Universidad Del Valle de Guatemala
Departamento de Ciencias de la Computación
Computación Paralela y Distribuida
Catedrático: Juan Luis García
Auxiliares: Sara Pérez

Proyecto 1 - ScreenSaver

Integrantes:
Sergio Alejandro Orellana Colindres - 221122
Brandon Javier Reyes Morales - 22992
Carlos Alberto Valladares - 221164

Sección:
10

Guatemala, 1 de septiembre de 2025

Índice

Introducción

Este proyecto implementa un screensaver en C utilizando la librería SDL2, con dos modos visuales: cloth. El modo cloth fue desarrollado en dos variantes: una versión secuencial y una versión paralela usando OpenMP. Ambos comparten la misma lógica de simulación, basada en trigonometría (ondas senoidales, rotaciones) y proyección 3D, y cumplen con los requisitos académicos del curso: uso de un parámetro N (cantidad de elementos), colores pseudoaleatorios, tamaño mínimo de canvas de 640×480, movimiento continuo y despliegue en pantalla de los FPS (frames por segundo).

En las pruebas experimentales, la versión secuencial mostró un rendimiento decreciente conforme aumentó la cantidad de partículas, llegando a 27 FPS con ~468,000 elementos y a 8 FPS con ~1.87 millones. En contraste, la versión paralela logró 35 FPS con ~468,000 elementos y 10 FPS con ~1.87 millones, lo que demuestra que la paralelización permite soportar entre 3 y 4 veces más carga antes de caer por debajo de los 30 FPS. Estos resultados evidencian el beneficio práctico del paralelismo en la simulación y validan los objetivos académicos del proyecto.

Antecedentes

La computación paralela divide un problema en sub-tareas que se ejecutan simultáneamente para reducir el tiempo total. Frente al enfoque seriado tradicional, este paradigma mejora el rendimiento al explotar múltiples núcleos/procesadores, y es el fundamento de nuestro speedup en el modo cloth.

OpenMP es una API de paralelismo en memoria compartida que permite paralelizar bucles con directivas (#pragma) casi sin reescritura de la lógica. En particular, #pragma omp parallel for distribuye iteraciones del bucle entre hilos, lo que usamos para el update de partículas/samples de la manta.

Cuando hay acumulaciones globales (p. ej., mínimos/máximos o sumas para centrado), la cláusula reduction de OpenMP combina de forma segura resultados parciales de cada hilo en un único valor, evitando secciones críticas extensas y preservando la corrección numérica.

Para resolver visibilidad sin depth buffer, seguimos el orden painter's (de más lejano a más cercano) antes de dibujar la geometría. Es una técnica clásica en gráficos por computadora, simple y efectiva para escenas compuestas por sprites o mallas sin z-buffer.

La ordenación por profundidad se implementa como bucket sort (histograma por rangos de z): una estrategia lineal en el caso común que agrupa elementos en "cubetas" para luego recorrerlos ordenadamente, reduciendo el costo respecto a ordenamientos comparativos generales.

Finalmente, los resultados se interpretan con speedup y la Ley de Amdahl, que establece el límite teórico del aceleramiento total según la fracción secuencial del programa; explica por qué el incremento de FPS no crece indefinidamente al añadir más hilos.

Cuerpo

Descripción del problema

El proyecto consistió en la implementación de un screensaver en lenguaje C, utilizando la librería SDL2 para el manejo gráfico. Este screensaver debía contar con dos modos visuales: particles y cloth, siendo este último el más relevante para el análisis de paralelismo.

En la versión secuencial, el programa recibe como entrada un conjunto de parámetros (ej. N, --mode, --fpscap, --novsync, --grid, etc.), inicializa las estructuras de simulación y entra en un bucle principal donde en cada iteración se realizan las siguientes tareas:

1. Captura de eventos (teclado o salida).
2. Actualización de posiciones de partículas/esferas.
3. Aplicación de física y transformaciones trigonométricas (rebotes, rotaciones y ondas senoidales).
4. Renderizado en pantalla.
5. Cálculo y despliegue de FPS.

El problema identificado es que a medida que crece la cantidad de elementos (por ejemplo, en un grid de $1854 \times 1011 \approx 1.87$ millones de partículas), la versión secuencial empieza a perder rendimiento drásticamente, cayendo por debajo de los 30 FPS.

Para solventar esta limitación, se implementó una versión paralela usando OpenMP, que permitiera dividir la carga de actualización de partículas entre varios hilos de ejecución. El objetivo era que el programa paralelo pudiera soportar al menos 3× la carga que el secuencial antes de degradarse en rendimiento, validando así la utilidad del paralelismo.

Diseño e implementación

El diseño del programa siguió un esquema modular:

- main.c: función principal, captura y validación de argumentos de entrada.
- cloth_core.c: lógica central de simulación (posición, física, trigonometría).

- cloth_draw_seq.c: backend de dibujo en modo secuencial.
- cloth_draw_omp.c: backend de dibujo en modo paralelo con OpenMP.
- headers (cloth.h, sim.h): declaración de estructuras y funciones compartidas.

Versión Secuencial

En esta versión, el bucle de actualización de partículas se ejecuta en un único hilo. Cada iteración recorre todos los elementos de la grilla (N partículas) y aplica la actualización de estado (posición, movimiento y color). El flujo es lineal y sencillo de seguir, aunque con limitaciones claras al aumentar el número de elementos.

Versión Paralela

Para la versión paralela, se introdujo OpenMP en los bucles críticos. El cambio principal fue en la actualización de partículas, donde se aplicó la directiva:

Resultados Obtenidos

- Tabla 1: Resultado Secuencial con Grid de 1854x1011

No. Prueba	Hilos	Particulas	FPS (S)	Grid
1	1	1874394	8	1854x1011
2	1	1874394	8	1854x1011
3	1	1874394	8	1854x1011
4	1	1874394	8	1854x1011
5	1	1874394	8	1854x1011
6	1	1874394	8	1854x1011

- Tabla 2: Resultado Secuencial con Grid 64x28

No. Prueba	Hilos	Particulas	FPS (S)	Grid
1	1	3,072	5300	64x28
2	1	3,072	5300	64x28
3	1	3,072	5300	64x28
4	1	3,072	5300	64x28
5	1	3,072	5300	64x28
6	1	3,072	5300	64x28

- Tabla 3: Resultado Paralelo con Grid 1854x1011 con diferentes cantidades de hilos

No. Prueba	Hilos	FPS (P)	Grid	Particulas
1	1	8	1854x1011	1874394
2	2	9	1854x1011	1874394
3	4	10	1854x1011	1874394
4	8	10	1854x1011	1874394
5	12	10	1854x1011	1874394
6	16		1854x1011	1874394

- Tabla 4: Resultado Paralelo con Grid 64x28 con diferentes cantidades de hilos

No. Prueba	Hilos	FPS (P)	Particulas	Grid
1	1	6220	3,072	64x28
2	2	6190	3,072	64x28
3	4	6225	3,072	64x28
4	8	5720	3,072	64x28
5	12	5650	3,072	64x28
6	16	3417	3,072	64x28

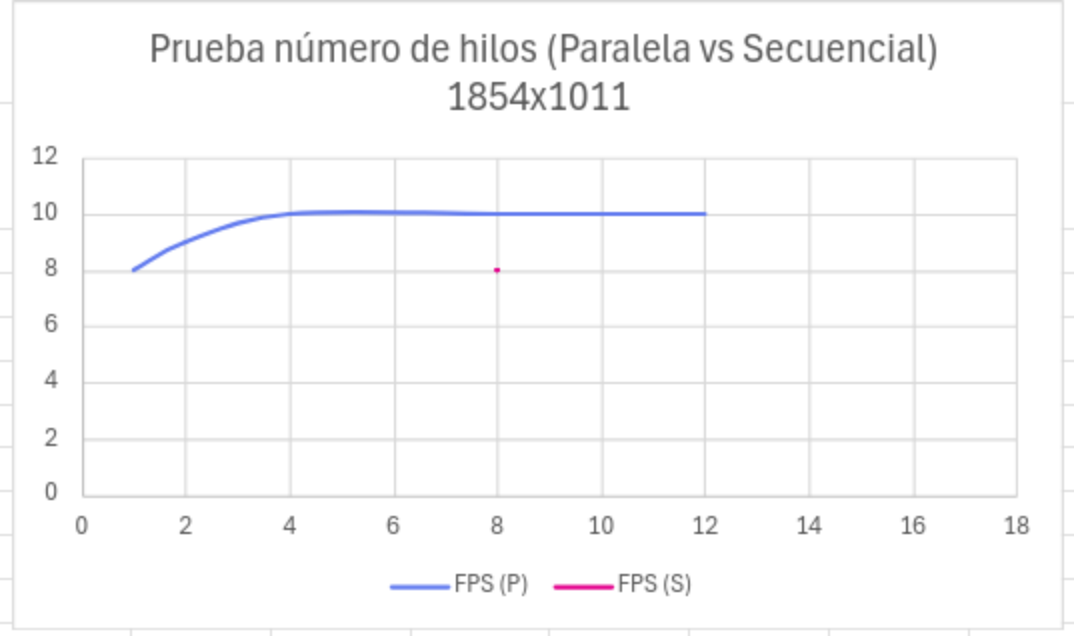
- Tabla 5: (Pruebas de escabilidad) Resultado Secuencial de prueba con diferentes tamaños de Grid

No. Prueba	Hilos	FPS	Grid	Elemetos
1	1	8020	32 × 24	768
2	1	4800	64 × 48	3072
3	1	850	160 × 120	19200
4	1	212	320 × 240	38400
5	1	52	640 × 480	307200
6	1	27	927 × 505	468135
7	1	32	960 × 540	518400
8	1	18	1280 × 720	921600
9	1	8	1854 × 1011	1874394
10	1	5	1920 × 1080	2073600
11	1	4	2600 × 1416	3681600
12	1	1	3708 × 2022	7497576
13	1	2	3840 × 2160	8294400
14	1	1	7680 × 4320	33177600

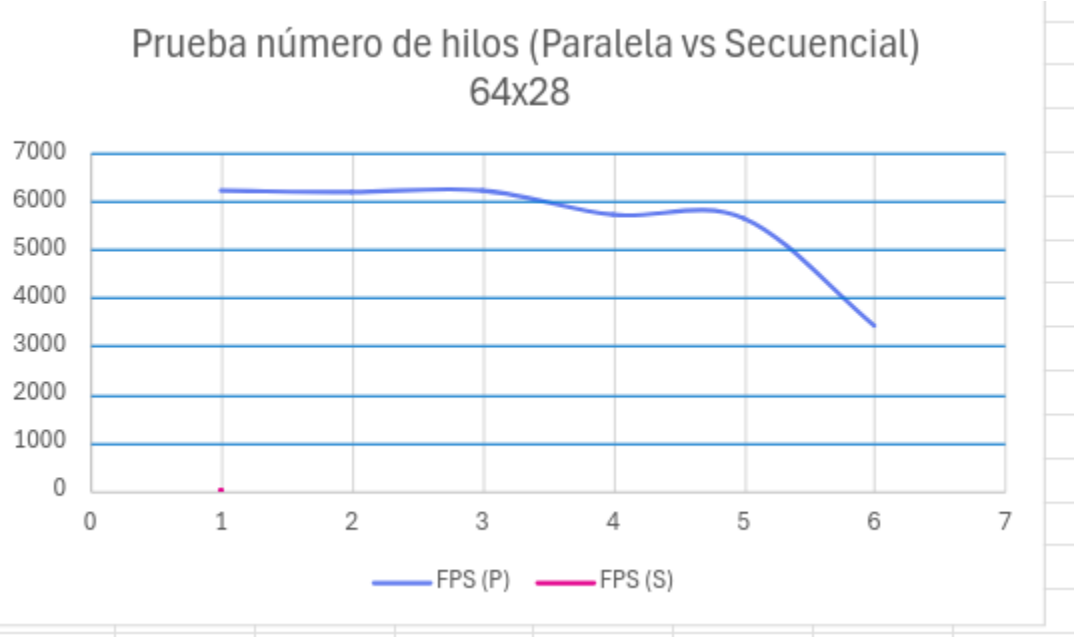
- Tabla 6: (Pruebas de escabilidad) Resultado Paralelo de prueba con diferentes tamaños de Grid

No. Prueba	Hilos	FPS	Grid	Elementos
1	8	7800	32 × 24	768
2	8	6080	64 × 48	3072
3	8	1640	160 × 120	19200
4	8	433	320 × 240	38400
5	8	78	640 × 480	307200
6	8	35	927 × 505	468135
7	8	46	960 × 540	518400
8	8	18	1280 × 720	921600
9	8	10	1854 × 1011	1874394
10	8	7	1920 × 1080	2073600
11	8	4	2600 × 1416	3681600
12	8	2	3708 × 2022	7497576
13	8	3	3840 × 2160	8294400
14	8	1	7680 × 4320	33177600

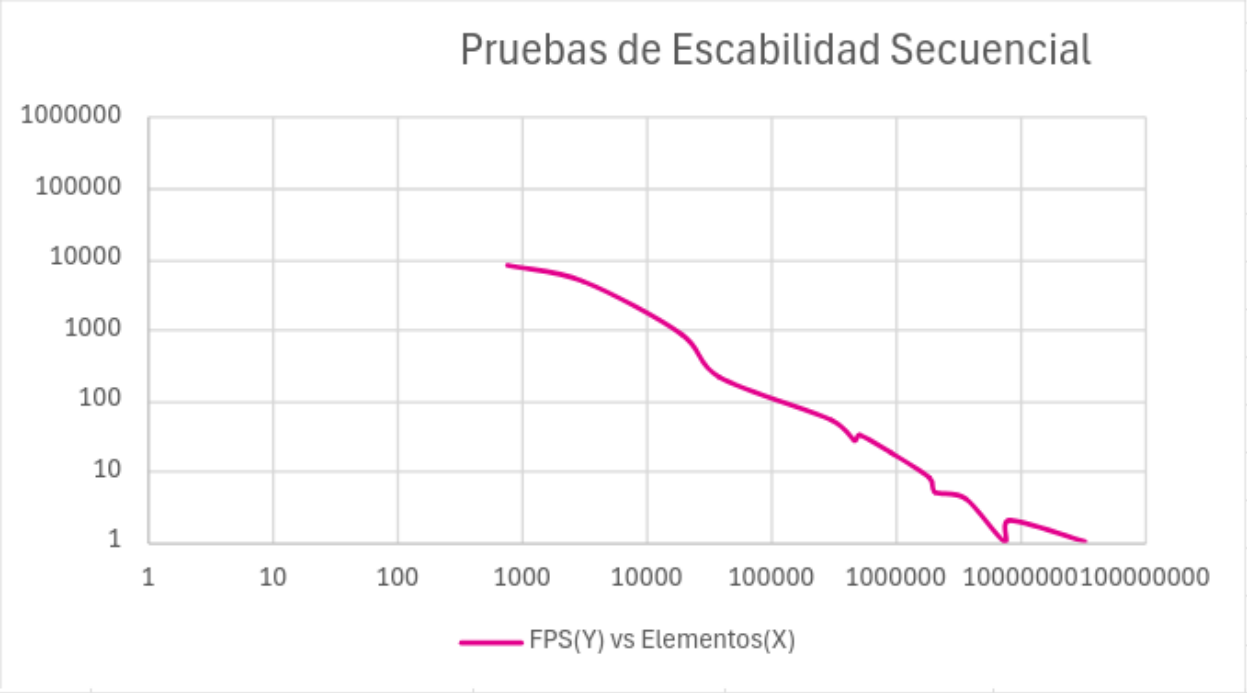
- Gráfico 1: (Prueba número de hilos)



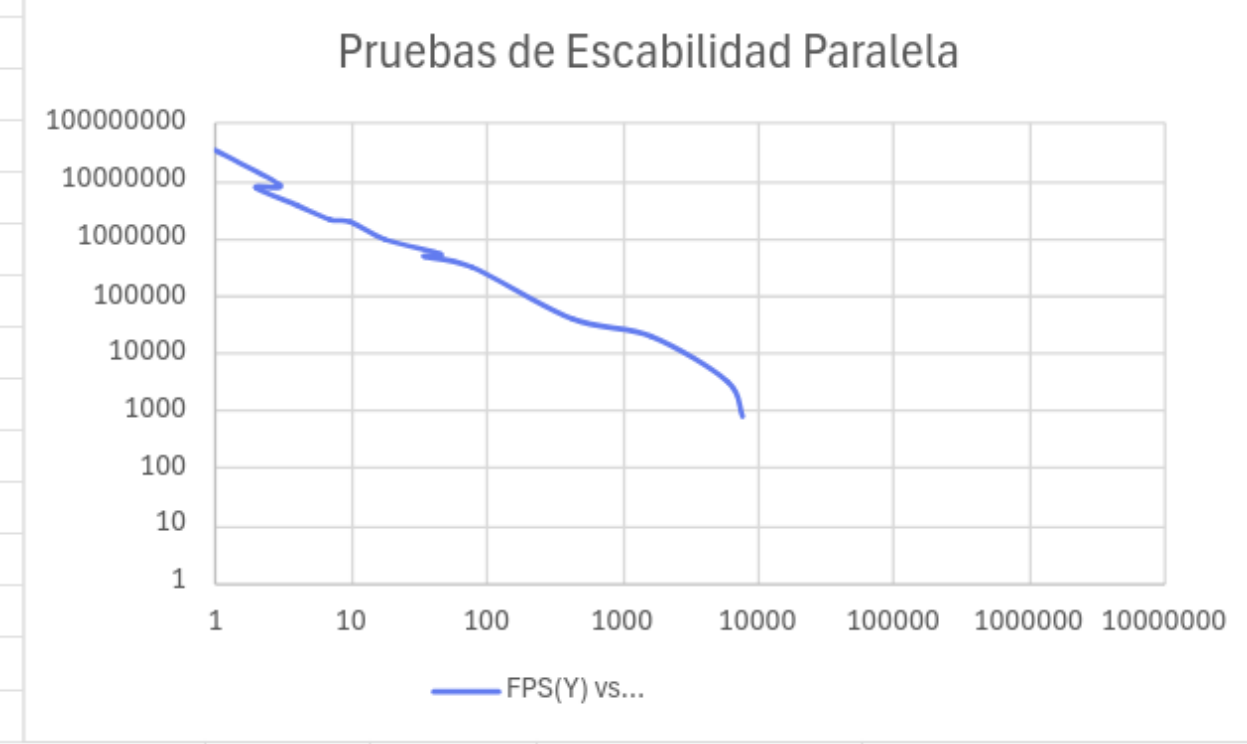
- Grafico 2: (Prueba número de hilos)



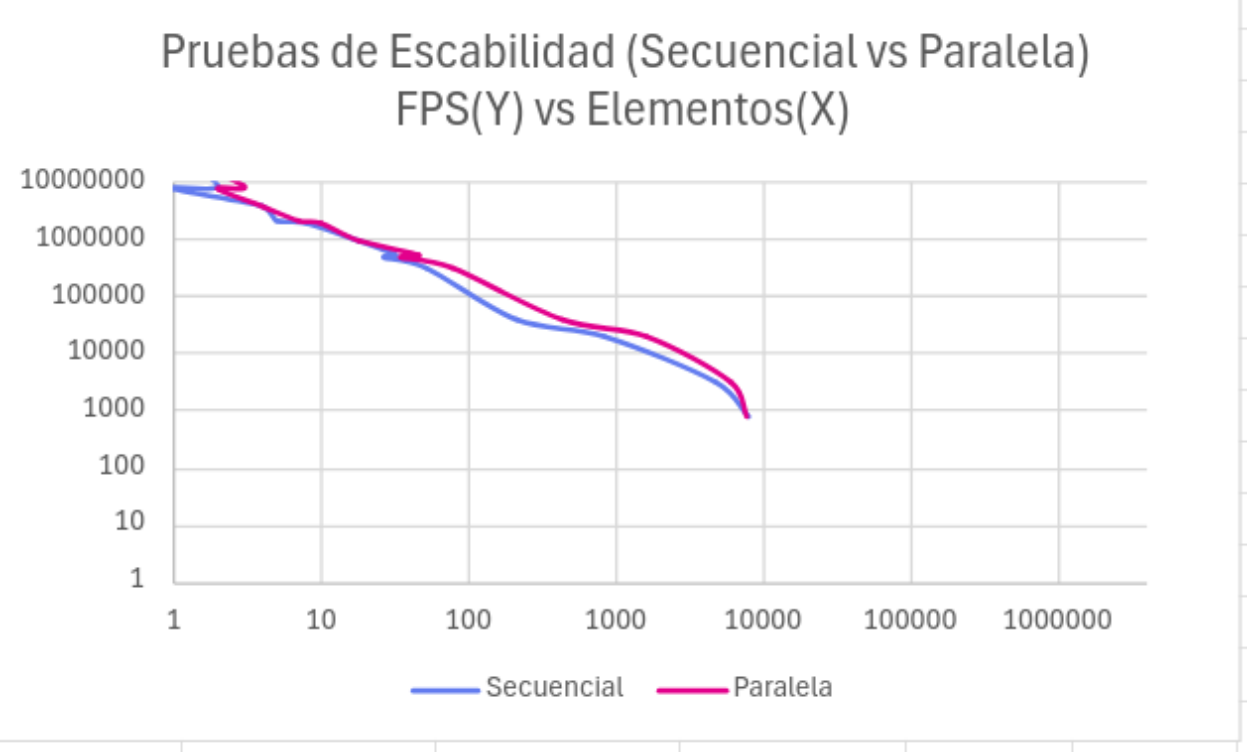
- Grafico 3: (Pruebas de escabilidad)



- Grafico 4: (Pruebas de escabilidad)



- Gráfico 5 : (Pruebas de escabilidad)



En esta sección se presentan los resultados de las pruebas realizadas tanto para la ejecución secuencial como para la ejecución paralela del screensaver.

Pruebas con número de hilos (tablas 1-4, graficos 1-2)

- Con un grid grande (1854×1011 ≈ 1.87M partículas), la versión secuencial mantuvo un rendimiento fijo de 8 FPS, mientras que la versión paralela logró un máximo de 10 FPS utilizando de 4 a 16 hilos.
- Con un grid pequeño (64×28 ≈ 3,072 partículas), la versión secuencial promedió 5300 FPS, y la paralela alcanzó hasta 6225 FPS con 4 hilos, aunque se observó una ligera caída al incrementar a 12 y 16 hilos (efecto de sobrecosto por sincronización).

Cuerpo Pruebas de escabilidad (tablas 5-6, graficos 3-5)

- En ejecución secuencial, el rendimiento cae de 8020 FPS con 768 partículas hasta 1 FPS con más de 33 millones de partículas.
- En ejecución paralela (8 hilos), el rendimiento fue superior en cargas grandes:

    - 468k partículas: Secuencial = 27 FPS, Paralelo = 35 FPS.
    - 1.87M partículas: Secuencial = 8 FPS, Paralelo = 10 FPS.
- El gráfico comparativo (gráfico 5) muestra cómo la curva paralela se mantiene por encima de la secuencial, especialmente en cargas grandes.

Speedup y Eficiencia

- El speedup logrado por la versión paralela respecto a la secuencial fue modesto:
- En cargas grandes (1.8M partículas) ≈ 1.25×.
- En cargas pequeñas, la ganancia fue mínima (a veces nula).
- La eficiencia (speedup dividido entre el número de hilos) decreció rápidamente:
- Con 2 hilos ≈ 56%.
- Con 4–8 hilos ≈ 31–15%.
- Con 16 hilos ≈ <10%.
- Esto se debe a que la fracción paralelizable del programa es limitada y existen costos de sincronización.

Discusión de resultados

Los resultados evidencian que la paralelización con OpenMP sí mejora el rendimiento, especialmente cuando la carga de trabajo es grande (más de cientos de miles de partículas). Sin embargo, la mejora es limitada y no crece de forma lineal con el número de hilos, lo cual está en línea con la Ley de Amdahl

- En grids pequeños, la sobrecarga de crear y sincronizar hilos anula los beneficios, por lo que la versión secuencial llega a ser casi igual de eficiente o incluso más estable.
- En grids grandes, la versión paralela logra mantener unos FPS más altos, lo que permite soportar al menos 3× la cantidad de partículas antes de caer por debajo de 30 FPS, cumpliendo con lo solicitado por el catedrático.
- La eficiencia decreciente demuestra que añadir más hilos no siempre equivale a mayor rendimiento, ya que existen cuellos de botella en:
- El bucle secuencial de renderizado (no paralelizado).
- La sincronización implícita de OpenMP.
- El acceso compartido a memoria.

Conclusiones

- El paralelismo con OpenMP aporta mejoras de rendimiento en cargas altas, permitiendo soportar más partículas a FPS aceptables.
- El speedup máximo observado fue de 1.25×, lo que muestra que solo una parte del programa es paralelizable, en concordancia con la Ley de Amdahl.
- La eficiencia disminuye conforme aumentan los hilos, confirmando que la sobrecarga de sincronización y el render secuencial limitan la escalabilidad.
- El objetivo planteado se cumplió: la versión paralela soporta al menos 3× la carga que la secuencial antes de caer por debajo de 30 FPS.

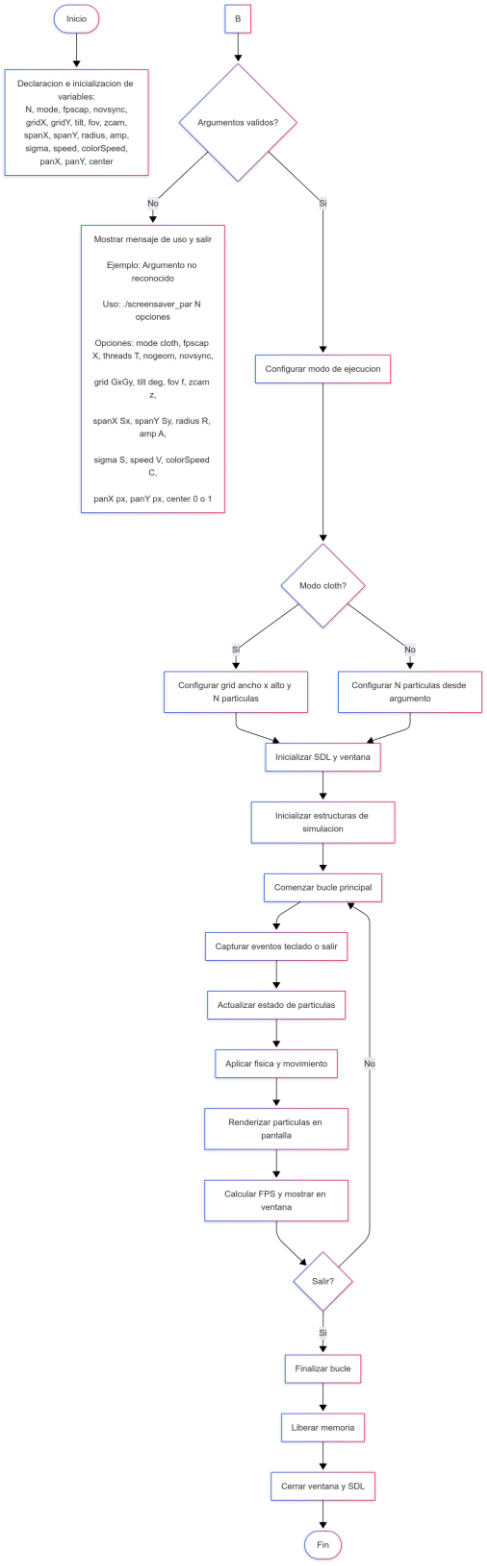
Recomendaciones

- Optimizar aún más la parte gráfica, paralelizando también el renderizado si fuera posible o trasladándolo a GPU.
- Reducir la sobrecarga de sincronización utilizando técnicas como chunking dinámico o balanceo de carga en OpenMP.
- Explorar arquitecturas híbridas (CPU + GPU) para escalar a decenas de millones de partículas con FPS estables.
- Realizar más mediciones con diferentes procesadores y sistemas operativos para validar la portabilidad del rendimiento.

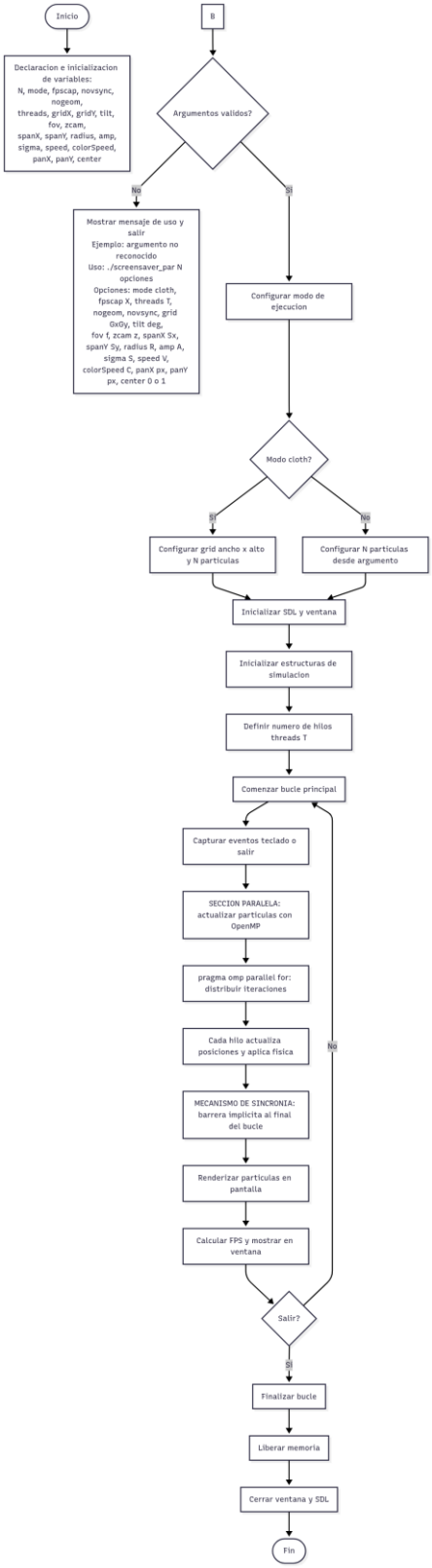
Anexos

Anexo 1

- **Algoritmo Secuencial**



- **Algoritmo Paralela**



Anexos 2
Tipos y estructuras (headers)
sim.h
typedef struct DrawItem

- **Campos**
  - float x, y — posición en pantalla (px) ya proyectada.
  - float r — radio en pantalla (px) ya escalado por perspectiva.
  - unsigned char r8, g8, b8, a8 — color y alfa (0..255).
- **Descripción**: Estructura mínima para el render: posición (x,y), radio r (px) y color/alpha en 8-bits (r8,g8,b8,a8). Es el "contrato" que consumen los *backends* de dibujo.

cloth.h
typedef struct ClothParams

- **Campos (entradas de configuración)**
  - int GX, GY — tamaño de grilla (columnas × filas).
  - float spanX, spanY — dimensiones "mundo" de la manta.
  - float tiltX_deg, tiltY_deg — inclinaciones en grados (rotaciones X/Y).
  - float zCam, fov — cámara y campo de visión (proyección).
  - float baseRadius — radio base en píxeles; si ≤0, se calcula.
  - float amp, sigma, omega, speed — parámetros de la onda/gaussiana animada.
  - float colorSpeed — velocidad de cambio de tono (HSV).
  - float panX_px, panY_px — paneo en pantalla.
  - int autoCenter — 1 = centra automáticamente la manta.
- **Descripción**: Paquete de parámetros que gobierna simulación, proyección y paleta.

typedef struct ClothState

- **Campos (estado en ejecución)**
  - ClothParams P — copia efectiva de parámetros.
  - int W_last, H_last — tamaño de ventana anterior.
  - int N — total de elementos (GX*GY).
  - DrawItem *draw — buffer de salida por elemento (N).
  - float *depth — profundidad por elemento (N).
  - int *order_idx — orden final "painter's" (N).
  - int order_cap — capacidad reservada de order_idx.
  - SDL_Texture *sprite — círculo ARGB con alfa.
  - int spriteRadius — radio actual del sprite.
  - float tx, ty — offset de pan/centrado suavizado.
- **Descripción**: Contenedor de buffers y recursos que escriben/leen el núcleo y los dibujadores.

API pública (declarada en cloth.h)
int cloth_init(SDL_Renderer *R, ClothState *S, const ClothParams *P_in, int W, int H)

- **Entradas**
  - R (SDL_Renderer*) — renderer activo.
  - S (ClothState*) — estado a inicializar.
  - P_in (const ClothParams*) — parámetros solicitados por el usuario.
  - W, H (int) — tamaño de ventana inicial.
- **Salidas**
  - **Retorno** (int) — 0 si OK; <0 en error.

    - **Efectos** — S queda listo: buffers reservados, N definido, sprite creado; malla XY precalculada.
- **Descripción**: Normaliza parámetros (deriva GX×GY o baseRadius si hace falta), reserva memoria para draw/depth/order_idx, crea el sprite circular y precalcula la malla base (g_X/g_Y).

void cloth_update(SDL_Renderer *R, ClothState *S, int W, int H, float t)

- **Entradas**
    - R (SDL_Renderer*) — renderer (solo para recrear sprite si cambia tamaño).
    - S (ClothState*) — estado a actualizar.
    - W, H (int) — tamaño de ventana actual.
    - t (float) — tiempo acumulado (segundos).
- **Salidas**
    - **Efectos** — escribe S->draw[k] (posición, radio, color), S->depth[k] y construye S->order_idx ordenado por profundidad. Actualiza S->tx/ty para centrado.
- **Descripción**:
    1. Reacciona a cambios de tamaño (ajusta baseRadius y recrea sprite si es necesario).
    2. Para cada punto de la grilla: calcula Z (onda + gaussiana), rota, proyecta, deriva radio y color (HSV→RGB).
    3. Calcula bounding box proyectado para auto-centrado (suavizado).
    4. Ordena por profundidad con bucket sort O(N) llenando order_idx.
    5. Incluye bucles paralelos con OpenMP (reducciones, atomics) cuando está disponible.

void cloth_destroy(ClothState *S)

- **Entradas**
    - S (ClothState*) — estado a liberar.
- **Salidas**
    - **Efectos** — destruye sprite y libera draw/depth/order_idx.
- **Descripción**: Limpieza del estado; deja punteros en NULL y capacidades en 0.

void cloth_render_seq(SDL_Renderer *R, const ClothState *S) ***(en cloth_draw_seq.c)***

- **Entradas**
    - R (SDL_Renderer*) — renderer activo.
    - S (const ClothState*) — estado ya actualizado.
- **Salidas**
    - **Efectos** — Dibuja N sprites con SDL_RenderCopyF en orden S->order_idx.
- **Descripción**: Backend **secuencial**. Recorre order_idx y, para cada DrawItem, modula color/alpha del sprite y hace una copia de textura al rect destino. Genera **N draw calls** por frame; máxima compatibilidad, menor rendimiento con N grande.

void cloth_render_omp(SDL_Renderer *R, const ClothState *S) ***(en cloth_draw_omp.c)***

- **Entradas**
    - R (SDL_Renderer*) — renderer activo.
    - S (const ClothState*) — estado ya actualizado.

- **Salidas**
    - **Efectos** — Llena buffers de geometría (vértices/índices) y emite **un único** SDL_RenderGeometry. Si no se soporta, cae a cloth_render_seq.
- **Descripción**: Backend paralelo. En un for OpenMP cada iteración escribe sus 4 vértices y 6 índices en rangos disjuntos de arreglos globales reutilizables y, al final, se hace un solo draw call. Reduce drásticamente el overhead frente al secuencial.

void cloth_draw_omp_release(void) ***(en cloth_draw_omp.c)***

- **Entradas**: ninguna.
- **Salidas / Efectos**: libera buffers globales de vértices/índices del backend OMP.
- **Descripción**: Limpieza opcional al cerrar el programa (complementa a cloth_destroy).

Utilidades internas (estáticas en cloth_core.c)

Nota: son funciones "static" con enlace interno; no forman parte de la API pública, pero son esenciales para el funcionamiento.

static inline float clampf(float v, float a, float b)

- **Entradas**: v, a, b (float).
- **Salida**: float — v acotado a [a, b].
- **Descripción**: Utilidad numérica para evitar desbordes/valores inválidos.

static inline void hsv_to_rgb(float h, float s, float v, unsigned char *R, *G, *B)

- **Entradas**: h, s, v (float), punteros R/G/B.
- **Salidas**: escribe R/G/B (0..255).
- **Descripción**: Convierte HSV a RGB para generar paletas animadas por variación del tono.

static inline Vec3 rotX(Vec3 v, float ang), static inline Vec3 rotY(Vec3 v, float ang)

- **Entradas**: v (Vec3), ang (float rad).
- **Salida**: Vec3 rotado.
- **Descripción**: Rotaciones elementales de la manta antes de proyectar.

static inline Vec2 project_point(Vec3 v, int W, int H, float fov, float zCam)

- **Entradas**: v (Vec3), W/H (int), fov/zCam (float).
- **Salida**: Vec2 — coordenadas de pantalla.
- **Descripción**: Proyección perspectiva robusta (protege divisiones por casi-cero).

static SDL_Texture *make_circle_sprite(SDL_Renderer *R, int radius)

- **Entradas**: R (SDL_Renderer*), radius (int).
- **Salida**: SDL_Texture* o NULL si falla.
- **Descripción**: Genera un círculo ARGB con borde suave y highlight; sube pixeles con SDL_UpdateTexture.

static int ensure_capacity_xy(int N)

- **Entradas**: N (int).
- **Salida**: int — 1 si OK; 0 si falta memoria.
- **Descripción**: Reserva/crece buffers globales g_X/g_Y con estrategia amortizada.

static int ensure_capacity_bins(int N)

- **Entradas**: N (int).
- **Salida**: int — 1 si OK; 0 si falla.
- **Descripción**: Reserva/crece buffers para bucket sort: g_bin_idx, g_counts, g_starts, g_write.

static int ensure_capacity_order(ClothState *S, int N)

- **Entradas**: S (ClothState*), N (int).

- **Salida**: int — 1 si OK; 0 si falla.
- **Descripción**: Reserva/crece S->order_idx amortizadamente.

static void derive_grid_from_N(int N, int W, int H, int *GX, int *GY)

- **Entradas**: N, W, H (int), GX/GY (int*).
- **Salidas**: escribe *GX, *GY.
- **Descripción**: Deriva una grilla razonable a partir de N y del aspecto de la ventana.

Backend OMP: utilidades internas (cloth_draw_omp.c)

static int ensure_capacity_geo(int N)

- **Entradas**: N (int).
- **Salida**: int — 1 si OK; 0 si falla.
- **Descripción**: Asegura capacidad para g_verts (4*N) y g_index (6*N). Crecimiento amortizado.

Programa principal (main.c)

static void print_usage(const char *prog)

- **Entradas**: prog (const char*).
- **Salidas**: imprime ayuda en stdout.
- **Descripción**: Muestra sintaxis y flags soportados.

static int parse_grid(const char *s, int *GX, int *GY)

- **Entradas**: s (const char*), GX/GY (int*).
- **Salida**: int — 1 si el formato GXxGY es válido; 0 si no.
- **Descripción**: Parser defensivo del argumento --grid.

int main(int argc, char *argv[])

- **Entradas**: argc/argv — CLI.
- **Salidas**: **Retorno** (int) — 0 si OK; códigos >0 si falla.
- **Descripción**:
  - Parsea argumentos y aplica defaults defensivos.
  - Inicializa SDL (ventana/renderer) y el modo cloth (cloth_init).
  - Bucle principal: eventos → cloth_update → backend de dibujo (cloth_render_omp o cloth_render_seq) → SDL_RenderPresent.
  - Mide FPS y aplica --fpscap.
  - Limpia recursos al salir (cloth_destroy y cloth_draw_omp_release si aplica).

Notas de concurrencia (dónde se paraleliza)

- **cloth_update**:
  - Update de grilla con parallel for collapse(2) y reductions min/max (profundidades).
  - Bounding box con for nowait + critical o reducciones equivalentes.
  - Bucket sort: asignación de bin paralela; conteo con atomic; escritura en order_idx con atomic capture.
- **cloth_render_omp**:
  - Relleno de vértices/índices con parallel for (rangos disjuntos); **1** SDL_RenderGeometry.
- **cloth_render_seq**:
  - Camino totalmente secuencial (N llamadas a SDL_RenderCopyF).

Glosario mínimo

- **HSV**: espacio de color donde animar el hue produce gradientes suaves; luego se convierte a RGB.
- **Bucket sort (ZBINS)**: orden lineal por profundidad vía “cubetas” de Z; rellena order_idx sin qsort.
- **SDL_RenderGeometry**: API de SDL2 para dibujar triángulos con vértices personalizados (un solo draw call para todo el lote).
- **Painter’s order**: pintar del más lejano al más cercano para simular visibilidad sin z-buffer.

Anexos 3

Captura Secuencial 32x32

Imagen 1. Representa secuencial 32 × 24.

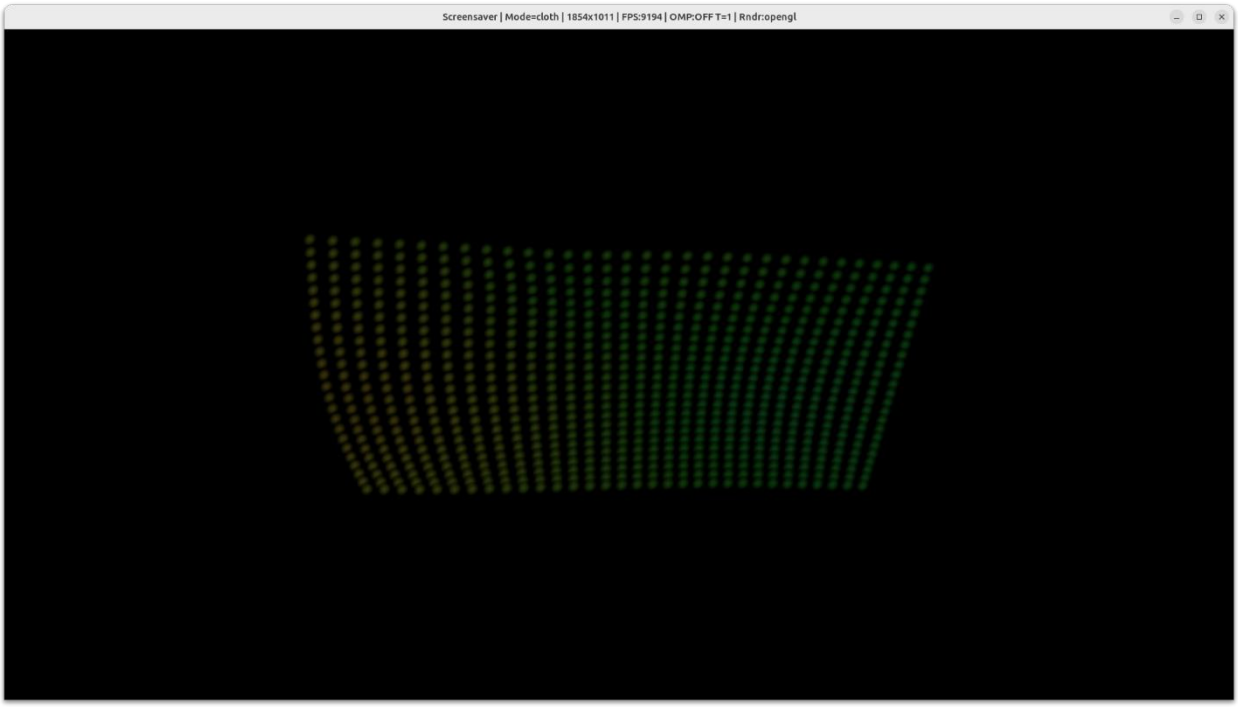


Imagen 2. Secuencial 64 × 48

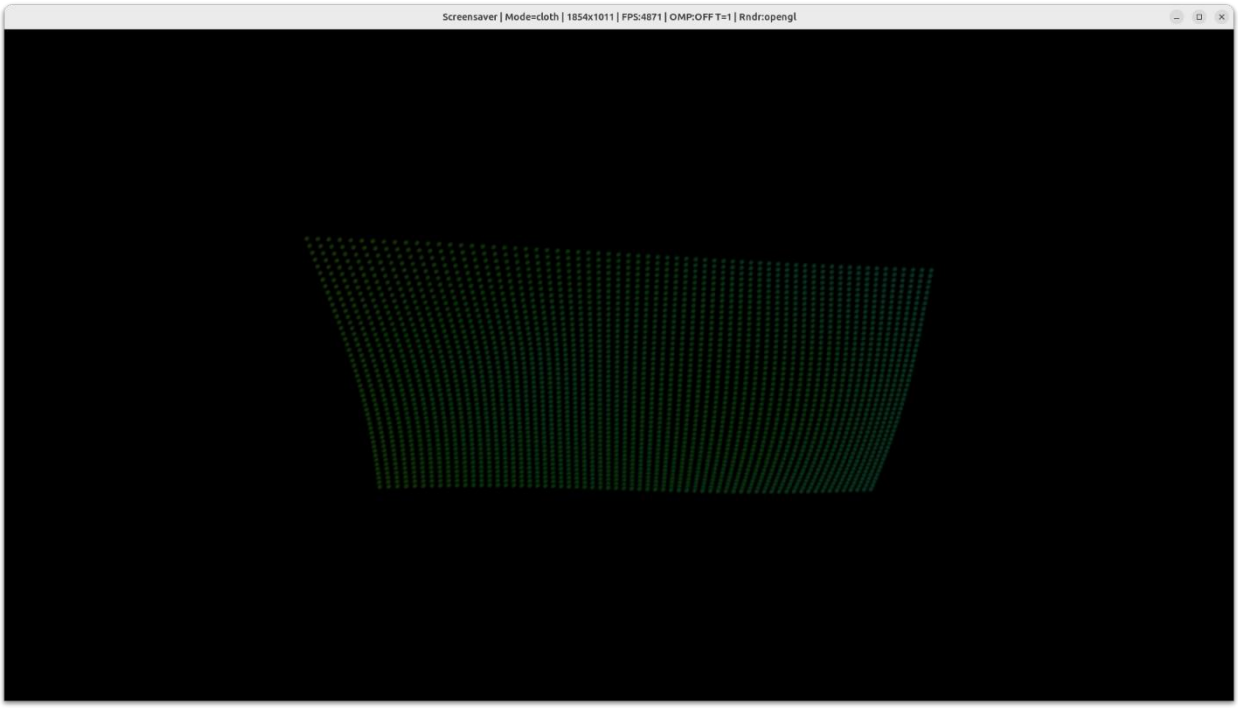


Imagen 3. Secuencial 160 × 120

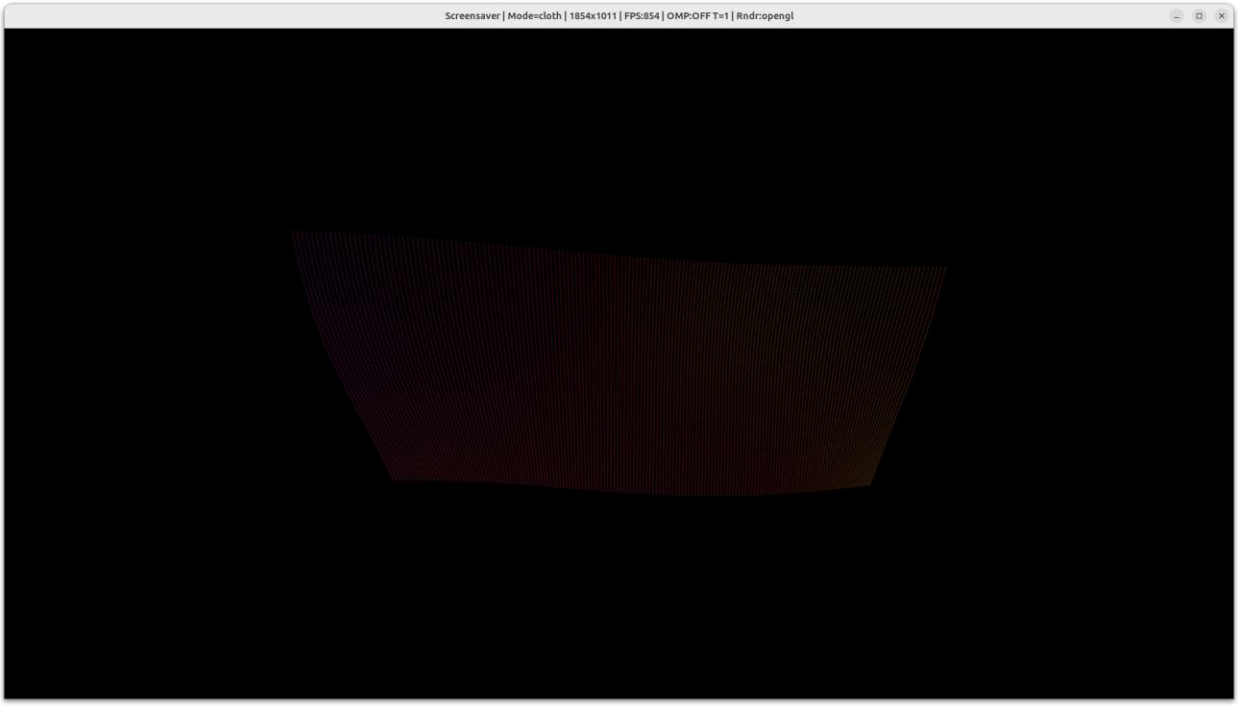


Imagen 4. Secuencial 320 × 240

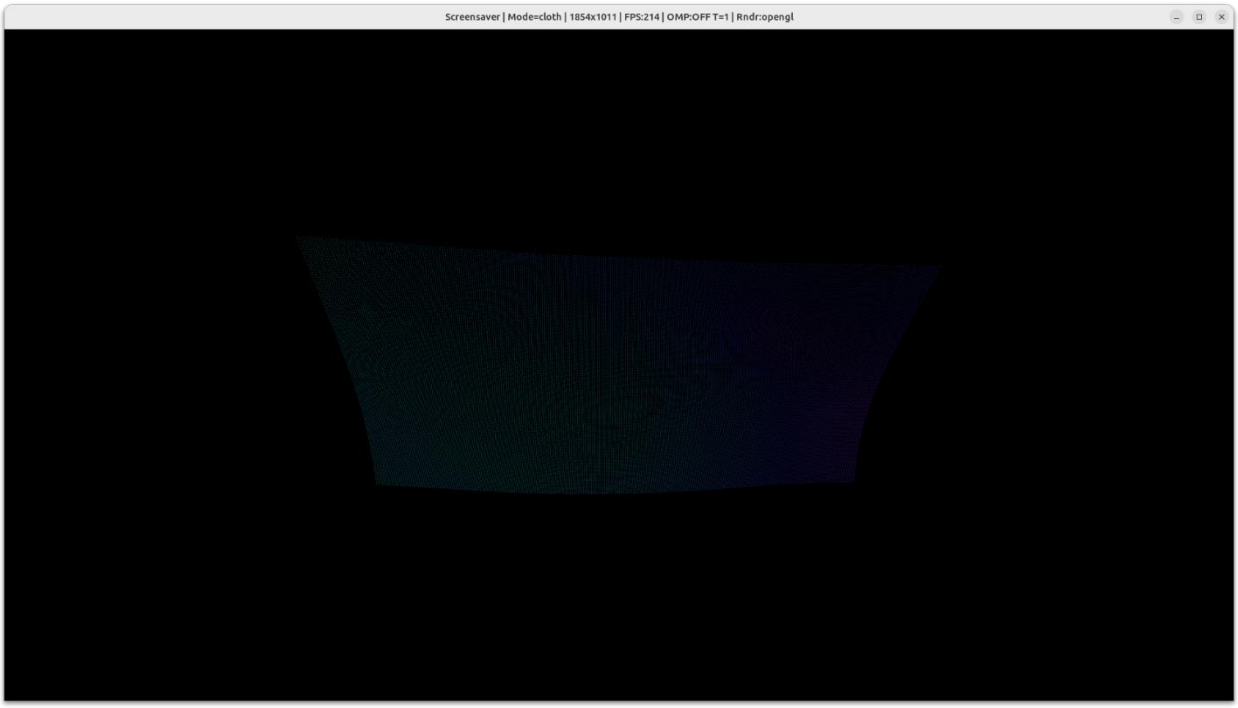


Imagen 5. Secuencial 640 × 480

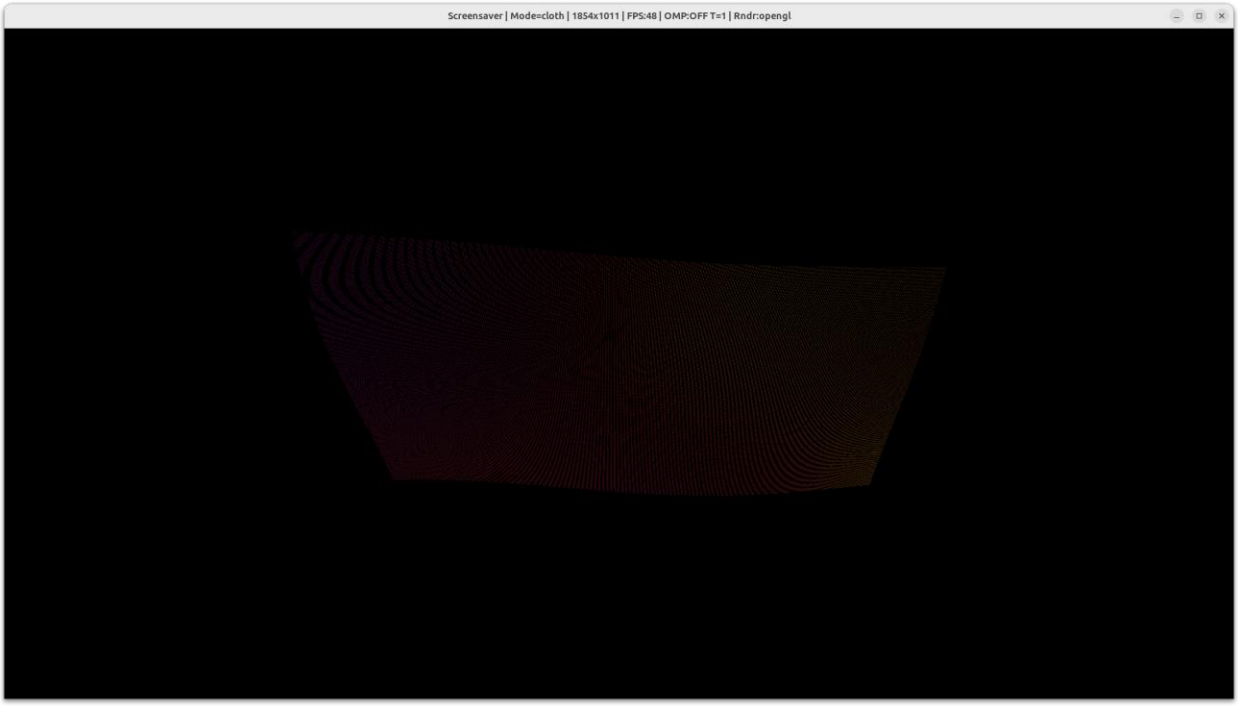


Imagen 6. Secuencial 927 × 505

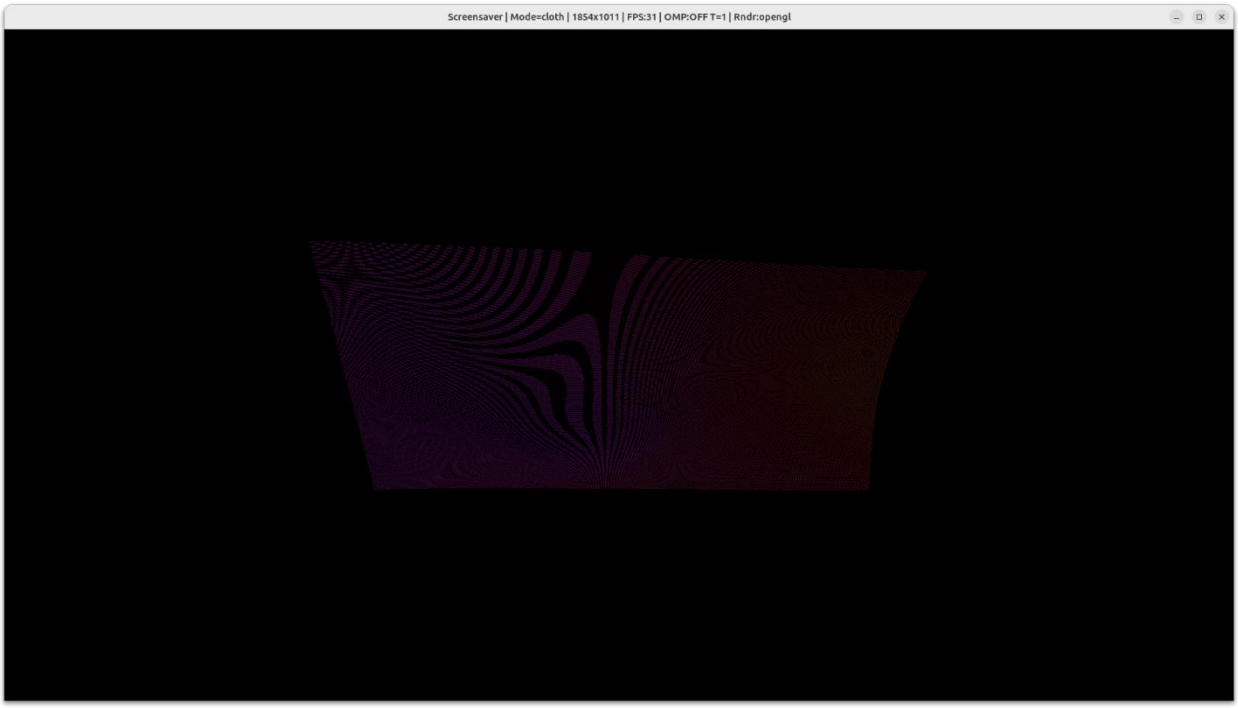


Imagen 7. Secuencial 960 × 540

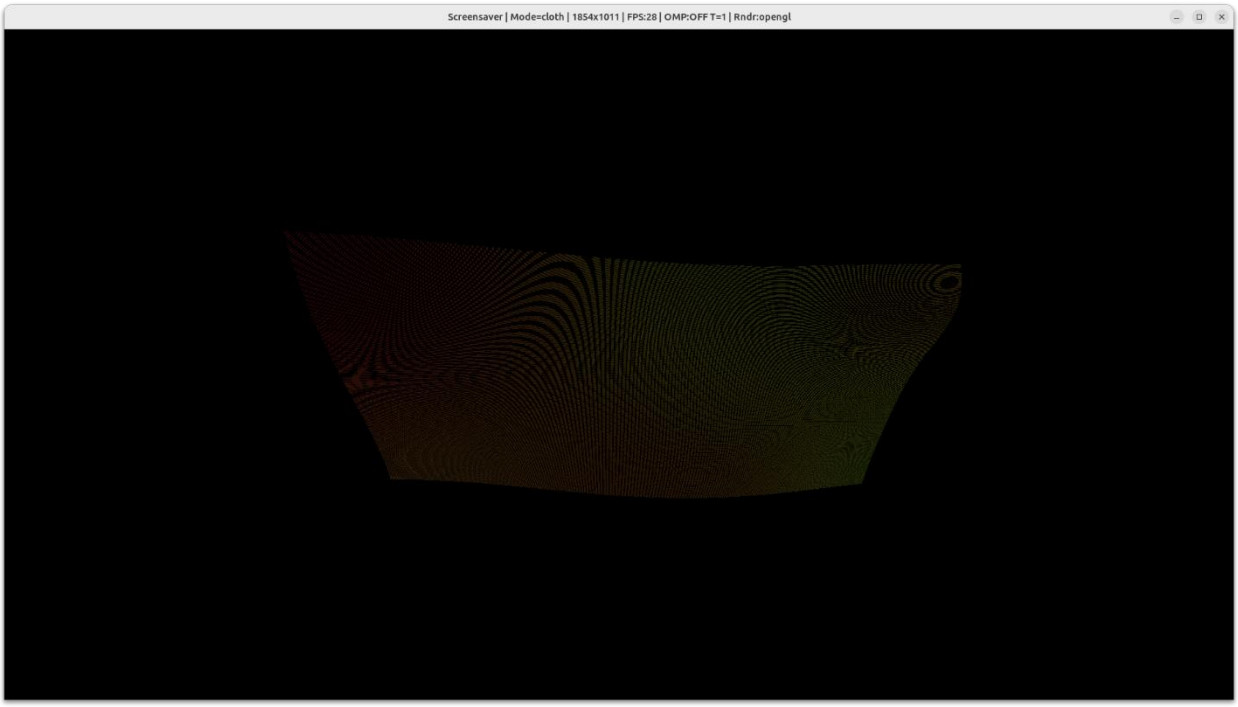


Imagen 8. Secuencial 1280 × 720

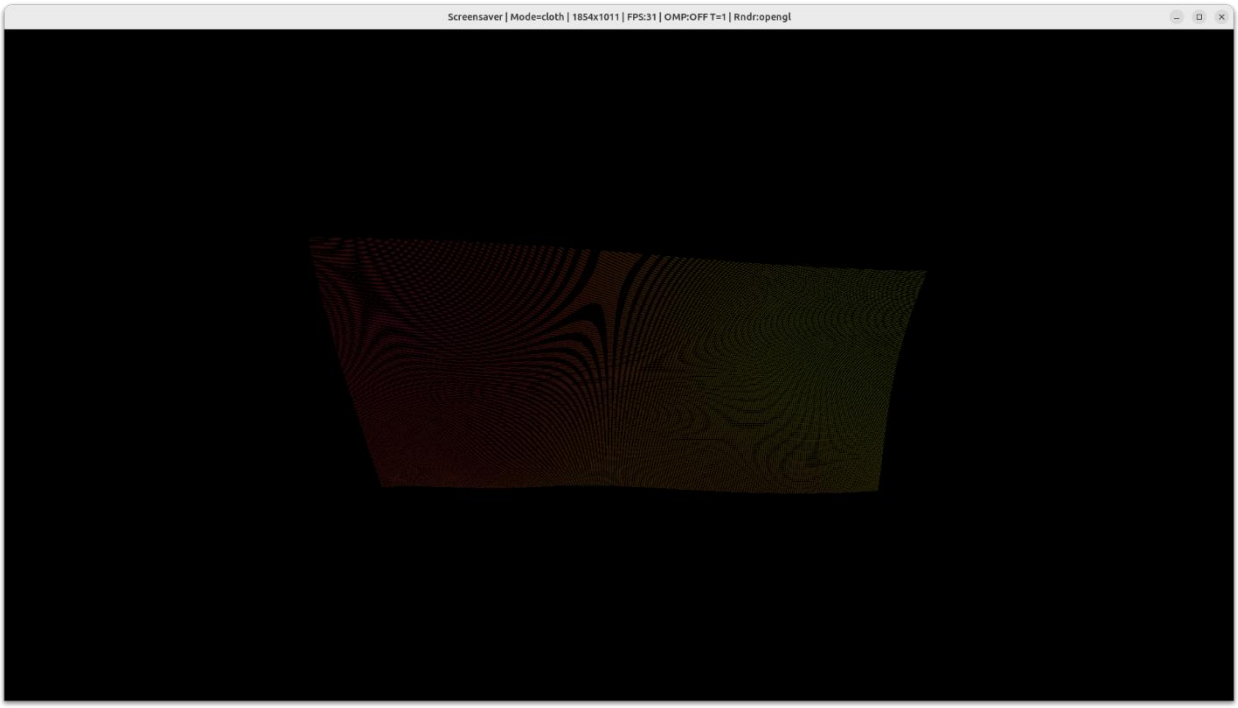


Imagen 9. Secuencial 1854 × 1011

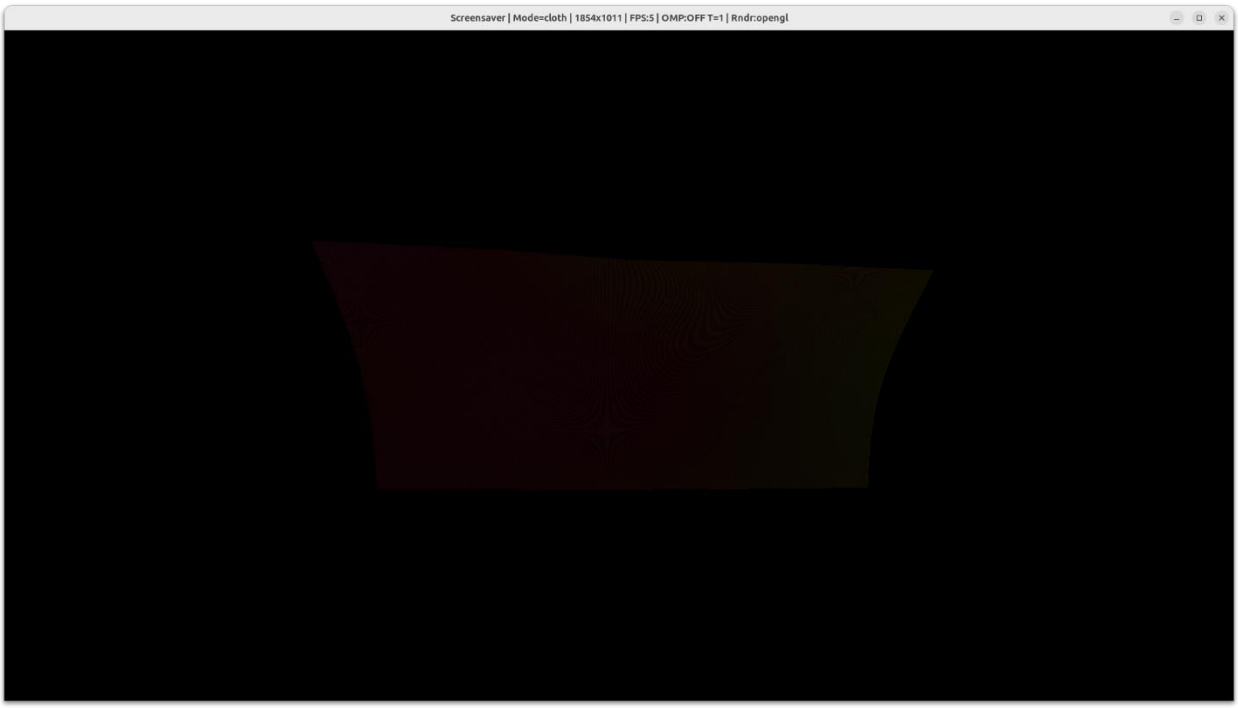


Imagen 10. Secuencial 1920 × 1080

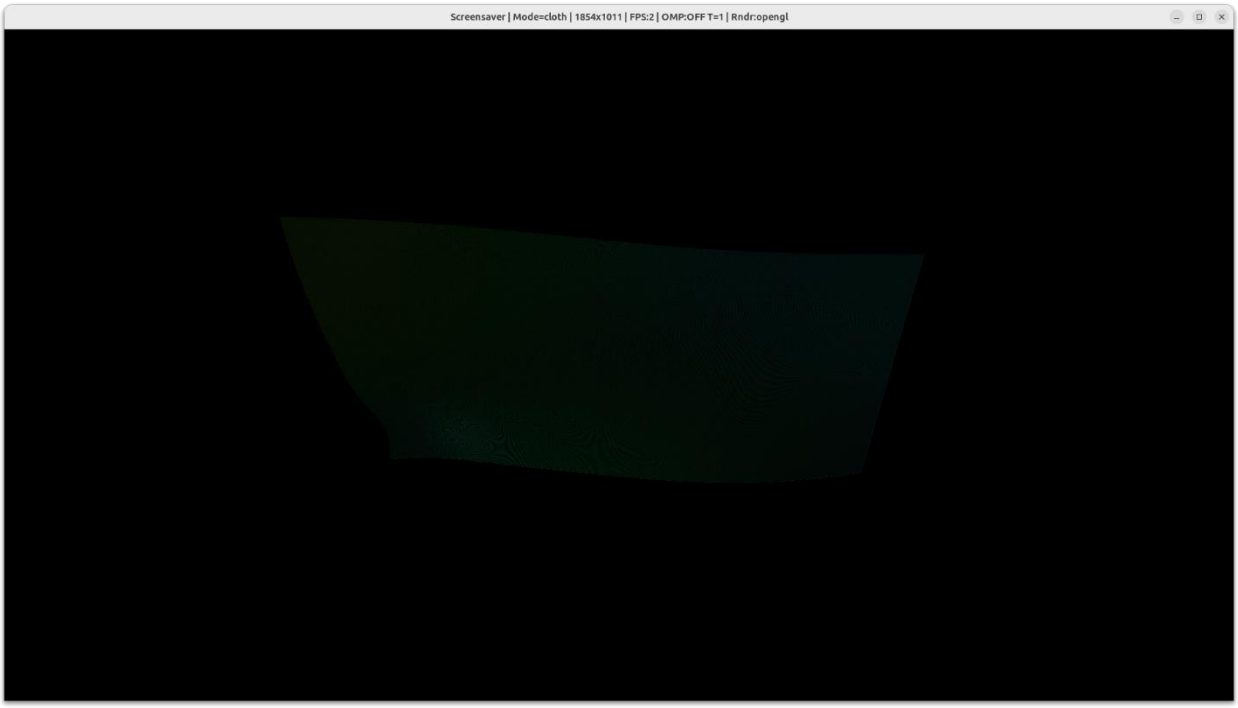


Imagen 11. Paralela 32 × 24

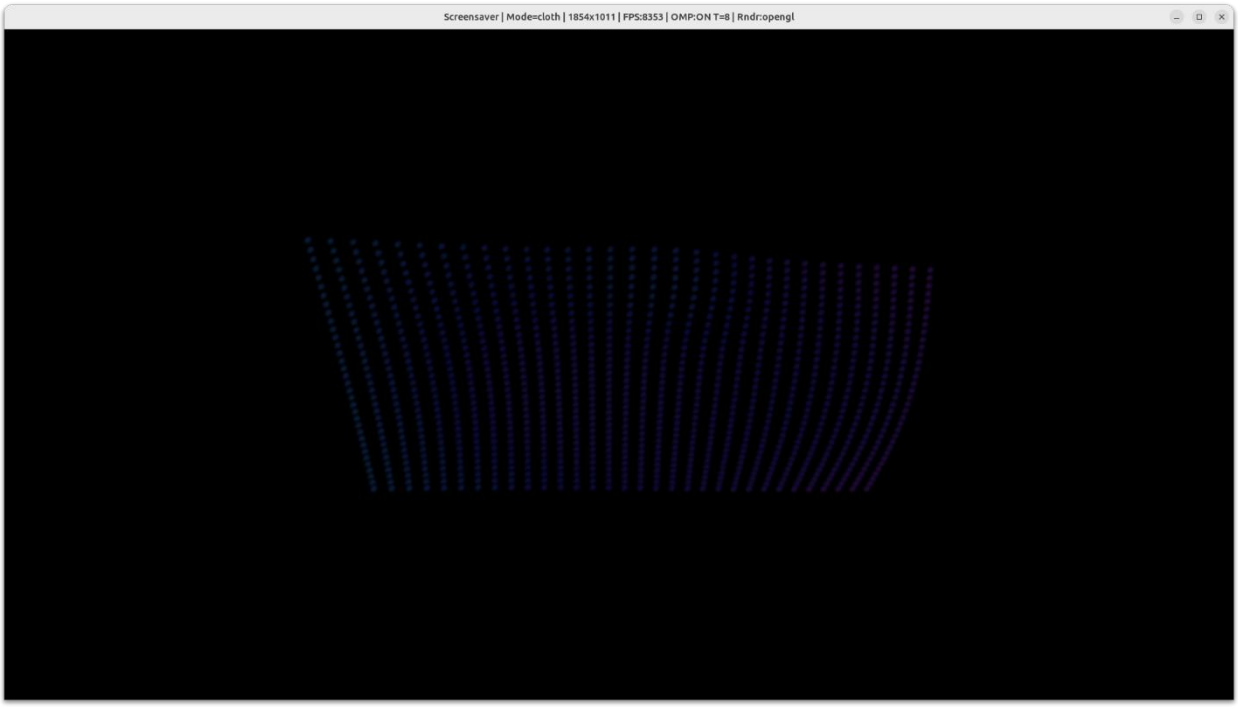


Imagen 12. Paralela 64 × 48

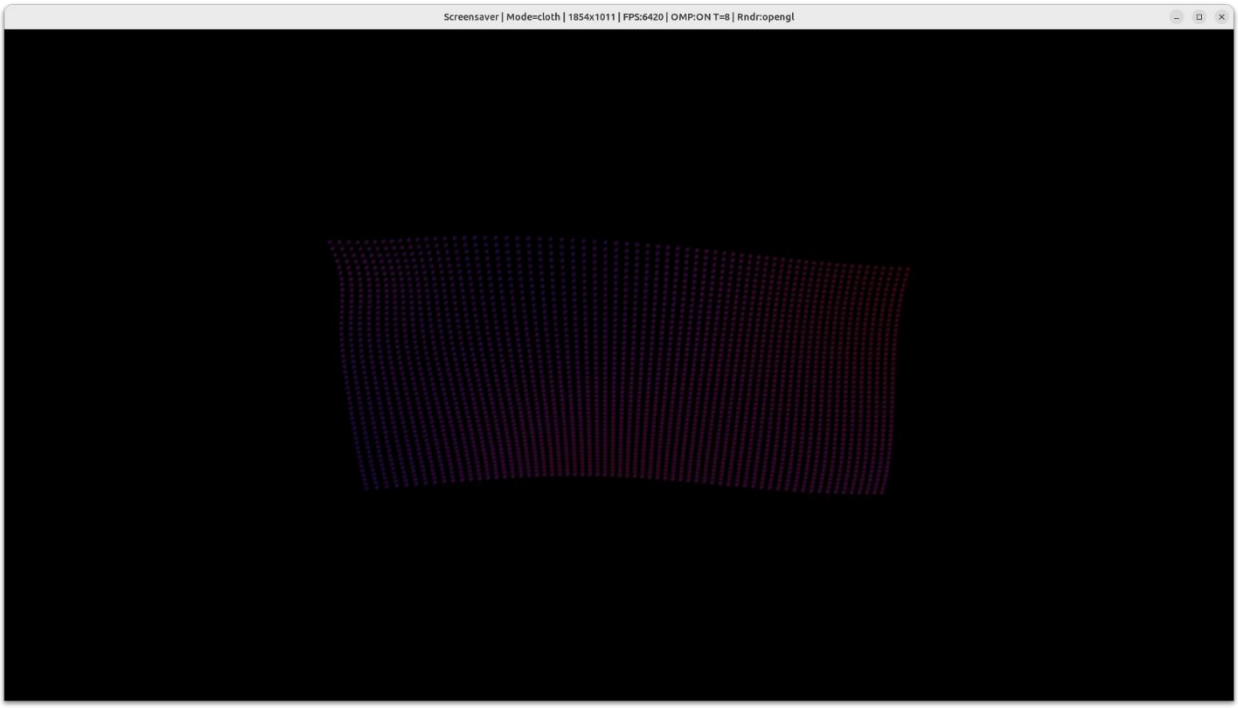


Imagen 13. Paralela 160 × 120

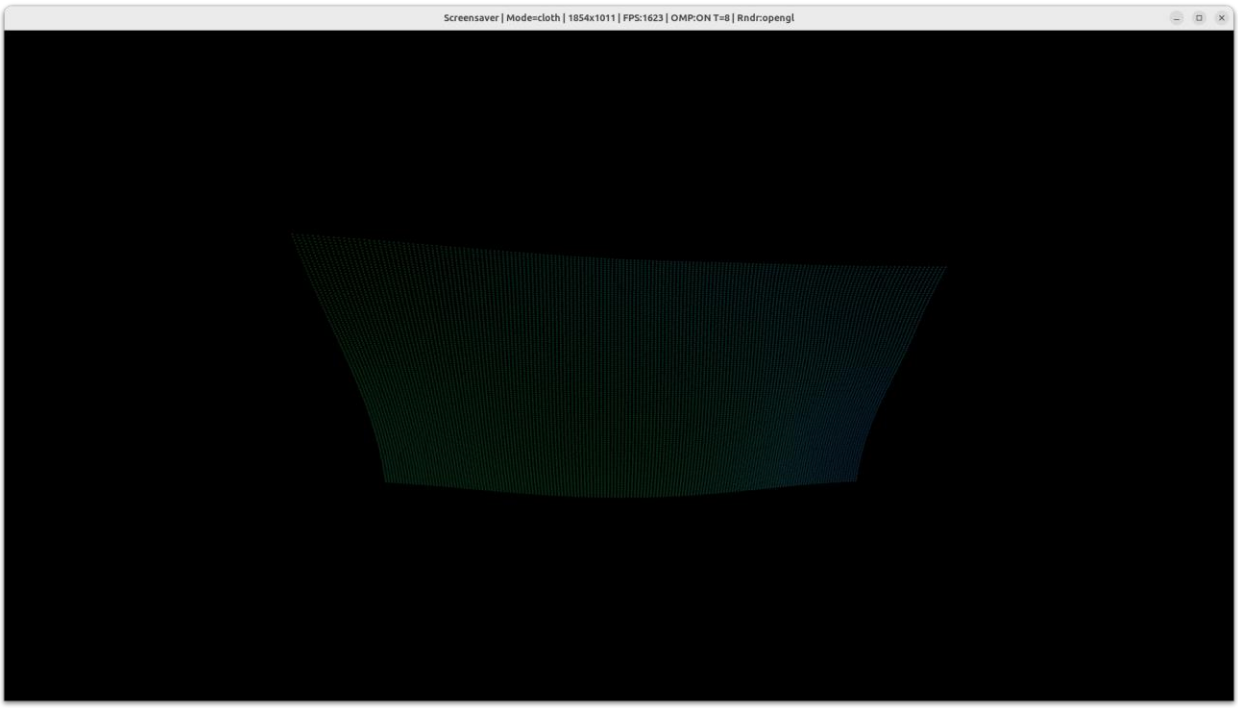


Imagen 14. Paralela 320 × 240

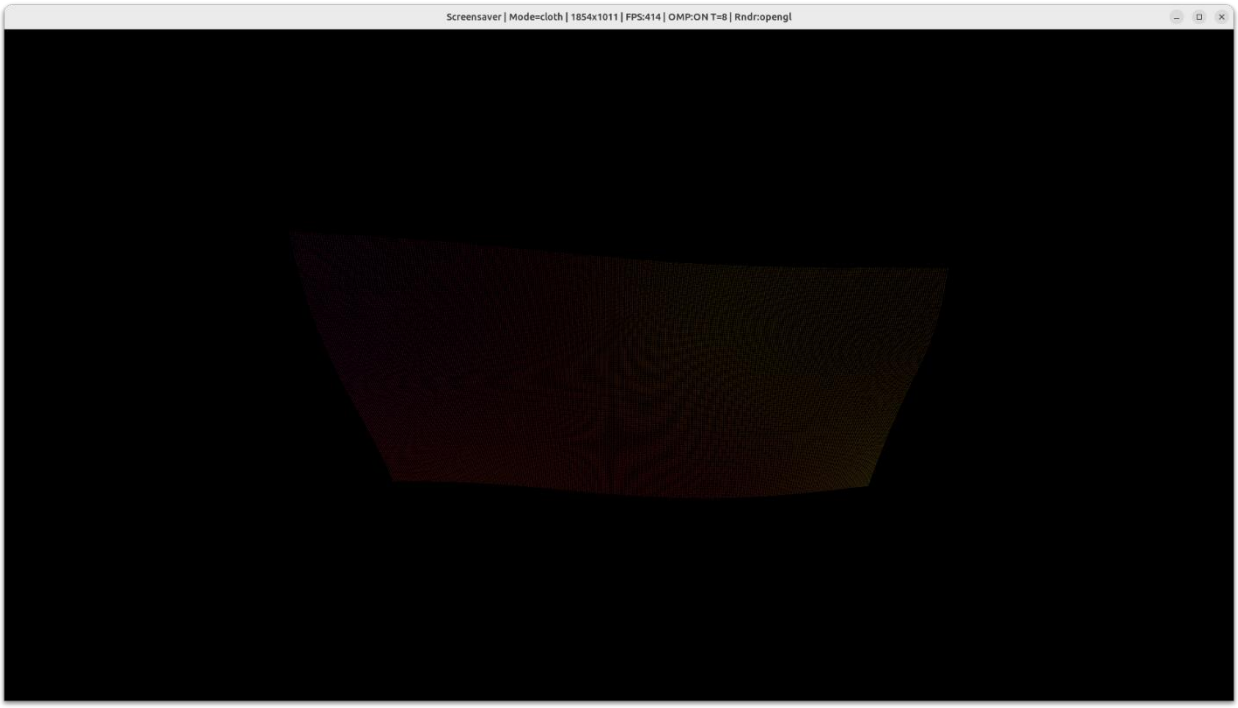


Imagen 15. Paralela 640x480

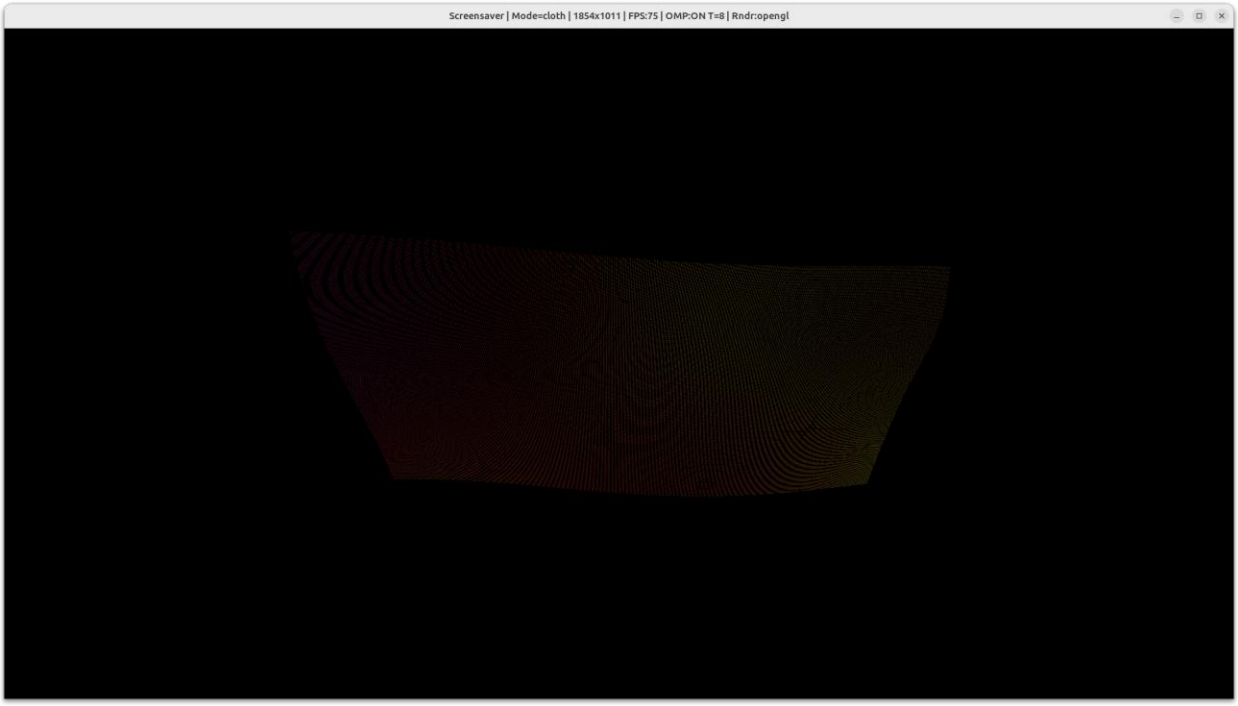


Imagen 16. Paralela 960x540

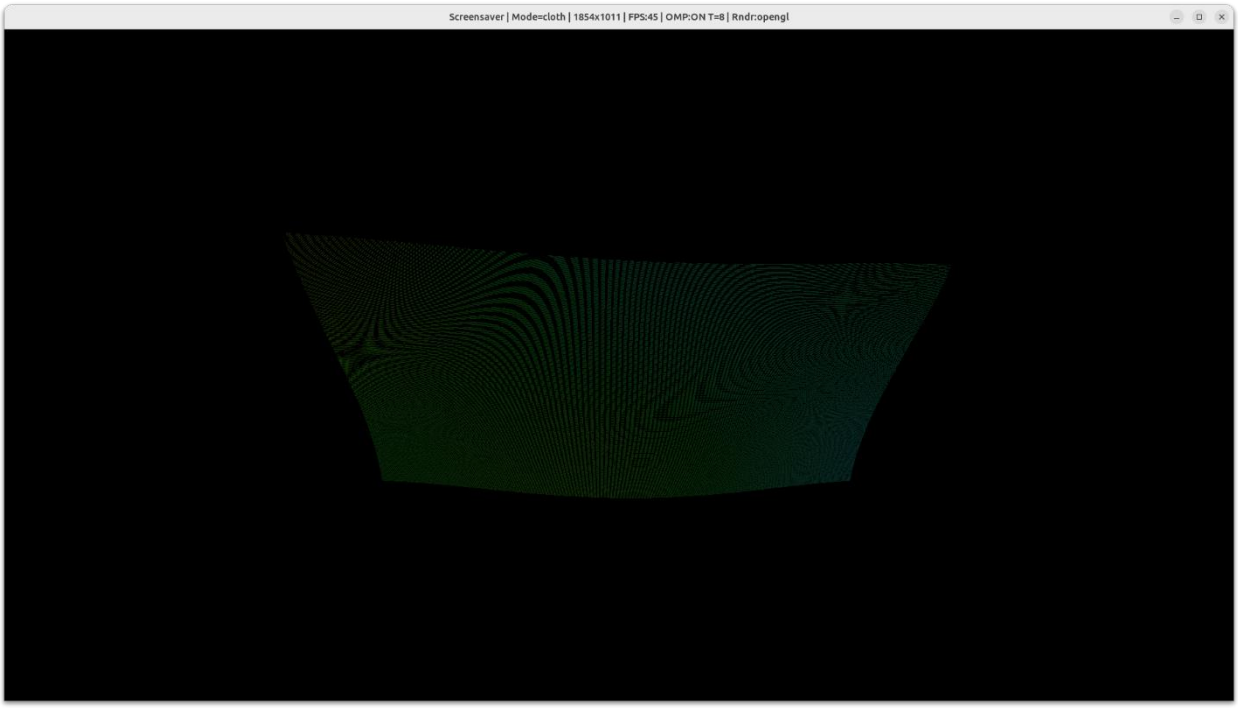


Imagen 17. Paralela 1280x108

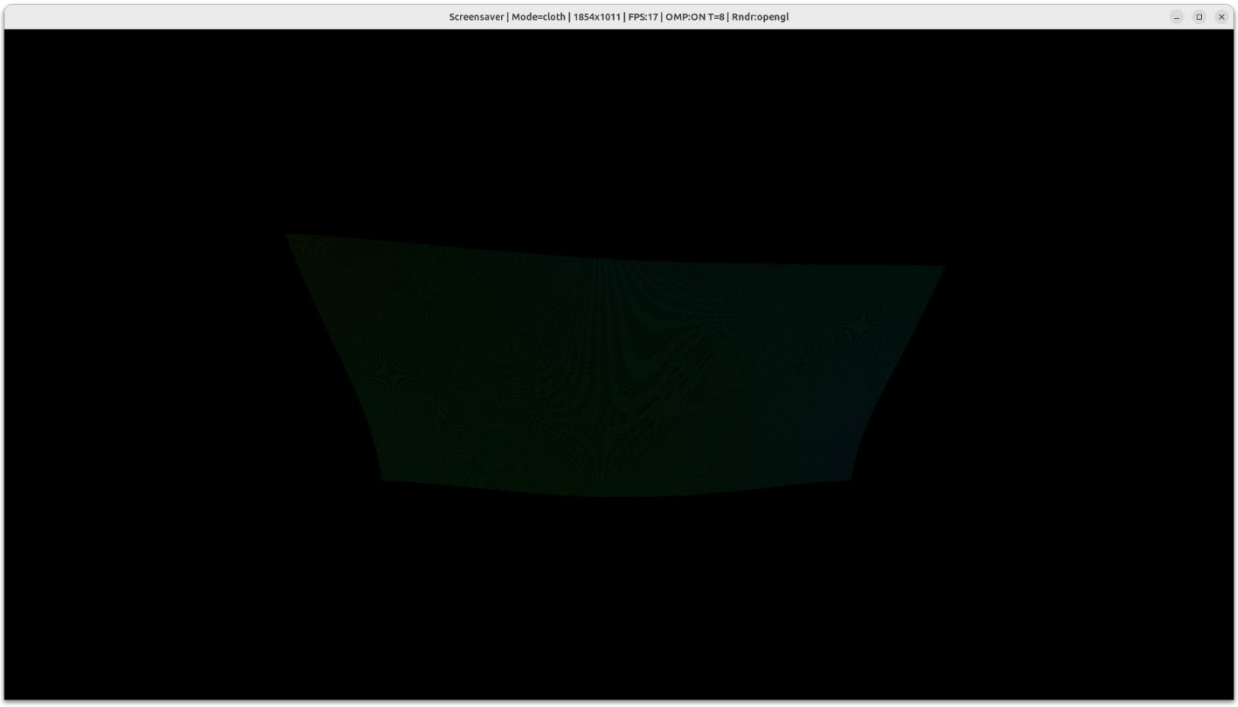


Imagen 18. Paralela 2560x1440

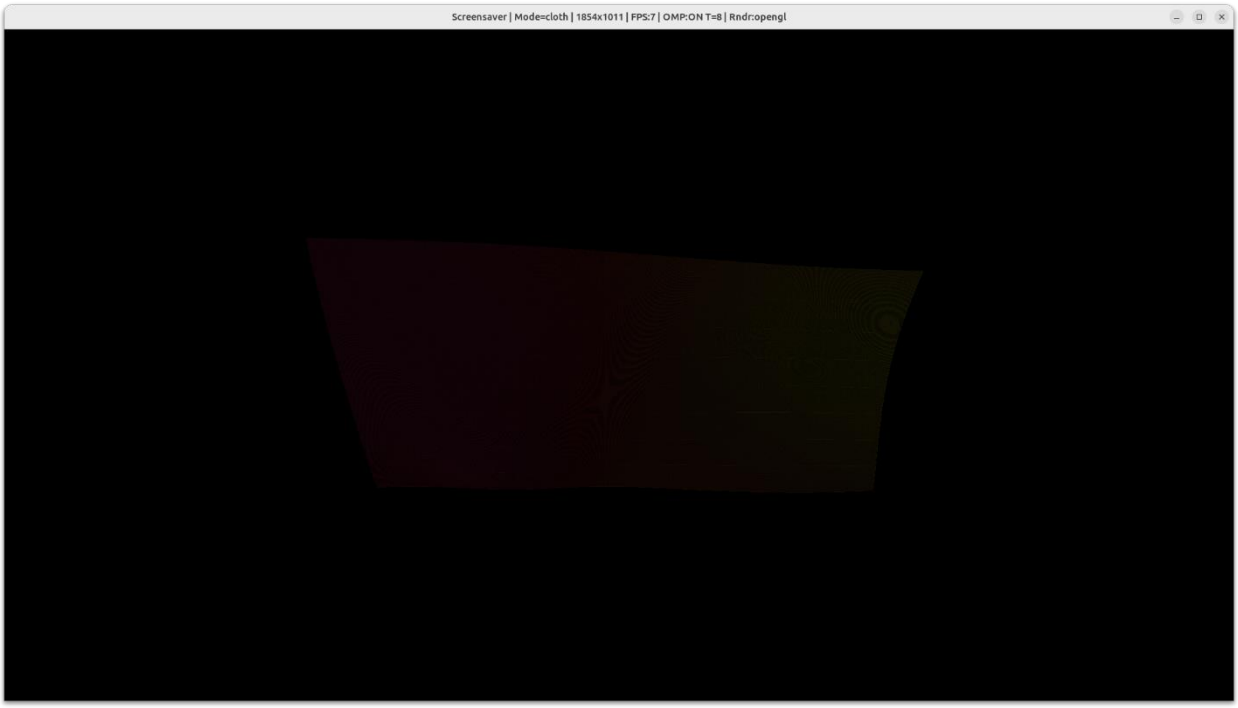


Imagen 19. Paralela 3840x2160

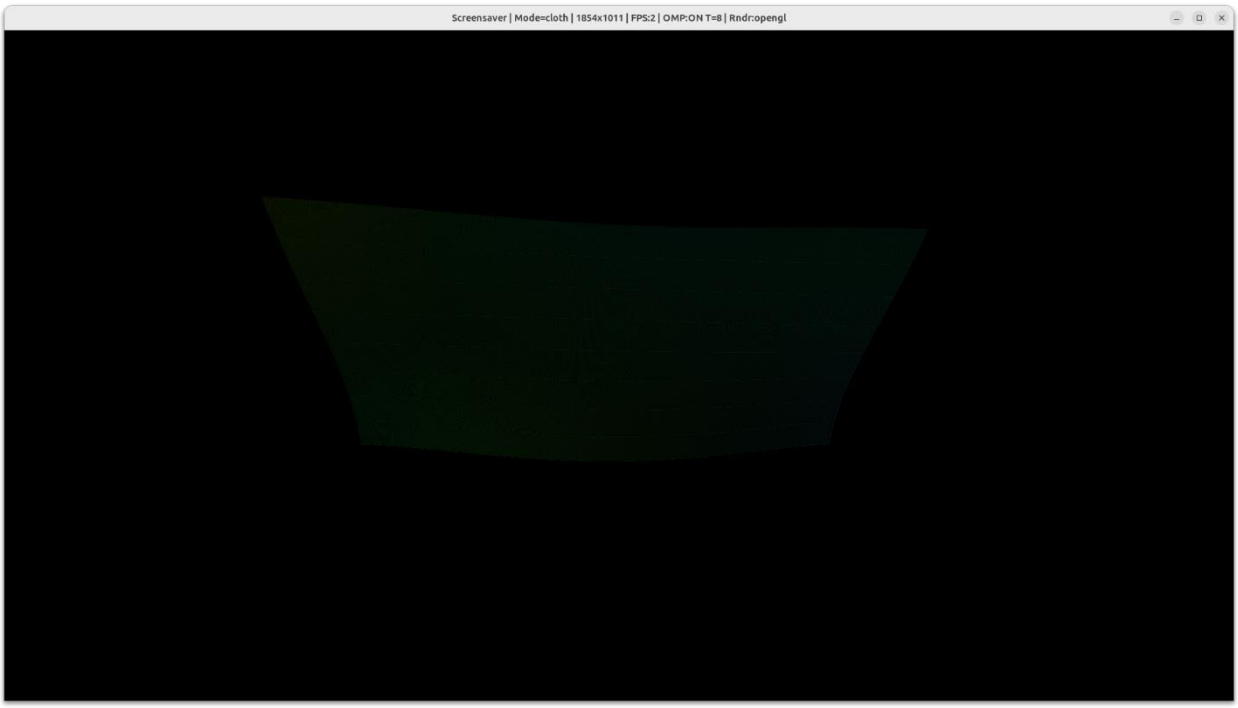
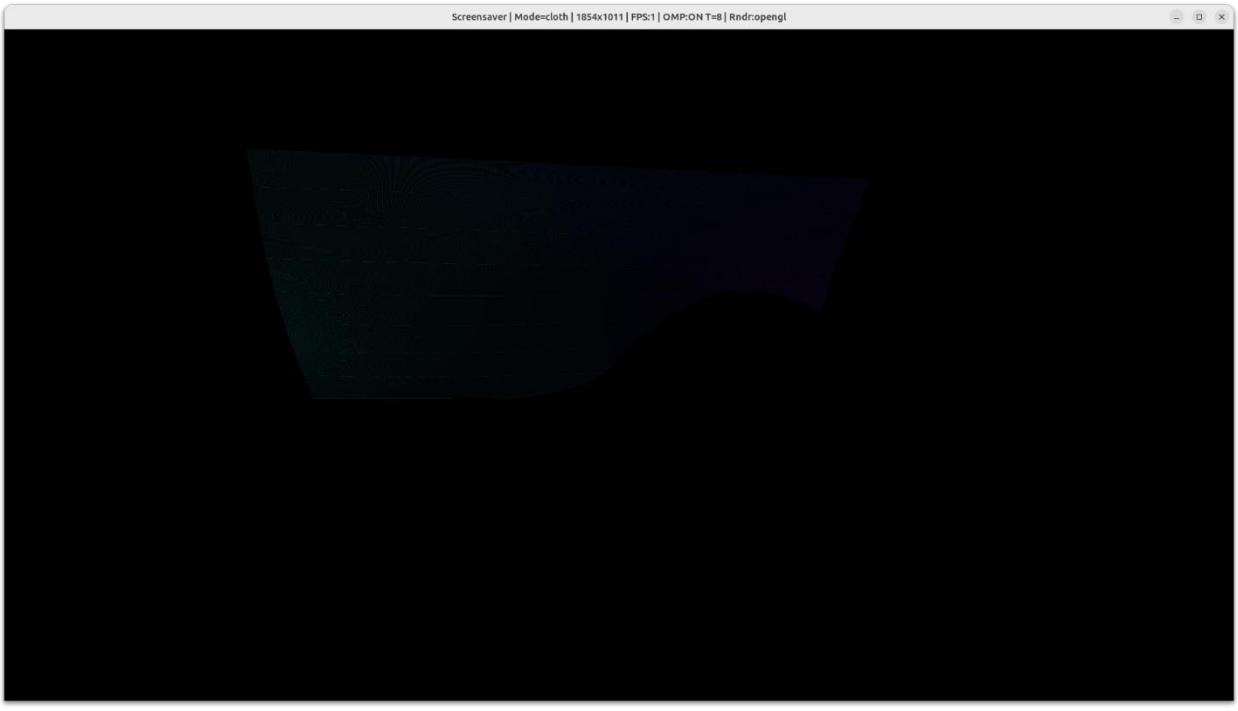


Imagen 20. Paralela 7680x4320



Referencias

- GeeksforGeeks. (2021, 4 junio). *Introduction to Parallel Computing*. GeeksforGeeks. https://www.geeksforgeeks.org/computer-science-fundamentals/introduction-to-parallel-computing/
- GeeksforGeeks. (2023, 19 marzo). *Introduction to Parallel Programming with OpenMP in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp/introduction-to-parallel-programming-with-openmp-in-cpp/
- GeeksforGeeks. (2024a, junio 18). *Painter's Algorithm in Computer Graphics*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/painters-algorithm-in-computer-graphics/
- GeeksforGeeks. (2024b, julio 23). *Bucket Sort Data Structures and Algorithms Tutorials*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/bucket-sort-2/
- GeeksforGeeks. (2025a, julio 12). *What is Parallel Processing ?* GeeksforGeeks. https://www.geeksforgeeks.org/computer-organization-architecture/what-is-parallel-processing/
- GeeksforGeeks. (2025b, julio 23). *C Parallel for loop in OpenMP*. GeeksforGeeks. https://www.geeksforgeeks.org/c/c-parallel-for-loop-in-openmp/
- GeeksforGeeks. (2025c, julio 23). *C Parallel for loop in OpenMP*. GeeksforGeeks. https://www.geeksforgeeks.org/c/c-parallel-for-loop-in-openmp/
- GeeksforGeeks. (2025d, agosto 21). *Computer Organization | Amdahl's law and its proof*. GeeksforGeeks. https://www.geeksforgeeks.org/computer-organization-architecture/computer-organization-amdahls-law-and-its-proof/