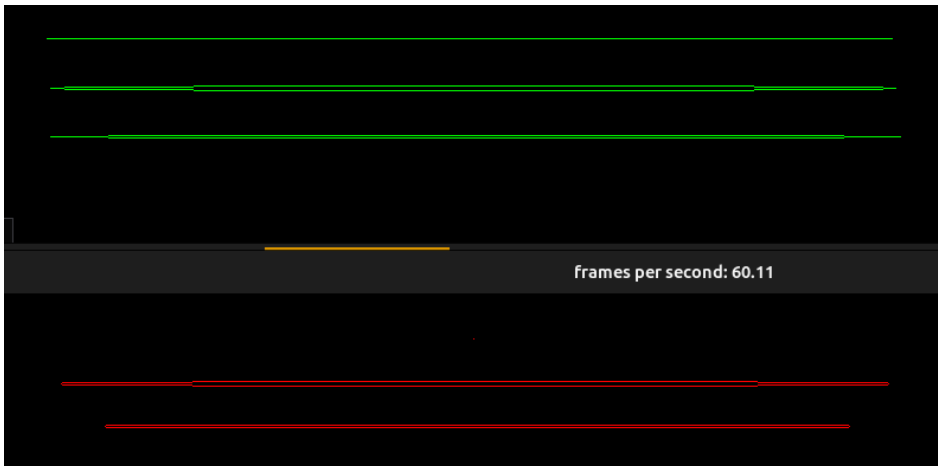
Facultad De Ciencias - UCV
Escuela de computación
CI: 30391915
Nombre: Rafael Eduardo Contreras

Tarea #2 - Algoritmo de despliegue de elipses optimizado

El algoritmo de despliegue de elipses usa la ecuación implícita para calcular el siguiente punto a dibujar. Este algoritmo utiliza como argumentos el centro de la elipse y los valores “a” y “b”. Creamos la versión original sin modificaciones y una versión optimizada, probamos que la salida de ambos programas fuera equivalente y la velocidad de los mismos. Este informe expone los resultados. Además, se agregaron links al repositorio de github correspondiente para que el lector pueda tener referencias al código.

Antes de continuar, existe un caso borde que el algoritmo original no contemplaba. Este ocurría cuando la elipse tenía un “b” muy pequeño. Este se arregla en la función drawEdgeCase, esta simplemente dibuja la línea recta (solo en bordes hasta cierto punto) que falta para completar la elipse. Nótese en la figura de abajo que la elipse de menor tamaño roja (que no contemplan este caso borde) fueron dibujados con coordenadas mayores a su tamaño (son más pequeños de lo que deberían ser).



Arriba líneas verdes que ejecutan la función **drawEdgeCase**. Abajo líneas rojas sin la función, se puede observar que esta elipse tienen un tamaño menor, además de que cuando b=0 desaparece completamente.

Como correr y usar el programa

En los archivos entregados, el main.cpp se corre igual que la anterior tarea sobre líneas. Este tiene una interfaz similar a la tarea 1 también. La única diferencia es que al dibujar elipses random, el color ya no es aleatorio, se usa el color seleccionado en pantalla.

Para correr los tests se debe compilar el archivo “test.cpp”, para Linux el Makefile para compilar el archivo se encuentra aqui. Para Windows, en el editor de texto usado para la tarea 1, es

necesario sacar del proyecto main.cpp y dejar solo test.cpp. Esto es debido a que ambos archivos definen una función “main”.

Optimizaciones

Para lograr llegar al algoritmo optimizado comenzamos con el hecho de que todas las variables dentro del ciclo crecen de forma lineal. Esto nos permite mover ciertos valores de tal manera que su modificación no afecte la ubicación del siguiente punto a trabajar. Un ejemplo de esto es la modificación de los valores m_x y m_y, inicialmente estos son los valores a comparar en el primer while.

Luego de definir los valores de m_x y m_y y como estos valores crecen, nos enfocamos en reemplazar las multiplicaciones que nos dan d por sumas y restas. Usando equivalencias simples logramos hacer esto, sobrando una constante que sumamos a d. Ahora llegábamos a resultados similares a “d += m_x + consta”. Para deshacernos del **consta** en varias instancias inicializábamos m_x con esta variable, hacíamos lo mismo con m_y para mantener la equivalencia de la condición del while y en otras instancias de “d += m_x + m_y + constb”, estábamos este incremento a la constante ya existente. Usando esta metodología y varias iteraciones se logró optimizar las operaciones del algoritmo.

Tests de comparación

Para estar seguros de que nuestro algoritmo optimizado obtiene los mismos resultados que el original, sobrecargamos la función “setPixel”. Esta función va a guardar los puntos generados por el algoritmo original. Luego hará lo mismo con el optimizado y comparará ambos vectores (también el tamaño de los vectores debe ser igual).

Es importante acotar que los tests de comparación se hacen con elipses creadas aleatoriamente. Además, el tamaño de ventana (máximo, ancho y alto) es variable, esto debido a que al testear las elipses en pantallas grandes, estos se deforman si no se usa long long. Por cada tamaño de ventana distinta se prueban 50 elipses. Las elipses comparadas se guardan en archivos dentro de “comparison”, aquí puede obtener las elipses generadas por cada algoritmo.

Los tamaños de ventana comparados son: 100x100, 200x200, 500x500 y 8000x8000.

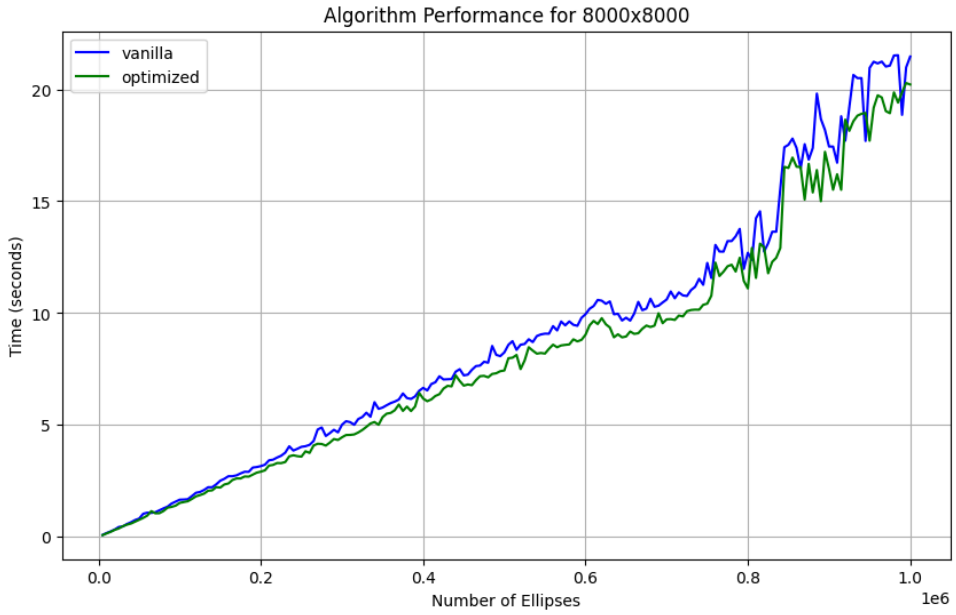
Benchmarks y resultados

La función de benchmark recibe la altura y anchura máxima, esto es crucial, puesto que una ventana grande tendrá elipses más costosas de generar (por el tamaño). La forma en la que los tests miden el desempeñó de cada algoritmo es constante y consistente, no usamos elipses aleatorias esta vez.

Cada prueba consiste en generar una lista de elipses random y luego probar nuestros algoritmos sobre esa lista. Tuvimos varios problemas de inconsistencia durante las pruebas

debido al caching del CPU, el algoritmo que era corrido con el cache ya lleno tenia ventaja sobre el otro. Para solucionar esto al probar un algoritmo, primero lo corriamos con todas las elipses, luego lo corriamos nuevamente tomando el tiempo. Finalmente los test consisten, en correr un loop que va desde 0 hasta n con saltos de m. En cada iteración se dibuja el vector de n elipses aleatorias y en cada nuevo test se generan nuevas elipses.

Los resultados demuestran que el algoritmo optimizado más rápido que el original. Esto se observa en el siguiente gráfico al comparar una prueba cualquiera (sin ruido) del algoritmo original y comparar su tiempo con el del algoritmo optimizado. Este gráfico es para un ancho y altura de elipse de 8000x8000 y 1.000.000 elipses.

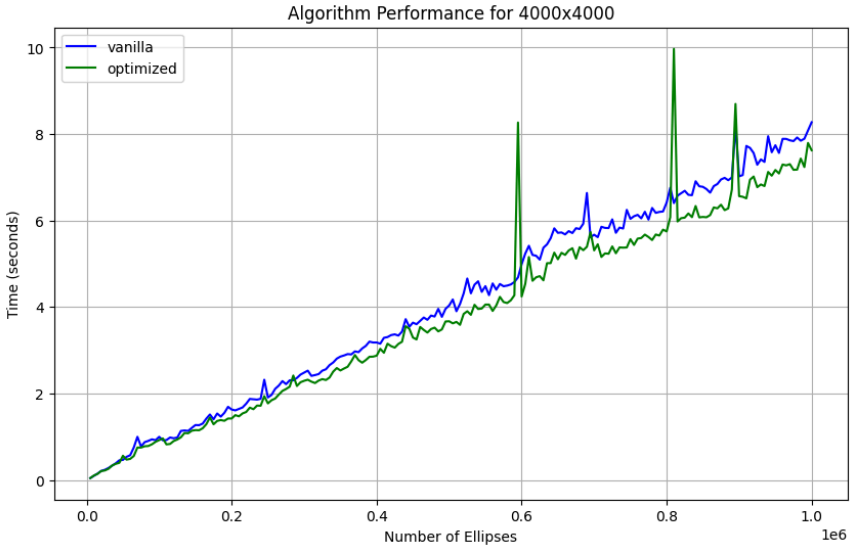


Desempeñó de algoritmos hasta 800k elipses. En pantalla de 8000x8000.

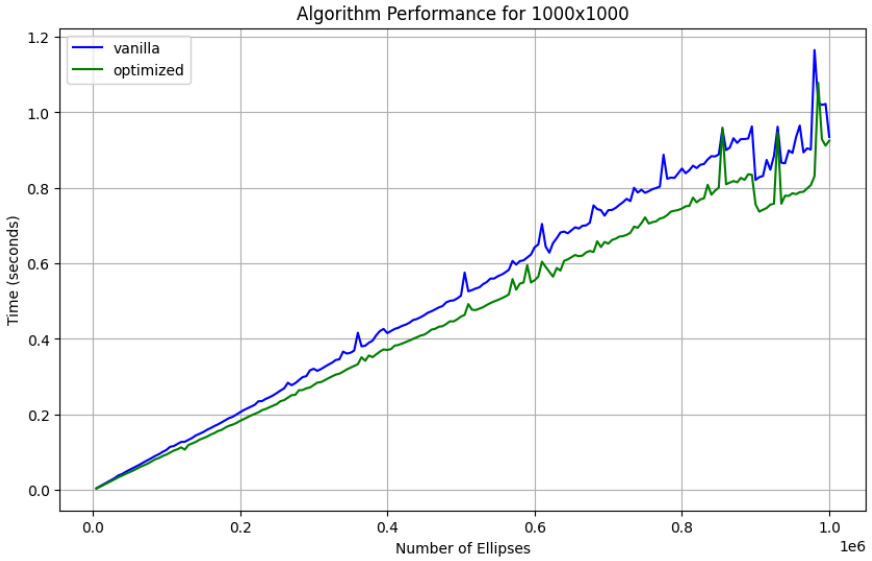
La mejora del algoritmo es apenas notable, tomando en cuenta la cantidad de elipses necesaria para observar el cambio. A partir de unos 200k elipses la mejora llega al 9%, esto lo sabemos por los csv generados, en 200k el algoritmo original toma 3.16s y el optimizado 2.91. En los 600k elipses el algoritmo original corre en 9.95s y el optimizado 9.03. En general se ve una tendencia de un 9% de mejora, excepto a partir de 800k elipses, donde parece que superamos el tamaño de caché y ambos tiempos incrementan drásticamente. Más resultados en la siguiente tabla:

Algoritmo	100k	200k	300k	400k	500k	600k	700k	800k	900k	1M
Original	1.65s	3.16s	5.01s	6.65s	8.24s	9.95s	10.6s	12.69s	17.44s	21.46s
Optimizado	1.51s	2.91s	4.44s	6.17s	7.43s	9.03s	9.72s	11.10s	16.44s	20.21s

A continuación otros benchmarks que fueron realizados con pantallas de distinto tamaño. Esto influye en la creación de elipses, ya que los mismos ahora serán menos costosos de crear por tener un menor tamaño.



Desempeñó de algoritmos hasta 800k elipses. En pantalla de 4000x4000.



Desempeñó de algoritmos hasta 800k elipses. En pantalla de 1000x1000.