya esté familiarizado con la interfaz de los dos extremos y las dos ”manijitas” para manipularla.

del algoritmo implementado. El zoom muestra que hay pixeles pintados más de una vez[3] y pixeles que faltan (al menos un lugar donde se rompe la contigüidad). En este ejemplo se utilizó el algoritmo de DDA para curvas en su versión más básica, sin agregar las correcciones necesarias para evitar estos problemas generados por la *aproximación* (no es exacto) del salto en base a la derivada. Si el alumno también decide utilizar DDA deberá corregir estos problemas. Si el alumno en cambio utiliza subdivisiones, deberá implementar correctamente la rasterización de segmentos (en cuanto a asegurarse de incluir solo uno de los extremos, recordar el criterio del rombo) para evitar que se repinten los pixeles donde se unen dos segmentos.

En la ventana de zoom se puede utilizar la rueda del ratón para cambiar el nivel de zoom, o el drag para mover el area ampliada (también puede moverla desde la ventana principal con el vértice gris que está en la esquina superior izquierda de dicha area).

Las funciones a implementar tienen los prototipos:

```
void drawSegment(paintPixelFunction paintPixel, glm::vec2 p0, glm::vec2 p1);
void drawCurve  (paintPixelFunction paintPixel, curveEvalFunction evalCurve);
```

El argumento `paintPixel` es un puntero a función. Cada vez que el algoritmo de rasterización decida que debe pintar un pixel, lo deberá pintar a través de esta función (ej: `paintPixel(p);` siendo `p` un `glm:vec2`).

En el caso del segmento de recta, los argumentos restantes (`p0` y `p1`) son los puntos extremos del segmento.

En el caso de la curva, el argumento restante es una función que representa a la curva paramétrica, y que dado un valor del parámetro (habitualmente denominado t o u, que en este caso irá entre 0 y 1), calcula y retorna el punto de la curva y su derivada para ese valor. Por esto el tipo de retorno de la función `f` es un `struct` con dos campos, `p` y `d` que se corresponden a punto y derivada respectivamente. Por ejemplo, si utiliza `auto r = evalCurve(t)`, siendo `t` un flotante entre `0` y `1`, entonces en `r.p` tendrá el punto y en `r.d` el vector derivada (ambos de tipo `glm::vec2`).

[3] Las curvas y los segmentos se pintan con $alpha \neq 1$, es decir semi-transparentes, entonces si un pixel se pinta dos veces se verá más oscuro que el resto.