Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación


# Proyecto 1 - ScreenSaver

Marco Ramirez 19588
Christian Perez 19710
Estuardo Hernández 19202

Guatemala, 22 de Marzo de 2023

# Introducción

El objetivo de este proyecto es desarrollar un Screen Saver utilizando la tecnología OpenMP y el paradigma de paralelismo. Un Screensaver es una aplicación que se ejecuta cuando el equipo no está en uso, y su objetivo es evitar que se queme la pantalla y ahorrar energía. En este proyecto, se propone desarrollar un Screensaver que aproveche al máximo el rendimiento del procesador mediante el uso de OpenMP, una tecnología de programación en paralelo que permite la ejecución simultánea de múltiples tareas en un solo procesador. El resultado será un screensaver eficiente y rápido que permitirá al usuario disfrutar de su ordenador sin preocupaciones mientras ahorra energía. Este proyecto no solo aborda la optimización del rendimiento en el desarrollo de aplicaciones, sino que también tiene un impacto positivo en la conservación del medio ambiente, al reducir el consumo de energía.

# Antecedentes

Los screensavers empiezan cuando el ratón y el teclado han estado ociosos por un período específico de tiempo. Se utilizan por tres razones:

1. Para proteger una pantalla de una quemadura del fósforo causada por imágenes estáticas.
2. Para ocultar información delicada dejada sobre una pantalla.
3. Como herramienta de marketing para las empresas.

Para crear un screensaver, la mayoría de los desarrolladores crean un módulo de código fuente conteniendo 3 funciones requeridas y las enlazan con la librería de screensaver. Un módulo de screensaver es responsable solo por configurarse a sí mismo y para proveer efectos visuales. Una de las 3 funciones requeridas en un módulo de screensaver está contenida dentro del cuerpo principal del programa. Esta función procesa específicos mensajes y pasa cualquier mensaje no procesable de regreso a la librería de screensaver (Espichan, 2002).

La segunda función requerida en un módulo de screensaver es una pantalla de Configuración. Esta función debe mostrar una caja de diálogo que permite al usuario configurar el screensaver. Windows muestra la caja de diálogo de configuración cuando el usuario selecciona el botón configurar 6 en la caja de diálogo de screensaver de la Pantalla de Panel de Control (Espichan, 2002).

La tercera función requerida en un módulo de screensaver es guardar las especificaciones del cuadro de diálogo en el Registry de Windows. Esta función debe ser llamada por la aplicación de screensaver. Sin embargo las aplicaciones que no requieren ventanas especiales o controles personalizados en la caja de diálogo de configuración pueden simplemente retornar TRUE. Las aplicaciones que requieren ventanas especiales o controles personalizados deberían usar esta función para registrarlas en el Windows (Espichan, 2002).

En adición a crear un módulo que soporta esas tres funciones, un screensaver debería suplir un ícono. Este ícono es visible solo cuando el screensaver esté corriendo como una aplicación monousuario. El ícono debe ser identificado en el archivo de recursos del screensaver (Espichan, 2002).

La importancia de un screensaver es que sirve como herramienta de marketing y publicidad para una empresa. Es un elemento puramente decorativo y puede convertirse en una manera de difundir su imagen y hacerse más cercana a sus clientes (Espichan, 2002).

También son muy importantes los conceptos y teorías más comunes de la física clásica para la simulación de escenarios en tiempo real, tal como son la cinética y la cinemática de cuerpos sólidos; y la generación de partículas. Se presta atención también a la matemática implicada en dar solución a la detección de colisiones de objetos, así como conceptos tales como matrices, cuaterniones y vectores necesarios para calcular las rotaciones, traslaciones, fuerzas y otros elementos requeridos en la simulación o creación de un screensaver e incluso videojuegos (Arteaga, 2012).

También hay que tener presente que la física de los videojuegos a menudo, y a diferencia de otro tipo de simulaciones, no requiere una gran precisión; Es más, no es conveniente buscar resultados precisos, ya que esto demandaría que se consuman muchos recursos de procesamiento. Lo que sí se requiere es velocidad en el proceso, por lo que muchas veces se simplificarán los escenarios o se usarán supuestos para que la simulación sea más dinámica (Arteaga, 2012).

# Cuerpo

La biblioteca que utilizamos para graficar y generar todo los elementos que incluye nuestro screensaver se llama Simple DirectMedia Layer (SDL). Es una biblioteca de desarrollo multiplataforma que brinda acceso de bajo nivel a hardware de audio, teclado, mouse, joystick y gráficos a través de OpenGL y Direct3D. Se utiliza por software de reproducción de video, emuladores y juegos populares. Además, admite oficialmente Windows, macOS, Linux, iOS y Android. Cabe mencionar que está escrito en C, funciona de forma nativa con C++ y hay enlaces disponibles para otros lenguajes, incluidos C# y Python (SDL, 2022).

Otro recurso de mucha ayuda fue un artículo de Microsoft en donde explica con mayor detalle el uso de OpenMP para C++. Temas como programación en paralelo, paralelización y vectorización automáticas. simultaneidad, multithreading, etc. Sirvió de apoyo para comprender conceptos y de pronto buscar algo que se acoplara o fuera aplicable a nuestro proyecto (Microsoft, 2022).

Por último, se buscó en varias páginas web, ejemplos de screensavers para tomar nuestra decisión final sobre el diseño de screensaver que queríamos hacer. La principal fue una llamada Screensavers Planet donde muestra una cantidad inmensa de ideas que otros usuarios han desarrollado, hay muy buenos ejemplos y nos sirvió para tomar ideas y construir algo nuevo (Screensavers Planet, 2023).

A continuación, se describe la paralelización de nuestro programa:

```
#pragma omp parallel for num_threads(N_threads) shared(stars)
    for (int i = 0; i < MAX_NUM_STARS; ++i)
    {
        Star star;
        star.x = std::rand() % SCREEN_WIDTH;
        star.y = std::rand() % SCREEN_HEIGHT;
        star.speed = std::rand() % MAX_SPEED + 1;
#pragma omp critical
        stars.push_back(star);
    }
```

Motivo:
Para paralelizar el código se utilizó la directiva pragma omp parallel for, que divide el bucle for entre los hilos en paralelo. También se ha añadido una sección crítica con la directiva pragma omp critical para garantizar que el acceso al vector stars sea seguro y no cause condiciones de carrera entre los hilos. Además, compartimos la variable stars.

```
#pragma omp parallel for num_threads(N_threads) shared(comets) // Crear las
    for (int i = 0; i < num_comets; ++i)
    {
        Comet comet;
        comet.x = std::rand() % (SCREEN_WIDTH - COMET_SIZE);
        comet.y = std::rand() % (SCREEN_HEIGHT - COMET_SIZE);
        comet.vx = std::rand() % (MIN_SPEED_COMETS + 1) + MIN_SPEED_COMETS;
        if (std::rand() % 2 == 0)
        {
            comet.vx = -comet.vx;
        }
        comet.vy = std::rand() % (MIN_SPEED_COMETS + 1) + MIN_SPEED_COMETS;
        if (std::rand() % 2 == 0)
        {
            comet.vy = -comet.vy;
        }       You, 17 hours ago • Paralela Done
#pragma omp critical
        comets.push_back(comet);
    }
```

Motivo:
Para paralelizar el código se utilizó la directiva pragma omp parallel for, que divide el bucle for entre los hilos en paralelo. También se ha añadido una sección crítica con la directiva pragma omp critical para garantizar que el acceso al vector comets sea seguro y no cause condiciones de carrera entre los hilos. Además, se comparte la variable comets.

```
#pragma omp parallel for num_threads(N_threads) private(angle)
        for (int i = 0; i < num_comets; ++i)
        {
            Comet comet;
            angle = std::rand() % 360;
            angle = angle * 3.14159265 / 180;
            comet.x = std::rand() % (SCREEN_WIDTH - COMET_SIZE);
            comet.y = 0;
            comet.vx = std::rand() % (MAX_SPEED_COMETS - MIN_SPEED_COMETS + 1) + MIN_SPEED_COMETS;
            comet.vy = std::rand() % (MAX_SPEED_COMETS - MIN_SPEED_COMETS + 1) + MIN_SPEED_COMETS;
            if (std::rand() % 2 == 0)
            {
                comet.vx = comet.vy * tan(angle);
            }
#pragma omp critical
            {
                comets.push_back(comet);
            }
        }
```

Motivo:

Para paralelizar el código se utilizó la directiva pragma y especificar el número de hilos (num_threads) que deseamos utilizar. La idea es dividir el trabajo de crear cada cometa entre los hilos, de manera que cada hilo se encargue de crear un subconjunto de cometas y luego se unan los resultados. Además, como en este caso hay una dependencia de datos en la variable angle, podemos utilizar la cláusula private para crear una copia privada de angle para cada hilo. La directiva pragma parallel for permite la ejecución paralela del bucle for, mientras que la cláusula num_threads especifica el número de hilos que se utilizarán para la ejecución. Además, la cláusula private se utiliza para crear una copia privada de angle para cada hilo. Por último, la cláusula critical se utiliza para evitar que dos hilos agreguen elementos a comets al mismo tiempo y provoquen una condición de carrera.

# Conclusiones

- Durante el transcurso del proyecto, se logró demostrar las mejoras de FPS del Screen Saver utilizando los paradigmas de OpenMP, evidenciando la importancia y/o cómo utilizar estas paradigmas permiten al código ser más eficiente al momento de querer explotar las velocidad de la máquina. Cabe mencionar que a pesar que es posible paralelizar cualquier función o fragmento de código, no siempre se tendrá un buen resultado, ya que se debe de considerar otros factores como race condition.
- OpenMP nos permite paralelizar y mejorar la eficiencia de un código, sin embargo, si no se sabe utilizar los paradigmas de la forma correcta, esto podría provocar un resultado distinto, como un loop infinito o un programa mucho más lento que de manera secuencial.
- Se logró implementar un screensaver visualmente atractivo con ayuda de la biblioteca SDL juntando dos ideas preestablecidas para crear un screensaver distinto.

# Recomendaciones

- Se recomienda realizar este proyecto en un entorno ya preparado, ya que la idea es aprovechar los recursos de la máquina, por ende, desarrollarlo en una máquina virtual pierde el sentido de esto.
- Si se desea implementar OpenMP y SDL2 en un proyecto desarrollado en Windows se recomienda utilizar Msys2, ya que facilita la integración de estos juntos con WIndows.
- También se recomienda profundizar o implementar otras técnicas de paralelización, ya sea a nivel de software o hardware para obtener mejores resultados en desempeño y eficiencia del proyecto.

# Apéndice

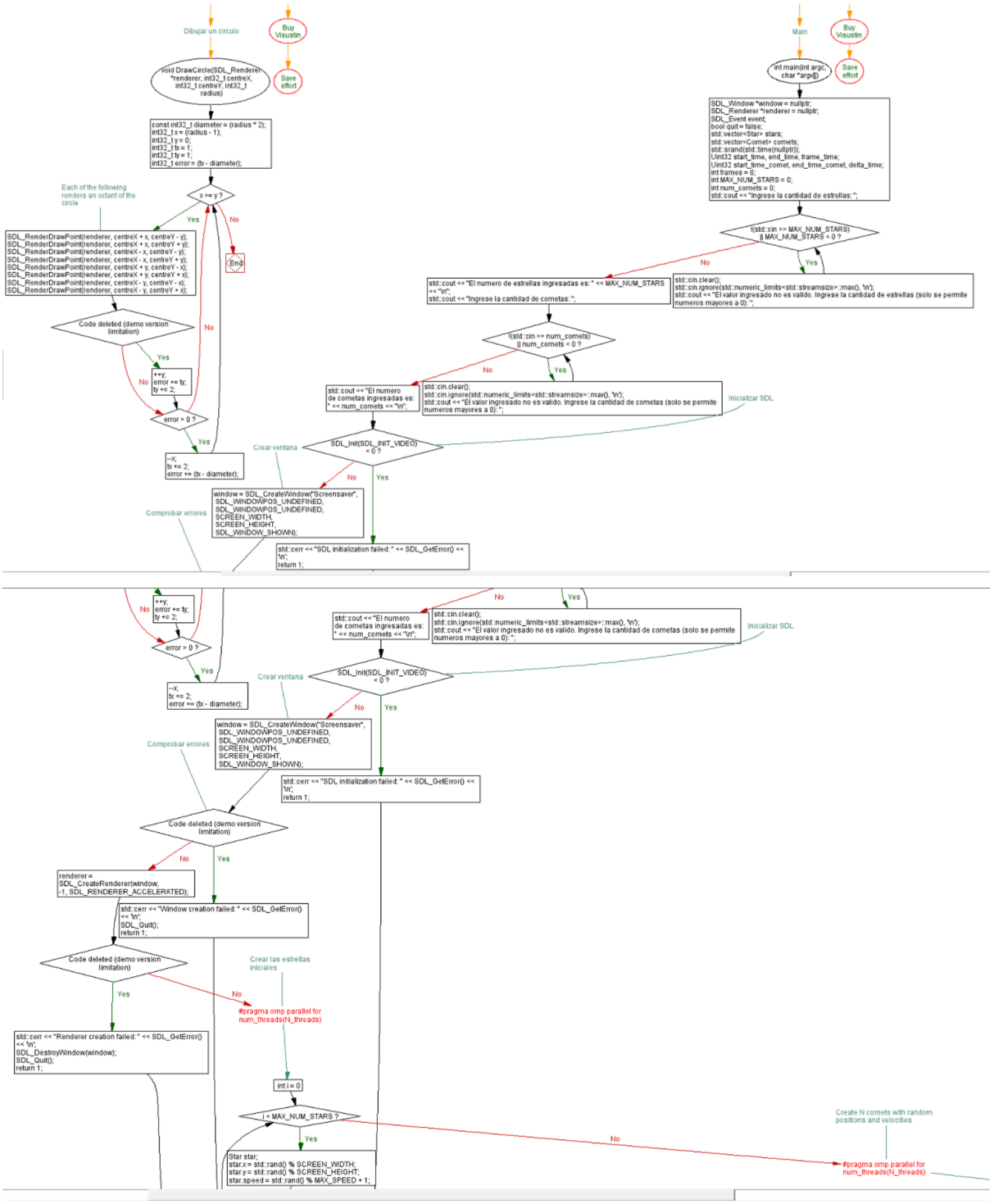
El archivo *comets.cpp* contiene la implementación del movimiento, físicas como colisiones y renderización del objeto cometa con sus atributos de posición y velocidad.

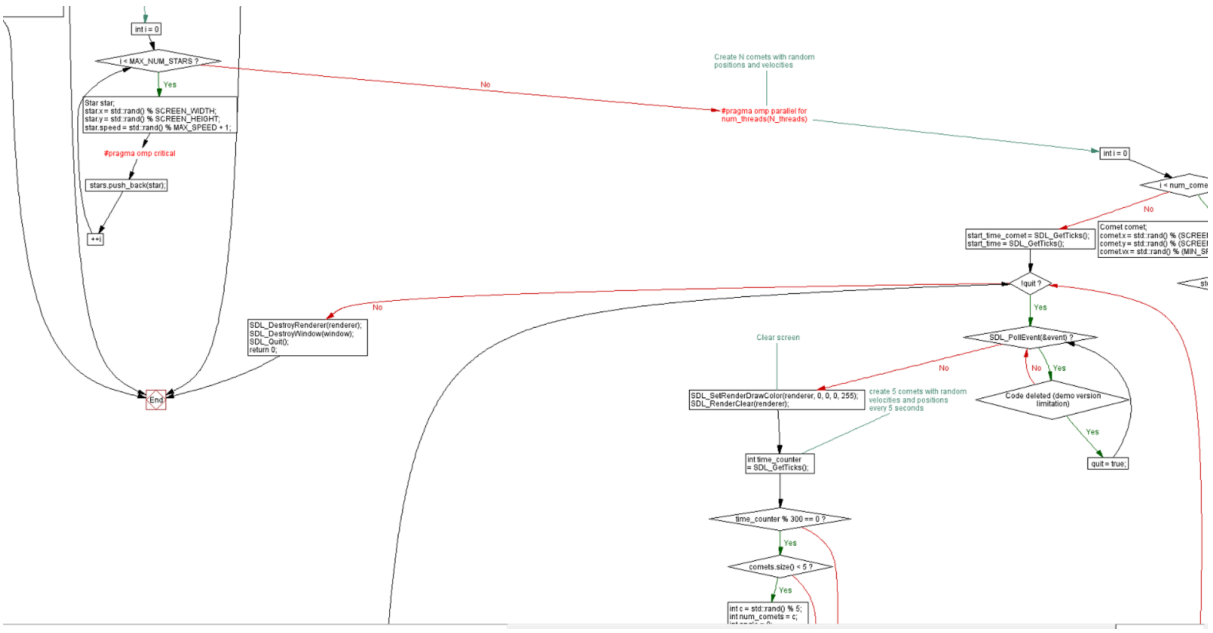
El archivo *main.cpp* es el archivo principal del proyecto en forma secuencial, incluye la implementación del fondo dinámico de estrellas y el objeto estrella con atributos de color, posición y velocidad y métodos de movimiento. Al usuario le pide que ingrese la cantidad de estrellas y cometas a generar, finalmente despliega el screensaver con el número de FPS.

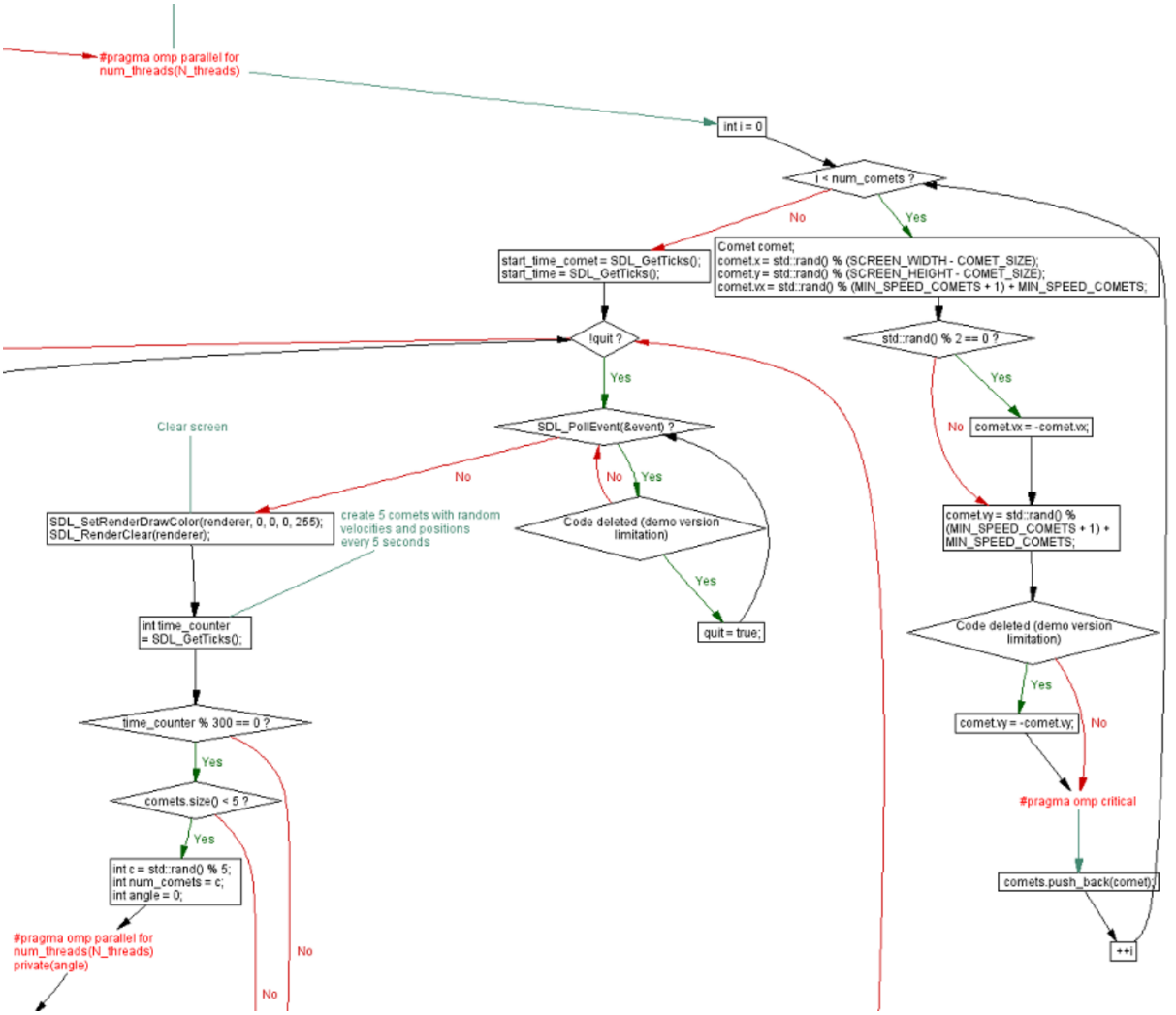
Por último, el archivo *paralela.cpp* tiene la misma funcionalidad del main pero con la aplicación de OpenMP para optimizar las tareas con la implementación de paralelismo utilizando threads y directivas propias.

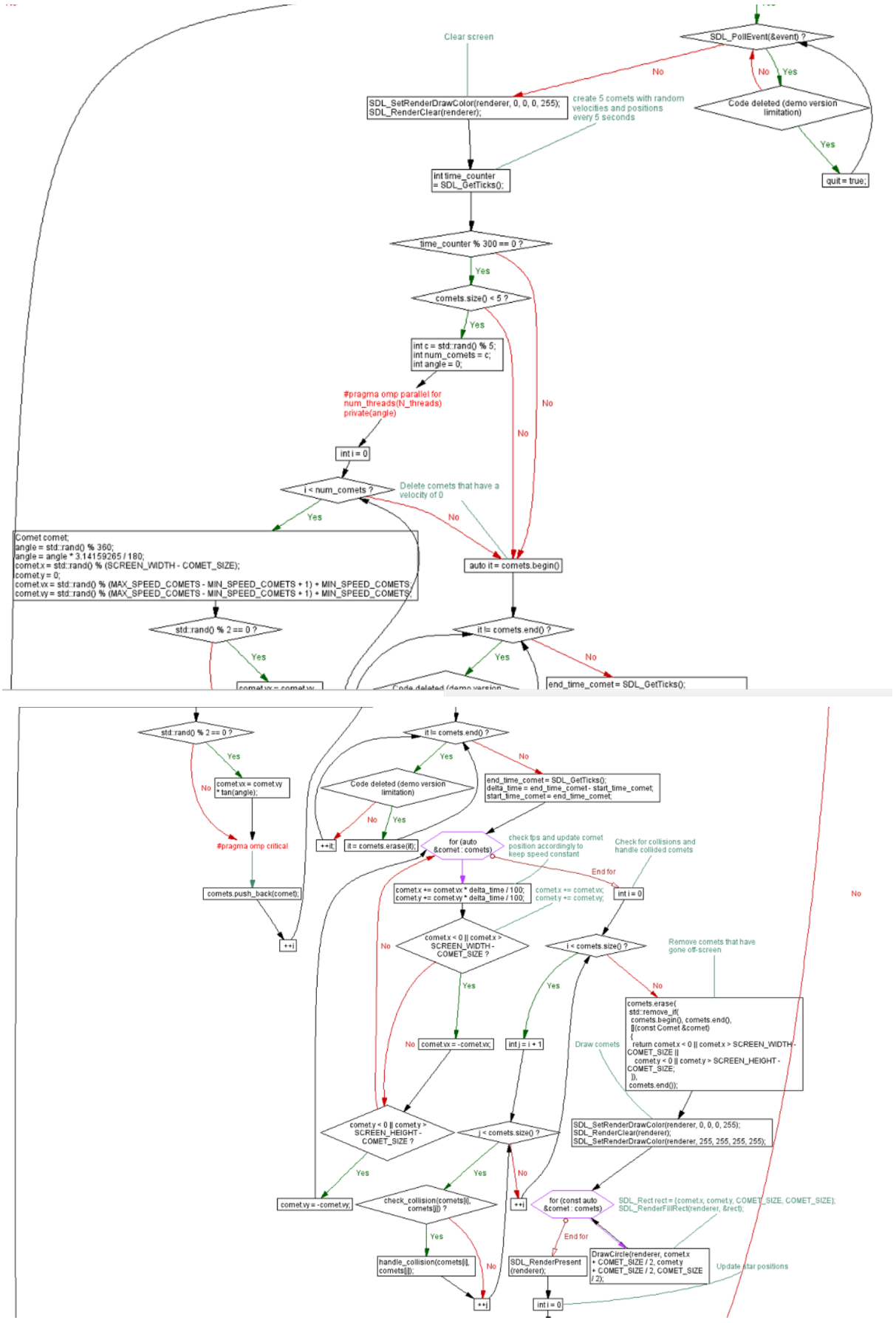
# Anexo 1

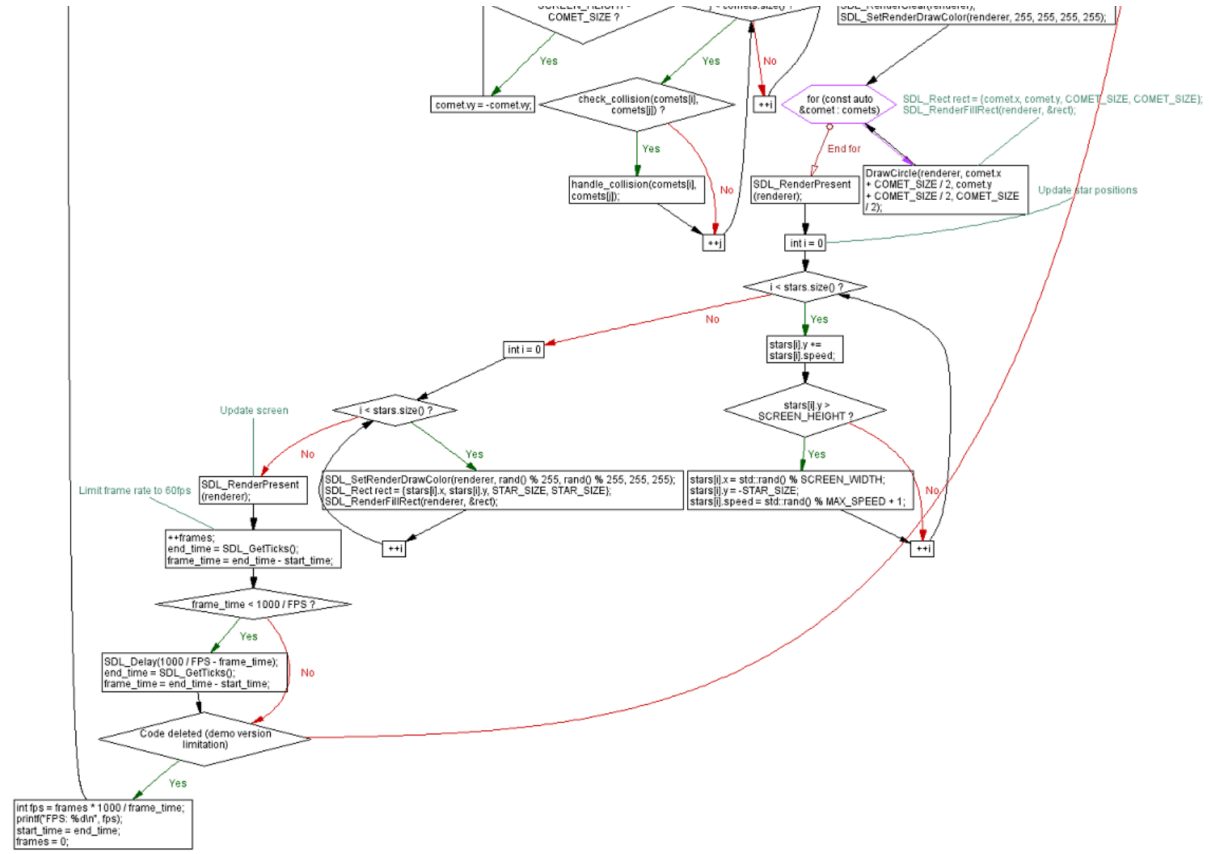
## Diagrama de flujo de programa paralelo

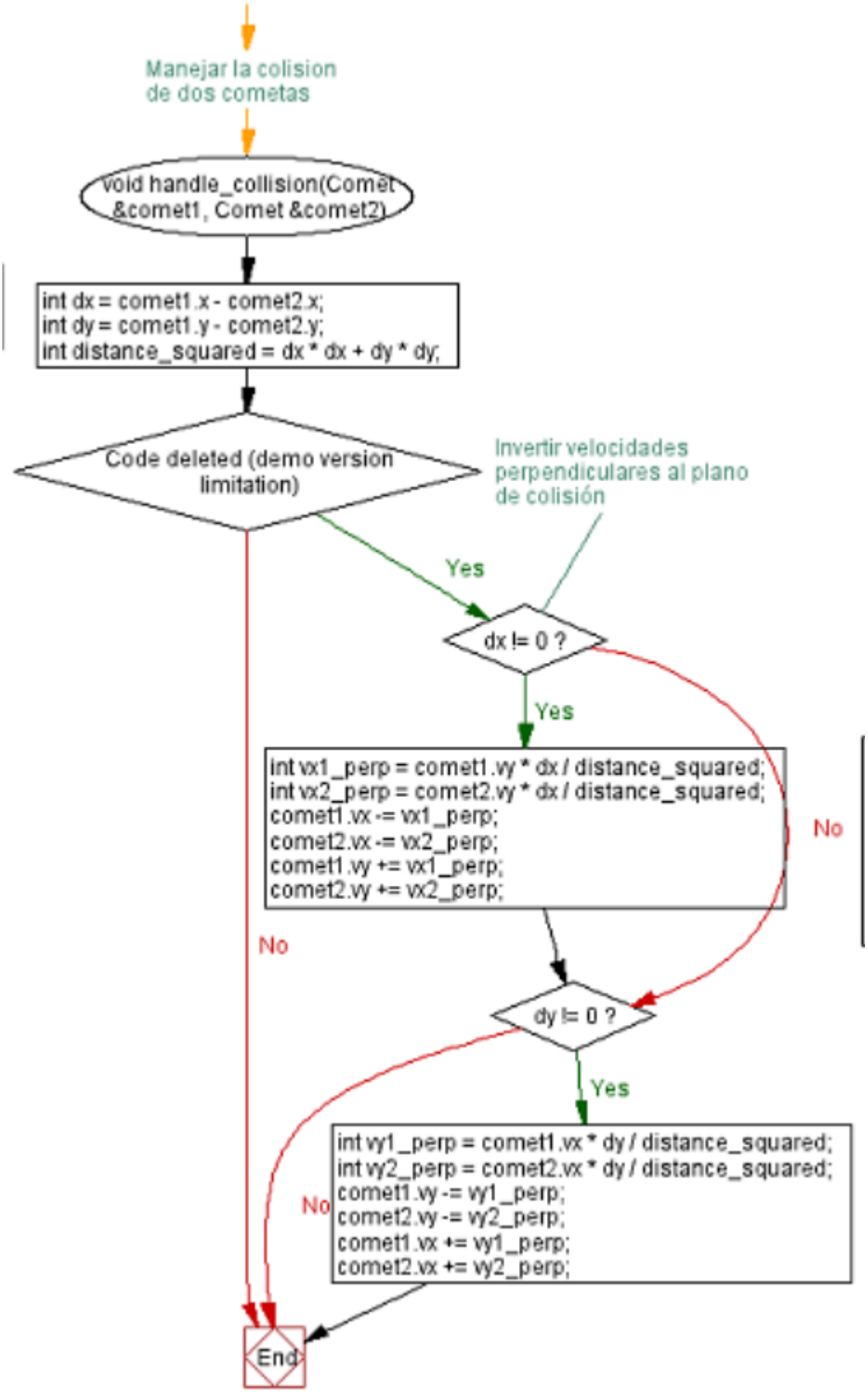


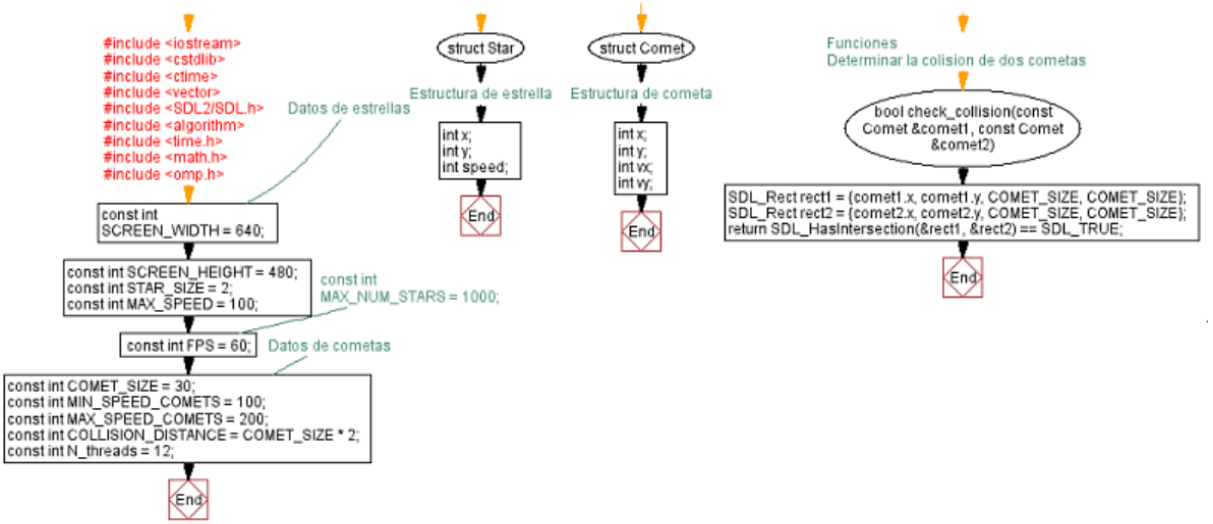

int i = 0
i < MAX_NUM_STARS ?
Yes
No
Star star;
star.x = std::rand() % SCREEN_WIDTH;
star.y = std::rand() % SCREEN_HEIGHT;
star.speed = std::rand() % MAX_SPEED + 1;
#pragma omp critical
stars.push_back(star);
++i
End
Create N comets with random positions and velocities
#pragma omp parallel for num_threads(N_threads)
int i = 0
start_time_comet = SDL_GetTicks();
start_time = SDL_GetTicks();
Comet comet;
!quit ?
SDL_DestroyRenderer(renderer);
SDL_DestroyWindow(window);
SDL_Quit();
return 0;
Clear screen
SDL_PollEvent(&event) ?
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
SDL_RenderClear(renderer);
create 5 comets with random velocities and positions every 5 seconds
Code deleted (demo version limitation)
quit = true;
int time_counter = SDL_GetTicks();
time_counter % 300 == 0 ?
comets.size() < 5 ?
int c = std::rand() % 5;
int num_comets = c;


#pragma omp parallel for num_threads(N_threads)
int i = 0
i < num_comets ?
No
Yes
start_time_comet = SDL_GetTicks();
start_time = SDL_GetTicks();
Comet comet;
comet.x = std::rand() % (SCREEN_WIDTH - COMET_SIZE);
comet.y = std::rand() % (SCREEN_HEIGHT - COMET_SIZE);
comet.vx = std::rand() % (MIN_SPEED_COMETS + 1) + MIN_SPEED_COMETS;
!quit ?
std::rand() % 2 == 0 ?
comet.vx = -comet.vx;
Clear screen
SDL_PollEvent(&event) ?
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
SDL_RenderClear(renderer);
create 5 comets with random velocities and positions every 5 seconds
Code deleted (demo version limitation)
quit = true;
comet.vy = std::rand() % (MIN_SPEED_COMETS + 1) + MIN_SPEED_COMETS;
Code deleted (demo version limitation)
comet.vy = -comet.vy;
#pragma omp critical
comets.push_back(comet);
++i
int time_counter = SDL_GetTicks();
time_counter % 300 == 0 ?
comets.size() < 5 ?
int c = std::rand() % 5;
int num_comets = c;
int angle = 0;
#pragma omp parallel for num_threads(N_threads) private(angle)


Clear screen
SDL_PollEvent(&event) ?
No
Yes
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
SDL_RenderClear(renderer);
create 5 comets with random velocities and positions every 5 seconds
Code deleted (demo version limitation)
quit = true;
int time_counter = SDL_GetTicks();
time_counter % 300 == 0 ?
comets.size() < 5 ?
int c = std::rand() % 5;
int num_comets = c;
int angle = 0;
#pragma omp parallel for num_threads(N_threads) private(angle)
int i = 0
i < num_comets ?
Delete comets that have a velocity of 0
Comet comet;
angle = std::rand() % 360;
angle = angle * 3.14159265 / 180;
comet.x = std::rand() % (SCREEN_WIDTH - COMET_SIZE);
comet.y = 0;
comet.vx = std::rand() % (MAX_SPEED_COMETS - MIN_SPEED_COMETS + 1) + MIN_SPEED_COMETS;
comet.vy = std::rand() % (MAX_SPEED_COMETS - MIN_SPEED_COMETS + 1) + MIN_SPEED_COMETS;
std::rand() % 2 == 0 ?
auto it = comets.begin()
it != comets.end() ?
end_time_comet = SDL_GetTicks();
#pragma omp critical
comets.push_back(comet);
++i
comet.vx = comet.vy * tan(angle);
++it
it = comets.erase(it);
end_time_comet = SDL_GetTicks();
delta_time = end_time_comet - start_time_comet;
start_time_comet = end_time_comet;
for (auto &comet : comets)
check fps and update comet position accordingly to keep speed constant
Check for collisions and handle collided comets
End for
comet.x += comet.vx * delta_time / 100;
comet.y += comet.vy * delta_time / 100;
comet.x += comet.vx;
comet.y += comet.vy;
int i = 0
comet.x < 0 || comet.x > SCREEN_WIDTH - COMET_SIZE ?
i < comets.size() ?
Remove comets that have gone off-screen
comets.erase(
std::remove_if(
comets.begin(), comets.end(),
[](const Comet &comet)
{
return comet.x < 0 || comet.x > SCREEN_WIDTH - COMET_SIZE ||
comet.y < 0 || comet.y > SCREEN_HEIGHT - COMET_SIZE;
}),
comets.end());
comet.vx = -comet.vx;
int j = i + 1
Draw comets
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
SDL_RenderClear(renderer);
SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
comet.y < 0 || comet.y > SCREEN_HEIGHT - COMET_SIZE ?
j < comets.size() ?
comet.vy = -comet.vy;
check_collision(comets[i], comets[j]) ?
++j
for (const auto &comet : comets)
SDL_Rect rect = {comet.x, comet.y, COMET_SIZE, COMET_SIZE};
SDL_RenderFillRect(renderer, &rect);
End for
DrawCircle(renderer, comet.x + COMET_SIZE / 2, comet.y + COMET_SIZE / 2, COMET_SIZE / 2);
SDL_RenderPresent(renderer);
Update star positions
handle_collision(comets[i], comets[j]);
int i = 0


COMET_SIZE ?
Yes
Yes
No
SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
comet.vy = -comet.vy;
check_collision(comets[i], comets[j]) ?
++i
for (const auto &comet : comets)
SDL_Rect rect = {comet.x, comet.y, COMET_SIZE, COMET_SIZE};
SDL_RenderFillRect(renderer, &rect);
Yes
End for
handle_collision(comets[i], comets[j]);
No
SDL_RenderPresent (renderer);
DrawCircle(renderer, comet.x + COMET_SIZE / 2, comet.y + COMET_SIZE / 2, COMET_SIZE / 2);
Update star positions
++i
int i = 0
i < stars.size() ?
No
Yes
int i = 0
stars[i].y += stars[i].speed;
Update screen
i < stars.size() ?
stars[i].y > SCREEN_HEIGHT ?
No
Yes
Yes
Limit frame rate to 60fps
SDL_RenderPresent (renderer);
SDL_SetRenderDrawColor(renderer, rand() % 255, rand() % 255, 255, 255);
SDL_Rect rect = {stars[i].x, stars[i].y, STAR_SIZE, STAR_SIZE};
SDL_RenderFillRect(renderer, &rect);
stars[i].x = std::rand() % SCREEN_WIDTH;
stars[i].y = -STAR_SIZE;
stars[i].speed = std::rand() % MAX_SPEED + 1;
No
++i
++i
++frames;
end_time = SDL_GetTicks();
frame_time = end_time - start_time;
frame_time < 1000 / FPS ?
Yes
SDL_Delay(1000 / FPS - frame_time);
end_time = SDL_GetTicks();
frame_time = end_time - start_time;
No
Code deleted (demo version limitation)
Yes
int fps = frames * 1000 / frame_time;
printf("FPS: %d\n", fps);
start_time = end_time;
frames = 0;


Manejar la colision
de dos cometas
void handle_collision(Comet
&comet1, Comet &comet2)
int dx = comet1.x - comet2.x;
int dy = comet1.y - comet2.y;
int distance_squared = dx * dx + dy * dy;
Code deleted (demo version
limitation)
Invertir velocidades
perpendiculares al plano
de colisión
Yes
dx != 0 ?
Yes
int vx1_perp = comet1.vy * dx / distance_squared;
int vx2_perp = comet2.vy * dx / distance_squared;
comet1.vx -= vx1_perp;
comet2.vx -= vx2_perp;
comet1.vy += vx1_perp;
comet2.vy += vx2_perp;
No
No
dy != 0 ?
Yes
int vy1_perp = comet1.vx * dy / distance_squared;
int vy2_perp = comet2.vx * dy / distance_squared;
comet1.vy -= vy1_perp;
comet2.vy -= vy2_perp;
comet1.vx += vy1_perp;
comet2.vx += vy2_perp;
No
End


#include <iostream>
#include <cstdlib>
#include <ctime>
#include <vector>
#include <SDL2/SDL.h>
#include <algorithm>
#include <time.h>
#include <math.h>
#include <omp.h>
Datos de estrellas
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
const int STAR_SIZE = 2;
const int MAX_SPEED = 100;
const int MAX_NUM_STARS = 1000;
const int FPS = 60;
Datos de cometas
const int COMET_SIZE = 30;
const int MIN_SPEED_COMETS = 100;
const int MAX_SPEED_COMETS = 200;
const int COLLISION_DISTANCE = COMET_SIZE * 2;
const int N_threads = 12;
End
struct Star
Estructura de estrella
int x;
int y;
int speed;
End
struct Comet
Estructura de cometa
int x;
int y;
int vx;
int vy;
End
Funciones
Determinar la colision de dos cometas
bool check_collision(const Comet &comet1, const Comet &comet2)
SDL_Rect rect1 = {comet1.x, comet1.y, COMET_SIZE, COMET_SIZE};
SDL_Rect rect2 = {comet2.x, comet2.y, COMET_SIZE, COMET_SIZE};
return SDL_HasIntersection(&rect1, &rect2) == SDL_TRUE;
End

# Anexo 2

Catálogo de funciones

- check_collision(const Comet &comet1, const Comet &comet2): verifica si hay colisión entre dos cometas.
- handle_collision(Comet &comet1, Comet &comet2): maneja la colisión entre dos cometas.
- DrawCircle(SDL_Renderer *renderer, int32_t centreX, int32_t centreY, int32_t radius): dibuja un círculo en la posición especificada.
- main(int argc, char *argv[]): función principal del programa, que maneja la creación y la ejecución de la ventana de SDL, el dibujo de las estrellas y los cometas, y la lógica de su movimiento y colisión.
- SDL_Window *window = nullptr: Puntero a la ventana creada.
- SDL_Renderer *renderer = nullptr: Puntero al renderizador creado.
- SDL_Event event: Estructura que contiene los eventos generados por el usuario.

# Anexo 3

Cálculo de SpeedUp y Eficiencia

Estrellas: 2,000
Cometas: 100
Threads: 8

| Secuencial | Paralelo |
|---|---|
| 8.655 | 5.474 |
| 3.213 | 3.253 |
| 4.419 | 2.477 |
| 4.209 | 2.161 |
| 2.97 | 2.334 |
| 4.6932 | 3.1398 |
| 2.625 | 3.332 |
| 2.997 | 2.04 |
| 2.619 | 1.718 |
| 2.43 | 1.709 |
| 3.88302 | 2.76378 |

| | |
|---|---|
| SpeedUp | 1.40 |
| Eficiencia | 0.18 |

```
PS C:\Users\maqui\OneDrive\Escritorio\UVG\UVG S9\Computacion Paralela y Distribuida\Proyecto1_Paralela> .\paralela.exe
Ingrese la cantidad de estrellas: 2000
El numero de estrellas ingresadas es: 2000
Ingrese la cantidad de cometas: 100
El numero de cometas ingresadas es: 100
Tiempo secuencial: 1.718000
```

```
PS C:\Users\maqui\OneDrive\Escritorio\UVG\UVG S9\Computacion Paralela y Distribuida\Proyecto1_Paralela> .\main.exe
Ingrese la cantidad de estrellas: 2000
El numero de estrellas ingresadas es: 2000
Ingrese la cantidad de cometas: 100
El numero de cometas ingresadas es: 100
Tiempo secuencial: 4.209000
```

Enlace del GitHub: https://github.com/MarcoRamirezGT/Proyecto1_Paralela

# Referencias


Microsoft. (26 de Septiembre de 2022). *Microsoft*. Obtenido de OpenMP en Visual C++: https://learn.microsoft.com/es-es/cpp/parallel/openmp/openmp-in-visual-cpp?view=msvc-170

Screensavers Planet. (16 de Marzo de 2023). *Screensavers Planet*. Obtenido de Screensavers for Windows: https://www.screensaversplanet.com/screensavers/?windows=on&sort=date&order=desc

SDL. (22 de Noviembre de 2022). *Simple Directmedia Layer*. Obtenido de Simple Directmedia Layer: https://www.libsdl.org

Espichan Beretta, P. I. (2002). Aplicación MFC para Windows: creación de un protector de pantalla para la UNMSM.

Arteaga Alvarez, J. P. (2012). Motor de física para videojuegos.