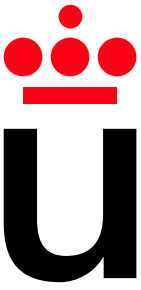

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Diseño y Desarrollo de Videojuegos

Curso 2022-2023

Trabajo Fin de Grado

FRAMEWORK PARA EDICIÓN INVERSA DE ANIMACIONES BASADO EN SIMULACIÓN DIFERENCIABLE

Autor: Daniel Martí Casanova
Tutor: Miguel Ángel Otaduy Tristán

Agradecimientos

Me gustaría dedicar este trabajo a toda la gente que me ha ayudado en el desarrollo de este trabajo de fin de grado a lo largo de todo el año.

Gracias a Mickeal por ofrecerme una base sobre la que empezar a trabajar cuando no sabia ni por donde comenzar.

Gracias a los chicos del laboratorio y, en especial, a Pablo por toda la ayuda que me brindó con el módulo de Python.

Gracias a Igor por su ayuda en primero, sin él dudo que hubiese llegado hasta aquí.

Y por último, quiero agradecer a Miguel Ángel; no podría haber pedido un mejor tutor. Siempre dispuesto a hacerme un hueco para explicarme las cosas todas las veces que hicieran falta, o para responderme un correo cargado de dudas y preguntas.

Resumen

La edición manual de parámetros de simulaciones es un proceso lento y tedioso en el que una persona ejecuta múltiples veces una simulación, probando distintos parámetros, hasta obtener el resultado deseado.

En este trabajo de fin de grado se propone un método para automatizar este proceso utilizando aprendizaje automático. El objetivo es que el usuario pueda proporcionar una escena a simular e información sobre el comportamiento deseado de dicha simulación (vídeos, captura de movimiento... etc.) y que los parámetros se ajusten de forma automática en base a dicha información.

Existen proyectos similares, pero en este caso se ha utilizado la simulación diferenciable para acelerar el proceso de optimización mediante el método de *backpropagation* para el cálculo del gradiente.

Se ha desarrollado una herramienta a modo de prueba de concepto para demostrar la viabilidad del método descrito y se han realizado múltiples experimentos con el fin de evaluar tanto la precisión de las estimaciones como el tiempo de ejecución. Por último, se proponen múltiples mejoras posibles de cara a futuros desarrollos.

Palabras clave:

- Aprendizaje automático.
- Simulación diferenciable.
- Optimización.
- Edición inversa de animaciones.
- Integración implícita.

Índice de contenidos

Índice de tablas

Índice de figuras

1 Introducción

Los simuladores de físicas de telas actuales permiten simular el comportamiento de telas con un alto grado de fidelidad. Un motor de físicas es un software capaz de proporcionar una simulación aproximada de ciertos sistemas físicos, en este caso sólidos deformables. Existen múltiples métodos para simular físicas, todos con sus pros y contras, pero el principal objetivo de todos ellos es proporcionar una simulación realista y estable, intentando ser lo óptimos y rápidos posibles. En el caso del proyecto actual, al estar trabajando en el contexto de los videojuegos, se pretende que las simulaciones se ejecuten en tiempo real y no afecten negativamente a la experiencia del usuario.

1.1. Diseño de simulaciones en motores de videojuegos frente a herramientas de optimización y *machine learning*

Para aprovechar al máximo las posibilidades de la animación basada en simulación de físicas y obtener un comportamiento satisfactorio, se deben ajustar un gran número de parámetros de la escena a simular de forma cuidadosa. Durante este proceso de ajuste es difícil predecir qué parámetros darán un comportamiento más cercano al deseado.

Actualmente, lo más habitual es ajustar las simulaciones de forma manual. Un artista modifica cada uno de los parámetros y ejecuta la simulación para ver

el resultado de los cambios realizados, repitiendo este proceso tantas veces como sean necesarias hasta conseguir el comportamiento deseado.

Las herramientas de *machine learning* permiten agilizar en gran medida este proceso, automatizando el proceso de selección de los parámetros y ahorrando una gran cantidad de tiempo al artista. Basta con proporcionar una muestra de cómo se debe comportar la simulación para que se estimen los parámetros necesarios para obtener un resultado lo más similar posible al deseado. De este modo, se edita la simulación de forma inversa, es decir, se obtienen los parámetros de la escena a partir del resultado deseado.

1.2. La simulación diferenciable como herramienta para acelerar el proceso de estimación

Para resolver este problema mediante *machine learning*, se puede enfocar como un problema de optimización en el que se buscan unos parámetros para la simulación que proporcionen un resultado óptimo.

Existen multitud de técnicas de optimización. Por un lado, hay métodos numéricos que ejecutan la misma simulación una gran cantidad de veces modificando ligeramente los parámetros para observar cómo cambia el resultado final y, en función de eso, modifican los parámetros en una dirección u otra. Estos métodos son ampliamente utilizados debido a que son efectivos, pero no tan eficientes. Además, la complejidad aumenta exponencialmente con el número de parámetros, así que en el caso de una tela con cientos de parámetros no es una opción viable.

Por otro lado, existen métodos analíticos, más difíciles de calcular e implementar, pero que ofrecen una complejidad independiente al número de parámetros y requieren realizar menos simulaciones en general. Para aplicar estos métodos es necesario extraer más información de la simulación, en forma de derivadas de la función de coste respecto a cada uno de los parámetros. Para obtener el conjunto de estas derivadas, comúnmente conocido como gradiente, es necesario implementar un motor de simulación de físicas diferenciable propio para poder hacer los cálculos necesarios.

La simulación de físicas diferenciable es una familia de técnicas punteras muy potentes que aplica métodos basados en gradientes para aprender y controlar sistemas físicos [1].

La idea de la 'simulacióon diferenciable' es muy similar al de '*differentiable rendering*', o 'renderizado diferenciable', una metodología con gran éxito en visión artificial, ya que permite hacer diferenciable el proceso de síntesis de imagen.

Los algoritmos de renderizado diferenciable existentes [2] se centran en la computación de derivadas de imágenes que muestran comportamientos complejos de la luz (p. ej. sombras suaves, interreflexión y cáusticas) con respecto a parámetros arbitrarios de la escena, como la posición de la cámara, la geometría del objeto o las propiedades de los materiales. El renderizado diferenciable es un ingrediente clave para resolver muchos problemas del renderizado inverso, es decir, la búsqueda de configuraciones de escena que optimicen las funciones objetivo especificadas por el usuario, usando métodos basados en gradientes. Este proyecto plantea una idea similar, haciendo diferenciable la síntesis de movimiento.

2 Objetivos

En este trabajo se ha desarrollado una prueba de concepto para demostrar la viabilidad del uso de la técnica de *backpropagation* para la estimación de forma automática y eficiente de los parámetros de una simulación dentro del contexto del desarrollo de videojuegos.

2.1. Descripción del problema

El objetivo es que un artista pueda configurar una escena con múltiples objetos deformables, en este caso telas, y no tenga que realizar el tedioso trabajo de modificar manualmente los parámetros de cada tela para lograr el efecto deseado. En su lugar, el artista deberá proporcionar datos diseñados a mano, procedentes de captura de movimiento o incluso vídeo del material que se desea imitar y, a partir de esta información, la herramienta calculará los parámetros óptimos para que las características de la tela simulada sean lo más similares posibles.

Para ello, se han cumplido múltiples subobjetivos, a saber:

- Se ha desarrollado un motor de simulación de físicas diferenciable y eficiente que permite simular telas interactuando con distintas fuerzas como la gravedad o la fricción con el viento. El motor se ha desarrollado en C++ por su eficiencia superior y ha sido exportado como una librería de enlace dinámico (DLL) para poder ser utilizado en distintos motores de videojuegos.

- Se ha utilizado *Unity* para mostrar gráficamente el resultado de estas simulaciones.

- Se ha implementado una interfaz de usuario en *Unity* con la que se pueda configurar una escena con distintos objetos y ajustar los parámetros de cada uno de ellos, así como seleccionar cuales de estos parámetros deben ser estimados.

- Se ha creado un *script* de *Python* que, mediante *machine learning* y el algoritmo de *backpropagation*, estima los parámetros seleccionados y devuelve esa información al motor. Para resolver la optimización de parámetros se ha utilizado la herramienta *SciPy*, que permite abstraerse de tener que implementar el algoritmo de optimización. De este modo tan solo ha sido necesario leer los datos de la escena, construir una función de coste y calcular el gradiente de dicha función.

2.2. Alternativas

La simulación de telas tiene una historia relativamente larga en el mundo de los gráficos por ordenador. Durante los últimos cincuenta años se ha investigado mucho al respecto y se han generado múltiples modelos de simulación. No obstante, existen muy pocos proyectos centrados en la automatización de la selección de parámetros de dichas simulaciones.

Desde la propia universidad se han desarrollado varios proyectos que combinan la simulación de físicas de telas con métodos de machine learning, como *"Data-Driven Estimation of Cloth Simulation Models"*[3] o *"Design and Fabrication of Flexible Rod Meshes"*[4], ambos obteniendo resultados notables.

Existe además un proyecto de la universidad *Carnegie Mellon* titulado *"Estimating Cloth Simulation Parameters from Video"*[5] en el que desarrolló un *framework* para la optimización de parámetros a partir de fragmentos de vídeo de telas de muestra. Los resultados también son alentadores, aunque los propios autores instan a que se investigue más en esta dirección.

2.3. Metodología

Se ha seguido una metodología de trabajo tradicional. Al principio del proyecto se definieron los requisitos del *framework*, así como la estructura de éste y las herramientas que se iban a utilizar para su desarrollo de acuerdo a los objetivos propuestos. El trabajo se ha divido en múltiples fases, cada una de ella

imprescindible para poder empezar a desarrollar la siguiente. Las fases han sido las siguientes:

- Configuración de la escena en *Unity* y serialización de la misma en un *JSON*.
- Conexión de *Unity* con una librería C++.
- Implementación del motor de físicas en la librería C++.
- Cálculo del gradiente en el motor.
- Conexión del motor con *Python* mediante *PyBind11*.
- Optimización de parámetros con *ScyPy* desde *Python*, utilizando el gradiente calculado.

3

Descripción matemática del problema

3.1. Simulación de telas

La simulación de telas consiste en simular el comportamiento de las telas dentro de un programa informático, normalmente dentro del contento de los gráficos por ordenador. Existen tres tipos de métodos para simular telas: modelo discreto acorde a la estructura del material (a escala de hilo), modelo continuo de superficie deformable (discretizado, p. ej. por elementos finitos), modelo de elementos discretos (p.ej. masa muelle).

En este caso se ha utilizado el método de masa muelle [6], que consiste en discretizar la tela en una colección de nodos unidos entre ellos por muelles. Por cada vértice de la malla existe un nodo que se simula como una partícula con una masa proporcional al área de la tela que representa.

Estos nodos poseen una posición, una velocidad y una masa. Se utiliza la mecánica newtoniana para modelar el comportamiento de cada partícula mediante un motor físico basado en la ley básica del movimiento (Segunda ley de Newton):

$$F = m * a \tag{3.1}$$

Se les aplican distintas fuerzas que hacen que el conjunto se comporte como una tela. Fuerzas como la gravedad o el rozamiento con el viento deforman la tela, mientras que la tensión de los muelles evita que la tela se estire o se doble

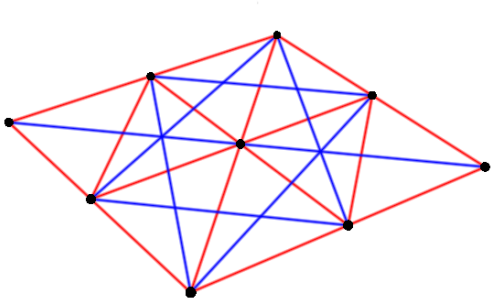

Figura 3.1: Ejemplo de una malla con 9 nodos representados con puntos unidos entre ellos por muelles representados por lineas. Los muelles rojos son muelles de tracción, los cuales tienen una rigidez mayor y cuyo propósito es combatir las fuerzas que tiran de la tela intentando estirarla. Los muelles azules son los muelles de flexión, tienen una menor rigidez y su objetivo es impedir que la malla pueda plegarse de forma no realista sobre sus aristas.

más de la cuenta.

Una vez definido el modelo a simular, es necesario hablar de cómo se va a integrar la ecuación diferencial resultante, la cual devuelve la posición de los vértices en cada instante de tiempo. Para ello se ha utilizado el método numérico conocido como *backward* Euler.

3.2. *Backward* Euler o Euler implícito

El método de *backward* Euler o Euler implícito es uno de los métodos numéricos más básicos para la resolución de ecuaciones diferenciales ordinarias. Es similar al método de Euler estándar, pero se diferencia de este en que se trata de un método implícito.

Los métodos explícitos calculan el estado de un sistema en un momento posterior a partir del estado del sistema en el momento actual, mientras que los métodos implícitos encuentran una solución resolviendo una ecuación que involucra tanto el estado actual del sistema como el posterior. Por tanto, para un método explícito

$$x(t+h) = F(x(t))$$

, mientras que para un método implícito es necesario resolver la ecuación

$$F(x(t), x(t+h)) = 0$$

para encontrar $x(t+h)$.

Los métodos implícitos requieren un cálculo adicional (resolver la ecuación an-

terior) y pueden ser mucho más difíciles de implementar. Se usan porque muchos problemas que surgen en la práctica son rígidos, por lo que el uso de un método explícito requiere pasos de tiempo poco prácticos para mantener el error en el resultado acotado. Los métodos implícitos tienen una mayor estabilidad numérica, es decir, son menos susceptibles a la acumulación de errores de redondeo a lo largo de múltiples pasos. Esto es importante ya que en caso de no ser estable, se podría producir una gran desviación del resultado final con respecto a la solución exacta. Se dice que un algoritmo cuyo error de aproximación no aumenta es numéricamente estable.

En estos casos es más rápido utilizar un método implícito con pasos de tiempo más grandes. De este modo, aunque el cálculo de cada paso de forma individual es más costoso, se requieren menos pasos para realizar una simulación estable.

3.2.1. Aplicación a la 2ª Ley de Newton

Aplicando el método de *backward* Euler a la 2º Ley de Newton para calcular la dinámica de las partículas y obtener las nuevas posiciones y velocidades, se obtienen las siguientes ecuaciones:

- Fórmula de integrador de *backward* Euler:

$$x(t_0 + h) = x(t_0) + \frac{dx}{dt}(t_0 + h) \cdot h$$

- Aplicación a 2ª ley de Newton(2 ODEs orden 1):

$$x(t + h) = x(t) + h \cdot v(t + h)$$

$$v(t + h) = v(t) + h \cdot a(t + h)$$

Al aplicarlo a la 2ª ley de Newton hay una dificultad fundamental, no se pueden evaluar las fuerzas en el nuevo paso de tiempo, así que se aproximan las derivadas con los valores aproximados mediante un polinomio de Taylor de primer orden 1 (es decir, una linealización), basado en valores 'futuros'. De este modo, las fuerzas quedan expresadas en función de las nuevas velocidades. Este método tiene un error de orden uno en el tiempo.

- Multiplicando la ecuación de la velocidad por la masa:

$$x(t + h) = x(t) + h \cdot v(t + h)$$

$$m \cdot v(t + h) = m \cdot v(t) + h \cdot F(x(t + h), v(t + h))$$

- Linealización de la fuerza:

$$F(x(t+h), v(t+h)) = F(x(t), v(t) + \frac{dF}{dx}(x(t+h) - x(t)) + \frac{dF}{dv}(v(t+h) - v(t))$$

- Sustituyendo fuerzas y la integración de posición en la ecuación de velocidad:

$$(m - h\frac{dF}{dv} - h^2\frac{dF}{dx}) \cdot v(t+h) = (m - h\frac{dF}{dv}) \cdot v(t) + h \cdot F(x(t), v(t))$$

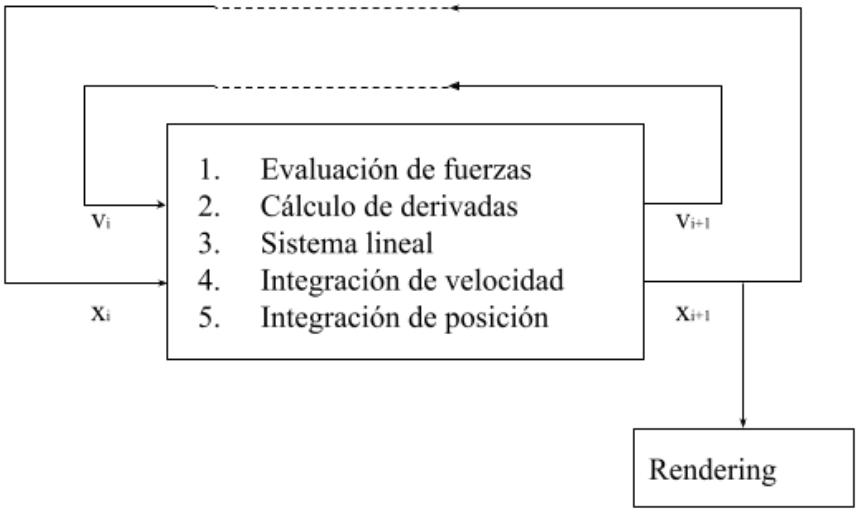

Figura 3.2: Diagrama del algoritmo de simulación mediante *backward* Euler.

Existe la posibilidad de reinterpretar las ecuaciones vistas anteriormente desde otro punto de vista. A la hora de realizar una simulación se resuelven las ecuaciones que proporcionarán el siguiente estado de la simulación, pero en la optimización se realiza una estimación teniendo en cuenta que se cumplan las ecuaciones de la física.

$$\begin{aligned} x_{i+1} - x_i - h \cdot v_{i+1} &= 0 \\ c = M \cdot v_{i+1} - M \cdot v_i - h \cdot f_{i+1} &= 0 \end{aligned} \tag{3.2}$$

Por tanto, más adelante se explicará como estas nuevas ecuaciones se utilizarán como restricciones en el proceso de optimización para garantizar que se sigan las leyes de la física.

3.3. Fijado de vértices

Es imprescindible contar con la funcionalidad de fijar vértices, es decir, excluir ciertos vértices de las físicas para que estos se queden fijos en el espacio. De este modo se evita que caigan al vacío y se puede ver cómo se comporta la tela.

El *fixing* se puede interpretar como una *constraint* adicional, que fija las velocidades de algunos de los vértices a 0.

$$S\ v_{i+1} = 0$$

Donde S es una matriz de selección (tiene identidad en la diagonal de las filas seleccionadas, y cero en el resto). Con un *fixing* tan sencillo, la ecuación de las restricciones (3.2) se puede reescribir como:

$$c = M\ v_{i+1} – M\ v_i - h(I – S^T S)F = 0$$

Donde $I – S^T S$ es una matriz que pone a 0 todas las componentes de F que están fijadas. Esta formulación aprovecha que v_i ya tiene velocidades a 0 donde corresponde, y M es diagonal. En un caso más general sería algo más complicada. En la práctica tan solo ha sido necesario poner a 0 las filas correspondientes de $\frac{dF}{dp}$.

3.4. Optimización

La optimización implica encontrar valores para los parámetros de entrada que maximicen o minimicen una función objetivo o de coste. En este caso, dada una escena con uno o varios objetos a simular, se ha usado esta técnica para encontrar la combinación de parámetros que proporcionen una simulación lo más similar posible a un comportamiento objetivo, generado a partir de unos parámetros desconocidos. Para hacer el proceso más eficiente se ha usado un algoritmo de optimización basado en gradiente, por lo que ha sido necesario calcular la derivada de la función objetivo.

3.4.1. Algoritmo de optimización

Existen multitud de algoritmos de optimización que permiten resolver problemas de optimización de forma más o menos eficiente. Los algoritmos de optimización del método de Newton son aquellos algoritmos que hacen uso de la segunda derivada de la función objetivo para encontrar este mínimo.

La primera derivada de una función es la tasa de cambio o la pendiente de

la función en un punto específico. Por tanto, la derivada puede seguirse cuesta abajo mediante un algoritmo de optimización hacia los mínimos de la función (los valores de entrada que dan como resultado la salida más pequeña de la función objetivo).

La derivada de segundo orden es la derivada de la derivada, o la tasa de cambio de la curvatura. La segunda derivada se puede utilizar para encontrar de manera más eficiente los valores óptimos de la función objetivo. La derivada de segundo orden nos permite saber en qué dirección movernos (al igual que la de primer orden), pero también permite estimar cuánto movernos en esa dirección, lo que se denomina el tamaño de paso. Los algoritmos que hacen uso de la derivada de segundo orden se denominan algoritmos de optimización de segundo orden.

BFGS es un algoritmo de optimización de segundo orden. Es un acrónimo, denominado así por sus cuatro co-descubridores: Broyden, Fletcher, Goldfarb y Shanno. Pertenece a un grupo de algoritmos que son una extensión del algoritmo de optimización del método de Newton, denominados métodos cuasi-newtonianos.

El método de Newton es un algoritmo de optimización de segundo orden que utiliza la matriz hessiana, la matriz de derivadas segundas, pero cuya limitación es que requiere el cálculo de la inversa de ésta. Esta es una operación computacionalmente costosa y puede no ser estable dependiendo de las propiedades de la función objetivo.

Los métodos cuasi-Newton son algoritmos de optimización de segundo orden que aproximan la inversa de la matriz hessiana mediante el gradiente, lo que significa que no es necesario que la hessiana y su inversa estén disponibles o calculadas con precisión para cada paso del algoritmo.

La principal diferencia entre los diferentes algoritmos de optimización cuasi-newtonianos es la forma específica en que se calcula la aproximación de la hessiana inversa.

El algoritmo BFGS es una forma específica de actualizar el cálculo de la hessiana inversa, en lugar de volver a calcularla en cada iteración. BFGS aproxima la hessiana mediante sucesivas evaluaciones del gradiente.

Este, o sus extensiones, es uno de los algoritmos de optimización cuasi-newtonianos o incluso de segundo orden más populares utilizados para la optimización numérica.

Existen variaciones del método como la L-BFGS, una versión que limita el uso de memoria, especialmente interesante en problemas con muchos parámetros (más de 1000). También existe BFGS-B, que añade la posibilidad de añadir restricciones a los parámetros. En el caso de este proyecto, por ejemplo, se han restringido todos los números iguales o inferiores a 0.

En el caso de este trabajo, se ha usado una librería que ya implementa el

algoritmo de BFGS, por lo que no es necesario profundizar en el funcionamiento interno del algoritmo.

3.4.2. Función de coste

En un problema de optimización es fundamental escoger una función de coste que, dado un conjunto de parámetros, devuelva un valor que indique la diferencia o el error entre una simulación realizada con los parámetros estimados actuales y el objetivo ideal. Esta función de coste será utilizada por el algoritmo de optimización para hallar el valor de los parámetros que devuelvan el menor valor posible en la función de coste.

$$p = arg\ min\ g(x_n, x_{n-1}, ...\ p)$$

En la función anterior g es la función de coste. p es el conjunto de parámetros que se proporciona a la simulación, los cuales definen el comportamiento de esta y el consecuente aumento o decremento del coste, en este caso se trata de las masas de los nodos o las rigideces de los muelles. Y $x_n, x_{n-1}, ...$ es el estado inicial de la simulación, definido mediante las posiciones y velocidades de cada uno de los vértices.

La dificultad del problema radica en definir el error a lo largo de cada uno de los pasos de la simulación y, más complicado aún, calcular el gradiente de dicho error. El error se acumula a lo largo de todos los pasos y se debe encontrar una forma de definir cómo afecta el cambiar del valor de cada uno de los parámetros al error a lo largo de un número de pasos de la simulación.

Una función de coste simple pero efectiva que tiene en cuenta no solo el estado final de la simulación, sino la trayectoria completa, consiste en sumar las diferencias de posición de cada uno de los vértices respecto a la simulación objetivo y acumularlas a lo largo de cada uno de los pasos de la simulación. La fórmula sería:

$$g = \sum_i |x_i(p) - x_i^*)|^2 \tag{3.3}$$

3.4.3. Cálculo del gradiente con *backpropagation*

Por otro lado, para el cálculo de gradiente se ha utilizado el método de *backpropagation*. En el campo de aprendizaje automático, *backpropagation* es un algoritmo ampliamente utilizado para entrenar redes neuronales. Consiste en ajustar los pesos de una red neuronal en función de la tasa de error (es decir, coste)

obtenida en una iteración anterior mediante la regla de la cadena.

Aplicado al contexto actual, se puede utilizar para construir la derivada de la función de coste respecto a cada uno de los parámetros de la simulación, es decir, el gradiente de la función de coste.

El algoritmo se basa en el método de la derivación por regla de cadena, el cual permite derivar una composición de funciones.

Si tenemos una función compuesta de la forma

$$F(x) = f(u(x))$$

entonces su derivada respecto a x está dada por

$$F'(x) = f'(u) * u'(x)$$

o en notación con diferenciales

$$\frac{dF}{dx} = \frac{df}{du}\frac{du}{dx}$$

La derivada en un paso concreto es un parámetro de la derivada del paso siguiente. Por tanto, para obtener el gradiente, hay que ir "deshaciendo" operaciones desde la salida (el último paso) hasta la entrada (el estado de la simulación antes de empezar) aplicando la regla de la cadena en cada iteración. Aplicable a cualquier concatenación de funciones. A cada uno de los pasos se le llama "paso *backward*".

De este modo se obtiene la derivada parcial de cada paso con respecto al anterior y se multiplica por las derivadas parciales de los pasos anteriores para así obtener el gradiente.

3.5. Cálculo del gradiente en el problema actual

Se han aplicado las técnicas descritas anteriormente para hallar los parámetros físicos de uno o más objetos en una escena de modo que la simulación resultante sea lo más parecida a una simulación objetivo, generada mediante unos parámetros fijos pero desconocidos para el sistema.

Los parámetros de los objetos que se pueden optimizar son la masa de sus nodos o la rigidez de los muelles que los forman. Tanto las masas como las rigideces se pueden optimizar de forma homogénea (un mismo valor del parámetro para todos los nodos o muelles) o heterogénea (un valor independiente para cada uno de los nodos o muelles).

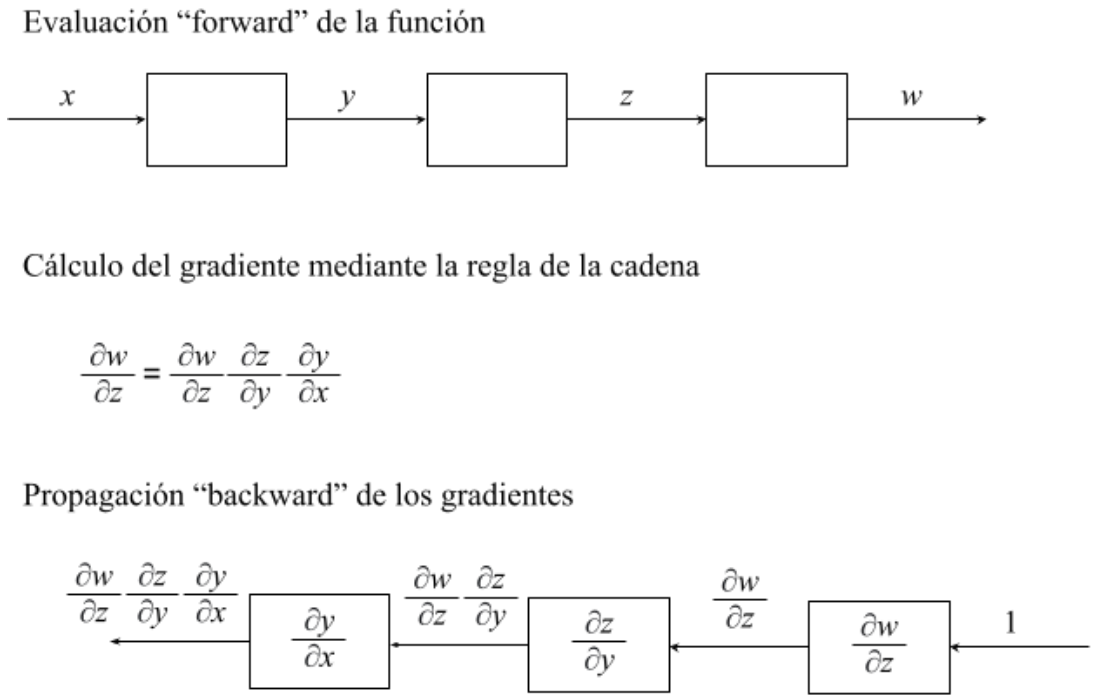

Figura 3.3: Cálculo del gradiente mediante *backpropagation*.

3.5.1. Bloque *forward*

Entender el bloque *forward* es el primer paso para el cálculo del gradiente de la función de coste. La función de este bloque es, dado un estado de la simulación y uno conjunto de parámetros, generar el siguiente estado de esa simulación tras un cierto paso de tiempo.

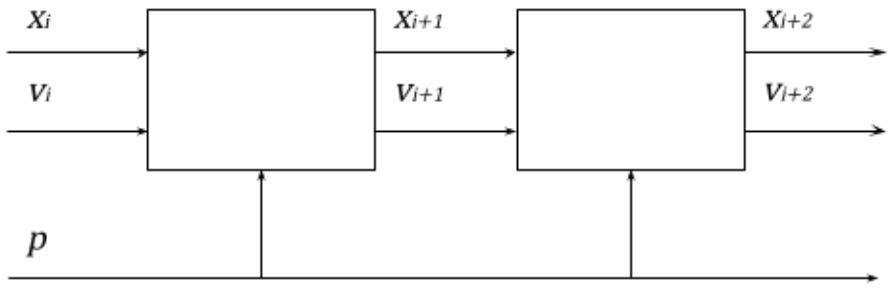

Figura 3.4: Diagrama del paso *forward*.

El estado de la simulación se define como el conjunto de posiciones y velocidades de cada uno de los nodos de la simulación, y los parámetros son las masas de los nodos y las rigideces de los muelles.

3.5.2. Bloque *backward*

En el bloque *backward* se calcula la derivada de la función de coste respecto a los parámetros, la posición y la velocidad en un paso concreto a partir de los estados actual y siguiente y las derivadas del paso siguiente. El objetivo es obtener las derivadas parciales respecto a los pasos siguientes.

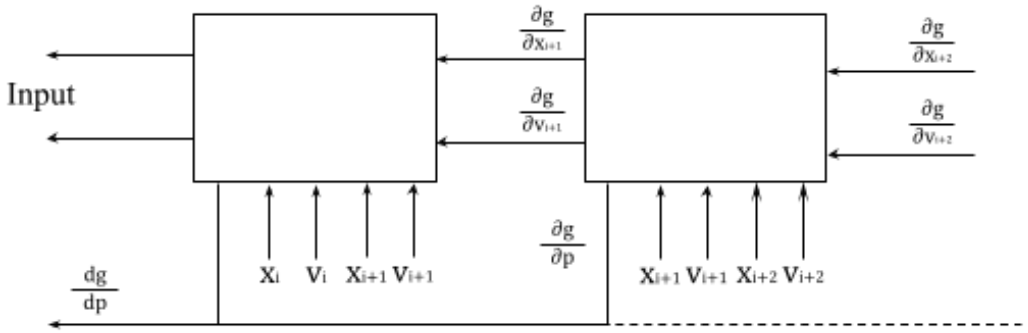

Figura 3.5: Diagrama del paso *backward*.

Para ello, se ha utilizado el método del adjunto, que consiste en concadenar los resultados anteriores a la expresión principal, cambiando el orden de las operaciones. Esto es a lo que se le conoce como propagar el gradiente a lo largo de una simulación.

$$\frac{dg}{dp} = \frac{\partial g}{\partial p} + \frac{\partial g}{\partial x}\frac{dx}{dp} = \frac{\partial g}{\partial p} - \frac{\partial g}{\partial x}\frac{\partial c}{\partial x}^{-1}\frac{\partial c}{\partial p}$$

El paso *backward* solo requiere resolver un sistema lineal por paso, por lo que no es más costoso que la propia simulación.

$u^T = \frac{\partial g}{\partial x}\frac{\partial c}{\partial x}^{-1}$ puede ser obtenido resolviendo el sistema lineal $\frac{\partial c}{\partial x}^T u = \frac{\partial g}{\partial x}^T$

Para calcular el gradiente se expresa éste como la salida del paso.

$$\frac{dg}{dp} = \frac{\partial g}{\partial x_{i+1}}\frac{dx_{i+1}}{dp} + \frac{\partial g}{\partial v_{i+1}}\frac{dv_{i+1}}{dp}$$

A continuación, se expresa respecto a las entradas del paso, usando la regla de la cadena y las jacobianas de la salida respecto a las entradas.

$$\frac{dx_{i+1}}{dp} = \frac{\partial x_{i+1}}{\partial p} + \frac{\partial x_{i+1}}{\partial x_i}\frac{dx_i}{dp} + \frac{\partial x_{i+1}}{\partial v_i}\frac{dv_i}{dp}$$

$$\frac{dv_{i+1}}{dp} = \frac{\partial v_{i+1}}{\partial p} + \frac{\partial v_{i+1}}{\partial x_i}\frac{dx_i}{dp} + \frac{\partial v_{i+1}}{\partial v_i}\frac{dv_i}{dp}$$

$$\frac{dg}{dp} = \left(\frac{\partial g}{\partial x_{i+1}}\frac{i+1}{\partial p} + \frac{\partial g}{\partial v_{i+1}}\frac{\partial v_{i+1}}{\partial p}\right) + \left(\frac{\partial g}{\partial x_{i+1}}\frac{i+1}{\partial xi} + \frac{\partial g}{\partial v_{i+1}}\frac{\partial v_{i+1}}{\partial x_i}\right)\frac{dx_i}{dp} + \left(\frac{\partial g}{\partial x_{i+1}}\frac{i+1}{\partial v_i} + \frac{\partial g}{\partial v_{i+1}}\frac{\partial v_{i+1}}{\partial v_i}\right)\frac{dv_i}{dp}$$

Y así es como finalmente se obtiene la estructura del paso *backward*.

$$\left(\frac{\partial g}{\partial p}, \frac{\partial g}{\partial x_i}, \frac{\partial g}{\partial v_i}\right) = Backward\left(x_{i+1}, v_{i+1}, p, \frac{\partial g}{\partial x_{i+1}}, \frac{\partial g}{\partial v_{i+1}}\right)$$

Dentro del paso *backward*, para el cálculo de las jacobianas, se toman las derivadas parciales con respecto a las entradas para sustituirlas en la ecuación de la dinámica reinterpretada como una restricción, como ya se ha explicado anteriormente (3.2), y se aplica el teorema de la función implícita para obtener las derivadas de cada uno de los parámetros.

$$x_{i+1} - x_i - hv_{i+1} = 0 \quad \frac{\partial x_{i+1}}{\partial p} = h\frac{\partial v_{i+1}}{\partial p} \quad \frac{\partial x_{i+1}}{\partial x_i} = I + h\frac{\partial v_{i+1}}{\partial x_i} \quad \frac{\partial x_{i+1}}{\partial v_i} = h\frac{\partial v_{i+1}}{\partial v_i}$$

$$M\frac{\partial v_{i+1}}{\partial x_i} - h\frac{\partial f}{\partial v}\frac{\partial v_{i+1}}{\partial x_i} - h\frac{\partial f}{\partial x}\frac{\partial x_{i+1}}{\partial x_i} = 0 \rightarrow \frac{\partial v_{i+1}}{\partial x_i} = \left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)^{-1} h\frac{\partial f}{\partial x}$$

$$M\frac{\partial v_{i+1}}{\partial v_i} - M - h\frac{\partial f}{\partial v}\frac{\partial v_{i+1}}{\partial v_i} - h\frac{\partial f}{\partial x}\frac{\partial x_{i+1}}{\partial v_i} = 0 \rightarrow \frac{\partial v_{i+1}}{\partial v_i} = \left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)^{-1} M$$

$$M\frac{\partial v_{i+1}}{\partial p} - h\frac{\partial f}{\partial v}\frac{\partial v_{i+1}}{\partial p} - h\frac{\partial f}{\partial x}\frac{\partial x_{i+1}}{\partial p} + \frac{\partial c}{\partial p} = 0 \rightarrow \frac{\partial v_{i+1}}{\partial p} = -\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)^{-1}\frac{\partial c}{\partial p}$$

Poniendo todo junto quedan las tres ecuaciones para calcular cada una de las

salidas:

$$\frac{\partial g}{\partial p} = \frac{\partial g}{\partial x_{i+1}}\frac{\partial x_{i+1}}{\partial p} + \frac{\partial g}{\partial v_{i+1}}\frac{\partial v_{i+1}}{\partial p} =$$
$$\left(h\frac{\partial g}{\partial x_{i+1}} + \frac{\partial g}{\partial v_{i+1}}\right)\frac{\partial v_{i+1}}{\partial p} = -\left(h\frac{\partial g}{\partial x_{i+1}} + \frac{\partial g}{\partial v_{i+1}}\right)\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)^{-1}\frac{\partial c}{\partial p}$$

$$\frac{\partial g}{\partial x_i} = \frac{\partial g}{\partial x_{i+1}}\frac{\partial x_{i+1}}{\partial x_i} + \frac{\partial g}{\partial v_{i+1}}\frac{\partial v_{i+1}}{\partial x_i} =$$
$$\frac{\partial g}{\partial x_{i+1}} + \left(h\frac{\partial g}{\partial x_{i+1}} + \frac{\partial g}{\partial v_{i+1}}\right)\frac{\partial v_{i+1}}{\partial x_i} =$$
$$\frac{\partial g}{\partial x_{i+1}} + \left(h\frac{\partial g}{\partial x_{i+1}} + \frac{\partial g}{\partial v_{i+1}}\right)\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)^{-1} h\frac{\partial f}{\partial x}$$

$$\frac{\partial g}{\partial v_i} = \frac{\partial g}{\partial x_{i+1}}\frac{\partial x_{i+1}}{\partial v_i} + \frac{\partial g}{\partial v_{i+1}}\frac{\partial v_{i+1}}{\partial v_i} =$$
$$\left(h\frac{\partial g}{\partial x_{i+1}} + \frac{\partial g}{\partial v_{i+1}}\right)\frac{\partial v_{i+1}}{\partial v_i} = \left(h\frac{\partial g}{\partial x_{i+1}} + \frac{\partial g}{\partial v_{i+1}}\right)\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)^{-1} M$$

Y una vez simplificadas quedan así:

$$\frac{\partial g}{\partial p} = -u^T\frac{\partial c}{\partial p} \qquad \frac{\partial g}{\partial x_i} = \frac{\partial g}{\partial x_{i+1}} + hu^T\frac{\partial f}{\partial x} \qquad \frac{\partial g}{\partial v_i} = u^T M$$

Siento u la solución del siguiente sistema lineal:

$$\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)u = h\frac{\partial g^T}{\partial x_{i+1}} + \frac{\partial g^T}{\partial v_{i+1}}$$

La derivada de la restricción c depende de los parámetros a optimizar. En el caso de las masas es:

$$\frac{\partial c}{\partial p} = v_1 - v$$

Mientras que para las rigideces es:

$$\frac{\partial c}{\partial p} = h * \frac{\partial F}{\partial p}$$

3.5.3. Algoritmo

Con los bloques *forward* y *backward* ya construidos es momento de construir el algoritmo de *backpropagation* para el cálculo del gradiente de la función de coste.

El primer paso es realizar los n pasos del paso *forward* y almacenar cada uno de los estados. A continuación, se inicializar las derivadas parciales de la posición y velocidad en cada uno de los pasos de la simulación. Para ello, en el caso de las posiciones se usa la derivada local de la función de coste respecto a las posiciones, expresada como:

$$\frac{dg}{dx_i} = 2(x_i - x_i^*)$$

Y para las velocidades se inicializan todos los pasos a 0, ya que en esta función de coste en concreto no aparecen velocidades. Además, se inicializa $\frac{dg}{dp}$ como la derivada parcial $\frac{\partial g}{\partial p}$ en el último paso, que en este caso también es 0.

Una vez calculadas las derivadas parciales se procede a recorrer cada uno de los pasos de la simulación en sentido inverso, es decir, empezando por el final y recorriéndolos todos hasta llegar al principio. Para cada paso recorrido se hace uso del bloque *backward* para calcular las derivadas parciales de posiciones, velocidades y coste. Es importante empezar por el final porque el paso *backward* hace uso de las derivadas de la posición y velocidad del paso siguiente.

for i = n − 1 to 0

$$\left(\Delta\frac{\partial g}{\partial p}, \Delta\frac{\partial g}{\partial x_i}, \Delta\frac{\partial g}{\partial v_i}\right) = \ Backward \ \left(x_{i+1}, v_{i+1}, p, \frac{\partial g}{\partial x_{i+1}}, \frac{\partial g}{\partial v_{i+1}}\right)$$

$$\frac{dg}{dp} += \Delta\frac{\partial g}{\partial p}$$
$$\frac{\partial g}{\partial x_i} += \Delta\frac{\partial g}{\partial x_i}$$
$$\frac{\partial g}{\partial v_i} += \Delta\frac{\partial g}{\partial v_i}$$

Las derivadas parciales del coste se van sumando en cada paso, y el resultado final es el gradiente de la función de coste. Un vector con tantas posiciones como parámetros a optimizar haya en la simulación que contiene la derivada del coste respecto a cada uno de estos parámetros.

4

Descripción informática del problema

4.1. Diseño e implementación

El proyecto consta de tres módulos separados, cada uno en un lenguaje distinto acorde a las necesidades de cada uno de ellos.

En primer lugar, el motor de simulación diferenciable desarrollado en C++ se encarga de los cálculos más costosos en un entorno que proporciona el máximo rendimiento posible y permite sacar partido de herramientas de *multithreading*.

En segundo lugar, se ha desarrollado una interfaz en *Unity* con C# que permite al usuario configurar fácilmente las simulaciones y visualizar el resultado de las estimaciones.

Por último, un módulo de *Python* que hace uso de avanzadas herramientas de optimización y *machine learning* para implementar el algoritmo de *backpropagation* para la estimación de los parámetros de la simulación.

4.2. Motor de simulación

El objetivo del motor es, dada una escena en formato *JSON*, simular los infinitos estados de la simulación siguientes y devolver estos estados como una colección de posiciones de vértices. El archivo de la escena contiene tanto información global de la escena común para todos los objetos como información exclusiva de cada

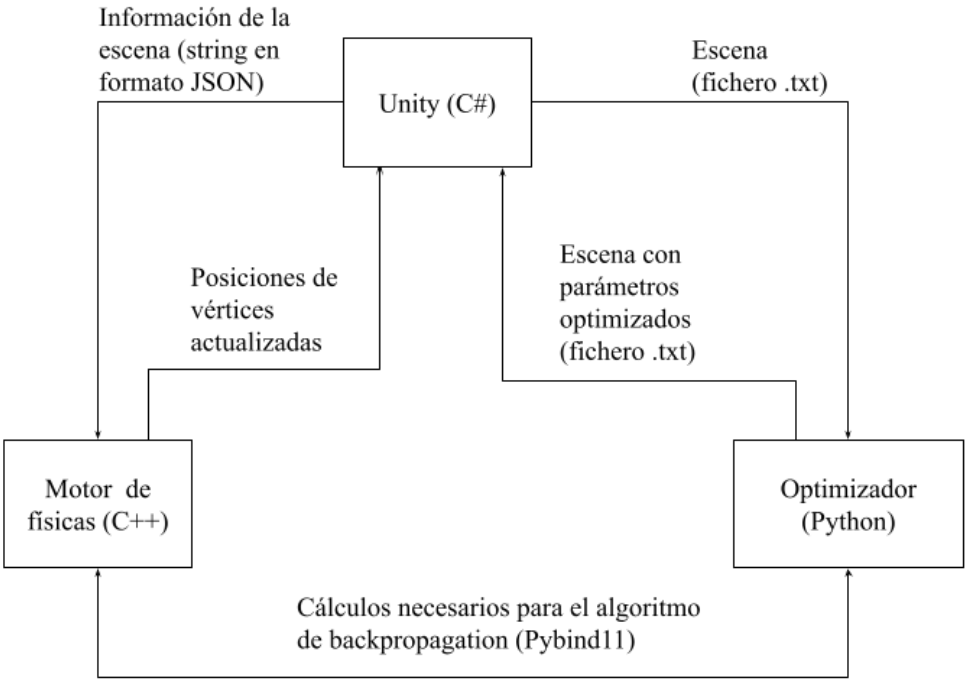

Figura 4.1: Estructura del *framework*.

objeto, así como parámetros individuales para cada uno de los nodos o muelles que lo forman.

Este programa deberá ejecutar un gran número de cálculos realmente costosos lo más rápido posible. Por ello, se ha escogido desarrollarlo en C++, un lenguaje versátil y potente que proporciona un rendimiento superior frente a otras alternativas.

4.2.1. Implementación del *backward* Euler

El motor se encarga de realizar todos los cálculos del método de integración de *backward* Euler descritos anteriormente en la descripción matemática del problema. Las fórmulas describen el comportamiento como un estado que agrupa a todas las partículas, pero representar y operar sobre dicho estado no resulta trivial.

No se puede recorrer cada nodo en un bucle realizando los cálculos necesarios porque para calcular la posición de cada nodo se tiene en cuenta el estado de cada uno de los otros nodos, así como la derivada de las posiciones de cada uno de ellos. Por ello, se ha replanteado el problema utilizando notación matricial, lo

cual soluciona todos los problemas anteriores. Para ello, ha sido necesario llevar a cabo un proceso de ensamblado de las matrices que representan cada uno de los parámetros de las ecuaciones. La información de cada uno de los nodos ha sido almacenada en las posiciones correspondientes de las respectivas matrices.

De este modo, en lugar de tener n vectores de tres posiciones para cada uno de los nodos, se construye un vector de 3 x n posiciones donde se concadenan las posiciones de cada uno de los nodos. Este mismo proceso se realiza para las velocidades y las fuerzas.

$$[x_0, y_0, z_0, x_1, y_1, z_1, \dots]$$

Por su parte, las masas se agrupan en una matriz de 3n x 3n, donde las masas de cada nodo se colocan en la diagonal de la matriz.

$$\begin{bmatrix} m_0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & m_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & m_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & m_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots \end{bmatrix}$$

Por último, con las derivadas de posiciones y velocidades también se construye una matriz, aunque en este caso no solo se rellena la diagonal. En estas matrices se almacenan los valores en las posiciones correspondientes según la fuerza de cada nodo respecto a cada uno de los nodos que le influencian.

Se ha utilizado Eigen para la gestión de las matrices, así como los cálculos que se realizan con ellas. Eigen es una librería de alto nivel escrita en C++ para álgebra lineal, operaciones matriciales y vectoriales, transformaciones geométricas, *solvers* numéricos y algoritmos relacionados. Ha facilitado enormemente el trabajo, ya que ha proporcionado una implementación eficiente de las operaciones a realizar con las matrices, así como varias herramientas para solucionar sistemas de ecuaciones lineales.

Una de las funcionalidades de Eigen que más ha beneficiado a este proyecto es su implementación de matrices dispersas. En caso de haber usado matrices convencionales, el rendimiento habría sido muy inferior, ya que éstas almacenan en memoria los valores de todas las posiciones de la matriz, incluso cuando su valor es 0. Por tanto, al aumentar el número de nodos en la escena, el tamaño de las matrices crece de forma exponencial.

Las matrices dispersas, por otro lado, tienen la ventaja de que tan solo almacenan aquellos valores de la matriz distintos a 0. Por tanto, en matrices poco

pobladas (como es el caso en la mayoría de las matrices de este proyecto) se requiere almacenar muchísima menos información y, a la hora de hacer cálculos, se ahorran muchas operaciones en las que se suman o restan 0, por lo que no afectarían al resultado final.

Otra funcionalidad de Eigen que se ha utilizado es su *solver* de sistemas de ecuaciones lineales, el cual se ha utilizado tanto para el cálculo del *backward* Euler como para el cálculo del paso *backward*. Eigen ofrece varios métodos para resolver sistemas lineales cuando la matriz de coeficientes es dispersa, aunque tras probar varios y medir su tiempo de ejecución el escogido fue el algoritmo iterativo del gradiente conjugado.

	h = 0,1	h = 0,01
Inicialización	0,05 ms	0,06 ms
Cálculo de fuerzas	1,78 ms	1,28 ms
Construcción de matrices	1,05 ms	1,18 ms
Operaciones con matrices	0,35 ms	0,42 ms
Fijado de vértices	0,05 ms	0,06 ms
Solver	1,85 ms	0,75 ms
Total	**5,12 ms**	**3,75 ms**

Tabla 4.1: Tiempo de ejecución medio de cada una de las fases del cálculo del paso mediante *backward* Euler.

Como se puede observar en la tabla 4.1, al aumentar el paso de tiempo al *solver* le cuesta más tiempo encontrar una solución para el sistema. No obstante, al usar un paso de tiempo mayor, disminuye la frecuencia de los cálculos, es decir, se necesitan menos pasos para integrar un mismo espacio de tiempo. En el ejemplo anterior, por ejemplo, para un segundo de simulación con el paso de tiempo pequeño serían necesarios 375 milisegundos, mientras que con el paso de tiempo grande tan solo serían 51.2 milisegundos.

4.2.2. *Multithreading*

El motor ofrece la opción de hacer uso de técnicas de paralelismo para mejorar enormemente la fluidez a la hora de usarlo en aplicaciones en tiempo real. En caso de especificarlo, el motor se inicializará en un hilo aparte y podrá ser accedido en cualquier momento para consultar el estado de la escena.

Para ello, ha sido necesario hacer uso de herramientas de sincronización de C++, así como aplicar técnicas para solucionar los problemas que esto podría

acarrear.

El problema principal es garantizar que en el momento en que se solicita la información de la escena desde fuera del motor, esta información no esté siendo escrita. Para ello, se podría utilizar un *mutex* en el *array* donde se almacena el resultado de los cálculos, pero esto tendría una gran desventaja. Si se intentara acceder a esta información mientras se estuvieran realizando los cálculos, el hilo del programa desde el que se accede al motor debería pausarse y esperar a que terminaran los cálculos.

Para solucionar este problema, se ha añadido un segundo *array* auxiliar al que se copia la información de la escena cuando se terminan los cálculos y del que se extrae la información cuando ésta es solicitada y se ha utilizado un *mutex* para garantizar que no se escriba en dicho *array* mientras está siendo leído y viceversa.

4.2.3. *Backpropagation*

Pese a que el algoritmo de *backpropagation* se ha implementado en *Python* y se describirá más adelante, el motor necesita implementar las funciones para los pasos *backward* y *forward*.

La implementación del paso *forward* consiste en utilizar de nuevo el *backward* Euler, aprovechando la función ya implementada previamente. De este modo, dado un estado de una escena, se calcula el siguiente estado tras un paso de tiempo.

El paso *backward* recibe como parámetro un *array* de valores de parámetros que contiene todos los parámetros a optimizar, independientemente de si pertenecen a distintos objetos, si se tratan de masas o rigideces o incluso si son parámetros homogéneos (un mismo valor para todo el conjunto de nodos o muelles) o heterogéneos (un valor distinto para cada nodo o muelle).

Cabe recordar que para obtener el gradiente en el paso *backward* es necesario calcular $\frac{dc}{dp}$, y éste es distinto según los parámetros a optimizar. Por tanto, para poder obtener $\frac{dc}{dp}$ se subdivide el *array* de parámetros en varios *arrays*, agrupados según los parámetros que representan, se opera con ellos y luego se concadenan los *arrays* resultantes de estas operaciones en un nuevo *array*, $\frac{dc}{dp}$.

4.2.4. DLL

La librería se ha exportado como una DLL, es decir una librería dinámica de enlaces. Esto permite que se use el código de C++ ya compilado en un *script* de C#, el lenguaje que utiliza *Unity*. Las funciones de la librería pueden ser importadas en C# y usadas normalmente, con la ventaja de que al estar escritas

en C++ el rendimiento es muy superior.

4.3. Proyecto de *Unity*

Se ha utilizado *Unity* para desarrollar una interfaz que permita al usuario configurar fácilmente una escena, ejecutar el optimizador y ver los resultados visualmente de una forma atractiva. Se trata de un proyecto muy simple, ya que el grueso de los cálculos se realiza externamente, así que el objetivo ha sido hacerlo lo más cómodo en intuitivo posible.

La escena consta de un mánager, que se encarga de comunicarse con el motor, y los múltiples objetos que conforman la escena, ya sean objetos simulables o "*fixers*". Estos *fixers* son objetos a los que no les afectan las físicas, pero que fijan los vértices que se encuentran en su interior al inicio de la escena.

Además, sea añadido un menú para ejecutar desde *Unity* el *script* de *Python*, aunque de una forma algo rudimentaria, ya que debido a la incompatibilidad de las versiones utilizadas, no ha sido posible integrar *Python* directamente en *Unity*.

4.3.1. Funcionamiento

Cuando se ejecuta la simulación el mánager se encarga de recopilar la información de todos los objetos a simular, crear un *string* en formato *JSON* que contenga la información de toda la escena (los parámetros generales de la misma y los atributos de cada objeto) e inicializar el motor con dicha información. Además, asocia un identificador único a cada objeto.

Una vez se esté ejecutando el motor, desde el la función de *FixedUpdate* de cada objeto, se solicitan las posiciones de los vértices al *manager*, que a su vez se los solicita al motor. Para ello, se pasa el identificador del objeto, para que el motor devuelva los vértices correspondientes.

En caso de ejecutar el optimizador, se crea el mismo *JSON*, pero se almacena en un fichero *.txt* para leerlo desde *Python*.

4.3.2. Interfaz

En los parámetros de la escena se configurar aspectos generales de la simulación, así como parámetros relativos a la optimización:

- Si el motor funcionará en el mismo hilo de *Unity* o en uno distinto. La segunda opción ofrece más fluidez.

- El método de integración.
- El paso de tiempo.
- La tolerancia del *solver* que se utiliza en el cálculo del *backward* Euler.
- Número de pasos a tener en cuenta para la optimización.

Por otro lado, desde los ajustes de los objetos simulables que se creen se podrá modificar:

- Rigidez.
- Densidad.
- Amortiguamiento.
- Coeficiente de rozamiento con el aire.

Además, se puede configurar fácilmente qué parámetros se desean optimizar, tanto de forma homogénea como heterogénea.

4.4. Módulo de *Python*

Se ha utilizado *Python* para implementar el algoritmo de *backpropagation* debido a que existen herramientas de optimización muy potentes, que facilitan enormemente solucionar problemas como este. El objetivo de este módulo es, dada una escena en la que hay que optimizar ciertos parámetros, hallar los valores de esos parámetros para obtener una simulación lo más similar posible a un comportamiento objetivo.

Dado que se trata de una prueba de concepto y sólo se pretende demostrar la viabilidad del método, el comportamiento objetivo se obtiene simulando la escena con unos parámetros asignados arbitrariamente, para luego ejecutar la optimización y comprobar cuánto se han acercado los valores obtenidos a los valores usados inicialmente.

No obstante, el código es completamente modular, por lo que se podría proporcionar un comportamiento objetivo distinto, siempre y cuando se acompañe de una función de coste y de un gradiente.

4.4.1. *SciPy*

El módulo utilizado para resolver el problema de optimización es *SciPy*, una biblioteca de computación científica basada en *NumPy*. *SciPy* contiene módulos para optimización, álgebra lineal, integración, interpolación, resolución de ODEs y otras tareas para la ciencia e ingeniería. Al igual que *NumPy*, *SciPy* es de código abierto.

SciPy Optimize proporciona funciones para minimizar (o maximizar) funciones objetivo. Dispone de una multitud de métodos de optimización, entre ellos BFGS y su variante L-BFGS-B, de los cuales ya se ha hablado anteriormente. Se han realizado pruebas de eficiencia y se ha confirmado que L-BFGS-B es el método más adecuado para el problema actual, ya que es el más eficiente en cuanto a tiempo.

Para hacer funcionar el optimizador tan solo se necesita proporcionar una función de coste, el gradiente de dicha función y los parámetros iniciales.

La función de coste se ha definido anteriormente, y en el cálculo del gradiente es donde se ha implementado el algoritmo de *backpropagation*.

4.4.2. *Pybind 11*

Para el cálculo del gradiente es necesario hacer uso del motor, ya que los pasos *forward* y *backward* están implementados ahí. Para conectar un módulo de *Python* con una librería de C++ se ha utilizado *Pybind11*, una biblioteca de encabezados que permite exponer tipos de C++ en *Python* y viceversa.

Pybind11 ha resultado una opción especialmente interesante gracias a que implementa nativamente la conversión de tipos de *Eigen* a *NumPy*, por lo que se pueden utilizar directamente con *SciPy*.

Tan solo ha sido necesario definir una interfaz en C++ con las funciones y tipos necesarios y usar *CMake* para compilar el módulo de *Python*.

5 Experimentos

Se han propuesto varias configuraciones de escenas y se han estimado distintos parámetros. Por cada uno de estos experimentos se compara el comportamiento obtenido con el objetivo tanto de forma visual como numéricamente.

Visualmente se muestran dos telas, una tela con un material estándar y otra con un material transparente verde. La tela transparente representa el comportamiento objetivo, mientras que la normal representa el comportamiento obtenido al simular la escena con los parámetros optimizados.

Además, se mide de forma numérica el error en el resultado mediante el error cuadrático medio. Para calcularlo, se suman los errores cuadráticos de todos los nodos a lo largo de todos los *frames*. Luego se divide por el número de nodos multiplicado por el número de *frames*. Finalmente se calcula la raíz cuadrada de este valor. Esto da un error medio que es independiente del número de nodos y *frames*, a diferencia de la función de coste utilizada. Para los siguientes experimentos se han utilizado los mismos parámetros, 100 pasos con un paso de tiempo de 0,02 segundos.

5.1. Experimento 1: un lateral fijado

En primer lugar, se han estimado los parámetros de una tela fijada a lo largo de uno de sus lados. Se han estimado las masas de los nodos y la rigidez de los muelles por separado, tanto de forma homogénea como heterogénea.

Figura 5.1: Lateral fijado, estimación de masa homogénea.

En el caso de la masa homogénea 5.1 se puede apreciar una diferencia visual significativa entre el comportamiento objetivo y el obtenido, pero aun así el resultado es satisfactorio, porque las características de la tela son similares. A lo largo de una simulación de varios segundos, un pequeño error en los parámetros se va acumulando y hace que ambas telas, tras unos segundos, estén en posiciones muy distintas. No obstante, lo importante es que las características sean similares a la muestra. En este sentido se puede observar como la tela con los parámetros optimizados no se estira más que la original, por ejemplo.

Aunque si se puede apreciar que, al tener una masa uniforme para todos los nodos, los nodos de los bordes, que representan menos superficie de la tela, son tienen una masa mayor a la que debería. Esto se traduce en que los bordes de la tela son más densos que el resto de ella, como se puede observar claramente.

El proceso de obtener este parámetro ha durado 48,7 segundos, y el error obtenido es de 2,42. A continuación, se compararán estos valores con los obtenidos estimando otros parámetros.

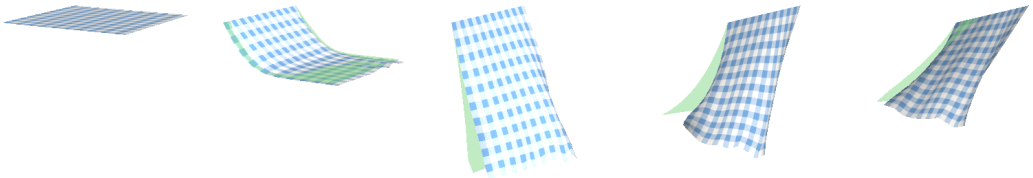

Figura 5.2: Lateral fijado, estimación de masas heterogéneas.

El cálculo de las masas heterogéneas 5.2 ha ofrecido un resultado similar, aunque algo mejor. A simple vista se puede observar cómo, de nuevo, existe una separación entre los dos comportamientos. Pero las características de la tela siguen siendo muy similares, y el problema de los bordes más densos se ha visto claramente mitigado. En este caso se han tardado 73 segundos y el error obtenido es 0,92.

A continuación, se han estimado las rigideces de los muelles, lo cual ha arro-

jado resultados bastante mejores.

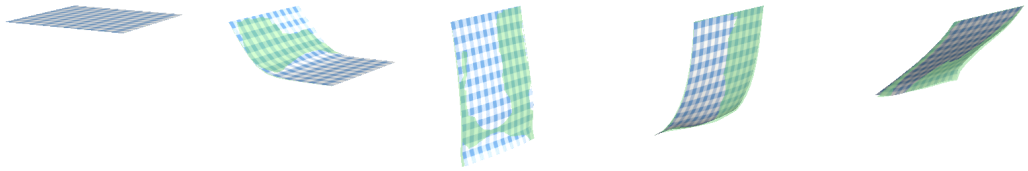

Figura 5.3: Lateral fijado, estimación de rigidez homogénea.

En 109 segundos se ha obtenido un resultado casi perfecto, a pesar de que se trata un parámetro uniforme para todos los muelles y en el objetivo los muelles tienen distintas rigideces. Visualmente 5.3 las dos telas se comportan de forma casi idéntica, y la tasa de error es de tan solo 0,5. Resulta interesante que en caso de haber calculado un valor medio de la rigidez de esos muelles y haber usado este valor como parámetro uniforme, se habría obtenido un resultado sustancialmente peor. En concreto, la tasa de error sería de 6,19.

Usar parámetros uniformes podría permitir ciertas optimizaciones en el motor, aunque no se ha investigado al respecto. Pero en caso de hacerlo, tener una herramienta capaz de escoger un parámetro común a todos los nodos de modo que el comportamiento se asemeje lo más posible al original sería realmente útil.

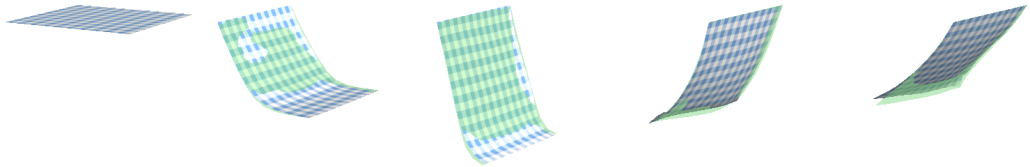

Figura 5.4: Lateral fijado, estimación de rigideces heterogéneas.

La estimación de rigideces de forma heterogénea ha ofrecido resultados aún más impresionantes tanto visualmente 5.4 como numéricamente, pues el error en este caso es de tan solo 0,04. Ha tardado 90 segundos en obtener este resultado casi perfecto.

5.2. Experimento 2: cuatro esquinas fijadas

En el experimento anterior la estimación de las masas ha dejado bastante que desear, pero a continuación se muestra un experimento en el que el resultado

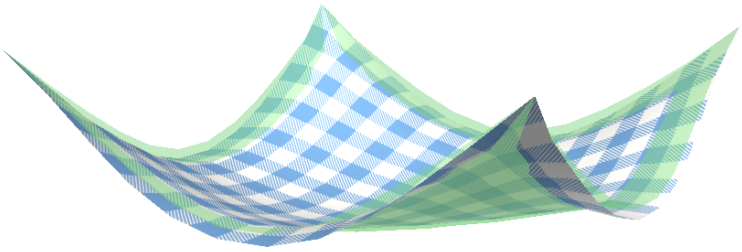

Figura 5.5: Esquinas fijadas, estimación de masa homogénea.

sí ha sido satisfactorio. Se han repetido los mismos experimentos con una tela exactamente igual, pero en este caso fijada por las cuatro esquinas.

La masa homogénea 5.5 obtenida ofrece un comportamiento muy cercano al objetivo, aunque se puede apreciar que, de nuevo, los bordes de la tela son más pesados, mientras que el centro es más ligero. No obstante, la tasa de error es de tan solo 0,82 y ha tardado 130 segundos.

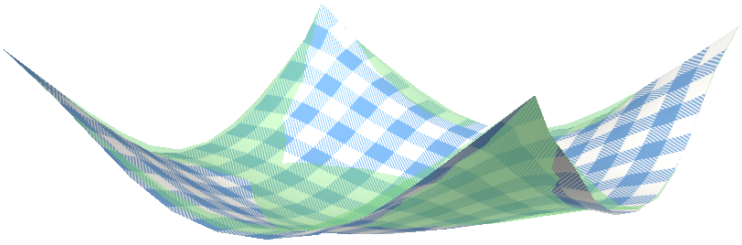

Figura 5.6: Esquinas fijadas, estimación de masas heterogéneas.

La estimación heterogénea 5.6 ha conseguido un resultado mejor, ha conseguido capturar el detalle de que las masas de los nodos de los bordes deben ser

algo más ligeras y el solapamiento es casi perfecto. Ha tardado 73 segundos y el error obtenido ha sido de 0,05.

De este modo queda claro que, dependiendo de la configuración de la escena, ciertos parámetros serán más o menos difíciles de estimar debido a que ejercen una menor influencia en el resultado final. Gracias a estos experimentos cobra sentido el haber utilizado un algoritmo de optimización de segundo orden en el que el gradiente se calcula de forma analítica mediante *backpropagation*. En una misma escena con condiciones idénticas, el número de parámetros no afecta a la complejidad del problema. Como se ha podido observar en los ejemplos anterior, se puede incluso dar el caso de que estimar un solo parámetro tarde más que un gran número de ellos, dependiendo del problema en concreto. Usando métodos de optimización más simples se habría requerido tanto tiempo que no sería viable su utilización, ya que la complejidad crecería de forma exponencial con el número de parámetros.

5.3. Experimento 3: bandera

Por último, se ha configurado un experimento para un caso algo más cercano a un uso real, una bandera ondeando al viento. La tela tiene un lateral fijado y se ha añadido la fuerza del viento a la simulación. Se ha estimado la rigidez de forma heterogénea.

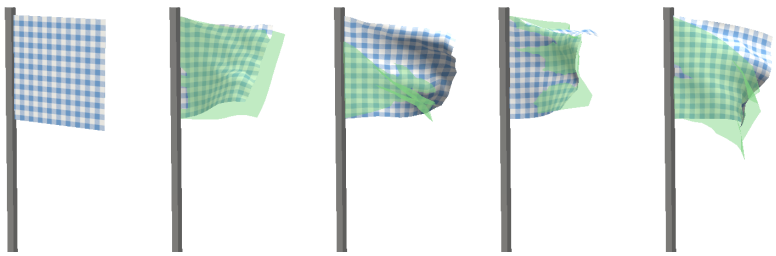

Figura 5.7: Bandera, estimación de rigideces heterogéneas.

La tasa de error obtenida es de tan solo 0,2. A pesar de que se trata de un valor relativamente pequeño, visualmente 5.7 la separación es mucho mayor a los experimentos anteriores. Esto se debe a la fuerza del viento, que añade más variabilidad al sistema, haciendo que el error acumulado a lo largo de los *frames* se intensifique. No obstante, el resultado es una tela que tiene un comportamiento muy similar al objetivo, pues no parece estirarse más de lo que lo hace el original y se arruga de una forma similar.

5.4. Problemas

Durante la realización de los experimentos anteriores se han observado distintos problemas. A continuación se analizarán y se propondrá una posible solución.

5.4.1. Selección de número de pasos de tiempo

Un parámetro clave para la correcta estimación de los parámetros es el número de pasos de simulación empleados para medir el error durante el proceso de optimización. Para evaluar el efecto que tiene, se ha repetido el experimento anterior múltiples veces con distintos valores para observar el efecto que tiene en los resultados.

Al calcular el error cuadrático con un número de pasos bajo, se ha observado que, en ocasiones, un experimento con un error bajo puede ofrecer un comportamiento indeseable tras unos pocos segundos de simulación. Por este motivo, para garantizar la precisión del error, se ha calculado el error cuadrático utilizando los parámetros estimados, pero en una simulación con muchos más pasos.

Pasos	Error n pasos	Error 500 pasos
20	0	0,51
40	0,02	0,6
60	0,08	119,88
80	0,14	139,98
100	0,2	0,6

Tabla 5.1: Ejemplo de la diferencia entre el error obtenido con los n pasos utilizados para estimar los parámetros y el error obtenido con un número de pasos fijo mucho más alto.

Al comparar los dos errores 5.1 se puede observar que el error en la simulación con 500 pasos es siempre mayor. Por lo general es normal que el error aumente, ya que se acumula a lo largo de todos los pasos. No obstante, se puede observar como en el caso de los 60 y 80 pasos el error se dispara. En esta ocasión el optimizador ha encontrado unos parámetros que ofrecen un resultado muy similar al comportamiento objetivo durante los primeros instantes de la simulación, pero a la larga hacen que se desestabilice completamente.

Utilizando el error calculado con más pasos para los experimentos siguientes se garantiza que situaciones como ésta se tendrán en cuenta a la hora de comparar resultados.

Pasos	Error	Tiempo de ejecución(s)
20	0,51	96,1
40	0,6	101,6
60	119,88	65,7
80	139,98	104,7
100	0,6	208
120	400,79	98,7
140	291,82	12,7
160	291,82	15,1
180	291,82	20,7
200	291,82	22,4

Tabla 5.2: Error obtenido utilizando distintos números de pasos al estimar rigideces heterogéneas en la escena de la bandera.

Como se puede observar en los resultados 5.2, al aumentar el número de pasos el error no disminuye. Además, con 60 y 80 pasos el error es muy grande y, a partir de los 100, se dispara. Esto puede deberse a la naturaleza ondulante de la tela al ser golpeada por el viento, lo cual puede haber causado que el optimizador encuentre mínimos locales y se quede atascado.

Al realizar el mismo experimento, pero con la primera escena (un lateral fijado y sin viento) estimando también las rigideces heterogéneas se observa 5.3 que el comportamiento es muy distinto. El error se mantiene estable, aunque también llama la atención que aumentar el número de pasos más allá de 40 no aumenta la precisión del resultado. De nuevo, es probable que se quede atascado en un mínimo local, aunque esta vez mucho más cercano a la solución ideal.

Una posible solución sería diseñar una función objetivo que no mida el error en forma de secuencia temporal, es decir, el error vértice-vértice. Existen alternativas que miden el error mediante un comportamiento más descriptivo de la dinámica, como velocidades, amplitud y frecuencia de las oscilaciones...

El objetivo es encontrar una forma de medir el movimiento característico de la tela mediante una métrica descriptiva simplificada del movimiento de la superficie. Para ello, en proyectos similares se ha usado como métrica las distancias anisotrópicas de los vértices [7] o, en el caso de comparar con un vídeo, los pliegues y la silueta de la tela [5].

Pasos	Error	Tiempo de ejecución(s)
20	0,16	26,5
40	0,02	605,7
60	0,04	171,6
80	0,04	272,5
100	0,04	313,5
120	0,05	643,5
140	0,04	658
160	0,05	567,4
180	0,05	906,1
200	0,05	2899,4

Tabla 5.3: Error obtenido utilizando distintos números de pasos al estimar rigideces heterogéneas en la escena con un lateral fijado.

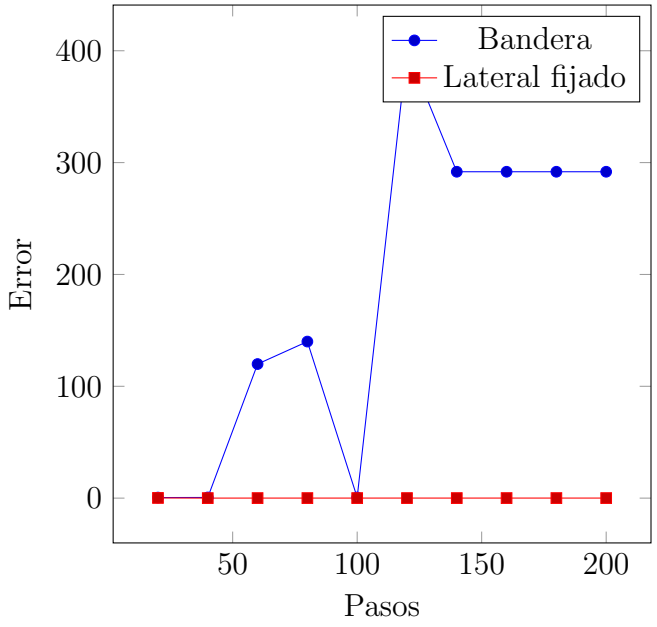

Figura 5.8: Comparación visual entre los errores obtenidos en los dos experimentos anteriores (5.2 y 5.3).

5.4.2. Estimación de parámetros de nodos fijados

El mayor problema de estimar los parámetros de una tela con alguno de sus vértices fijados es que la masa de aquellos nodos que estén fijados o la rigidez de

los muelles que unen dos nodos fijados es imposible de estimar. Por tanto, si la escena se reconfigura y se quiere usar la misma tela habría que volver a ejecutar la optimización.

Figura 5.9: Estimación de las masas fijando los vértices centrales de la tela y, con la información obtenida, simulación de la misma tela pero fijada a lo largo de dos de sus lados.

Como se puede observar 5.9, al estimar los parámetros de la tela con los vértices del centro fijados, se estima una masa muy inferior a la real. Al cambiar el fijado de los vértices se puede ver claramente como los vértices centrales tienen una masa muy inferior a la de los bordes.

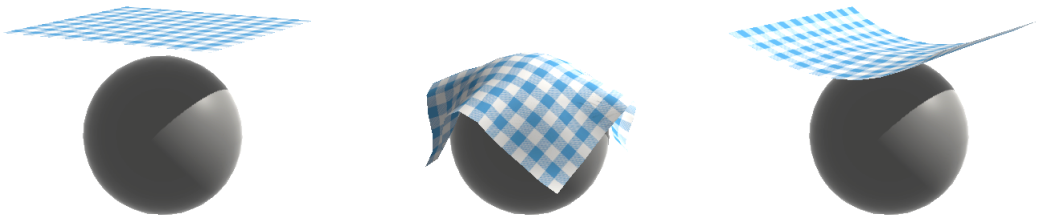

Figura 5.10: Estimación de las masas sin fijar ningún vértice, solo usando colisiones, y simulación de la misma tela pero fijada a lo largo de dos de sus lados.

No obstante, si en lugar de fijar vértices se usan colisiones para generar una escena 5.10, sí se pueden estimar todos los parámetros de la tela. En la última imagen se puede apreciar como las masas de la parte central de la tela sí se corresponden con el resultado esperado .

6

Conclusiones

Se puede afirmar que la prueba de concepto ha sido un éxito. Se ha demostrado la viabilidad de la utilización de un algoritmo de optimización para la edición inversa de animaciones basadas en simulación diferenciable. En los experimentos realizados se puede ver cómo, aunque los resultados no siempre son perfectos, son realmente alentadores y muestran potencial de cara a futuros desarrollos. En el futuro, utilizando los conceptos aquí descritos, se podría crear una herramienta realmente útil que ahorre mucho tiempo a artistas trabajando en el campo de los videojuegos o películas de animación.

El método matemático de *backpropagation* ha sido una herramienta clave no solo para el correcto funcionamiento del *framework*, sino para lograr que este se ejecute a una velocidad suficiente como para que sea una alternativa viable al proceso de selección manual de parámetros. Incluso a pesar de que tanto el motor de físicas como el módulo de *Python* podrían ser enormemente optimizados para mejorar los tiempos de ejecución y reducir cálculos redundantes.

No obstante, el resultado de las estimaciones, no siempre aporta resultados suficientemente precisos. En el estado actual, la configuración de la escena y los parámetros de la optimización juegan un papel muy importante en el error. Utilizar una función objetivo distinta, más avanzada y que no tuviera en cuenta únicamente la posición de los vértices podría solucionar el problema. Por este motivo, se ha tenido en cuenta la necesidad de probar distintas funciones objetivo y se ha proporcionado al usuario la posibilidad de cambiarlas fácilmente en el módulo de *Python*.

6.1. Propuestas de trabajos futuros

De cara a futuro, como ya se ha mencionado en el apartado 5.4, sería muy interesante estudiar y probar distintas funciones objetivo y comparar los resultados obtenidos. Existen multitud de alternativas que se han utilizado en proyectos similares y probablemente ofrecerían resultados mejores. Medir la tasa de error utilizando técnicas más avanzadas podría maximizar la precisión de las estimaciones.

Además, serían necesario hacer la herramienta más versátil, proporcionando una interfaz que simplifique la definición de parámetros a estimar y haciendo que la librería admita distintos tipos de parámetros a optimizar. Actualmente tan solo se pueden estimar masas y rigideces, pues el parámetro a estimar tiene implicaciones en el cálculo de las derivadas de las restricciones y en el motor tan solo están implementadas las derivadas de la restricción según si se estima la masa o la rigidez. Quizás se podría generalizar estos cálculos para admitir cualquier otro parámetro.

También sería interesante desarrollar un *framework* de más alto nivel que conecte las tres herramientas que se han desarrollado (proyecto de *Unity*, librería de C++ y módulo de *Python*) y permita agilizar el proceso de estimación de parámetros. Actualmente hacer funcionar es algo engorroso, ya que se deben seguir varios pasos (generar el archivo *JSON* de la escena en *Unity*, ejecutar el *script* de *Python* y llevar el nuevo *JSON* de vuelta a *Unity* para sustituir los valores en la escena). Además, para modificar la función objetivo o definir nuevos parámetros para optimizar, habría que editar un *script* de *Python*. Todos estos problemas se podrían solucionar con una nueva herramienta que conecte las tres partes del *framework* actual y ofrezca una interfaz gráfica donde introducir los datos y ver directamente el resultado, así como *feedback* visual del proceso de optimización.

Por último, sería imprescindible añadir la opción de utilizar información del mundo real, ya sean vídeos o captura de movimiento, para estimar los parámetros de la escena. Sería necesario un nuevo módulo que, mediante inteligencia artificial, fuera capaz de comparar el resultado de una simulación con los datos proporcionados. La complejidad de esto es alta, pero sin esta opción la herramienta pierde mucho valor en comparación a las alternativas que sí ofrecen la posibilidad.

Bibliografía

[1] J. Liang and M. C. Lin, "Differentiable physics simulation," in *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*, 2019. [Online]. Available: https://openreview.net/forum?id=p-SG2KFY2

[2] S. Zhao, W. Jakob, and T.-M. Li, "Physics-based differentiable rendering: From theory to implementation," in *ACM SIGGRAPH 2020 Courses*, ser. SIGGRAPH '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388769.3407454

[3] E. Miguel, D. Bradley, B. Thomaszewski, B. Bickel, W. Matusik, M. A. Otaduy, and S. Marschner, "Data-driven estimation of cloth simulation models," may 2012. [Online]. Available: http://www.gmrv.es/Publications/2012/MBTBMOM12

[4] J. Perez, B. Thomaszewski, S. Coros, B. Bickel, J. A. Canabal, R. Sumner, and M. A. Otaduy, "Design and fabrication of flexible rod meshes," 2015. [Online]. Available: http://www.gmrv.es/Publications/2015/PTCBCSO15

[5] K. S. Bhat, C. D. Twigg, J. K. Hodgins, P. K. Khosla, Z. Popović, and S. M. Seitz, "Estimating cloth simulation parameters from video," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '03. Goslar, DEU: Eurographics Association, 2003, p. 37–51.

[6] A. Selle, M. Lentine, and R. Fedkiw, "A mass spring model for hair simulation," *ACM Trans. Graph.*, vol. 27, no. 3, p. 1–11, aug 2008. [Online]. Available: https://doi.org/10.1145/1360612.1360663

[7] C. Romero, M. A. Otaduy, D. Casas, and J. Perez, "Modeling and Estimation of Nonlinear Skin Mechanics for Animated Avatars," *Computer Graphics Forum (Proc. Eurographics)*, 2020.