UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SIMULACIÓN DE TELAS REPRESENTADAS POR TRIÁNGULOS: TEORÍA E IMPLEMENTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

SEBASTIÁN ALEJANDRO VEGA TICHAUER

PROFESORA GUÍA:
Nancy Hitschfeld Kahler

MIEMBROS DE LA COMISIÓN:
Gonzalo Navarro Badino
Mauricio Palma Lizana

Este trabajo ha sido parcialmente financiado por:
FONDECYT 1211484

SANTIAGO DE CHILE
2023

SIMULACIÓN DE TELAS REPRESENTADAS POR TRIÁNGULOS: TEORÍA E IMPLEMENTACIÓN

La simulación de telas ha tomado cada vez más peso con el pasar de los años. Tanto para simulaciones físicas exactas, animaciones o videojuegos, se han dedicado importantes recursos en el desarrollo de nuevas y mejores estrategias de simulación.

La reciente pandemia y el aumento de la compra de ropa online aporta la motivación inicial a este trabajo. ¿Por qué no existe probadores virtuales? ¿es caro computacionalmente simular telas? Para obtener una noción inicial que permita a futuro responder estas preguntas, se estudian las bases de las simulaciones de tela y se desarrolla un programa para poder visualizarlas en tiempo real de manera interactiva.

El programa de visualización permite combinar distintos algoritmos y ver su resultado en una escena sencilla: una tela cayendo sobre un modelo 3D a elección. Tanto las características de la tela, el objeto y los parámetros de los algoritmos se pueden manipular desde la interfaz del programa antes o durante la simulación.

Una simulación tiene tres componentes principales: un modelo que define cómo responde la tela ante el cambio, un método de integración numérica que calcula la próxima posición de la tela a partir de sus posiciones previas y un método de colisión que detecta y corrige cuando la tela intersecta consigo misma u otro objeto.

Este trabajo estudia e implementa en formato de librería dos modelos, masa - resorte y mecánicas continuas; un método de integración, Verlet; y un método de colisión, acelerado con octree. La librería desarrollada es utilizada por el programa de visualización, pero es independiente a este, por lo que permite implementar nuevos algoritmos y evaluar su desempeño (velocidad y memoria) de manera sencilla.

Queda como principal trabajo futuro mejorar el encapsulamiento general de las capas que componen el programa de visualización e implementar en la librería algoritmos que consideren fuerzas externas como viento y roce; y colisiones de la tela consigo misma.

Sí se pudo burro,
sí se pudo.

Agradecimientos

A mi familia por haber patrocinado esta ~~no tan corta~~ travesía universitaria.

A mi profesora guía Nancy Hitschfeld Kahler por haber aportado el optimismo que me faltaba.

A Roberto León Valdez de calidad de vida por darme perspectiva.

A mis amigos por nunca coincidir y sin quererlo dejarme terminar este trabajo.

A los autores de todos los libros, artículos, foros y blogs que dieron forma a este texto.

Tabla de Contenido

1. Introducción **1**
- 1.1. Contexto y motivación . . . 1
- 1.2. Objetivos . . . 1
  - 1.2.1. Objetivos generales . . . 2
  - 1.2.2. Objetivos específicos . . . 2
- 1.3. Metodología . . . 2
- 1.4. Estructura de la memoria . . . 2

2. Estado del arte **4**
- 2.1. Programas profesionales . . . 4
  - 2.1.1. Animación . . . 4
  - 2.1.2. Videojuegos . . . 4
  - 2.1.3. Diseño . . . 5
- 2.2. Soluciones en CPU . . . 5
- 2.3. Soluciones en GPU . . . 5
- 2.4. Machine Learning . . . 6

3. Trabajo previo **8**
- 3.1. Simulación en navegador web . . . 8
- 3.2. Simulación utilizando GPU . . . 9

4. Teoría e implementación **11**
- 4.1. Representación . . . 11
  - 4.1.1. Discretización espacial . . . 11
  - 4.1.2. Discretización temporal . . . 12
  - 4.1.3. Formalización . . . 13
  - 4.1.4. Estructura de datos: Half-Edge . . . 14
- 4.2. Ciclo de la simulación . . . 15
- 4.3. Modelos de fuerzas y restricciones . . . 16
  - 4.3.1. Masa - Resorte . . . 16
    - 4.3.1.1. Aplicación de restricciones . . . 17
    - 4.3.1.2. Aplicación de fuerzas . . . 19
  - 4.3.2. Mecánicas continuas . . . 20
    - 4.3.2.1. Aplicación de fuerzas . . . 22
- 4.4. Evolución en el tiempo . . . 26
  - 4.4.1. Integración de Verlet . . . 27
- 4.5. Manejo de colisiones . . . 27
  - 4.5.1. Detección . . . 27

4.5.1.1. Tela - Objeto . . . 28
4.5.1.1.1 Octree . . . 28
4.5.2. Respuesta . . . 29

5. Evaluación **32**
5.1. Modelos . . . 32
5.2. Detección de colisiones . . . 34

6. Programa de simulación **36**
6.1. Requerimientos . . . 36
6.2. Dependencias . . . 37
6.3. Arquitectura . . . 37
6.3.1. Simulación . . . 38
6.3.1.1. Objeto . . . 39
6.3.1.2. Modelo . . . 41
6.3.1.3. Integración . . . 42
6.3.1.4. Colisión . . . 42
6.3.2. Renderizado . . . 43
6.3.2.1. Objeto según API . . . 44
6.3.2.2. Framebuffer . . . 45
6.3.2.3. Shaders . . . 46
6.3.2.4. Luces . . . 46
6.3.2.5. Cámara . . . 47
6.3.3. Interfaz . . . 47

7. Conclusiones **50**

8. Trabajo futuro **51**

Bibliografía **52**

Anexos **54**
A. Diagramas de clase . . . 54
B. Intersección entre primitivas . . . 56
B.1. Triángulo - Plano (Test) . . . 56
B.2. Triángulo - Caja (Test) . . . 56
B.3. Segmento - Caja (Test) . . . 59
B.4. Segmento - Triángulo (Intersección) . . . 60
C. Configuración utilizada por Figura . . . 62
C.1. Figura 4.1 . . . 62
C.2. Figura 4.7 . . . 62
C.3. Figura 4.8 . . . 62
C.4. Figura 4.9 . . . 62
C.5. Figura 4.12 . . . 63
C.6. Figura 4.17 . . . 63
C.7. Figura 4.18 . . . 63
C.8. Figura 5.1 . . . 63
D. Código fuente . . . 64

D.1. Modelos de fuerzas y restricciones 64
D.1.1. Masa - Resorte 64
D.1.2. Mecánicas continuas 70
D.2. Evolución en el tiempo 78
D.2.1. Integración de Verlet 79
D.3. Manejo de colisiones 80
D.3.1. Octree 80

Índice de Tablas

3.1. Rendimiento de los simuladores por cantidad de partículas después de 600 frames. 9
4.1. Pasos de tiempo para los frames por segundo más comunes. 13
4.2. Partículas para la malla de ejemplo ejemplo de la Figura 4.1. 14
4.3. Caras para la malla de ejemplo ejemplo de la Figura 4.1. 14
4.4. Arcos para la malla de ejemplo ejemplo de la Figura 4.1. 14
4.5. Medio arcos para la malla de ejemplo ejemplo de la Figura 4.1. 15

Índice de Ilustraciones

3.1. Simulador web escrito en Javascript. 8
3.2. Rendimiento del simulador web después de 600 frames. 9
3.3. Rendimiento del simulador con GPU después de 600 frames. 10
3.4. Simulador con uso de GPU. Inestabilidad en el sistema. 10
4.1. Tela en su estado inicial y dejada caer sobre el modelo 3D *Stanford bunny*. . . 12
4.2. Ejemplo de Half-Edge en una malla simple de cuatro vértices. 14
4.3. Tela en reposo, deformación de estiramiento, deformación de cizallamiento y deformación de plegado respectivamente. 16
4.4. Efecto de un resorte entre dos partículas donde L es la distancia natural entre estas. 16
4.5. Resortes de estiramiento, de cizallamiento y plegado respectivamente. 17
4.6. Ejemplo de violación de una restricción al aplicar otra. 17
4.7. Estiramiento de la tela con 1, 5 y 10 iteraciones de restricción. 18
4.8. Telas con 100, 2500 y 10000 partículas y 10 subciclos con aplicación de restricciones. 18
4.9. Telas con 100, 2500 y 10000 partículas y 10 subciclos con cálculo de fuerzas. . 20
4.10. Triángulos en espacio 2D (u, v) y 3D (x, y, z). 21
4.11. Triángulos adyacentes para el cálculo de fuerza de plegado. 24
4.12. Modelo de mecánicas continuas con y sin cálculo de fuerzas de plegado respectivamente. 26
4.13. Ejemplo de integración de Verlet. 27
4.14. División del espacio por un *Octree*. 28
4.15. Triángulo guardado en nodo que lo contiene y en cada nodo que lo intersecta respectivamente. 29
4.16. Colisión y dos posibles respuestas. 30
4.17. Ejemplo de fricción infinita. 30
4.18. Ejemplo de fricción nula. 31
5.1. Tiempo de ejecución para generar 600 frames con modelo de masa - resorte y mecánicas continuas. 32
5.2. Tiempo de ejecución por etapa de simulación. 33
5.3. Renderizado de resultado de los modelos. Modelo masa - resorte con aplicación de fuerzas, masa - resorte con aplicación de restricciones, mecánicas continuas con plegado y mecánicas continuas sin plegado respectivamente. 34
5.4. Cantidad de verificaciones realizadas con fuerza bruta. 34
5.5. Cantidad de verificaciones realizadas con uso de octree. 35
6.1. Programa de simulación. 36
6.2. Arquitectura principal del programa. 37
6.3. Arquitectura de la capa de simulación. 38

6.4. Arquitectura de la capa de renderizado. 43
6.5. Estructura de la interfaz de usuario. 49

Capítulo 1

Introducción

1.1. Contexto y motivación

Dentro del área de la computación gráfica, ya sea para el entretenimiento, comercio electrónico o para simulaciones físicas de gran precisión, el problema de modelar ropa y tela ha ido tomando cada vez más peso. Tanto es así que grandes empresas de animación como Pixar y Dreamworks dedican importantes recursos en el desarrollo de nuevas y mejores estrategias para este fin.

En el contexto de pandemia que se ha vivido y sigue viviendo a nivel mundial, se ha visto un aumento en la venta de ropa de manera online que ha llegado hasta un 19 % por año en algunos países como Estados Unidos[1]. Esto es ciertamente positivo a nivel económico, sin embargo, acarrea problemas al usuario quien no puede probarse las prendas antes de comprarlas. Así a la vez que aumentan las ventas online, también aumentan las devoluciones que han subido hasta un 6 % por año en Estados Unidos[2].

Si el usuario supiera de antemano cómo le va a quedar la ropa antes de comprarla, se podría disminuir la tasa de devoluciones. Debido a esto surgen las preguntas ¿por qué no se ven simuladores de ropa en las tiendas online? ¿no resulta de interés o es computacionalmente inviable? ¿existirán modelos que permitan simular ropa con bajo costo computacional?

Este caso concreto y tantos otros más, como pueden ser el realismo en las películas animadas o la optimización de la ropa de los personajes en videojuegos, muestran que puede ser de interés el estudio de la simulación de telas. Con esto en mente, el siguiente trabajo no busca responder las preguntas anteriormente presentadas, si no, entregar las bases necesarias y desarrollar un prototipo de software con el que evaluar los algoritmos clásicos, y así entender lo relevante al tema para eventualmente, con mayor investigación, sí poder contestarlas.

1.2. Objetivos

[1] https://finance.yahoo.com/news/insights-global-clothing-b2c-e-103800717.html
[2] https://cnb.cx/3rRe028

1.2.1. Objetivos generales

Estudiar la simulación de telas, sus distintos componentes y estrategias más extendidas. Desarrollar un programa que implemente los distintos componentes y estrategias; y permita visualizar la simulación en tiempo real.

1.2.2. Objetivos específicos

1. Estudiar los componentes y el ciclo general de la simulación de telas.
2. Desarrollar un programa que permita visualizar e implementar los componentes y estrategias de la simulación.
3. Estudiar e implementar los modelos de masa - resorte y mecánicas continuas.
4. Estudiar e implementar la integración de Verlet.
5. Estudiar e implementar colisiones aceleradas con *Octree*.
6. Evaluar los algoritmos implementados.

1.3. Metodología

La metodología empleada es a grandes rasgos iterativa. Se estudian las bases de la simulación de telas y paralelamente se empieza el desarrollo de un programa simple para visualizarlas. Mientras que se avanza en los estudios se implementan las distintas estrategias y algoritmos vistos. A partir de cada nueva necesidad que presenta la simulación se genera una nueva versión del programa capaz de suplirla. Una vez finalizados los estudios, se refactoriza el programa separando su funcionalidad en capas, de tal forma que la simulación sea capaz de funcionar por sí sola, independiente del renderizado e interfaz de usuario.

1.4. Estructura de la memoria

Los capítulos de este trabajo están estructurados de la siguiente manera:

- En el capítulo 2 “Estado del arte”, se presentan las tecnologías actualmente usadas en la industria y los últimos avances en cuanto a algoritmos y estrategias para la simulación de telas.
- En el capítulo 3 “Trabajo previo”, se muestra de manera superficial dos proyectos realizados con anterioridad que sirven de base para este trabajo.
- En el capítulo 4 “Teoría e implementación”, se ahonda en los principios de una simulación de telas junto con algunos algoritmos y estrategias largamente utilizadas. Este capítulo ofrece una visión teórica complementada con experimentos y discusiones sobre los problemas enfrentados durante su implementación en este trabajo.
- En el capítulo 5 “Evaluación”, se muestran los resultados de la simulación que justifican las decisiones tomadas en el capítulo 4.

- En el capítulo 6 “Programa de simulación”, se explica la arquitectura de un programa desarrollado con el objetivo de visualizar en tiempo real el efecto de los algoritmos y estrategias involucradas en la simulación. Este capítulo sirve de documentación para el uso individual del código como un *framework* de simulación.
- En el capítulo 7 “Conclusiones”, se expone la completitud de este trabajo y reflexiones acerca del aporte que ha generado realizarlo.
- En el capítulo 8 “Trabajo futuro”, se hace un repaso superficial de las falencias encontradas y los posibles próximos pasos en la extensión del programa de simulación desarrollado.

Capítulo 2

Estado del arte

Cuando se refiere al estado del arte de las simulaciones de telas, se debe tener en cuenta que no hay una gran solución que esté por sobre todas las demás y sirva para todos los casos de uso. Dependiendo del contexto en donde se quiera aplicar la solución (una animación, un videojuego o una aplicación de realidad virtual) puede requerirse utilizar distintos algoritmos de colisiones, volumen de datos, nivel de precisión e incluso hardware.

2.1. Programas profesionales

2.1.1. Animación

Es el sector del entretenimiento, específicamente la industria cinematográfica la principal responsable de los avances en el estado del arte, sin embargo, estas suelen hacer uso de tecnologías *made in house* y, por lo tanto, no se sabe mucho de su funcionamiento interno.

Existen programas profesionales que sirven como punto de encuentro para las múltiples tecnologías necesarias a la hora de generar animaciones regidas por reglas físicas, que incluyen estrategias y algoritmos para simular el comportamiento de telas. Algunos de los más notables en esta área son *Houdini*, utilizado en películas de Marvel como Ant-man & The Wasp (2018) o de Disney como Zootopía (2016) y en series como Game of Thrones (temporada 6, 2017); *Maya Autodesk* en películas de Marvel como Avengers: Infinity War (2018) o de Sony Pictures como Spider-Man: Into the Spider-Verse (2018); y *Fizt*, la herramienta creada por Pixar utilizada en sus películas más actuales como Coco (2017) y Onward (2020).

2.1.2. Videojuegos

En menor medida y más enfocados en el área de los videojuegos, existen otras alternativas, en su mayoría tecnologías creadas por cada gran productora, como son *Anvil Engine* de Ubisoft con el que se realizó Assassin's Creed Odyssey (2018); *SPACKLE* de Sucker Punch con el que se realizó Ghost of Tsushima (2020); *RAGE* de Rockstar Games con el que realizó Red Dead Redemption 2 (2018) o *Decima Engine* de Guerrilla Games con el que se realizó Horizon Forbidden West (2022).

Existen además otras opciones que son de libre uso como *Unreal Engine* con el que se está realizando Senua's Saga: Hellblade II (2023) y *Unity* con el que se realizó Genshin Impact (2020). Ambos motores incluyen las herramientas necesarias, pero básicas para la simulación

de telas (modelo de masas y resortes), estas pueden ser extendidas o utilizar *plugins* externos que otorguen las funcionalidad o algoritmos específicos que se deseen utilizar, por ejemplo, *uDraper* para Unreal Engine que permite simulaciones de tela en tiempo real. Esta práctica es usual para adaptar los motores a las exigencias de cada videojuego.

2.1.3. Diseño

Finalmente para el caso del diseño de vestuario con fines industriales o para ser exportados a otros motores, se tienen programas de pago donde los más notables son *Marvelous Designer* utilizado para la creación de vestuario del videojuego Cyberpunk 2077; *CLO* utilizado por empresas como Levis o *Browzwear* utilizado por empresas como Adidas.

2.2. Soluciones en CPU

Distintas soluciones para cada problema relacionado a las simulaciones de telas se han presentado los últimos años, si bien no se puede asegurar que estas estrategias sean las utilizadas por las grandes compañías que lideran el mercado (dado que estas no son públicas), es un hecho que aportan mejoras significativas a la simulación ya sea en velocidad o precisión.

Eulerian-on-Lagrangian Cloth Simulation (2018) [1] resuelve el problema de generar una simulación realista cuando la tela interactúa con bordes afilados como puede ser el extremo de una mesa, añadiendo vértices eulerianos que se procesan con un mayor grado de libertad que el resto de la tela comúnmente manejado con vértices lagrangianos. Con un objetivo similar *An improved mean curvature-based bending model for cloth simulation* (2021) [2] divide la tela en subzonas y utiliza su curvatura promedio para generar las fuerzas de flexión que comúnmente son generadas enlaces (resortes) consiguiendo un resultado realista sin aumentar significativamente el costo computacional.

Publicaciones como *An Implicit Frictional Contact Solver for Adaptive Cloth Simulation* (2018) [3] se han enfocado en mejorar el tratamiento de la fricción entre tela-tela y tela-objeto, haciendo uso principal de las velocidades para realizar sus cálculos y una estrategia de refinamiento de nodos dinámicos para trabajar los contactos en cualquier sitio de la tela.

Además del modelo comúnmente utilizado de masas y resortes donde se describe la tela como una superficie homogénea de partículas con enlaces, existe otro modelo, el llamado modelo de hilos, que busca representar de manera más fiel el comportamiento de la tela teniendo en consideración la forma en que están entretejidos sus hilos. Dentro de este campo *Mixing Yarns and Triangles in Cloth Simulation* (2020) [4] propone mezclar ambos, haciendo uso del modelo tradicional para las zonas simples y del modelo de hilos solo para las más complejas, ya que este resulta más caro computacionalmente.

2.3. Soluciones en GPU

Al igual que en otras áreas, no ha sido hasta esta última década que los avances en hardware han permitido y vuelto de interés la computación paralela en la simulación de telas, desarrollando no solo versiones paralelas de los algoritmos ya existente, sino también, generando nuevas soluciones pensadas específicamente para las condiciones del hardware existente.

La idea de que la complejidad de la tela no es lineal, por lo tanto, su solución tampoco debe serla no es nueva, sin embargo, debido al gran coste computacional de estas no han sido opciones populares en ciertos ámbitos que requieren velocidad. La publicación *Parallel Multigrid for Nonlinear Cloth Simulation* (2018) [5], propone utilizar un modelo multigrilla para enfrentar el problema, dividiendo la tela según el nivel de complejidad que requiere y aprovechando los múltiples núcleos de la GPU (o CPU si se prefiere), realizar los cálculos de forma paralela, logrando una solución no lineal en tiempo competente.

I-Cloth: Incremental Collision Handling for GPU-Based Interactive Cloth Simulation (2018) [6] presenta un algoritmo de detección colisión basado en hashing espacial (bajo la idea de que entre frames las posiciones no suelen tener grandes cambios) que permite distribuir la carga computacional entre los núcleos de la GPU junto a un algoritmo de resolución de colisiones de gran precisión, también paralelo, que les permite tomar tiempos de iteración altos y disminuir el tiempo de procesado global. *P-Cloth: Interactive Complex Cloth Simulation on Multi-GPU Systems using Dynamic Matrix Assembly and Pipelined Implicit Integrators* (2020) [7] se mantiene en la idea de generar soluciones pensadas para GPU, pero va un paso más allá logrando desarrollar algoritmos integración y colisiones en paralelo que hacen uso de matrices sparse para tratar volúmenes de datos que no se podrían procesar utilizando solo una GPU, enfocándose en minimizar las lecturas necesarias y cuellos de botella que se producen en la comunicación GPU-GPU.

Time-Domain Parallelization for Accelerating Cloth Simulation (2018) [8] toma un enfoque distinto al usual, en lugar de paralelizar los algoritmos respecto a los datos, lo hace respecto al tiempo. Logra esto realizando una simulación inicial en baja resolución, la cual es tomada como base para dirigir la simulación final de forma paralela en la GPU, esto con el fin de evitar los cuellos de botella relacionados a la comunicación de datos.

GPU-Based Simulation of Cloth Wrinkles at Submillimeter Levels (2021) [9] también opta por un enfoque poco común, se postula que al tratarse con altos volúmenes de datos (millones de triángulos), los suficientes para estar simulando a niveles submilimétricos, resulta más conveniente utilizar una grilla estructurada que una triangulación debido a la gran compatibilidad de estas con la GPU y que a resoluciones tan altas el resultado no será distinto al ojo humano. La publicación logra resolver el problema utilizando esta representación de las telas mediante un método de descenso basado en bloques que les permite realizar pocos accesos a memoria.

2.4. Machine Learning

Durante los últimos años, el uso de *machine learning* para la resolución de distintas problemáticas ha ido en aumento y las simulaciones físicas, particularmente aquellas relacionadas con las telas, no han sido una excepción.

Las técnicas para aplicar *machine learning* a la simulación son diversas, en *Hierarchical Cloth Simulation using Deep Neural Networks* (2018) [10] se propone combinar la simulación clásica basada en físicas con el uso de redes neuronales profundas, haciendo uso de la simulación por físicas más costosa solo en las secciones más simples y dejando aquellas que

requieren gran resolución a la red neuronal que tiene un costo más bajo, pero a su vez menor precisión; *Efficient Cloth Simulation using Miniature Cloth and Upscaling Deep Neural Networks* (2019) [11] usa una estrategia ya popular en el procesamiento de imágenes y videos, entrenar una red capaz de recibir versiones de menor tamaño de ciertos datos (en este caso telas) y aumentarles su resolución, de esta forma se procesa una versión miniaturizada de la tela con una cantidad considerablemente menor de puntos mediante una simulación física convencional y luego se aumenta utilizando la red neuronal; *Swish: Neural Network Cloth Simulation on Madden NFL 21* (2021) [12], desarrollado por la compañía de videojuegos EA logra simular mediante una red neuronal a partir de esqueletos de personajes su vestimenta en tiempo real evitando usar simulaciones físicas para su caso de uso particular.

Capítulo 3

Trabajo previo

Previo a este trabajo y con el fin original de comprender la complejidad de las simulaciones de tela, se realizan dos proyectos que servirán como base para los estudios presentado en este informe y el desarrollo de un programa de simulación robusto.

3.1. Simulación en navegador web

El primer proyecto realizado consiste en un simulador web concebido para responder las preguntas "¿por qué no existen probadores virtuales online?" y "¿es caro computacionalmente?". Este fue escrito en Javascript haciendo uso de la librería Three.js.

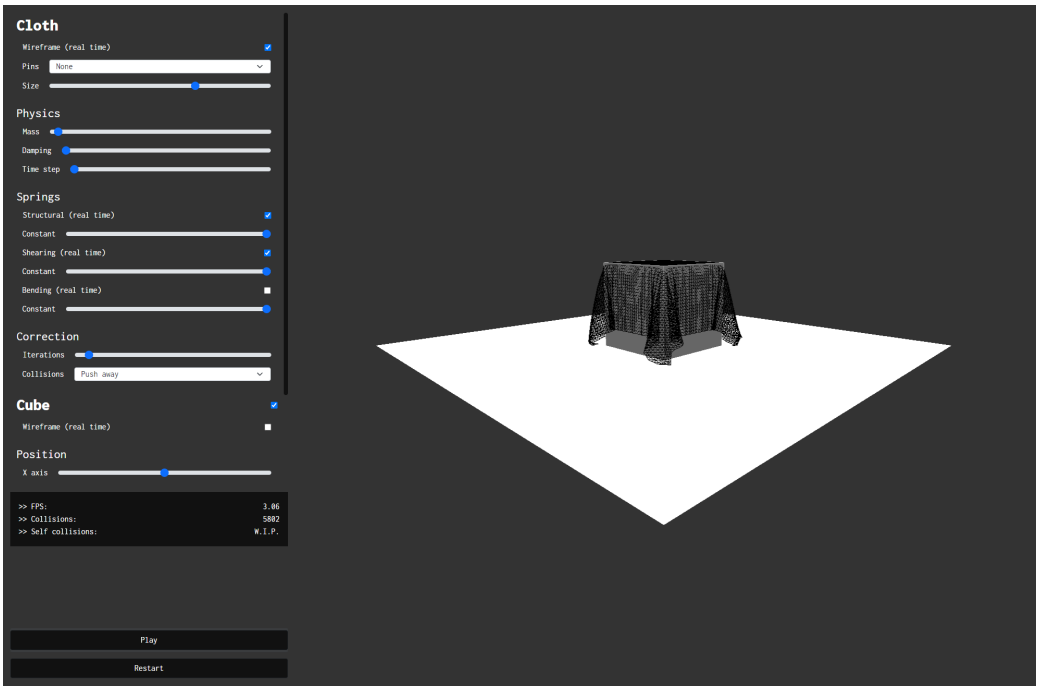


Figura 3.1: Simulador web escrito en Javascript.

Mediante un modelo de masa - resorte con restricciones (Capítulo 4.3.1.1) e integración de Verlet (Capítulo 4.4.1) se simula la caída de una tela sobre un cubo. Las colisiones son detectadas mediante fuerza bruta y manejadas empujando las partículas hacia fuera de las caras que atraviesan (Capítulo 4.5.2).

A partir de este proyecto se obtienen las primeras nociones sobre las distintas secciones que componen una simulación y la dificultad de calcular esta en tiempo real cuando se tiene una alta densidad de partículas. El rendimiento obtenido (detalles del hardware utilizado en Capítulo 5) como se muestra en la Figura 3.2 y Tabla 3.1 es claramente inviable en un escenario real, sin embargo, se tiene un amplio margen de mejora partiendo por la optimización de las colisiones donde el uso de fuerza bruta es la peor estrategia disponible.

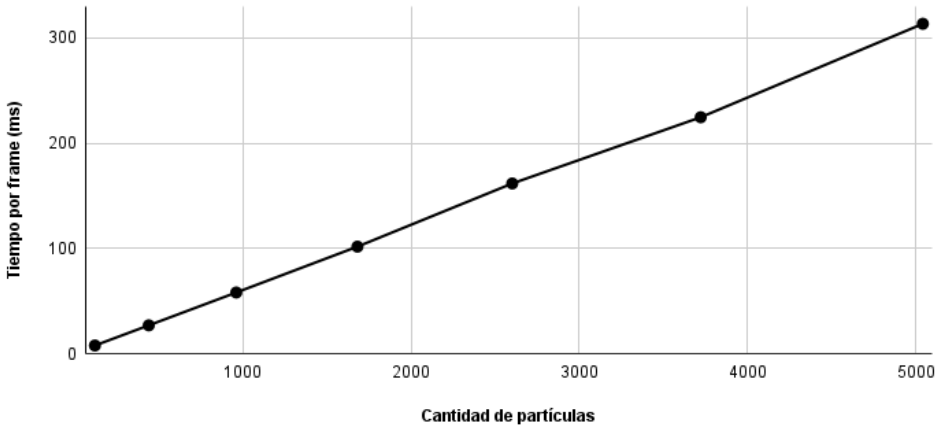


Figura 3.2: Rendimiento del simulador web después de 600 frames.

3.2. Simulación utilizando GPU

Del primer proyecto se generan la preguntas "¿se puede paralelizar la simulación?" y "¿qué tanto mejorará el rendimiento si se hace uso de una GPU?". Para responder esto se replica la solución web utilizando el lenguaje C++ por su velocidad, la API gráfica OpenGL para generar la imagen y CUDA para programar en la GPU.

El programa resultante tiene un rendimiento mucho mayor a la versión web como se observa en la Tabla 3.1 y Figura 3.3, a pesar de hacer uso de los mismos métodos ineficientes para colisiones.

Tabla 3.1: Rendimiento de los simuladores por cantidad de partículas después de 600 frames.

	CPU	**GPU**
121	008.16 ms	002.09 ms
441	027.37 ms	001.94 ms
961	058.94 ms	001.64 ms
1681	102.11 ms	001.66 ms
2601	163.16 ms	001.67 ms
3721	226.32 ms	001.65 ms
5041	315.79 ms	001.67 ms

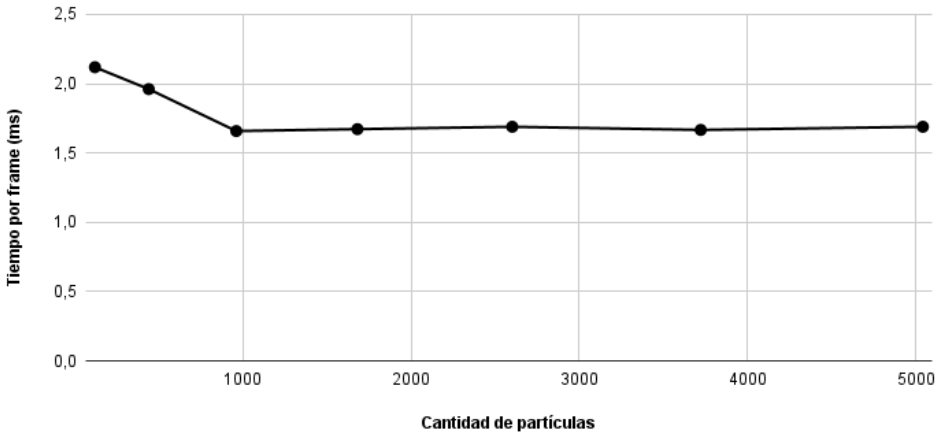


Figura 3.3: Rendimiento del simulador con GPU después de 600 frames.

Sin embargo, haber realizado una transcripción directa de Javascript a C++ sin haber considerado las particularidades del lenguaje lo vuelve mucho más propenso a inestabilidades debido a problemas de precisión, como se observa en la Figura 3.4.

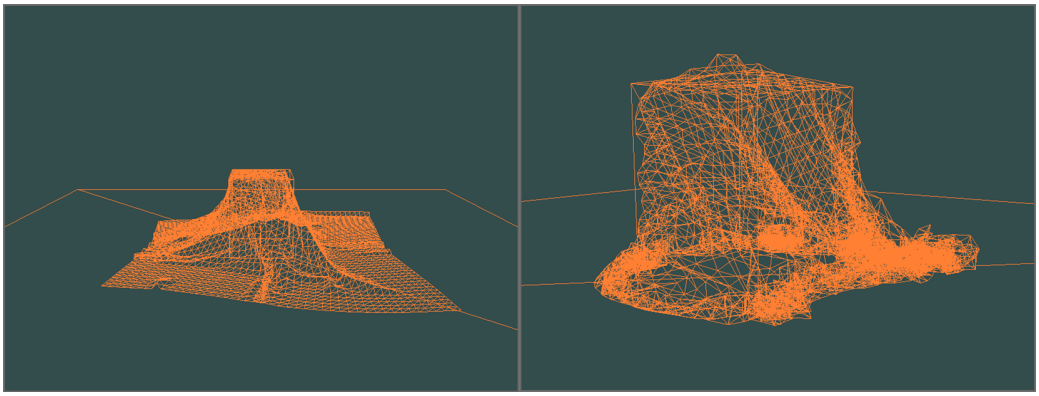


Figura 3.4: Simulador con uso de GPU. Inestabilidad en el sistema.

Si bien la mejora es notable, solo se logra paralelizar la integración y las colisiones, aplicando las restricciones del modelo desde la CPU debido a la naturaleza de su cálculo. Este problema se puede solucionar fácilmente utilizando fuerzas en lugar de restricciones (Capítulo 4.3.1.2).

De este proyecto se obtiene el conocimiento sobre C++ y OpenGL que será la base para la creación del programa de simulación presentado en este trabajo.

Capítulo 4

Teoría e implementación

El siguiente capítulo introduce a la teoría necesaria para lograr la simulación de una tela. Se parte explicando cómo describir computacionalmente una tela, para luego entrar en el ciclo de simulación y finalmente ahondar en los distintos algoritmos que lo componen. Esto es acompañado de observaciones tomadas durante la implementación de cada paso de la simulación.

Todas las imágenes de simulación en este capítulo (y todo este trabajo) se han generado utilizando el programa visto en más detalle en el capítulo 6.

4.1. Representación

Como es común en las ciencias de la computación, para poder representar un objeto continuo se requiere poder discretizarlo de algún modo. En la realidad los objetos están compuestos por partículas unidas entre sí por fuerzas y se puede tomar esta misma noción, a una escala computacionalmente asequible, para representar un trozo de tela.

4.1.1. Discretización espacial

Supondremos que una tela es un objeto plano de forma cuadrada, compuesto por partículas las cuales se encuentran a una cierta distancia unas de otras. En la práctica esto significa que la tela está representada por *vértices* (partículas) y *arcos* formando una malla triangular como se observa en la Figura 4.1.

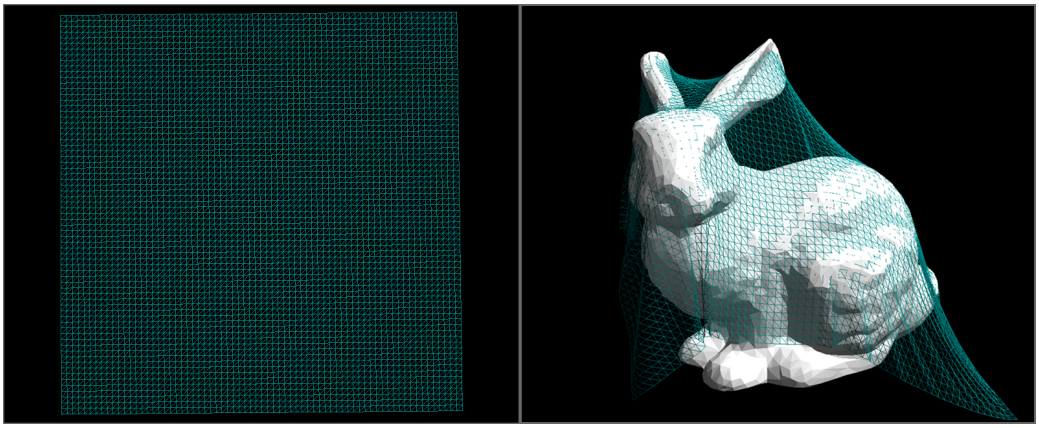


Figura 4.1: Tela en su estado inicial y dejada caer sobre el modelo 3D *Stanford bunny*.

Se debe notar que la disposición de los arcos no es necesariamente una representación de fuerzas entre partículas, si no una decisión de diseño a la hora de representar la tela. Una malla triangular es sencilla de dibujar en pantalla mediante APIs gráficas a la vez que permite utilizar ciertas estructuras de datos y algoritmos sobre ella. Si bien existen trabajos que utilizan mallas cuadradas [13], estos pueden ser adaptados de forma directa a mallas triangulares y por ende, hacer uso de las mismas estrategias vistas en este informe.

Con la tela discretizada, el objetivo de la simulación es saber dónde se encuentra cada partícula en cada instante de tiempo, para esto es preciso conocer la *posición*, *masa* y *fuerza* ejercida sobre cada una de estas. Es posible, en lugar de utilizar la fuerza ejercida sobre una partícula, utilizar su *velocidad* o *aceleración* ya que se puede pasar fácilmente de una a otra como se muestra en las Ecuaciones 4.1.

$$\begin{aligned} f &= ma \\ a &= \frac{\Delta v}{\Delta t} \end{aligned} \tag{4.1}$$

Donde f corresponde a la fuerza, m a la masa, a a la aceleración, v a la velocidad y t al tiempo.

Nuevamente, manejar fuerzas en lugar de velocidades o aceleraciones es una decisión de diseño que facilitará la integración con ciertos algoritmos y modelos que se verán más adelante.

4.1.2. Discretización temporal

Teniendo la tela en un estado inicial (posición y fuerzas de sus partículas, ya que la masa es constante) se busca saber cuál será su estado en un próximo *paso de tiempo*, pero se debe definir qué tamaño de paso usar, ya que la naturaleza continua del tiempo nos exige discretizarlo para trabajar con él. Comúnmente se eligen pasos de tiempo tales que permitan calcular 24, 30 o 60 siguientes estados por cada segundo de simulación (no necesariamente de tiempo real), es decir, generar una animación a 24, 30 o 60 *frames por segundo* (fps).

Tabla 4.1: Pasos de tiempo para los frames por segundo más comunes.

FPS	Δt
24	41.6 ms
30	33.3 ms
60	16.6 ms

A diferencia de cuando se tiene tiempo suficiente para calcular cada siguiente estado de la simulación y luego unirlos para generar una animación, realizar esta en tiempo real es un desafío completamente distinto que escala de forma no lineal con el tamaño de la tela. El principal problema es que no se puede conocer de antemano el tiempo que tardará la simulación en realizar todos los cálculos necesarios para obtener el siguiente estado y por ende, no se puede definir su paso de tiempo. Comúnmente se utiliza para calcular el siguiente estado un paso de tiempo equivalente a lo que haya tardado en calcularse el estado actual, que si bien es un método sencillo para definirlo, en caso de que el tiempo sea muy pequeño o muy grande, puede llevar a problemas de precisión o inestabilidades numéricas que afecten la simulación.

Utilizar un paso de tiempo dinámico permite realizar animaciones de la simulación en tiempo real, pero no nos asegura una cantidad estable de fps a lo largo de esta ni tampoco la precisión y estabilidad que se obtienen de calcular con un paso de tiempo fijo. Esto último depende en gran medida de los modelos y algoritmos que se utilicen para calcular los estados de la simulación y son estos los que a fin de cuentas permiten utilizar pasos de tiempo más grandes o más pequeños.

4.1.3. Formalización

Hasta ahora se han mencionado algunos conceptos que serán largamente utilizados en este informe y por tanto es preciso ahondar en ellos para no causar confusión con definiciones que puedan encontrarse en otros trabajos.

Se referirá a *simulación* como el cálculo que realizamos para obtener los subsiguientes estados de una tela y a *animación* como el dibujado en pantalla de los estados de esta en sus correspondientes instantes de tiempo.

Una *partícula* P se define como un conjunto de posición, masa y fuerza. La combinación de estas propiedades se denomina *estado de la partícula*.

$$P_i = \langle \vec{p_i}, m_i, \vec{f_i} \rangle$$

Donde $\vec{p_i}$ corresponde a la posición de la partícula i, m_i a su masa y $\vec{f_i}$ a las fuerzas ejercidas sobre ella.

Una *tela* T se define como un conjunto de partículas. La combinación del conjunto de estados de partículas que la componen se denomina *estado de la tela*.

$$T = \langle P_0, ..., P_{n-1} \rangle$$

Donde n corresponde a la cantidad de partículas que componen la tela.

4.1.4. Estructura de datos: Half-Edge

Para almacenar el objeto que representa la tela, se hace uso de la estructura de datos Half-Edge como se observa en ejemplo de la Figura 4.2. Esto nos permitirá tener un acceso rápido a la vecindad de cada partícula, lo que resulta útil a la hora de calcular fuerzas.

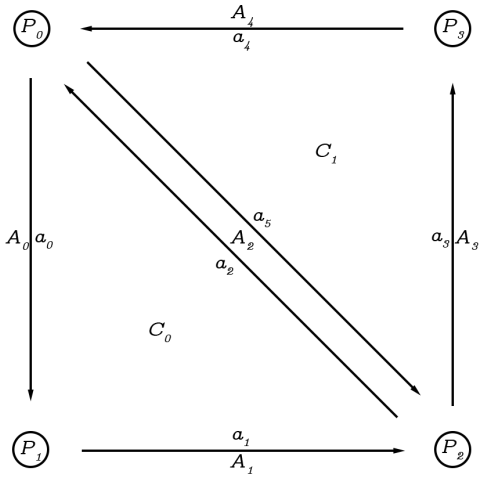


Figura 4.2: Ejemplo de Half-Edge en una malla simple de cuatro vértices.

A cada partícula se le asocia una posición, masa y fuerzas ejercidas sobre ella. Con objetivo de realizar ciertos cálculos como la velocidad y aceleración, se le asocia también su *posición anterior*.

Tabla 4.2: Partículas para la malla de ejemplo ejemplo de la Figura 4.1.

Partícula	Posición	Masa	Fuerza	Posición anterior
P_0	$\vec{p_0}$	m_0	$\vec{f_0}$	$\vec{p_{a0}}$
P_1	$\vec{p_1}$	m_1	$\vec{f_1}$	$\vec{p_{a1}}$
P_2	$\vec{p_2}$	m_2	$\vec{f_2}$	$\vec{p_{a2}}$
P_3	$\vec{p_3}$	m_3	$\vec{f_3}$	$\vec{p_{a3}}$

A cada cara y arco se le asocia uno de los medio arco que la componen.

Tabla 4.3: Caras para la malla de ejemplo ejemplo de la Figura 4.1.

Cara	Medio arco
C_0	a_0
C_1	a_3

Tabla 4.4: Arcos para la malla de ejemplo ejemplo de la Figura 4.1.

Arco	Medio arco
A_0	a_0
A_1	a_1
A_2	a_2
A_3	a_3
A_4	a_4

A cada medio arco se le asocia una partícula de origen, su cara incidente, su medio arco gemelo, su medio arco anterior y su medio arco siguiente.

Tabla 4.5: Medio arcos para la malla de ejemplo ejemplo de la Figura 4.1.

Medio arco	Origen	Cara incidente	Gemelo	Anterior	Siguiente
a_0	P_0	C_0	-	a_2	a_1
a_1	P_1	C_0	-	a_0	a_2
a_2	P_2	C_0	a_5	a_1	a_0
a_3	P_2	C_1	-	a_5	a_4
a_4	P_3	C_1	-	a_3	a_5
a_5	P_0	C_1	a_2	a_4	a_3

4.2. Ciclo de la simulación

Una simulación de telas involucra tres etapas que se repiten en ciclos para obtener constantemente el siguiente estado de esta durante su ejecución. No tienen un orden específico y dependen del modo en que se calcule el comportamiento del material, el cual se modela mediante fuerzas y restricciones que definen cómo reacciona la tela ante los cambios. Estas etapas son:

- *Modelo:* etapa dedicada a la aplicación de fuerzas y restricciones que definen el comportamiento de la tela.
- *Integración:* etapa dedicada al cálculo de la posición siguiente de las partículas que componen la tela.
- *Colisión:* etapa dedicada a la detección y corrección de colisiones entre Tela - Objeto y Tela - Tela.

Dependiendo del modelo utilizado puede ser necesario ejecutar más de una vez las etapas de modelo y/o colisión por cada estado de la simulación (múltiples *iteraciones*) como se muestra en el Código de ejemplo 4.1. En estos casos resulta más efectivo [14] dividir el cálculo del estado de la simulación en *subciclos* ejecutando solo una vez cada etapa como se muestra en el Código de ejemplo 4.2.

Código 4.1: Ejemplo de ciclo con iteraciones.

```
1 while simulación_en_curso:
2     etapa_integración(Δt)
3
4     for iteraciones:
5         etapa_modelo(Δt)
6         etapa_colisión()
```

Código 4.2: Ejemplo de ciclo con subciclos.

```
1 while simulación_en_curso:
2     for subciclos:
3         etapa_integración(Δt / subciclos)
4         etapa_modelo(Δt / subciclos)
5         etapa_colisión()
```

Esta estrategia de subciclos permite además, utilizar pasos de tiempo que normalmente serían incompatibles con ciertos modelos por su tamaño. Disminuyendo el paso de tiempo utilizado por cada subciclo se hace posible su cálculo, a la vez que conserva el paso de tiempo total del ciclo para mantener una animación estable.

4.3. Modelos de fuerzas y restricciones

Un *modelo* es el conjunto de algoritmos y reglas que definen el comportamiento de la tela. Puede definir cómo y con qué magnitud actúan fuerzas sobre cada partícula que compone la tela, imponer restricciones o aplicar modificaciones al cumplir ciertas condiciones.

Los modelos explicitan cómo debe reaccionar la tela ante tres tipos de deformaciones principales: estiramiento, cizallamiento y plegado. Estas deformaciones se observan en la Figura 4.3.

Figura 4.3: Tela en reposo, deformación de estiramiento, deformación de cizallamiento y deformación de plegado respectivamente.

Se entrará en detalles sobre dos modelos relevantes por el gran uso que han tenido y las bases que han sentado para la creación de nuevos modelos. El primero, *masa - resorte*, se suele utilizar por su sencillez y bajo coste computacional en el ámbito de los videojuegos; el segundo, *mecánicas continuas*, ha sido enfocado al ámbito de la animación por sus resultados más realistas, pero con un alto coste computacional.

4.3.1. Masa - Resorte

Como su nombre lo indica, el modelo se basa en considerar la tela como un conjunto de masas (partículas) unidas entre sí por resortes. Si dos partículas se alejan demasiado, el resorte se encargará de acercarlas y si por el contrario se acercan demasiado, se encargará de alejarlas.

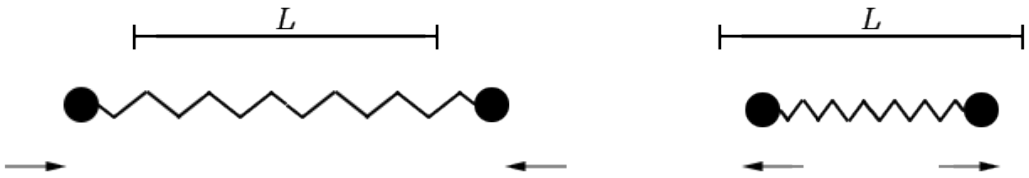


Figura 4.4: Efecto de un resorte entre dos partículas donde L es la distancia natural entre estas.

Por cada tipo de deformación, el modelo añade un tipo de resorte para contrarrestarla. Para evitar el estiramiento, se añaden resortes entre una partícula y sus vecinas inmediatas en la horizontal y vertical; para el cizallamiento, se añaden resortes entre una partícula y sus vecinas inmediatas en la diagonal; para el plegado, se añaden resortes entre una partícula y sus vecinas a una partícula de distancia en la horizontal y vertical. La disposición de los resortes se observa en la Figura 4.5.

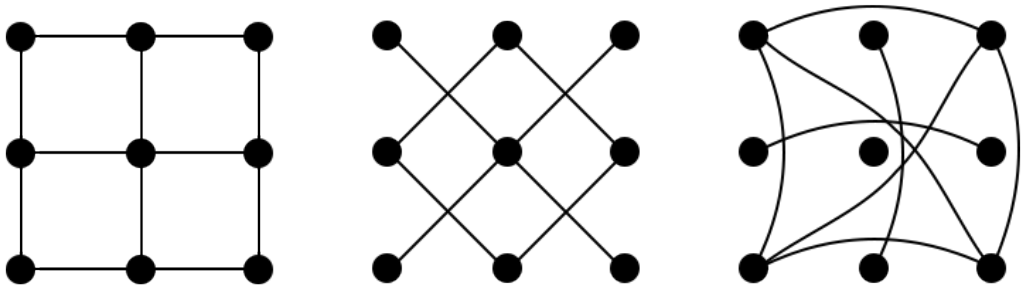


Figura 4.5: Resortes de estiramiento, de cizallamiento y plegado respectivamente.

4.3.1.1. Aplicación de restricciones

Una manera sencilla de aplicar los resortes es en forma de restricción: una vez movida la tela en un ciclo, exigir a cada resorte, uno por uno, que se estire o contraiga hasta volver a su largo inicial.

Si un resorte de largo inicial L une a dos partículas P_i y P_j, la corrección a la posición de estas últimas se calcula como se muestra en las Ecuaciones 4.2.

$$\begin{aligned}\Delta\vec{p_i} &= \frac{1}{2}(\|\vec{p_j} - \vec{p_i}\| - L)\frac{\vec{p_j} - \vec{p_i}}{\|\vec{p_j} - \vec{p_i}\|} \\ \Delta\vec{p_j} &= \frac{1}{2}(\|\vec{p_i} - \vec{p_j}\| - L)\frac{\vec{p_i} - \vec{p_j}}{\|\vec{p_i} - \vec{p_j}\|}\end{aligned} \tag{4.2}$$

El *amortiguamiento* (pérdida de energía) del sistema simula disminuyendo en una fracción la corrección de las partículas, como se muestra en las Ecuaciones 4.3.

$$\begin{aligned}\Delta\vec{p_i} &= -k_d\frac{1}{2}(\|\vec{p_j} - \vec{p_i}\| - L)\frac{\vec{p_j} - \vec{p_i}}{\|\vec{p_j} - \vec{p_i}\|} \\ \Delta\vec{p_j} &= -k_d\frac{1}{2}(\|\vec{p_i} - \vec{p_j}\| - L)\frac{\vec{p_i} - \vec{p_j}}{\|\vec{p_i} - \vec{p_j}\|}\end{aligned} \tag{4.3}$$

Donde k_d es el coeficiente de amortiguamiento, un valor entre 0 y 1.

Utilizar este método conlleva ciertas desventajas, cada vez que se aplica la restricción sobre un resorte, se corre el riesgo de modificar resortes vecinos que ya han sido corregidos como se observa en la Figura 4.6. Por esta razón se debe aplicar la restricción reiteradas veces hasta converger en resultados visualmente correctos.

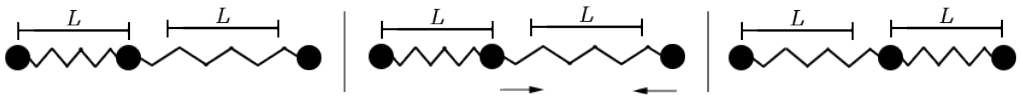


Figura 4.6: Ejemplo de violación de una restricción al aplicar otra.

Es necesario encontrar un balance en la cantidad de veces que se aplique la restricción. Si esta es baja, los resortes no lograrán volver suficientemente rápido a su largo original y la tela se verá anormalmente estirada, como se observa en la Figura 4.7; y si es alta, el coste computacional también será alto.

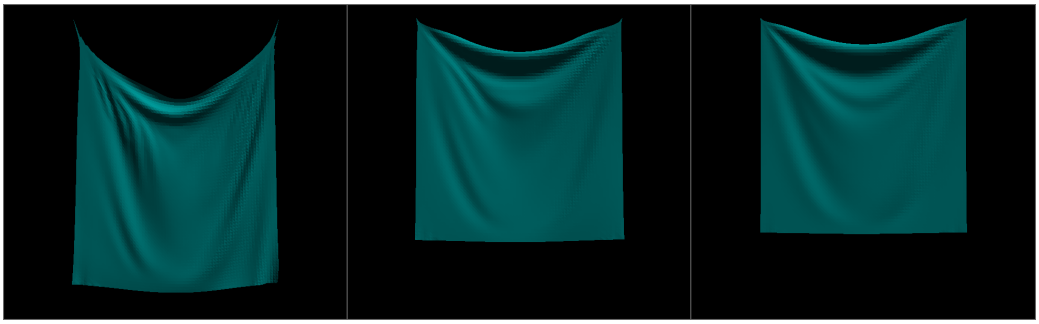


Figura 4.7: Estiramiento de la tela con 1, 5 y 10 iteraciones de restricción.

Una tela con mayor cantidad de partículas implica una mayor cantidad de resortes, los que a su vez requieren iterar más veces las restricciones para obtener resultados visualmente correctos. Un ejemplo de esto se observa en la Figura 4.8 donde para una misma cantidad de aplicación de restricciones al aumentar la cantidad de partículas la tela se estira cada vez más.

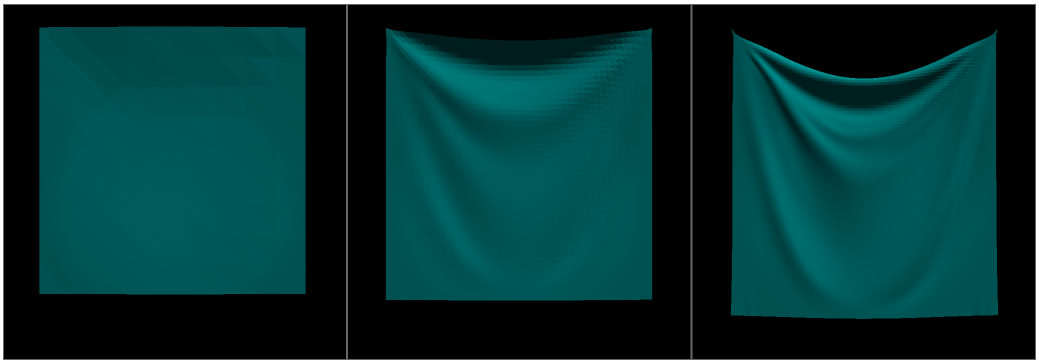


Figura 4.8: Telas con 100, 2500 y 10000 partículas y 10 subciclos con aplicación de restricciones.

Debido a esto y dependiendo del costo computacional que se esté dispuesto a pagar, puede ser conveniente usar subciclos o iteraciones para controlar la cantidad de veces que se calculan las etapas de integración y colisiones, como se observa en los Códigos 4.3 y 4.4.

Código 4.3: Ciclo de modelo masa - resorte con restricciones. Uso de iteraciones cuando la cantidad de veces que deben aplicarse es alta.

```
while simulación_en_curso:
    etapa_integración(Δt)

    for iteraciones:
        etapa_modelo(Δt)

    etapa_colisión()
```

Código 4.4: Ciclo de modelo masa - resorte con restricciones. Uso de subciclos cuando la cantidad de veces que deben aplicarse es baja.

```
while simulación_en_curso:
    for subciclos:
        etapa_integración(Δt / subciclos)
        etapa_modelo(Δt / subciclos)
        etapa_colisión()
```

Otra desventaja presente es la dificultad de este método para ser paralelizable, debido a que el efecto de un resorte puede afectar a sus vecinos, no es posible resolver todas las restricciones al mismo tiempo. Si bien podría realizarse la corrección en grupos de resortes

siguiendo algún patrón, sigue sin ser la paralelización completa que sí pueden lograr otros métodos como el que se verá en la siguiente sección.

4.3.1.2. Aplicación de fuerzas

En lugar de exigir a los resortes uno a uno que muevan directamente las partículas que conectan, se calcula la fuerza que ejercen sobre estas para delegar la modificación de su posición a la etapa de integración. De esta forma cada partícula obtiene una fuerza que representa la interacción con todos los resortes que la conectan, evitando tener que realizar múltiples iteraciones de corrección como sucede con el método anterior.

Si un resorte de largo inicial L une a dos partículas P_i y P_j, el aporte a la fuerza ejercida sobre estas se calcula siguiendo la *ley de Hooke* como se muestra en las Ecuaciones 4.4

$$\begin{aligned} \Delta\vec{f_i} &= k_s(\|\vec{p_j} - \vec{p_i}\| - L)\frac{\vec{p_j} - \vec{p_i}}{\|\vec{p_j} - \vec{p_i}\|} \\ \Delta\vec{f_j} &= k_s(\|\vec{p_i} - \vec{p_j}\| - L)\frac{\vec{p_i} - \vec{p_j}}{\|\vec{p_i} - \vec{p_j}\|} \end{aligned} \tag{4.4}$$

Donde k_s es la constante elástica del resorte.

El amortiguamiento del sistema se simula añadiendo una fuerza que se oponga al movimiento, como se muestra en las Ecuaciones 4.5.

$$\begin{aligned} \Delta\vec{f_i} &= k_d(\vec{v_j} - \vec{v_i}) \\ \Delta\vec{f_j} &= k_d(\vec{v_i} - \vec{v_j}) \end{aligned} \tag{4.5}$$

Donde k_d es el coeficiente de amortiguamiento, un valor entre 0 y 1.

Las velocidades para calcular el amortiguamiento no se guarda dentro de la estructura Half-Edge utilizada, se calculan utilizando la posición previa y el paso de tiempo.

$$\begin{aligned} \vec{v_i} &= \frac{\vec{p_i} - \vec{p_{prev_i}}}{\Delta t} \\ \vec{v_j} &= \frac{\vec{p_j} - \vec{p_{prev_j}}}{\Delta t} \end{aligned} \tag{4.6}$$

Donde $\vec{p_{prev}}$ es la posición previa de la partícula y Δt el paso de tiempo utilizado.

Con este método no se generan diferencias significativas en la tela al aumentar la cantidad de partículas que la componen como ocurre con el uso de restricciones. Esto se observa en la Figura 4.9.

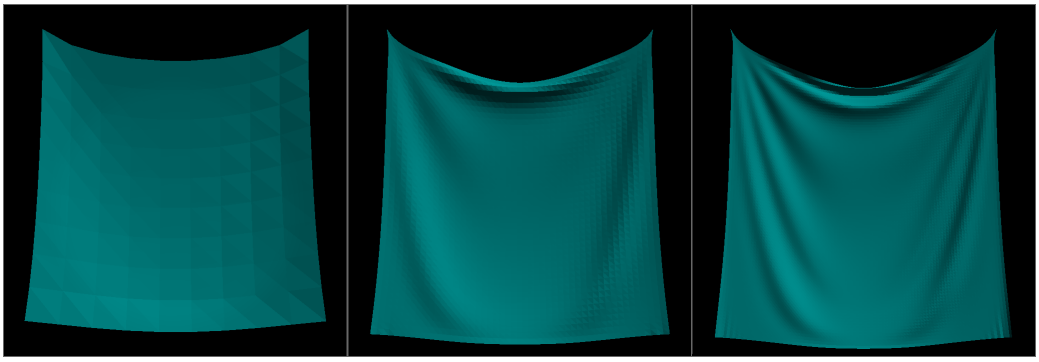


Figura 4.9: Telas con 100, 2500 y 10000 partículas y 10 subciclos con cálculo de fuerzas.

Debido a que no se requiere calcular más de una vez cada etapa, se hace uso de subciclos como se observa en el Código 4.5.

Código 4.5: Ciclo de modelo masa - resorte con cálculo de fuerzas.

```
while simulación_en_curso:
    for subciclos:
        etapa_modelo(Δt / subciclos)
        etapa_integración(Δt / subciclos)
        etapa_colisión()
```

A diferencia del método con restricciones, cada resorte puede realizar su cálculo de forma independiente y luego sumarlo al total de fuerzas sobre cada partícula, por lo que es sumamente paralelizable. Sin embargo, no está exento de desventajas, las variables k_s y k_d son de naturaleza empírica, por tanto se deben encontrar manualmente valores que generen una buena representación visual para cada configuración sin provocar inestabilidad numérica.

4.3.2. Mecánicas continuas

El modelo de masa - resorte es sencillo de implementar y obtiene resultados visualmente aceptables. Sin embargo, depende de la forma en que se conecten los resortes y los coeficientes elegidos, es decir, se basa en una decisión de diseño y no en las físicas reales de la tela. Esto genera un problema a la hora de representar materiales con comportamientos específicos. El modelo de mecánicas continuas [15] representa la tela ciñéndose de mejor forma a sus físicas internas por lo que logra resultados más realistas.

Sea la tela en su estado inicial, sin fuerzas aplicadas, se mapean los triángulos que componen su malla en 3D (x, y, z) a un espacio en 2D (u, v). El mapeo inverso de 2D a 3D se define mediante una función $W(u, v)$.

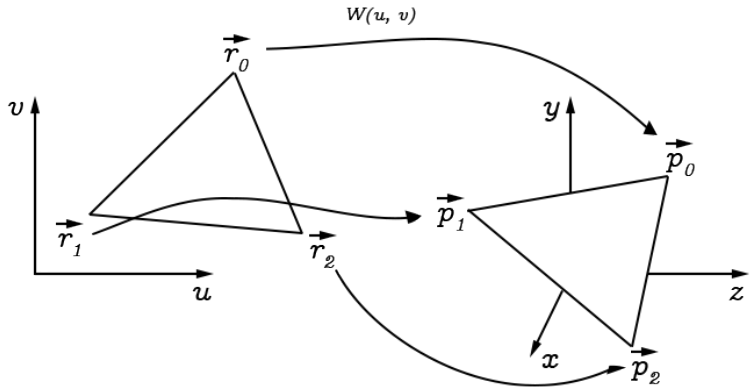


Figura 4.10: Triángulos en espacio 2D (u, v) y 3D (x, y, z).

Un triángulo definido por partículas con posiciones $\vec{p_0}$, $\vec{p_1}$ y $\vec{p_2}$ en el espacio (x, y, z), estará definido por las posiciones $\vec{r_0}$, $\vec{r_1}$ y $\vec{r_2}$ respectivamente en el espacio (u, v), como se observa en la Figura 4.10.

No es necesario conocer explícitamente la función $W(u, v)$ ya que solo se hace uso del estado inicial mapeado para compararlo con la deformación que tenga la tela durante la simulación, análogo al largo inicial de los resortes utilizado para calcular las fuerzas ejercidas por estos.

En este trabajo la tela en estado inicial se encuentra paralela al plano XZ a una altura L de este. Se utiliza el mismo plano que contiene la tela para mapear al espacio (u, v) prescindiendo de la coordenada $\hat{y}$.

$$M(x\hat{x} + y\hat{y} + z\hat{z}) = x\hat{u} + z\hat{v} \tag{4.7}$$

Donde M es la función que mapea del espacio (x, y, z) al espacio (u, v).

A partir de las coordenadas (x, y, z) y (u, v) de cada triángulo, se definen las siguientes variables.

$$\begin{aligned}
\Delta x_1 &= \vec{p_1} - \vec{p_0} \\
\Delta x_2 &= \vec{p_2} - \vec{p_0} \\
\Delta u_1 &= \vec{r_{1\hat{u}}} - \vec{r_{0\hat{u}}} \\
\Delta u_2 &= \vec{r_{2\hat{u}}} - \vec{r_{0\hat{u}}} \\
\Delta v_1 &= \vec{r_{1\hat{v}}} - \vec{r_{0\hat{v}}} \\
\Delta v_2 &= \vec{r_{2\hat{v}}} - \vec{r_{0\hat{v}}}
\end{aligned} \tag{4.8}$$

Se calcula además el área del triángulo en las coordenadas (u, v).

$$a = \frac{1}{2}(\vec{r_1} - \vec{r_0}) \times (\vec{r_2} - \vec{r_0}) \tag{4.9}$$

Considerando el mapeo definido en la Ecuación 4.7, las variables definidas sobre el espacio (u, v) se reescriben al espacio (x, y, z).

$$
\begin{aligned}
\Delta u_1 &= \vec{p_1}_{\hat{x}} - \vec{p_0}_{\hat{x}} \\
\Delta u_2 &= \vec{p_2}_{\hat{x}} - \vec{p_0}_{\hat{x}} \\
\Delta v_1 &= \vec{p_1}_{\hat{z}} - \vec{p_0}_{\hat{z}} \\
\Delta v_2 &= \vec{p_2}_{\hat{z}} - \vec{p_0}_{\hat{z}} \\
a &= \frac{1}{2}(\vec{p_1} - \vec{p_0}) \times (\vec{p_2} - \vec{p_0})
\end{aligned}
\tag{4.10}
$$

Es importante entender que el mapeo y cálculo de Δu_1, Δu_2, Δv_1, Δv_2 y a se realiza solo una vez sobre la tela en reposo antes de iniciar la simulación. Las coordenadas (u, v) de un triángulo no cambian a pesar de que este se mueva en las coordenadas (x, y, z). Solo cálculo de Δx_1 y Δx_2 se realiza en cada paso de la simulación.

La relación entre las variables definidas en las Ecuaciones 4.8 se escribe como sigue.

$$
\begin{aligned}
\Delta x_1 &= W_u \Delta u_1 + W_v \Delta v_1 \\
\Delta x_2 &= W_u \Delta u_2 + W_v \Delta v_2
\end{aligned}
\tag{4.11}
$$

A partir de donde se despeja W en función de la posición de las partículas.

$$
\begin{aligned}
W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) &= \frac{(\vec{p_1} - \vec{p_0})\Delta v_2 - (\vec{p_2} - \vec{p_0})\Delta v_1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \\
W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) &= -\frac{(\vec{p_1} - \vec{p_0})\Delta u_2 - (\vec{p_2} - \vec{p_0})\Delta u_1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1}
\end{aligned}
\tag{4.12}
$$

Las derivadas parciales de las Ecuaciones 4.12 necesarias para el cálculo de fuerzas se muestran en las Ecuaciones 4.13. Estas al depender solo de las variables Δu_1, Δu_2, Δv_1 y Δv_2, se calculan junto a ellas antes del inicio de la simulación.

$$
\begin{aligned}
\frac{\partial W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_0}} &= \frac{\Delta v_1 - \Delta v_2}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} I_{3x3} \\
\frac{\partial W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_1}} &= \frac{\Delta v_2}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} I_{3x3} \\
\frac{\partial W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_2}} &= -\frac{\Delta v_1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} I_{3x3} \\
\frac{\partial W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_0}} &= \frac{\Delta u_2 - \Delta u_1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} I_{3x3} \\
\frac{\partial W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_1}} &= -\frac{\Delta u_2}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} I_{3x3} \\
\frac{\partial W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_2}} &= \frac{\Delta u_1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} I_{3x3}
\end{aligned}
\tag{4.13}
$$

Donde I es la matriz identidad.

4.3.2.1. Aplicación de fuerzas

El modelo calcula las fuerzas para contrarrestar las deformaciones en grupos de partículas mediante el uso de una *función condición* $C(x)$ donde x son las posiciones de las partículas

en el grupo. Cada tipo de deformación tiene asociada una función condición diferente.

La fuerza aportada por el modelo, independiente de la deformación objetivo, se calcula como sigue.

$$\Delta f_i = -k_s \frac{\partial C(x)}{\partial x_i} C(x) \tag{4.14}$$

Donde k_s es la constante elástica del material y i el índice de la partícula en el grupo que se está calculando.

El amortiguamiento asociado al sistema se calcula de igual manera utilizando la función condición.

$$\Delta f_i = -k_d \frac{\partial C(x)}{\partial x_i} \sum_{i}^{n-1} \left(\frac{\partial C(x)}{\partial x_i} v_i \right) \tag{4.15}$$

Donde k_d es el coeficiente de amortiguamiento, i el índice de la partícula en el grupo que se está calculando y n el total de partículas en el grupo.

Para las fuerzas de estiramiento se calcula en grupos de tres partículas P_0, P_1 y P_2 que corresponden a cada triángulo en la malla que compone la tela. La función condición de estiramiento es la que sigue.

$$C(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) = \begin{bmatrix} C_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) \\ C_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) \end{bmatrix} = a^{\frac{3}{4}} \begin{bmatrix} \|W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2})\| - b_u \\ \|W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2})\| - b_v \end{bmatrix} \tag{4.16}$$

Donde b_u y b_v son componentes para ajustar la desviación del estiramiento respecto a la posición inicial de la tela.

Las derivadas parciales de la función condición para el estiramiento.

$$\begin{aligned} \frac{\partial C_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_i}} &= a^{\frac{3}{4}} \frac{\partial W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_i}} \frac{W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2}))}{\|W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2}))\|} \\ \frac{\partial C_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_i}} &= a^{\frac{3}{4}} \frac{\partial W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_i}} \frac{W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2}))}{\|W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2}))\|} \end{aligned} \tag{4.17}$$

De igual forma, para las fuerzas de cizallamiento se calcula en grupos de tres partículas P_0, P_1 y P_2 que corresponden a cada triángulo en la malla que compone la tela. La función condición de cizallamiento es la que sigue.

$$C(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) = a^{\frac{3}{4}} W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) \cdot W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) \tag{4.18}$$

La derivada parcial de la función condición para el cizallamiento.

$$\frac{\partial C(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_i}} = a^{\frac{3}{4}} \left(\frac{\partial W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_i}} \cdot W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) + W_u(\vec{p_0},\ \vec{p_1},\ \vec{p_2}) \cdot \frac{\partial W_v(\vec{p_0},\ \vec{p_1},\ \vec{p_2})}{\partial \vec{p_i}} \right) \tag{4.19}$$

La fuerza de plegado se calcula sobre grupos de cuatro partículas P_0, P_1, P_2 y P_3 que corresponden cada par de triángulos adyacentes en la malla de compone la tela, como se

observa en la Figura 4.11.

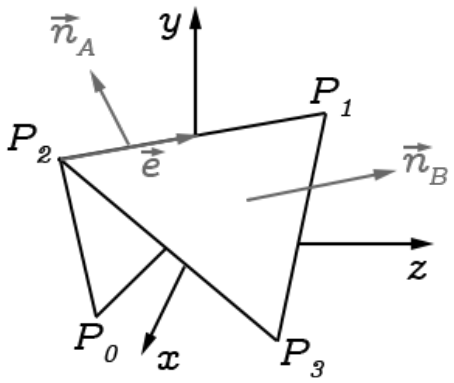


Figura 4.11: Triángulos adyacentes para el cálculo de fuerza de plegado.

Las normales $\vec{n_A}$ y $\vec{n_B}$ de los triángulos y el arco $\vec{e}$ que los une se calculan como muestran las Ecuaciones 4.20.

$$\begin{aligned} \vec{n_A} &= (\vec{p_2} - \vec{p_0}) \times (\vec{p_1} - \vec{p_0}) \\ \vec{n_B} &= (\vec{p_1} - \vec{p_3}) \times (\vec{p_2} - \vec{p_3}) \\ \vec{e} &= \vec{p_1} - \vec{p_2} \end{aligned} \tag{4.20}$$

A partir de las Ecuaciones 4.20 se definen las siguientes funciones trigonométricas.

$$\begin{aligned} cos(\theta) &= \frac{\vec{n_A}}{\|\vec{n_A}\|} \cdot \frac{\vec{n_B}}{\|\vec{n_B}\|} \\ sen(\theta) &= \left(\frac{\vec{n_A}}{\|\vec{n_A}\|} \cdot \frac{\vec{n_B}}{\|\vec{n_B}\|} \right) \cdot \frac{\vec{e}}{\|\vec{e}\|} \end{aligned} \tag{4.21}$$

Las derivadas parciales de las Ecuaciones 4.20.

$$\frac{\partial \vec{n_A}}{\partial \vec{p_0}} = \begin{bmatrix} 0 & -(\vec{p_2} - \vec{p_1})_{\hat{z}} & (\vec{p_2} - \vec{p_1})_{\hat{y}} \\ (\vec{p_2} - \vec{p_1})_{\hat{z}} & 0 & -(\vec{p_2} - \vec{p_1})_{\hat{x}} \\ -(\vec{p_2} - \vec{p_1})_{\hat{y}} & (\vec{p_2} - \vec{p_1})_{\hat{x}} & 0 \end{bmatrix}$$

$$\frac{\partial \vec{n_A}}{\partial \vec{p_1}} = \begin{bmatrix} 0 & -(\vec{p_0} - \vec{p_2})_{\hat{z}} & (\vec{p_0} - \vec{p_2})_{\hat{y}} \\ (\vec{p_0} - \vec{p_2})_{\hat{z}} & 0 & -(\vec{p_0} - \vec{p_2})_{\hat{x}} \\ -(\vec{p_0} - \vec{p_2})_{\hat{y}} & (\vec{p_0} - \vec{p_2})_{\hat{x}} & 0 \end{bmatrix}$$

$$\frac{\partial \vec{n_A}}{\partial \vec{p_2}} = \begin{bmatrix} 0 & -(\vec{p_1} - \vec{p_0})_{\hat{z}} & (\vec{p_1} - \vec{p_0})_{\hat{y}} \\ (\vec{p_1} - \vec{p_0})_{\hat{z}} & 0 & -(\vec{p_1} - \vec{p_0})_{\hat{x}} \\ -(\vec{p_1} - \vec{p_0})_{\hat{y}} & (\vec{p_1} - \vec{p_0})_{\hat{x}} & 0 \end{bmatrix}$$

$$\frac{\partial \vec{n_A}}{\partial \vec{p_3}} = 0_{3x3}$$

$$\frac{\partial \vec{n_B}}{\partial \vec{p_0}} = 0_{3x3}$$

$$\frac{\partial \vec{n_B}}{\partial \vec{p_1}} = \begin{bmatrix} 0 & -(\vec{p_2} - \vec{p_3})_{\hat{z}} & (\vec{p_2} - \vec{p_3})_{\hat{y}} \\ (\vec{p_2} - \vec{p_3})_{\hat{z}} & 0 & -(\vec{p_2} - \vec{p_3})_{\hat{x}} \\ -(\vec{p_2} - \vec{p_3})_{\hat{y}} & (\vec{p_2} - \vec{p_3})_{\hat{x}} & 0 \end{bmatrix} \tag{4.22}$$

$$\frac{\partial \vec{n_B}}{\partial \vec{p_2}} = \begin{bmatrix} 0 & -(\vec{p_3} - \vec{p_1})_{\hat{z}} & (\vec{p_3} - \vec{p_1})_{\hat{y}} \\ (\vec{p_3} - \vec{p_1})_{\hat{z}} & 0 & -(\vec{p_3} - \vec{p_1})_{\hat{x}} \\ -(\vec{p_3} - \vec{p_1})_{\hat{y}} & (\vec{p_3} - \vec{p_1})_{\hat{x}} & 0 \end{bmatrix}$$

$$\frac{\partial \vec{n_B}}{\partial \vec{p_3}} = \begin{bmatrix} 0 & -(\vec{p_1} - \vec{p_2})_{\hat{z}} & (\vec{p_1} - \vec{p_2})_{\hat{y}} \\ (\vec{p_1} - \vec{p_2})_{\hat{z}} & 0 & -(\vec{p_1} - \vec{p_2})_{\hat{x}} \\ -(\vec{p_1} - \vec{p_2})_{\hat{y}} & (\vec{p_1} - \vec{p_2})_{\hat{x}} & 0 \end{bmatrix}$$

$$\frac{\partial \vec{e}}{\partial \vec{p_0}} = 0_{3x3}$$

$$\frac{\partial \vec{e}}{\partial \vec{p_1}} = I_{3x3}$$

$$\frac{\partial \vec{e}}{\partial \vec{p_2}} = -I_{3x3}$$

$$\frac{\partial \vec{e}}{\partial \vec{p_3}} = 0_{3x3}$$

Las derivadas parciales de las Ecuaciones 4.21.

$$\frac{\partial cos(\theta)}{\partial \vec{p_i}} = \frac{\partial \vec{n_A}}{\partial \vec{p_i}} \odot \frac{\vec{n_B}}{\|\vec{n_A}\| \, \|\vec{n_B}\|} + \frac{\partial \vec{n_B}}{\partial \vec{p_i}} \odot \frac{\vec{n_A}}{\|\vec{n_A}\| \, \|\vec{n_B}\|}$$

$$\frac{\partial sen(\theta)}{\partial \vec{p_i}} = \left(\frac{1}{\|\vec{n_A}\|} \frac{\partial \vec{n_A}}{\partial \vec{p_i}} \otimes \frac{\vec{n_B}}{\|\vec{n_B}\|} + \frac{1}{\|\vec{n_B}\|} \frac{\partial \vec{n_B}}{\partial \vec{p_i}} \otimes \frac{\vec{n_A}}{\|\vec{n_A}\|} \right) \odot \frac{\vec{e}}{\|\vec{e}\|} + (\vec{n_A} \times \vec{n_B}) \odot \frac{1}{\|\vec{e}\|} \frac{\partial \vec{e}}{\partial \vec{p_i}} \tag{4.23}$$

Donde los operadores $\odot\otimes$ son producto punto y producto cruz respectivamente, pero aplicados a cada fila de la matriz que se encuentra a la izquierda del operador y guardando el resultado en forma de vector.

Finalmente la función condición de plegado es la que sigue.

$$C(\theta) = arctan\left(\frac{sen(\theta)}{cos(\theta)}\right) \tag{4.24}$$

La derivada parcial de la función condición para el plegado.

$$\frac{\partial C(\theta)}{\partial \vec{p_i}} = cos(\theta)\frac{\partial sen(\theta)}{\partial \vec{p_i}} - sen(\theta)\frac{\partial cos(\theta)}{\partial \vec{p_i}} \tag{4.25}$$

Se debe notar que aunque las ecuaciones tomen como argumento a θ, este está en función de la posición de las partículas como se muestra en las Ecuaciones 4.20 y 4.21. Además, algunas ecuaciones se han modificado del trabajo original en base a las correcciones relativas al área de los triángulos y la fuerza de amortiguamiento encontradas en [16].

El modelo consigue resultados realistas, pero es fácil ver que el costo computacional es considerablemente más alto que el modelo de masa - resorte debido principalmente al cálculo de las fuerzas de plegado. Sin embargo, estas no tienen un impacto visual significativo, como se observa en la Figura 4.12, por lo que puede ser conveniente no considerarlas en la simulación.

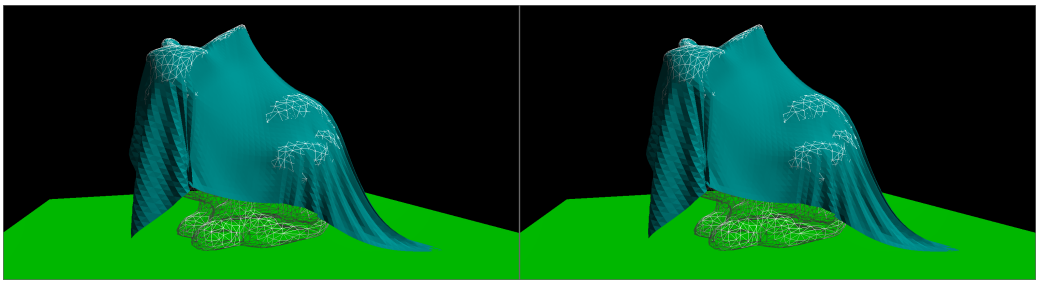


Figura 4.12: Modelo de mecánicas continuas con y sin cálculo de fuerzas de plegado respectivamente.

Al igual que en la aplicación de fuerzas del modelo masa - resorte, no se realiza una modificación a las partículas de forma inmediata, por lo que el cálculo de las fuerzas es sumamente paralelizable.

4.4. Evolución en el tiempo

El objetivo principal de la simulación es poder responder la pregunta ¿dónde se encontrarán las partículas que componen la tela dentro de un cierto paso de tiempo? El métodos para dar respuesta a esto es a través de integraciones numéricas que hacen uso de las posiciones y fuerzas aplicadas que han tenido las partículas en el pasado para calcular sus posiciones futuras. Estos cálculos se realizan durante la llamada etapa de integración de la simulación.

Existe múltiples métodos de integración numérica para resolver este problema: *Euler hacia adelante*, *Euler hacia atrás*, *Runge–Kutta*, *Verlet*, etc. A continuación se entrará en detalles

sobre la integración de Verlet por su simpleza y bajo coste computacional.

4.4.1. Integración de Verlet

Definiendo $\vec{p_i}(t)$ como la posición de una partícula y $f_i\vec{(t)}$ sus fuerzas en función de un tiempo t. La integración de Verlet se define como sigue.

$$\vec{p_i}(t + \Delta t) = 2\vec{p_i}(t) - \vec{p_i}(t - \Delta t) + \frac{\vec{f_i}(t)}{m_i}\Delta t^2 \tag{4.26}$$

Reescribiendo la Ecuación 4.26 se añade una variable para simular el amortiguamiento.

$$\vec{p_i}(t + \Delta t) = \vec{p_i}(t) + (1 - k_d)\,(\vec{p_i}(t) - \vec{p_i}(t - \Delta t)) + \frac{\vec{f_i}(t)}{m_i}\Delta t^2 \tag{4.27}$$

Donde k_d es el coeficiente de amortiguamiento, un valor entre 0 y 1.

Se puede entender la integración tal que, para obtener la siguiente posición de una partícula, se toma su posición actual, se desplaza primero según su *inercia* expresada como $(1 - k_d)(\vec{p_i}(t) - \vec{p_i}(t - \Delta t))$ y luego según las fuerzas ejercidas sobre ella. Esto se puede observar en el ejemplo de la Figura 4.13.

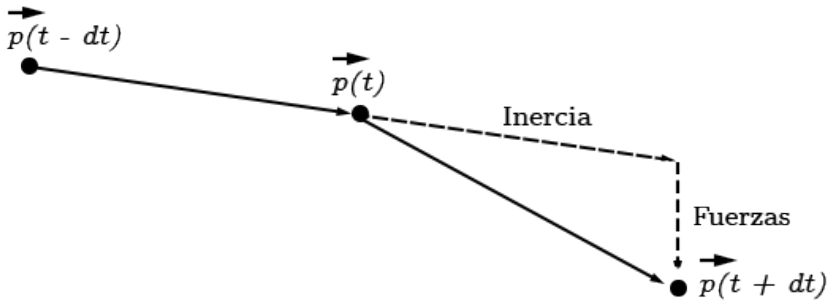


Figura 4.13: Ejemplo de integración de Verlet.

4.5. Manejo de colisiones

La etapa de colisión es en general la más costosa de calcular. Consta de dos fases, una primera fase de *detección* de colisiones donde se buscan todas las partículas tal que su trayectoria haya intersectado con algún triángulo en la malla del objeto a colisionar y una segunda fase de *respuesta* donde se corrige la posición de todas las partículas que se hayan encontrado en la primera.

Se divide la detección de colisiones en dos tipos: *tela - objeto* y *tela - tela*. A continuación se entrará en detalles solo acerca de la colisión tela - objeto haciendo uso de *Octrees*, dado que es la estrategia implementada en el programa desarrollado en este trabajo.

4.5.1. Detección

La fase de detección debe comprobar el trayecto de cada partícula que compone la tela contra todos los triángulos que componen la malla del objeto a colisionar. Por esta razón, es

importante el uso de algoritmos y estructuras de datos que nos permitan filtrar los pares de partículas con triángulos que con toda seguridad no colisionarán.

Independiente de la estructura de datos que se utilice para facilitar la detección de colisiones, es común que estas hagan uso de algoritmos para la verificación de intersección entre primitivas. Aquellos utilizados en este informe se pueden encontrar en el Anexo B.

4.5.1.1. Tela - Objeto

En este trabajo se asume que los objetos contra los que colisiona la tela son sólidos y se mantienen estáticos durante la simulación.

4.5.1.1.1. Octree

Un *octree* es una estructura de datos que divide el espacio tridimensional recursivamente en ocho octantes. Se guarda en forma de árbol, donde cada nodo representa una zona del espacio y cada uno de sus ocho hijos uno de los octantes que la componen.

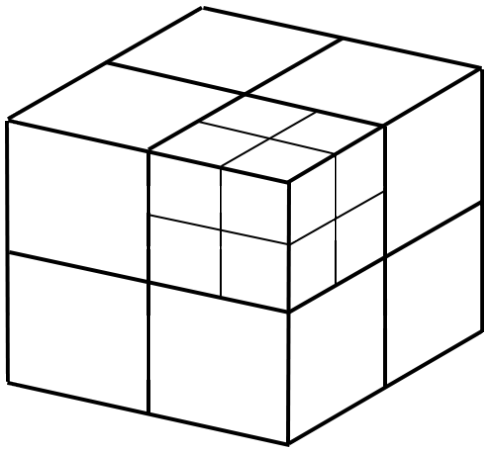


Figura 4.14: División del espacio por un *Octree*.

Utilizando un *octree* se divide el espacio de los objetos a colisionar guardando los triángulos de su malla en los nodos que los intersectan. Se presentan dos posibilidades, guardar los triángulos en los nodos que los contienen completamente, o bien, guardar los triángulos en cada nodo que los intersecte. Una representación de esto se observa en la Figura 4.15.

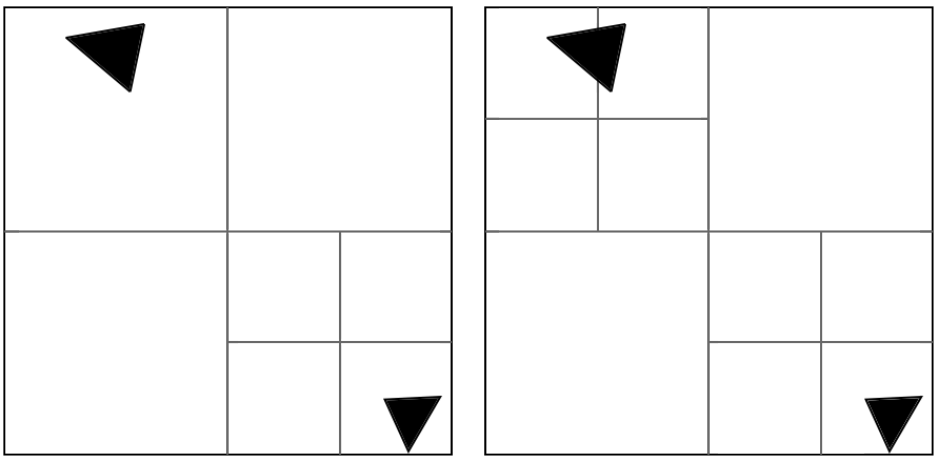


Figura 4.15: Triángulo guardado en nodo que lo contiene y en cada nodo que lo intersecta respectivamente.

Guardar triángulos según la primera opción provoca que el espacio potencial en el que se realice el cálculo de verificación de la colisión sea mayor y por ende, la trayectoria de más partículas caigan en él. Esto produce una cantidad importante de verificaciones innecesarias que aumenta con el tamaño del espacio que representa el *octree*. Este trabajo fija la altura del árbol en ocho y guarda los triángulos en todas las hojas que los intersectan, aumentando el costo en memoria a cambio de disminuir el costo total de las verificaciones de colisión.

Realizar el filtrado de posibles triángulos con los que colisiona una partícula consiste en bajar por el árbol hasta las hojas de forma recursiva, siguiendo el camino tal que el segmento generado por la trayectoria de la partícula intersecte con sus nodos. Una vez se ha llegado a las hojas, el conjunto de todos los triángulos que contienen son los candidatos a verificación de colisiones.

Es interesante notar que si se tiene más de un objeto con el que se desea colisionar, es conveniente generar un *octree* por cada uno de ellos en lugar de uno solo que contenga a todos. Debido a las limitaciones de memoria un *octree* no puede tener una altura demasiado elevada lo que se refleja en el tamaño del espacio que representa cada hoja. Mientras menos espacio abarque la estructura, el tamaño por hoja será menor y a su vez la cantidad de potenciales triángulos con los que intersecta y debe guardar también, reduciendo la cantidad de verificaciones a realizar. Además de esto, reducir el tamaño de la estructura para que contenga lo más ajustado posible al objeto, permite que las partículas que no se encuentren dentro de su zona sean rápidamente desechadas en la primera verificación realizada en la raíz del árbol. Estos motivos justifican, si se está dispuesto a pagar más en memoria, el costo extra de tener más de una estructura para manejar las colisiones de distintos objetos, puesto que el costo de obtener las potenciales colisiones es mínimo en comparación a la verificación de la colisión como tal.

4.5.2. Respuesta

La respuesta a una colisión tiene un efecto directo en la visualización de la tela y su interacción con el entorno. El método común de manejar una partícula que atraviesa la cara de un objeto, es empujarla para que vuelva a quedar del mismo lado de la cara donde estaba

antes de ocurrir la colisión. La posición exacta en que se deja la partícula luego de empujarla será la principal responsable de obtener una interacción de objetos realista y la presencia o ausencia de artefactos en la visualización.

Sea $\vec{p_i}(t)$ la posición de una partícula en función de un tiempo t, $\vec{n}$ la normal de un triángulo intersectado por la trayectoria de la partícula y $\vec{I}$ el punto de intersección de esta. Se definen dos posibles respuestas a la colisión, devolver la partícula a partir del punto I una distancia ϵ en dirección a $\vec{p_i}(t - \Delta t)$, o bien, devolver la partícula a partir del punto obtenido al proyectar $\vec{p_i}(t)$ sobre el triángulo intersectado una distancia ϵ con dirección $\vec{n}$. Una representación de esto se observa en la Figura 4.16.

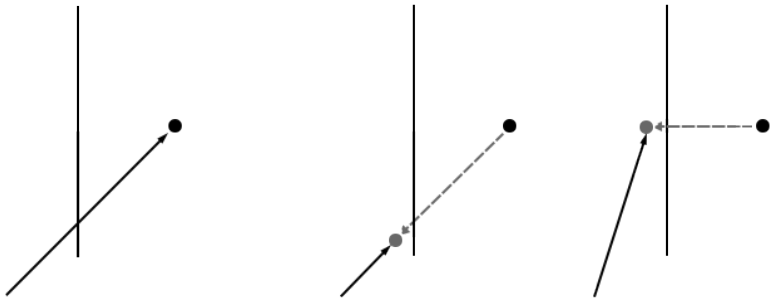


Figura 4.16: Colisión y dos posibles respuestas.

El primer tipo de respuesta se escribe como sigue.

$$\vec{p_i}(t) = \vec{I} - \epsilon \frac{\vec{I} - \vec{p_i}(t - \Delta t)}{\left\| \vec{I} - \vec{p_i}(t - \Delta t) \right\|} \tag{4.28}$$

Al devolver cada vez la partícula a la posición de intersección y moverla una pequeña distancia en dirección a su posición anterior se produce un efecto equivalente a fricción infinita, como se observa en la Figura 4.17 donde la tela cuelga de una caja sin caer a pesar de que su zona de contacto es mínima.

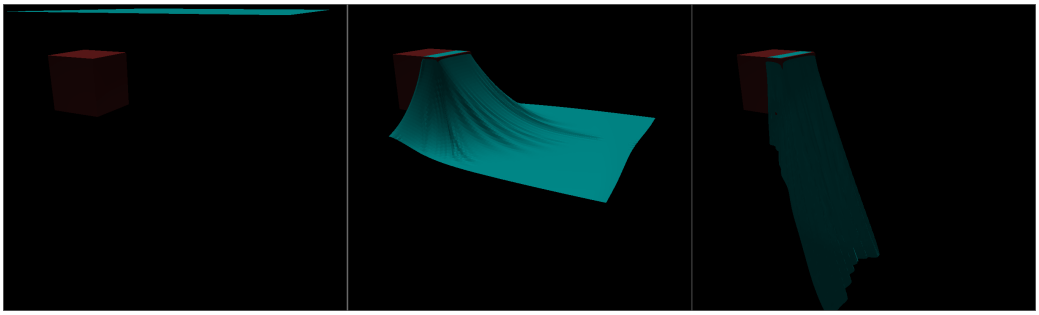


Figura 4.17: Ejemplo de fricción infinita.

El segundo tipo de respuesta se escribe como sigue.

$$\vec{p_i}(t) = \vec{p_i}(t) + \vec{n}(\epsilon - \vec{n} \cdot (\vec{p_i}(t) - \vec{I})) \tag{4.29}$$

Al contrario que antes, se genera un efecto de fricción nula, esto resulta en que las partículas

“resbalan” sobre la superficie. En la práctica la fuerza entre partículas no logra contrarrestar este efecto y caen por los contornos de los objetos de manera anormal. Esto se puede observar en la Figura 4.18.

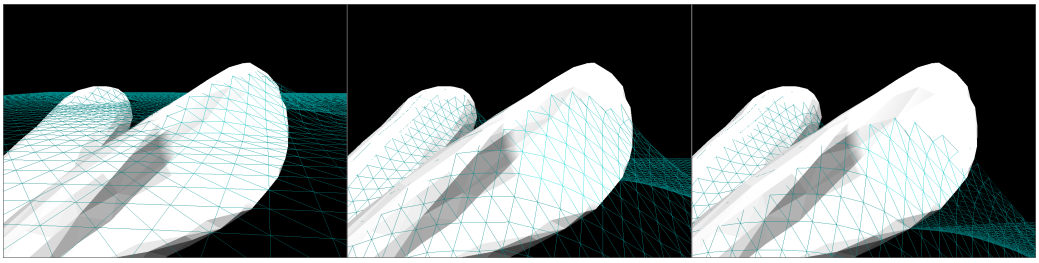


Figura 4.18: Ejemplo de fricción nula.

Ninguno de los tipos de respuesta vistos es ideal, hace falta considerar la fricción generada por el material de ambos objetos, sin embargo, no se ahondará en ello en este trabajo. Se decide utilizar el primer método para la simulación.

Capítulo 5

Evaluación

Los resultados mostrados en este capítulo fueron obtenidos utilizando:

- Sistema: Windows 11 Home x64
- Procesador: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz
- Memoria SSD: SK hynix BC501 HFM256GDJTNG-8310A
- Memoria RAM: 16 GB
- Tarjeta gráfica: NVIDIA GeForce GTX 1050 (notebook) 3 GB

5.1. Modelos

Al generar una simulación con los modelos de masa - resorte (restricciones y fuerzas) y mecánicas continuas (con y sin aplicación de plegado) se obtienen los tiempos de ejecución para 600 frames de la Figura 5.1.

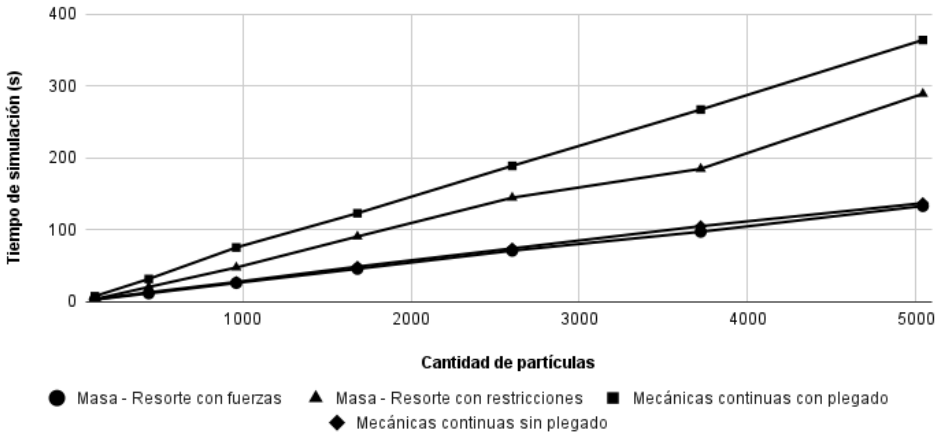


Figura 5.1: Tiempo de ejecución para generar 600 frames con modelo de masa - resorte y mecánicas continuas.

Se observa que el modelo de masa - resorte con aplicación de restricciones tarda casi dos veces el tiempo del mismo modelo con aplicación de fuerzas. A su vez, el modelo de mecánicas continuas con aplicación de plegado es considerablemente más lento, sin embargo, al no

calcular esta fuerza consigue el mismo e incluso mejor rendimiento que el modelo de masa - resorte con aplicación de fuerzas.

Una vista más detallada del tiempo de ejecución por cada etapa se muestra a continuación.

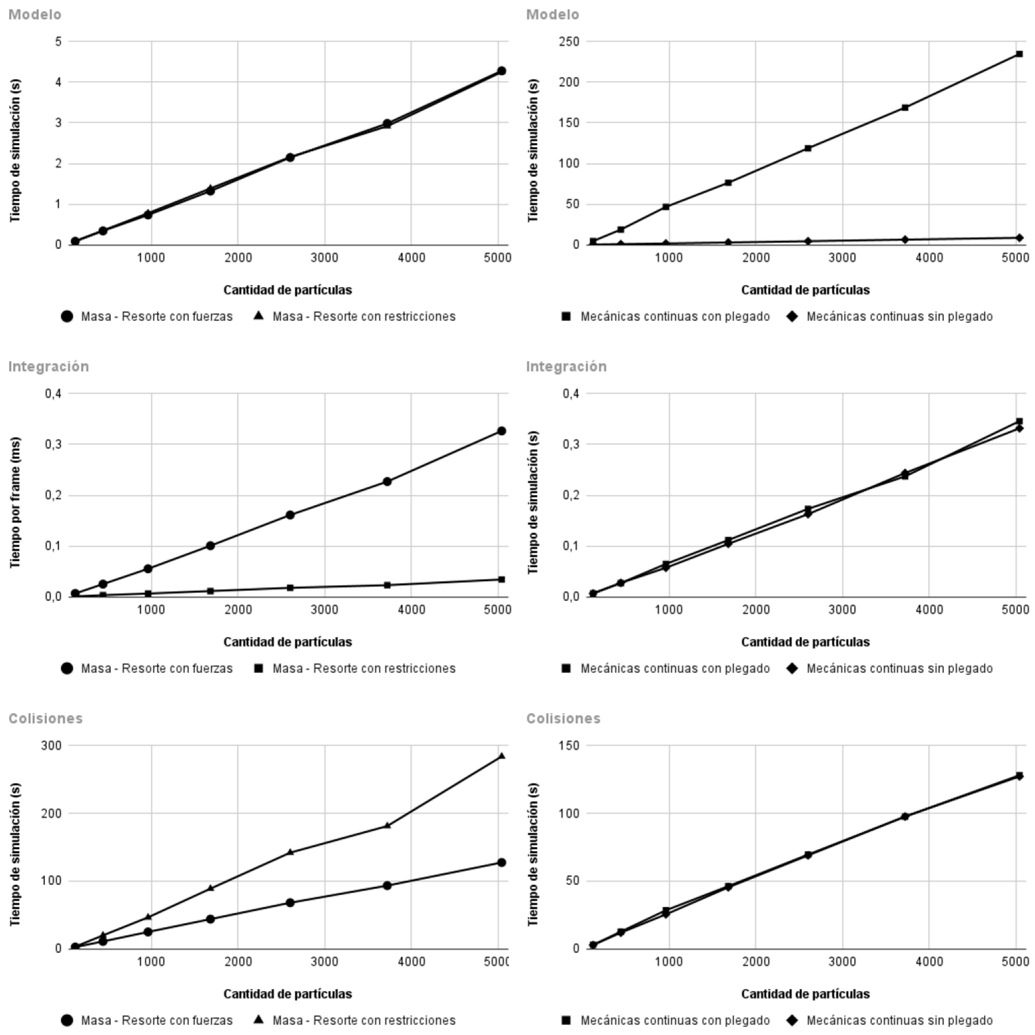


Figura 5.2: Tiempo de ejecución por etapa de simulación.

El hecho que el modelo de masa - resorte con restricciones tenga una cantidad de tiempo menor en la etapa de integración concuerda con su uso de iteraciones donde solo se ejecuta una vez por frame, mientras que su mayor tiempo en la etapa de colisión puede deberse a la constante violación de restricciones y consecuente corrección presente durante las iteraciones que caracterizan el modelo. Respecto al modelo de mecánicas continuas se observa el gran impacto que tiene el cálculo de plegado durante la etapa de modelo, sin afectar las etapas de integración y colisión.

Los renderizados del frame 600 de las simulaciones utilizando los distintos modelos se muestra en la Figura 5.3.

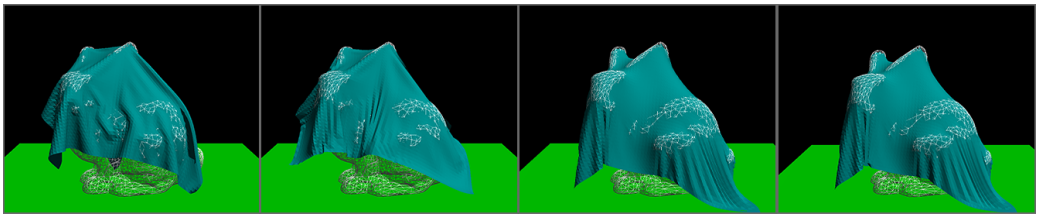


Figura 5.3: Renderizado de resultado de los modelos. Modelo masa - resorte con aplicación de fuerzas, masa - resorte con aplicación de restricciones, mecánicas continuas con plegado y mecánicas continuas sin plegado respectivamente.

El resultado de las simulaciones realizadas con mecánicas continuas son casi indistinguibles entre sí, lo que presenta la posibilidad de descartar el cálculo de fuerzas de plegado dado el poco impacto visual que genera y su alto coste computacional. En cuanto al realismo de la imagen, el modelo de masa - resorte no consigue su misma calidad, presenta arrugas demasiado gruesas a la vez que una mayor tensión. Si bien esto último puede ser provocado por la configuración de coeficientes utilizados, la dificultad para definirlos y la mínima diferencia de rendimiento en comparación al modelo de mecánicas continuas sin plegado, lo pone en una desventaja ya no solo visual, si no también práctica.

5.2. Detección de colisiones

La detección de colisiones es una tarea computacionalmente compleja, pudiendo ser más costosa que el manejo mismo de la colisión. Por esta razón es importante poder disminuir la cantidad de comparaciones totales requeridas para confirmar o desechar la colisión de una partícula.

En la Figura 5.4 se muestra el total de comparaciones realizadas con el método de fuerza bruta para verificar colisiones contra el modelo Stanford bunny (9468 caras) y un prisma rectángulo (12 caras) para la representación del piso después de 600 frames de simulación (mecánicas continuas sin plegado e integración de Verlet).

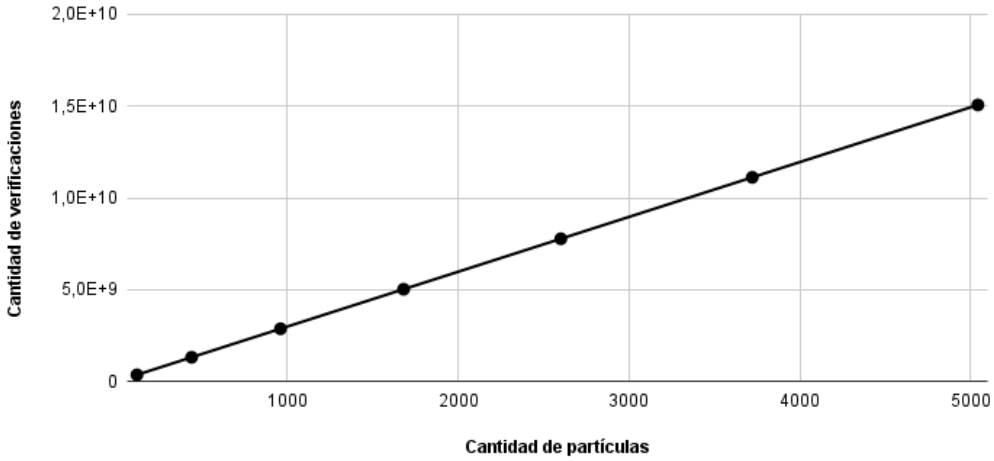


Figura 5.4: Cantidad de verificaciones realizadas con fuerza bruta.

Esto es orden de $\mathcal{O}(N(S+F))$ donde N es la cantidad de partículas que componen la tela, S la cantidad de caras que componen el modelo Standford bunny y F la cantidad de caras que componen el prisma rectangular del piso.

Bajo el mismo contexto, haciendo uso de la estructura de datos octree de forma tal que se tiene un octree por cada objeto a colisionar y un único octree para todos los objetos a colisionar, se obtienen los resultados de la Figura 5.5.

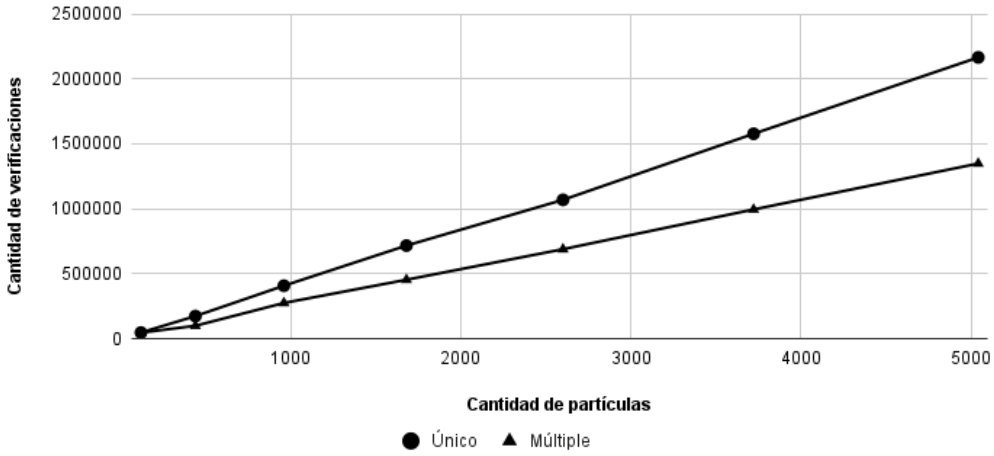


Figura 5.5: Cantidad de verificaciones realizadas con uso de octree.

La cantidad de verificaciones necesarias resultan entre cuatro y cinco órdenes de magnitud menor que con el uso de fuerza bruta, lo que justifica ampliamente el tiempo necesario para navegar el árbol. Además se observa que tener múltiples octree (uno por cada objeto a colisionar) permite reducir la cantidad de verificaciones al existir un área mayor fuera de este donde las partículas son desechadas directamente en el nivel raíz.

Capítulo 6

Programa de simulación

Con el objetivo implementar y probar las distintas estrategias estudiadas en la sección anterior, se desarrolla un programa capaz de mostrar en tiempo real la simulación de una tela cayendo sobre un objeto.

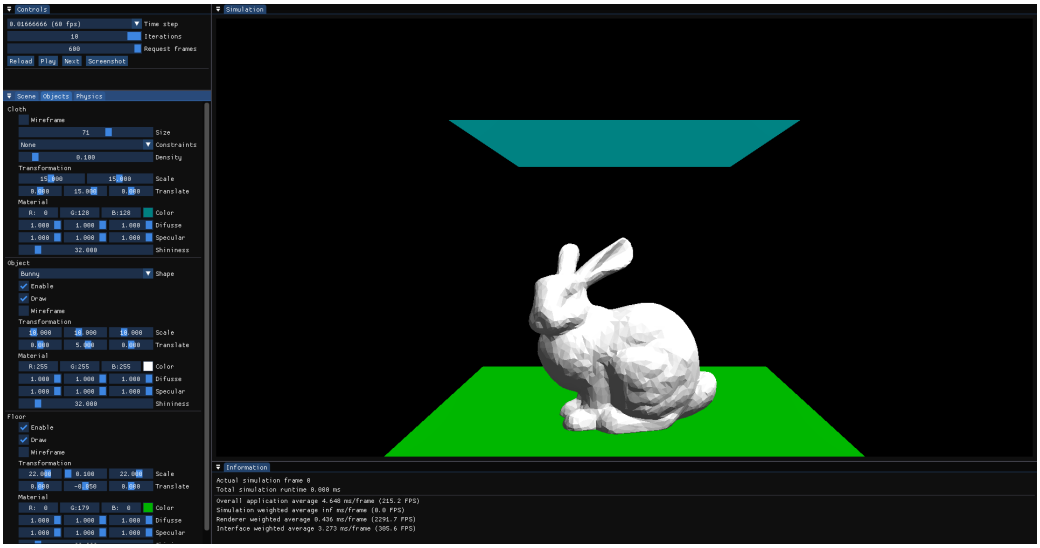


Figura 6.1: Programa de simulación.

Dado que el rendimiento es una prioridad en las simulaciones, el desarrollo del programa se lleva a cabo en C++ 20.

6.1. Requerimientos

Los requerimientos referidos a lo que debe ser capaz de cumplir y realizar el programa son los siguientes:

- Implementar un sistema de simulación con las siguientes características:
  - Ser capaz de representar una tela de tamaño y cantidad de partículas específica.
  - Ser capaz de representar objetos a colisionar a partir de modelos en formato OBJ.
  - Implementar fases de modelo, integración y colisión compatibles con cualquier algoritmo.

- Ser independiente de cualquier otro módulo del programa, debe ser funcional de manera individual.

- Implementar un sistema de renderizado con las siguientes características:
  - Ser capaz de dibujar el estado de la simulación en tiempo real.
  - Ser compatible cualquier API de gráficos 3D como OpenGL, Metal, Vulkan, etc.

- Implementar un sistema de interfaz con las siguientes características:
  - Permitir al usuario modificar las variables de las distintas etapas de la simulación.
  - Permitir controlar el flujo de la simulación como iniciar, detener, avanzar un estado, etc.
  - Permitir al usuario modificar los controles del renderizado como cámara, luces, shaders, etc.
  - Incrustar el sistema de renderizado.

6.2. Dependencias

El programa desarrollado hace uso de las siguientes librerías de terceros:

- Glad: Inicialización de OpenGL.
- GLFW: Manejo de ventanas.
- GLM: Manejo de matemáticas compatibles con OpenGL y GLSL.
- ImGui: Interfaz gráfica.
- stb: Guardado de imágenes.

6.3. Arquitectura

La estructura del programa se divide en tres capas: simulación, renderizado e interfaz. La capa de simulación es independiente de las demás y se encarga de todo lo referente al manejo de tela - objeto y cálculo de estados de simulación. La capa de renderizado tiene acceso a la información de la capa de simulación y se encarga de dibujar el estado de esta. La capa de interfaz tiene acceso a las otras capas y se encarga de dibujar y manejar las interacciones de la interfaz de usuario. La relación entre capas se observa en la Figura 6.2. Un diagrama de clases más detallado se encuentra en el Anexo A.

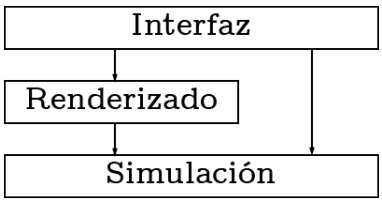


Figura 6.2: Arquitectura principal del programa.

Mientras el programa esté en uso, se ejecutan indefinidamente y en orden las capas de este, primero la simulación realiza el cálculo de el siguiente estado, luego el renderizado genera su imagen y finalmente la interfaz se dibuja. Este ciclo se puede ver a grandes rasgos en el Código 6.1.

Código 6.1: Flujo principal de la aplicación. Source.cpp.

```
while (renderer->is_running())
{
    //...

    simulation->step(delta_time);

    renderer->pre_render(delta_time);
    renderer->render(simulation);
    renderer->post_render();

    interface->pre_render();
    interface->render(simulation, renderer);
    interface->post_render();
}
```

6.3.1. Simulación

La capa de simulación está encargada de la creación de la tela y los objetos a colisionar, la creación y aplicación de los modelos, las integraciones y las colisiones. Su arquitectura es tal que la capa tiene acceso a un objeto tela y un objeto estático. El objeto tela tiene acceso a un modelo y un método de integración, los cuales a su vez tienen acceso al objeto tela. Del mismo modo el objeto estático tiene acceso a un método de colisión, que a su vez tiene acceso al objeto estático. Esta relación se puede observar en la Figura 6.3.

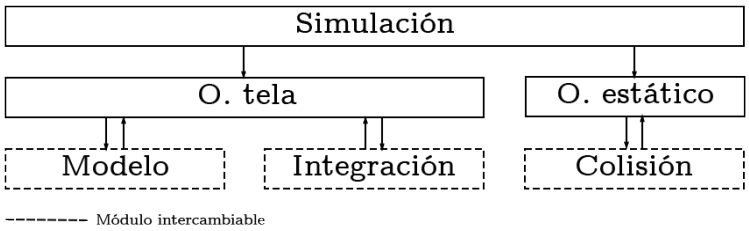


Figura 6.3: Arquitectura de la capa de simulación.

La idea detrás de que los modelos, las integraciones y las colisiones tengan acceso a los objetos es que apliquen sus efectos directamente sobre estos y no sea necesario utilizar memoria intermedia. Esto además facilita la separación de las etapas de la simulación (modelo, integración y colisión), permitiendo implementar distintas versiones de cada uno al extender sus clases abstractas correspondientes y cambiar el orden en que se ejecutan teniendo seguridad de que la información manejada se encuentra actualizada.

La inicialización de la capa se realiza como sigue:

Código 6.2: Inicialización de la capa de simulación. Simulation.cpp.

```
// Creación de objeto tela, su modelo y método de integración
cloth = new ClothObject(...);
cloth->model = new ExampleModel(*cloth);
cloth->integration = new ExampleIntegration(*cloth);

// Creación de objeto de colisión y su método de colisión
object = new StaticObject(...);
object->collision = new ExampleCollision(*object);
```

Para realizar un paso y obtener el siguiente estado de la simulación se puede realizar de la siguiente forma:

Código 6.3: Cálculo del siguiente estado de la simulación. Simulation.cpp.

```
for (int i = 0; i < iterations; ++i)
{
    // Se ejecuta la etapa de modelo
    cloth->model_stage(delta_time / iterations);
    // Se ejecuta la etapa de integración
    cloth->integration_stage(delta_time / iterations);
    // Se ejecuta la etapa de colisión
    object->collision_stage(*cloth);
    // Se actualizan las normales de la tela
    cloth->update_normals();
}
```

Se debe tener presente que tanto el orden de las etapas como el uso de iteraciones en lugar de subciclos es una decisión tomada por el usuario y responde a las necesidades de su simulación, las que en su mayoría, son definidas por la etapa de modelo.

6.3.1.1. Objeto

Se tienen dos tipos de objetos, objeto tela y objeto estático. En su implementación se hace uso de un objeto básico como padre del cual heredar la estructura de datos Half-Edge utilizada en ambos. Con el objetivo de a futuro extender el programa, la estructura se guarda completamente en arreglos que son fáciles de mover a GPU y en consecuencia todos los módulos trabajan accediendo a la información mediante índices en ellos. La implementación del objeto básico se puede observar en el Código 6.4.

Código 6.4: Objeto básico. Object.h.

```
class BasicObject
{
public:
    std::vector<glm::dvec3> vertices; // Vértices
    std::vector<glm::uvec2> neighbors; // Vecinos
    std::vector<unsigned int> twins; // Gemelos
    std::vector<unsigned int> origins; // Vértices de origen
    std::vector<unsigned int> incident_faces; // Caras incidentes
    std::vector<unsigned int> faces; // Caras
    std::vector<glm::dvec3> normals; // Normales

```

```
    BasicObject();

    virtual ~BasicObject() = 0;
};
```

El objeto tela hereda la estructura de datos del objeto básico agregándole listas de arcos, posiciones anteriores, fuerzas, masas, estados (movibles o no) y módulos de modelo e integración. Estas nuevas listas no han sido añadidas directamente al objeto base debido a que el objeto estático no hace uso de ellas y ocuparían memoria innecesaria. La estructura del objeto tela se puede observar en el Código 6.5 y sus funciones son las siguientes:

- `ClothObject(unsigned int num_particles, glm::vec3 center, glm::vec2 size, std::string, float density)`: Recibe la cantidad de partículas a lo largo y ancho de la tela, su centro, su tamaño, la configuración de estados de las partículas y la densidad de la tela. Inicializa un objeto tela con los parámetros recibidos.
- `update_normals()`: Actualiza las normales de las caras guardadas en el objeto.
- `update_masses()`: Actualiza las masas de las partículas según el área de las caras adyacentes.
- `model_stage(double dt)`: Recibe un paso de tiempo. Ejecuta la etapa de modelo utilizando el módulo de modelo del objeto.
- `integration_stage(double dt)`: Recibe un paso de tiempo. Ejecuta la etapa de integración utilizando el módulo de integración del objeto.

Código 6.5: Objeto tela. Object.h.

```
class ClothObject : public BasicObject
{
public:
    Model* model; // Módulo de modelo
    Integration* integration; // Módulo de integración

    std::vector<unsigned int> edges; // Arcos
    std::vector<glm::dvec3> previous_vertices; // Vértices anteriores
    std::vector<glm::dvec3> forces; // Fuerzas sobre las partículas
    std::vector<double> masses; // Masas de las partículas
    std::vector<bool> fixed; // Estado de las partículas

    float density; // Densidad de la tela

    ClothObject(unsigned int num_particles, glm::vec3 center, glm::vec2 size, std::string
      ↪ fixed_mode, float density);

    ~ClothObject();

    void update_normals();
    void update_masses();
    void model_stage(double dt);
```

```
22    void integration_stage(double dt);
23 };
```

El objeto estático hereda la estructura de datos del objeto básico agregándole un módulo de colisión. La estructura del objeto estático se puede observar en el Código 6.6 y sus funciones son las siguientes:

- `StaticObject(std::string filepath, glm::vec3 center, glm::vec3 size)`: Recibe la dirección de un archivo OBJ con el modelo 3D, su centro y su tamaño. Inicializa un objeto estático con los parametros recibidos.
- `collision_stage(ClothObject& cloth)`: Recibe un objeto tela. Ejecuta la etapa de colisión sobre el objeto tela recibido utilizando el módulo de colisión del objeto.

Código 6.6: Objeto estático. Object.h.

```
1  class StaticObject : public BasicObject
2  {
3  public:
4     Collision* collision; // Módulo de colisión
5
6     StaticObject(std::string filepath, glm::vec3 center, glm::vec3 size);
7
8     ~StaticObject();
9
10    void collision_stage(ClothObject& cloth);
11 };
```

6.3.1.2. Modelo

La estructura del modelo se puede observar en el Código 6.7 y sus funciones son las siguientes:

- `Model(ClothObject& src)`: Recibe un objeto tela. Inicializa el modelo y guarda una referencia al objeto tela.
- `apply(double dt)`: Recibe un paso de tiempo. Aplica las restricciones y/o fuerzas del modelo en el objeto tela.

Código 6.7: Plantilla para crear modelos. Model.h.

```
1  class Model
2  {
3  public:
4     Model(ClothObject& src);
5
6     virtual ~Model();
7
8     virtual void apply(double dt) = 0;
9
```

```
protected:
    ClothObject& src; // Objeto tela al que está asociado el módulo
};

// Implementación de un módulo de modelo
class ExampleModel : public Model { ... };
```

6.3.1.3. Integración

La estructura de la integración se puede observar en el Código 6.8 y sus funciones son las siguientes:

- `Integration(ClothObject& src)`: Recibe un objeto tela. Inicializa la integración y guarda una referencia al objeto tela.
- `integrate(double dt)`: Recibe un paso de tiempo. Modifica los vértices en el objeto tela según el algoritmo de integración.

Código 6.8: Plantilla para crear integraciones. Integration.h.

```
class Integration
{
public:
    Integration(ClothObject& src);

    virtual ~Integration() = 0;

    virtual void integrate(double dt) = 0;

protected:
    ClothObject& src; // Objeto tela al que está asociado el módulo
};

// Implementación de un módulo de integración
class ExampleIntegration : public Integration { ... };
```

6.3.1.4. Colisión

La estructura de la colisión se puede observar en el Código 6.9 y sus funciones son las siguientes:

- `Collision(StaticObject& src)`: Recibe un objeto estático. Inicializa la colisión y guarda una referencia al objeto estático.
- `collide(ClothObject& cloth)`: Recibe un objeto tela. Modifica los vértices en el objeto tela según el algoritmo de colisión.

Código 6.9: Plantilla para crear colisiones. Collision.h.

```
class Collision
```

```
{
public:
    Collision(StaticObject& src);

    virtual ~Collision();

    virtual void collide(ClothObject& cloth) = 0;

protected:
    StaticObject& src; // Objeto estático al que está asociado el módulo
};

// Implementación de un módulo de colisión
class ExampleCollision : public Collision { ... };
```

6.3.2. Renderizado

La capa de renderizado está encargada de dibujar los estados de la simulación en cada instante de tiempo haciendo uso de APIs gráficas. En concreto este trabajo hace uso de *OpenGL*, sin embargo, al igual que la capa de simulación, entrega clases abstractas para ciertos módulos que pueden ser reimplementados utilizando cualquier otra API. La capa tiene acceso a clases para el manejo de cámara y luces independiente de la API y clases abstractas para el manejo de objetos, *framebuffers* y *shaders* que deben ser implementados según la elección del usuario. La relación entre los módulos de la capa se puede observar en la Figura 6.4.

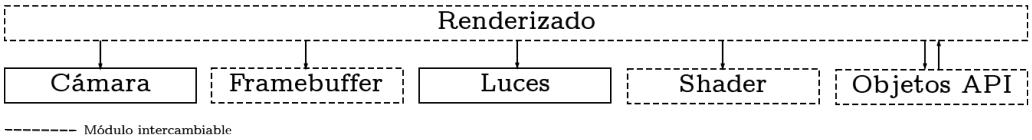


Figura 6.4: Arquitectura de la capa de renderizado.

La capa de renderizado en el momento que se escribe este trabajo, maneja el pre y post renderizado, además de la entrada del teclado y otras características que hacen uso directo de la API de OpenGL. Por esta razón debe implementarse a partir de una clase abstracta, sin embargo, en futuras versiones de este programa dichas funciones deberían ser delegadas a otro módulo dejando el flujo principal del renderizado independiente de la API elegida por el usuario. La estructura de la capa se puede observar en el Código 6.10 y sus funciones son las siguientes:

- `Renderer(const char* name, Simulation* simulation)`: Recibe un nombre y la capa de simulación. Inicializa la capa de renderizado con sus módulos y crea una ventana con el nombre recibido.
- `pre_render(double delta_time)`: Recibe un paso de tiempo. Actualiza las luces, *shaders*, activa el *framebuffer* y limpia la imagen.
- `render(Simulation* simulation)`: Recibe la capa de simulación. Dibuja el estado actual de la capa de simulación.

- `post_render()`: Recibe la entrada teclado y desactiva el *framebuffer*.
- `terminate()`: Limpia los *buffer* y llama a terminar a la API.
- `is_running()`: Retorna verdadero si la ventana de renderizado se está ejecutando.

Código 6.10: Plantilla para crear la capa de renderizado. Renderer.h.

```
class Renderer
{
public:
    Camera* camera; // Cámara
    Framebuffer* framebuffer; // Framebuffer
    Light* light; // Luces
    Shader* shader;   // Shaders
    Display* cloth_display; // Objeto tela según la API
    Display* object_display; // Objeto estático según la API

    Renderer(const char* name, Simulation* simulation);

    ~Renderer();

    void pre_render(double delta_time) = 0;
    void render(Simulation* simulation) = 0;
    void post_render() = 0;
    void terminate() = 0;
    bool is_running() = 0;
};

// Implementación de la capa de renderizado
class ExampleRenderer : public Renderer { ... };
```

6.3.2.1. Objeto según API

Los objetos, tanto el de tipo tela como estático guardan su información en listas organizadas como un Half-Edge. Las APIs gráficas no pueden utilizar directamente la información en este formato para dibujarlas, por lo que es necesario generar una representación intermedia que sea compatible con la API en cuestión y actualizarla antes de cada renderizado. La estructura de esta representación se observa en el Código 6.11 y sus funciones son las siguientes:

- `Display(BasicObject* src)`: Recibe un objeto básico. Guarda una referencia al objeto básico e inicializa una representación del objeto básico compatible la API utilizada.
- `transform()`: Transforma los datos de la estructura Half-Edge del objeto básico a un formato compatible con la API utilizada.
- `draw(Shader shader)`: Recibe el módulo de manejo de *shaders*. Dibuja el objeto haciendo uso de la representación propia de la API.

Código 6.11: Plantilla para crear objetos según API. Display.h.

```
class Display
{
public:
    BasicObject* src; // Objeto al que está asociada la representación

    Display(BasicObject* src);

    ~Display();

    void transform() = 0;
    void draw(Shader shader) = 0;
};

// Implementación del objeto según API
class ExampleDisplay : public Display { ... };
```

6.3.2.2. Framebuffer

La imagen de los objetos se guarda en un *framebuffer* con el objetivo de ser utilizado por la capa de interfaz. Su estructura se observa en el Código 6.12 y sus funciones son las siguientes:

- `Framebuffer(unsigned int width, unsigned int height)`: Recibe un ancho y una altura. Inicializa un *framebuffer* con las dimensiones recibidas.
- `create_buffer(unsigned int width, unsigned int height)`: Recibe un ancho y una altura. Crea un *framebuffer* con las dimensiones recibidas.
- `bind()`: Activa el *framebuffer*.
- `unbind()`: Desactiva el *framebuffer*.

Código 6.12: Plantilla para crear *framebuffer*. Framebuffer.h.

```
class Framebuffer
{
public:
    Framebuffer(unsigned int width, unsigned int height);

    ~Framebuffer();

    void create_buffer(unsigned int width, unsigned int height) = 0;
    void bind() = 0;
    void unbind() = 0;
};

// Implementación del framebuffer
class ExampleFramebuffer: public Framebuffer { ... };
```

6.3.2.3. Shaders

La creación de programas de *shaders*, su uso y paso de variables son relativos a la API en uso. Su estructura se observa en el Código 6.13 y sus funciones son las siguientes:

- `Shader(const char* vertex_path, const char* fragment_path)`: Recibe la dirección de un archivo vs y fs con un *vertex shader* y *fragment shader* respectivamente. Inicializa un programa de *shaders* con los *shaders* recibidos.
- `use()`: Activa el shader.
- `setBool(const std::string& name, bool value) / setInt(const std::string& name, int value) / ...`: Recibe un nombre y un valor. Pasa al shader una variable de cierto valor a partir de su nombre.

Código 6.13: Plantilla para el manejo de shaders. Shader.h.

```
class Shader
{
public:
    Shader(const char* vertex_path, const char* fragment_path);

    ~Shader();

    void use();
    void setBool(const std::string& name, bool value);
    void setInt(const std::string& name, int value);
    // ...
};

// Implementación del manejo de shaders
class ExampleShader: public Shader { ... };
```

6.3.2.4. Luces

El manejo de luces depende de que el *shader* utilizado en el renderizado soporte su uso. En este trabajo se soportan tres tipos de luces: direccional, puntual y focal. Se implementa una luz direccional, una luz focal y cuatro luces puntuales posicionadas en las esquinas de un cuadrado centrado en el centro y paralelo al plazo XZ. Su estructura se observa en el Código 6.14 y sus funciones son las siguientes:

- `Light()`: Inicializa las luces.
- `update(Shader* shader, Camera* camera)`: Actualiza las variables relativas a las luces en el *shader*.

Código 6.14: Manejo de luces. Light.h.

```
class Light
{
public:
```

```
    // ...

    Light();

    ~Light();

    void update(Shader* shader, Camera* camera);
};
```

6.3.2.5. Cámara

La estructura de la cámara se observa en el Código 6.15 y sus funciones son las siguientes:

- `Camera(glm::vec3 position)`: Recibe una posición. Inicializa la cámara en la posición recibida.
- `get_view_matrix()`: Retorna la matriz de vista de la cámara.
- `process_keyboard(const char* direction, float delta_time)`: Recibe una dirección y un paso de tiempo. Mueve la cámara en la dirección recibida.
- `process_mouse_movement(float x_offset, float y_offset)`: Recibe una distancia horizontal y una distancia vertical. Mueve el ángulo de vista e inclinación de la cámara según los parámetros recibidos.
- `process_mouse_scroll(float y_offset)`: Recibe una distancia vertical. Modifica el zoom de la cámara según la distancia recibida.

Código 6.15: Manejo de la cámara. Camera.h.

```
class Camera
{
public:
    // ...

    Camera(glm::vec3 position);

    ~Camera();

    glm::mat4 get_view_matrix();
    void process_keyboard(const char* direction, float delta_time);
    void process_mouse_movement(float x_offset, float y_offset);
    void process_mouse_scroll(float y_offset);

    // ...
};
```

6.3.3. Interfaz

La capa de interfaz está encargada del dibujado y las interacciones de la interfaz de usuario. Tiene acceso a las capas de simulación y renderizado para modificar sus variables. La estructura de la capa se observa en el Código 6.16 y sus funciones son las siguientes

- `Interface(Renderer* renderer)`: Recibe la capa de renderizado. Inicializa la interfaz de usuario.
- `pre_render()`: Aplica las configuraciones y estilos de la interfaz.
- `render(Simulation* simulation, Renderer* renderer)`: Recibe la capa de simulación y la capa de renderizado. Define la estructura de la interfaz de usuario y su funcionalidad.
- `post_render()`: Dibuja la interfaz de usuario definida.
- `terminate()`: Elimina el contexto de ImGui.

Código 6.16: Capa de interfaz. Interfaz.h.

```
class Interface
{
public:
   Interface(Renderer* renderer);

   ~Interface();

   void pre_render();
   void render(Simulation* simulation, Renderer* renderer);
   void post_render();
   void terminate();
};
```

Se hace uso de la rama *docking* de ImGui que permite conectar y mover de manera sencilla distintos componentes de una interfaz. La configuración actual del programa se encuentra en el archivo `imgui.ini` y puede modificarse según las necesidades del usuario. Si este requiere añadir opciones sobre un nuevo módulo de la simulación o renderizado puede crear una nueva sección o añadirlas bajo una existente como se observa en el Código 6.17

Código 6.17: Componente de la interfaz. Interfaz.cpp.

```
void Interface::render(Simulation* simulation, Renderer* renderer)
{
   // ...

   ImGui::Begin("ExampleComponent");

   if (enable_example_options) {
      // ...
   }

   ImGui::End();

   // ...
}
```

La interfaz de usuario en este trabajo consta de seis componentes distribuidos como se observa en la Figura 6.5. Estos son:

- Controles: Controles de la simulación. Inicio, pausa, reinicio.
- Simulación: Renderizado de la simulación.
- Escena: Configuraciones referentes al renderizado. Cámara, *shaders* y luces.
- Objetos: Configuraciones referentes al objeto tela y objeto estático.
- Físicas: Configuraciones referentes a las etapas de la simulación.
- Información: Datos sobre la simulación en curso.

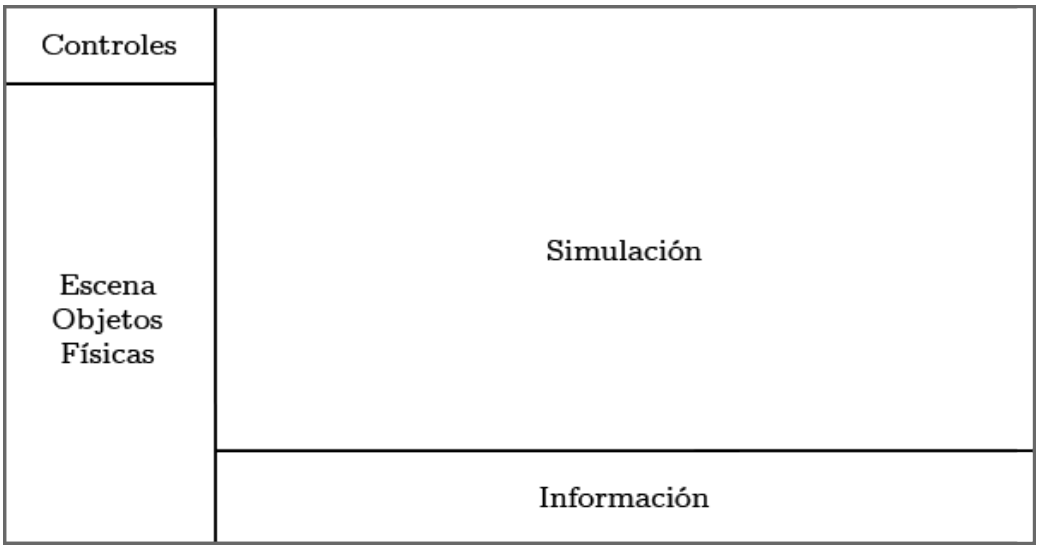


Figura 6.5: Estructura de la interfaz de usuario.

Capítulo 7

Conclusiones

En relación a las metas, este trabajo tenía como objetivo original, antes de llamarse como actualmente lo hace, el de adaptar la estrategia presentada en *Cloth Simulation with Triangular Mesh Adaptivity* [17] para su uso en GPU. Durante las primeras semanas de trabajo se hizo evidente que la complejidad de la tarea había sido subestimada y fue necesario cambiar de rumbo reduciendo su alcance. Tomando este contexto como punto de partida, no se puede decir que los objetivos iniciales fueron cumplidos. Sin embargo, los objetivos actuales, obtener el conocimiento y sentar las bases para poder implementar la estrategia mencionada y muchas otras, sí se han logrado cumplir.

A raíz de esto se presenta como crítica una práctica recurrente en esta clase de artículos, se da demasiado contenido por obvio y no se entra en detalles de implementación que muchas veces resultan críticos. La mención de un algoritmo, pero no su aplicación concreta para el contexto en cuestión, que puede no ser trivial, resulta frustrante y en un consumo de tiempo mayor del que podría parecer a primera vista.

Sobre el impacto de este trabajo, se espera que la teoría sirva de guía para quien desee entender lo básico de la simulación de telas, pero entregando nociones suficientes para comprender implementaciones más complejas. A su vez el programa desarrollado busca servir de referencia tanto para aquellos que quieran implementar una simulación por primera vez como para los que ya teniendo experiencia busquen un entorno donde probar nuevas estrategias de forma rápida.

Si bien se sientan bases para la comprensión general de la simulación de telas, el desarrollo del programa está lejos de estar completo y tiene un muy amplio rango de mejora. A nivel de arquitectura, las interacciones entre capas y el nivel de encapsulamiento general es demasiado público y debería tener más restricciones. A nivel de datos, existe un problema de precisión generado por la interacción de la capa de interfaz que no soporta tipos *double*. La interacción con el usuario también podría ser más intuitiva y permitir, por ejemplo, definir la forma de la iteración o subciclo a utilizar.

Además de lo aprendido referente a la simulación de telas, a lo largo de este trabajo se han podido comprender mejor los conceptos que ya se tenían de computación gráfica, se reforzaron las bases de cálculo y se afianzó la lectura de artículos científicos. También se mejoró en el dominio del lenguaje C++ para el desarrollo de programas.

Capítulo 8

Trabajo futuro

Como se mencionó en el capítulo anterior, el desarrollo del programa tiene un amplio rango de mejora, por ejemplo:

- Encapsular mejor la capa de renderizado de tal forma de minimizar la cantidad de sus módulos que requieren reimplementarse para añadir nuevas APIs gráficas. En particular, que la capa en sí no sea una clase abstracta.
- Aplicar alguna estrategia tal que no se requiera tener las variables de las capas públicas para que la capa de interfaz pueda acceder a ellas.
- Aplicar alguna estrategia que sirva de intermediario entre las capas de simulación e interfaz para no obligar a la capa de simulación a utilizar tipos *float* y luego transformarlos a *double* para ser compatible con la capa de interfaz.
- Aplicar alguna estrategia que permita asociar interfaces individuales a componentes de manera que no se deban añadir manualmente a la capa de interfaz.
- Ser capaz de considerar fuerzas de roce entre objetos.
- Ser capaz de considerar fuerzas externas como viento o interacciones del usuario.
- Añadir colisiones de tipo tela - tela.
- Implementar pasos de tiempo adaptativos.
- Implementar mallas cuadradas.
- Definir iteraciones y subciclos mediante interfaz, programado con estilo *blueprints*.
- Utilizar un *shader* para manejar las superficies que deberían estar ocultas.
- Utilizar distintos hilos para las capas de tal forma que no se genere un cuello de botella en la simulación, ni se detenga al generarse intestabilidad numérica o problemas de precisión.

Bibliografía

[1] Weidner, N. J., Piddington, K., Levin, D. I. W., y Sueda, S., "Eulerian-on-lagrangian cloth simulation," ACM Trans. Graph., vol. 37, no. 4, p. 50, 2018, doi:10.1145/3197517.3201281.

[2] Tan, X. y Guo, R., "An improved mean curvature-based bending model for cloth simulation," Int. J. Inf. Technol. Manag., vol. 20, no. 1/2, pp. 95–109, 2021, doi:10.1504/IJITM.2021.114158.

[3] Li, J., Daviet, G., Narain, R., Bertails-Descoubes, F., Overby, M., Brown, G. E., y Boissieux, L., "An implicit frictional contact solver for adaptive cloth simulation," ACM Trans. Graph., vol. 37, no. 4, p. 52, 2018, doi:10.1145/3197517.3201308.

[4] Casafranca, J. J., Cirio, G., Rodríguez, A., Miguel, E., y Otaduy, M. A., "Mixing yarns and triangles in cloth simulation," Comput. Graph. Forum, vol. 39, no. 2, pp. 101–110, 2020, doi:10.1111/cgf.13915.

[5] Wang, Z., Wu, L., Fratarcangeli, M., Tang, M., y Wang, H., "Parallel multigrid for nonlinear cloth simulation," Comput. Graph. Forum, vol. 37, no. 7, pp. 131–141, 2018, doi:10.1111/cgf.13554.

[6] Tang, M., Wang, T., Liu, Z., Tong, R., y Manocha, D., "I-cloth: incremental collision handling for gpu-based interactive cloth simulation," ACM Trans. Graph., vol. 37, no. 6, p. 204, 2018, doi:10.1145/3272127.3275005.

[7] Li, C., Tang, M., Tong, R., Cai, M., Zhao, J., y Manocha, D., "P-cloth: interactive complex cloth simulation on multi-gpu systems using dynamic matrix assembly and pipelined implicit integrators," ACM Trans. Graph., vol. 39, no. 6, pp. 180:1–180:15, 2020, doi:10.1145/3414685.3417763.

[8] Liang, J. y Lin, M. C., "Time-domain parallelization for accelerating cloth simulation," Comput. Graph. Forum, vol. 37, no. 8, pp. 21–34, 2018, doi:10.1111/cgf.13509.

[9] Wang, H., "Gpu-based simulation of cloth wrinkles at submillimeter levels," ACM Trans. Graph., vol. 40, no. 4, pp. 169:1–169:14, 2021, doi:10.1145/3450626.3459787.

[10] Oh, Y., Lee, T. M., y Lee, I., "Hierarchical cloth simulation using deep neural networks," en Proceedings of Computer Graphics International 2018, CGI 2018, Bintan Island, Indonesia, June 11-14, 2018 (Magnenat-Thalmann, N., Kim, J., Rushmeier, H. E., Lévy, B., Zhang, H. R., y Thalmann, D., eds.), pp. 139–146, ACM, 2018, doi:10.1145/3208159.3208162.

[11] Lee, T. M., Oh, Y., y Lee, I., "Efficient cloth simulation using miniature cloth and upscaling deep neural networks," CoRR, vol. abs/1907.03953, 2019, http://arxiv.org/abs/1907.03953.

[12] Lewin, C., "Swish: Neural network cloth simulation on madden NFL 21," en SIGGRAPH 2021: Special Interest Group on Computer Graphics and Interactive Techniques Conference, Talks, Virtual Event, USA, August 9-13, 2021, pp. 23:1–23:2, ACM, 2021, doi:10.1145/3450623.3464665.

[13] Villard, J. y Borouchaki, H., "Adaptive meshing for cloth animation," Eng. Comput., vol. 20, no. 4, pp. 333–341, 2005, doi:10.1007/s00366-005-0302-1.

[14] Macklin, M., Storey, K., Lu, M., Terdiman, P., Chentanez, N., Jeschke, S., y Müller, M., "Small steps in physics simulation," en Proceedings of the 18th annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA 2019, Los Angeles, CA, USA, July 26-28, 2019 (Lee, S., Schroeder, C. A., Spencer, S. N., Batty, C., y Huang, J., eds.), pp. 2:1–2:7, ACM, 2019, doi:10.1145/3309486.3340247.

[15] Baraff, D. y Witkin, A. P., "Large steps in cloth simulation," en Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1998, Orlando, FL, USA, July 19-24, 1998 (Cunningham, S., Bransford, W., y Cohen, M. F., eds.), pp. 43–54, ACM, 1998, doi:10.1145/280814.280821.

[16] Pritchard, D., "Implementing baraff and witkin's cloth simulation," 2003, http://davidpritchard.org/freecloth/docs/report.pdf.

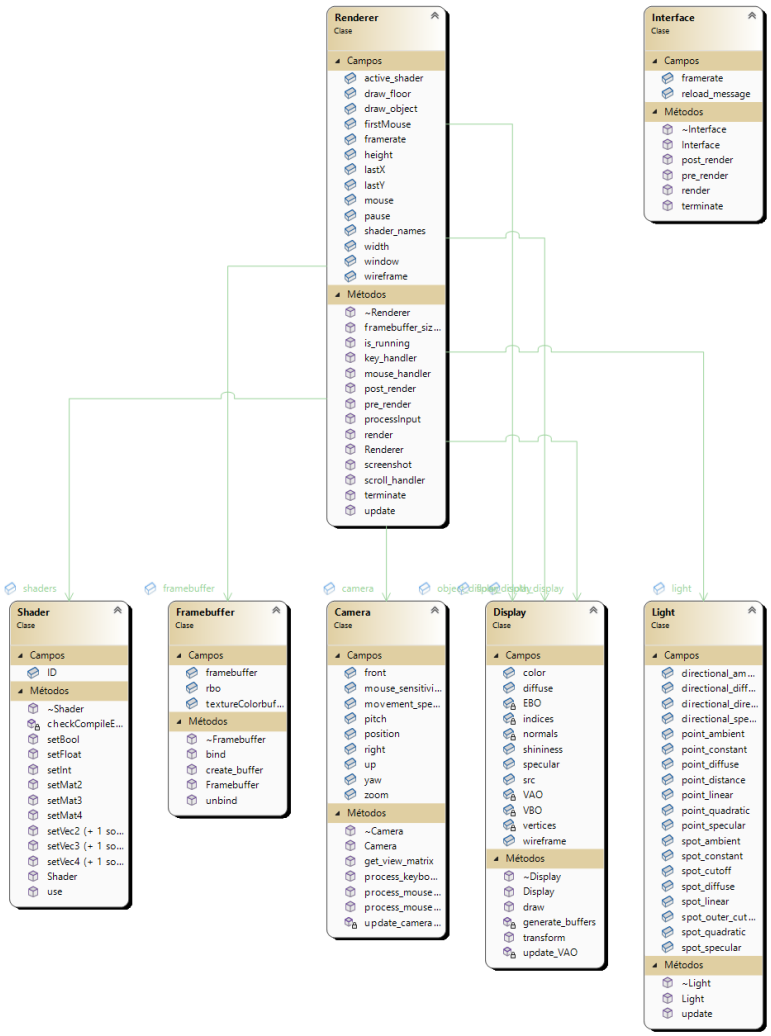
[17] de Oliveira, S. M. F., Vidal, C. A., Neto, J. B. C., Carvalho, L. L. D., y Maia, J. G. R., "Cloth simulation with triangular mesh adaptivity," en 27th SIBGRAPI Conference on Graphics, Patterns and Images, SIBGRAPI 2014, Rio de Janeiro, Brazil, August 27-30, 2014, pp. 235–242, IEEE Computer Society, 2014, doi:10.1109/SIBGRAPI.2014.20.

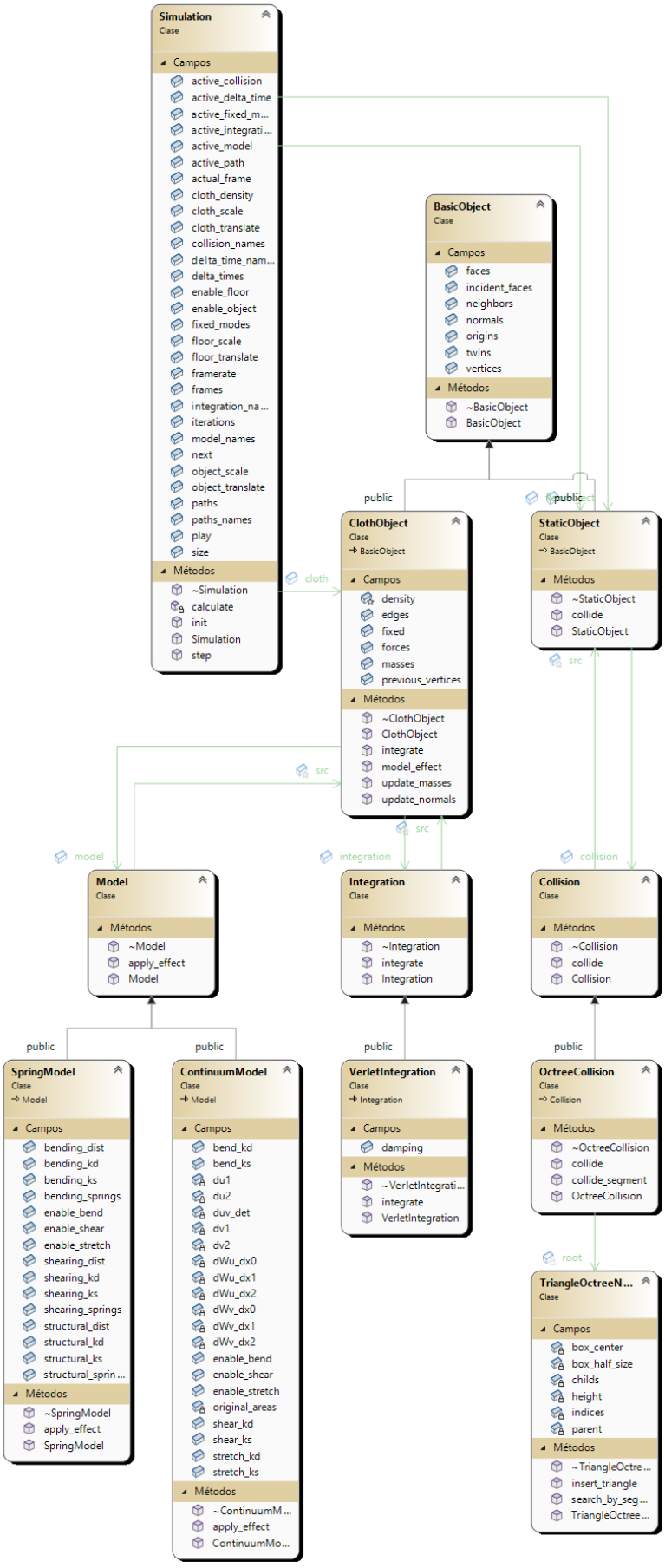
[18] Akenine-Möller, T., "Fast 3d triangle-box overlap testing," en International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2005, Los Angeles, California, USA, July 31 - August 4, 2005, Courses (Fujii, J., ed.), p. 8, ACM, 2005, doi:10.1145/1198555.1198747.

[19] Möller, T. y Trumbore, B., "Fast, minimum storage ray-triangle intersection," J. Graphics, GPU, & Game Tools, vol. 2, no. 1, pp. 21–28, 1997, doi:10.1080/10867651.1997.10487468.

Anexos

Anexo A. Diagramas de clase




Simulation
Clase
Campos
active_collision
active_delta_time
active_fixed_m...
active_integrati...
active_model
active_path
actual_frame
cloth_density
cloth_scale
cloth_translate
collision_names
delta_time_nam...
delta_times
enable_floor
enable_object
fixed_modes
floor_scale
floor_translate
framerate
frames
integration_na...
iterations
model_names
next
object_scale
object_translate
paths
paths_names
play
size
Métodos
~Simulation
calculate
init
Simulation
step
cloth
BasicObject
Clase
Campos
faces
incident_faces
neighbors
normals
origins
twins
vertices
Métodos
~BasicObject
BasicObject
public
public
ClothObject
Clase
BasicObject
Campos
density
edges
fixed
forces
masses
previous_vertices
Métodos
~ClothObject
ClothObject
integrate
model_effect
update_masses
update_normals
StaticObject
Clase
BasicObject
Métodos
~StaticObject
collide
StaticObject
src
src
src
model
integration
collision
Model
Clase
Métodos
~Model
apply_effect
Model
Integration
Clase
Métodos
~Integration
integrate
Integration
Collision
Clase
Métodos
~Collision
collide
Collision
public
public
public
public
SpringModel
Clase
Model
Campos
bending_dist
bending_kd
bending_ks
bending_springs
enable_bend
enable_shear
enable_stretch
shearing_dist
shearing_kd
shearing_ks
shearing_springs
structural_dist
structural_kd
structural_ks
structural_sprin...
Métodos
~SpringModel
apply_effect
SpringModel
ContinuumModel
Clase
Model
Campos
bend_kd
bend_ks
du1
du2
duv_det
dv1
dv2
dWu_dx0
dWu_dx1
dWu_dx2
dWv_dx0
dWv_dx1
dWv_dx2
enable_bend
enable_shear
enable_stretch
original_areas
shear_kd
shear_ks
stretch_kd
stretch_ks
Métodos
~ContinuumM...
apply_effect
ContinuumMo...
VerletIntegration
Clase
Integration
Campos
damping
Métodos
~VerletIntegrati...
integrate
VerletIntegration
OctreeCollision
Clase
Collision
Métodos
~OctreeCollision
collide
collide_segment
OctreeCollision
root
TriangleOctreeN...
Clase
Campos
box_center
box_half_size
childs
height
indices
parent
Métodos
~TriangleOctre...
insert_triangle
search_by_seg...
TriangleOctree...

Anexo B. Intersección entre primitivas

B.1. Triángulo - Plano (Test)

Código B.1: Test de intersección triángulo - plano. Geometry.cpp.

```
// Test de intersección plano - caja
bool plane_box_overlap(glm::dvec3 plane_n, glm::dvec3 plane_p, glm::dvec3 box_half_size)
{
   glm::dvec3 plane_min, plane_max;

   for (int i = 0; i < 3; ++i)
   {
      if (plane_n[i] > 0.0)
      {
         plane_min[i] = -1.0 * box_half_size[i] - plane_p[i];
         plane_max[i] = box_half_size[i] - plane_p[i];
      }
      else
      {
         plane_min[i] = box_half_size[i] - plane_p[i];
         plane_max[i] = -1.0 * box_half_size[i] - plane_p[i];
      }
   }

   if (glm::dot(plane_n, plane_min) > 0.0)
   {
      return false;
   }

   if (glm::dot(plane_n, plane_max) >= 0.0)
   {
      return true;
   }

   return false;
}
```

B.2. Triángulo - Caja (Test)

El test de intersección triángulo - caja utilizado en la construcción del *octree* para el manejo de colisiones se realiza siguiente el artículo *Fast 3D Triangle-Box Overlap Testing* [18].

Código B.2: Test de intersección triángulo - caja. Geometry.cpp

```
// Test de ejes
bool axis_test(double a, double b, double v11, double v12, double v21, double v22, double
    ↪ bhs1, double bhs2)
{
  double p1 = a * v11 + b * v12;
  double p2 = a * v21 + b * v22;
```

```
6
7     double min, max;
8
9     if (p1 < p2)
10    {
11      min = p1;
12      max = p2;
13    }
14    else
15    {
16      min = p2;
17      max = p1;
18    }
19
20    double rad = std::fabsf(a) * bhs1 + std::fabsf(b) * bhs2;
21
22    if (min > rad || max < -rad)
23    {
24      return false;
25    }
26    else
27    {
28      return true;
29    }
30  }
31
32  // Caja que contiene a un triángulo
33  std::pair<glm::dvec3, glm::dvec3> triangle_bounding_box(glm::dvec3 triangle_p1, glm::dvec3
        ↪ triangle_p2, glm::dvec3 triangle_p3)
34  {
35    glm::dvec3 min = glm::min(glm::min(triangle_p1, triangle_p2), triangle_p3);
36    glm::dvec3 max = glm::max(glm::max(triangle_p1, triangle_p2), triangle_p3);
37
38    return std::pair<glm::dvec3, glm::dvec3>(min, max);
39  };
40
41  // Test de intersección triángulo - caja
42  bool triangle_box_overlap(glm::dvec3 triangle_p1, glm::dvec3 triangle_p2, glm::dvec3
        ↪ triangle_p3, glm::dvec3 box_center, glm::dvec3 box_half_size)
43  {
44    // Mover el triángulo al espacio de la caja
45    triangle_p1 = triangle_p1 - box_center;
46    triangle_p2 = triangle_p2 - box_center;
47    triangle_p3 = triangle_p3 - box_center;
48
49    // Arcos del triángulo
50    glm::dvec3 triangle_e1 = triangle_p2 - triangle_p1;
51    glm::dvec3 triangle_e2 = triangle_p3 - triangle_p2;
52    glm::dvec3 triangle_e3 = triangle_p1 - triangle_p3;
53
54    // Bullet 3
55    if (!axis_test(triangle_e1.z, triangle_e1.y, triangle_p1.y, -1.0 * triangle_p1.z, triangle_p3.
```

```
    ↪ y, -1.0 * triangle_p3.z, box_half_size.y, box_half_size.z))
56  {
57    return false;
58  }
59  else if (!axis_test(triangle_e1.z, triangle_e1.x, -1.0 * triangle_p1.x, triangle_p1.z, -1.0 *
    ↪ triangle_p3.x, triangle_p3.z, box_half_size.x, box_half_size.z))
60  {
61    return false;
62  }
63  else if (!axis_test(triangle_e1.y, triangle_e1.x, triangle_p2.x, -1.0 * triangle_p2.y,
    ↪ triangle_p3.x, -1.0 * triangle_p3.y, box_half_size.x, box_half_size.y))
64  {
65    return false;
66  }
67  else if (!axis_test(triangle_e2.z, triangle_e2.y, triangle_p1.y, -1.0 * triangle_p1.z,
    ↪ triangle_p3.y, -1.0 * triangle_p3.z, box_half_size.y, box_half_size.z))
68  {
69    return false;
70  }
71  else if (!axis_test(triangle_e2.z, triangle_e2.x, -1.0 * triangle_p1.x, triangle_p1.z, -1.0 *
    ↪ triangle_p3.x, triangle_p3.z, box_half_size.x, box_half_size.z))
72  {
73    return false;
74  }
75  else if (!axis_test(triangle_e2.y, triangle_e2.x, triangle_p1.x, -1.0 * triangle_p1.y,
    ↪ triangle_p2.x, -1.0 * triangle_p2.y, box_half_size.x, box_half_size.y))
76  {
77    return false;
78  }
79  else if (!axis_test(triangle_e3.z, triangle_e3.y, triangle_p1.y, -1.0 * triangle_p1.z,
    ↪ triangle_p2.y, -1.0 * triangle_p2.z, box_half_size.y, box_half_size.z))
80  {
81    return false;
82  }
83  else if (!axis_test(triangle_e3.z, triangle_e3.x, -1.0 * triangle_p1.x, triangle_p1.z, -1.0 *
    ↪ triangle_p2.x, triangle_p2.z, box_half_size.x, box_half_size.z))
84  {
85    return false;
86  }
87  else if (!axis_test(triangle_e3.y, triangle_e3.x, triangle_p2.x, -1.0 * triangle_p2.y,
    ↪ triangle_p3.x, -1.0 * triangle_p3.y, box_half_size.x, box_half_size.y))
88  {
89    return false;
90  }
91
92  // Bullet 1
93  std::pair<glm::dvec3, glm::dvec3> bounding_box = triangle_bounding_box(triangle_p1,
    ↪ triangle_p2, triangle_p3);
94  glm::dvec3 min_bounding_box = bounding_box.first;
95  glm::dvec3 max_bounding_box = bounding_box.second;
96
97  if (glm::any(glm::greaterThan(min_bounding_box, box_half_size)) || glm::any(glm::
```

```
    ↪ lessThan(max_bounding_box, -1.0 * box_half_size)))
98  {
99    return false;
100 }
101
102 // Bullet 2
103 glm::dvec3 triangle_n = glm::cross(triangle_e1, triangle_e2);
104
105 if (!plane_box_overlap(triangle_n, triangle_p1, box_half_size))
106 {
107   return false;
108 }
109
110 return true;
111 }
```

B.3. Segmento - Caja (Test)

El test segmento - caja se realiza utilizando el teorema del eje de separación (*separating axis theorem*) como se muestra en el artículo semanal de 3D Kingdoms.

Código B.3: Test de intersección segmento - caja. Geometry.cpp.

```
1  bool segment_box_overlap(glm::dvec3 segment_start, glm::dvec3 segment_end, glm::dvec3
       ↪ box_center, glm::dvec3 box_half_size)
2  {
3    // Mover el segmento al espacio de la caja
4    glm::dvec3 LB1 = segment_start - box_center;
5    glm::dvec3 LB2 = segment_end - box_center;
6
7    // Centro y extensión
8    glm::dvec3 LMid = (LB1 + LB2) * 0.5;
9    glm::dvec3 L = (LB1 - LMid);
10   glm::dvec3 LExt = glm::dvec3(fabs(L.x), fabs(L.y), fabs(L.z));
11
12   // Teorema de separación de ejes
13   if (fabs(LMid.x) > box_half_size.x + LExt.x) return false;
14   if (fabs(LMid.y) > box_half_size.y + LExt.y) return false;
15   if (fabs(LMid.z) > box_half_size.z + LExt.z) return false;
16
17   if (fabs(LMid.y * L.z - LMid.z * L.y) > (box_half_size.y * LExt.z + box_half_size.z *
       ↪ LExt.y)) return false;
18   if (fabs(LMid.x * L.z - LMid.z * L.x) > (box_half_size.x * LExt.z + box_half_size.z *
       ↪ LExt.x)) return false;
19   if (fabs(LMid.x * L.y - LMid.y * L.x) > (box_half_size.x * LExt.y + box_half_size.y *
       ↪ LExt.x)) return false;
20
21   return true;
22 }
```

B.4. Segmento - Triángulo (Intersección)

El punto de intersección segmento - triángulo se calcula adaptando el algoritmo presentado en *Fast, Minimum Storage Ray-Triangle Intersection* [19].

Código B.4: Intersección segmento - triángulo. Geometry.cpp.

```
std::tuple<bool, glm::dvec3, double> segment_triangle_overlap(glm::dvec3 segment_start,
    ↪ glm::dvec3 segment_end, glm::dvec3 triangle_p1, glm::dvec3 triangle_p2, glm::dvec3
    ↪ triangle_p3)
{
  double EPSILON = 0.000000000001;

  glm::dvec3 ray_vector = segment_end - segment_start;

  glm::dvec3 triangle_e1 = triangle_p2 - triangle_p1;
  glm::dvec3 triangle_e2 = triangle_p3 - triangle_p1;

  glm::dvec3 h = glm::cross(ray_vector, triangle_e2);
  double a = glm::dot(triangle_e1, h);

  if (a > -EPSILON && a < EPSILON)
  {
    return { false, glm::dvec3(), -1 };
  }

  double f = 1.0 / a;
  glm::dvec3 s = segment_start - triangle_p1;
  double u = f * glm::dot(s, h);

  if (u < 0.0 || u > 1.0)
  {
    return { false, glm::dvec3(), -1 };
  }

  glm::dvec3 q = glm::cross(s, triangle_e1);
  double v = f * glm::dot(ray_vector, q);

  if (v < 0.0 || u + v > 1.0)
  {
    return { false, glm::dvec3(), -1 };
  }

  double t = f * glm::dot(triangle_e2, q);

  if (t > EPSILON)
  {
    glm::dvec3 ray_intersection = segment_start + ray_vector * t;
    double distance2_intersection = glm::distance2(segment_start, ray_intersection);
    double distance2_segment = glm::distance2(segment_start, segment_end);

    if (distance2_intersection < distance2_segment)
```

```
44      {
45        return { true, ray_intersection, distance2_intersection };
46      }
47      else
48      {
49        return { false, glm::dvec3(), -1 };
50      }
51    }
52    else
53    {
54      return { false, glm::dvec3(), -1 };
55    }
56  }
```

Anexo C. Configuración utilizada por Figura

C.1. Figura 4.1

Variable	Valor
Paso de tiempo	0.016
Subciclos	10
Frames	0/600
Dimensión de la tela	71^2
Densidad de la tela	0.2
Modelo	M. continuas
k_s estiramiento	500
k_d estiramiento	0.2
k_s cizallamiento	50
k_d cizallamiento	0.2
k_s plegado	-
k_d plegado	-
Colisión	Octree
Integración	Verlet
k_d global	0.01

C.2. Figura 4.7

Variable	Valor
Paso de tiempo	0.016
Iteraciones	1/5/10
Frames	1200
Dimensión de la tela	71^2
Densidad de la tela	0.2
Modelo	M - R (restr.)
k_s estiramiento	-
k_d estiramiento	0.0
k_s cizallamiento	-
k_d cizallamiento	0.0
k_s plegado	-
k_d plegado	0.0
Colisión	Octree
Integración	Verlet
k_d global	0.01

C.3. Figura 4.8

Variable	Valor
Paso de tiempo	0.016
Subciclos	10
Frames	1200
Dimensión de la tela	$10^2/50^2/100^2$
Densidad de la tela	0.2
Modelo	M - R (restr.)
k_s estiramiento	-
k_d estiramiento	0.0
k_s cizallamiento	-
k_d cizallamiento	0.0
k_s plegado	-
k_d plegado	0.0
Colisión	Octree
Integración	Verlet
k_d global	0.01

C.4. Figura 4.9

Variable	Valor
Paso de tiempo	0.016
Subciclos	10
Frames	1200
Dimensión de la tela	$10^2/50^2/100^2$
Densidad de la tela	0.2
Modelo	M - R (fuerzas)
k_s estiramiento	500
k_d estiramiento	0.0
k_s cizallamiento	50
k_d cizallamiento	0.0
k_s plegado	0.00001
k_d plegado	0.0
Colisión	Octree
Integración	Verlet
k_d global	0.01

C.5. Figura 4.12

Variable	Valor
Paso de tiempo	0.016
Subciclos	10
Frames	600
Dimensión de la tela	71^2
Densidad de la tela	0.2
Modelo	M. continuas
k_s estiramiento	500
k_d estiramiento	0.2
k_s cizallamiento	50
k_d cizallamiento	0.2
k_s plegado	0.00001/-
k_d plegado	0.2/-
Colisión	Octree
Integración	Verlet
k_d global	0.01

C.6. Figura 4.17

Variable	Valor
Paso de tiempo	0.016
Subciclos	10
Frames	0/300/1000
Dimensión de la tela	71^2
Densidad de la tela	0.2
Modelo	M. continuas
k_s estiramiento	500
k_d estiramiento	0.2
k_s cizallamiento	50
k_d cizallamiento	0.2
k_s plegado	-
k_d plegado	-
Colisión	Octree
Integración	Verlet
k_d global	0.01

C.7. Figura 4.18

Variable	Valor
Paso de tiempo	0.016
Subciclos	10
Frames	220/300/380
Dimensión de la tela	71^2
Densidad de la tela	0.2
Modelo	M. continuas
k_s estiramiento	500
k_d estiramiento	0.2
k_s cizallamiento	50
k_d cizallamiento	0.2
k_s plegado	-
k_d plegado	-
Colisión	Octree
Integración	Verlet
k_d global	0.01

C.8. Figura 5.1

Variable	Valor
Paso de tiempo	0.016
Subciclos/Iteraciones	10
Frames	600
Dimensión de la tela	11^2->71^2
Densidad de la tela	0.2
Modelo	-
k_s estiramiento	500
k_d estiramiento	0.2
k_s cizallamiento	50
k_d cizallamiento	0.2
k_s plegado	0.00001/ -
k_d plegado	0.2/ -
Colisión	Octree
Integración	Verlet
k_d global	0.01

Anexo D. Código fuente

D.1. Modelos de fuerzas y restricciones

```
class Model
{
public:
  Model(ClothObject& src);

  virtual ~Model();

  virtual void apply_effect(double dt) = 0;

protected:
  ClothObject& src;
};

Model::Model(ClothObject& src) : src(src) {}

Model::~Model() {}
```

```
Model::Model(ClothObject& src) : src(src) {}

Model::~Model()
{
}
```

D.1.1. Masa - Resorte

```
class SpringModelRestriction : public Model
{
public:
  SpringModelRestriction(ClothObject& src);

  ~SpringModelRestriction();

  void apply_effect(double dt);

  std::vector<glm::uvec2> structural_springs;
  std::vector<glm::uvec2> shearing_springs;
  std::vector<glm::uvec2> bending_springs;

  bool enable_stretch = true;
  bool enable_shear = true;
  bool enable_bend = true;

  double structural_dist;
  float structural_ks = 500;
  float structural_kd = 0.2;
```

```
21
22    double shearing_dist;
23    float shearing_ks = 50;
24    float shearing_kd = 0.2;
25
26    double bending_dist;
27    float bending_ks = 0.00001;
28    float bending_kd = 0.2;
29 };
30
31 class SpringModelForce : public Model
32 {
33 public:
34    SpringModelForce(ClothObject& src);
35
36    ~SpringModelForce();
37
38    void apply_effect(double dt);
39
40    std::vector<glm::uvec2> structural_springs;
41    std::vector<glm::uvec2> shearing_springs;
42    std::vector<glm::uvec2> bending_springs;
43
44    bool enable_stretch = true;
45    bool enable_shear = true;
46    bool enable_bend = true;
47
48    double structural_dist;
49    float structural_ks = 500;
50    float structural_kd = 0.2;
51
52    double shearing_dist;
53    float shearing_ks = 50;
54    float shearing_kd = 0.2;
55
56    double bending_dist;
57    float bending_ks = 0.00001;
58    float bending_kd = 0.2;
59 };
```

```
1  SpringModelRestriction::SpringModelRestriction(ClothObject& src) : Model(src)
2  {
3     int size = sqrt(src.vertices.size());
4
5     this->structural_springs = std::vector<glm::uvec2>(2 * size * (size - 1));
6     this->shearing_springs = std::vector<glm::uvec2>(2 * (size - 1) * (size - 1));
7     this->bending_springs = std::vector<glm::uvec2>(2 * size * (size - 2));
8
9     for (unsigned int i = 0; i < size; ++i)
10    {
11       for (unsigned int j = 0; j < size - 1; ++j)
```

```
12      {
13        this->structural_springs[2 * (i * (size - 1) + j) + 0] = glm::uvec2(i * size + j, i * size
    ↪ + j + 1);
14        this->structural_springs[2 * (i * (size - 1) + j) + 1] = glm::uvec2(j * size + i, (j + 1)
    ↪ * size + i);
15      }
16    }
17
18    for (unsigned int i = 0; i < size - 1; ++i)
19    {
20      for (unsigned int j = 0; j < size - 1; ++j)
21      {
22        this->shearing_springs[2 * (i * (size - 1) + j) + 0] = glm::uvec2(i * size + j, (i + 1) *
    ↪ size + j + 1);
23        this->shearing_springs[2 * (i * (size - 1) + j) + 1] = glm::uvec2(i * size + j + 1, (i +
    ↪ 1) * size + j);
24      }
25    }
26
27    for (unsigned int i = 0; i < size; ++i)
28    {
29      for (unsigned int j = 0; j < size - 2; ++j)
30      {
31        this->bending_springs[2 * (i * (size - 2) + j) + 0] = glm::uvec2(i * size + j, i * size +
    ↪ j + 2);
32        this->bending_springs[2 * (i * (size - 2) + j) + 1] = glm::uvec2(j * size + i, (j + 2) *
    ↪ size + i);
33      }
34    }
35
36    structural_dist = glm::l2Norm(src.vertices[this->structural_springs[0].x], src.vertices[this->
    ↪ structural_springs[0].y]);
37    shearing_dist = glm::l2Norm(src.vertices[this->shearing_springs[0].x], src.vertices[this->
    ↪ shearing_springs[0].y]);
38    bending_dist = glm::l2Norm(src.vertices[this->bending_springs[0].x], src.vertices[this->
    ↪ bending_springs[0].y]);
39  }
40
41  SpringModelRestriction::~SpringModelRestriction()
42  {
43  }
44
45  void SpringModelRestriction::apply_effect(double dt)
46  {
47    for (int i = 0; i < src.vertices.size(); ++i)
48    {
49      src.forces[i] = glm::dvec3(0.0, -9.81 * src.masses[i], 0.0);
50    }
51
52    if (enable_shear) {
53      for (int i = 0; i < shearing_springs.size(); i++)
54      {
```

```
55        glm::dvec3 x1 = src.vertices[shearing_springs[i].x];
56        glm::dvec3 x2 = src.vertices[shearing_springs[i].y];
57
58        glm::dvec3 correction = (1 - shearing_kd) * (glm::length(x2 - x1) - shearing_dist) * (
    ↪ x2 - x1) / glm::length(x2 - x1);
59
60        if (src.fixed[shearing_springs[i].x] && src.fixed[shearing_springs[i].y]) {
61          continue;
62        }
63        else if (src.fixed[shearing_springs[i].x]) {
64          src.vertices[shearing_springs[i].y] -= correction;
65        }
66        else if (src.fixed[shearing_springs[i].y]) {
67          src.vertices[shearing_springs[i].x] += correction;
68        } else {
69          src.vertices[shearing_springs[i].x] += correction * 0.5;
70          src.vertices[shearing_springs[i].y] -= correction * 0.5;
71        }
72      }
73    }
74
75    if (enable_stretch) {
76      for (int i = 0; i < structural_springs.size(); i++)
77      {
78        glm::dvec3 x1 = src.vertices[structural_springs[i].x];
79        glm::dvec3 x2 = src.vertices[structural_springs[i].y];
80
81        glm::dvec3 correction = (1 - shearing_kd) * (glm::length(x2 - x1) - structural_dist) *
    ↪ (x2 - x1) / glm::length(x2 - x1);
82
83        if (src.fixed[structural_springs[i].x] && src.fixed[structural_springs[i].y])
84        {
85          continue;
86        }
87        else if (src.fixed[structural_springs[i].x])
88        {
89          src.vertices[structural_springs[i].y] -= correction;
90        }
91        else if (src.fixed[structural_springs[i].y])
92        {
93          src.vertices[structural_springs[i].x] += correction;
94        }
95        else
96        {
97          src.vertices[structural_springs[i].x] += correction * 0.5;
98          src.vertices[structural_springs[i].y] -= correction * 0.5;
99        }
100     }
101   }
102
103   if (enable_bend) {
104     for (int i = 0; i < bending_springs.size(); i++)
```

```
105     {
106       glm::dvec3 x1 = src.vertices[bending_springs[i].x];
107       glm::dvec3 x2 = src.vertices[bending_springs[i].y];
108 
109       glm::dvec3 correction = (1 - shearing_kd) * (glm::length(x2 - x1) - bending_dist) * (
    ↪ x2 - x1) / glm::length(x2 - x1);
110 
111       if (src.fixed[bending_springs[i].x] && src.fixed[bending_springs[i].y])
112       {
113         continue;
114       }
115       else if (src.fixed[bending_springs[i].x])
116       {
117         src.vertices[bending_springs[i].y] -= correction;
118       }
119       else if (src.fixed[bending_springs[i].y])
120       {
121         src.vertices[bending_springs[i].x] += correction;
122       }
123       else
124       {
125         src.vertices[bending_springs[i].x] += correction * 0.5;
126         src.vertices[bending_springs[i].y] -= correction * 0.5;
127       }
128     }
129   }
130 }
131 
132 SpringModelForce::SpringModelForce(ClothObject& src) : Model(src)
133 {
134   int size = sqrt(src.vertices.size());
135 
136   this->structural_springs = std::vector<glm::uvec2>(2 * size * (size - 1));
137   this->shearing_springs = std::vector<glm::uvec2>(2 * (size - 1) * (size - 1));
138   this->bending_springs = std::vector<glm::uvec2>(2 * size * (size - 2));
139 
140   for (unsigned int i = 0; i < size; ++i)
141   {
142     for (unsigned int j = 0; j < size - 1; ++j)
143     {
144       this->structural_springs[2 * (i * (size - 1) + j) + 0] = glm::uvec2(i * size + j, i * size
    ↪ + j + 1);
145       this->structural_springs[2 * (i * (size - 1) + j) + 1] = glm::uvec2(j * size + i, (j + 1)
    ↪ * size + i);
146     }
147   }
148 
149   for (unsigned int i = 0; i < size - 1; ++i)
150   {
151     for (unsigned int j = 0; j < size - 1; ++j)
152     {
153       this->shearing_springs[2 * (i * (size - 1) + j) + 0] = glm::uvec2(i * size + j, (i + 1) *
```

```
    ↪ size + j + 1);
154       this->shearing_springs[2 * (i * (size - 1) + j) + 1] = glm::uvec2(i * size + j + 1, (i +
    ↪ 1) * size + j);
155     }
156   }
157
158   for (unsigned int i = 0; i < size; ++i)
159   {
160     for (unsigned int j = 0; j < size - 2; ++j)
161     {
162       this->bending_springs[2 * (i * (size - 2) + j) + 0] = glm::uvec2(i * size + j, i * size +
    ↪ j + 2);
163       this->bending_springs[2 * (i * (size - 2) + j) + 1] = glm::uvec2(j * size + i, (j + 2) *
    ↪ size + i);
164     }
165   }
166
167   structural_dist = glm::l2Norm(src.vertices[this->structural_springs[0].x], src.vertices[this->
    ↪ structural_springs[0].y]);
168   shearing_dist = glm::l2Norm(src.vertices[this->shearing_springs[0].x], src.vertices[this->
    ↪ shearing_springs[0].y]);
169   bending_dist = glm::l2Norm(src.vertices[this->bending_springs[0].x], src.vertices[this->
    ↪ bending_springs[0].y]);
170 }
171
172 SpringModelForce::~SpringModelForce()
173 {
174 }
175
176 void SpringModelForce::apply_effect(double dt)
177 {
178   for (int i = 0; i < src.vertices.size(); ++i)
179   {
180     src.forces[i] = glm::dvec3(0.0, -9.81 * src.masses[i], 0.0);
181   }
182
183   if (enable_shear) {
184     for (int i = 0; i < shearing_springs.size(); i++)
185     {
186       glm::dvec3 x1 = src.vertices[shearing_springs[i].x];
187       glm::dvec3 x2 = src.vertices[shearing_springs[i].y];
188
189       glm::dvec3 v1 = (x1 - src.previous_vertices[shearing_springs[i].x]) / dt;
190       glm::dvec3 v2 = (x2 - src.previous_vertices[shearing_springs[i].y]) / dt;
191
192       glm::dvec3 dx = x1 - x2;
193       glm::dvec3 dv = v1 - v2;
194
195       glm::dvec3 f = -1.0 * double(shearing_ks) * (glm::length(dx) - shearing_dist) * glm::
    ↪ normalize(dx);
196       glm::dvec3 d = -1.0 * double(shearing_kd) * dv;
197
```

```
198         src.forces[shearing_springs[i].x] += f + d;
199         src.forces[shearing_springs[i].y] -= f + d;
200       }
201     }
202
203     if (enable_stretch) {
204       for (int i = 0; i < structural_springs.size(); i++)
205       {
206         glm::dvec3 x1 = src.vertices[structural_springs[i].x];
207         glm::dvec3 x2 = src.vertices[structural_springs[i].y];
208
209         glm::dvec3 v1 = (x1 - src.previous_vertices[structural_springs[i].x]) / dt;
210         glm::dvec3 v2 = (x2 - src.previous_vertices[structural_springs[i].y]) / dt;
211
212         glm::dvec3 dx = x1 - x2;
213         glm::dvec3 dv = v1 - v2;
214
215         glm::dvec3 f = -1.0 * double(structural_ks) * (glm::length(dx) - structural_dist) *
      ↪ glm::normalize(dx);
216         glm::dvec3 d = -1.0 * double(structural_kd) * dv;
217
218         src.forces[structural_springs[i].x] += f + d;
219         src.forces[structural_springs[i].y] -= f + d;
220       }
221     }
222
223     if (enable_bend) {
224       for (int i = 0; i < bending_springs.size(); i++)
225       {
226         glm::dvec3 x1 = src.vertices[bending_springs[i].x];
227         glm::dvec3 x2 = src.vertices[bending_springs[i].y];
228
229         glm::dvec3 v1 = (x1 - src.previous_vertices[bending_springs[i].x]) / dt;
230         glm::dvec3 v2 = (x2 - src.previous_vertices[bending_springs[i].y]) / dt;
231
232         glm::dvec3 dx = x1 - x2;
233         glm::dvec3 dv = v1 - v2;
234
235         glm::dvec3 f = -1.0 * double(bending_ks) * (glm::length(dx) - bending_dist) * glm::
      ↪ normalize(dx);
236         glm::dvec3 d = -1.0 * double(bending_kd) * dv;
237
238         src.forces[bending_springs[i].x] += f + d;
239         src.forces[bending_springs[i].y] -= f + d;
240       }
241     }
242 }
```

D.1.2. Mecánicas continuas

```
1 class ContinuumModel : public Model
```

```
{
public:
  float stretch_ks = 500;
  float stretch_kd = 0.2;

  float shear_ks = 50;
  float shear_kd = 0.2;

  float bend_ks = 0.00001;
  float bend_kd = 0.2;

  bool enable_stretch = true;
  bool enable_shear = true;
  bool enable_bend = true;

  ContinuumModel(ClothObject& src);

  ~ContinuumModel();

  void apply_effect(double dt);

private:
  std::vector<double> original_areas;

  std::vector<double> du1;
  std::vector<double> du2;
  std::vector<double> dv1;
  std::vector<double> dv2;

  std::vector<double> duv_det;

  std::vector<double> dWu_dx0;
  std::vector<double> dWu_dx1;
  std::vector<double> dWu_dx2;

  std::vector<double> dWv_dx0;
  std::vector<double> dWv_dx1;
  std::vector<double> dWv_dx2;
};
```

```
ContinuumModel::ContinuumModel(ClothObject& src) : Model(src)
{
  original_areas = std::vector<double>(src.faces.size());

  du1 = std::vector<double>(src.faces.size());
  du2 = std::vector<double>(src.faces.size());
  dv1 = std::vector<double>(src.faces.size());
  dv2 = std::vector<double>(src.faces.size());

  duv_det = std::vector<double>(src.faces.size());

```

```
  dWu_dx0 = std::vector<double>(src.faces.size());
  dWu_dx1 = std::vector<double>(src.faces.size());
  dWu_dx2 = std::vector<double>(src.faces.size());

  dWv_dx0 = std::vector<double>(src.faces.size());
  dWv_dx1 = std::vector<double>(src.faces.size());
  dWv_dx2 = std::vector<double>(src.faces.size());

  for (int i = 0; i < src.faces.size(); ++i)
  {
    glm::dvec3 x0 = src.vertices[src.origins[src.neighbors[src.faces[i]].x]];
    glm::dvec3 x1 = src.vertices[src.origins[src.faces[i]]];
    glm::dvec3 x2 = src.vertices[src.origins[src.neighbors[src.faces[i]].y]];

    original_areas[i] = std::pow(glm::length(glm::cross(x1 - x0, x2 - x0)) / 2, 3.0 / 4.0);

    du1[i] = x1.x - x0.x;
    du2[i] = x2.x - x0.x;
    dv1[i] = x1.z - x0.z;
    dv2[i] = x2.z - x0.z;

    duv_det[i] = du1[i] * dv2[i] - du2[i] * dv1[i];

    dWu_dx0[i] = (dv1[i] - dv2[i]) / duv_det[i];
    dWu_dx1[i] = dv2[i] / duv_det[i];
    dWu_dx2[i] = -1 * dv1[i] / duv_det[i];

    dWv_dx0[i] = (du2[i] - du1[i]) / duv_det[i];
    dWv_dx1[i] = -1 * du2[i] / duv_det[i];
    dWv_dx2[i] = du1[i] / duv_det[i];
  }
}

ContinuumModel::~ContinuumModel()
{
}

void ContinuumModel::apply_effect(double dt)
{
  for (int i = 0; i < src.vertices.size(); ++i)
  {
    src.forces[i] = glm::dvec3(0.0, -9.81 * src.masses[i], 0.0);
  }

  if (enable_bend)
  {
    for (int i = 0; i < src.edges.size(); ++i)
    {
      unsigned int edge_1 = src.edges[i];
      unsigned int edge_2 = src.twins[src.edges[i]];

      if (edge_2 == NULL)
```

```
64          {
65            continue;
66          }
67
68          glm::dvec3 x0 = src.vertices[src.origins[src.neighbors[edge_1].x]];
69          glm::dvec3 x1 = src.vertices[src.origins[edge_2]];
70          glm::dvec3 x2 = src.vertices[src.origins[edge_1]];
71          glm::dvec3 x3 = src.vertices[src.origins[src.neighbors[edge_2].x]];
72
73          glm::dvec3 nA = src.normals[src.incident_faces[edge_1]];
74          glm::dvec3 nB = src.normals[src.incident_faces[edge_2]];
75          glm::dvec3 e = x1 - x2;
76
77          glm::dvec3 nA_normalized = glm::normalize(nA);
78          glm::dvec3 nB_normalized = glm::normalize(nB);
79          glm::dvec3 e_normalized = glm::normalize(e);
80
81          double nA_norm = glm::length(nA);
82          double nB_norm = glm::length(nB);
83          double e_norm = glm::length(e);
84
85          double cosTheta = glm::dot(nA_normalized, nB_normalized);
86          double sinTheta = glm::dot(glm::cross(nA_normalized, nB_normalized),
    ↪ e_normalized);
87
88          // Bend force
89
90          double C = std::atan2(sinTheta, cosTheta);
91
92          std::vector<glm::dvec3> qA = { x2 - x1, x0 - x2, x1 - x0, glm::dvec3(0.0, 0.0, 0.0) };
93          std::vector<glm::dvec3> qB = { glm::dvec3(0.0, 0.0, 0.0), x2 - x3, x3 - x1, x1 - x2 };
94          std::vector<int> qe = { 0, 1, -1, 0 };
95
96          std::vector<glm::dvec3> dnA_norm_dx0 = {
97            glm::dvec3(0, -1 * qA[0].z, qA[0].y) / nA_norm,
98            glm::dvec3(qA[0].z, 0, -1 * qA[0].x) / nA_norm,
99            glm::dvec3(-1 * qA[0].y, qA[0].x, 0) / nA_norm,
100         };
101         std::vector<glm::dvec3> dnA_norm_dx1 = {
102           glm::dvec3(0, -1 * qA[1].z, qA[1].y) / nA_norm,
103           glm::dvec3(qA[1].z, 0, -1 * qA[1].x) / nA_norm,
104           glm::dvec3(-1 * qA[1].y, qA[1].x, 0) / nA_norm,
105         };
106         std::vector<glm::dvec3> dnA_norm_dx2 = {
107           glm::dvec3(0, -1 * qA[2].z, qA[2].y) / nA_norm,
108           glm::dvec3(qA[2].z, 0, -1 * qA[2].x) / nA_norm,
109           glm::dvec3(-1 * qA[2].y, qA[2].x, 0) / nA_norm,
110         };
111         std::vector<glm::dvec3> dnA_norm_dx3 = {
112           glm::dvec3(0, -1 * qA[3].z, qA[3].y) / nA_norm,
113             glm::dvec3(qA[3].z, 0, -1 * qA[3].x) / nA_norm,
114             glm::dvec3(-1 * qA[3].y, qA[3].x, 0) / nA_norm,
```

```
115         };
116
117         std::vector<glm::dvec3> dnB_norm_dx0 = {
118           glm::dvec3(0, -1 * qB[0].z, qB[0].y) / nB_norm,
119             glm::dvec3(qB[0].z, 0, -1 * qB[0].x) / nB_norm,
120             glm::dvec3(-1 * qB[0].y, qB[0].x, 0) / nB_norm,
121         };
122         std::vector<glm::dvec3> dnB_norm_dx1 = {
123           glm::dvec3(0, -1 * qB[1].z, qB[1].y) / nB_norm,
124             glm::dvec3(qB[1].z, 0, -1 * qB[1].x) / nB_norm,
125             glm::dvec3(-1 * qB[1].y, qB[1].x, 0) / nB_norm,
126         };
127         std::vector<glm::dvec3> dnB_norm_dx2 = {
128           glm::dvec3(0, -1 * qB[2].z, qB[2].y) / nB_norm,
129             glm::dvec3(qB[2].z, 0, -1 * qB[2].x) / nB_norm,
130             glm::dvec3(-1 * qB[2].y, qB[2].x, 0) / nB_norm,
131         };
132         std::vector<glm::dvec3> dnB_norm_dx3 = {
133           glm::dvec3(0, -1 * qB[3].z, qB[3].y) / nB_norm,
134             glm::dvec3(qB[3].z, 0, -1 * qB[3].x) / nB_norm,
135             glm::dvec3(-1 * qB[3].y, qB[3].x, 0) / nB_norm,
136         };
137
138         std::vector<glm::dvec3> de_norm_dx0 = {
139           glm::dvec3(1 * qe[0], 0, 0) / e_norm,
140             glm::dvec3(0, 1 * qe[0], 0) / e_norm,
141             glm::dvec3(0, 0, 1 * qe[0]) / e_norm,
142         };
143         std::vector<glm::dvec3> de_norm_dx1 = {
144           glm::dvec3(1 * qe[1], 0, 0) / e_norm,
145             glm::dvec3(0, 1 * qe[1], 0) / e_norm,
146             glm::dvec3(0, 0, 1 * qe[1]) / e_norm,
147         };
148         std::vector<glm::dvec3> de_norm_dx2 = {
149           glm::dvec3(1 * qe[2], 0, 0) / e_norm,
150             glm::dvec3(0, 1 * qe[2], 0) / e_norm,
151             glm::dvec3(0, 0, 1 * qe[2]) / e_norm,
152         };
153         std::vector<glm::dvec3> de_norm_dx3 = {
154           glm::dvec3(1 * qe[3], 0, 0) / e_norm,
155             glm::dvec3(0, 1 * qe[3], 0) / e_norm,
156             glm::dvec3(0, 0, 1 * qe[3]) / e_norm,
157         };
158
159         glm::dvec3 dcosTheta_dx0 = glm::dvec3(
160           glm::dot(dnA_norm_dx0[0], nB_normalized) + glm::dot(dnB_norm_dx0[0],
    ↪ nA_normalized),
161             glm::dot(dnA_norm_dx0[1], nB_normalized) + glm::dot(dnB_norm_dx0[1],
    ↪ nA_normalized),
162             glm::dot(dnA_norm_dx0[2], nB_normalized) + glm::dot(dnB_norm_dx0[2],
    ↪ nA_normalized)
163         );
```

```
164         glm::dvec3 dcosTheta_dx1 = glm::dvec3(
165           glm::dot(dnA_norm_dx1[0], nB_normalized) + glm::dot(dnB_norm_dx1[0],
    ↪ nA_normalized),
166           glm::dot(dnA_norm_dx1[1], nB_normalized) + glm::dot(dnB_norm_dx1[1],
    ↪ nA_normalized),
167           glm::dot(dnA_norm_dx1[2], nB_normalized) + glm::dot(dnB_norm_dx1[2],
    ↪ nA_normalized)
168         );
169         glm::dvec3 dcosTheta_dx2 = glm::dvec3(
170           glm::dot(dnA_norm_dx2[0], nB_normalized) + glm::dot(dnB_norm_dx2[0],
    ↪ nA_normalized),
171           glm::dot(dnA_norm_dx2[1], nB_normalized) + glm::dot(dnB_norm_dx2[1],
    ↪ nA_normalized),
172           glm::dot(dnA_norm_dx2[2], nB_normalized) + glm::dot(dnB_norm_dx2[2],
    ↪ nA_normalized)
173         );
174         glm::dvec3 dcosTheta_dx3 = glm::dvec3(
175           glm::dot(dnA_norm_dx3[0], nB_normalized) + glm::dot(dnB_norm_dx3[0],
    ↪ nA_normalized),
176           glm::dot(dnA_norm_dx3[1], nB_normalized) + glm::dot(dnB_norm_dx3[1],
    ↪ nA_normalized),
177           glm::dot(dnA_norm_dx3[2], nB_normalized) + glm::dot(dnB_norm_dx3[2],
    ↪ nA_normalized)
178         );
179
180         glm::dvec3 dSinTheta_dx0 = glm::dvec3(
181           glm::dot(glm::cross(dnA_norm_dx0[0], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx0[0]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx0[0]),
182           glm::dot(glm::cross(dnA_norm_dx0[1], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx0[1]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx0[1]),
183           glm::dot(glm::cross(dnA_norm_dx0[2], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx0[2]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx0[2])
184         );
185         glm::dvec3 dSinTheta_dx1 = glm::dvec3(
186           glm::dot(glm::cross(dnA_norm_dx1[0], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx1[0]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx1[0]),
187           glm::dot(glm::cross(dnA_norm_dx1[1], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx1[1]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx1[1]),
188           glm::dot(glm::cross(dnA_norm_dx1[2], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx1[2]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx1[2])
189         );
190         glm::dvec3 dSinTheta_dx2 = glm::dvec3(
191           glm::dot(glm::cross(dnA_norm_dx2[0], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx2[0]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx2[0]),
192           glm::dot(glm::cross(dnA_norm_dx2[1], nB_normalized) + glm::cross(
```

```
    ↪ nA_normalized, dnB_norm_dx2[1]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx2[1]),
193         glm::dot(glm::cross(dnA_norm_dx2[2], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx2[2]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx2[2])
194       );
195       glm::dvec3 dSinTheta_dx3 = glm::dvec3(
196         glm::dot(glm::cross(dnA_norm_dx3[0], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx3[0]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx3[0]),
197         glm::dot(glm::cross(dnA_norm_dx3[1], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx3[1]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx3[1]),
198         glm::dot(glm::cross(dnA_norm_dx3[2], nB_normalized) + glm::cross(
    ↪ nA_normalized, dnB_norm_dx3[2]), e) + glm::dot(glm::cross(nA_normalized,
    ↪ nB_normalized), de_norm_dx3[2])
199       );
200
201       glm::dvec3 dC_dx0 = dSinTheta_dx0 * cosTheta - dcosTheta_dx0 * sinTheta;
202       glm::dvec3 dC_dx1 = dSinTheta_dx1 * cosTheta - dcosTheta_dx1 * sinTheta;
203       glm::dvec3 dC_dx2 = dSinTheta_dx2 * cosTheta - dcosTheta_dx2 * sinTheta;
204       glm::dvec3 dC_dx3 = dSinTheta_dx3 * cosTheta - dcosTheta_dx3 * sinTheta;
205
206       src.forces[src.origins[src.neighbors[edge_1].x]] += -1 * double(bend_ks) * dC_dx0 * C
    ↪ ;
207       src.forces[src.origins[edge_2]] += -1 * double(bend_ks) * dC_dx1 * C;
208       src.forces[src.origins[edge_1]] += -1 * double(bend_ks) * dC_dx2 * C;
209       src.forces[src.origins[src.neighbors[edge_2].x]] += -1 * double(bend_ks) * dC_dx3 * C
    ↪ ;
210
211       glm::dvec3 previous_x0 = src.vertices[src.origins[src.neighbors[edge_1].x]];
212       glm::dvec3 previous_x1 = src.vertices[src.origins[edge_2]];
213       glm::dvec3 previous_x2 = src.vertices[src.origins[edge_1]];
214       glm::dvec3 previous_x3 = src.vertices[src.origins[src.neighbors[edge_2].x]];
215
216       glm::dvec3 vx0 = (x0 - previous_x0) / dt;
217       glm::dvec3 vx1 = (x1 - previous_x1) / dt;
218       glm::dvec3 vx2 = (x2 - previous_x2) / dt;
219       glm::dvec3 vx3 = (x3 - previous_x3) / dt;
220
221       double dC_dt = glm::dot(dC_dx0, vx0) + glm::dot(dC_dx1, vx1) + glm::dot(
    ↪ dC_dx2, vx2) + glm::dot(dC_dx3, vx3);
222
223       src.forces[src.origins[src.neighbors[edge_1].x]] += -1 * double(bend_kd) * dC_dx0 *
    ↪ dC_dt;
224       src.forces[src.origins[edge_2]] += -1 * double(bend_kd) * dC_dx1 * dC_dt;
225       src.forces[src.origins[edge_1]] += -1 * double(bend_kd) * dC_dx2 * dC_dt;
226       src.forces[src.origins[src.neighbors[edge_2].x]] += -1 * double(bend_kd) * dC_dx3 *
    ↪ dC_dt;
227     }
228   }
229
```

```
230   for (int i = 0; i < src.faces.size(); ++i)
231   {
232     glm::dvec3 x0 = src.vertices[src.origins[src.neighbors[src.faces[i]].x]];
233     glm::dvec3 x1 = src.vertices[src.origins[src.faces[i]]];
234     glm::dvec3 x2 = src.vertices[src.origins[src.neighbors[src.faces[i]].y]];
235
236     glm::dvec3 dx1 = x1 - x0;
237     glm::dvec3 dx2 = x2 - x0;
238
239     glm::dvec3 previous_x0 = src.previous_vertices[src.origins[src.neighbors[src.faces[i]].x]];
240     glm::dvec3 previous_x1 = src.previous_vertices[src.origins[src.faces[i]]];
241     glm::dvec3 previous_x2 = src.previous_vertices[src.origins[src.neighbors[src.faces[i]].y]];
242
243     glm::dvec3 vx0 = (x0 - previous_x0) / dt;
244     glm::dvec3 vx1 = (x1 - previous_x1) / dt;
245     glm::dvec3 vx2 = (x2 - previous_x2) / dt;
246
247     glm::dvec3 Wu = (dx1 * dv2[i] - dx2 * dv1[i]) / duv_det[i];
248     glm::dvec3 Wv = (dx2 * du1[i] - dx1 * du2[i]) / duv_det[i];
249
250     // Stretch forces
251
252     if (enable_shear)
253     {
254       double Wu_norm = glm::length(Wu);
255       double Wv_norm = glm::length(Wv);
256
257       glm::dvec3 Wu_normalized = glm::normalize(Wu);
258       glm::dvec3 Wv_normalized = glm::normalize(Wv);
259
260       glm::dvec3 dCu_dx0 = original_areas[i] * dWu_dx0[i] * Wu_normalized;
261       glm::dvec3 dCu_dx1 = original_areas[i] * dWu_dx1[i] * Wu_normalized;
262       glm::dvec3 dCu_dx2 = original_areas[i] * dWu_dx2[i] * Wu_normalized;
263
264       glm::dvec3 dCv_dx0 = original_areas[i] * dWv_dx0[i] * Wv_normalized;
265       glm::dvec3 dCv_dx1 = original_areas[i] * dWv_dx1[i] * Wv_normalized;
266       glm::dvec3 dCv_dx2 = original_areas[i] * dWv_dx2[i] * Wv_normalized;
267
268       double Cu = original_areas[i] * (Wu_norm - 1);
269       double Cv = original_areas[i] * (Wv_norm - 1);
270
271       src.forces[src.origins[src.neighbors[src.faces[i]].x]] += -1 * double(stretch_ks) * (
    ↪ dCu_dx0 * Cu + dCv_dx0 * Cv);
272       src.forces[src.origins[src.faces[i]]] += -1 * double(stretch_ks) * (dCu_dx1 * Cu +
    ↪ dCv_dx1 * Cv);
273       src.forces[src.origins[src.neighbors[src.faces[i]].y]] += -1 * double(stretch_ks) * (
    ↪ dCu_dx2 * Cu + dCv_dx2 * Cv);
274
275       double dCu_dt = glm::dot(dCu_dx0, vx0) + glm::dot(dCu_dx1, vx1) + glm::dot(
    ↪ dCu_dx2, vx2);
276       double dCv_dt = glm::dot(dCv_dx0, vx0) + glm::dot(dCv_dx1, vx1) + glm::dot(
    ↪ dCv_dx2, vx2);
```

```
277
278         src.forces[src.origins[src.neighbors[src.faces[i]].x]] += -1 * double(stretch_kd) * (
    ↪ dCu_dx0 * dCu_dt + dCv_dx0 * dCv_dt);
279         src.forces[src.origins[src.faces[i]]] += -1 * double(stretch_kd) * (dCu_dx1 * dCu_dt
    ↪ + dCv_dx1 * dCv_dt);
280         src.forces[src.origins[src.neighbors[src.faces[i]].y]] += -1 * double(stretch_kd) * (
    ↪ dCu_dx2 * dCu_dt + dCv_dx2 * dCv_dt);
281       }
282
283       // Shear forces
284
285       if (enable_shear)
286       {
287         double C = original_areas[i] * glm::dot(Wu, Wv);
288
289         glm::dvec3 dC_dx0 = Wu * dWv_dx0[i] * original_areas[i] + Wv * dWu_dx0[i] *
    ↪ original_areas[i];
290         glm::dvec3 dC_dx1 = Wu * dWv_dx1[i] * original_areas[i] + Wv * dWu_dx1[i] *
    ↪ original_areas[i];
291         glm::dvec3 dC_dx2 = Wu * dWv_dx2[i] * original_areas[i] + Wv * dWu_dx2[i] *
    ↪ original_areas[i];
292
293         src.forces[src.origins[src.neighbors[src.faces[i]].x]] += -1 * double(shear_ks) * dC_dx0
    ↪ * C;
294         src.forces[src.origins[src.faces[i]]] += -1 * double(shear_ks) * dC_dx1 * C;
295         src.forces[src.origins[src.neighbors[src.faces[i]].y]] += -1 * double(shear_ks) * dC_dx2
    ↪ * C;
296
297         double dC_dt = glm::dot(dC_dx0, vx0) + glm::dot(dC_dx1, vx1) + glm::dot(
    ↪ dC_dx2, vx2);
298
299         src.forces[src.origins[src.neighbors[src.faces[i]].x]] += -1 * double(shear_kd) * dC_dx0
    ↪ * dC_dt;
300         src.forces[src.origins[src.faces[i]]] += -1 * double(shear_kd) * dC_dx1 * dC_dt;
301         src.forces[src.origins[src.neighbors[src.faces[i]].y]] += -1 * double(shear_kd) * dC_dx2
    ↪ * dC_dt;
302       }
303     }
304 }
```

D.2. Evolución en el tiempo

```
1 class Integration
2 {
3 public:
4   Integration(ClothObject& src);
5
6   virtual ~Integration() = 0;
7
8   virtual void integrate(double dt) = 0;
```

```
protected:
  ClothObject& src;
};
```

```
Integration::Integration(ClothObject& src) : src(src) {}

Integration::~Integration()
{
}
```

D.2.1. Integración de Verlet

```
class VerletIntegration : public Integration
{
public:
  float damping = 0.01;

  VerletIntegration(ClothObject& src);

  ~VerletIntegration();

  void integrate(double dt);
};
```

```
VerletIntegration::VerletIntegration(ClothObject& src) : Integration(src) {}

VerletIntegration::~VerletIntegration()
{
}

void VerletIntegration::integrate(double dt)
{
  int size = sqrt(src.vertices.size());

  for (unsigned int i = 0; i < src.vertices.size(); ++i)
  {
    if (src.fixed[i]) {
      continue;
    }

    glm::dvec3 buffer = glm::dvec3(src.vertices[i]);
    src.vertices[i] = src.vertices[i] + (src.vertices[i] - src.previous_vertices[i]) * (1.0 -
    ↪ damping) + dt * dt * src.forces[i] / src.masses[i];
    src.previous_vertices[i] = glm::dvec3(buffer);
  }
}
```

D.3. Manejo de colisiones

```
class Collision
{
public:
   Collision(StaticObject& src);

   virtual ~Collision();

   virtual void collide(ClothObject& cloth) = 0;

protected:
   StaticObject& src;
};
```

```
Collision::Collision(StaticObject& src) : src(src) {}

Collision::~Collision()
{
}
```

D.3.1. Octree

```
class TriangleOctreeNode
{
public:
   TriangleOctreeNode(int height, TriangleOctreeNode* parent, glm::dvec3 box_center, glm::
     ↪ dvec3 box_half_size);

   ~TriangleOctreeNode();

   void insert_triangle(unsigned int triangle_idx, glm::dvec3 triangle_p1, glm::dvec3
     ↪ triangle_p2, glm::dvec3 triangle_p3);
   std::set<unsigned int> search_by_segment(glm::dvec3 segment_start, glm::dvec3
     ↪ segment_end);

private:
   int height;

   TriangleOctreeNode* parent;
   std::vector<TriangleOctreeNode*> childs;

   glm::dvec3 box_center;
   glm::dvec3 box_half_size;

   std::set<unsigned int> indices;
};

class OctreeCollision : public Collision
```

```
{
public:
  OctreeCollision(StaticObject& src);

  ~OctreeCollision();

  void collide(ClothObject& cloth);

  std::tuple<bool, glm::dvec3, glm::dvec3> collide_segment(glm::dvec3 segment_start, glm::
    ↪ dvec3 segment_end);

private:
  TriangleOctreeNode* root;
};
```

```
TriangleOctreeNode::TriangleOctreeNode(int height, TriangleOctreeNode* parent, glm::dvec3
    ↪  box_center, glm::dvec3 box_half_size)
{
  this->height = height;

  this->parent = parent;
  this->box_center = box_center;
  this->box_half_size = box_half_size;

  this->childs = std::vector<TriangleOctreeNode*>(8, NULL);

  if (this->height > 1)
  {
    glm::dvec3 child_box_half_size = box_half_size / 2.0;

    this->childs[0] = new TriangleOctreeNode(this->height - 1, this, box_center + glm::
    ↪ dvec3(child_box_half_size.x, child_box_half_size.y, child_box_half_size.z),
    ↪ child_box_half_size);
    this->childs[1] = new TriangleOctreeNode(this->height - 1, this, box_center + glm::
    ↪ dvec3(-1.0f * child_box_half_size.x, child_box_half_size.y, child_box_half_size.z),
    ↪ child_box_half_size);
    this->childs[2] = new TriangleOctreeNode(this->height - 1, this, box_center + glm::
    ↪ dvec3(child_box_half_size.x, -1.0f * child_box_half_size.y, child_box_half_size.z),
    ↪ child_box_half_size);
    this->childs[3] = new TriangleOctreeNode(this->height - 1, this, box_center + glm::
    ↪ dvec3(child_box_half_size.x, child_box_half_size.y, -1.0f * child_box_half_size.z),
    ↪ child_box_half_size);
    this->childs[4] = new TriangleOctreeNode(this->height - 1, this, box_center + glm::
    ↪ dvec3(-1.0f * child_box_half_size.x, -1.0f * child_box_half_size.y,
    ↪ child_box_half_size.z), child_box_half_size);
    this->childs[5] = new TriangleOctreeNode(this->height - 1, this, box_center + glm::
    ↪ dvec3(-1.0f * child_box_half_size.x, child_box_half_size.y, -1.0f *
    ↪ child_box_half_size.z), child_box_half_size);
    this->childs[6] = new TriangleOctreeNode(this->height - 1, this, box_center + glm::
    ↪ dvec3(child_box_half_size.x, -1.0f * child_box_half_size.y, -1.0f *
    ↪ child_box_half_size.z), child_box_half_size);
```

```
22      this->childs[7] = new TriangleOctreeNode(this->height - 1, this, box_center + glm::
        ↪ dvec3(-1.0f * child_box_half_size.x, -1.0f * child_box_half_size.y, -1.0f *
        ↪ child_box_half_size.z), child_box_half_size);
23    }
24  }
25
26  TriangleOctreeNode::~TriangleOctreeNode()
27  {
28    for (int i = 0; i < childs.size(); ++i) {
29        delete childs[i];
30    }
31  }
32
33  void TriangleOctreeNode::insert_triangle(unsigned int triangle_idx, glm::dvec3 triangle_p1,
        ↪ glm::dvec3 triangle_p2, glm::dvec3 triangle_p3)
34  {
35    if (!triangle_box_overlap(triangle_p1, triangle_p2, triangle_p3, this->box_center, this->
        ↪ box_half_size))
36    {
37      return;
38    }
39
40    if (this->height == 1)
41    {
42      this->indices.insert(triangle_idx);
43    }
44    else
45    {
46      for (int i = 0; i < this->childs.size(); ++i)
47      {
48        this->childs[i]->insert_triangle(triangle_idx, triangle_p1, triangle_p2, triangle_p3);
49      }
50    }
51  }
52
53  std::set<unsigned int> TriangleOctreeNode::search_by_segment(glm::dvec3 segment_start,
        ↪ glm::dvec3 segment_end)
54  {
55    if (!segment_box_overlap(segment_start, segment_end, this->box_center, this->
        ↪ box_half_size))
56    {
57      return std::set<unsigned int>();
58    }
59
60    if (this->height == 1)
61    {
62      return this->indices;
63    }
64    else
65    {
66      std::set<unsigned int> result = std::set<unsigned int>();
67
```

```
68      for (int i = 0; i < this->childs.size(); ++i)
69      {
70        result.merge(this->childs[i]->search_by_segment(segment_start, segment_end));
71      }
72
73      return result;
74    }
75  }
76
77  OctreeCollision::OctreeCollision(StaticObject& src) : Collision(src)
78  {
79    glm::dvec3 min = glm::dvec3(5.0, 5.0, 5.0);
80    glm::dvec3 max = glm::dvec3(-5.0, -5.0, -5.0);
81
82    for (int i = 0; i < src.vertices.size(); ++i)
83    {
84      min = glm::min(min, src.vertices[i]);
85      max = glm::max(max, src.vertices[i]);
86    }
87
88    glm::dvec3 box_center = (min + max) / 2.0;
89    glm::dvec3 box_extend = glm::abs(max - box_center) + glm::dvec3(0.000001, 0.000001,
      ↪ 0.000001);
90
91    this->root = new TriangleOctreeNode(8, NULL, box_center, box_extend);
92
93    for (int i = 0; i < src.faces.size(); ++i)
94    {
95      this->root->insert_triangle(
96        i,
97        src.vertices[src.origins[src.neighbors[src.faces[i]].x]],
98        src.vertices[src.origins[src.faces[i]]],
99        src.vertices[src.origins[src.neighbors[src.faces[i]].y]]
100     );
101   }
102 }
103
104 OctreeCollision::~OctreeCollision()
105 {
106   delete root;
107 }
108
109 void OctreeCollision::collide(ClothObject& cloth)
110 {
111   for (int i = 0; i < cloth.vertices.size(); ++i)
112   {
113     auto [collide, intersection, collide_normal] = collide_segment(cloth.previous_vertices[i],
      ↪ cloth.vertices[i]);
114
115     if (collide)
116     {
117       cloth.vertices[i] = intersection + glm::normalize(cloth.previous_vertices[i] -
```

```
    ↪ intersection) * 0.0001;
118       cloth.vertices[i] = intersection + collide_normal * 0.0001;
119     }
120   }
121 }
122
123 std::tuple<bool, glm::dvec3, glm::dvec3> OctreeCollision::collide_segment(glm::dvec3
    ↪ segment_start, glm::dvec3 segment_end)
124 {
125   std::set<unsigned int> collision_candidates = this->root->search_by_segment(
    ↪ segment_start, segment_end);
126
127   unsigned int closest_idx = 0;
128   glm::dvec3 closest_intersection = segment_end;
129   double closest_distance2 = glm::distance2(segment_start, segment_end);
130
131   for (auto itr = collision_candidates.begin(); itr != collision_candidates.end(); itr++)
132   {
133     unsigned int idx = *itr;
134
135     auto [status, intersection, distance2] = segment_triangle_overlap(
136       segment_start,
137       segment_end,
138       src.vertices[src.origins[src.neighbors[src.faces[idx]].x]],
139       src.vertices[src.origins[src.faces[idx]]],
140       src.vertices[src.origins[src.neighbors[src.faces[idx]].y]]
141     );
142
143     if (status && distance2 < closest_distance2)
144     {
145       closest_idx = idx;
146       closest_intersection = intersection;
147       closest_distance2 = distance2;
148     }
149   }
150
151   if (closest_distance2 != glm::distance2(segment_start, segment_end))
152   {
153     return { true, closest_intersection, src.normals[closest_idx] };
154   }
155   else
156   {
157     return { false, glm::dvec3(), glm::dvec3() };
158   }
159 }
```