

UNIVERSIDAD DEL VALLE DE GUATEMALA
CC3069 - Computación Paralela y Distribuida
Sección 10
Ing. Miguel Novella



Proyecto 1 - ScreenSaver usando OpenMp

Diego Arredondo 19422
Julio Herrera 19402
Diego Álvarez 19498
Grupo 4

Link al repositorio: [ScreenSaver3DFunction](#)

GUATEMALA, 22 de marzo de 2023

Índice

Introducción	2
Antecedentes	3
Cuerpo	4
Anexo 1 - Diagrama de flujo del programa:	4
Anexo 2: catálogo de funciones	4
Anexo 3 – Bitácora de pruebas	7
Conclusiones	12
Recomendaciones	13
Apéndice	14
Referencias bibliográficas	15

Introducción

El presente informe tiene como objetivo comparar el rendimiento de un screensaver desarrollado de manera secuencial y de manera paralela. El criterio de mejora utilizado para la comparación es el número de cuadros por segundo (fps) a los que corre el programa en ambas implementaciones. La comparación se realiza para determinar si el rendimiento del screen saver mejora al ejecutarse de manera paralela y así, determinar cuál es la mejor opción para su utilización. Se espera que los resultados obtenidos en esta comparación permitan establecer una conclusión clara sobre cuál es la mejor forma de implementar un screen saver en términos de rendimiento.

Antecedentes

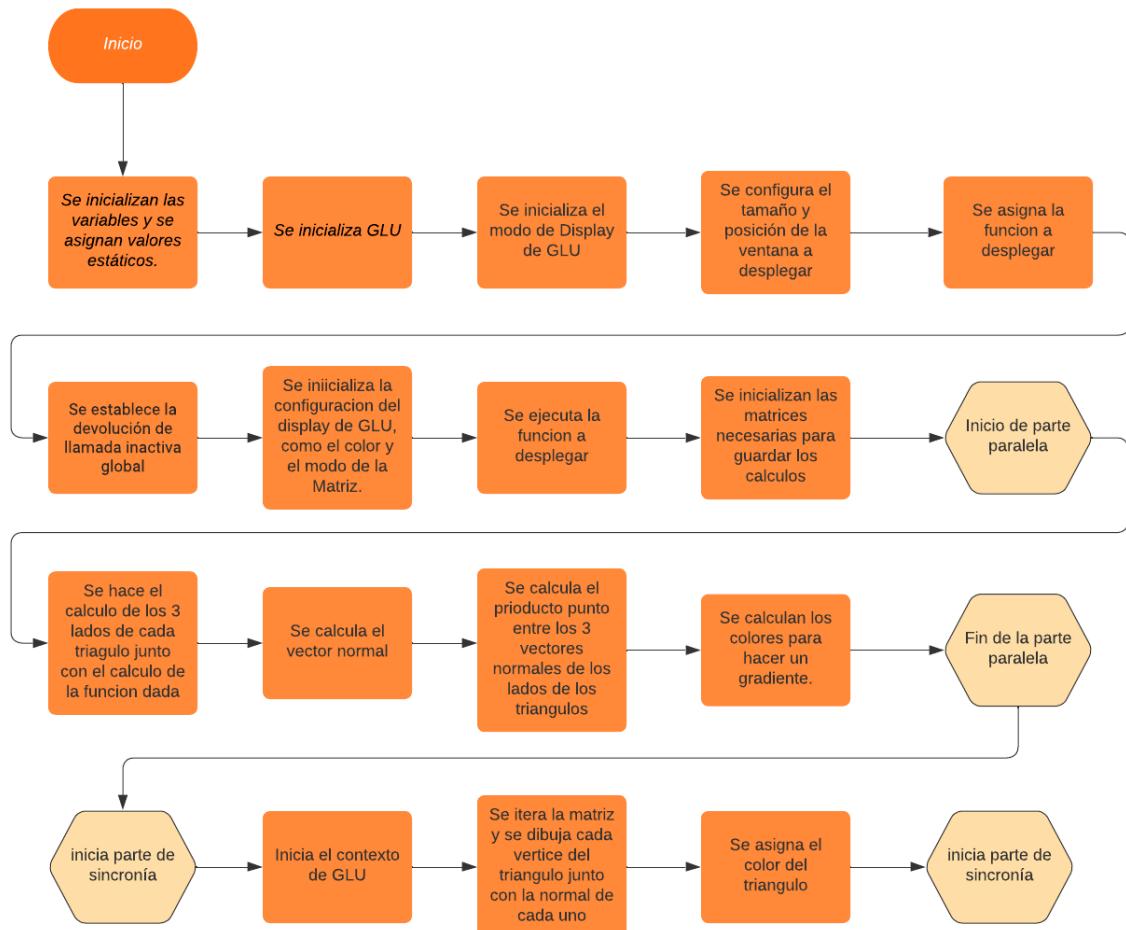
Los screen savers nacen en el año 1983 cuando se usaban monitores CRT, este tipo de monitores sufrían el problema que si se quedaban con la misma imagen en pantalla durante mucho tiempo, sucedía el “Burn-In” o quemado de pantalla ya que los fósforos que producen la luz en la pantalla tendían a perder su luminosidad cuando se usaba excesivamente un área específica, haciendo que quedaran imágenes “fantasma” en la pantalla. Para ello se crearon los *screen savers* para evitar que ningún área de la pantalla se quede durante un periodo largo de tiempo mostrando la misma imagen. (Lunduke, B., 2022)

Fue John Socha quien creó el que se considera el primer *screen saver*, un programa llamado SCRNSAVE, desde ese entonces se creó una tendencia muy apreciada tanto por los desarrolladores como por los usuarios. Los desarrolladores podían experimentar creativamente con las posibilidades gráficas por medio de algoritmos y por parte de los usuarios que consumían estos *screen savers* que ahora ya venían integrados en los sistemas operativos, como los clásicos laberintos de ladrillos, las tuberías de colores o las burbujas rebotando en pantalla. A finales de la época de los 90's se empezaron a comercializar las pantallas LCD que ya no sufrían del “Burn-In” por lo que los *screen savers* se volvieron innecesarios. (Stinson, L., 2022)

Hoy en día ya no es muy común ver los screen savers ya que es preferible suspender la pantalla para tener ahorro de energía o simplemente una imagen estática sin consumir recursos de la PC, por ello si se quiere hacer un *screen saver* como recurso estético es preferible que este sea atractivo y eficiente.

Cuerpo

Anexo 1 - Diagrama de flujo del programa:



Anexo 2: cat\u00e1logo de funciones

- **Init**
 - entrada: nada
 - salida: void
 - Descripci\u00f3n:
 - Define el color que se usará para limpiar la pantalla, es decir el color de fondo.
 - Indica que la cámara verá en modo ortogonal desde un punto de vista alejado del centro.
- **f**
 - entrada:

- moveFunX: variable float de la función matemática que al cambiar da una forma diferente.
 - x: float para indicar la posición en x en el plano.
 - z: float para indicar la posición en z en el plano.
- salida:
 - float: posición en y en el plano.
- Descripción:
 - aplica la función
$$y = moveFunX \cdot \cos(\sqrt{(x \cdot x) + (y \cdot y)} \cdot moveFunX \cdot 0.25)$$
- normalVector
 - entrada:
 - moveFunX: variable float de la función matemática que al cambiar da una forma diferente.
 - x: float para indicar la posición en x en el plano.
 - z: float para indicar la posición en z en el plano.
 - norm: puntero a float que guarda el array del vector normal.
 - Salida: void, ya que asigna valores a puntero.
 - Descripción:
 - Calcula el vector normal (x,y,z) aplicado a esa posición del plano para la función.
- idle
 - entrada: nada
 - salida: void
 - Descripción:
 - A nivel de OpenGL esta función se realiza en “background” durante cada renderizado de pantalla.
 - Se utiliza incrementar thetaX que define la rotación de la cámara alrededor de la función.
 - Aumenta moveFunX a partir del tiempo transcurrido, para animar la función al recibir este parámetro.
- keyboard:
 - entrada: nada
 - salida: void
 - Descripción:
 - Se utiliza para indicar las acciones al presionar ciertas teclas del teclado, todos los parámetros tienen límites.
 - “q” disminuye la velocidad de rotación de la cámara (puede ser negativa).
 - “w” aumenta la velocidad de rotación de la cámara.
 - “a” disminuye la amplitud de la animación de la función matemática.
 - “s” aumenta la amplitud de la animación de la función matemática.
 - “z” disminuye la velocidad de la animación matemática.
 - “x” aumenta la velocidad de la animación matemática.
- main:
 - entrada: nada
 - salida: int (exit del programa).
 - Descripción:

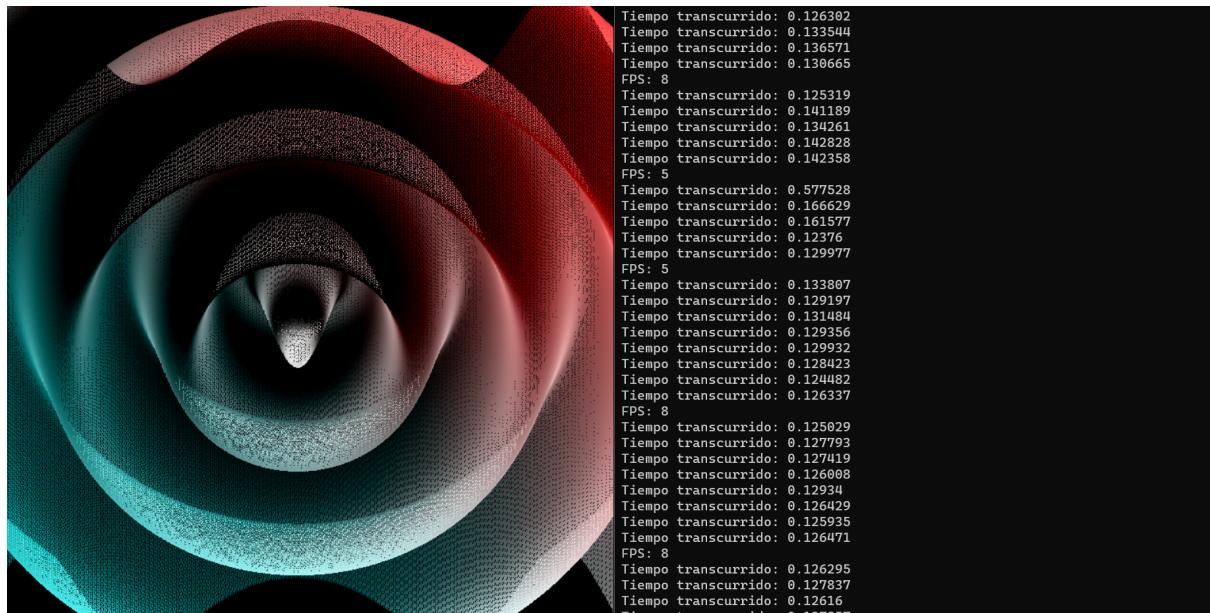
- Indica el flujo de OpenGL para correr el programa, levantar la ventana, también indica algunos parámetros de esta como el tamaño, la posición y el título. Indica las funciones que se ejecutarán en cada renderizado en pantalla, la función display, idle y también la de keyboard. Por último inicia el loop principal y retorna 0 al terminar.
- display:
 - entrada: nada
 - salida: void
 - Descripción:
 - Calcula el tiempo que se tarda en ejecutar cada renderizado en pantalla.
 - Muestra los frames en consola, calcula cuantos renderizados hizo en cada segundo, este dato solo se muestra cada segundo real que pasa.
 - Limpia la pantalla e indica el modo de renderizado respecto a la posición de la cámara, que va cambiando su rotación horizontal dada la variable thetaY.
 - Define las variables xGap y zGap que sirven para dar ese *offset* entre cada punto de la función y que cuadre con el tamaño del plano y la cantidad de puntos.
 - Para el paralelo, define los arrays que guardarán los datos del cálculo de cada posición m, n (o x,y en el plano), estos son:
 - Posición x,y,z para cada punto del triángulo a dibujar.
 - color de cada punto según la normal del punto y la dirección de luz.
 - Normal de cada punto del triángulo a dibujar.
 - Realiza un recorrido, por cada posición z, en cada posición x del plano, es decir por cada punto de la malla 2D, para calcular la posición respectiva en y. Esto se traduce a un doble for, uno en n y otro en m, el cual es paralelizado por la cláusula de OpenMP indicando el número de hilos, collapse 2 y schedule *dynamic* para distribuir la carga todas las iteraciones de ambos fors a los hilos disponibles, se comparó con static y varios *block_size* y *dynamic* dió mejores resultados. También se indica que los arrays son *shared*.
 - En cada iteración, calcula la posición x, y, z, la normal y el color de cada punto para así guardarlo en su respectiva posición de los arrays.
 - En la versión secuencial dibuja el triángulo dentro de cada iteración.
 - En la versión paralela, como solo el hilo maestro es capaz de renderizar en pantalla, se recorre nuevamente cada punto de la malla 2D para obtener los datos de los arrays y renderizar los triángulos.
 - Limpia el buffer.

Anexo 3 – Bitácora de pruebas

Para las pruebas se varió la cantidad de líneas calculadas representadas por el valor de n y m por lo que a continuación se presentarán diversos speed ups y eficiencias correspondientes a estos valores.

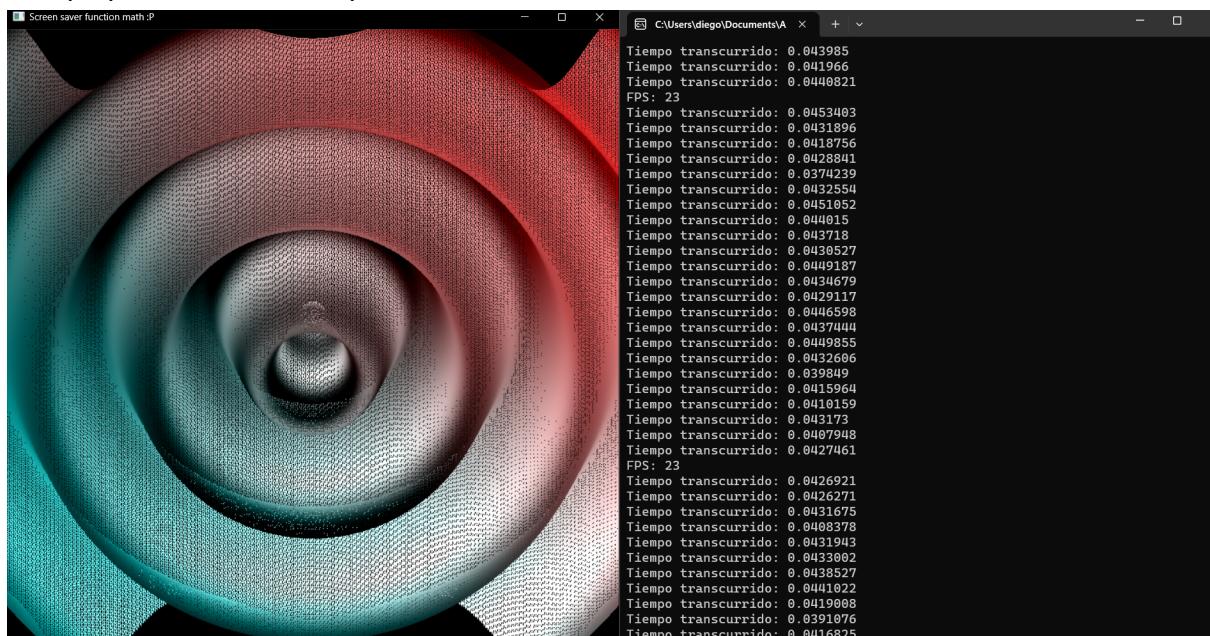
N: 300 y M: 300

Tiempo secuencial: 0.113s en promedio con 8 FPS.



Utilizando 10 hilos.

Tiempo paralelo: 0.04s en promedio con 23 FPS

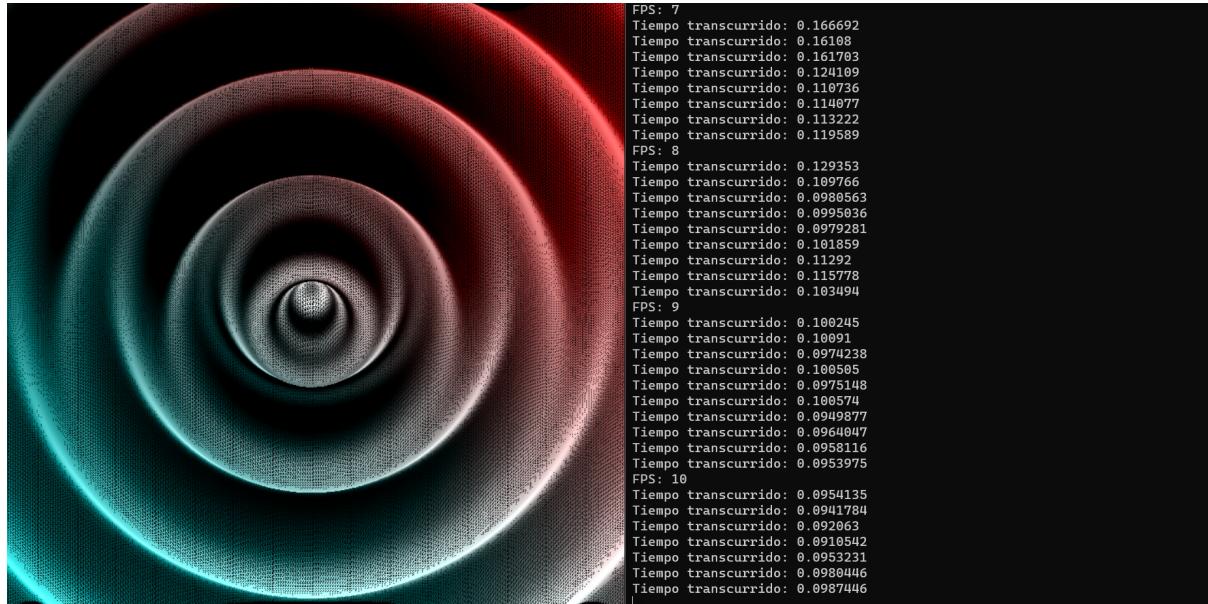


Speed Up: 2.825

Eficiencia: 0.2825

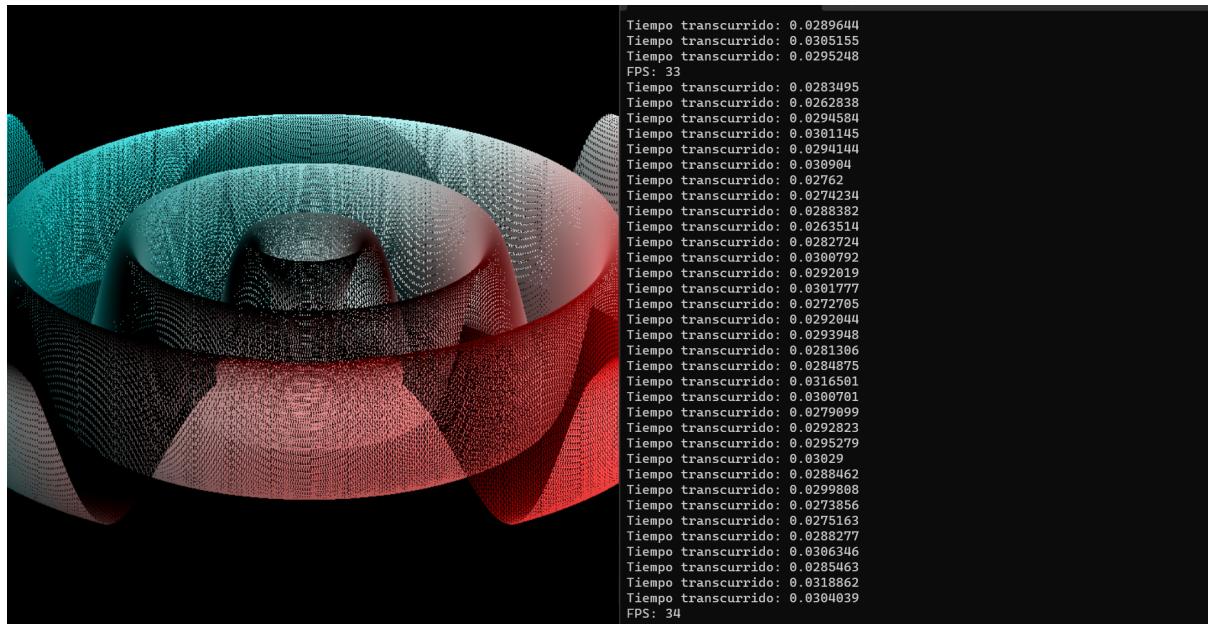
N: 250 y M: 250

Tiempo secuencial: 0.09s en promedio con 9 FPS.



Utilizando 10 hilos.

Tiempo paralelo: 0.023s en promedio con 33 FPS

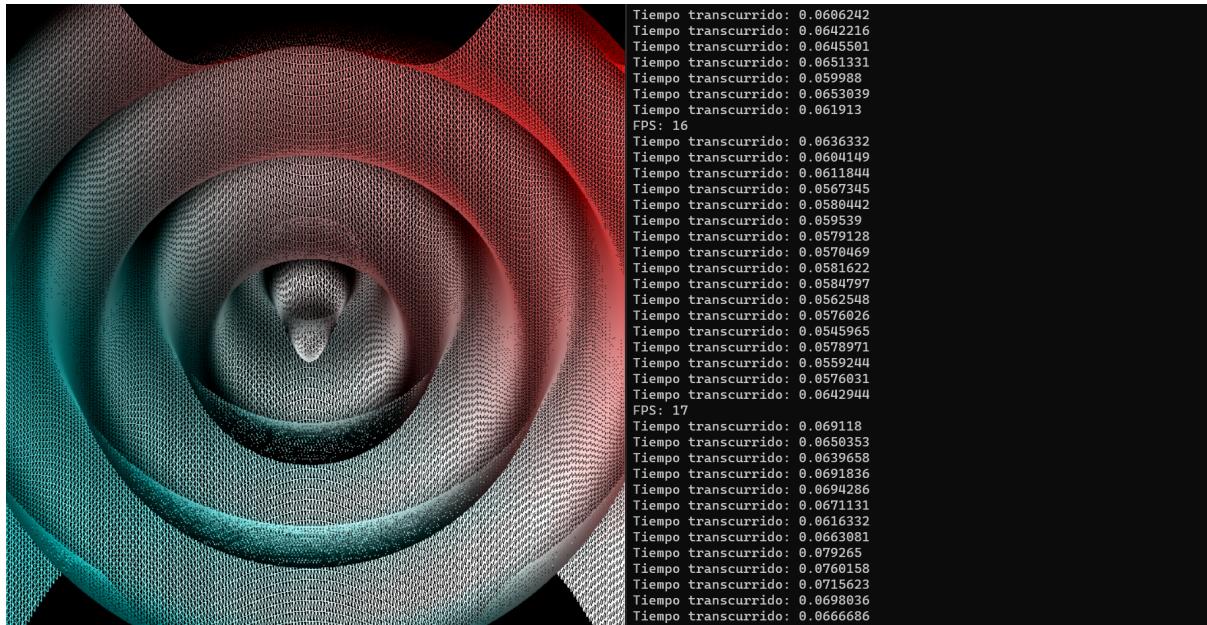


Speed Up: 3.91

Eficiencia: 0.391

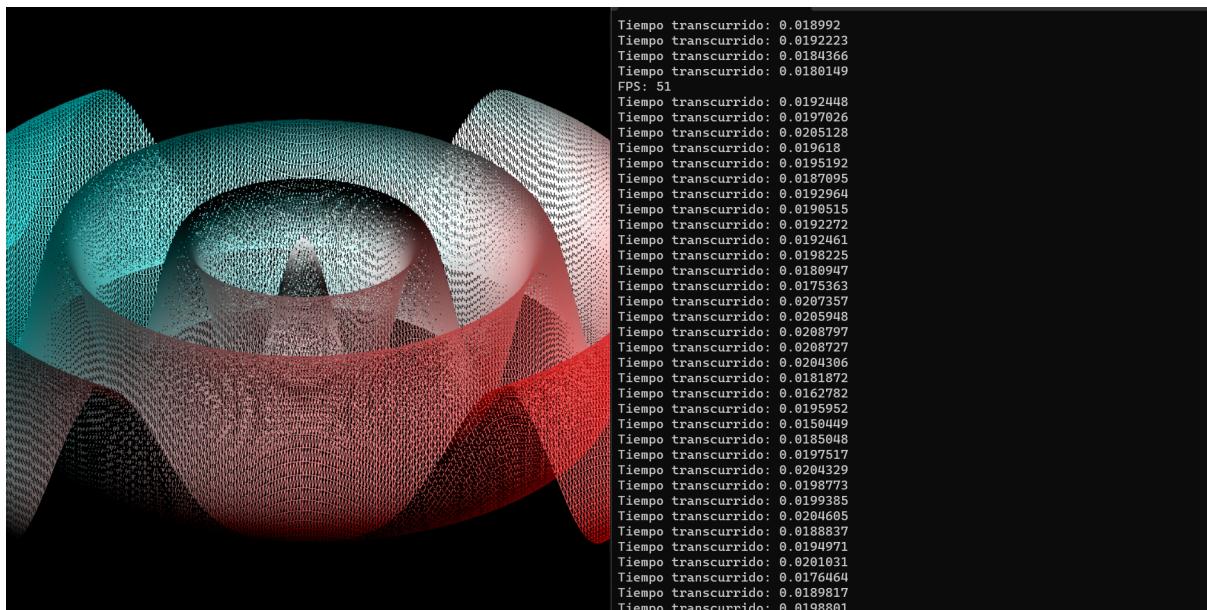
N: 200 y M: 200

Tiempo secuencial: 0.06s en promedio con 17 FPS.



Utilizando 10 hilos.

Tiempo paralelo: 0.019s en promedio con 50 FPS

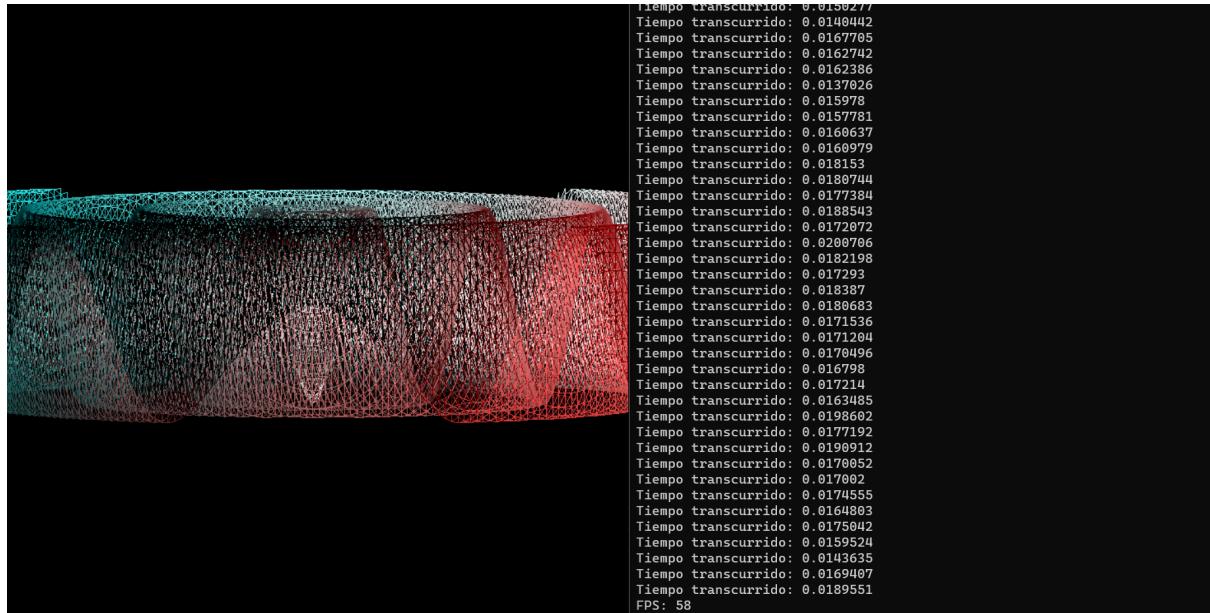


Speed Up: 3.16

Eficiencia: 0.316

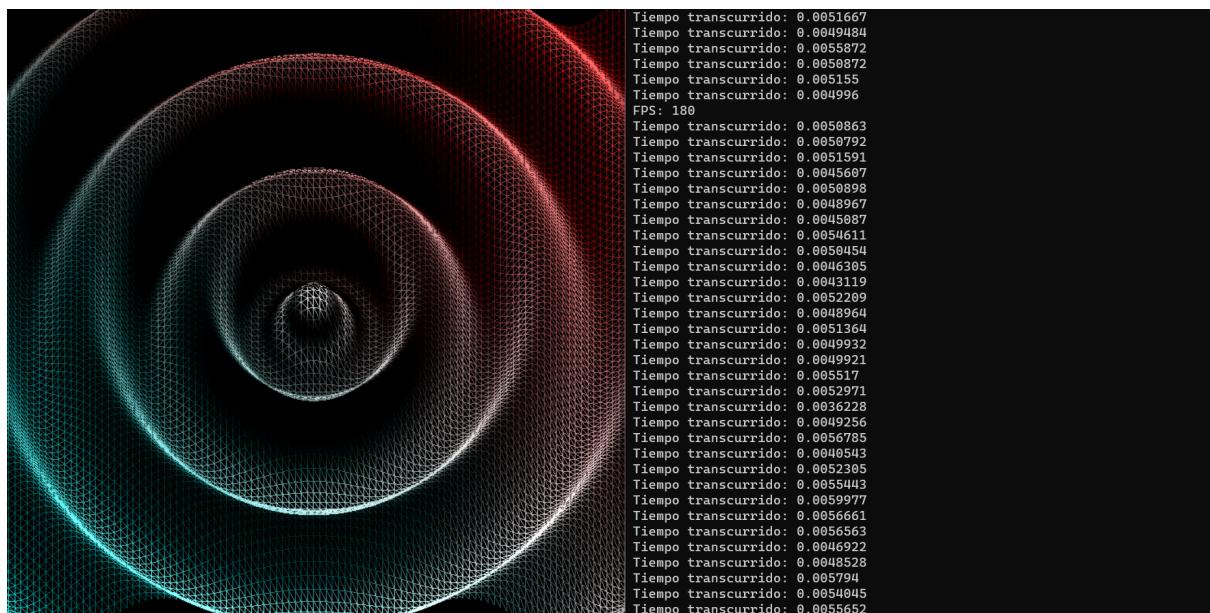
N: 100 y M: 100

Tiempo secuencial: 0.015s en promedio con 60 FPS.



Utilizando 10 hilos.

Tiempo paralelo: 0.005s en promedio con 180 FPS

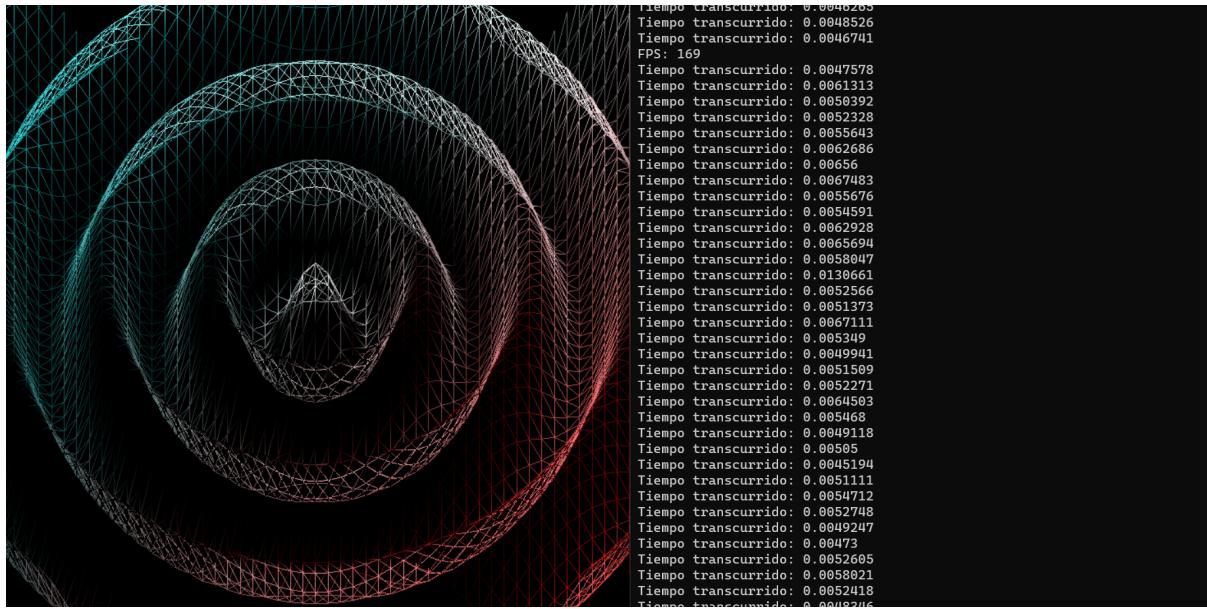


Speed Up: 3

Eficiencia: 0.3

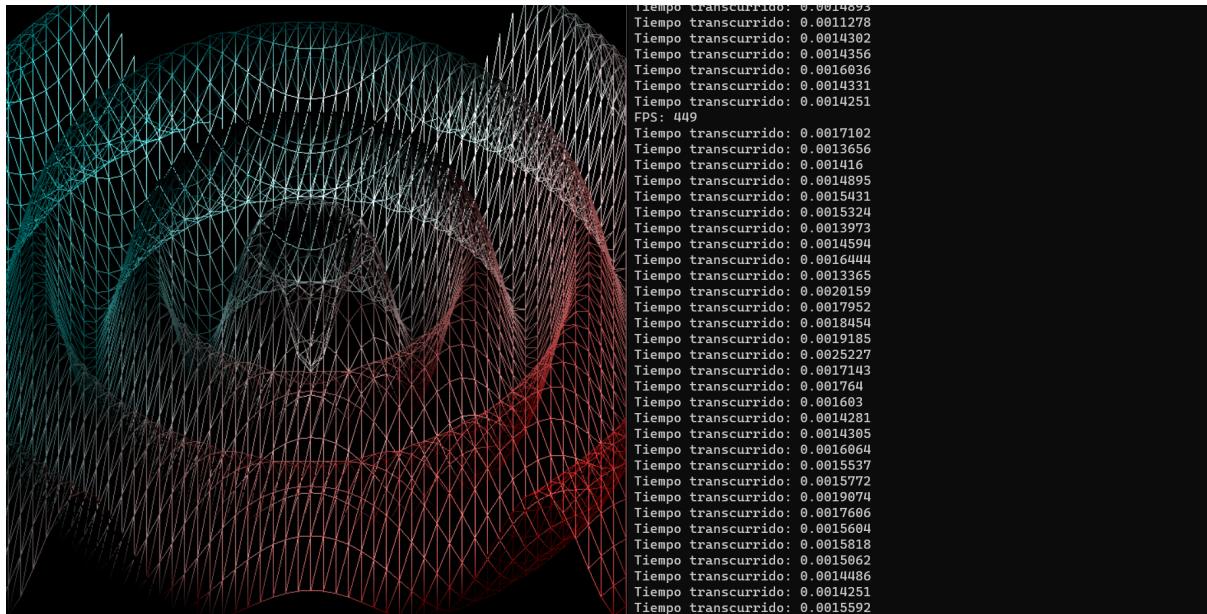
N: 50 y M: 50

Tiempo secuencial: 0.004s en promedio con 169 FPS.



Utilizando 10 hilos.

Tiempo paralelo: 0.0015s en promedio con 440 FPS



Speed Up: 2.667

Eficiencia: 0.2667

En promedio se obtuvo un speedup de 3.11 lo que nos indica que el programa en paralelo toma un tercio de tiempo con respecto al secuencial en ejecutarse. La eficiencia nos dice que un 31% de los recursos de procesamiento fueron utilizados en pro de automatizar la tarea. Que en este caso fue hacer el cálculo de los datos previos a pintarse.

Conclusiones

1. Dados los resultados obtenidos, se concluye que el paralelizar los cálculos de la posición de los puntos para dibujar los triángulos de una función matemática, hace más eficiente su graficación.
2. La estructura de *arrays* de C++ es adecuada para guardar los cálculos de todos los puntos, normales y colores para luego recorrerla y dibujar cada triángulo en pantalla, esto ya que al definir el tamaño de las listas se puede tener un fácil acceso a sus valores en tiempo constante.
3. La cláusula *schedule* en su modo *dynamic* es la mejor opción para distribuir el trabajo a los hilos, por sobre la cláusula *static* (con valores de *block_size* de 1, 5 y 16)
4. Debido a que la librería GLUT renderiza por defecto con un solo hilo, la paralelización de renderizado no es posible, por lo que aplicar paralelismo a los cálculos antes fue la solución para optimizar tiempos.

Recomendaciones

1. Dados los resultados, se recomienda implementar funciones matemáticas más complejas para poder evaluar también la eficiencia de la paralelización.
2. Se recomienda utilizar otro paquete de graficación, para evaluar si la paralelización es eficiente en distintos contextos de graficación en C++, como vulkan que aprovecha más la GPU.
3. Se recomienda profundizar en las posibilidades de OpenGL u otras librerías para permitir el renderizado paralelo y evitar la solución tomada en este proyecto, de guardar los valores para renderizarlos luego solo en el hilo principal.

Apéndice

Instalación GLUT

GLUT es una librería por parte de OpenGL la cual provee directo acceso a funciones de esta misma.

- Descargar los archivos binarios de GLUT para windows de <https://www.opengl.org/resources/libraries/glut/>.
- Extraer el contenido del archivo ZIP en un folder de fácil acceso.
- Copiar el archivo glut32.dll en el directorio C:\Windows\System32

- Abrir o crear un proyecto en Visual Studio, ir al explorador de soluciones y dar click en propiedades.
- En la sección “Propiedades de configuración” seleccionar “Enlazador” y luego “Entrada”.
- Seleccionar el campo “Dependencias adicionales” y agregar glut32.lib a la lista de dependencias.
- Luego, en la sección “Directorios de VC++”, seleccionar “Directorios de biblioteca” y agregar la ruta de donde descomprimió el zip en el paso 2.
- Guardar la configuración del proyecto y pruebe su programa.

Instalación MSYS2

MSYS2 es una colección de herramientas y librerías la cual provee un ambiente cómodo para el desarrollo de aplicaciones nativas para el software de Windows.

- Descargar el instalador de <https://www.msys2.org/>
- Seguir las instrucciones del instalador sobre donde colocar los archivos.
- Cuando se haya completado la instalación se debe correr el archivo “msys2_shell.bat”.
- Esto abrirá una consola en la cual para chequear su correcto funcionamiento deberá copiar, pegar y ejecutar el siguiente comando ‘pacman -S mingw-w64-x86_64-gcc’ el cual descargara herramientas para empezar a compilar.
- Para que el comando ‘gcc’ sea accesible en todo el sistema, deberá agregar una variable de entorno.
- En windows deberá ir a la búsqueda y escribir ‘editar variables de entorno’
- Seleccionar ‘Variables de entorno’
- Sobre las variables del usuario respectivo de su persona, seleccione PATH.
- Agregar la ruta del archivo en el que está alojado el ejecutable de msys2.

Referencias bibliográficas

Lunduke, Bryan. (2022). The Definitive History of Screensavers - Part 1. Substack. Extraído de: <https://lunduke.substack.com/p/the-definitive-history-of-screensavers>

Stinson, Liz. (2022). The Ever-changing Art of the Screensaver. Eye on Design. Extraído de: <https://eyeondesign.aiga.org/the-ever-changing-art-of-the-screensaver/>

The Khronos Group Inc. (s. f.). GLUT - The OpenGL Utility Toolkit. https://www.opengl.org/resources/libraries/glut/glut_downloads.php