Universitat Rovira i Virgili (URV) y Universitat Oberta de Catalunya (UOC)

Máster Universitario en Ingeniería Computacional y Matemática

TRABAJO FINAL DE MÁSTER

Área: Computación de Altas Prestaciones

Optimization of Muscle Simulation with Extended Position Based Dynamics applying Jacobi and Graph-Coloring Methods

Autor: Carlos Monteagudo Mañas

Tutor: Ester Arroyo Garriguez

Profesor: Josep Jorba Esteve

Madrid, 9 de julio de 2019

Agradecimientos

A mi familia, por apoyarme incondicionalmente y por empujarme a seguir adelante. A Marco, por idear un sistema tan alucinante, por confiar en mí y por mostrarme el camino. A Daniel, por transmitirme su conocimiento desinteresadamente y por tener siempre la respuesta correcta. A Juan, por valorarme cada día y por hacer cuanto ha estado en su mano para facilitarme el trabajo. A Ester y Josep, por tutorizar y supervisar este proyecto con el interés, la flexibilidad y la colaboración que han demostrado. Y a mis compañeros Quique, Rafa, Izar, Tamara, Luis, Víctor y Jorge, porque es una suerte compartir el tiempo con personas así.

Gracias.

Índice general

Índice de figuras

Índice de tablas

Resumen

El presente Proyecto Final de Master versa sobre la optimización del modelo *Muscle Simulation with Extended Position Based Dynamics* de Romeo et al. (2018) aplicando técnicas de paralelización en CPU basadas en *multi-threading*. El modelo original, MSXPBD, es presentado por los autores como un plug-in para Autodesk Maya en Python *single-threaded*, con unos resultados de simulación de muy alta calidad y unos costes de cómputo buenos pero con margen de mejora. En ese contexto, el desarrollo que se expone en este trabajo tiene el objetivo principal de optimizar dicho modelo gracias al uso de herramientas de paralelización para mejorar el rendimiento del sistema.

La aplicación de un sistema de simulación de músculos como el que nos ocupa se encuentra en el sector de los Efectos Visuales para cine y televisión. En este ámbito, los requerimientos de realismo son un aspecto fundamental para el público consumidor, pero desde la perspectiva de un estudio que trabaja en producción es aún más importante si cabe la rapidez con la que éste puede generar contenido de calidad. A raíz de ello surge la motivación de este proyecto, que no es otra que el interés por mejorar y acelerar los sistemas de simulación utilizados en la industria, y más concretamente, los sistemas de simulación de personajes.

El procedimiento que se ha llevado a cabo para la consecución de tal objetivo ha partido de un estudio del modelo de Romeo et al. para su re-implementación en C++ secuencial. Una vez se ha dispuesto del plug-in compilado y funcionando en single-threaded, ha tenido lugar el análisis de los costes de cómputo para, posteriormente, diseñar estrategias adecuadas de optimización sobre los bloques de ejecución menos eficientes. La paralelización del sistema se ha enfocado aplicando el Método de Jacobi y el Método de Coloreado de Grafos. Ambos métodos son paradigmas utilizados con éxito en la literatura sobre *Position Based Dynamics* y más recientemente sobre *Extended Position Based Dynamics*. Este último es, precisamente, el modelo de simulación de deformables del que partieron los autores de MSXPBD.

La implementación con Jacobi y con Coloreado de Grafos ha dado lugar a sendas versiones del plug-in en C++ multi-threaded con distintos resultados a nivel de visualización y de costes. Así, este trabajo también se ha ocupado de diseñar, implementar y exponer las modificaciones necesarias para transformar la versión secuencial en implementaciones paralelizadas, las cuales

utilizan OpenMP como tecnología de gestión y sincronización de múltiples hilos de ejecución. Los dos sistemas optimizados y la versión en single-threaded se han sometido a estudio desde el punto de vista de la calidad de la simulación y de la eficiencia para poder compararlos entre sí y con MSXPBD.

Al término de este desarrollo, se ha concluido que los objetivos de optimización se han satisfecho con diferentes grados de éxito en cada una de las implementaciones. Existe un salto sustancial de mejoría del rendimiento desde la versión original en Python a la versión secuencial en C++ compilado. Todavía más acentuado es ese salto respecto a las versiones multi-threaded, tanto con Jacobi como con el Método de Grafos.

En conclusión, este proyecto propone dos vías de optimización del modelo MSXPBD basadas en multi-threading con C++, cuyos resultados son satisfactorios y muy prometedores. Si bien se han cumplido con los objetivos marcados al inicio, los sistemas implementados tienen limitaciones que se explican en esta memoria y que nos ayudan a definir líneas de trabajo futuro. En el documento que presentamos aquí se recogen las dificultades encontradas durante la investigación así como las soluciones propuestas con la intención de marcar la dirección de esos desarrollos futuros.

Capítulo 1

Introducción

Los efectos visuales generados por ordenador constituyen una parte fundamental en la producción de contenido para cine, televisión, series e incluso publicidad. Entornos, ciudades, edificios, multitudes, efectos climáticos, vehículos o cualquier otro objeto de la realidad es susceptible de ser reproducido con tecnología virtual. Actualmente, el nivel de detalle que se puede conseguir es tan alto que seríamos prácticamente incapaces de diferenciar a través de una pantalla qué objetos son reales y qué objetos fueron añadidos mediante ordenador. De hecho, en la mayoría de casos, erraríamos. No obstante, la exigencia de mayor realismo sigue creciendo al mismo ritmo que la experiencia y la opinión crítica del público consumidor.

Por eso, la industria de los Efectos Visuales (VFX, del inglés *Visual Effects*) está en continuo esfuerzo por mejorar la calidad de su producto final y conseguir mayor credibilidad. Esta tendencia va íntimamente ligada al aumento de las prestaciones de los sistemas informáticos, los cuales pueden procesar mayor cantidad de datos y de forma más rápida. En este contexto de progreso en la generación de imágenes por ordenador (CGI, del inglés *Computer Generated Imagery*), en los últimos años ha incrementado el interés por mejorar en el ámbito de los personajes digitales y sus dinámicas, tanto de personas y animales como de cualquier criatura fantástica que podamos imaginar.

En el terreno de los gráficos, la simulación de dinámicas de personajes se ha llevado a cabo mediante un amplio abanico de soluciones, las cuales tratan de acercarse lo máximo posible a la anatomía de los seres vivos y su comportamiento. En el caso de los humanos, animales y otras criaturas, por ejemplo, estamos hablando de simular los músculos, la capa que los recubre llamada fascia, la grasa y finalmente la piel. Para ello, las técnicas que se aplican van desde el uso de deformadores lineales sobre los modelados geométricos hasta la aplicación de métodos de simulación sofisticados como el de Elementos Finitos. Lo cierto es que no existe un estándar consolidado en VFX a causa del alto coste computacional de dichos modelos, su complejidad y difícil parametrización o la falta de libertad y control artístico que tan importante es en este

sector.

Recientemente, ha surgido una línea de investigación nueva a partir del modelo *Extended Position Based Dynamics* (XPBD) de Macklin et al. [12] que originalmente estaba concebido para simulación de objetos deformables y elásticos. Es en el trabajo de Romeo et al. titulado *Muscle Simulation with Extended Position Based Dynamics* [16] (MSXPBD) donde se presenta, por primera vez, un sistema de simulación de músculos basado en XPBD. Este novedoso enfoque es descrito por los propios autores como el equilibrio entre la alta calidad en los resultados y la satisfacción de los requisitos que vienen impuestos por los estudios de VFX desde un entorno de producción real.

El sistema de Romeo et al. es presentado como un plug-in para Autodesk Maya implementando en Python *single-threaded*. Si bien los autores muestran satisfacción con los resultados, marcan como uno de los puntos de trabajo futuro la optimización de la implementación para conseguir tiempos de simulación más rápidos. Así, este proyecto surge como la continuación de MSXPBD para mejorar su eficiencia mediante la aplicación de técnicas de optimización y paralelización sobre una versión del plug-in reimplementada completamente en C++. En otras palabras, este proyecto combina la rama de la computación de altas prestaciones con la rama de la simulación de personajes para Efectos Visuales. La premisa fundamental será, por tanto, beneficiarnos del potencial de un lenguaje compilado como es C++ y de las metodologías de desarrollo de altas prestaciones para acelerar el cómputo de la simulación.

Con el fin de ofrecer al lector una vista preliminar de las etapas que tendrán lugar durante el desarrollo del proyecto, proponemos la siguiente enumeración.

- **Introducción y Planificación del Proyecto**.
  En esta primera fase, se recopila la información necesaria para comprender el Estado del Arte, definir el proyecto y su alcance y establecer los objetivos de forma clara. Además, se presenta una explicación de las tareas que deben desarrollarse para la consecución de los objetivos, su duración en el tiempo y los ítems o entregables que deben obtenerse al final de cada una de ellas. También se incluye aquí el análisis de costes del proyecto. Por último, se justifica la metodología y la tecnología que se aplicarán durante el desarrollo del mismo.

- **Implementación secuencial**.
  Como se ha sugerido con anterioridad, este proyecto puede entenderse como una continuación del desarrollo iniciado por Romeo et al. Tanto es así, que el objetivo de esta segunda etapa es rediseñar e implementar su mismo sistema pero con una estructura mejorada y en lenguaje C++ sin paralelización. Al final de esta tarea, debemos disponer de un plug-in compilado que pueda utilizarse en Maya. Finalizada la implementación,

será necesario realizar un conjunto de test que certifiquen el correcto funcionamiento del sistema, comparando con los resultados que se obtienen en el trabajo previo.

- **Análisis y Diseño de la paralelización**.
  Esta etapa consiste en un estudio exhaustivo del código obtenido en la fase anterior para la detección de los bloques de ejecución que deben paralelizarse. El objetivo final de esta tarea es diseñar los cambios a implementar para traducir el código a su versión optimizada.

- **Implementación de la paralelización**.
  Una vez se ha diseñado la algorítmica de la paralelización, se lleva a cabo la implementación en sí. El resultado a obtener aquí es el plug-in final compilado para Maya. De nuevo, tras una etapa de implementación, es preciso someter a pruebas nuestro sistema para cerciorarnos de su usabilidad.

- **Recopilación y Análisis de resultados**.
  En esta fase se pretende recopilar datos relacionados con los tiempos de cómputo de las dos soluciones desarrolladas y realizar un profundo estudio comparativo. El objetivo que se persigue es el de ser capaces de demostrar que la optimización llevada a cabo tiene un impacto positivo sobre las prestaciones del sistema.

- **Documentación**.
  Finalmente, se redactará una memoria del proyecto donde se expliquen con detalle las premisas y el desarrollo de todas las tareas realizadas. Además, se generarán vídeos demostrativos con los resultados de las simulaciones.

Capítulo 2

Estado del Arte

En este capítulo vamos a estudiar la literatura que está relacionada con los pilares sobre los que se sostiene este proyecto, esto es, simulación de músculos para Efectos Visuales y el modelo *Position Based Dynamics* (PBD). Explicaremos el artículo de Romeo et al. sobre simulación de músculos ya que es el verdadero punto de partida del presente trabajo, pero antes, será necesario que ofrezcamos una visión general del modelo original PBD de Müller et al. y de su posterior extensión por Macklin et al., puesto que son la base algorítmica y metodológica de esta nueva línea de investigación en cuestión de simulación de músculos.

2.1. Sistemas de simulación de músculos

En la literatura recopilada sobre modelado y simulación de músculos hemos encontrado aproximaciones con diferentes enfoques. Debemos comenzar destacando el estudio realizado por Lee et al. y publicado en 2012 [10] donde se exponen diversos modelos mecánicos, modelos geométricos y otros basados en física.

Desde esos primeros estudios quedaron patentes varios aspectos a comentar. Por un lado, la complejidad de los músculos en cuanto a estructura, ya que pueden presentar múltiples tendones y varias zonas de inserción en los huesos del esqueleto. Por otro lado, la diversidad en cuanto a topología y forma, puesto que sólo en el cuerpo humano existen músculos planares, fusiformes, penniformes, multipenniformes y circulares, entre otros. Además, otro aspecto crucial es la dirección de las fibras musculares, las cuales juegan un papel protagonista durante la activación de los músculos e influyen directamente en su deformación.

Gasser et al. [6] demostraron con modelos mecánicos que las fuerzas internas que se producen en las fibras son el origen de las contracciones musculares. Asimismo, en estudios posteriores por Zajac y Michael [25] se vio que también influye la direccionalidad de las fibras en el comportamiento global.

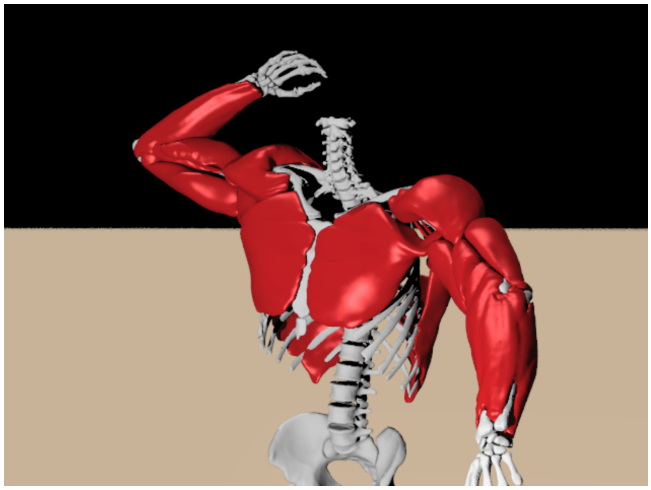

Figura 2.1: Simulación quasi-estática de los músculos del torso con el modelo de Teran et al. basado en FEM. Fuente: *Robust Quasistatic Finite Elements and Flesh Simulation* [21].

Desde el punto de vista de la simulación por ordenador, los modelos basados en geometría no han sido capaces de reproducir con realismo las dinámicas de los músculos debido a toda la complejidad comentada. Por ese motivo, recientemente han surgido otras aproximaciones basadas en modelos mucho más sofisticados.

Se han propuesto modelos mixtos que combinan deformación procedural con simulación física, como ocurre en los trabajos de Comer et al. [4] y Milne et al. [14]. Incluso existen casos en los que se aplican resultados directamente extraídos de escáneres por resonancia magnética sobre animales reales y también personas [9].

No obstante, las técnicas más habituales y extendidas en la actualidad hacen uso de modelos numéricos basados en el Método de Elementos Finitos (FEM), como es el caso de Teran et al. [21] (Figura 2.1) y Jacobs et al. [7], y en el Método de Volúmenes Finitos (FVM), como el trabajo de Teran et al. en [20].

Las técnicas con elementos finitos han sido adoptadas por los investigadores y las compañías de VFX para desarrollar sus propios sistemas de simulación de músculos. Un ejemplo concreto lo vemos en el sistema Tissue [24] desarrollado por la compañía Weta (Figura 2.2) o en el solver de simulación de músculos implementado de forma nativa en el software Houdini [18], que es uno de los estándares en la industria.

El principal motivo de esta tendencia hacia FEM está en la alta calidad de los resultados. Sin embargo, sus desventajas son la complejidad de la implementación, la dificultad de configuración de los personajes antes de comenzar a simular, la falta de controles intuitivos similares a los que un artista 3D está acostumbrado a manejar y la conservación del volumen incluso durante la contracción del músculo. En la Tabla 2.1 se resumen estos y otros puntos a favor y en contra de los modelos FEM para simulación de músculos.

Figura 2.2: Ejemplos de simulaciones con Tissue de Weta para el largometraje de Avatar. Fuente: *www.3dart.it*.

Para afrontar tales limitaciones, se suelen implementar herramientas que agilizan la configuración de un personaje como ocurre con Ziva VFX [23] (Figura 2.3) o que permiten transferir esas configuraciones entre personajes, como hacen Seo et al. [17]. Otras soluciones consisten en proporcionar a los artistas automatismos de modelado que les permitan generar geometrías ya preparadas para las simulaciones [22].

Es importante aclarar que el problema sigue sin converger a un modelo único y que las vías de investigación siguen estando abiertas. Tanto es así, que uno de los principales rasgos de la contracción muscular como es la ganancia de volumen por el aumento del flujo sanguíneo

Ventajas	**Desventajas**
Alta calidad de resultados	Alto coste computacional
Interacción coherente entre músculos	Implementación muy compleja
Contracción de fibras realista	Difícil configuración
Estabilidad de la simulación	Parámetros poco intuitivos
	Conservación de volumen
	Reduce control y libertad artística

Tabla 2.1: Ventajas y Desventajas de los modelos de simulación basados en FEM.

Figura 2.3: Ejemplo de cuadrúpedo con el modelo de músculos de Ziva VFX. Fuente: *zivadynamics.com /ziva-vfx* [23].

durante la activación, es un punto difícilmente alcanzable por los métodos con elementos finitos ya que éstos están diseñados para conservar el volumen por completo e incondicionalmente.

Por todo ello, otros enfoques drásticamente diferentes deberán asumir estos retos. De hecho, el método *Position Based Dynamics*, que exponemos en la sección 2.2 está haciendo su entrada en la industria de la animación por ordenador [19].

2.2. Position Based Dynamics

El método *Position Based Dynamics* (PBD) fue propuesto por Müller et al. en 2007 [15]. Se trata de un modelo de simulación basado en sistemas de partículas capaz de reproducir comportamientos de objetos rígidos, deformables, elásticos y fluidos (ver Figura 2.4). La dinámica de partículas se calcula a partir de la aplicación de restricciones sobre sus posiciones. Sus principales características son la rapidez, controlabilidad, la estabilidad y la consecuente alta aplicabilidad en entornos interactivos (ver Tabla 2.2 para más detalles sobre las ventajas y desventajas de PBD enfocado a simulación de músculos).

Las restricciones son condiciones que afectan a la posición de las partículas y que deben satisfacerse durante la simulación. Si a lo largo del tiempo, una fuerza externa o una colisión desplaza a las partículas de su estado de equilibrio, el método actúa para recuperar lo máximo posible ese equilibrio, intentando que todas las restricciones se cumplan.

Todo objeto viene representado por N vértices y M restricciones. Cada vértice $i \in [1, ..., N]$ tiene masa m_i, posición $\mathbf{r}_i$ y velocidad $\mathbf{v}_i$. Cada restricción $j \in [1, ..., M]$ tiene: una cardinalidad

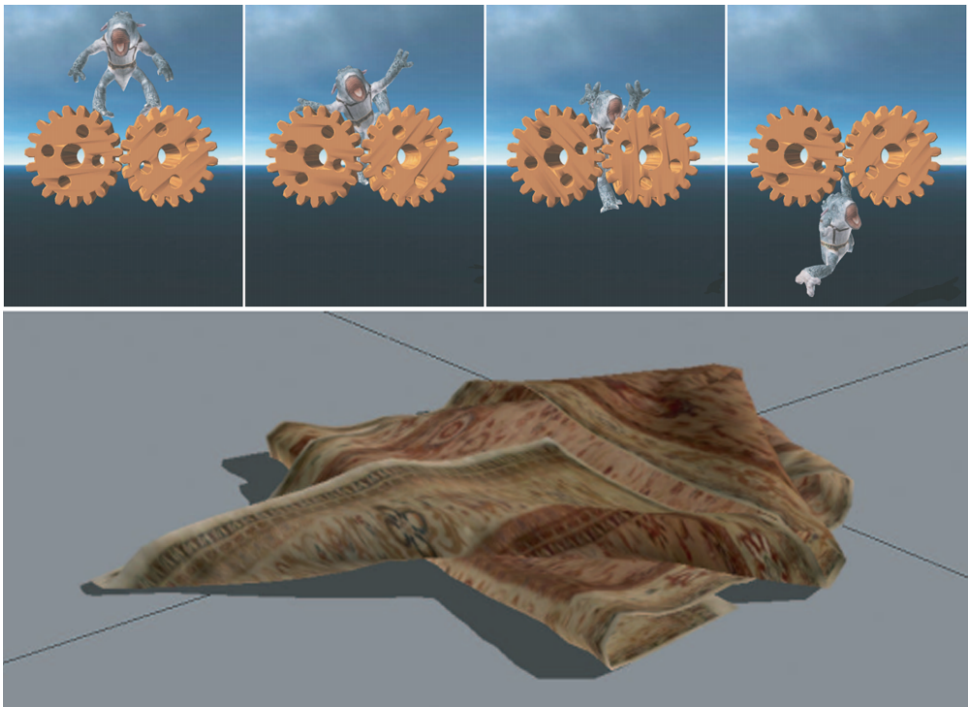

Figura 2.4: Simulaciones de objetos deformables con PBD: volúmenes cerrados (arriba) y tejidos (abajo). Fuente: *Position Based Dynamics* [15].

n_j como número de partículas implicadas; una función de estado $C_j : \mathbb{R}^{3n_j} \rightarrow \mathbb{R}$ que evalúa el cumplimiento o no de la restricción; un conjunto de índices que hacen referencia a los vértices $i_1, ..., i_{n_j}$ (donde $i_k \in [1, ..., N]$); y un parámetro de rigidez $k_j \in [0, ..., 1]$ que indica la resistencia de la restricción. Además, las restricciones pueden ser de *igualdad* o *desigualdad*. Toda restricción j de igualdad se satisface cuando su función de evaluación $C_j = 0$; mientras que si es de desigualdad se satisface cuando $C_j \geq 0$.

Una de las ventajas que tiene el método es que las restricciones se pueden utilizar con distintas finalidades. Dependiendo de ellas, la función de evaluación tendrá una forma u otra.

Ventajas	**Desventajas**
Buena calidad de resultados	Modelo no basado en física
Interacción coherente entre músculos	Escasa literatura relacionada
Contracción de fibras realista	
Estabilidad de la simulación	
Reducido coste computacional	
Configuración sencilla e intuitiva	
Aplicable en entornos interactivos	
Control y libertad artística	

Tabla 2.2: Ventajas y Desventajas de los modelos de simulación basados en PBD.

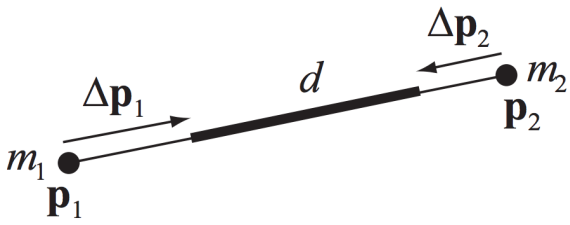

Figura 2.5: Proyección de la restricción de distancia. Las correcciones $\Delta\mathbf{p}_i$ se orientan para hacer cumplir la distancia d y son pesadas teniendo en cuenta la inversa de la masa de cada partícula ($w_i = 1/m_i$). Fuente: *Position Based Dynamics* [15].

Las hay para limitar la distancia entre partículas (restricción de distancia) o los pliegues en la superficie del objeto (restricción de *bending*), para garantizar la conservación del volumen (restricción de volumen) o también computar colisiones con otros objetos y con la propia malla. De hecho, existen otras publicaciones que proponen soluciones alternativas a algún tipo de restricción concreto, por ejemplo, la alternativa a la restricción de *bending* para curvaturas propuesta por Kelager et al. en [8].

Para comprender mejor el concepto de restricción, vamos a exponer el tipo más simple basado en la distancia entre dos partículas cuyas posiciones son $\mathbf{p}_1$ y $\mathbf{p}_2$. Se trata de una restricción de igualdad con cardinalidad 2 e índices $[1, 2]$ cuya función de evaluación es $C(\mathbf{p}_1, \mathbf{p}_2) = |\mathbf{p}_1 - \mathbf{p}_2| - d$ siendo d la distancia entre las partículas en el estado de equilibrio. En cada iteración de la simulación, se evalúa $C(\mathbf{p}_1, \mathbf{p}_2)$ y si el resultado es distinto de cero, se calculan los desplazamientos $\Delta\mathbf{p}$ de cada partícula para hacer cumplir la restricción (ver Figura 2.5). Los desplazamientos tienen lugar en la dirección del gradiente $\nabla_{\mathbf{p}} C$ para que dado un conjunto de posiciones de partículas $\mathbf{p}$ se cumpla que

$$\mathbf{C}(\mathbf{p} + \Delta\mathbf{p}) \approx C(\mathbf{p}) + \nabla_{\mathbf{p}} C(\mathbf{p}) \cdot \Delta\mathbf{p} = 0. \tag{2.1}$$

Para forzar que $\Delta\mathbf{p}$ tenga lugar en la dirección del gradiente $\nabla_{\mathbf{p}} C$, se toma un escalar λ tal que

$$\Delta\mathbf{p} = \lambda \nabla_{\mathbf{p}} C(\mathbf{p}). \tag{2.2}$$

La expresión general para calcular todos los $\Delta\mathbf{p}$ se obtiene sustituyendo la ecuación 2.2 en 2.1, despejando λ y volviéndolo a introducir en 2.2:

$$\Delta\mathbf{p} = -\frac{C(\mathbf{p})}{|\nabla_{\mathbf{p}} C(\mathbf{p})|^2} \nabla_{\mathbf{p}} C(\mathbf{p}). \tag{2.3}$$

Para la corrección de un vértice individual $\mathbf{p}_i$ se atiende a la ecuación siguiente:

$$\Delta\mathbf{p}_i = -s\nabla_{\mathbf{p}_i}C(\mathbf{p}_1, ..., \mathbf{p}_n), \tag{2.4}$$

donde s es un factor de escalado común para todos los vértices de la restricción que se obtiene según la ecuación 2.5. El término w_j es la inversa de la masa de la partícula j.

$$s = \frac{C(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_j w_j \left|\nabla_{\mathbf{p}_j}C(\mathbf{p}_1, ..., \mathbf{p}_n)\right|^2} \tag{2.5}$$

El cálculo del gradiente $\nabla_{\mathbf{p}}C$ depende del tipo de la restricción. En el caso de la distancia entre $\mathbf{p}_1$ y $\mathbf{p}_2$ que estamos comentando tenemos que $\nabla_{\mathbf{p}_1}C(\mathbf{p}_1, \mathbf{p}_2) = \mathbf{n}$ y $\nabla_{\mathbf{p}_2}C(\mathbf{p}_1, \mathbf{p}_2) = -\mathbf{n}$, donde:

$$\mathbf{n} = \frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|}. \tag{2.6}$$

Una vez se han obtenido las correcciones $\Delta\mathbf{p}$, se aplica el coeficiente de rigidez para conseguir comportamientos más blandos o más duros. Esto se realiza multiplicando dichas correcciones por el parámetro k', que es una transformación de k tal que $k' = 1 - (1 - k)^{1/n_s}$.

Un punto importante del método PBD es la estructura del bucle de simulación. En primer lugar, se integran las fuerzas externas para actualizar las velocidades $\mathbf{v}_i$ de todos los vértices y calcular después las posiciones estimadas $\mathbf{p}_i$ (no las posiciones reales $\mathbf{x}_i$). Sobre estas predicciones se generan las restricciones de colisión que puedan surgir en tiempo de ejecución. Y seguidamente, se lleva a cabo el proceso de evaluación y cómputo de las restricciones, proceso que Müller et al. denominan *proyección de restricciones*.

La proyección realiza el cálculo de los $\Delta\mathbf{p}$ y los suma a las predicciones $\mathbf{p}$ de manera secuencial recorriendo todas las restricciones. Además, las correcciones no se aplican directamente en un único paso, sino que lo hacen parcialmente en un bucle interno para que las últimas no inhiban a las primeras. El algoritmo aplica el paradigma iterativo de Gauss-Seidel.

Cuando el bucle de iteraciones del solver que proyecta las restricciones ha terminado, se obtiene la velocidad real de cada vértice como $\mathbf{v}_i = \frac{\mathbf{p}_i - \mathbf{x}_i}{\Delta t}$. Finalmente, se actualiza la posición real con el valor de las estimaciones: $\mathbf{x}_i = \mathbf{p}_i$.

Position Based Dynamics se ha popularizado y consolidado en el campo de los gráficos como un método de simulación para elásticos y deformables porque ofrece rapidez, estabilidad y control. Esto ha motivado su estudio y ampliación en publicaciones posteriores. Es el caso de [3] donde se combina PBD y *Shape Matching* para simulación de sólidos, o en [13] donde se desarrolla un framework para efectos en tiempo real basado en PBD. Además, ha inspirado el desarrollo de otros métodos entre los que cabe destacar *Position Based Fluids* de Macklin y Müller [11] para simulación de fluidos que se fundamenta en el cómputo de restricciones de

Métrodo	Principios básicos
Jacobi	Todas las restricciones se calculan en paralelo Los $\Delta\mathbf{p}_i$ no modifican $\mathbf{p}_i$ directamente, sino que se acumulan El sumatorio de los $\Delta\mathbf{p}_i$ se aplica al final de cada iteración El sumatorio de los $\Delta\mathbf{p}_i$ se aplica promediado por el número de restricciones
Grafos	Las restricciones se dividen en grupos de ejecución Las restricciones de un mismo grupo no comparten partículas Los grupos se computan secuencialmente Las restricciones de cada grupo se computan en paralelo
Híbrido	Agrupar restricciones en k grupos Los primeros $k-1$ grupos están balanceados y aplican el método de Grafos El último grupo aplica el método Jacobi

Tabla 2.3: Ideas clave de los métodos de paralelización de PBD existentes en la literatura.

posición para mantener constante la densidad en el líquido.

2.2.1. Métodos de paralelización

La optimización del modelo Position Based Dynamics se ha convertido en un tema de interés para los investigadores en simulación de gráficos, sobretodo, con el fin de integrarlo en aplicaciones interactivas. La idea básica a tener en cuenta cuando se habla de paralelización de PBD es que el solver se fundamenta en un método Gauss-Seidel, esto es, las restricciones del sistema se procesan una tras otra actualizando directamente las posiciones de las partículas afectadas. Si trasladamos esto a una implementación en paralelo con múltiples hilos donde varias restricciones pueden afectar a una misma partícula, el comportamiento del sistema se vuelve impredecible al estar actualizando su posición desde diferentes restricciones simultáneamente.

Para resolver esta problemática, en la literatura se han encontrado diversos enfoques que están recogidos en el trabajo de Bender et al. de 2015 titulado *Position-Based Simulation Methods in Computer Graphics* [2]. En los apartados siguientes se exponen los principios de cada uno de ellos, pero véase la Tabla 2.3 a modo de resumen.

2.2.1.1. Método de Jacobi

La implementación de PBD con un enfoque jacobiano permite procesar todas las restricciones en paralelo. El modo de hacerlo es calculando y acumulando la corrección de cada restricción actualizando la posición de las partículas sólo al final de cada iteración del solver. En paralelo, todas las restricciones calculan el cambio de posición de sus partículas pero no lo aplican directamente. Las partículas son las encargadas de ir almacenando estas correcciones. Cuando ha terminado este bloque paralelizado, cada partícula actualiza su posición dividiendo el acu-

mulado por el número total de restricciones que le afectan. Siendo $\Delta\mathbf{p}_i$ el cambio de posición acumulado para la partícula i; y n_i el número total de restricciones que actúan sobre ella; se obtiene el cambio de posición final de una iteración $\Delta\tilde{\mathbf{p}}_i$ como:

$$\Delta\tilde{\mathbf{p}}_i = \frac{1}{n_i}\Delta\mathbf{p}_i. \tag{2.7}$$

Según Macklin et al. [13], las limitaciones de este método son que no garantiza completamente la conservación del momento y que en ocasiones, es necesario incrementar el número de restricciones porque el resultado muestra una excesiva relajación del sistema (denominada por los autores *successive over-relaxation*, *SOR*). No obstante, gracias al promediado de las correcciones el sistema converge siempre a una solución. Para evitar la sobre-relajación, se introduce un nuevo parámetro en la ecuación:

$$\Delta\tilde{\mathbf{p}}_i = \frac{w}{n_i}\Delta\mathbf{p}_i \tag{2.8}$$

donde w se propone como controlador de sobre-relajación sucesiva y debe cumplir $1 \leq w \leq 2$.

2.2.1.2. Método de coloreado de grafos

El coloreado de grafos (del inglés *Graph-Coloring Methods*) consiste en separar el total de las restricciones en grupos que actúan sobre conjuntos de partículas diferentes. En otras palabras, una partícula no puede estar compartida por dos restricciones dentro de un mismo grupo. Si esta condición se cumple, todas las restricciones de un grupo pueden procesarse en paralelo con el método *Gauss-Seidel* tradicional sin riesgos de conflicto o inestabilidades. Una vez completada la ejecución de un grupo y actualizadas las posiciones de las partículas se continúa con el siguiente grupo de restricciones, repitiendo este ciclo hasta que todas las restricciones han sido procesadas.

Este método funciona bien cuando el grado de compartición de partículas entre restricciones es bajo. En caso contrario, la distribución en grupos termina siendo pobre y no balanceada, con grupos de gran tamaño al principio de la ejecución y grupos muy pequeños al final. Esta descompensación de tamaños reduce el impacto de mejora por la paralelización, especialmente al procesar grupos con pocas restricciones.

2.2.1.3. Método Híbrido

Fratarcangeli et al. [5] propusieron un método híbrido para aprovechar las ventajas del *graph-coloring* y del método con Jacobi. Su aproximación consiste en utilizar un grafo coloreado que divide el conjunto de restricciones en k grupos de manera que los primeros $k-1$ grupos están

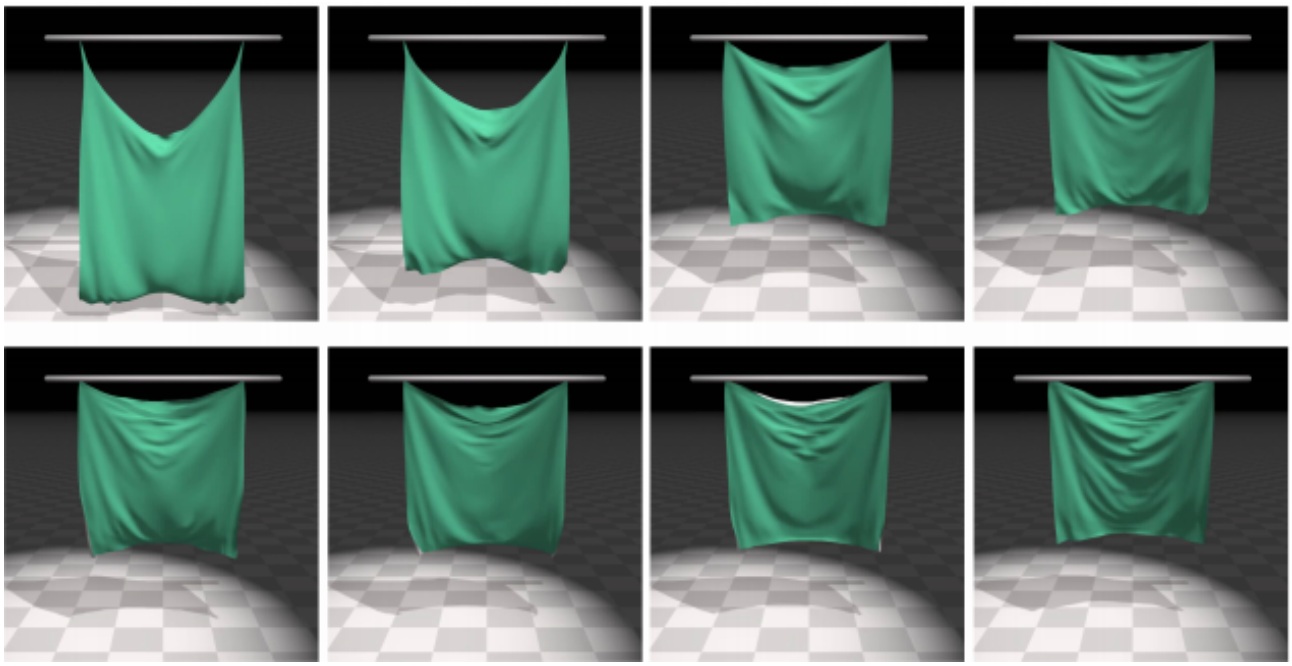

Figura 2.6: Comparativa entre PBD (arriba) y XPBD (abajo) de simulaciones con 20, 40, 80, y 160 iteraciones de izquierda a derecha. Fuente: *XPBD: Position-Based Simulation of Compliant Constrained Dynamics* [12].

bien balanceados. Así, primero se computan esos $k-1$ grupos con Gauss-Seidel, y finalmente, se procesa el grupo restante con el método Jacobi.

2.2.2. Extended Position Based Dynamics

El método *Extended Position Based Dynamics* (XPBD) es, como su nombre indica, una extensión de PBD propuesta por Macklin et al. en 2016 [12] que solventa las limitaciones del modelo original. Estas limitaciones tienen que ver con la dependencia de la rigidez de los objetos simulados en función del tamaño de paso de integración y del número de iteraciones. Dicho en otras palabras, aunque dejemos el parámetro k de PBD (que a priori se entendía como el factor de rigidez) constante, si modificamos el paso de integración y/o las iteraciones, el sistema ofrece resultados diferentes que al final son interpretados como un cambio en la elasticidad de los objetos.

La ampliación que aporta XPBD al solver original soluciona este problema y permite obtener comportamientos similares con independencia del número de iteraciones (ver Figura 2.6) y del paso de integración. Sin entrar en mucho detalle, la modificación se basa en la introducción de unos factores llamados multiplicadores de Lagrange λ. Cada restricción C_j del sistema depende

de uno de estos multiplicadores, los cuales se definen como

$$\lambda_j = -\tilde{\alpha}^{-1} C_j(\mathbf{p}), \tag{2.9}$$

donde

$$\tilde{\alpha} = \frac{1}{k\Delta t^2}. \tag{2.10}$$

Introduciendo el tamaño de paso de integración en la ecuación anterior es la manera con la que los autores logran aplicar el concepto de energía potencial elástica con mayor precisión que la conseguida con PBD.

El multiplicador λ tiene su influencia en la cantidad de corrección que se aplica sobre las posiciones $\mathbf{p}$ en una sola iteración en la restricción C_j, es decir, sobre el factor s visto en la ecuación 2.5 de la sección 2.2. Los autores de XPBD redefinen el original s de la restricción C_j como $\Delta\lambda_j$, tal que:

$$\Delta\lambda_j = \frac{C_j(\mathbf{p}_i) - \tilde{\alpha}_j \lambda_j}{w_j \nabla C_j \nabla C_j^T + \tilde{\alpha}_j} \tag{2.11}$$

Es importante señalar que cada fotograma de la simulación se inicia con el parámetro $\lambda_j = 0$, y se va actualizando en cada iteración del solver tal que $\lambda_j = \lambda_j + \Delta\lambda_j$. Al final de cada iteración, se obtiene la corrección en la posición de un vértice aplicando el resultado de la ecuación 2.11 sobre la s de la ecuación 2.4.

2.3. Simulación de músculos con Extended Position Based Dynamics

Conocidas las limitaciones de los modelos de simulación de músculos presentes en la literatura, en 2018 Romeo et al. proponen el primer acercamiento a la simulación de músculos con XPBD en su trabajo *Muscle Simulation with Extended Position Based Dynamics* (MSXPBD) [16]. Los autores hacen valer las ventajas de flexibilidad, estabilidad y controlabilidad de XPBD para diseñar un sistema que es capaz de reproducir comportamientos realistas en las dinámicas de los músculos y superar algunos problemas que otros modelos no resuelven, como la ganancia de volumen durante la activación.

El trabajo que presentan introduce algunas variaciones al modelo original para adaptarlo a las necesidades dinámicas requeridas por un sistema de músculos. Por una parte, introducen un algoritmo de generación de estructura interna que les permite reforzar el comportamiento rígido de los músculos. La estructura interna que generan consiste en un una especie de médula

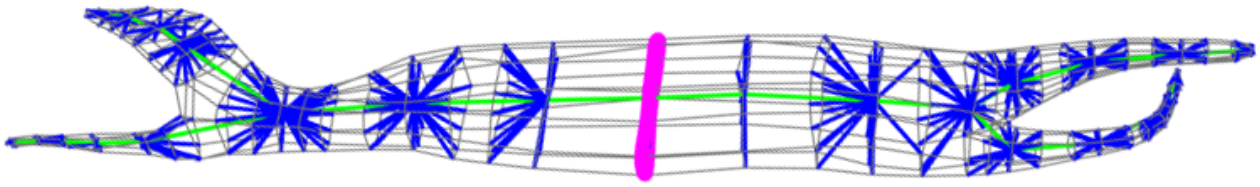

Figura 2.7: Estructura interna del bíceps humano: en verde la médula interna; en azul las conexiones entre los puntos internos de la médula y los vértices de la superficie; en violeta la selección de vértices raíz. Fuente: *Muscle Simulation with Extended Position Based Dynamics* [16].

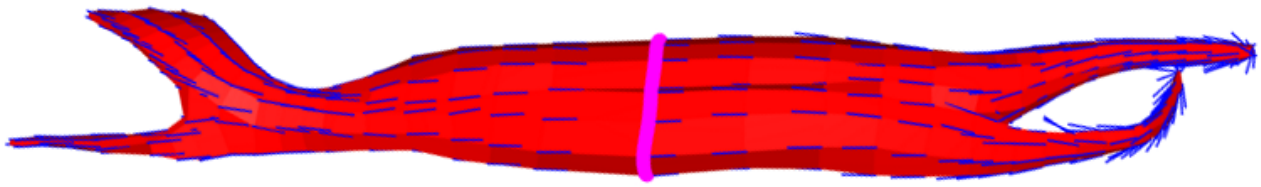

Figura 2.8: Orientación de las fibras en el bíceps humano: en violeta la selección de vértices raíz para la estructura interna; en azul los vectores de fibra resultantes por cada vértice. Fuente: *Muscle Simulation with Extended Position Based Dynamics* [16].

que se extiende por el interior de toda la geometría cuyos puntos se conectan con los vértices de la superficie (Figura 2.7). Dicha estructura se construye de manera automatizada a partir de una selección de vértices raíz.

Asimismo, utilizan la propia estructura interna para calcular la dirección de las fibras. Demuestran que la estructura interna que obtienen es correcta con cualquier forma muscular, y de este modo, la rama interna resultante es también coherente con la dirección de fibras que cabría esperar en un músculo real (ver Figura 2.8). Gracias a ello, solventan el problema de la variedad topológica y estructural de los músculos, y el condicionante de la direccionalidad de fibras.

Por otra parte, aprovechan la restricción de volumen definida por Müller et al. en el artículo original de PBD [15] y su factor de sobre-presión (*overpressure*) para relacionarlo con el nivel de activación del músculo y así conseguir reproducir el aumento de tamaño durante la contracción tal y como se evidencia en la Figura 2.9. Éste es uno de los aspectos que los sistemas con FEM no son capaces de afrontar, mientras que la flexibilidad de XPBD sí lo permite.

Además, para obtener una activación en los músculos acorde al movimiento del personaje, desarrollan un sistema de sensores basado en la rotación, velocidad y aceleración de los huesos y las articulaciones del esqueleto (ver Figura 2.10). Con ello, sólo con la animática del personaje obtienen automáticamente los valores de activación de cada músculo.

Otro de los aspectos relevantes de MSXPBD es el modo en que se simula la interacción entre

Figura 2.9: Flexión del codo: a) simulación con XPBD original; b) simulación con MSXPBD incluyendo activación de los músculos; c) simulación con MSXPBD incluyendo activación y ganancia de volumen. Fuente: *Muscle Simulation with Extended Position Based Dynamics* [16].

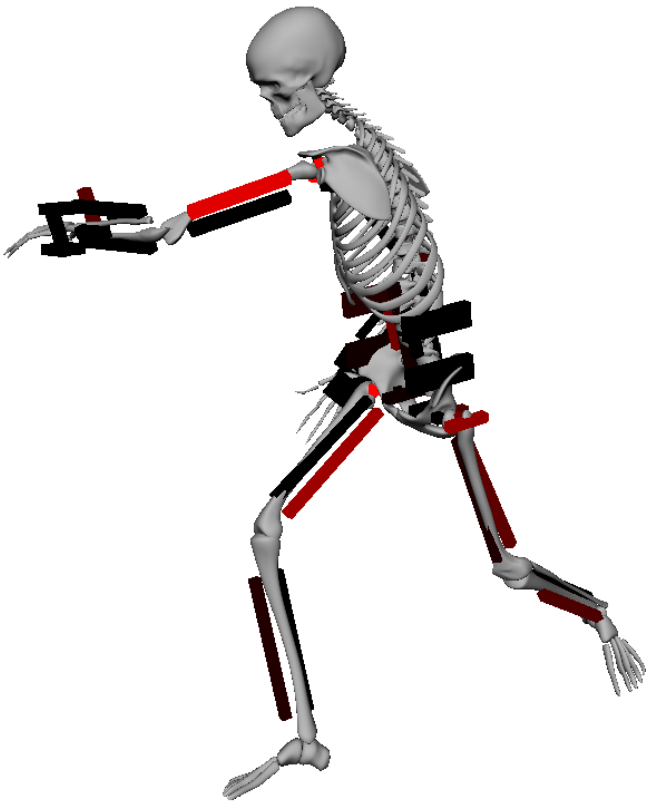

Figura 2.10: Activación de los sensores durante un movimiento de lucha de un humano. Fuente: *Muscle Simulation with Extended Position Based Dynamics* [16].

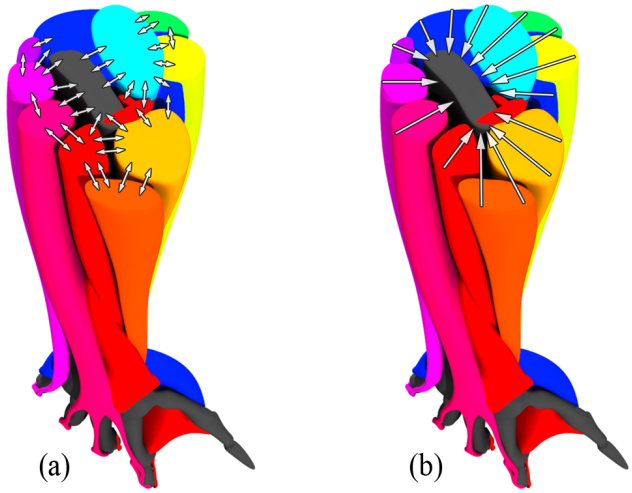

Figura 2.11: Conexiones que establecen las restricciones externas para la interacción entre músculos (a) y entre músculos y esqueleto (a, b). Fuente: *Muscle Simulation with Extended Position Based Dynamics* [16].

los músculos. Mediante la definición de restricciones externas a modo de conexiones músculo-músculo y músculo-hueso (Figura 2.11), se consigue reproducir la colisión entre músculos así como los efectos de deslizamiento o *sliding* típicos de la anatomía humana.

No obstante, la aportación principal de MSXPBD sobre el modelo original se refleja en una actualización de la restricción de distancia. El concepto fundamental es que la distancia de equilibrio entre dos vértices se modifica en función del nivel de activación del músculo y de la orientación de la arista que los une respecto a la dirección de las fibras.

Recordando la definición de la restricción de distancia en PBD, $C(\mathbf{p}_1, \mathbf{p}_2) = |\mathbf{p}_1 - \mathbf{p}_2| - d$, los autores redefinen d como σ tal que

$$\sigma = \left| \left((\mathbf{r}_1 - \mathbf{r}_2) \circ \left(\overrightarrow{\mathbf{1}} - \phi a \right) \right) \right|, \tag{2.12}$$

donde $\mathbf{r}_1$ y $\mathbf{r}_2$ son las posiciones en el instante incial de los dos vértices implicados, $a \in [0, 1]$ es el nivel de activación y ϕ es el vector positivo resultado del promedio de los vectores fibra de sendos vértices, es decir,

$$\phi = \text{abs} \left(\frac{\hat{\mathbf{f}}_1 + \hat{\mathbf{f}}_2}{|\hat{\mathbf{f}}_1 + \hat{\mathbf{f}}_2|} \right). \tag{2.13}$$

Con este planteamiento, lo que ocurre a nivel práctico cuando una arista se activa con un cierto valor de a por encima de cero es que la distancia de equilibrio se ve reducida proporcionalmente a la orientación de la arista respecto a la dirección de las fibras (ver Figura 2.12). Dicho de otro modo, una arista totalmente alineada con las fibras se contraería con una proporción

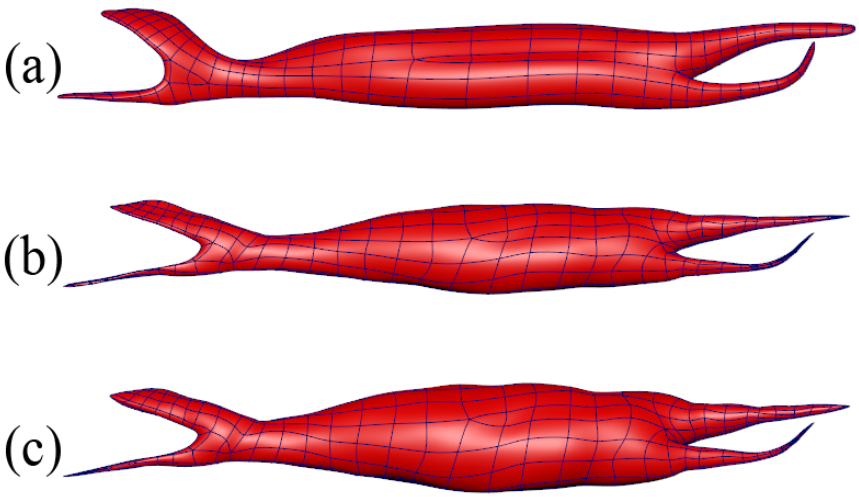

Figura 2.12: Comparativa de un bíceps humano: a) en estado de reposo; b) activado aplicando la restricción de fibra de MSXPBD; c) activado y con ganancia de volumen. Fuente: *Muscle Simulation with Extended Position Based Dynamics* [16].

igual a a.

Paralelamente a todo lo anterior, hay que señalar que un músculo en el instante inicial de la simulación puede no estar totalmente relajado, ya que esto depende del modelado de la geometría y de la pose inicial del personaje. Por eso, en MSXPBD se aplica el concepto de activación inicial a_0 para obtener, a partir de la longitud de arista inicial, la distancia en reposo que habría entre los dos vértices si realmente el músculo estuviera totalmente relajado. Para ello, se define ϵ para reemplazar al componente $\mathbf{r}_1 - \mathbf{r}_2$ de la ecuación 2.12.

$$\epsilon = \frac{\mathbf{r}_1 - \mathbf{r}_2}{\overrightarrow{\mathbf{1}} - \phi a_0} \tag{2.14}$$

De modo que, finalmente, la distancia d del algoritmo orignal de Müller et al. queda reemplazada por $\hat{\sigma}$ con la forma:

$$\hat{\sigma} = \mid \left(\epsilon \circ \left(\overrightarrow{\mathbf{1}} - \phi a \right) \right) \mid \tag{2.15}$$

En términos de bucle de simulación, MSXPBD tiene en cuenta lo importante que es la interacción entre músculos puesto que se conectan unos con otros a través de las restricciones externas. Por ello, se necesita computar los músculos progresivamente, esto es, por cada iteración del solver, se computa una única iteración de cada músculo. Así, cuando un músculo calcula su estado respecto a sus contiguos, tiene la información geométrica de ellos actualizada tras el cómputo de la iteración anterior.

Las conclusiones de Romeo et al. se traducen en que presentan un sistema novedoso, estable,

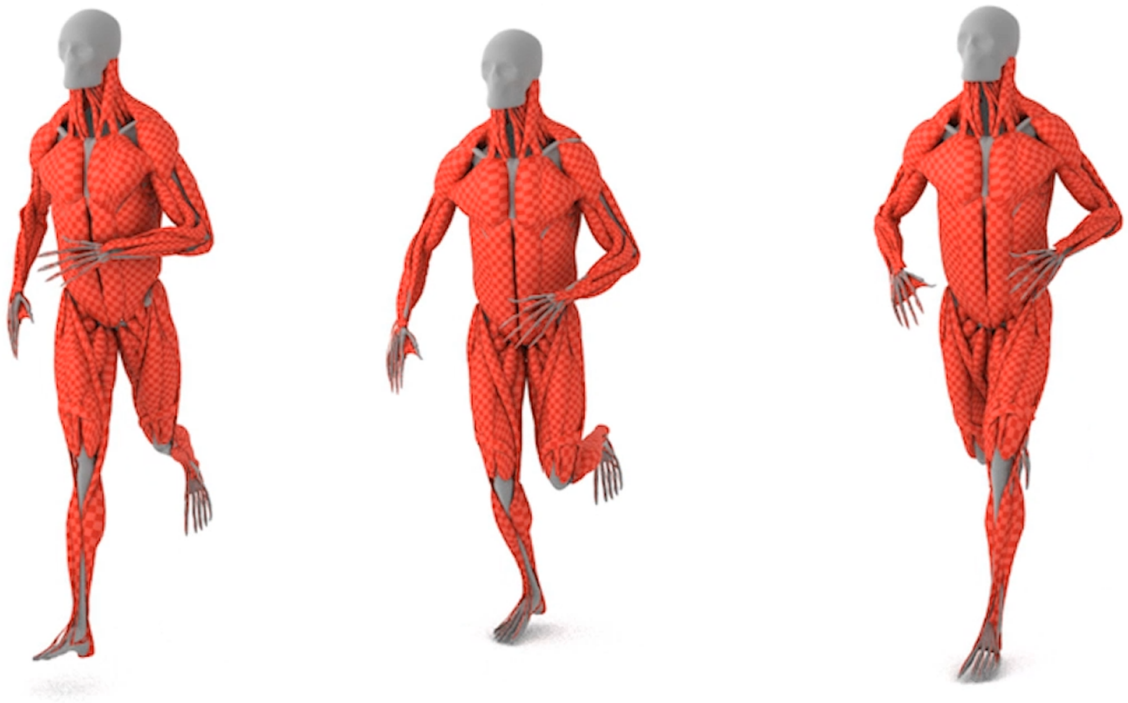

Figura 2.13: Simulación de un cuerpo humano completo corriendo. Fuente: *Muscle Simulation with Extended Position Based Dynamics* [16].

controlable y que ofrece resultados realistas incluso en escenarios complejos con gran cantidad de músculos (Figura 2.13). Además, añaden que los tiempos de cómputo, pese a que el sistema no está optimizado, son comparables a los tiempos que se manejan con otros sistemas basados en FEM consolidados en la industria de los VFX.

Capítulo 3

Motivación

Un sistema de simulación de músculos debe satisfacer los requisitos básicos de producir resultados de alta calidad, conseguir que los músculos interaccionen de manera realista entre ellos y que su activación y relajación sea creíble y coherente con el movimiento del personaje. Si además queremos que sea posible su utilización dentro de un estudio de Efectos Visuales, el sistema debe cumplir también otros requisitos que tienen que ver con las necesidades y limitaciones que existen en una producción audiovisual.

El primero de estos requisitos de producción es la rápida convergencia en la simulación. Es necesario que se puedan obtener resultados pronto para revisarlos, detectar los aspectos a mejorar, modificar la configuración del sistema para aplicar cambios y volver a simular. Iterando sobre este ciclo de trabajo, es posible llegar a la versión final con la mayor calidad y en el menor tiempo.

En segundo lugar, es deseable que la configuración del sistema sea lo más sencilla posible para que la integración del mismo dentro del flujo de trabajo de un estudio de VFX sea suave. De hecho, en una empresa de este sector participan diferentes departamentos que trabajan coordinadamente sobre los mismos planos de una película o una serie. Por tanto, la comunicación y el intercambio de material entre ellos debe fluir.

Otro requisito importante tiene que ver con la simplicidad de uso de los controles del sistema. Es decir, el sistema tiene que ser utilizado por artistas 3D que deben centrarse en crear los resultados dramáticos que se les exigen y no en lidiar con complejos parámetros. Dicho de otro modo, es importante que los controles del sistema sean intuitivos y que los cambios que se introduzcan generen soluciones predecibles.

Por último, pero no menos importante, un sistema de músculos para VFX debe ser flexible y ofrecer libertad artística para poder obtener los resultados creativos que los supervisores y directores buscan. Tal control artístico debe garantizarse incluso en simulaciones que no son completamente realistas o físicamente correctas.

Para ofrecer una vista resumida de los requerimientos que debe cumplir todo sistema de simulación de músculos para VFX, a continuación se listan los requisitos fundamentales y de producción.

- Resultados de muy alta calidad.
- Adecuada interacción entre los músculos.
- Coherente activación y relajación de los músculos.
- Estabilidad de la simulación.
- Rápida convergencia de la simulación.
- Fácil configuración del sistema.
- Controles y parámetros intuitivos.
- Flexibilidad y libertad artística.
- Fácil instalación y puesta en marcha del sistema en producción.
- Sencilla integración de posibles actualizaciones del sistema.

Tratando de cumplir estas premisas, las investigaciones que han predominado más recientemente se basan en métodos numéricos como Elementos Finitos (FEM) o Volúmenes Finitos (FVM). Si bien estas aproximaciones producen resultados de calidad y muy realistas, tienen importantes limitaciones a la hora de satisfacer los requisitos de producción. Por ejemplo, el alto coste computacional reduce su aplicabilidad ya que no se dispone de grandes márgenes de tiempo. Además, son métodos complejos y difíciles de configurar por parte de los artistas.

Por estas razones entre otras, comienzan a cobrar interés modelos con enfoques drásticamente diferentes que ofrecen soluciones rápidas de computar y fáciles de controlar. Estamos hablando, concretamente, del modelo *Position Based Dynamics* (PBD) propuesto por Müller et al. en 2007 [15] y su extensión en 2016 *Xpbd: Position-based simulation of compliant constrained dynamics* por Macklin et al. [12]. Tanto es así, que la aplicabilidad de XPBD para simulación de músculos ya ha sido demostrada por Romeo et al. en *Muscle Simulation with Extended Position Based Dynamics* (MSXPBD) [16] en 2018.

Los autores de MSXPBD hacen especial hincapié en que con su extensión de XPBD consiguen el equilibrio entre los objetivos a nivel de calidad de resultados y los objetivos a nivel de usabilidad en la industria de los VFX. Los autores detallan que el sistema se implementó como un plug-in en lenguaje Python totalmente integrado en Autodesk Maya y sin paralelizar.

Comentan que los costes de cómputo son equiparables a los de otros sistemas de simulación consolidados en la industria, los cuales están basados mayormente en FEM con implementaciones paralelizadas. Por ello, proponen una línea de investigación a futuro centrada en la optimización y paralelización de su sistema original para alcanzar tiempos de simulación aún más rápidos dadas las características propias del modelo XPBD que lo hacen altamente optimizable.

Dicha optimización comienza con la sustitución del lenguaje de programación. Python es un lenguaje interpretado que se caracteriza por su sencillez de la sintaxis, su corta curva de aprendizaje y su flexibilidad para utilizarse orientado a objetos o para *scripting*. No obstante, cuando se trata de aplicaciones complejas que son exigentes a nivel de cálculo, difícilmente puede ofrecer niveles de eficiencia como los de lenguajes compilados. El sistema que se estudia en este trabajo es un buen ejemplo de este tipo de aplicaciones, y por tanto, debemos realizar el desarrollo sobre un lenguaje que nos garantice cierta rapidez de cómputo *per se*. La alternativa a Python que ofrece Autodesk Maya para la implementación e integración de plug-ins es C++. Por estos condicionantes, se ha decidido enfocar este proyecto como el desarrollo de MSXPBD en forma de plug-in en C++ para Maya, aplicando las técnicas de optimización y paralelización pertinentes.

La motivación del proyecto que presentamos aquí, por tanto, surge a partir del trabajo futuro indicado por Romeo et al., y se resume en el diseño, implementación y pruebas de una versión optimizada del sistema MSXPBD. Por todo lo expuesto en el presente capítulo, y si la consecución de este proyecto logra mejorar las prestaciones de MSXPBD, no es descabellado pensar que las futuras líneas de investigación en simulación de músculos se decanten por la vía XPBD dejando de lado los costosos métodos numéricos con Elementos Finitos.

Capítulo 4

Objetivos

Vamos a definir los objetivos que este proyecto debe alcanzar teniendo en cuenta que se enmarca dentro del área de la Computación de Altas Prestaciones (HPC, del inglés *High Performance Computing*) y partiendo de la motivación justificada en el capítulo anterior. Dicha motivación nos obliga a definir tanto objetivos técnicos como objetivos de aplicabilidad en el seno de la industria de los Efectos Visuales.

Como objetivos técnicos incluimos las premisas básicas e intrínsecas al proyecto así como los aspectos más ingenieriles que deseamos cubrir:

- **Rediseñar el sistema original MSXPBD**.
  Definir un nuevo diseño que permita la aplicación de técnicas de optimización y paralelización tomando como punto de partida el modelo *Muscle Simulation with Extended Position Based Dynamics*.

- **Implementar el sistema en C++**.
  Mover la implementación a C++ para que la ganancia de prestaciones sea más notable, teniendo en cuenta que el sistema original MSXPBD fue presentado en código Python single-threaded.

- **Optimizar el código**.
  Aplicar técnicas de paralelización basadas en el uso de OpenMP para optimizar el modelo y acelerar los tiempos de cómputo de las simulaciones, ofreciendo así un sistema en C++ multi-threaded.

- **Introducir los Métodos de Jacobi y de Coloreado de Grafos**.
  De las diferentes técnicas de paralelización de PBD y XPBD existentes en la literatura, estudiar y aplicar el Método de Jacobi y el Método de Coloreado de Grafos, dando lugar en realidad, a dos versiones distintas del sistema multi-threaded.

- **Aplicar conceptos de Computación de Altas Prestaciones**.
  Aplicar los conocimientos adquiridos en la asignatura de Computación de Altas Prestaciones para estudiar y localizar los bloques de código que son optimizables y para seleccionar las técnicas más adecuadas que permitan exprimir al máximo los márgenes de mejora.

- **Garantizar la estabilidad de las simulaciones**.
  Ofrecer un sistema convergente en todos los escenarios para que no se produzcan inestabilidades en las simulaciones independientemente de la parametrización del modelo.

Identificamos como objetivos de aplicabilidad aquellos fines que no condicionan el éxito del proyecto en términos técnicos pero que son indispensables para integrar el sistema en la industria una vez finalizado el desarrollo:

- **Generar un plug-in compilado para Maya**.
  Ofrecer el sistema como plug-in integrable en Maya, el cual se utilizará como entorno de visualización donde mostrar los resultados del proyecto, aprovechando que es uno de los software de uso estándar en la industria de los Efectos Visuales y la Animación 3D.

- **Garantizar la alta calidad en los resultados**.
  Mantener o mejorar la calidad y el realismo de las simulaciones que ya se obtienen con el sistema original MSXPBD.

- **Facilitar la configuración del sistema**.
  Mantener o mejorar la fluidez de MSXPBD para obtener resultados, realizar modificaciones en la configuración del sistema y volver a simular con rapidez.

- **Ofrecer controles intuitivos**.
  Mantener o mejorar la sencillez de los controles que se le ofrecen al usuario al mismo tiempo que se garantiza la coherencia entre un cambio introducido en la parametrización y el resultado que se obtiene por ese cambio.

- **Garantizar la flexibilidad artística**.
  Mantener la libertad artística para satisfacer las necesidades creativas.

Capítulo 5

Metodología y Gestión del Proyecto

En este capítulo vamos a sentar las bases metodológicas que van a conducir el desarrollo del proyecto. En los apartados que siguen, planteamos el proceso que tendrá lugar para diseñar, implementar y testear la optimización del sistema de simulación de músculos MSXPBD. Por otra parte, expondremos un estudio preliminar sobre la paralelización analizando el modelo original en términos algorítmicos. También se incluye una sección dedicada a las tecnologías y el hardware que se empleará durante el desarrollo. Además, realizaremos la gestión del proyecto necesaria para detallar el alcance, la planificación de tareas y la estimación de los costes asociados.

5.1. Metodología

La metodología que vamos a aplicar responde al esquema típico en desarrollo de software basado en diseño, implementación y pruebas. Tal y como se ha adelantado en el Capítulo 4, en este proyecto se va a desarrollar una optimización del modelo MSXPBD que implica su implementación en C++ secuencial para posteriormente introducir las mejoras en una versión paralelizada. Por tanto, estamos hablando de dos etapas similares que van a replicar este proceso basado en diseño, implementación y pruebas.

En primer lugar, el desarrollo en C++ secuencial comenzará por la realización de un estudio de MSXPBD para comprender el modelo y su algoritmia. Esto servirá para presentar el diseño de clases y completar la implementación. En la fase siguiente de pruebas se evaluarán los resultados y la corrección del sistema para extraer conclusiones en términos de calidad de la simulación y costes de cómputo.

En segundo lugar, el desarrollo de la versión optimizada comienza por analizar los resultados de coste obtenidos con la versión secuencial para detectar los puntos críticos sobre los que debe actuar la paralelización. En el diseño de la optimización se determinará qué estrategia seguir

para la adaptación del código y la introducción de las técnicas de paralelización pertinentes. En este punto, se llevará a cabo la integración de dos de los métodos de paralelización presentados en la sección de Estado del Arte 2.2.1: el Método Jacobi y el Método de Coloreado de Grafos. Por tanto, el diseño, implementación y pruebas de la versión en C++ *multi-threaded* tendrá, a su vez, estas dos vertientes.

En las respectivas fases de pruebas de cada sistema se realizarán test de simulación sobre los mismos escenarios con idéntica configuración para que la comparativa final sea lo más objetiva posible. En lo que a análisis de calidad de la simulación se refiere, proponemos siete escenarios que pondrán a prueba el sistema, ya que están orientados a detectar fallos y deficiencias de forma muy clara. De menor a mayor complejidad, los siete escenarios **Test** son:

- **Test 1** (T1): Un sólo músculo simple con forma de esfera ovalada que, partiendo de un estado de reposo, se activa y se relaja. El objetivo de este test es confirmar que el sistema de simulación funciona adecuadamente a nivel global, es decir, comunicación entre la aplicación y el plug-in, integración de fuerzas, cómputo de las restricciones, actualización del objeto dentro de la escena y visualización del resultado. Ver Figura 5.1.

- **Test 2** (T2): Un bíceps anatómico que se contrae y se relaja. El objetivo aquí es comprobar que el sistema funciona también cuando simulamos una geometría más compleja, tanto a nivel de número de vértices como de bifurcación en los tendones. Así, comprobaremos que la estructura interna del músculo que se genera es la correcta y que la dirección de las fibras musculares también si la contracción del músculo es coherente con lo que cabría esperar en un bíceps. Ver Figura 5.2.

- **Test 3** (T3): Un bíceps anatómico que se contrae y se relaja por la flexión del codo del esqueleto. El principal objetivo de esta prueba es determinar si la conexión e interacción del músculo con objetos estáticos (objetos colisionadores de la escena que no se computan como músculos) es la adecuada. Ver Figura 5.3.

- **Test 4** (T4): Bíceps y brachialis anatómicos que se contraen y se relajan de manera síncrona. En este caso, se pretende demostrar que el sistema es capaz de simular varios músculos al mismo tiempo y que, además, la interacción inter-muscular funciona correctamente. También interaccionan con un plano estático. Ver Figura 5.4.

- **Test 5** (T5): Un brazo humano completo con 17 músculos que simulan la flexión del codo. El objetivo de este test es comprobar la escalabilidad del sistema y determinar si la implementación ofrece buenos resultados cuando la complejidad del cálculo es elevada. Ver Figura 5.5.

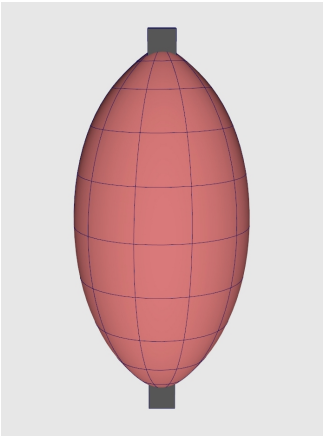

Figura 5.1: Escena de Maya para T1.

- **Test 6** (T6): Un brazo humano completo con 17 músculos en una escena donde el personaje hace movimientos de lucha. En esta animación, los desplazamientos de los huesos son rápidos y bruscos. Por ello, este test permite demostrar la estabilidad del sistema sometiéndolo a estrés y a fuerzas e inercias muy grandes. Ver Figura 5.5.

- **Test 7** (T7): Cuerpo humano completo con 132 músculos en una escena en la que personaje reproduce movimientos de caminar y correr. La simulación de este escenario tendrá lugar en 6 procesos independientes: se divide el cuerpo en lado izquierdo y lado derecho, y además, se separan brazos, torso y piernas. El objetivo del test es reproducir la estrategia de simulación que realizan Romeo et al. en el sistema original MSXPBD en Python. Aquí se pretende validar la aplicabilidad de nuestra implementación en un caso de producción real con un personaje completo. Ver Figura 5.6.

La evaluación de los resultados en lo que a costes se refiere es una parte fundamental en este proyecto para justificar la optimización que se pretende conseguir. Por ello, se realizará un estudio comparativo entre los tiempos de cómputo que se obtienen con la versión secuencial, con Jacobi y con el método de Grafos. En este análisis de costes se llevarán a cabo cuatro casos de estudio con número de restricciones y de músculos creciente. La idea fundamental es evaluar el grado de mejora que se consigue con las versiones optimizadas a medida que aumenta la complejidad del problema. Estos casos de prueba los catalogaremos como **Profiling**:

- **Profiling 1** (P1): Un músculo con forma de esfera ovalada sujeto por dos puntos en sus extremos y con un único objeto colisionador a modo de target estático, a saber, un plano. Ver Figura 5.7.

- **Profiling 2** (P2): Dos músculos simples, un óvalo y un cubo, sujetos por un vértice de un extremo. Están interconectados para colisionar entre sí, además de con un plano que

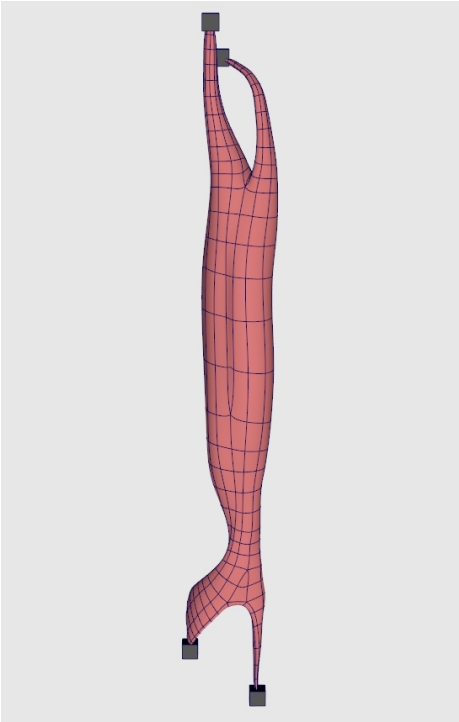

Figura 5.2: Escena de Maya para T2.

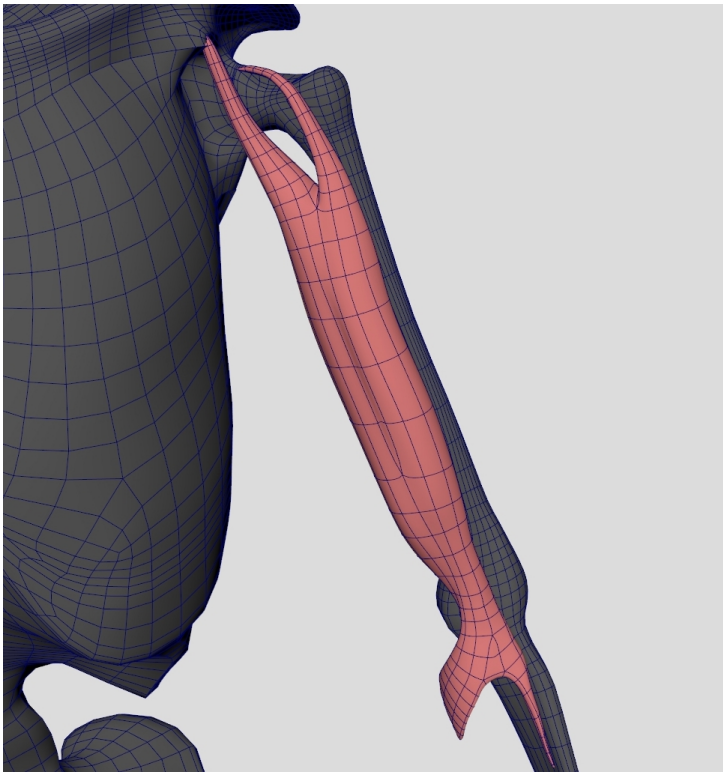

Figura 5.3: Escena de Maya para T3.

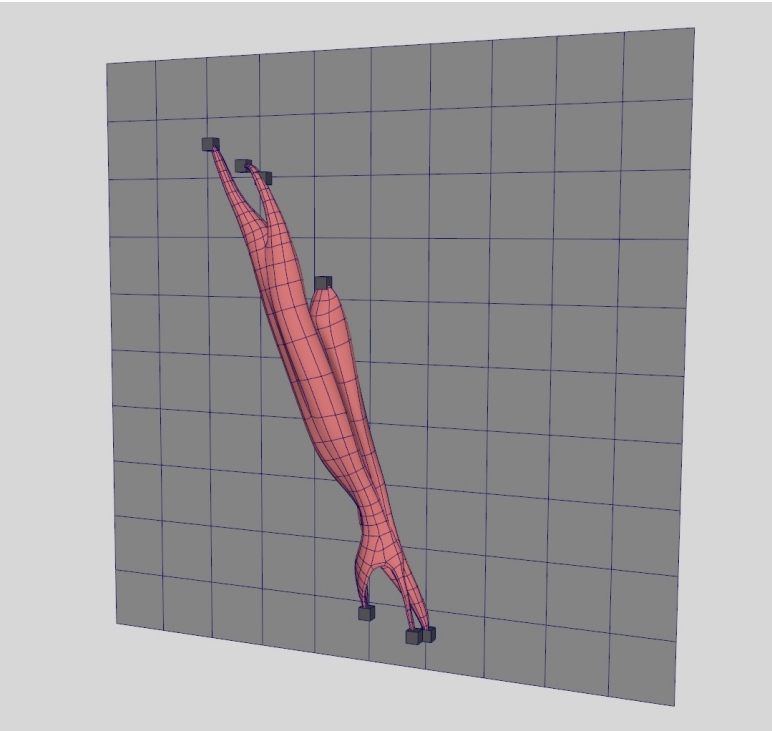

Figura 5.4: Escena de Maya para T4 y P3.

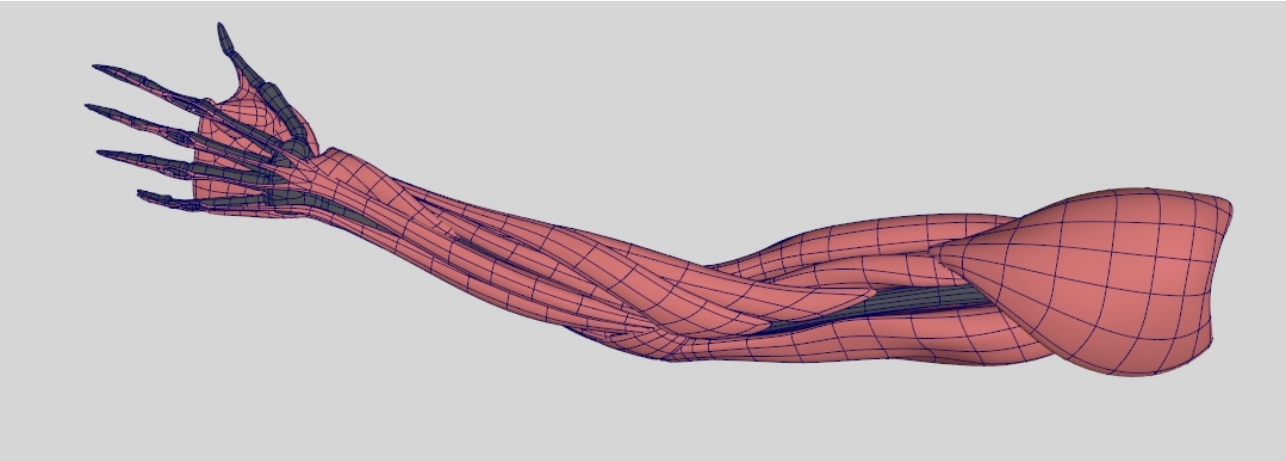

Figura 5.5: Escena de Maya para T5, T6 y P4.

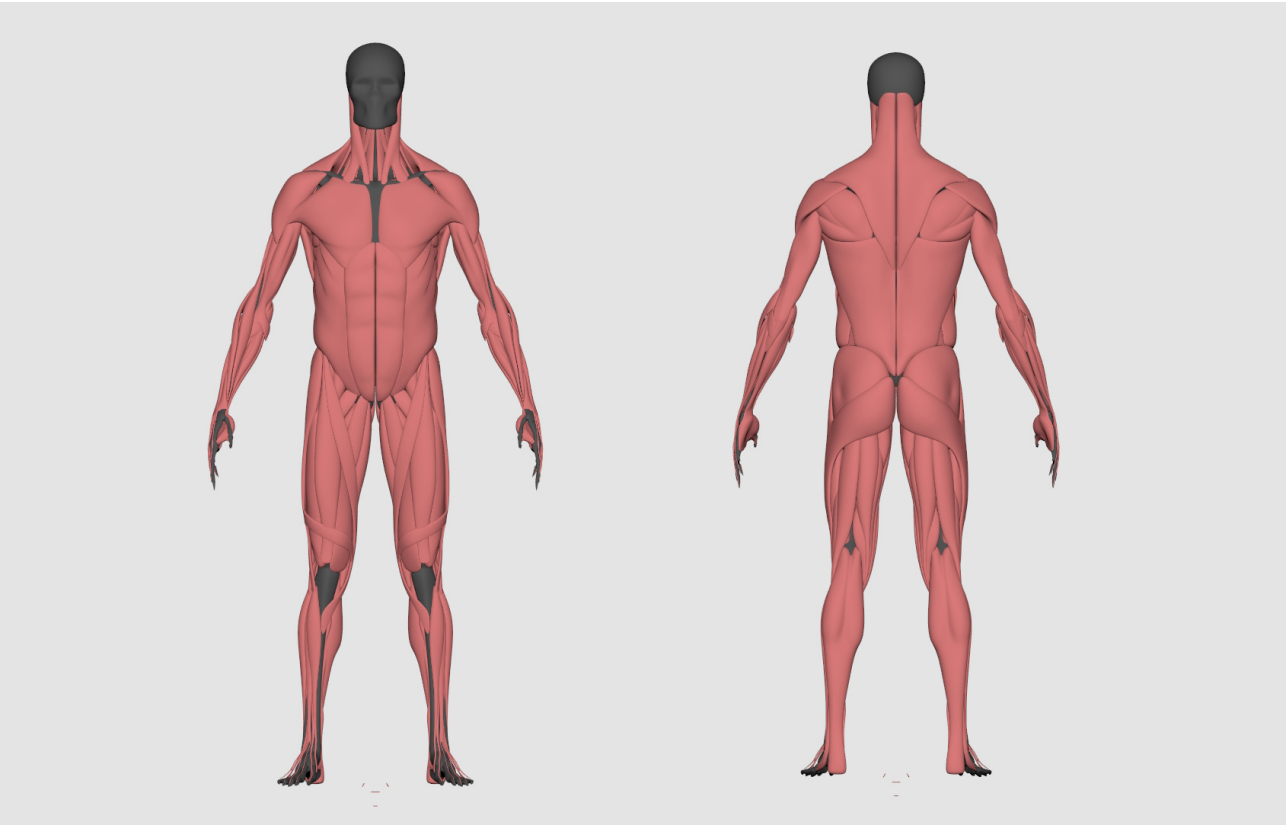

Figura 5.6: Escena de Maya para T7.

actúa como objeto estático. Ver Figura 5.8.

- **Profiling 3** (P3): Bíceps y brachialis anatómicamente realistas con cuatro y dos puntos de sujeción respectivamente. Están interconectados como en el caso anterior además del plano estático. Es el mismo escenario que T4 pero se libera uno de los puntos de sujeción del bíceps, ver Figura 5.4.

- **Porfiling 4** (P4): Brazo humano anatómicamente realista con 17 músculos sujetos al esqueleto por varios vértices en sus extremos. Cada músculo interacciona con un conjunto de músculos colisionadores que se asigna por vecindario, además de los huesos más próximos del esqueleto. Es el mismo escenario que T5, ver Figura 5.5.

Dada la variedad de estas escenas propuestas y la complejidad del modelo MSXPBD en sí mismo, los escenarios P1 a P4 darán lugar a distintas configuraciones del sistema en lo que a número de restricciones se refiere. Esto deberá tenerse en cuenta a la hora de analizar los tiempos de cómputo ya que la exigencia de cálculo depende directamente de la cantidad de restricciones. A modo esquemático, en las Tablas 5.1, 5.2, 5.3 y 5.4 se muestra el conjunto de restricciones por tipo resultantes en cada test de profiling, además del recuento total en

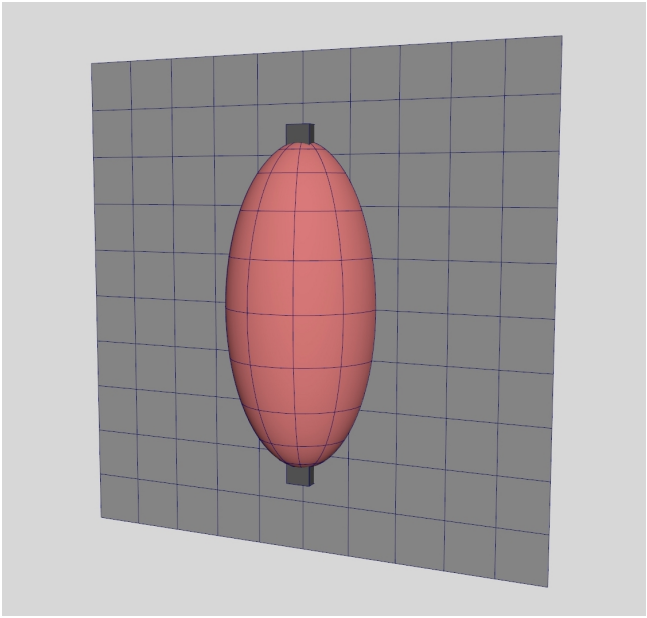

Figura 5.7: Escena de Maya para P1.

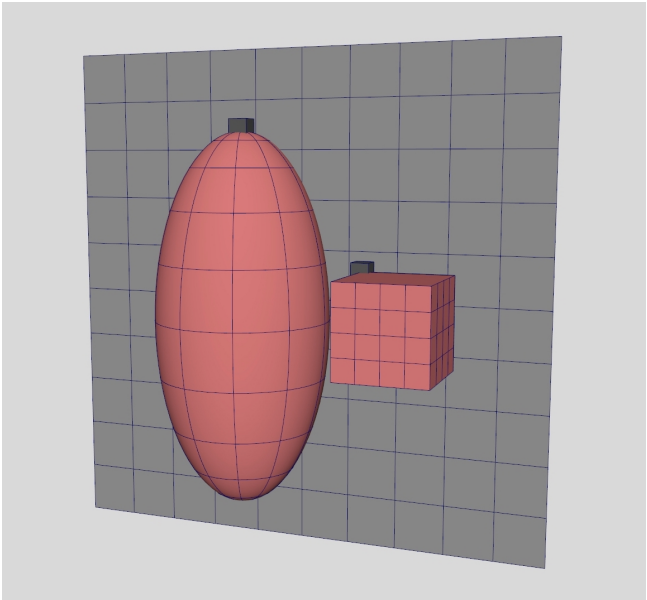

Figura 5.8: Escena de Maya para P2.

Fiber	Attach	BrAttach	Sliding	Soft	Hard	Internal	Branch	**TOTAL**
190	2	2	92	92	92	92	8	**571**

Tabla 5.1: Número de restricciones por tipo en P1.

Fiber	Attach	BrAttach	Sliding	Soft	Hard	Internal	Branch	**TOTAL**
334	2	2	172	172	172	172	12	**1040**

Tabla 5.2: Número de restricciones por tipo en P2.

la última columna. En la cabecera de las tablas se ha utilizado un seudónimo en inglés para relacionarlos con su breve descripción en la lista que sigue:

- Fiber (*FiberConstraint*): conexión entre dos vértices de la malla del músculo que implementa la activación muscular del MSXPBD, una por cada arista de la geometría.
- Attach (*AttachConstraint*): sujeción de un vértice de la malla del músculo a un punto externo, por ejemplo, de un hueso.
- BrAttach (*BranchAttachConstraint*): sujeción de un punto de la estructura interna del músculo a un punto externo, por ejemplo, de un hueso.
- Sliding (*SlidingConstraint*): conexión de un vértice de la malla del músculo con un punto en la malla de un target o colisionador (permite deslizamiento sobre la superficie).
- Soft (*SoftConstraint*): conexión de un vértice de la malla del músculo con un punto en la malla de un target o colisionador (no permite deslizamiento y actúa como un muelle).
- Hard (*HardConstraint*): conexión de un vértice de la malla del músculo con un punto en la malla de un target o colisionador (mantiene completamente la transformación relativa).
- Internal (*InternalConstraint*): conexión de un vértice de la malla del músculo con un punto de la estructura interna.
- Branch (*BranchConstraint*): conexión entre dos puntos de la estructura interna.

Para la medición de tiempos se van a agrupar las instrucciones del bucle de simulación según un criterio de funcionalidad, es decir, bloques de código en los que cada uno desarrolla una

Fiber	Attach	BrAttach	Sliding	Soft	Hard	Internal	Branch	**TOTAL**
1327	6	6	666	666	666	666	30	**4035**

Tabla 5.3: Número de restricciones por tipo en P3.

Fiber	Attach	BrAttach	Sliding	Soft	Hard	Internal	Branch	**TOTAL**
6015	65	65	3024	3024	3024	3140	193	**18567**

Tabla 5.4: Número de restricciones por tipo en P4.

tarea concreta del solver. Para poder extraer conclusiones más acertadas del análisis, se fijará el número de iteraciones en 5 para todos los casos y además, se simularán 120 fotogramas para después obtener el promedio de tiempos de cada bloque de código. Los datos que se extraigan permitirán decidir qué bloques se deben paralelizar. De este modo, el cálculo de costes final con el sistema multi-threaded se basará en medir y comparar cuánto se reduce el tiempo de cómputo absoluto de tales bloques respecto al sistema single-threaded. Con vistas a facilitar la lectura, vamos a catalogar los bloques como **Bloques de Ejecución** (BE), teniendo en cuenta el orden en que se computan:

- **Evaluar Grafo** (BE1): Conjunto de instrucciones ejecutadas desde la capa de aplicación que tienen como objetivo recorrer el grafo de la escena para recopilar los datos necesarios: configuración del escenario, información de geometrías, parametrización de los nodos, etc.

- **Actualizar Colisionadores** (BE2): Este paso se ocupa de volcar la información de geometría de los targets estáticos (objetos no computados como músculos) desde la aplicación al solver.

- **Integración** (BE3): Ya en la capa core, integración numérica de las fuerzas externas y de las velocidades de todos los vértices y puntos de la estructura interna de todos los músculos.

- **Actualizar Targets** (BE4): El solver le actualiza a cada músculo la información de geometría de sus targets correspondientes.

- **Closest Data** (BE5): Proceso mediante el cual cada vértice de cada músculo actualiza los datos relacionados con las restricciones externas de sliding, soft y hard sobre cada target: cálculo de puntos más próximos, vectores normales, coordenadas baricéntricas, intersecciones, etc.

- **Proyección Restricciones** (BE6): Se corresponde con el cálculo de las correcciones de posición para cada vértice que cada restricción debe aplicar.

- **Actualizar Geometría** (BE7): Cada músculo vuelca el resultado obtenido al final de cada iteración del solver en la malla. Es el último paso a nivel de músculo, es decir, después de obtener las nuevas posiciones tras la proyección de restricciones.

- **Aplicar Predicciones** (BE8): Paso final a nivel de restricción que se encarga de aplicar los deltas de posición calculados sobre las predicciones de los vértices afectados. En este punto, se obtienen las nuevas posiciones que deberán visualizarse en la escena antes de pasar al siguiente fotograma.

Otra medición de costes interesante para evaluar este proyecto tiene que ver con el tiempo medio de cómputo por fotograma. Realizando este examen en los cuatro escenarios de profiling, veremos qué impacto tiene el número de restricciones sobre la eficiencia del sistema en su totalidad. Se compararán los resultados de este análisis pertenecientes a la versión secuencial y a las implementaciones con Jacobi y con Coloreado de Grafos, así, también deduciremos qué método es más apropiado para optimizar MSXPBD y en qué espectro de mejora nos movemos.

En esta misma línea, se pretende comparar los resultados con los tiempos de cálculo que Romeo et al. exponen en [16]. Tanto es así, que la motivación principal de este proyecto es conseguir una implementación más eficiente que el original MSXPBD en Python. Por tanto, será necesario definir un experimento lo más parecido posible al que describen los autores en el artículo. En su caso, para el cálculo de costes, diseñan un escenario en el que dividen la simulación del cuerpo humano completo en 6 procesos independientes (brazos por separado, piernas por separado y lo mismo con las dos mitades del torso). Configuran un sistema con 135780 restricciones y 10 iteraciones en el solver de simulación. Así, una vez dispongamos del plug-in compilado, trataremos de definir una escena similar que contenga el mismo (o parecido) número de restricciones.

La propuesta metodológica para este proyecto concluye con una recopilación de las limitaciones del sistema extraídas del análisis de resultados, así como una definición de los puntos a mejorar y de las líneas a desarrollar como trabajo futuro.

5.2. Estudio previo a la optimización

El propósito de esta sección es analizar la estructura del modelo de simulación para detectar los puntos del sistema que son susceptibles de ser paralelizados. Se trata de un análisis preliminar, ya que el definitivo tendrá lugar sobre el sistema en su versión secuencial una vez que esté implementado. Dicha tarea tendrá mucho peso en el global del proyecto porque cuanto más exhaustivo sea dicho estudio, mejores decisiones podremos tomar acerca de la paralelización y mayores beneficios obtendremos a nivel de prestaciones.

De hecho, uno de los fundamentos del área HPC es conocer y analizar con detalle el algoritmo para aplicar las metodologías adecuadas en los puntos adecuados. En otras palabras, será necesario hacer un profiling de la implementación secuencial para medir los tiempos de ejecución. De esta manera podremos saber qué cálculos tienen mayor coste y, por tanto, qué bloques

de código deben paralelizarse. Sabremos también si esos bloques ejecutan llamadas de código C++ puro desde la capa core, o se corresponden con paso de información por parte de la capa intermedia, o bien si se trata de llamadas a la propia API de Maya. Con toda esa información se podrán tomar decisiones acertadas sobre la paralelización de los bucles o la reimplementación de algunas partes del código para favorecer la optimización.

Como decíamos, para poder completar este detallado estudio es necesario disponer de la implementación del sistema en versión secuencial. Por ese motivo, de momento nos vamos a limitar a exponer el algoritmo del modelo y a plantear las hipótesis de paralelización desde un punto de vista teórico. En el Algoritmo 1 se muestra el bucle de simulación del modelo MSXPBD, esto es, el conjunto de instrucciones que se ejecutan en cada fotograma.

Algorithm 1 Pseudocódigo del bucle de simulación

```
1:  Recoger datos globales de la escena
2:  Enviar datos globales al solver
3:  for all músculos m do
4:      Recoger datos locales de m
5:      Enviar datos locales de m al solver
6:      Actualizar velocidades de los vértices de m
7:      Actualizar posiciones de los vértices de m
8:  for all iteraciones i do
9:      for all músculos m do
10:         Leer geometrías de músculos vecinos a m
11:         for all restricción r do
12:             Computar r
13:         Actualizar la geometría de m
14: for all músculos m do
15:     Actualizar velocidad de los vértices de m
16:     Enviar posiciones resultantes a la aplicación
17:     Actualizar vértices en la geometría de m para la visualización
```

Como introducción al análisis de la optimización del sistema, se adjunta la tabla 5.5 en la que se estudian los bucles presentes en el Algoritmo 1 para determinar las posibilidades de paralelización. La columna *capa* se refiere a la(s) capa(s) de abstracción encargada(s) de ejecutar cada bloque de instrucciones, a saber: *core* si son cálculos propios del modelo MSXPBD, *aplicación* si son operaciones a nivel de API de Maya, o *intermedia* si son instrucciones de comunicación entre el core y la aplicación.

Es necesario aclarar algunas cuestiones de la tabla 5.5. Por un lado, vemos que los bucles 3-7 y 14-17 están indicados como paralelizables. Dichos bucles implican la participación de las tres capas, incluyendo la capa de aplicación. En estos casos, tendremos que asegurarnos previamente de que las llamadas a la API de Maya son compatibles con directivas de paralelización. Si esto

	Capa	Par.	Comentarios
3-7	Aplicación Intermedia Core	Sí	La recogida de parámetros particulares de cada músculo es independiente una de otra. También lo es la integración de fuerzas y velocidades.
8-13	Aplicación Intermedia Core	No	El estado de la partículas tras el cómputo de una iteración influye directamente en el cómputo de la iteración siguiente. La convergencia de MSXPB es posible gracias a ello.
9-13	Aplicación Intermedia Core	No	El cómputo de un músculo depende del estado de la geometría de los músculos vecinos con los que se conecta. Se necesita la información más actualizada posible.
11-12	Core	Sí	Los métodos **Jacobi** y **Coloreado de Grafos** permiten calcular las restricciones de manera simultánea.
14-17	Aplicación Intermedia Core	Sí	La actualización de las velocidades de las partículas de un músculo es independiente del resto de músculos. Ocurre lo mismo con la actualización de los vértices en la geometría.

Tabla 5.5: Estudio de la paralelización del algoritmo. Cada columna refleja de izquierda a derecha: índices de línea en el algoritmo; capa(s) que ejecuta(n) las instrucciones; posibilidad de paralelización; y breve justificación de la columna anterior.

no es posible, será necesario dividir esos bucles en varias partes para poder optimizar, al menos, aquellas instrucciones que se ejecutan únicamente desde la capa core. Esta y otras cuestiones serán las que se amplíen más adelante durante el diseño de la paralelización.

Por otro lado, el bucle 11-12 hace mención a los métodos de Jacobi y de Coloreado de Grafos. Tal y como se introdujo en el capítulo de Estado del Arte, el método Jacobi es una de las técnicas de paralelización aplicables al modelo XPBD, el cual está basado en la acumulación de las correctivas calculadas por cada restricción para aplicarlas de forma promediada sólo una vez al término de cada iteración, justo antes de procesar la siguiente iteración. Por lo que respecta al Coloreado de Grafos, cabe recordar que consiste en la agrupación de las restricciones aplicando un criterio independencia entre sí en lo que a compartición de vértices afectados se refiere. Con esta separación se garantiza la convergencia del sistema y la atomicidad a nivel de escritura de los deltas de posición (consultar la subsección 2.2.1).

5.3. Discusión de la tecnología

La tecnología que se va a utilizar para la realización del proyecto, tanto a nivel de desarrollo como de posterior aplicación viene condicionada por la plataforma de destino que hará uso del sistema, el lenguaje de programación y las técnicas de optimización que se pretenden aplicar. A continuación, resumimos las decisiones tomadas acerca de estas cuestiones.

El sistema de simulación de músculos que presentamos está pensado para su aplicación en

la industria de los Efectos Visuales y la Animación 3D. En este sector se utilizan múltiples programas a lo largo de toda una producción. Algunos están especializados en sólo una parte del *pipeline* de trabajo y otros son más generalistas, abarcando desde las primeras etapas de modelado hasta fases de simulación, animación e incluso iluminación. En este proyecto utilizaremos Autodesk Maya como plataforma sobre la que implementar el sistema de músculos. El motivo principal es que se trata de un software de carácter más generalista y es uno de los más utilizados en VFX.

En cuanto a lenguaje de programación, las alternativas de implementación de plug-ins para Maya que existen son dos: Python y C++. Como se ha dicho con anterioridad, el modelo de simulación MSXPBD de Romeo et al. en el que basaremos nuestro desarrollo está implementado en Python. Dado que el objetivo fundamental que se ha marcado es la optimización de dicho modelo, vamos a optar por utilizar C++ por ser un lenguaje compilado muy potente y para continuar con la línea de trabajo futuro propuesta en el artículo original de MSXPBD.

Si hablamos de las herramientas para el estudio de costes, utilizaremos la librería *chrono* de C++. Nuestra principal preocupación en este punto es identificar los bloques de código que ejercen de cuello de botella durante la simulación, no tanto la comparativa entre las distintas herramientas de análisis en sí. Por eso, trataremos de ser prácticos al respecto y no sobrecargar el desarrollo del proyecto con las dificultades que puedan conllevar el uso e integración de herramientas de análisis específicas dentro de Maya.

En lo referente a las técnicas de paralelización, el análisis realizado en la sección 5.2 anterior y los métodos estudiados en 2.2.1, hacen constar que XPBD es paralelizable aplicando el paradigma de memoria compartida, es decir, paralelismo a nivel de tareas. Esto significa que el cómputo se divide en tareas que son independientes entre sí y que se ejecutan simultáneamente en los diferentes cores.

Esta cuestión nos lleva a encontrar una interfaz que nos permita implementar tal paralelismo en C++. Se ha decidido utilizar OpenMP basándonos en los conocimientos adquiridos en la asignatura de Computación de Altas Prestaciones y en su compatibilidad con Maya. Una de sus características principales es que permite paralelizar programas secuenciales sin el esfuerzo de reescribirlos por completo, lo cual es una ventaja para nosotros ya que realizaremos primeramente la implementación del sistema en su versión secuencial. Así, OpenMP nos permitirá especificar qué bloques del código secuencial se ejecutarán en paralelo.

Respecto a la tecnología hardware, la herramienta de la que disponemos es un único equipo portátil modelo MacBook Pro Retina de mediados de 2012. Las especificaciones técnicas son:

SO: MacOS High Sierra 10.13.6
Procesador: 2,3 GHz Intel Core i7

Memoria: 8 GB 1600 MHz DDR3
Gráfica Integrada: NVIDIA GeForce GT 650M 1024 MB
Gráfica PCIe: Intel HD Graphics 4000 1536 MB
Cantidad de procesadores: 1
Cantidad total de núcleos: 4
Caché de nivel 2 (por núcleo): 256 KB
Caché de nivel 3: 6 MB

5.4. Gestión de alcance y planificación

En este capítulo presentamos la planificación del proyecto en términos de tareas, tiempos de ejecución y documentos a entregar asociados a las mismas. Para ello, retomamos el listado de etapas de desarrollo que se proponen en el Capítulo 1 y las reagrupamos para ajustarlas a un calendario basado en cuatro bloques que coinciden con los cuatro hitos de entregas:

1. **Bloque 1:** Diseño y Estructura del Proyecto.
   En este primer bloque se define el proyecto y su motivación, se recopilan los requisitos y los objetivos junto con el Estado del Arte, se planifica la distribución de tareas con su duración y costes y se discute la metodología. En este último punto, se incluye un estudio previo del sistema que se quiere optimizar con la finalidad de detectar los puntos de la implementación que son paralelizables.

2. **Bloque 2:** Implementación del sistema en C++ single-thread.
   En el segundo bloque se realiza la implementación del sistema en C++ secuencial y las pruebas asociadas para demostrar el funcionamiento, así como el análisis de los resultados y de los tiempos de cómputo.

3. **Bloque 3.** Implementación del sistema en C++ multi-thread.
   El tercer bloque comienza con un análisis exhaustivo del código secuencial para definir los cálculos que se quieren paralelizar. Hecho esto, se estudian las técnicas de HPC más apropiadas en cada caso para posteriormente implementarlas, a saber, los métodos de Jacobi y Coloreado de Grafos. Por tanto, este bloque se resume en el análisis, diseño, implementación y pruebas del sistema en C++ paralelizado para cada modelo. Del mismo modo que en el Bloque 2, se incluyen los pertinentes estudios de coste y de calidad de las simulaciones.

4. **Bloque 4:** Documentación.
   Para finalizar, el bloque cuarto se ocupa de recoger y documentar todo el desarrollo del proyecto en la memoria final. También formará parte del proceso de documentación la generación de simulaciones y la producción y edición de vídeos con ejemplos de uso y de resultados del sistema.

En cuanto a planificación, por un lado, las tareas deben ajustarse al intervalo temporal delimitado por las fechas de inicio y fin del proyecto: 5 de noviembre de 2018 y 15 de julio de 2019 respectivamente, lo que hacen un total de 36 semanas. Por otro lado, deben tenerse en cuenta periodos vacacionales y días festivos que puedan causar la dilatación en el tiempo de algunas tareas. En el caso que nos ocupa, existe un periodo de descanso entre el 28 de noviembre y el 11 de diciembre de 2018, ambos incluidos.

Asimismo, es importante conocer el esfuerzo computable al proyecto por parte del desarrollador, esto es, su carga de trabajo diaria o semanal. Por motivos laborales y personales, la dedicación a las tareas académicas se ve reducida a una media de 15 horas semanales. Teniendo en cuenta el solapamiento con otras asignaturas que se cursan durante el primer semestre, la dedicación resultante varía a lo largo de los meses según la siguiente distribución:

- Del 5 de noviembre al 20 de enero (11 semanas). Dedicación muy reducida por solapamiento con dos asignaturas del máster. Distribuyendo equitativamente el margen de 15 horas, resultan sólo 5 horas semanales al proyecto.

- Del 21 de enero al 15 de julio (25 semanas). Dedicación más amplia debido al término de las asignaturas paralelas. En este periodo, las 15 horas semanas pasan a ser exclusivamente para el proyecto final.

En definitiva, la disponibilidad del autor para desarrollar el proyecto es de 430 horas repartidas a lo largo de las 36 semanas comentadas. Partiendo de estos condicionantes y pensando en los periodos vacacionales y el intervalo de fechas de inicio y de fin, se ha calculado la duración de cada tarea en la Tabla 5.6.

De acuerdo con los hitos de entrega establecidos y su adecuación al contenido de los cuatro bloques de tareas principales, se han definido los entregables que se listan a continuación (consultar Tabla resumen 5.7):

1. **Entregables 1**. Documento escrito que desarrolle los puntos que siguen: introducción, Estado del Arte, motivación, objetivos, planificación, análisis de costes y metodología.

2. **Entregables 2**. Vídeo con una compilación de ejemplos de simulaciones obtenidas con el sistema en su versión secuencial. Se espera que sean ejemplos donde se pueda comprobar la

	Tarea	Inicio	Final	Horas	Semanas
1	Diseño y Planificación	5/nov	2/dic	20	4
2	Implementación C++ single-thread	3/dic	31/mar	180	17
2.1	Implementación core	3/dic	17/feb	90	11
2.2	Implementación plug-in en Maya	18/feb	3/mar	30	2
2.3	Implementación capa intermedia	4/mar	17/mar	30	2
2.4	Pruebas	18/mar	24/mar	15	1
2.5	Calidad de la simulación	25/mar	26/mar	5	0.33
2.6	Análisis de costes	27/mar	31/mar	10	0.66
3	Implementación C++ multi-thread	1/abr	16/jun	170	11
3.1	Estudio y diseño de la paralelización	1/abr	28/abr	60	4
3.2	Implementación core	29/abr	31/may	70	4.75
3.3	Adaptación plug-in en Maya	1/jun	5/jun	10	0.75
3.4	Pruebas	6/jun	11/jun	15	1
3.5	Calidad de la simulación	12/jun	13/jun	5	0.3
3.6	Análisis de costes	14/jun	16/jun	10	0.2
4	Documentación	17/jun	15/jul	60	4
4.1	Redacción de la memoria	17/jun	4/jul	40	2.5
4.2	Generación de simulaciones y vídeos	5/jul	15/jul	20	1.5
	TOTAL	**5/nov**	**15/jul**	**430**	**36**

Tabla 5.6: Planificación del Proyecto. Desglose de los cuatro bloques principales en subtareas, cada una con su fecha de inicio y de fin y la duración total en horas con su aproximación a semanas.

dinámica de los músculos, su interacción y su correcta activación. Además, un documento con el análisis de los resultados en cuanto a calidad de las simulaciones y tiempos de cómputo.

3. **Entregables 3**. De forma análoga al punto anterior, aquí se precisa de un vídeo con las simulaciones obtenidas con el sistema optimizado, tanto con Jacobi como con Coloreado de Grafos, junto a un documento de análisis de resultados y tiempos. Este documento debe contener también el estudio previo a la implementación sobre las decisiones tomadas desde el punto de vista de la paralelización del sistema.

4. **Entregables 4**. Memoria final del proyecto además de un vídeo recopilatorio de las entregas 2 y 3, añadiendo nuevas demostraciones y comparativas entre los sistemas implementados. Asimismo, se debe incluir también un vídeo presentación en el que el autor expone el trabajo realizado.

	Tarea	Entregable
1	Diseño y Planificación	Documento de planificación
2	Implementación C++ single-thread	Análisis de resultados + Vídeo demostrativo
3	Implementación C++ multi-thread	Análisis de resultados + Vídeo demostrativo
4	Documentación	Memoria final + Vídeo resumen + Exposición

Tabla 5.7: Entregables. Resumen de entregables por cada bloque de tareas.

5.5. Estimación de costes

Como es habitual en gestión de proyectos, la estimación de costes económicos se divide en dos niveles correspondientes a los recursos humanos y a los recursos materiales. En lo que a recursos humanos se refiere, la totalidad del desarrollo correrá a cargo del alumno. Para calcular el coste laboral asociado, se debe analizar previamente las funciones a desempeñar y determinar a qué perfil profesional se adecúa. Siendo un proyecto de investigación ingenieril y carácter técnico, vamos a atender a los salarios publicados en el XVIII Convenio Colectivo de Empresas en Ingeniería y Oficinas de Estudios Técnicos del BOE[1], con fecha del miércoles 18 de enero de 2017.

En su Artículo 33, se fijan los salarios en diez niveles. El perfil del autor sería cercano al Nivel 2, es decir, al perfil de los diplomados y titulados de primer ciclo universitario. La retribución anual para este grupo es de 17.544,40 € . Asimismo, en el Artículo 22 se limita la jornada laboral a un máximo de 1.800 horas al año. Con estos datos, podemos decir que el salario es de 9,75 €/hora.

A fin de tener en cuenta los costes en Seguridad Social, debemos consultar las Bases y Tipos de Cotización para este año 2018. Esta información se puede encontrar en la página web del Ministerio de Empleo y Seguridad Social[2]. Los cargos a los que debe hacer frente una empresa que contrata a un trabajador del perfil que se ha comentado, son la suma de: contingencias comunes, de desempleo, de fogasa y de formación. Según la publicación del Ministerio, estos costes en procentaje respecto al salario del trabajador son del 28,30 %, 7,05 %, 0,20 % y 0,70 %, respectivamente. Esto hace un total del 36,25 %, que aplicado al salario por hora implica un aumento de 3,53€/hora.

Finalmente, nos queda un coste por recurso de 13,28€/hora. Si el número de horas de trabajo previstas es de 430, obtenemos un total de 5710,40 €. A esta cantidad, añadimos un margen del 5 % en contingencias para cubrir imprevistos (285,52€), de forma que el coste final de los recursos humanos asciende a 5995,92€.

Por otra parte, están los recursos materiales necesarios para el desarrollo de las tareas.

[1] https://www.boe.es/boe/dias/2017/01/18/pdfs/BOE-A-2017-542.pdf
[2] http://www.seg-social.es/wps/portal/wss/internet/Trabajadores/CotizacionRecaudacionTrabajadores/36537

A nivel de hardware, como es obvio en un proyecto basado en simulación por computador, el principal recurso a utilizar es un ordenador. Concretamente, se utilizará un MacBook Pro Retina de 15" que cumple con la siguiente descripción técnica: sistema Operativo OSX High Sierra, versión 10.13.6; procesador Intel Core i7 2.3GHz; memoria RAM de 8GB a 1600MHz DDR3; almacenamiento de 256GB en disco SSD; y gráfica NVIDIA GeForce GT 650M 1024MB.

El coste neto de este recurso es de 1.990€. Si consideramos un tiempo de vida útil de 4 años y que la amortización de este tipo de recursos va en función de los meses de uso, tenemos un coste amortizado de 41,46€/mes. Según la planificación del capítulo 5.4, el proyecto se extiende durante ocho meses y medio. Por tanto, el coste global es de 352,41€. Aplicando el 5 % de margen de confianza, obtenemos un coste final de 370,03 €.

A nivel de software, tal y como detallaremos en la sección 5.3, el proyecto se va a implementar íntegramente en CLion 2018.2.3 y a compilar con CMake 3.11. El software para las pruebas de simulación y visualización de resultados es Autodesk Maya 2018. Por un lado, el uso de CMake tiene coste nulo al ser una plataforma *open-source*. Y por otro lado, para el uso de CLion y Maya se han obtenido sendas licencias de estudiante. En definitiva, el coste computable al proyecto por uso de software es cero.

También son recursos materiales la conexión a internet y la luz eléctrica. Una tarifa media de conexión a internet de 30MB asciende a 14,90€/mes. Aplicando los ocho meses y medio de duración del proyecto y sumando las contingencias, el coste de internet es de 132,98 €.

Para el cálculo del coste de la luz eléctrica, nos guiamos por el consumo doméstico medio mensual, ya que no se necesitan grandes instalaciones. De hecho, el consumo por luz se reduce a un ordenador portátil. Por ello, tomemos la cifra aproximada de 30€/mes. Con el mismo procedimiento que para la conexión a internet, el coste aquí asciende a 267,75€, considerando el 5 % de margen de confianza.

En la Tabla 5.8 se recopilan los costes humanos y materiales, cuya suma nos da un total de 6766,68 €.

Recurso	€/hora	Horas	€/mes	Meses	Conting.(%)	Total
Hardware	-	-	41, 46	8,5	5	370,03
Software	-	-	0	8,5	5	0
Internet	-	-	14,9	8,5	5	132,98
Luz eléctrica	-	-	30	8,5	5	267,75
Trabajador	13,28	430	-	-	5	5995,92
					Total (€) =	**6766,68**

Tabla 5.8: Costes del proyecto.

Capítulo 6

Sistema Single-Threaded

En este capítulo se expone el diseño algorítmico, la definición de clases, el esquema de ejecución y el análisis de resultados del *Sistema Single-Threaded*, al cual nos referiremos de ahora en adelante como SST. El desarrollo de esta versión secuencial del sistema corresponde al Bloque 2 de tareas presentado en la Sección 5.4. Los apartados que siguen están estructurados de forma que: en 6.1 se expone el estudio de los algoritmos más importantes del SST; en 6.2 y 6.3 se recogen los diagramas de clases y de secuencia respectivamente; y por último, en 6.4 se realiza el análisis de los resultados obtenidos con el SST en términos de calidad de la simulación y costes de cómputo.

6.1. Definición Algorítmica

Para comprender la complejidad del modelo de simulación que se pretende implementar, vamos a retomar el algoritmo genérico del bucle de simulación (Algoritmo 1, Sección 5.2) para detallar aquellos bloques de pseudocódigo más complejos y relevantes.

Comenzando por el cómputo a nivel de aplicación, una parte fundamental que se lleva a cabo es la lectura de datos de la escena y su envío al solver (instrucciones 1-2 del Algoritmo 1). En cada fotograma de la simulación, se debe recoger el estado actualizado de la escena, esto es, leer y empaquetar la información de las geometrías (músculos, huesos y otros objetos colisionadores), la configuración de los parámetros del solver (diferencial de tiempo, fotograma actual, fuerzas externas, etc.) y también la configuración particular de cada músculo (fibras, activación, ganancia de volumen, etc.). Téngase en cuenta que distinguimos entre parámetros constantes y estáticos. Los primeros son aquellos que no cambian durante la simulación, es el caso del diferencial de tiempo, el número de iteraciones o las escalas de tiempo y espacio, entre otros. Los segundos son aquellos que pueden variar durante la simulación, por ejemplo, el nivel de activación de un músculo o su valor de ganancia de volumen.

Algorithm 2 Pseudocódigo de la ejecución del grafo de Maya en cada fotograma

```
1: Leer parámetros dinámicos globales
2: Leer lista de músculos                                   ▷ Nodos músculo activos
3: if frame == startFrame then
4:     Leer parámetros constantes globales
5:     Inicializar objeto solver                            ▷ Objeto C++ plano
6:     Inicializar colisionadores                           ▷ Objetos de la escena no músculos
7: for all nodo músculo m do
8:     if frame = startFrame then
9:         Leer parámetros constantes y dinámicos locales
10:        Inicializar objeto muscle                        ▷ Objeto C++ plano
11:        Inicializar restricciones de muscle
12:        solver.addMuscle(muscle)
13:    else
14:        Leer parámetros dinámicos locales
15:        Actualizar muscle con parámetros dinámicos locales   ▷ De la aplicación al solver
16: if frame ≠ startFrame then
17:    Actualizar solver con parámetros dinámicos globales      ▷ De la aplicación al solver
18:    Leer geometría de los colisionadores
19:    Actualizar la información de los colisionadores en el solver ▷ De la aplicación al solver
20:    Computar solver
21: for all nodo músculo m do
22:    Actualizar geometría de m en la escena                   ▷ Del solver a la aplicación
```

La recopilación de esta información depende directamente del modo en el que la aplicación manipula y almacena los datos. En nuestro caso, Maya ofrece un grafo de nodos interconectados que guardan toda la información necesaria. Para el desarrollo del SST, tendremos que implementar nuestros propios nodos, y por eso, las conexiones del grafo definirán no sólo los caminos de transmisión de datos sino también las relaciones entre las clases que implementaremos en la Sección 6.2.

Por otro lado, la aplicación también se encarga del paso contrario, es decir, una vez que el solver ha finalizado la ejecución para el fotograma actual, debe actualizar el escenario con los nuevos estados de las geometrías a consecuencia de las deformaciones que el solver aplica sobre los músculos.

El Algoritmo 2 resume las instrucciones que se llevan a cabo en cada fotograma desde el punto de vista de Maya, tanto de preparación previa a la ejecución del solver como de actualización final de la escena. Tal algoritmo se corresponde con la ejecución del método `compute()` del nodo principal que ejercerá de controlador de todo el sistema. Este método es llamado automáticamente por la aplicación en cada cambio de fotograma durante la simulación.

Si nos centramos ahora en el cómputo a nivel de solver, nos referimos precisamente al

conjunto de instrucciones que se lanzan en la línea 20 del Algoritmo 2. El identificador *solver* representa un objeto en C++ plano aislado de Maya que implementa el modelo MSXPBD propiamente dicho. En el Algoritmo 3 se despliega ese conjunto de instrucciones para detallar los cálculos que tienen lugar.

Algorithm 3 Pseudocódigo del cómputo del solver

```
1:  for all músculo m do
2:      for all vértice v de m do
3:          Integrar fuerzas y velocidades de v
4:  for all iteración global i do
5:      for all músculo m do
6:          if i ≤ m.iterations then
7:              for all target t de m do          ▷ Músculos y colisionadores conectados a m
8:                  Actualizar geometría de t
9:              Computar restricciones
10:     for all músculo m do
11:         Actualizar geometría de m
12: for all músculo m do
13:     for all vértice v de m do
14:         Actualizar velocidad de v
15:         Actualizar posición de v
```

El bloque de instrucciones 1-3 del Algoritmo 3 representa la integración numérica de las fuerzas externas (gravedad) y de las velocidades. Aplicando el método numérico de Euler Semi-Implícito, el desglose de esta parte se detalla en el Algoritmo 4. Como aclaración en cuanto a la nomenclatura, téngase en cuenta lo siguiente: $\mathbf{v}_i$ es el vector velocidad del vértice i; $\mathbf{x}_i$ es la posición en el espacio del vértice i al inicio del fotograma; $\mathbf{p}_i$ es la posición en el espacio del vértice i resultante de la integración; $\mathbf{f}_{ext}$ es el vector de fuerzas externas equivalente a la gravedad en el espacio; w_i es la inversa de la masa del vértice i; y Δt es el diferencial de tiempo, es decir, el tamaño de paso del integrador.

Algorithm 4 Pseudocódigo de la integración numérica con Euler Semi-Implícito

```
1: for all m en muscles do
2:     for all vertex i de m do
3:         v_i ← v_i + Δt w_i f_ext(x_i)          ▷ Integración de fuerzas
4:         Amortiguar v_i
5:         p_i ← x_i + Δt v_i                     ▷ Integración de velocidades
```

Completada la integración de fuerzas y velocidades, se obtienen unas posiciones de los vértices que, usando la terminología del modelo XPBD, se denominan predicciones $\mathbf{p}_i$. Las predicciones son las posiciones que las restricciones utilizan para calcular la corrección que es

necesaria aplicar para conseguir llevar el sistema a un estado estable. Este cálculo se realiza en la línea 9 del Algoritmo 3 a nivel de músculo y se denomina *proyección de restricciones*. En el Algoritmo 5 se presenta una generalización de este cómputo, ya que las operaciones concretas para obtener del valor de la restricción (término C en la formulación del modelo, línea 6) difiere dependiendo del tipo de restricción (fibra, volumen, estructura interna, conexiones intermusculares, conexiones con el esqueleto, etc.).

Algorithm 5 Pseudocódigo de la proyección de restricciones en SST.

1: **for all** restricción c **do**
2: **for all** vértice i incluido en c **do**
3: Actualizar $\mathbf{p}_i$ en c
4: Actualizar w_i en c
5: Actualizar datos de conexiones de i con los targets en c
6: Calcular C
7: Calcular $\Delta\lambda_c$ ▷ Multiplicadores de XPBD
8: **for all** vértice i incluido en c **do**
9: Calcular $\Delta\mathbf{p}_i$
10: $\mathbf{p}_i \leftarrow \mathbf{p}_i + \Delta\mathbf{p}_i$
11: $\lambda_c \leftarrow \lambda_c + \Delta\lambda$ ▷ Actualizar multiplicadores de XPBD

Por último, para cerrar el bucle de simulación del solver, las sentencias 12-15 del Algoritmo 3 se ocupan de convertir las predicciones obtenidas tras el cómputo de las restricciones en las nuevas posiciones de los vértices de los músculos, justo antes de actualizar sus geometrías en la escena de Maya. En el Algoritmo 6 se presenta una versión más detallada de estos cálculos.

Algorithm 6 Pseudocódigo de la actualización final de las posiciones de los vértices

1: **for all** m en *muscles* **do**
2: **for all** vertex i de m **do**
3: $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i)/\Delta t$ ▷ Actualizar velocidades
4: $\mathbf{x}_i \leftarrow \mathbf{p}_i$ ▷ Actualizar posiciones con las predicciones
5: **for all** restricción c **do**
6: $\lambda_c \leftarrow 0$ ▷ Reiniciar multiplicadores de XPBD

6.2. Diseño de clases

En este apartado se expone el diseño de clases que implementa el SST. Dada la envergadura del proyecto, se han elaborado diversos diagramas de clases para separar la información y facilitar así su interpretación. En primer lugar, la Figura 6.1 muestra la totalidad de las clases e interfaces C++ que se implementan, incluyendo sus relaciones, herencias y multiplicidad.

En este diagrama se han omitido los miembros y métodos para ofrecer una idea clara de las relaciones presentes en el sistema.

Como ya se expone en la Sección 5.2, el sistema está articulado en tres capas: aplicación, capa intermedia y core. A consecuencia de esto, los detalles a nivel de miembros de las clases se han reservado para las Figuras 6.2, 6.3, 6.4 y 6.5 que se adjuntan más adelante, las cuales describen, respectivamente: los nodos de Maya (capa de aplicación); la especificación de los nodos sensores (capa de aplicación); las interfaces y clases en C++ plano que implementan el modelo MSXPBD (capa core e intermedia); y la especificación de las restricciones de MSXPBD (capa core).

Si nos fijamos, para comenzar, en la parte inferior con fondo naranja de la Figura 6.1, encontramos el nodo principal *SolverNode* que se encarga de instanciar los objetos necesarios para computar MSXPBD además de leer la información del resto de nodos de Maya.

Al *SolverNode* se conectan los músculos que se quieren computar a través de los nodos *MuscleNode*. A su vez, cada uno de éstos se conecta con un *WeightsNode* que define, por vértice, el comportamiento de las conexiones con otros músculos vecinos. El nodo *AttachmentNode* es un tipo de nodo que nos permite identificar puntos de anclaje de cada músculo con otros objetos de la escena, por ejemplo, un hueso. Los sensores *DistanceSensorNode*, *RotationSensorNode* y *LinearSensorNode* se encargan de leer distancias, rotaciones, velocidades y aceleraciones de las articulaciones del esqueleto para calcular los niveles de activación de los músculos asociados a esos movimientos. Cada sensor se conecta con un *TriggerNode* que filtra y limpia sus valores calculados para obtener curvas de activación y relajación configurables.

En lo referente a la capa intermedia (franja verde en la Figura 6.1), las clases *MMuscle* y *MMesh* guardan una relación única entre sí, es decir, todo *MMuscle* almacena su información de geometría en un objeto *MMesh*. Ambas son clases derivadas de sus respectivas interfaces *CMuscle* y *CMesh*. Para aclarar la nomenclatura utilizada, se identifica con «M» inicial la implementación para Maya de la interfaz que tiene el mismo nombre pero con «C» inicial.

Llegando a la capa core (parte superior de la Figura 6.1), hallamos aquellas clases e interfaces que no utilizan ningún método de la API de Maya, es decir, que están totalmente abstraídas de la aplicación y que, por tanto, serían extrapolables a otro software de simulación. La clase *CSolver* ejerce de controlador para el cómputo de MSXPBD, es por eso, que existe una única instancia suya en el nodo principal *SolverNode*. Por cada nodo *MuscleNode* conectado al *SolverNode*, la clase *CSolver* instanciará y almacenará un objeto de tipo *MMuscle*. Asimismo, cada *MMuscle* instanciará y almacenará su *MMesh*, un vector de *CPoint* para representar los puntos de la estructura interna del músculo, un vector de restricciones (en él están incluidos todos los tipos de restricción), un vector de *CAttahcment* (uno por cada *AttachmentNode* conectado al *MuscleNode* correspondiente) y un vector de objetos *ClosestTargetData* que almacenan infor-

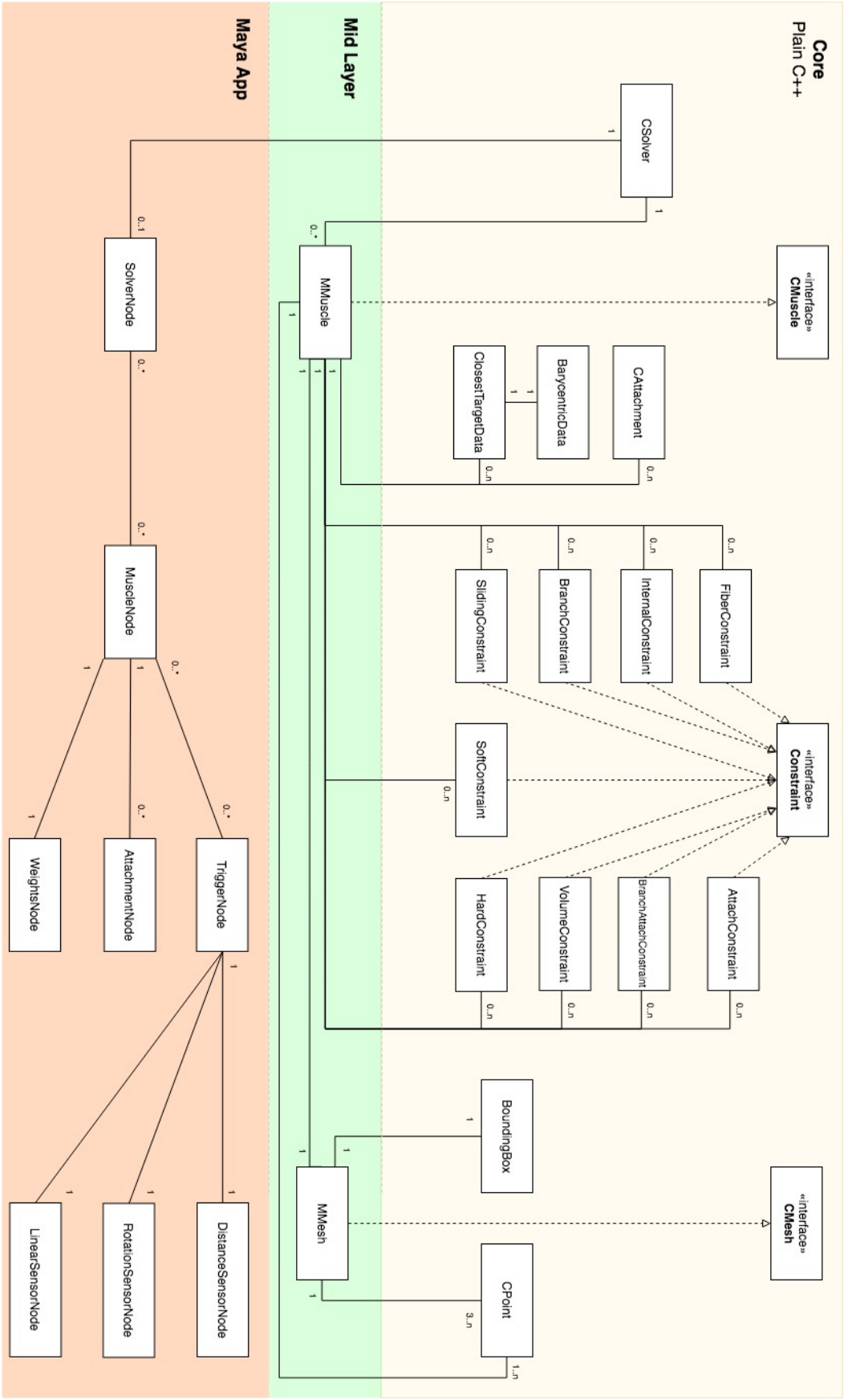

Figura 6.1: Diagrama de clases.

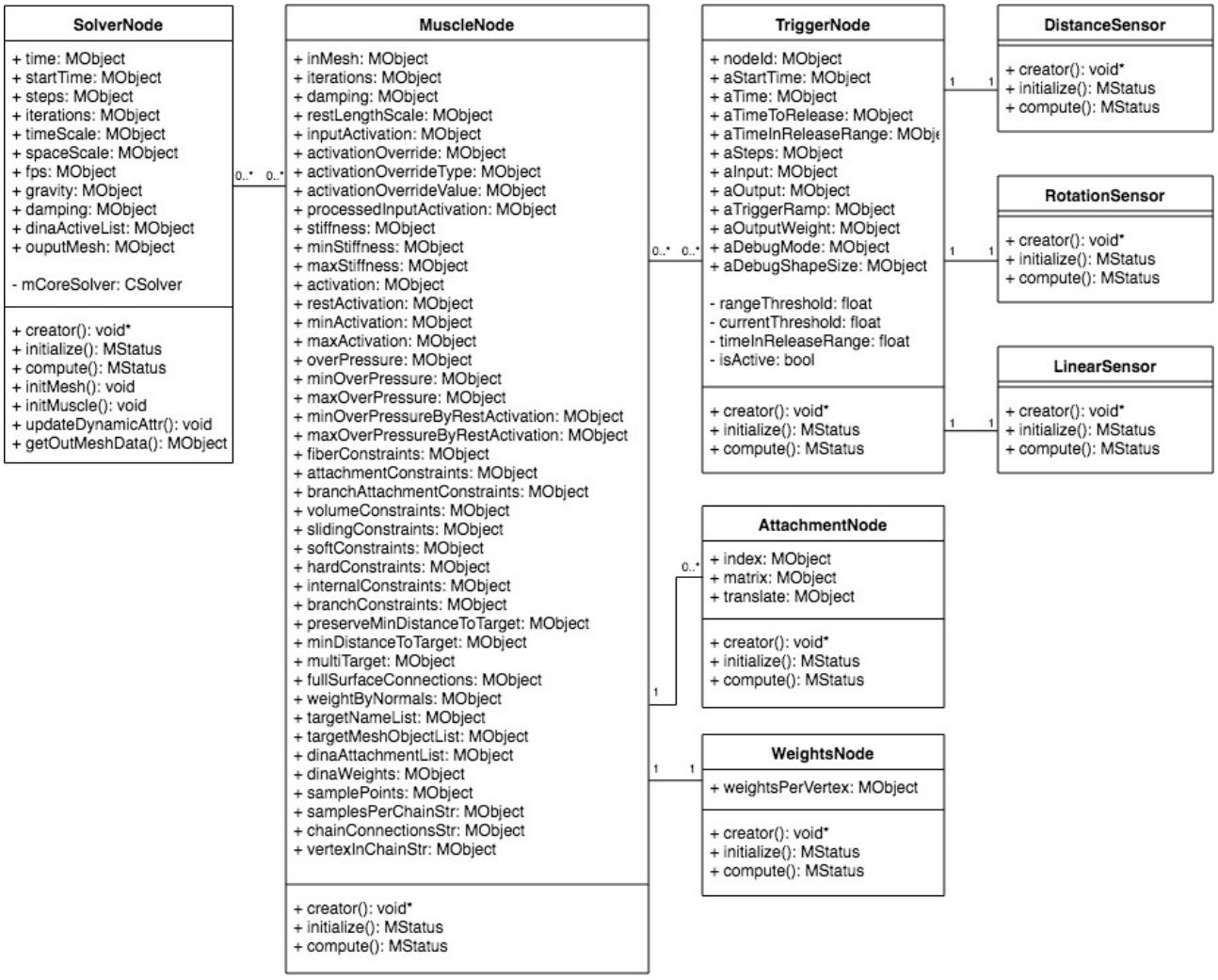

Figura 6.2: Diagrama de clases detallado de la capa de aplicación: nodos de Maya.

mación de geometría de los objetos vecinos (los denominaremos frecuentemente como *targets* y vienen a ser otros músculos incluidos en el solver o también huesos y objetos colisionadores estáticos). Cada instancia *MMesh*, por su parte, almacena un vector de *CPoint* que representan cada uno de los vértices de la geometría. En esta capa core, también encontramos otras clases auxiliares como *BarycentricData* o *BoundingBox* que permiten encapsular datos geométricos para facilitar su manipulación durante el cómputo.

Como se ha adelantado con anterioridad, la Figura 6.2 muestra los atributos y métodos miembro de los nodos de Maya que se van a implementar. Por un lado, los nodos *SolverNode*, *MuscleNode*, *AttachmentNode* y *WeightsNode* heredan de la clase *MPxNode* implementada en la API de Maya [1]. Por otro lado, el nodo *TriggerNode* y los sensores son clases derivadas de *MPxLocatorNode*, también de la API de Maya. La especificación de los sensores *DistanceSensor*, *RotationSensor* y *LinearSensor* se expone en la Figura 6.3.

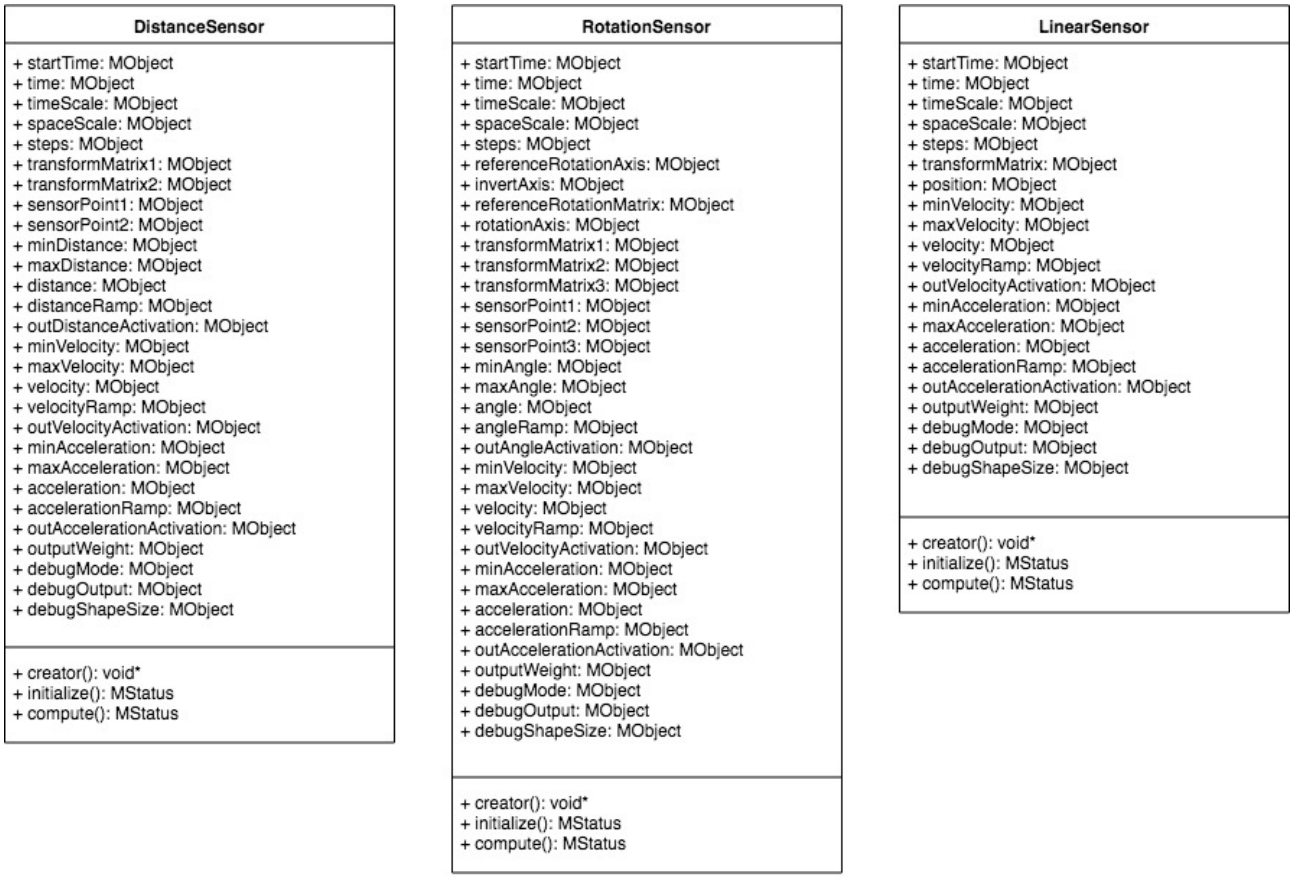

Figura 6.3: Diagrama de clases detallado de la capa de aplicación: sensores.

Las clases que constituyen la capa core y la capa intermedia se exponen en la Figura 6.4. Téngase en cuenta las relaciones de multiplicidad que se recogen en el diagrama para comprender las dependencias entre las clases. Por ejemplo, el objeto único *CSolver* se encarga de almacenar un vector de músculos *MMuscle* que debe computar en cada fotograma.

Por su parte, una instancia de *MMuscle*, la cual implementa los métodos virtuales de *CMuscle*, almacena información sobre la geometría del músculo en forma de objeto *MMesh*. También es importante el vector de *CPoint* para los puntos internos, ya que el modelo MSXPBD precisa de la generación de una estructura interna basada en puntos internos que se conectarán con los vértices de la superficie. Cada objeto *CPoint* guarda información sobre su posición actual, velocidad, aceleración, predicción de la futura nueva posición, dirección de la fibra muscular, etc. Todo músculo, además, almacena información relacionada con los puntos de los targets a los que cada vértice de la *MMesh* se conecta: sus posiciones en la malla del target, la normal, matriz de transformación, coordenadas baricéntricas en el triángulo de la malla, índice de dicho triángulo, etc. Estos datos quedan bien encapsulados en las clases *ClosestTargetData* y *BarycentricData*. Por último, tal y como se ha comentado anteriormente, los *MMuscle* también almacenan un vector de *CAttachment* para mantener actualizada la información de los puntos

en el espacio a los que el músculo permanece fijado.

Por lo que respecta al objeto *MMesh*, cabe destacar que implementa los métodos virtuales de la interfaz *CMesh*. Entre ellos, encontramos métodos cruciales para el cálculo de las interacciones entre músculos y targets que permiten, por ejemplo, la búsqueda de puntos más cercanos, el cómputo de intersecciones o la reconstrucción de la malla actualizada. Cada instancia *MMesh* almacena su vector de puntos *CPoint* que constituyen los vértices de la malla del músculo. También guarda relación con un objeto *BoundingBox* que representa la envolvente del músculo en el espacio y que es muy importante para el cálculo del volumen.

Recordemos que MSXPBD tiene su punto de partida en los modelos de simulación de deformables PBD y XPBD (secciones 2.2 y 2.2.2), y que están basados en restricciones de posición entre los vértices de la malla. Recordemos también que los autores de estos modelos proponen distintos tipos de restricción, por ejemplo, de distancia, volumen, colisiones, *bending*, etc. Romeo et al. en su modelo de simulación de músculos, se ocupan de adaptar la restricción de distancia original además de proponer nuevos tipos que se vuelven necesarios para conseguir que los objetos adquieran el comportamiento deseado. En la Figura 6.5 se muestra la totalidad de las restricciones implementadas, siendo todas ellas derivadas de la interfaz *Constraint*.

La clase *FiberCosntraint* se corresponde, precisamente, con la adaptación de Romeo et al. de la restricción de distancia. Otros tipos basados en la distancia son las clases *InternalConstraint* y *BranchConstraint*. La primera tiene que ver con la conexión entre los puntos internos del músculo y los vértices de la malla. La segunda, computa las distancias entre puntos internos únicamente. De forma análoga, *AttachConstraint* (para vértices de la malla) y *BranchAttachConstraint* (para puntos internos) son restricciones que fuerzan al músculo a mantenerse anclado a los puntos definidos por los *CAttachment* de la Figura 6.4. Como su nombre indica, *VolumeConstraint* implementa la restricción que garantiza la conservación de volumen. En el caso de MSXPBD, se aplica la ganancia de volumen adecuada cuando el músculo se contrae.

Por último, *SlidingConstraint*, *SoftConstraint* y *HardConstraint* son restricciones relacionadas con la interacción del músculo con los targets. Cada una de ellas, aplica un criterio diferente de dónde debe estar un vértice del músculo respecto a un punto en la malla del target correspondiente. Por un lado, *SlidingConstraint* aplica la restricción de distancia original entre un vértice de la malla y un punto cualquiera en la superficie del target, permitiendo que ese punto viaje sobre dicha superficie a lo largo de la simulación para reproducir un efecto de deslizamiento. Por otro lado, *SoftConstraint* aplica la misma restricción de distancia pero mantiene fijo el punto del target al que se conecta, es decir, actúa a modo de muelle. Y por otro, *HardConstraint* fuerza a respetar la matriz de transformación del vértice respecto al punto en el target existente al inicio de la simulación. Para más detalle sobre las restricciones que implementa MSXPBD, consúltese el artículo original de Romeo et al. [16].

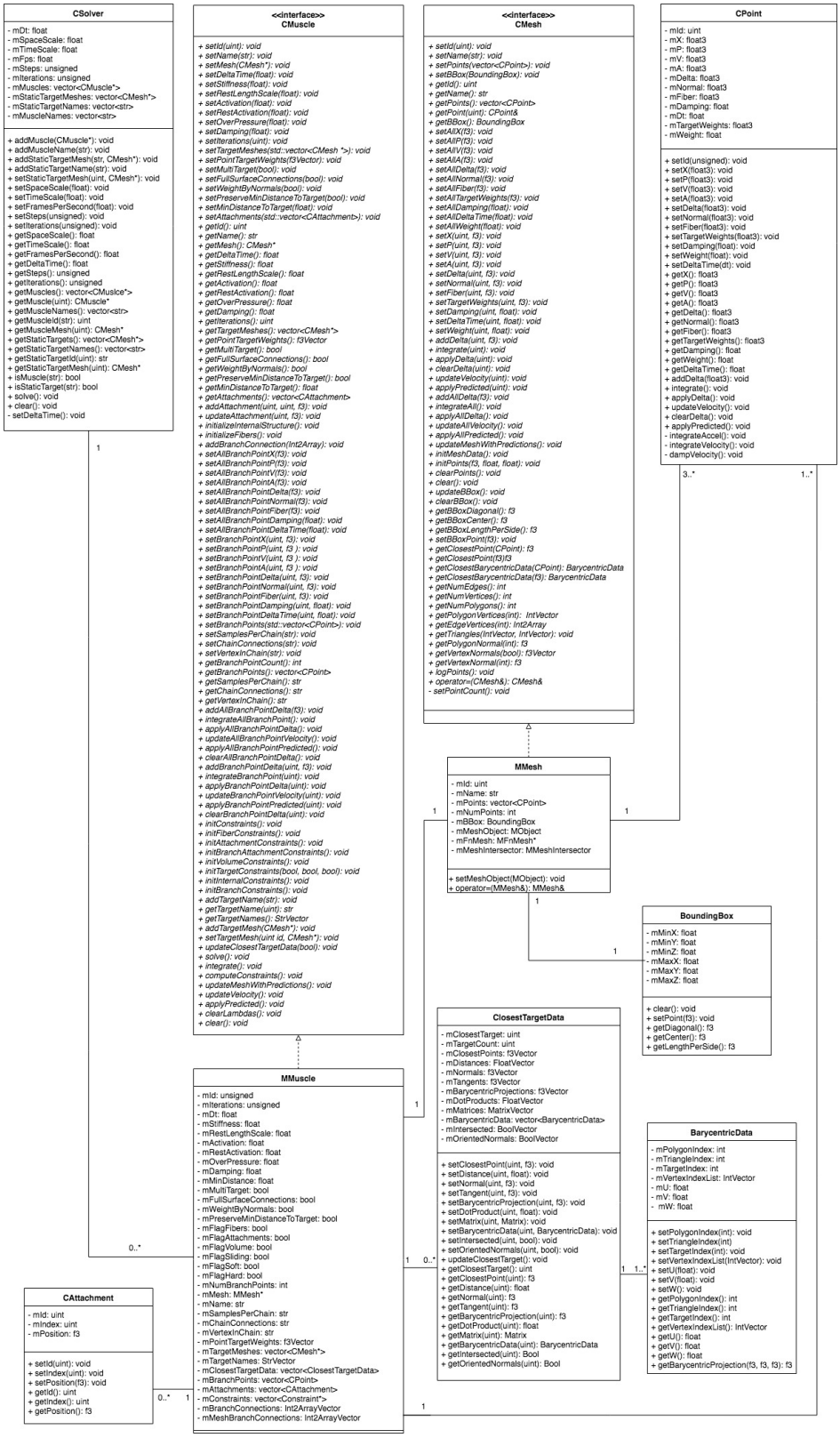

Figura 6.4: Diagrama de clases detallado de las capas core e intermedia.

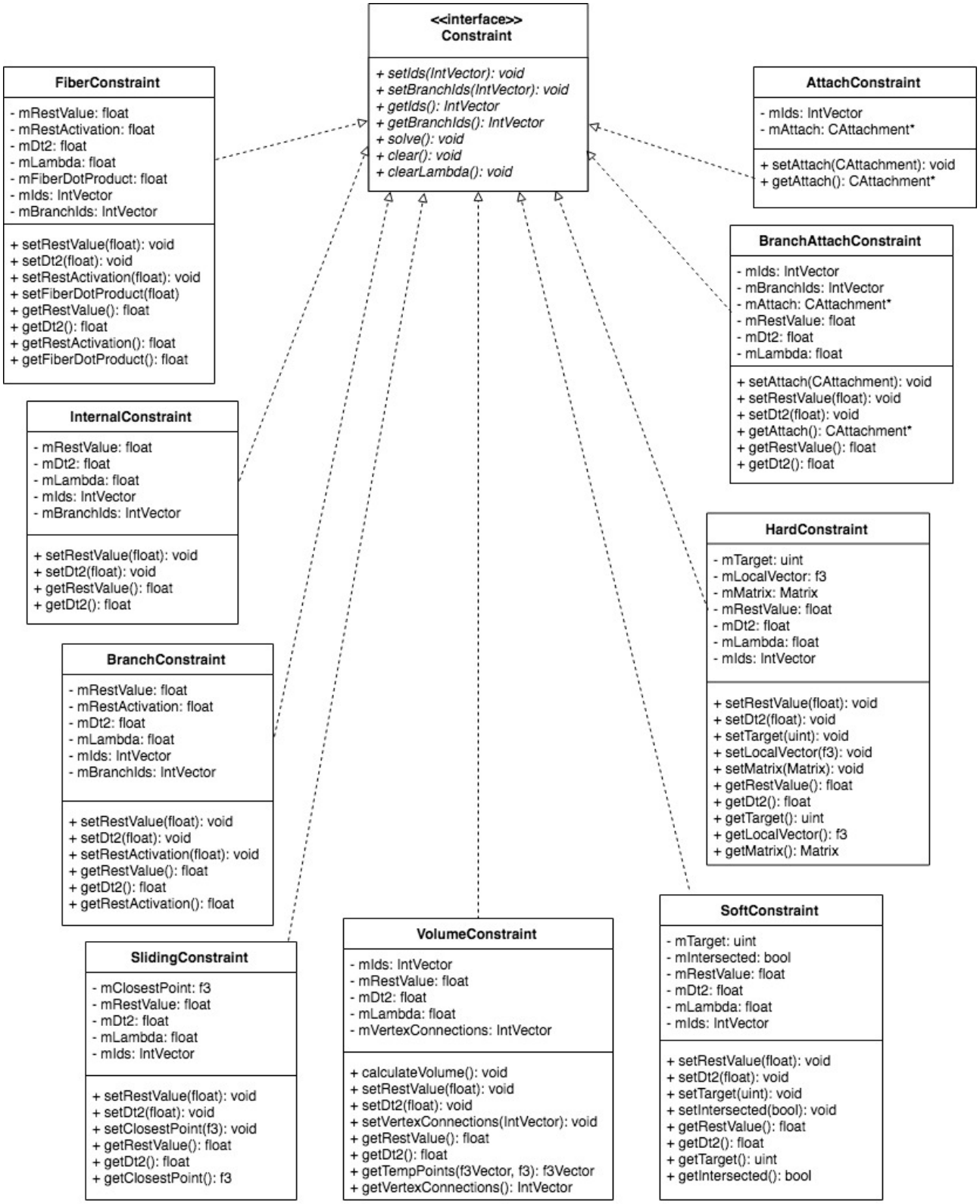

Figura 6.5: Diagrama de clases detallado de la capa core: restricciones.

6.3. Diagrama de secuencia

En esta sección presentamos el diagrama de secuencia del bucle de simulación del SST para mostrar las interacciones en orden secuencial que tienen lugar entre los objetos principales del sistema. El diagrama de la Figura 6.6 reproduce el paso de mensajes entre dichos objetos y ofrece una versión gráfica de los Algoritmos 3, 4, 5 y 6.

El nodo *SolverNode* da el control al *CSolver* para la ejecución del bucle y lo primero que se computa es la integración de fuerzas y velocidades con la llamada `integrate` que el solver hace por cada *MMuscle*. Esta integración tiene lugar en los puntos de la malla, es decir, en los objetos *CPoint* que representan los vértices de la geometría, y también en los objetos *CPoint* que representan la estructura interna del músculo. Por este motivo, *MMuscle* ejecuta una primera llamada de `integrateAll` a su instancia *MMesh* y una segunda llamada *integrateAllBranchPoints* para los puntos internos. Cada *CPoint* completa la integración en tres pasos: `integrateAccel`, `dampVel` e `integrateVel`.

Seguidamente, se inicia el bucle de iteraciones del modelo MSXPBD. En cada iteración, el primer paso consiste en la petición a cada músculo por parte del solver de sus targets. Es el propio solver el que se encarga de mandarles la información de geometría actualizada de esos targets.

Llegados a este punto, el sistema está preparado para ejecutar la proyección de restricciones, que se inicia con el mensaje `computeConstraints` que el solver manda a cada músculo. A modo de pre-cálculo, los músculos actualizan su *bounding box* y la información de puntos en la malla de los targets a los que cada vértice se conecta mediante las restricciones externas de sliding, soft y hard. Por cada restricción, se realiza la llamada a su método `solve` y se actualizan los *CPoint* afectados por esa restricción. La actualización consiste en añadir la deformación calculada por la restricción (`addDelta`), aplicar el delta acumulado sobre las predicciones (`applyDelta`) y limpiar el vector delta para la siguiente iteración (`clearDelta`). Del mismo modo que en la integración numérica del inicio, este paso de actualización de puntos tiene lugar en dos niveles: vértices de la malla y puntos de la estructura interna.

Finalizado el cómputo de las restricciones, se da paso a la actualización de la geometría del músculo. El solver lanza el método `updateMeshWithPredictions` de cada músculo, y cada músculo hace lo propio con su *MMesh*. A continuación, se actualiza la velocidad de los puntos y se aplican las predicciones como nuevas posiciones. De nuevo, estos dos pasos se procesan para los vértices de la malla y también para los puntos de la estructura interna.

El último paso antes de cerrar el bucle de simulación es el reinicio de los multiplicadores de XPBD en cada restricción. Esto se realiza con la llamada a `clearLambdas` que cada músculo hace sobre sus restricciones.

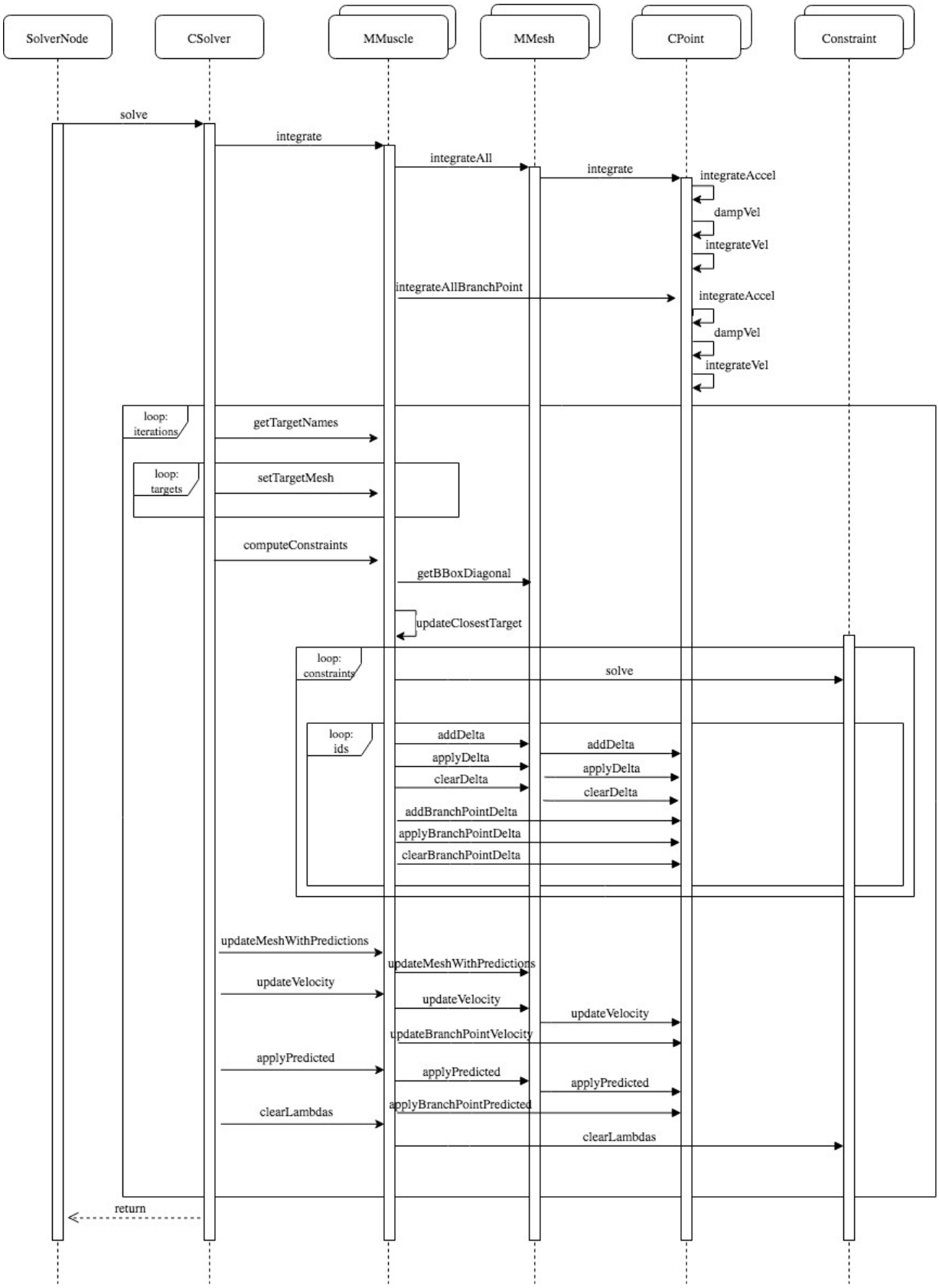

Figura 6.6: Diagrama de secuencia del bucle de simulación.

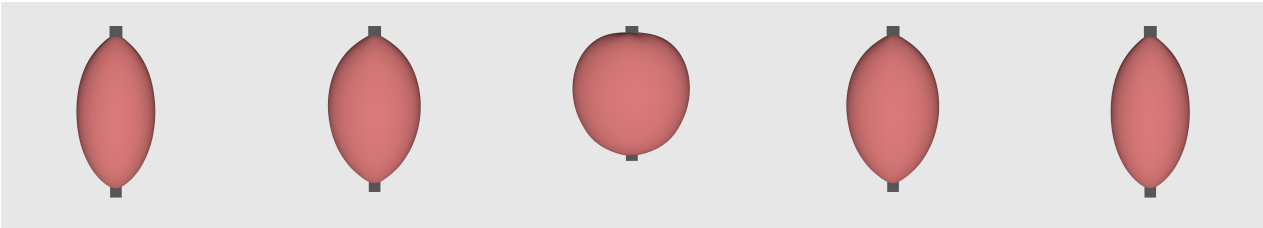

Figura 6.7: Activación y relajación de un músculo simple con SST (T1).

6.4. Análisis de resultados

En esta sección, vamos a enfocar el análisis de resultados desde dos puntos de vista. En primer lugar, en el apartado 6.4.1 mostraremos los resultados a nivel gráfico mediante imágenes extraídas de las simulaciones realizadas en los distintos escenarios de Test propuestos en la Metodología. Del mismo modo y en segundo lugar, en el punto 6.4.2 estudiaremos los resultados en términos de coste de computación en los diferentes casos de Profiling.

6.4.1. Calidad de la simulación

Para extraer conclusiones sobre la calidad de la simulación se han ejecutado los test T1 a T7 (ver Sección 5.1). Comenzando por T1, en la Figura 6.7 se muestran cinco fotogramas extraídos de la simulación. Se puede observar, de izquierda a derecha, que el músculo con forma de óvalo está relajado y comienza a activarse alcanzando la máxima contracción en el fotograma central. A partir de ahí, comienza a relajarse hasta que recupera la forma inicial de reposo. Atendiendo a estos resultados, podemos decir que la implementación del SST es correcta en tanto que se consigue una simulación estable donde el músculo se activa y se relaja según lo esperado.

El T2 consiste en la simulación de un músculo anatómicamente realista, en este caso, el bíceps. La Figura 6.8 recoge tres fotogramas durante la activación (fila superior) y tres fotogramas de la relajación (fila inferior). Del mismo modo que en el test anterior, se demuestra que la implementación ofrece resultados satisfactorios también cuando se trata de geometrías más complejas.

Por lo que respecta al T3, se pone a prueba el sistema de interacción entre los músculos computados por el solver y las geometrías externas que ejercen como colisionadores. En esta simulación, concretamente, se consideran los huesos del esqueleto como targets colisionadores del bíceps anatómico. La Figura 6.9 muestra los resultados de la prueba desde una perspectiva lateral (arriba) y otra frontal (abajo). Como conclusión, se puede afirmar que las restricciones externas que computan las relaciones músculo-esqueleto están cumpliendo su función.

La Figura 6.10 presenta los resultados del T4. En este test se pretendía comprobar la estabilidad del sistema a la hora de simular múltiples músculos al mismo tiempo y se ha

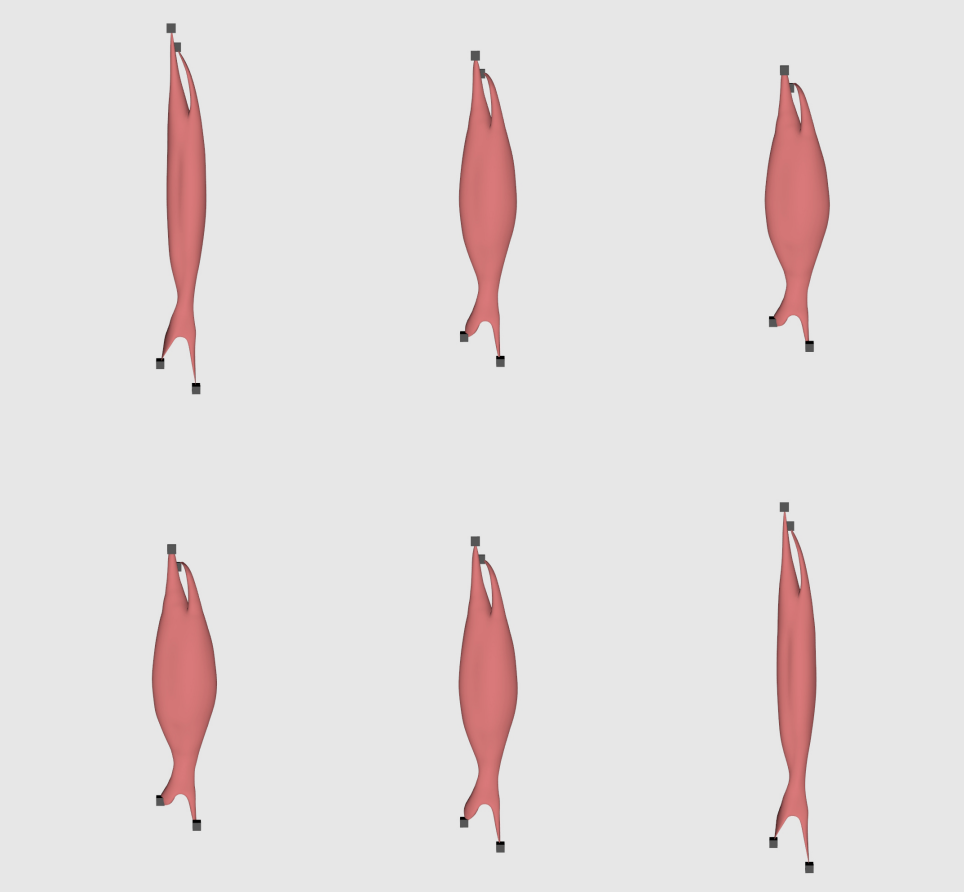

Figura 6.8: Activación y relajación del bíceps anatómico con SST (T2).

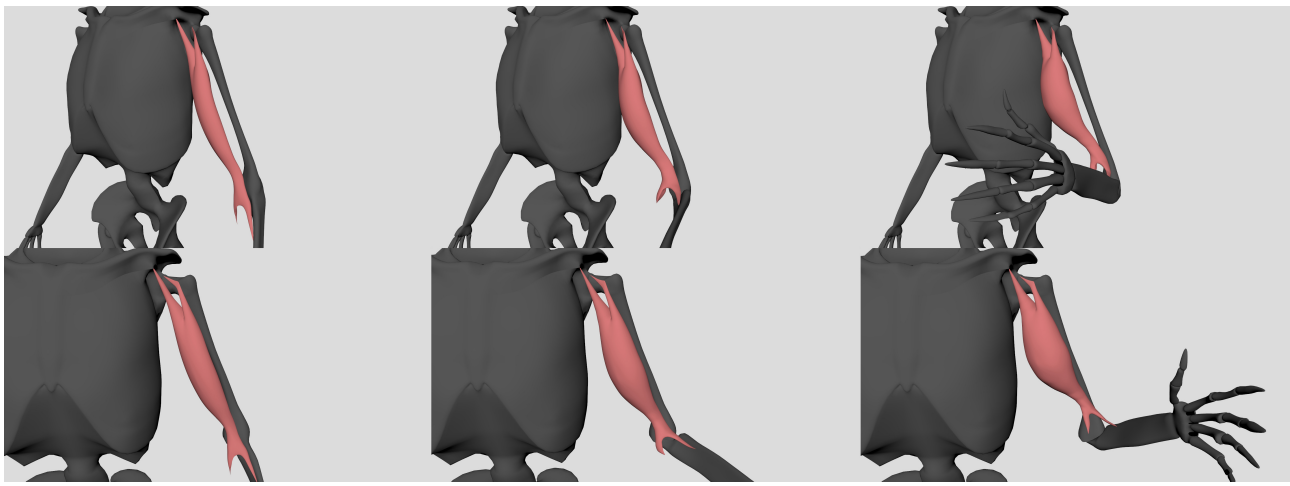

Figura 6.9: Activación y relajación del bíceps anatómico conectado al esqueleto con SST (T3).

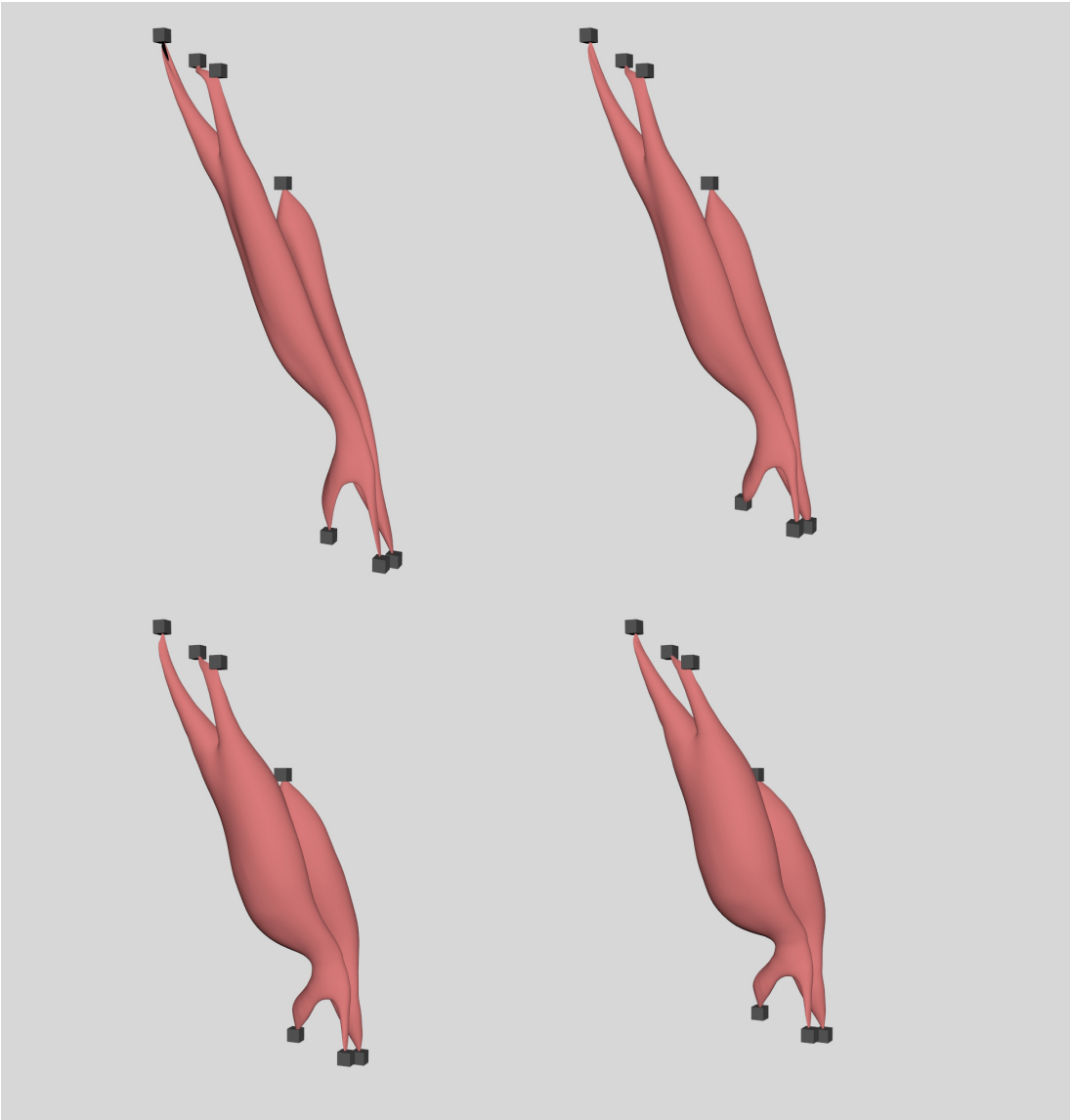

Figura 6.10: Activación de bíceps y brachialis anatómicos conectados entre sí con SST (T4).

observado que el cómputo de MSXPBD en nuestra versión del SST ofrece buenos resultados. Por otra parte, el test también ha servido para confirmar que la restricciones externas músculo-músculo funcionan correctamente.

En el T5 se simula la flexión del codo con todos los músculos del brazo, los cuales se contraen (ver en la Figura 6.11) debido a los sensores de activación a los que están conectados. Cabe destacar la ganancia de volumen y la forma que adopta el bíceps en el momento de máxima flexión. A tenor de estos resultados, se puede decir que el sistema de activación y la interacción inter-muscular y con el esqueleto mantienen coherencia a pesar del aumento en la complejidad de la escena.

En T6 se pone a prueba el SST en términos de estabilidad y convergencia cuando la anima-

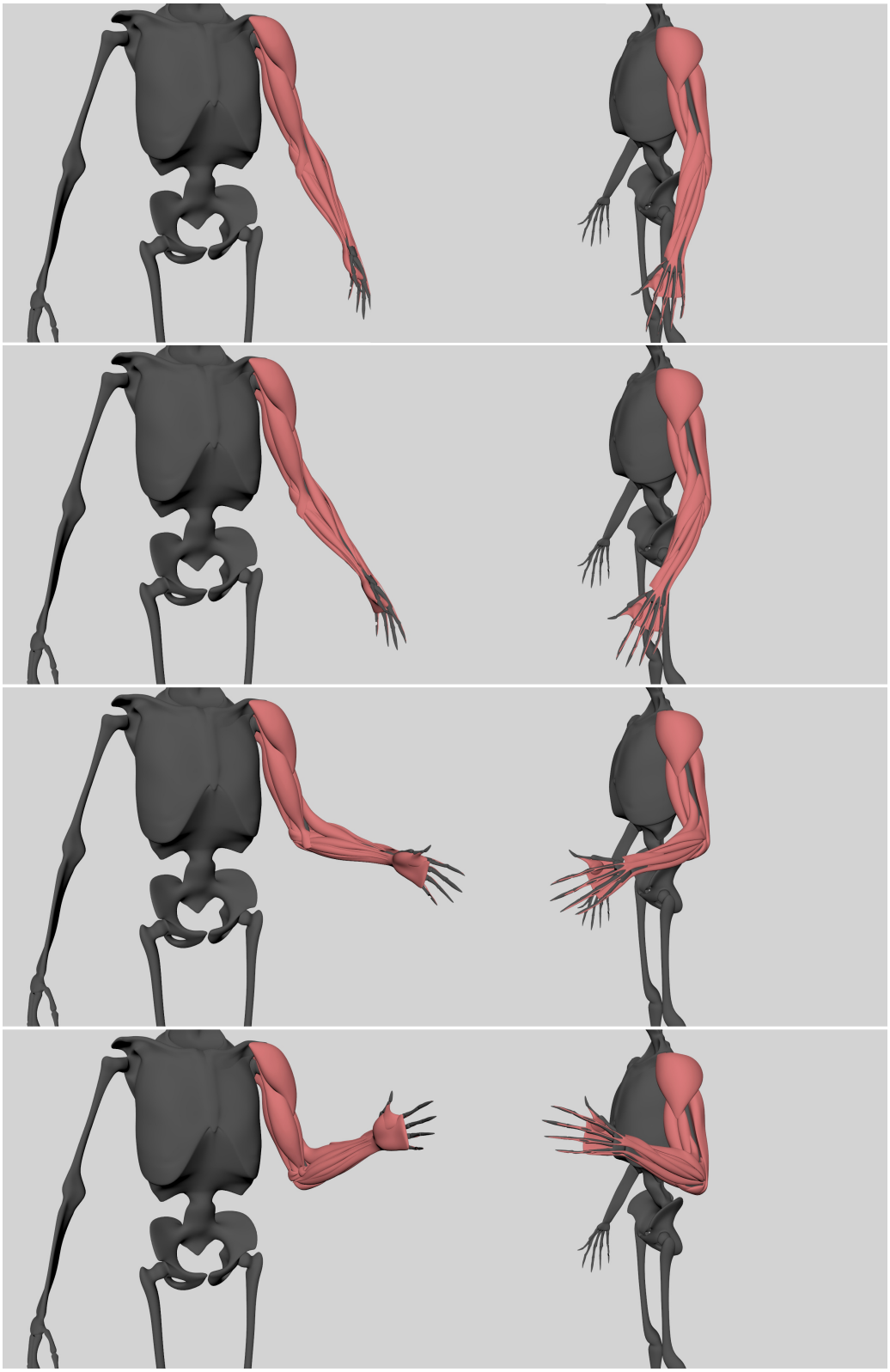

Figura 6.11: Simulación del brazo completo durante la flexión del codo con SST (T5).

ción del personaje realiza movimientos rápidos y con mucho desplazamiento. Éste es un punto crítico a la hora de validar un sistema de simulación como el que nos ocupa, ya que para ser aplicable debe garantizar resultados aceptables en entornos llevados al límite. Las diferentes capturas que se muestran en las Figuras 6.12 y 6.13 demuestran que los músculos se mantienen compactos y sujetos tanto al esqueleto como a sus vecinos. Los niveles de activación también manifiestan una correcta activación dependiendo de las articulaciones del hombro y el codo.

Por último, se ha simulado el cuerpo completo del personaje 3D tal y como describe el test T7 en la Metodología: 6 procesos independientes para separar brazos, piernas y torso, y a su vez en lados derecho e izquierdo. Si observamos algunos fotogramas de la simulación obtenida (ver Figura 6.14 con cámara frontal y 6.15 con cámara trasera), se demuestra la estabilidad de la implementación y la correcta interacción entre los músculos.

En algunos fotogramas de T7, se ha podido ver una falta de rigidez en determinadas zonas del cuerpo, por ejemplo, en los músculos isquiotibiales (ver fotograma abajo-derecha de 6.15). Hay que tener en cuenta que todas las simulaciones aquí realizadas se han completado con la configuración de los músculos por defecto, es decir, no se han manipulado los valores de los nodos *MuscleNode* que se crean. Esto significa que existe un alto grado de mejora en el comportamiento si se dedica un tiempo a editar esos atributos. De este modo, se podría evitar el exceso de relajación comentado mediante el aumento de la rigidez, el número de iteraciones, el peso de las conexiones sliding, soft y hard o modificando los rangos de activación y sobrepresión.

6.4.2. Cálculo de Costes

En este apartado se exponen los resultados de costes de computación en las simulaciones definidas en la Metodología como P1, P2, P3 y P4. Las Figuras 6.16, 6.17, 6.18 y 6.19 recogen, respectivamente, los resultados de la medición de tiempos en esos casos de test. Cada bloque de instrucciones (BE) de los escogidos en la Sección 5.1 tiene su columna vertical correspondiente que refleja el coste de cómputo medio respecto al coste total medio de un fotograma, en valores normalizados $\in [0, 1]$.

La conclusión más evidente que podemos extraer al analizar los resultados es que *Closest Data* (BE5) es la tarea que mayor impacto tiene sobre el coste total en los cuatro escenarios, alcanzando valores de 0,55, 0,62, 0,48 y 0,70 respectivamente. Recordemos que esta tarea hace referencia al proceso de actualización de la información necesaria para las conexiones músculo-músculo y músculo-esqueleto. El proceso, además, lo realiza cada vértice de cada músculo en cada iteración del solver y por cada uno de los targets conectados. Tratándose de un bloque de código que se ejecuta muchas veces por fotograma y unido a que los cálculos implicados son costosos en sí mismos (por ejemplo, dado un punto en el espacio, buscar el punto más cercano en la superficie de una geometría), estos resultados son entendibles.

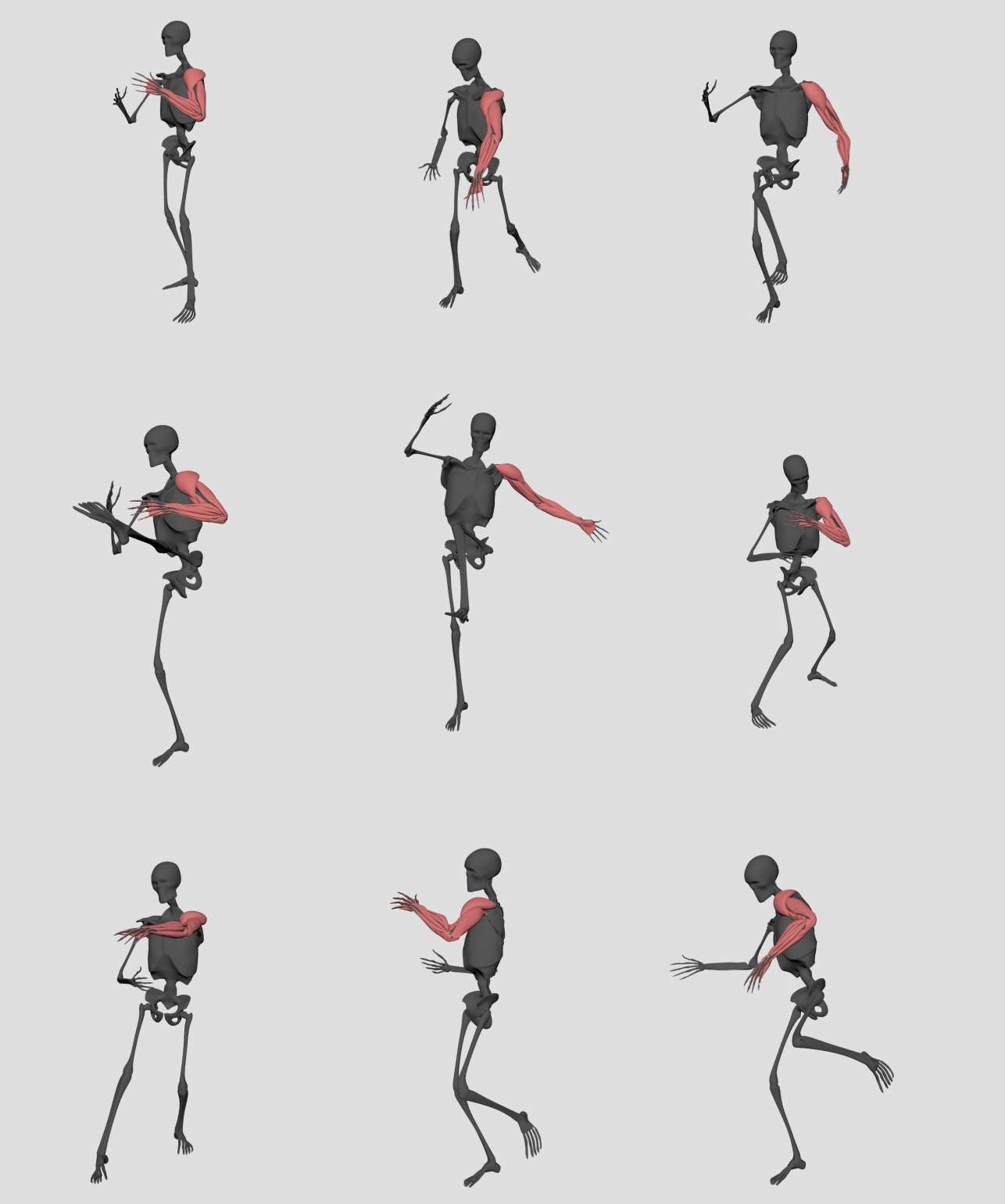

Figura 6.12: Simulación del brazo completo sobre movimientos de lucha con SST (T6).

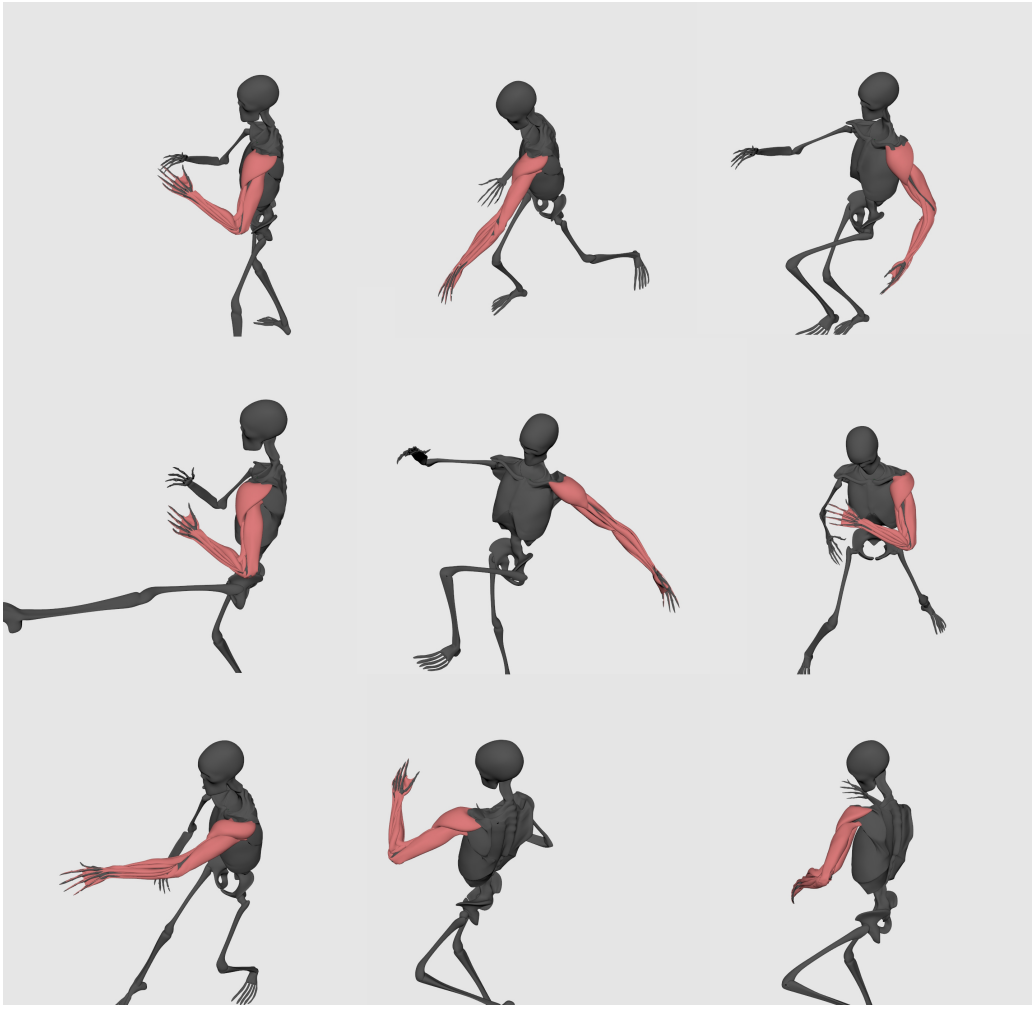

Figura 6.13: Perspectiva detalle de la simulación del brazo completo sobre movimientos de lucha con SST (T6).

La diferencia de coste de BE5 respecto al resto de tareas es muy acentuada, sobretodo en P4, debido a que en este escenario hay mayor complejidad de targets ya que cada músculo puede llegar a conectarse hasta con un total de 6 targets, entre músculos vecinos y huesos. Esto implica que en cada iteración del solver, si un músculo tiene 6 targets por ejemplo, cada uno de sus vértices tiene que calcular 6 veces: el punto más cercano en la malla del target, índice del polígono, índices de los vértices en el polígono, vectores normal, tangente y binormal, si hay intersección o no y la matriz de transformación del vértice respecto al punto en el polígono.

La segunda tarea con mayor coste es la *Proyección de Restricciones*, BE6, con valores de 0,22, 0,19, 0,26 y 0,15. Los cálculos que aquí se realizan tienen que ver con el núcleo del modelo de simulación MSXPBD, que se ocupa de calcular las correcciones de posición de todos los vértices del sistema. Evidentemente, a mayor número de restricciones, mayor coste asociado en términos absolutos. Pero lo que vemos en las Figuras 6.16, 6.17, 6.18 y 6.19 son costes relativos

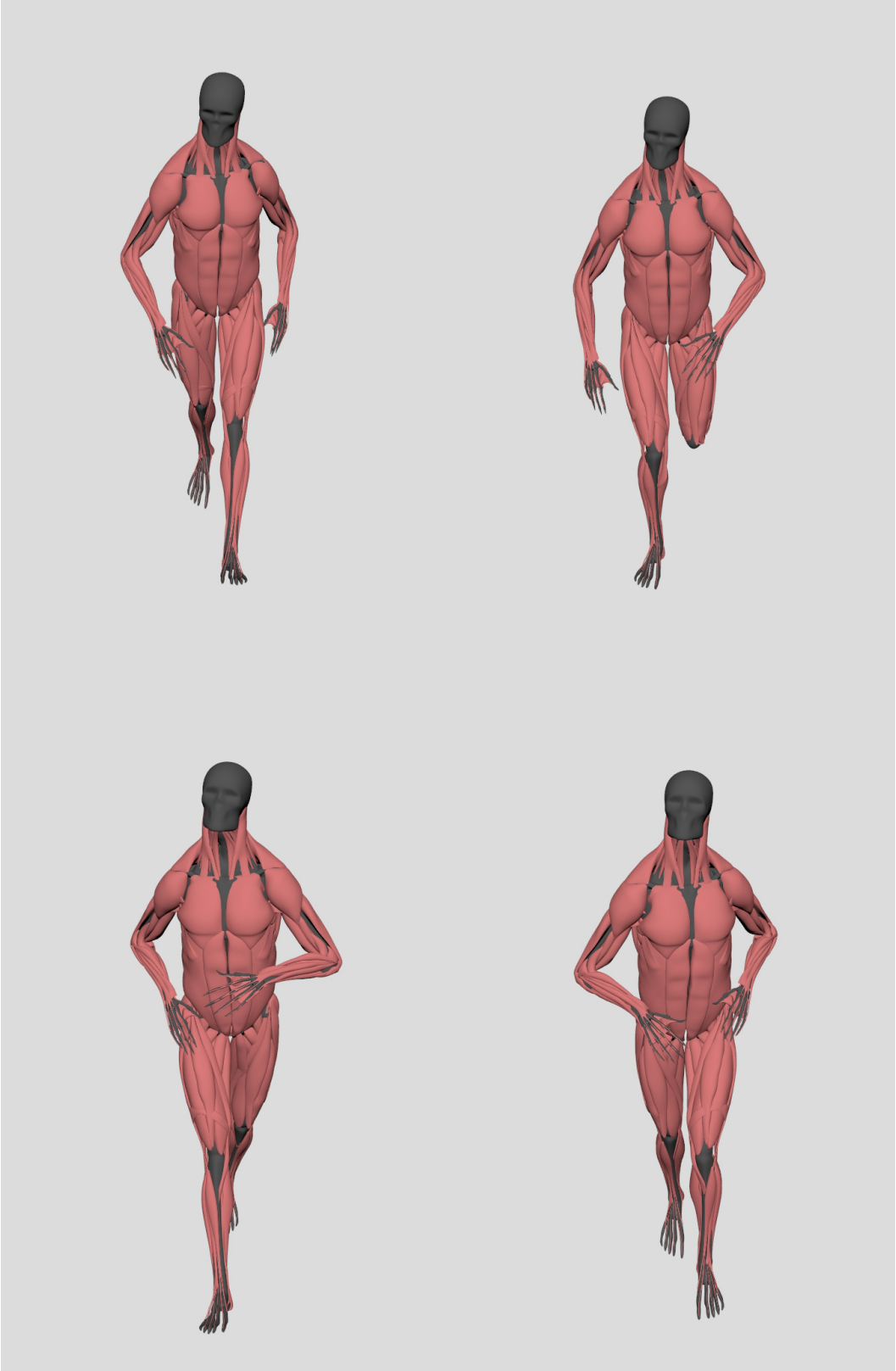

Figura 6.14: Simulación del cuerpo completo caminando y corriendo con SST (T7).

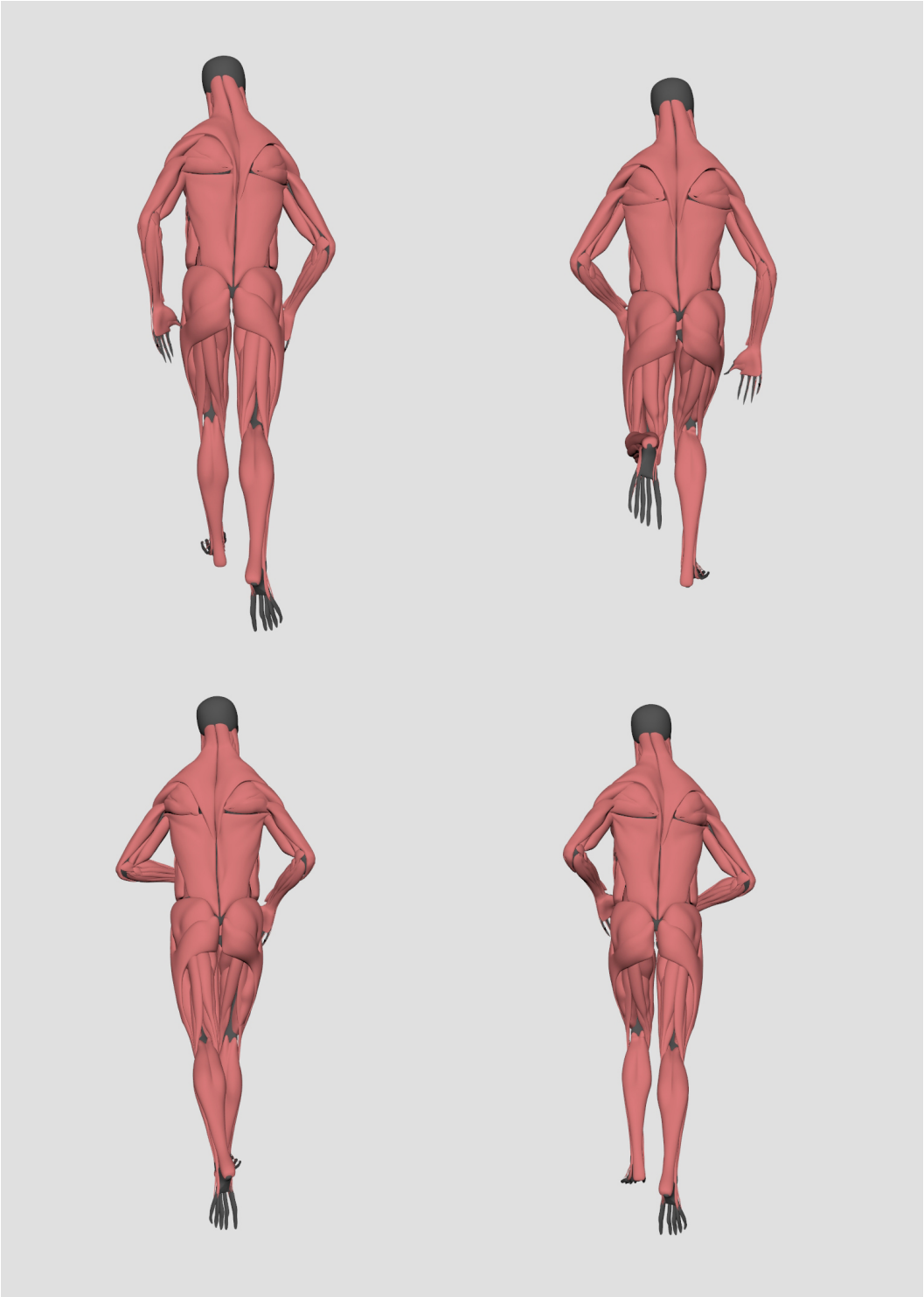

Figura 6.15: Simulación del cuerpo completo caminando y corriendo con SST (T7).

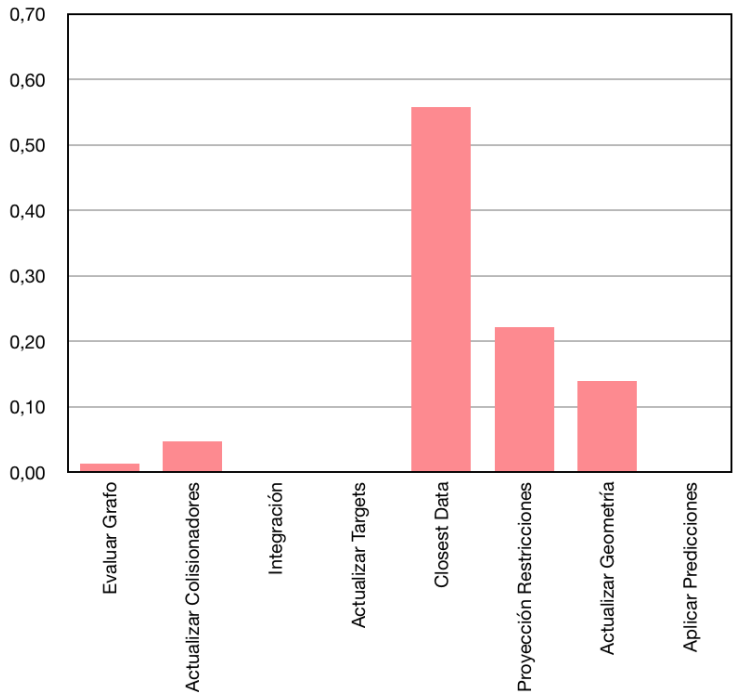

Figura 6.16: Coste medio por bloque de tareas por fotograma con SST en P1.

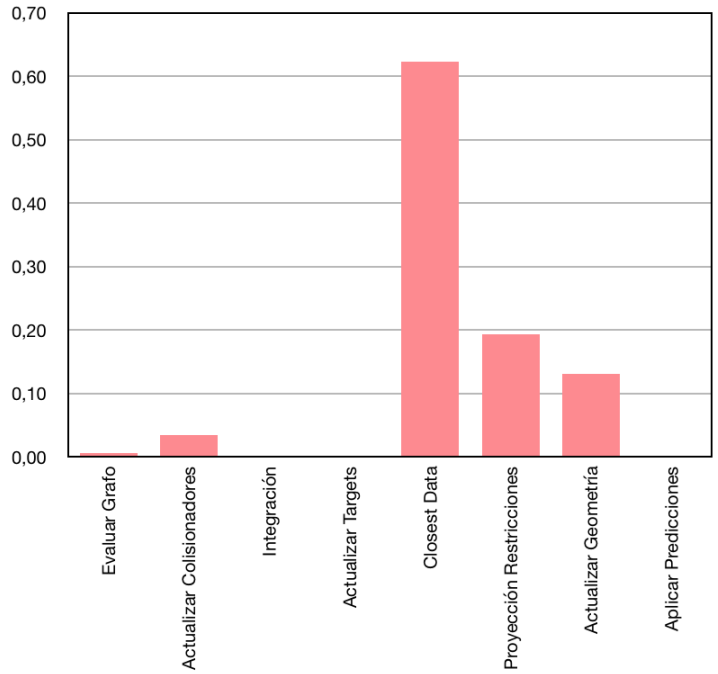

Figura 6.17: Coste medio por bloque de tareas por fotograma con SST en P2.

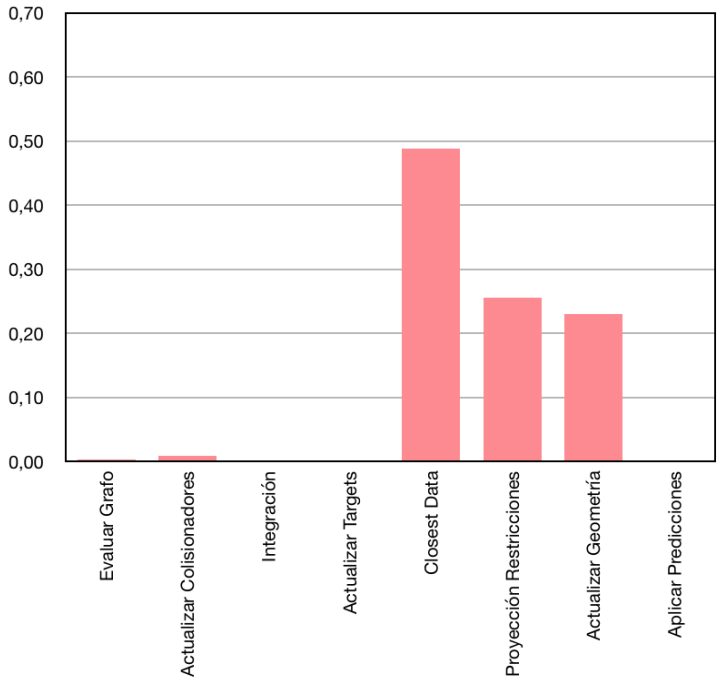

Figura 6.18: Coste medio por bloque de tareas por fotograma con SST en P3.

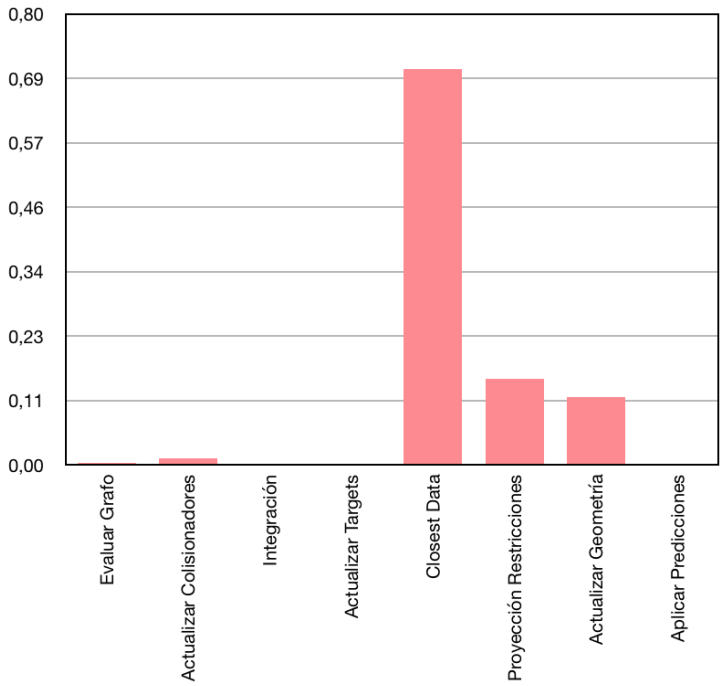

Figura 6.19: Coste medio por bloque de tareas por fotograma con SST en P4.

Profiling	Restricciones	Tiempo (ms/f)
P1	571	11,32
P2	1040	24,57
P3	4035	71,49
P4	18567	531,92

Tabla 6.1: Tiempos absolutos en milisegundos por fotograma del SST en P1, P2, P3 y P4.

respecto al coste total del fotograma. Por este motivo, las columnas de BE6 se mantienen con alturas similares, ya que la complejidad de los casos P1 a P4 penaliza más a BE5.

Otros bloques de tareas con cierta representación en las gráficas son *Actualizar Colisionadores* (BE2) y *Actualizar Geometría* (BE7). En ambos casos, hablamos de tareas que tienen que ver con la capa de aplicación: la primera se ocupa de leer información geométrica en la escena y empaquetarla para enviarla al core; y la segunda es su opuesta, ya que se encarga de volcar en las geometrías de la escena la nueva disposición de los vértices al finalizar cada iteración del solver. Tiene sentido que estas tareas afecten al tiempo por fotograma puesto que los cálculos implicados aquí llevan consigo la lectura y escritura de puntos en Maya, la comunicación entre la capa de aplicación y la capa core, y la consecuente necesidad de traducción y conversión de tipos de datos.

En resumen, los cuatro bloques comentados anteriormente (BE5, BE6, BE2 y BE7), acaparan casi el total del tiempo de cálculo por fotograma, forzando a que el coste relativo de los restantes bloques de tareas analizados sea prácticamente despreciable.

Si hacemos un breve repaso al tiempo total medio por fotograma de los cuatro escenarios sometidos a test, podemos ver la Tabla 6.1 donde se relaciona el tiempo medio por fotograma (milisegundos) en cada uno de los test sometidos a profiling respecto al número de restricciones presentes en el sistema. Como era de esperar, el tiempo absoluto de cómputo crece a medida que la complejidad del sistema aumenta. No obstante, este aumento de tiempo no es estrictamente lineal. En la Figura 6.20 se ha realizado una regresión polinomial cuadrática para ofrecer una versión más gráfica de estos datos.

6.4.3. Estudio comparativo con el modelo original

En este apartado vamos a hacer una breve comparativa de tiempos de cómputo entre el sistema single-threaded implementado en el capítulo y el modelo original MSXPBD. Según lo expuesto en el Estado del Arte 2.3, Romeo et al. presentan un sistema implementado en Python sin paralelización. Los datos de coste que exponen afirman que consiguen resolver el cuerpo humano completo en 44,04 segundos por fotograma dividiendo la simulación en 6 procesos (brazos, torso y piernas, divididos a su vez en lado izquierdo y lado derecho cada uno).

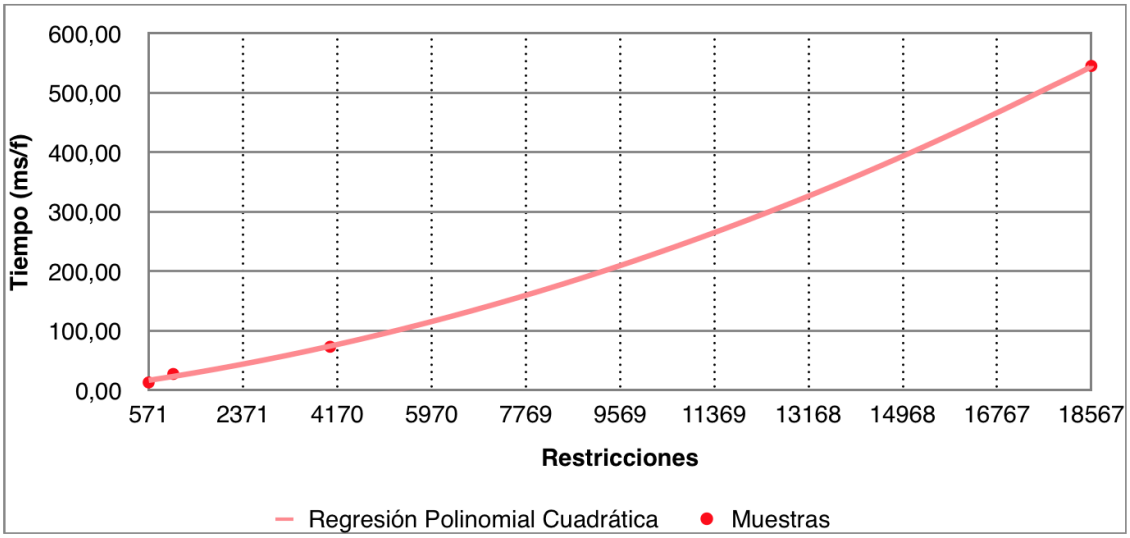

Figura 6.20: Ajuste polinomial del coste de SST en P1, P2, P3 y P4.

Su configuración da lugar a un total de 135780 restricciones con 10 iteraciones en el solver. En unidades de frames por segundo, los tiempos de simulación del sistema se traducen a 0,023 fps.

Para comparar nuestro SST en C++ single-threaded se ha dispuesto de una simulación del humano 3D también separada en las mismas 6 secciones con 10 iteraciones, generando 127270 restricciones, un número ligeramente inferior al del caso de Romeo et al. El tiempo medio de coste obtenido ha sido de 1,414 segundos por fotograma, dicho de otro modo, una tasa de refresco de 0,707 fps. Si bien nuestra cantidad de restricciones es menor (se reduce en un 7 %), la ganancia en velocidad de cálculo es mucho mayor, entorno a 30 veces más rápida. Esto significa que el diseño e implementación desarrollados para la traducción del modelo de Python a C++ han sido muy satisfactorios en términos de costes.

Así y todo, estas comparaciones no tienen en cuenta las diferencias de prestaciones hardware, cosa que resta méritos a los resultados obtenidos en este proyecto debido a que nuestros experimentos están realizados en una máquina considerablemente menos potente que la que utilizan Romeo et al. en su artículo. Su equipo de pruebas es un Intel Core i7-4790K (4,00GHz) con 32GB de RAM, frente a nuestro Intel Core i7 (2,3GHz) con 8GB de RAM. Esto acentúa, todavía más si cabe, la valoración positiva de la implementación realizada en esta sección.

Capítulo 7

Sistema Multi-Threaded

En este capítulo presentamos el análisis, diseño, implementación y resultados del *Sistema Multi-Threaded*, en adelante SMT. El desarrollo de dicho sistema encaja dentro del Bloque 3 de tareas presentado en la Sección 5.4. Al comienzo del capítulo, se exponen las decisiones tomadas en cuanto al diseño de la versión paralelizada de MSXPBD en base a los resultados obtenidos en el apartado 6.4.2. Seguidamente, tal y como se adelanta en la Metodología 5.1, se exponen las versiones de SMT que aplican el Método Jacobi y el Método de Coloreado de Grafos. Al término, se recogen los resultados de las simulaciones y los tiempos de cálculo para compararlos entre sí y con el SST original.

7.1. Análisis de la paralelización

El análisis que presentamos aquí fundamenta sus premisas en los tiempos de coste del SST recopilados y comentados en el apartado 6.4.2. A continuación, se exponen las decisiones y estrategias de optimización que se llevarán a cabo sobre los puntos críticos de la implementación secuencial.

La primera idea que cabe resaltar es que, de los bloques de ejecución sometidos a estudio en los test de profiling, existen cuatro tareas cuyos ratios de coste respecto al total son muy pequeños. Tanto es así, que prácticamente no tienen representación en las gráficas 6.16, 6.17, 6.18 y 6.19. Estamos hablando de las tareas catalogadas como *Evaluar Grafo*, *Integración*, *Actualizar Targets* y *Aplicar Predicciones* (BE1, BE3, BE4 y BE8, respectivamente). Atendiendo a estos datos, a pesar de considerarse susceptibles de ser optimizadas en el estudio preliminar de la Sección 5.2, en las implementaciones del SMT que realizaremos se descarta la introducción de directivas de paralelización en dichas tareas, esto es, seguirán tratándose como bloques de ejecución secuencial. Por tanto, las técnicas de optimización se centrarán en reducir los tiempos de ejecución de BE5, BE6, BE2 y BE7.

El principal bloque a optimizar es BE5, a saber, *Closest Data*. Según los datos obtenidos en las mediciones sobre el SST, observamos que claramente se corresponde con el cuello de botella del sistema. Debido a la naturaleza de MSXPBD en el que los músculos colaboran e interaccionan entre sí para adquirir un comportamiento realista, la magnitud del problema crece conforme aumenta la cantidad de músculos en la escena de simulación. Por tanto, la optimización de BE5 es primordial para conseguir una mejora importante en el SMT respecto al SST, ya que el objetivo final es ser capaces de simular escenas con elevado número de músculos. La idea fundamental para optimizar esta parte se puede resumir en conseguir que se realicen los cálculos implicados en el mayor número de vértices al mismo tiempo, es decir, paralelización a nivel de vértice.

El segundo bloque crítico del sistema es la *Proyección de Restricciones*, BE6. Estamos hablando del núcleo del modelo XPBD (y más concretamente, de MSXPBD) cuya optimización se llevará a cabo mediante la aplicación de los métodos Jacobi y Coloreado de Grafos, extraídos de la literatura relacionada (ver Estado del Arte 2.2.1). Los enfoques de estos paradigmas son muy distintos y llevan consigo cambios estructurales de código. Es por eso que especificaremos las estrategias de cada uno de ellos por separado dando lugar, en realidad, a dos versiones independientes del *Sistema Multi-Threaded*.

Continuando con los bloques de ejecución a paralelizar, llegamos a BE2 que tiene que ver con la lectura y envío de información geométrica de los objetos colisionadores desde la aplicación al core. El coste de esta tarea crece conforme aumenta el número de objetos estáticos (no músculos) en el escenario. La premisa para optimizar esta parte se basa en que la recogida de los datos sea concurrente para todos los colisionadores. Dicho de otro modo, paralelizar a nivel de objeto.

Concluyendo con el conjunto de tareas a optimizar, el bloque BE7 se encarga de la actualización y escritura de las nuevas posiciones de los vértices en la geometría de la capa de aplicación. Este bloque de instrucciones es ejecutado por el solver, de modo que la estrategia de paralelización es a nivel de músculo.

7.2. Diseño e implementación

En la presente sección vamos a unificar las fases de diseño e implementación de la paralelización. Expondremos de forma algorítmica los bloques de ejecución que se optimizan, comentando la reestructuración de código realizada respecto a la versión secuencial y las directivas de OpenMP introducidas.

Por un lado, en el apartado 7.2.1 se elabora la paralelización de los bloques que son independientes de la proyección de restricciones de MSXPBD y que por tanto, son comunes a las dos

versiones del SMT que se plantean en este capítulo. Teniendo esto en cuenta, hablaremos de *paralelización genérica* para referirnos a la optimización de dichos bloques, concretamente BE5, BE2 y BE7. Por otro lado, el bloque BE6 es el que se desarrollará por separado en 7.2.2 y 7.2.3 por las diferencias intrínsecas de los métodos de Jacobi y Coloreado de Grafos respectivamente.

7.2.1. Paralelización genérica

Comenzando por el bloque de tareas más crítico, *Closest Data* (BE5), conviene aclarar en qué contexto se ejecuta. Si consultamos el Algoritmo 3 que expone todo el cómputo del solver por fotograma, hablamos de que BE5 se corresponde al bucle de las instrucciones 7-8. Se trata de un bucle que, partiendo del músculo que se está calculando, recorre todos sus targets conectados (músculos, colisionadores, huesos) y, por cada vértice, realiza una serie de cálculos geométricos complejos. A grandes rasgos:

Algorithm 7 Pseudocódigo del Bloque de Ejecución *Closest Data* (BE5)

1: **for all** muscle m **do**
2: **for all** target t de m **do**
3: **for all** vértice v de m **do**
4: Cálculos relacionados con v y t

Puesto que el número de vértices siempre será mucho mayor que el número de targets conectados a un músculo o que el número total de músculos dentro del sistema, la paralelización más adecuada aquí es a nivel de vértice. Los cálculos que tienen lugar en la línea 4 del Algoritmo 7 son los mismos para todos los vértices, es decir, que todas las ejecuciones del bucle más interno están balanceadas. Todo lo dicho, nos conduce a introducir una directiva de paralelización de OpenMP con *schedule* estático. Además, debido a que este bucle se ejecuta dentro de la clase *MMuscle*, hay que tener en cuenta los miembros de la clase y las variables locales que serán compartidas por los distintos hilos, a saber, `mMesh`, `mClosestTargetData`, `mTargetMeshes` y el booleano `initialize`. Por todo lo expuesto, la directiva que se implementa es:

```
#pragma omp parallel for shared(mMesh, mClosestTargetData, mTargetMeshes, initialize) schedule(static)
for (unsigned id=0; id<mMesh->getNumVertices(); id++) {
    // Execute
}
```

El segundo bloque genérico es BE2 y se corresponde con la actualización de los colisionadores desde la capa de aplicación hasta el solver, esto es, lectura de sus geometrías en la escena y su traducción y almacenamiento en el solver. Este proceso es llevado a cabo por el nodo de Maya *SolverNode* en cada fotograma de la simulación y su ejecución tiene lugar en la línea

18 del Algoritmo 2. A nivel de implementación, se trata de un bucle que itera por todas las geometrías estáticas del sistema. Se les considera estáticas en tanto que no forman parte de la lista de músculos del solver pero sí pueden variar su forma y disposición en el espacio durante la simulación, por ejemplo, un hueso del esqueleto cuando se articula.

La paralelización de este bloque, por tanto, se realiza a nivel de objeto colisionador. El único miembro de la clase *SolverNode* compartido es `mCoreSolver`. La planificación de la directiva `omp` adecuada es estática ya que la ejecución interna del bucle es igual para todos los casos, sin condicionales ni bucles integrados que puedan perjudicar el balanceo de los hilos.

```
#pragma omp parallel for shared(mCoreSolver) schedule(static)
for (unsigned i=0; i<staticTargets.size(); i++) {
    // Execute
}
```

El último bloque de ejecución tenido en cuenta para la paralelización genérica está relacionado con la escritura de las nuevas posiciones de los vértices en las geometrías de la escena (BE7). Estas nuevas posiciones son el resultado de calcular todas las restricciones de todos los músculos del sistema en cada iteración. Así pues, es un cálculo que lo realiza la instancia de *CSolver* notificando a cada músculo que debe volcar su nuevo estado en la escena. Cada músculo, finalmente, realizará una serie de conversiones de datos y operaciones con la API de Maya que no precisan de iteración por vértice. Por todo ello, la paralelización que se implementa es a nivel del bucle de músculos en la clase *CSolver*. Del mismo modo que con BE2, en este caso la planificación del bloque `omp` es estática. En cuanto a miembros de la clase compartidos por todos los hilos, se encuentra la lista de músculos `mMuscles`.

```
#pragma omp parallel for shared(mMuscles) schedule(static)
for (unsigned muscleId=0; muscleId<mMuscles.size(); muscleId++) {
    // Execute
}
```

7.2.2. Paralelización con el Método Jacobi

En este apartado presentamos la implementación del *Sistema Multi-Threaded* con el *Método Jacobi* para la paralelización de la proyección de restricciones en el modelo MSXPBD. Nos referiremos a él como *SMT-J*. La premisa fundamental de este método es que, para que sea posible computar las restricciones concurrentemente, éstas no pueden modificar directamente las posiciones de los vértices afectados. En su lugar, los *deltas de posición* (esto es, los vectores desplazamiento que cada restricción calcula para cada vértice) deben acumularse dentro de una misma iteración para ser aplicados sólo una vez al final de la misma. La aplicación de este sumatorio de deltas la realiza cada vértice sobre sí mismo promediando por el número de

	Nombre	Tipo
Miembros	mDeltaX	std::atomic<int>
	mDeltaY	std::atomic<int>
	mDeltaZ	std::atomic<int>
	mNumConstraints	unsigned
	mOverRelaxation	float
Métodos	addConstraint	void
	getNumConstraints	unsigned
	setNumConstraints	void
	getOverRelaxation	float
	setOverRelaxation	void
	forceApplyDelta	void

Tabla 7.1: Métodos y miembros de la clase *CPoint* añadidos a la implementación de SMT-J.

restricciones que le afectan. Además, este promediado debe ser escalado por un factor llamado de sobre-relajación (consultar Ecuación 2.8).

Dicha premisa supone un cambio considerable en la implementación. En primer lugar, necesitamos realizar modificaciones en la clase *CPoint*. La Tabla 7.1 recopila los miembros de la clase y métodos que han sido añadidos en esta versión del SMT-J. Así, `mNumConstraints`, `mOverRelaxation`, sus respectivos métodos *setters* y *getters* y también `addConstraint` facilitan la configuración de las ecuaciones 2.7 y 2.8. Durante la inicialización del sistema, cuando un músculo construye todas sus restricciones, se va incrementando el contador `mNumConstraints` con llamadas a `addConstraint`. Gracias a esto, la implementación del método `applyDelta` (que ya existía en el SST) se ha actualizado para aplicar coherentemente el promediado de deltas junto con el escalado por sobre-relajación.

Es importante destacar el uso de `std::atomic<int>` para acumular de forma separada las componentes de los *delta*. Durante el desarrollo, diversas pruebas confirmaron que era menos costoso este diseño que mantener un *array* de tres elementos. Además, el estándar C++11 no incluye implementación del método `std::atomic::fetch_add` para flotantes. Por ello, nos servimos de tipos enteros con el consecuente ajuste de la parte entera y decimal. Sin lugar a dudas, esto supone una pérdida de precisión que podrá resolverse con C++20, el cual incluirá `fetch_add` para tipos flotantes.

Por lo que respecta al método `forceApplyDelta`, podemos describirlo como el renombrado del `applyDelta` original de SST, ya que permite la escritura directa de un delta sobre la posición del vértice, sin pasar por el acumulado ni el promediado. Esta funcionalidad ha sido necesaria para garantizar que ciertas restricciones apliquen totalmente su corrección. Por ejemplo, hablamos de las restricciones de *attachment* gracias a las cuales los músculos permanecen sujetos al esqueleto. En ningún caso queremos que, con motivo del promediado y la pérdida de

rigidez, los puntos de anclado lleguen a separarse de los huesos.

A parte de un cambio en el diseño de *CPoint*, el enfoque jacobiano nos obliga a remodelar el método `computeConstraints` de *MMuscle*. Este método es el encargado de computar una iteración de MSXPBD a nivel de músculo. Tal cosa implica: recorrer la lista de restricciones, empaquetar los datos que cada una necesita para calcular, proyectarlas, recoger los resultados, acumularlos y finalmente aplicarlos. Es, por tanto, el método que traduce en código el planteamiento del Método Jacobi. En el Algoritmo 8 presentamos la implementación de este método que iremos comentando para justificarlo (nótese la diferencia respecto a su homólogo en el SST, ver Algoritmo 5).

Algorithm 8 Pseudocódigo de la proyección de restricciones en SMT-J.

1: **for all** restricción Gauss-Seidel c **do** ▷ Restricciones que no promedian sus deltas
2: **for all** vértice i incluido en c **do**
3: Actualizar $\mathbf{p}_i$ en c
4: Actualizar w_i en c
5: Actualizar datos de conexiones de i con los targets en c
6: Calcular C
7: Calcular $\Delta\lambda_c$
8: **for all** vértice i incluido en c **do**
9: Calcular $\Delta\mathbf{p}_i$
10: $\mathbf{p}_i \leftarrow \mathbf{p}_i + \Delta\mathbf{p}_i$ ▷ Llamada a `forceApplyDelta`
11: $\lambda_c \leftarrow \lambda_c + \Delta\lambda$
12:
13: **for all** restricción Jacobi c **do** ▷ Restricciones que sí promedian sus deltas
14: **for all** vértice i incluido en c **do**
15: Actualizar $\mathbf{p}_i$ en c
16: Actualizar w_i en c
17: Actualizar datos de conexiones de i con los targets en c
18: Calcular C
19: Calcular $\Delta\lambda_c$
20: **for all** vértice i incluido en c **do**
21: Calcular $\Delta\mathbf{p}_i$
22: Acumular $\Delta\mathbf{p}_i$ ▷ Uso de `std::atomic::fetch_add` a nivel de *CPoint*
23: $\lambda_c \leftarrow \lambda_c + \Delta\lambda$
24:
25: **for all** vértice i **do**
26: $\mathbf{p}_i \leftarrow \mathbf{p}_i + \Delta\mathbf{p}_i * w/n_i$ ▷ w (`mOverRelaxation`), n_i (`mNumConstraints` de i))

El bloque de instrucciones 1-11 del Algoritmo 8 describe el cálculo de todas aquellas restricciones consideradas fuera del planteamiento jacobiano, es decir, aquellas cuyos deltas deben ser aplicados completamente. Por ese motivo se les ha etiquetado como restriciones *Gauss-Seidel*

ya que mantienen el mismo algoritmo que en el SST. El único cambio a nivel de implementación se halla en la línea 10, puesto que en esta versión el método que se ejecuta es `forceApplyDelta`.

En la línea 13 comienza el bucle que implementa el Método Jacobi propiamente dicho y, por tanto, es el bucle que paralelizaremos por completo. En él, se proyectan todas las restricciones que participan en el proceso de promediado y están preparadas para poder ser ejecutadas concurrentemente. A nivel de código, la modificación para aplicar Jacobi se observa en la línea 22 únicamente. Es aquí donde cada vértice afectado por una restricción escribe el delta en el acumulado de tipo `std::atomic<int>` por componentes. Es importante que esta escritura sea atómica ya que puede darse el caso de que dos restricciones pidan acumular un delta al mismo vértice en el mismo instante. Para evitar las inestabilidades que esto produciría, ha sido necesaria la introducción de estas operaciones de carácter bloqueante.

Como se ha dicho, el bucle sobre las restricciones de Jacobi puede paralelizarse mediante un bloque `omp` cuyas variables miembro compartidas son `mConstraints`, `mMesh` y `mBranchPoints` (esta última, recordemos, es la lista de puntos de la estructura interna). La configuración del *schedule* es dinámica ya que el flujo de código varía mucho entre restricciones de disinta clase. Por ejemplo, la restricción de volumen es muy costosa porque trabaja sobre todos los vértices de la superficie del músculo haciendo operaciones complejas, mientras que la restricción de *attachment* afecta a un único punto al que sólo tiene que forzarle una traslación.

```
#pragma omp parallel for shared(mConstraints, mMesh, mBranchPoints) schedule(dynamic)
for (unsigned i=mJacobiIdx; i<mConstraints.size(); i++) {
    // Execute
}
```

Al finalizar la proyección de restricciones, llegamos al punto de aplicar los acumulados. Esto ocurre en el bucle de las líneas 25-26. Cada vértice del músculo en cuestión (incluidos también los puntos de la estructura interna) ya tiene almacenados los deltas de todas sus restricciones, de manera que puede aplicar la Ecuación 2.8 para obtener su nueva posición y estar listo para la iteración siguiente. En este bucle hay total independencia de datos, ya que son operaciones que ocurren dentro de la clase *CPoint*, lo que significa que es un bucle paralelizable sin compartición de variables y con distribución de hilos estática.

```
#pragma omp parallel for schedule(static)
for (unsigned id=0; id<mMesh->getNumVertices(); id++) {
    // Execute
}
#pragma omp parallel for schedule(static)
for (unsigned id=0; id<mBranchPoints.size(); id++) {
    // Execute
}
```

7.2.3. Paralelización con el Método de Coloreado de Grafos

El método de paralelización que vamos a implementar en este apartado es el *Coloreado de Grafos* (también llamado *Graph-Coloring* o simplemente *Método de Grafos*) dando lugar a una segunda versión del *Sistema Multi-Threaded*, a saber, *SMT-G*. El enfoque basado en grafos consiste en la separación de las restricciones en grupos independientes entre sí, entendiendo independientes como la prohibición de que exista más de una restricción en el grupo que afecte al mismo vértice. Si se cumple esta premisa, las restricciones de un mismo grupo pueden ejecutarse en paralelo sin riesgos de concurrencia de acceso a los datos. Tal y como se comenta en el Estado del Arte 2.2.1, cuanto menor sea el grado de compartición de vértices entre restricciones, mayor será el tamaño de los grupos y, por consiguiente, mayor nivel de paralelización. En la situación opuesta, cuando las partículas están conectadas a muchas restricciones, se generan muchos grupos de reducido tamaño con consecuencias muy malas para la eficiencia del sistema. El balanceo de carga entre grupos también es importante, ya que la existencia de grupos grandes y otros muy pequeños afecta negativamente a la ocupación de los hilos durante la ejecución.

Dada la implementación de la que partimos, que es la realizada para el SST donde la proyección de restricciones se hace recorriendo los músculos secuencialmente, la posibilidad que tenemos de introducir el método de grafos se reduce a realizar el agrupamiento de restricciones dentro de cada músculo. Considerar todas las restricciones del solver juntas (ignorando el músculo al que pertenecen) para construir la agrupación en términos globales del sistema se antoja inviable. Este planteamiento más global obligaría a un nuevo diseño de la arquitectura del plug-in, cosa que se excede del alcance de este proyecto. Por esta razón, la separación de restricciones en grupos se realiza a nivel interno de cada músculo.

La complejidad del *Graph-Coloring* se encuentra no tanto en la proyección de restricciones en sí, sino en el algoritmo de agrupación que se necesita en el momento de inicializar el sistema. En nuestra implementación, se ha diseñado una clase nueva que facilita la asignación del grupo correcto a cada restricción, a la que llamamos *ColoringGroups*. A grandes rasgos, esta clase contiene dos vectores miembro en los que cada posición hace referencia a un grupo. Por eso, los índices de uno y otro están interrelacionados en el sentido de que uno almacena los IDs de restricción añadidos a cada grupo, y otro los IDs de vértices pertenecientes a esos grupos. La implementación de esta clase (ver Figura 7.1) permite construir los grupos de manera sencilla durante la inicialización del solver e indexar fácilmente las restricciones que se pueden ejecutar en paralelo durante el cómputo de las mismas. Cuando un *MMuscle* crea una restricción nueva, éste le da el control a *ColoringGroups*, quien se encarga de encontrar un grupo disponible en el que esa restricción no comparta vértices con ninguna otra.

La información relacionada con la agrupación de restricciones sólo se calcula una vez, al comienzo de la simulación. A partir de ahí, la integración del método en la implementación

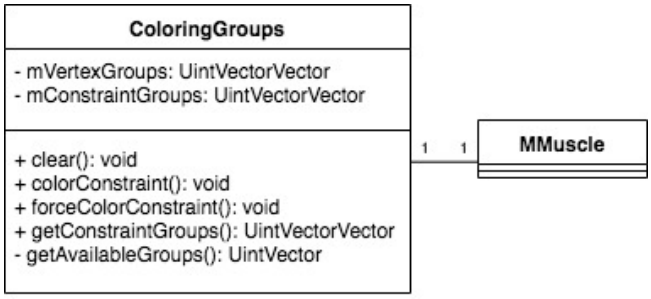

Figura 7.1: Diagrama de la clase *ColoringGroups* y su dependencia con *MMuscle*.

original del SST es directa. En el Algoritmo 9 que se adjunta, vemos el pseudocódigo de la proyección de restricciones de SMT-G. Si nos fijamos, es exactamente igual al Algoritmo 5 pero añade un bucle externo que recorre los grupos obtenidos con la clase *ColoringGroups* implementada.

Algorithm 9 Pseudocódigo de la proyección de restricciones en SMT-G.

1: **for all** grupo g **do**
2: **for all** restricción c en g **do**
3: **for all** vértice i incluido en c **do**
4: Actualizar $\mathbf{p}_i$ en c
5: Actualizar w_i en c
6: Actualizar datos de conexiones de i con los targets en c
7: Calcular C
8: Calcular $\Delta\lambda_c$
9: **for all** vértice i incluido en c **do**
10: Calcular $\Delta\mathbf{p}_i$
11: $\mathbf{p}_i \leftarrow \mathbf{p}_i + \Delta\mathbf{p}_i$
12: $\lambda_c \leftarrow \lambda_c + \Delta\lambda$

La optimización que propone el Método de Grafos, finalmente, supone la paralelización del bucle interno que comienza en la línea 2, es decir, calcular los grupos secuencialmente y las restricciones en paralelo. Por un lado, a nivel de planificación para la directiva `omp` se puede utilizar `dynamic` ya que habrá restricciones con costes de cálculo muy variados. Por otro lado, las variables compartidas por todos los hilos son los miembros `mConstraints`, `mMesh` y `mBranchPoints` igual que ocurría en la versión con Jacobi.

```
UintVectorVector groups = mGroups.getConstraintGroups();
for(auto &g : groups) {
    # pragma omp parallel for shared(mConstraints, mMesh, mBranchPoints) schedule(dynamic)
    for(unsigned c=0; c<g.size(); c++) {
        // Execute
    }
}
```

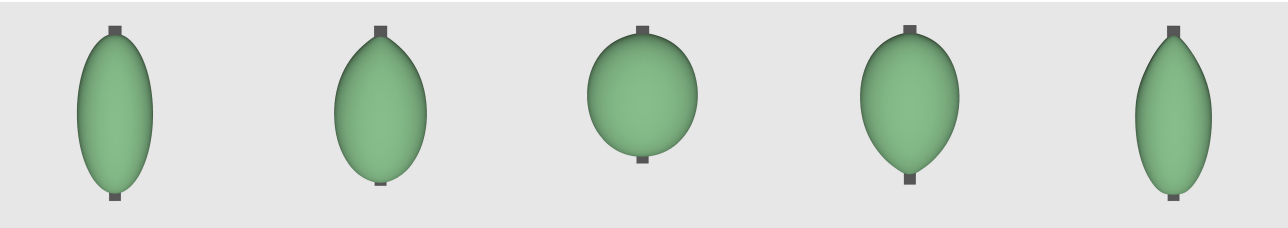

Figura 7.2: Activación y relajación de un músculo simple con SMT-J (T1).

7.3. Análisis de resultados

Esta sección está dedicada al análisis de los resultados de la paralelización tanto desde el punto de vista de la calidad de las simulaciones en términos de comportamiento, como desde el punto de vista del coste de cómputo y eficiencia. Someteremos a estudio las dos versiones del sistema multi-threaded implementadas en el presente capítulo: la versión con Jacobi y la versión con el Método de Grafos.

El apartado 7.3.1 que sigue se va a estructurar, a su vez, en dos sub-apartados para separar claramente el análisis de SMT-J y de SMT-G. Haremos hincapié en los detalles más relevantes de cada uno que nos ayuden a comparar ambos sistemas con el SST y para extraer las conclusiones pertinentes. Por su parte, el apartado 7.3.2 se centrará en comprender cómo cambian los tiempos gracias a las optimizaciones implementadas.

7.3.1. Calidad de la simulación

7.3.1.1. Calidad de la simulación con SMT-J

El SMT con el Método Jacobi ha sido sometido a los test T1 a T7 planteados en la Metodología. A continuación, veremos qué respuesta hemos obtenido con el plug-in implementado aplicando el enfoque jacobiano. Comenzando con T1 (ver Figura 7.2), se puede observar que el músculo ovalado reproduce el ciclo de activación y relajación con éxito. Se puede notar, no obstante, que en el fotograma final no recupera completamente la forma inicial, denotando cierta relajación respecto a ella.

En el test T2 se somete a prueba el bíceps anatómico para comprobar que la activación y ganancia de volumen funciona correctamente en una geometría compleja. La Figura 7.3 demuestra que con Jacobi también se obtienen buenos resultados: en la fila superior se realiza la contracción; y en la fila inferior la relajación. Además, en este caso, el bíceps sí que recupera completamente la forma.

En lo referente a T3 para la interacción con el esqueleto, Figura 7.4, podemos decir que Jacobi resuelve bien la simulación, ya que se observa una contracción del músculo correcta,

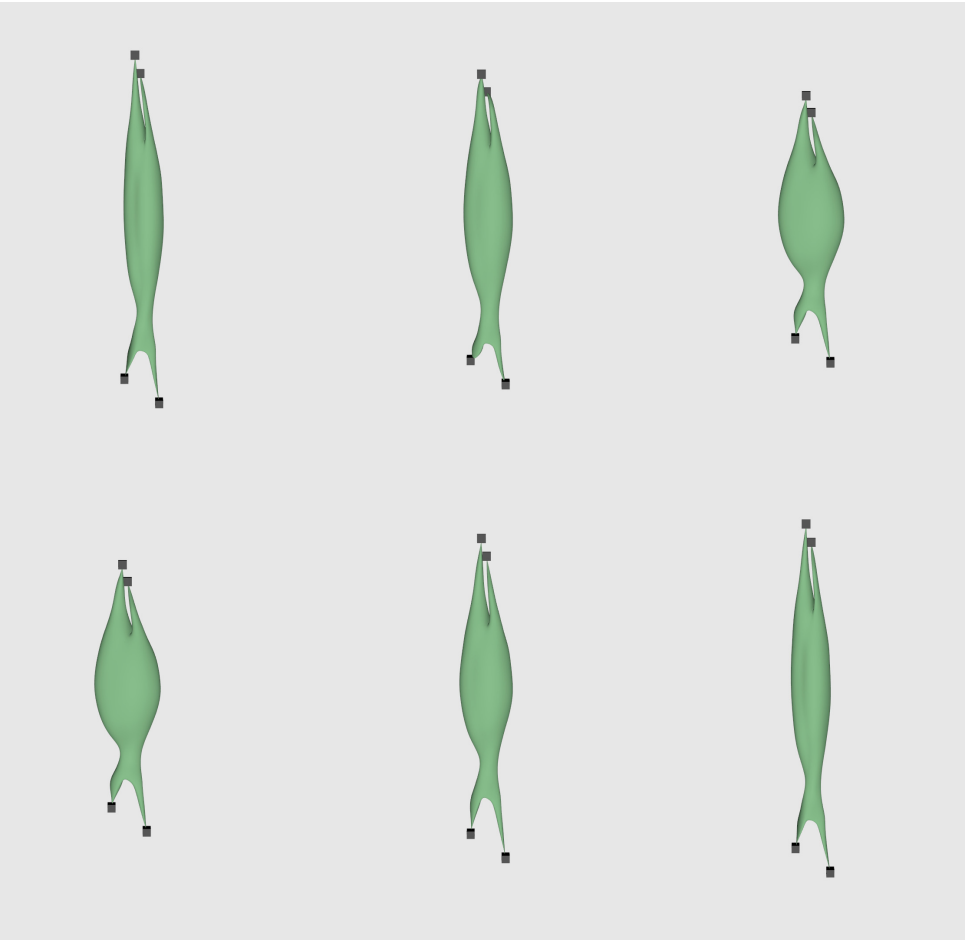

Figura 7.3: Activación y relajación del bíceps anatómico con SMT-J (T2).

además de una ganancia de volumen y un cambio de forma que es coherente con la articulación del esqueleto. En la imagen se pueden ver tres fotogramas durante la flexión del codo (vista lateral arriba y vista frontal abajo) que evidencian este comportamiento.

Las conexiones entre músculos se ponen a prueba en el test T4. La Figura 7.5 recoge cuatro instantes de la simulación de un bíceps y un brachialis que interaccionan entre sí durante el proceso de activación. Igual que en T3, podemos decir que el cambio de forma es coherente con la activación de fibras y el incremento de los volúmenes por esta contracción.

En el test T5, llegamos a la simulación con todos los músculos del brazo para comprobar el comportamiento del sistema cuando la complejidad de la escena es alta. Se adjunta, en la Figura 7.6, una perspectiva frontal (columna izquierda) y otra lateral (columna derecha) de cuatro fotogramas generados durante el proceso de flexión del codo. Se puede ver que los resultados son satisfactorios y que, igual que ocurre en los test anteriores, la activación de los músculos genera cambios de forma realistas.

El test T6 está diseñado para comprobar la estabilidad del sistema dada la naturaleza de la

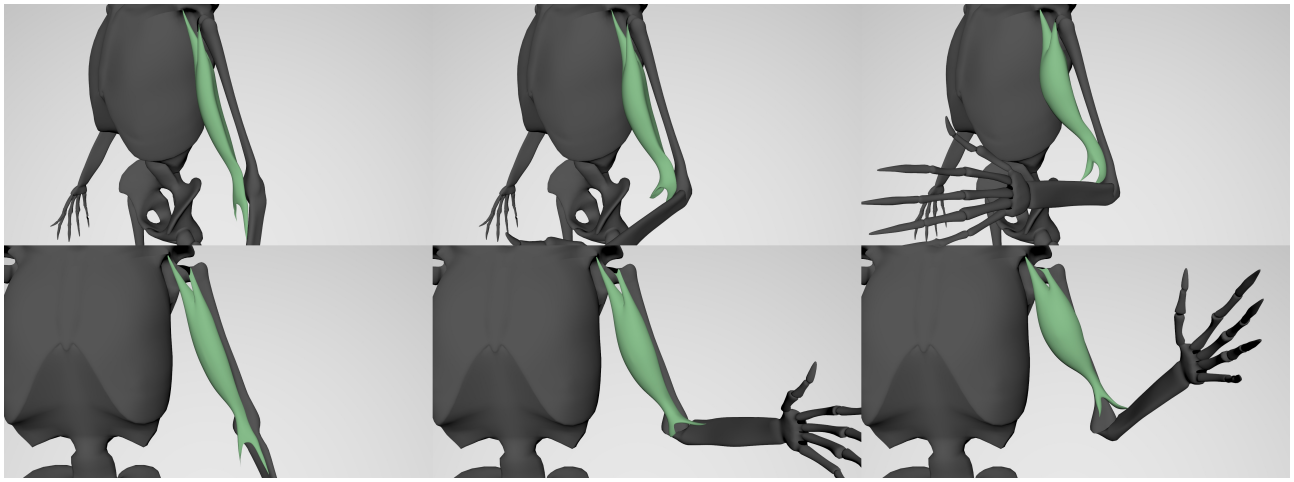

Figura 7.4: Activación y relajación del bíceps anatómico conectado al esqueleto con SMT-J (T3).

animación del esqueleto, la cual contiene movimientos que dan lugar a inercias muy elevadas. La Figura 7.7 presenta cuatro fotogramas del principio de la simulación y, como se puede observar, SMT-J no es capaz de mantener los músculos acoplados al esqueleto y el sistema se rompe.

Las pruebas realizadas para tratar de resolver este problema se han basado en un incremento del número de iteraciones, aumento de la rigidez de los músculos, aumento del rango de activación y también aumento del peso de las restricciones de tipo *HardConstraint* que son las que más limitan el desplazamiento de los vértices. Ninguna de estas decisiones ha logrado estabilizar el sistema. La literatura especializada advierte de que el Método Jacobi causa relajación del sistema y que, el cociente de sobre-relajación puede ayudar a compensar la pérdida de rigidez pero no siempre es suficiente. En nuestro caso, aumentar este factor tampoco ha resuelto el problema, ya que un valor demasiado alto del mismo, también ocasiona que el sistema se vuelva impredecible.

Llegados a este punto, podemos concluir que nuestra implementación del SMT-J funciona bien en escenarios simples en los que la animación del esqueleto es ligera y los movimientos suaves, pero no es aplicable sobre escenas más complejas donde las deformaciones que sufren los músculos exceden ciertos límites y las restricciones no pueden corregirlos para hacer converger la simulación.

Con el objetivo de combatir esta deficiencia de nuestro sistema, se ha decidido aplicar una estrategia nueva no heredada del modelo original de MSXPBD. Dicha estrategia se basa en forzar que todos los vértices estén conectados al esqueleto vía *HardConstraint* y que estas restricciones se computen siempre al principio de cada iteración y sin formar parte del algoritmo de promediado que aplica Jacobi. La intención es, básicamente, forzar la colocación de los vértices al principio del fotograma en sus posiciones correctas respecto al esqueleto y, una vez

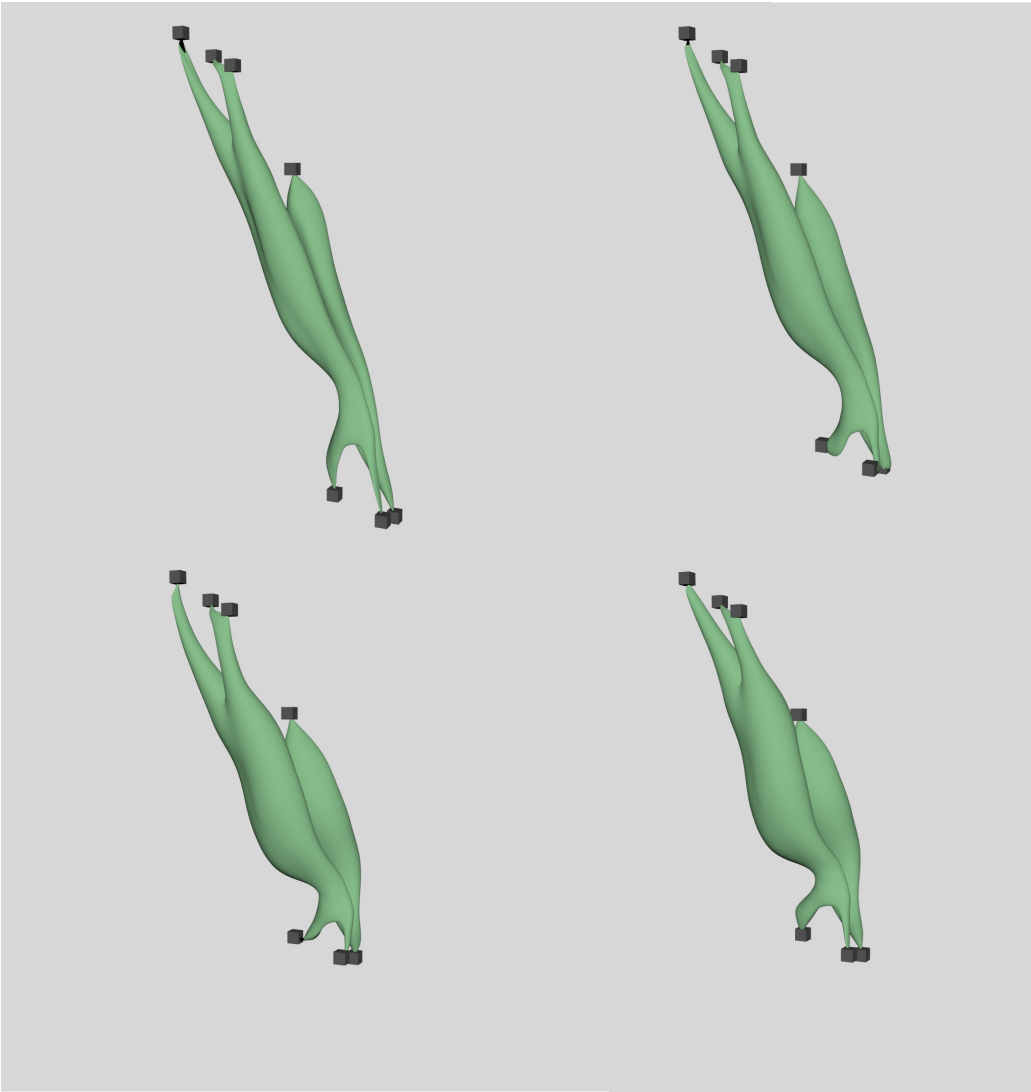

Figura 7.5: Activación de bíceps y brachialis anatómicos conectados entre sí con SMT-J (T4).

hecho esto, computar el resto de restricciones al estilo jacobiano. Los resultados que se obtienen con este nuevo enfoque se recogen en la Figura 7.8.

Los resultados de SMT-J con la adición forzada de conexiones al esqueleto resuelve el problema de la inestabilidad en T6. Ahora, los músculos se mantienen sujetos al esqueleto a pesar de que sus movimientos sean rápidos y exagerados. El hecho de que la introducción de estas nuevas restricciones corrija el problema, demuestra que la implementación del sistema puede ser correcta pero que la sobre-relajación de Jacobi es, realmente, un gran inconveniente para su aplicación sobre MSXPBD.

Si bien la convergencia se recupera al añadir más restricciones, es cierto que debemos fijarnos con detalle en las dinámicas que tienen lugar en los vértices. En los vídeos generados, se ha detectado que el exceso de conexiones de hard convierten a los músculos en objetos muy rígidos

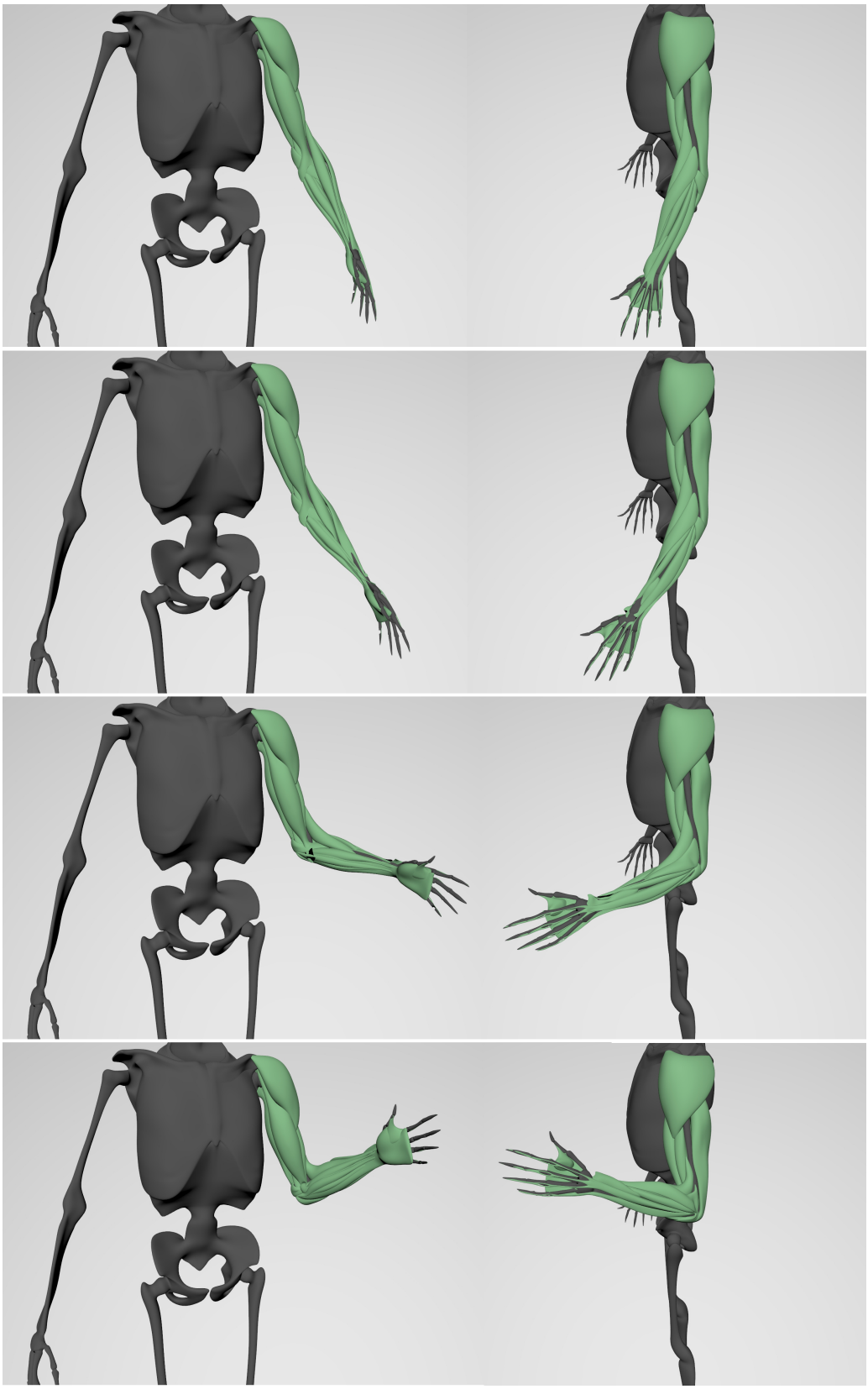

Figura 7.6: Simulación del brazo completo durante la flexión del codo con SMT-J (T5).

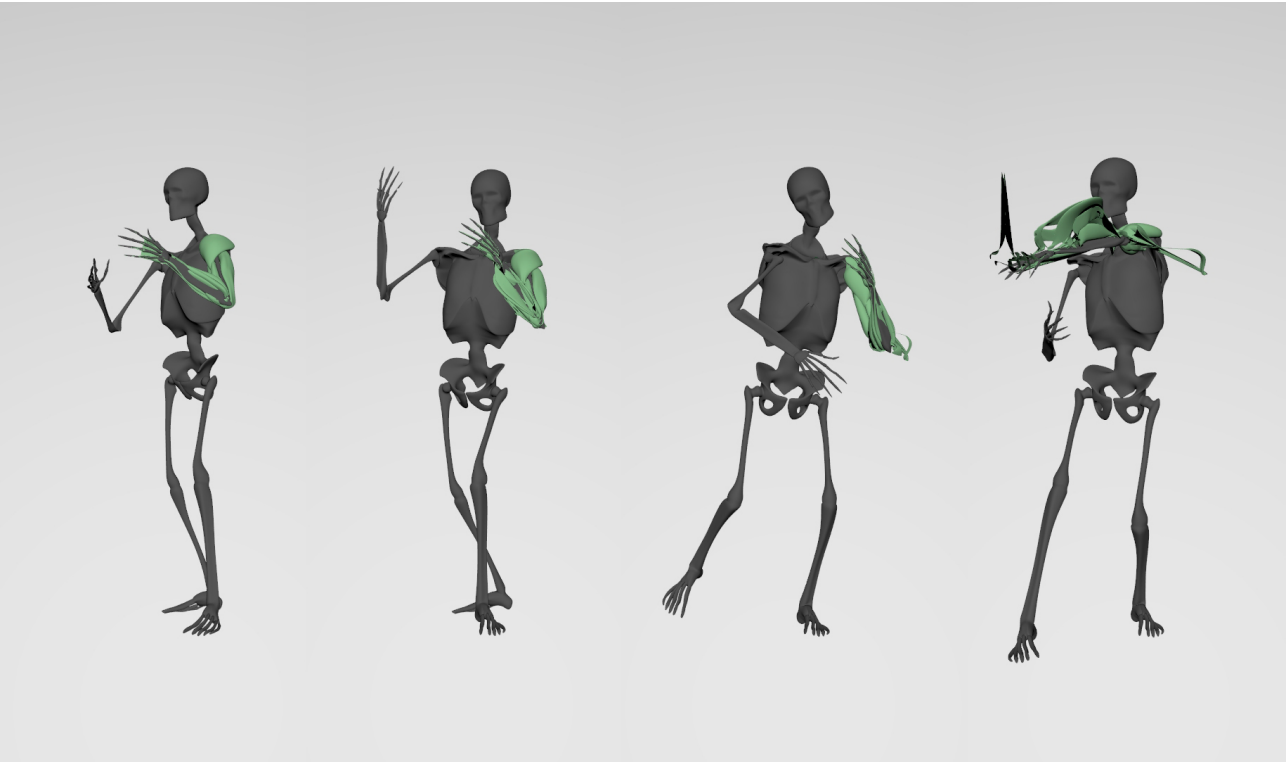

Figura 7.7: Simulación fallida del brazo completo sobre movimientos de lucha con SMT-J (T6).

y casi no se aprecian rebotes o efectos de *jiggle* que se observan típicamente en otros sistemas de simulación de músculos. A pesar de que estos detalles no se puedan apreciar en fotogramas estáticos, se adjunta la Figura 7.9 donde se muestran tomas más cercanas de la simulación realizada para el T6 con estas nuevas conexiones. La conclusión importante que debemos sacar es que, si bien el uso de más restricciones ayuda a estabilizar el sistema con Jacobi, estamos introduciendo niveles enormes de rigidez, lo cual es un inconveniente muy grande al perder las dinámicas que es capaz de generar MSXPBD.

Para finalizar con el conjunto de test sobre STM-J, se ha ejecutado T7. Conocido el problema comentado sobre el exceso de relajación en T6, este nuevo test ha tenido en cuenta las conexiones extra entre vértices de los músculos y del esqueleto. Algunos instantes de la simulación se pueden ver las Figuras 7.10 y 7.11 con perspectivas frontal y trasera respectivamente.

Por un lado, se puede decir que los resultados son buenos en la mayoría de los músculos, ya que permanecen compactos y sujetos al esqueleto evitando problemas como el visto en la Figura 7.7. Por otro lado, debemos hacer hincapié en la rigidez de los mismos. Es cierto que el sistema converge y es estable pero la dureza de los músculos reduce mucho las dinámicas que cabría esperar cuando, por ejemplo, se producen apoyos e impactos de los pies en el suelo. Además, es justo señalar que existe un problema evidente en los isquiotibiales (consultar vista trasera

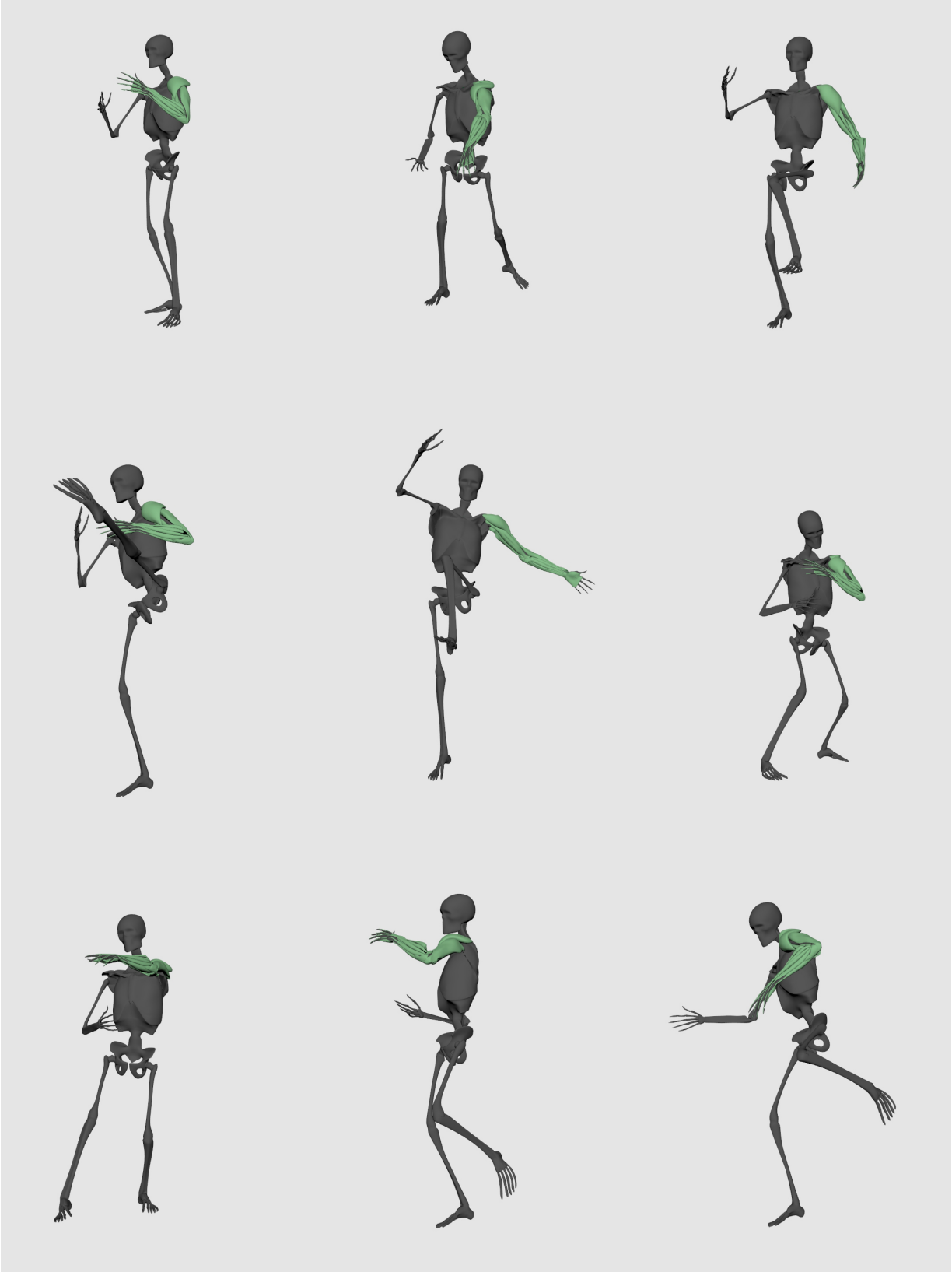

Figura 7.8: Simulación del brazo completo sobre movimientos de lucha con SMT-J con el forzado de *HardConstraints* de todos lo vértices al esqueleto (T6).

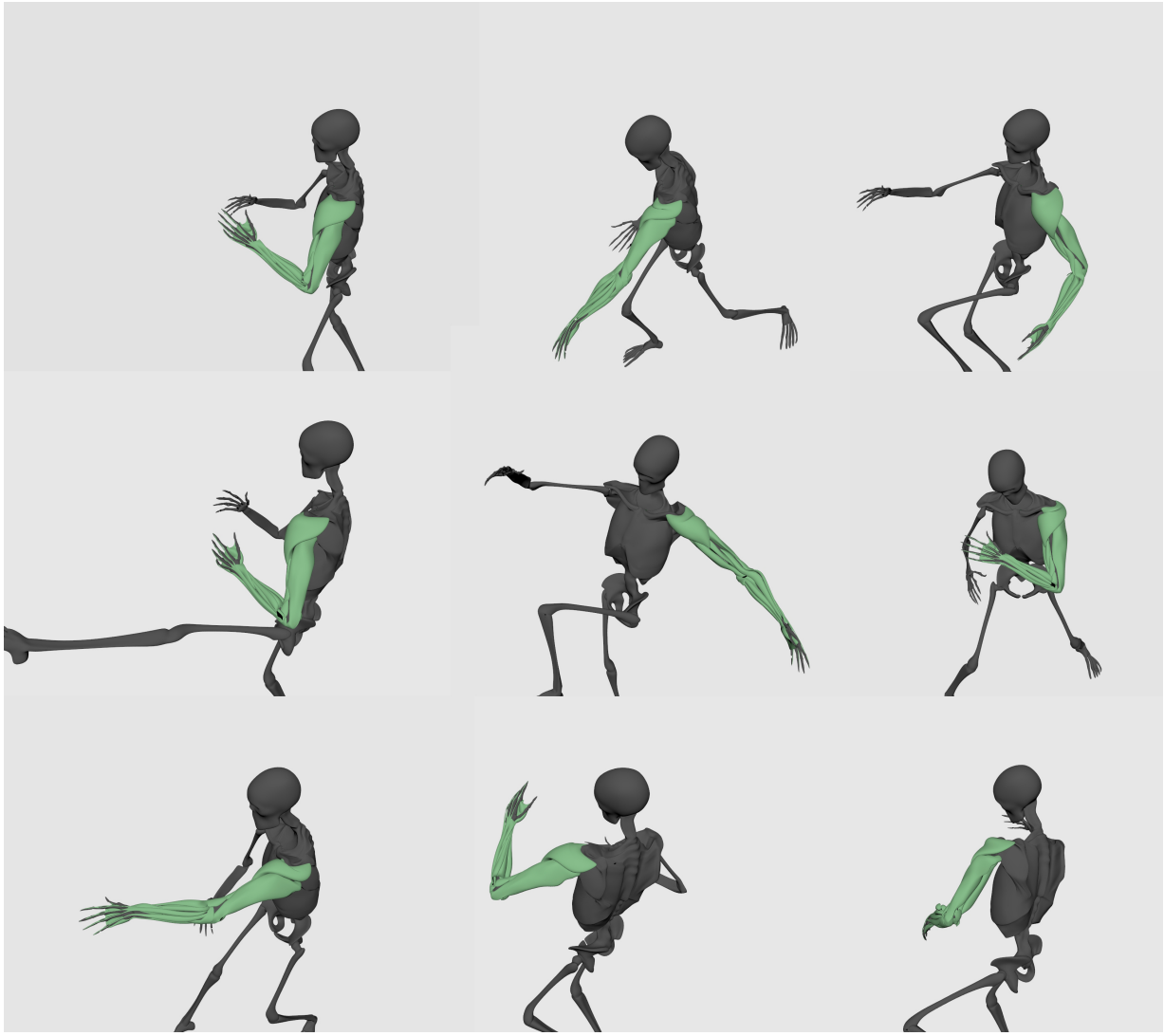

Figura 7.9: Perspectiva detalle de la simulación del brazo completo sobre movimientos de lucha con SMT-J con el forzado de *HardConstraint* de todos lo vértices al esqueleto (T6).

7.11). Los músculos de esa zona están totalmente fuera de la interacción con sus vecinos, lo que produce un comportamiento que no es aceptable. Posiblemente se deba a que, recordemos, este test se realiza con la configuración por defecto en los *MuscleNode*. Por eso, la primera vía para intentar resolver este problema sería modificar cuestiones relativas a la rigidez, activación, músculos vecinos o la dureza de las conexiones sliding, soft y hard. Si el problema persiste, podría ser que estuviéramos ante una limitación importante del sistema con el Método Jacobi.

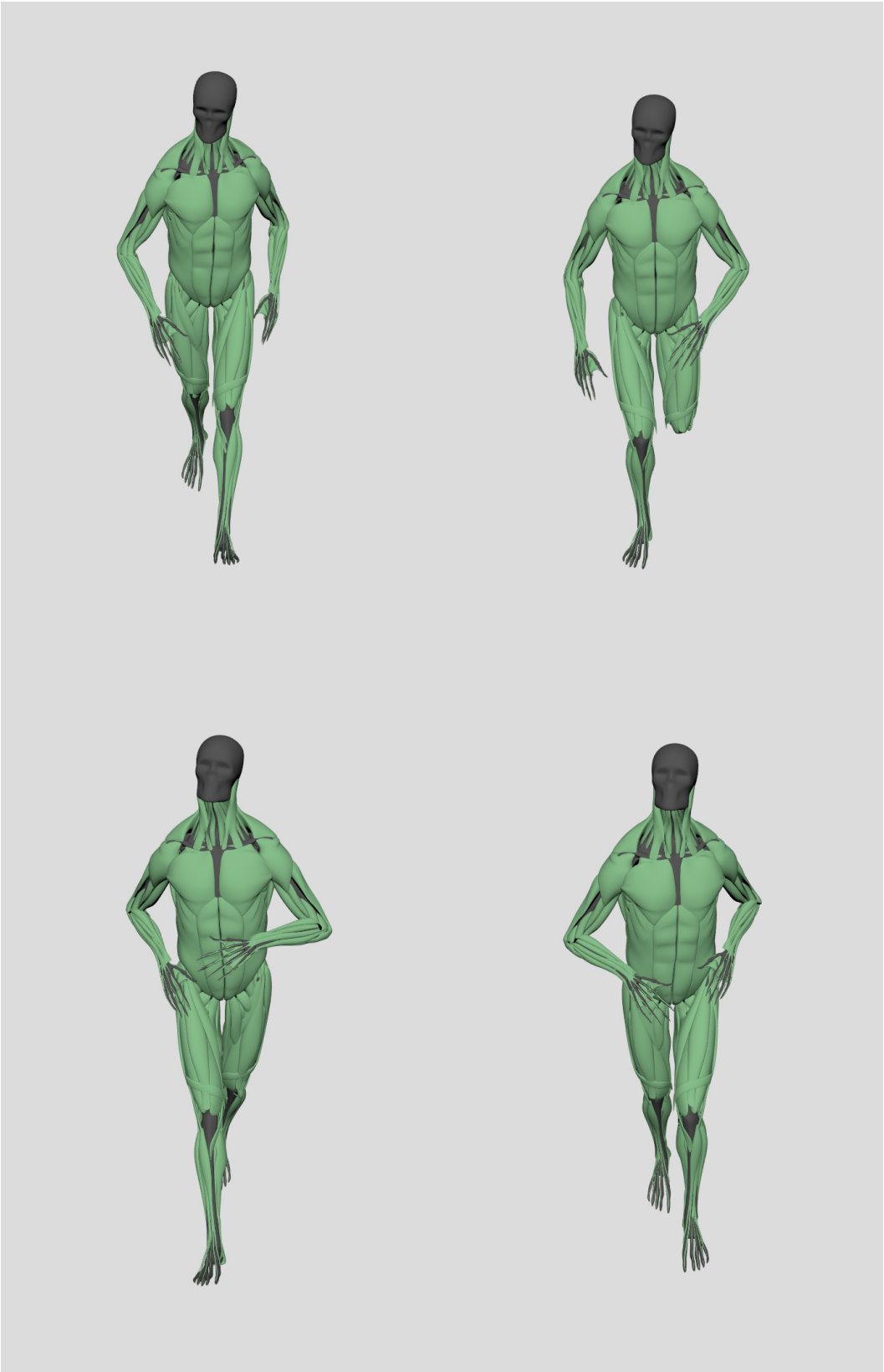

Figura 7.10: Simulación del cuerpo completo caminando y corriendo con SMT-J (T7).

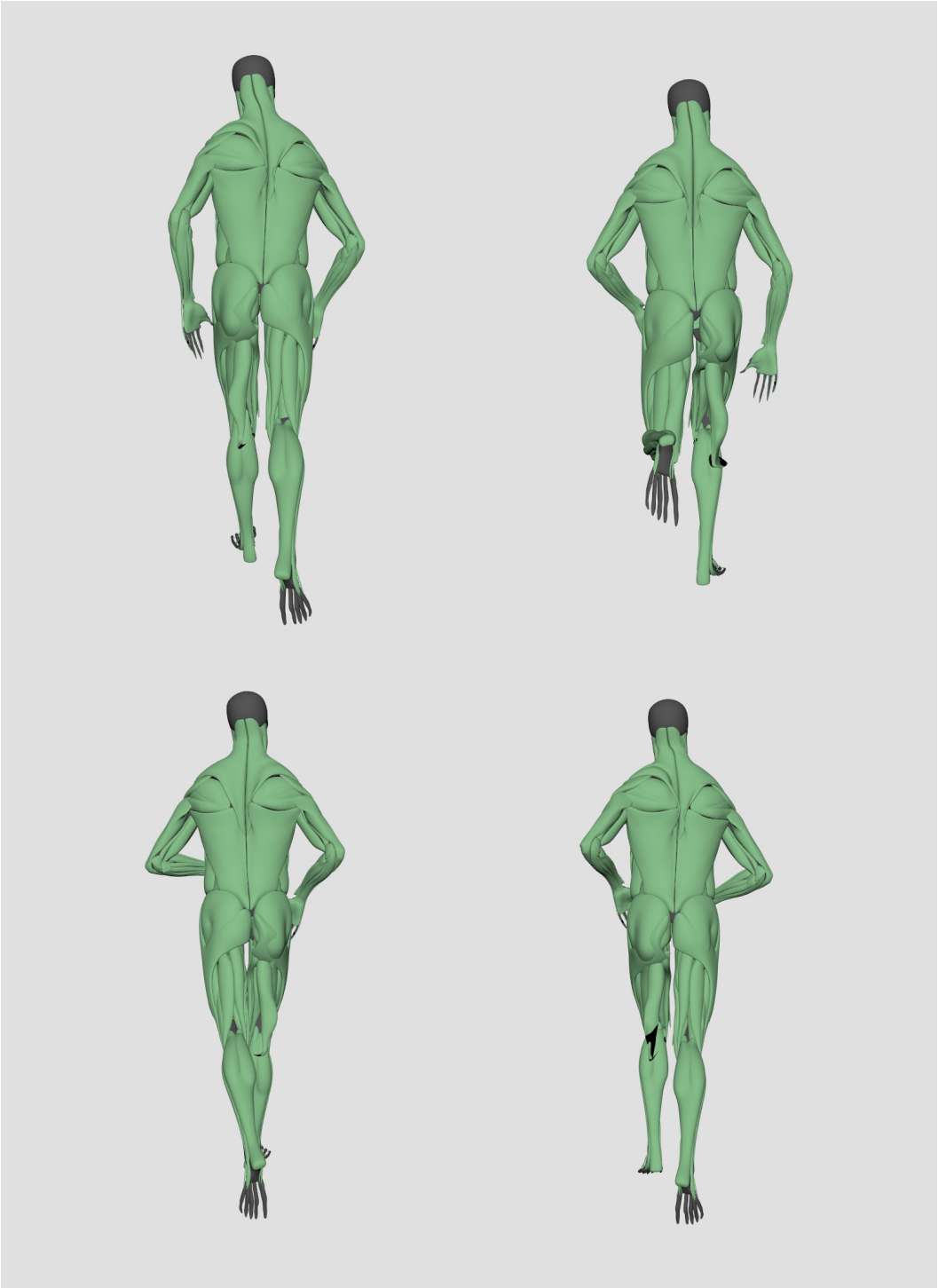

Figura 7.11: Simulación del cuerpo completo caminando y corriendo con SMT-J (T7).

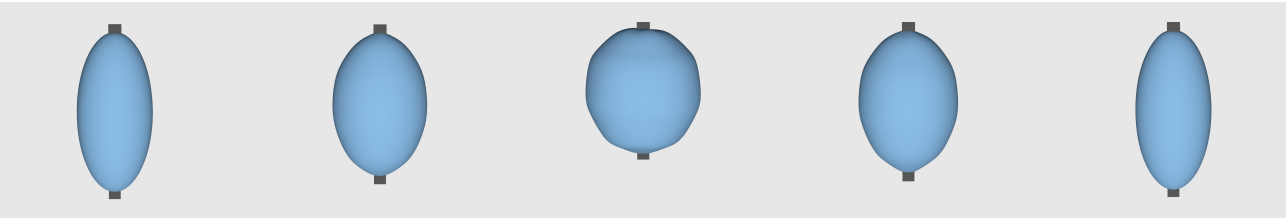

Figura 7.12: Activación y relajación de un músculo simple con SMT-G (T1).

7.3.1.2. Calidad de la simulación con SMT-G

En este apartado presentamos los resultados logrados con el sistema basado en el Método de Grafos. Del mismo modo que se hizo con SST y SMT-J, vamos a mostrar ejemplos de las simulaciones de los test T1 a T7. En primer lugar, el test T1 ha confirmado que la agrupación de restricciones implementada también mantiene el funcionamiento básico de nuestro sistema, en tanto que el músculo se contrae satisfactoriamente (Figura 7.12). En el fotograma central de esta figura, podemos ver cómo en el instante de máxima activación, el músculo ovalado no adquiere la forma perfectamente redondeada que se obtuvo en las Figuras 6.7 o 7.2. Esto se debe, principalmente, al orden de ejecución de las restricciones a causa de la distribución en grupos. Tal distribución conlleva a que las restricciones *FiberConstraint* que se calculan en el último grupo tengan mayor influencia sobre el resultado final que las demás.

En el T2 de la Figura 7.13 se demuestra que SMT-G funciona bien con un músculo más complejo. La activación y relajación del bíceps es buena en lo que a cambio de forma y volumen se refiere. Además, el patrón de deformación no deseado que se observaba el fotograma central de la Figura 7.12, queda disimulado en este caso gracias a que la densidad de la malla es mayor.

Siguiendo con el estudio del bíceps, en T3 se evalúa la interacción músculo-esqueleto. La Figura 7.14 resume la fase de contracción del mismo en tres fotogramas (una perspectiva lateral en la fila superior y una más frontal en la inferior). De nuevo, podemos afirmar que SMT-G responde bien cuando añadimos la presencia de conexiones externas con el esqueleto.

Continuando con el listado de test, en T4 se simulan el bíceps y el brachialis anatómicos con el objetivo de comprobar si la implementación es capaz de resolver bien el problema cuando se trata de múltiples músculos que interaccionan entre sí. Se han recogido en la Figura 7.15 cuatro fotogramas de la contracción. Nótese las formas que adquieren los dos músculos cuando la activación es máxima, denotando mayor rigidez y aumentando el volumen en las zonas esperadas.

La prueba de simulación con el brazo completo se introduce en T5: la flexión del codo debe activar bíceps, brachialis y parte del antebrazo. Siguiendo el criterio aplicado a SST y SMT-J, en esta ocasión también proponemos cuatro capturas del proceso de contracción desde dos perspectivas diferentes, frontal y perfil. La validación de SMT-G desde el punto de vista

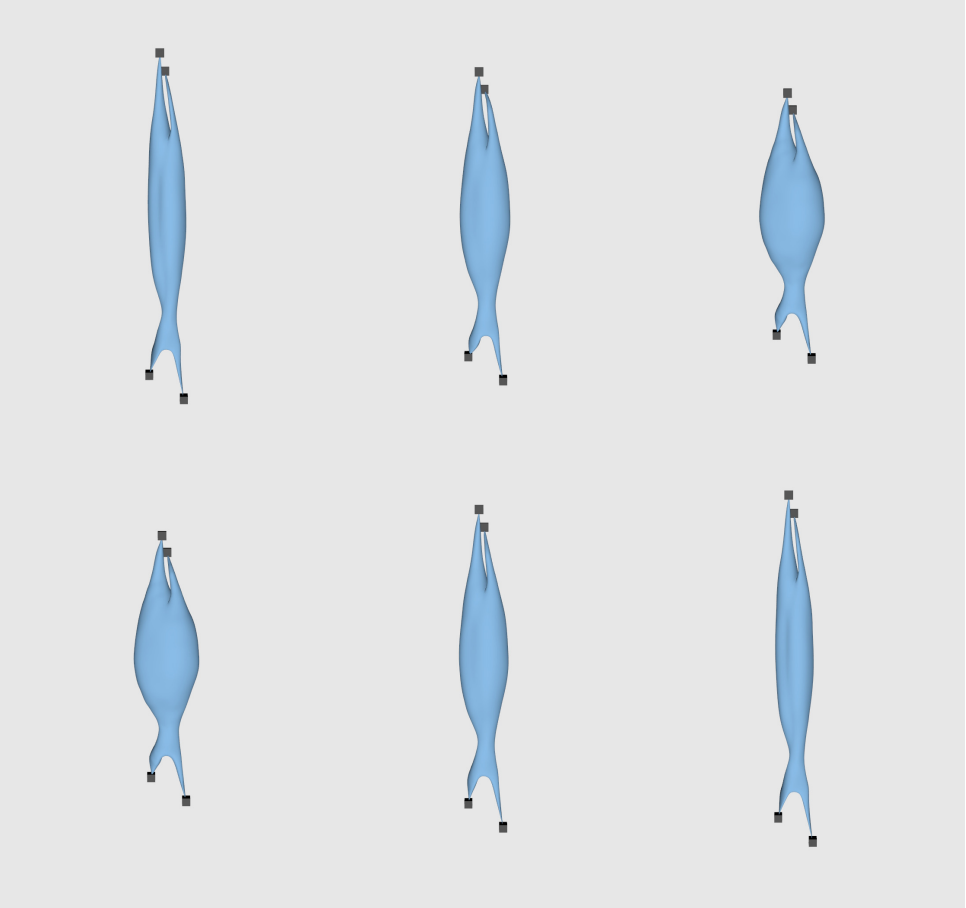

Figura 7.13: Activación y relajación del bíceps anatómico con SMT-G (T2).

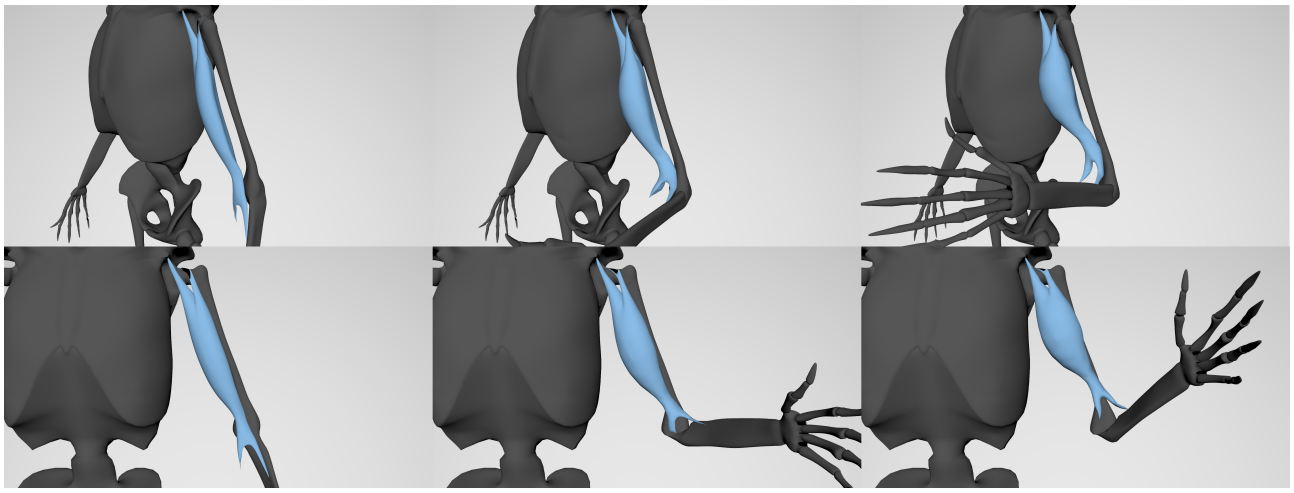

Figura 7.14: Activación y relajación del bíceps anatómico conectado al esqueleto con SMT-G (T3).

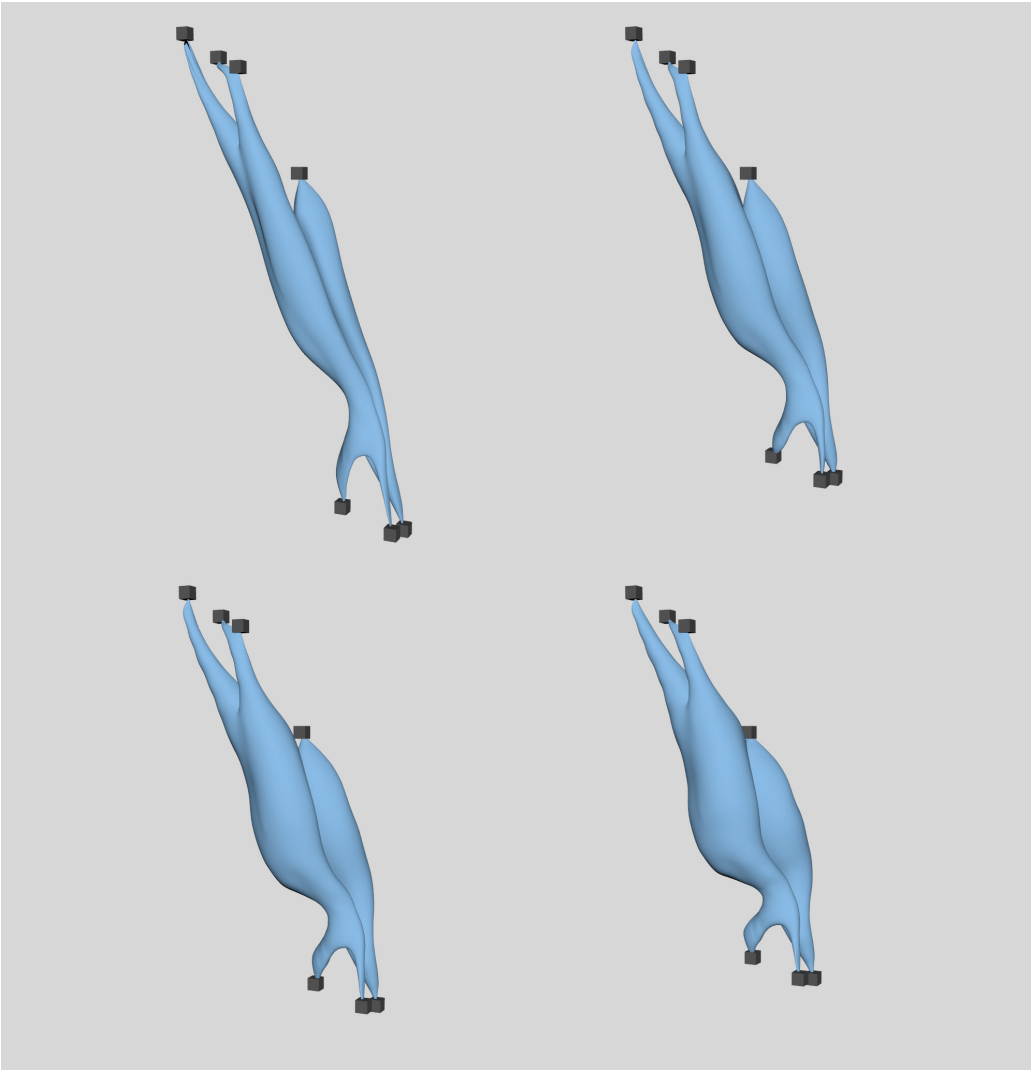

Figura 7.15: Activación de bíceps y brachialis anatómicos conectados entre sí con SMT-G (T4).

de la complejidad de la escena a nivel de número de músculos queda demostrada con este ejemplo (Figura 7.16), ya que las dinámicas y formas que terminan adquiriendo los músculos son adecuadas a lo que cabría esperar. Además, la interacción intermuscular responde bien a pesar de la densidad de vértices y del número de targets.

En lo que respecta al test T6, se lleva el sistema al límite en cuestiones de convergencia. Tal y como se ha visto en el apartado anterior 7.3.1.1, no es sencillo conseguir que éste sea capaz de mantener a los músculos bien acoplados para evitar que la simulación falle. Sin embargo, SMT-G supera el examen según los resultados observados en las Figuras 7.17 y 7.18. A pesar de la animática complicada del esqueleto, las dinámicas que se obtienen nos invitan a pensar que el enfoque basado en grafos es el enfoque correcto en comparación con los problemas experimentados con Jacobi. Gracias a que el Método de Grafos aplica, en última instancia, el

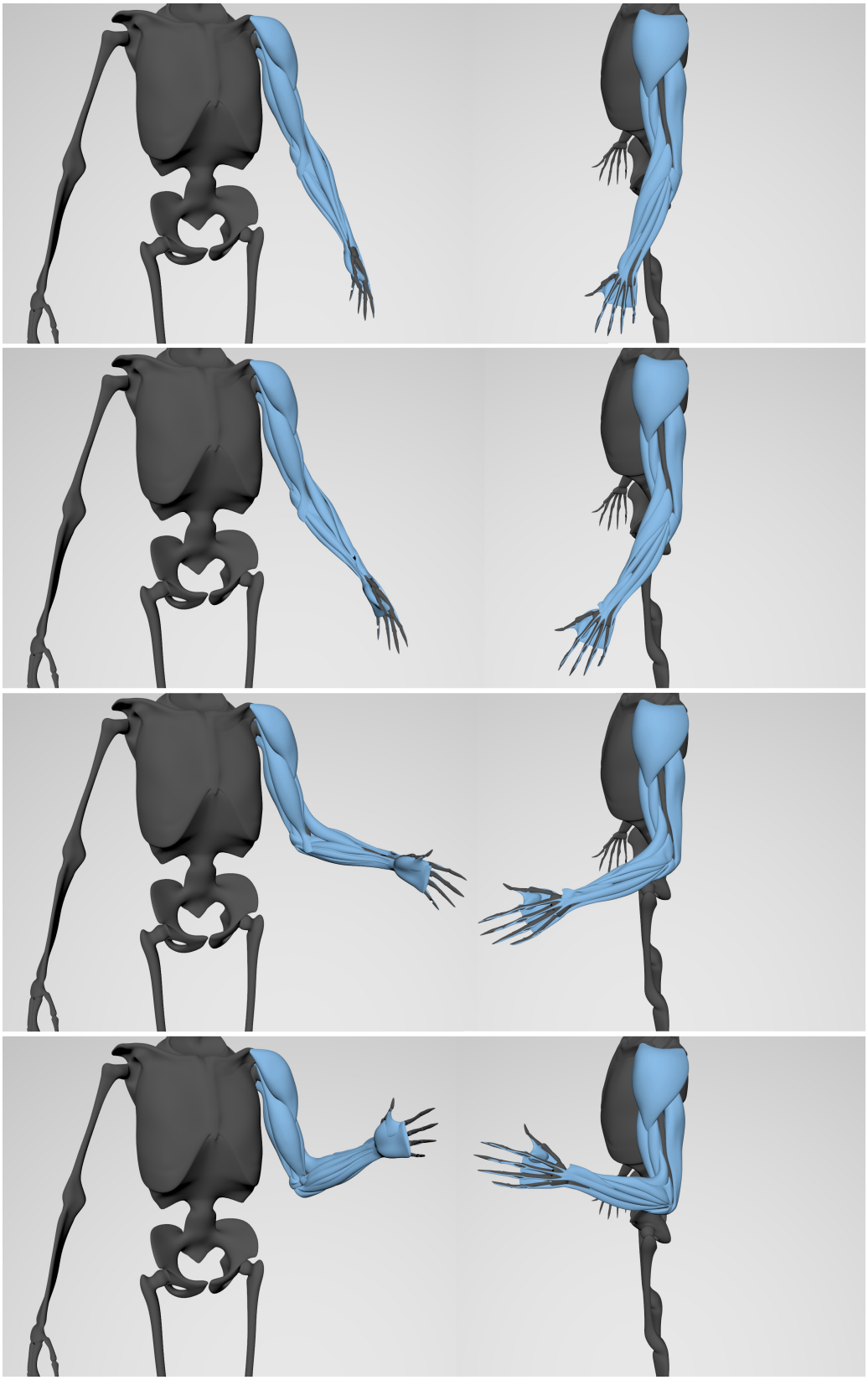

Figura 7.16: Simulación del brazo completo durante la flexión del codo con SMT-G (T5).

paradigma Gauss-Seidel a nivel de aplicación de los deltas de posición, hemos conseguido un modelo paralelizado de MSXPBD que cumple con los objetivos marcados para este proyecto.

Por último, en T7 simulamos el cuerpo completo para poner a prueba el resto de músculos del cuerpo. La respuesta que ha tenido SMT-G en este test ha sido muy buena. Como se puede ver en las Figuras 7.19 y 7.20, las dinámicas que se logran con el Método de Grafos son coherentes a la animación del esqueleto: la interacción entre músculos es correcta, no se perciben problemas como ocurría con Jacobi en este mismo test, la activación de las piernas se adecúa a los impactos con el suelo, etc. Si bien es cierto, podríamos decir que hay zonas con excesiva relajación, de nuevo, en la parte trasera de las piernas. Todo indica a que esta relajación se podría corregir incrementando el factor de `stiffness` en esos *MuscleNode* o bien aumentando el umbral de activación.

7.3.2. Cálculo de Costes

El estudio de costes que vamos a realizar sobre SMT-J y SMT-G utilizan los escenarios de simulación P1, P2, P3 y P4 establecidos en la Metodología 5.1 para comparar con los tiempos de SST. La comparativa que realizaremos será midiendo únicamente los cuatro bloques de ejecución sometidos a paralelización, a recordar: BE2, BE5, BE6 y BE7.

Pero antes de entrar en los detalles de cada uno, el análisis de todo sistema con paralelización nos obliga a definir primero un aspecto crucial: el número de hilos que podrán ejecutarse concurrentemente. La limitación de este número viene dada por las características del equipo en el que realizamos las pruebas, el cual quedó especificado en la Sección 5.3. Cabe recordar que el hardware en cuestión proporciona un total de 4 hilos reales que aumentan hasta 8 con el uso de *Hyper-threading*. Con estos datos, hemos sometido el equipo a examen para determinar un número fijo de hilos para todas nuestras simulaciones con SMT-J y SMT-G con la intención de que las comparativas sean lo más ajustadas posibles. Así, utilizando un mismo escenario de referencia (misma cantidad de músculos, restricciones, iteraciones y parametrización en general), hemos ido cambiando el número de threads que gestionará OpenMP, esto es, modificando el valor de la variable de entorno `OMP_NUM_THREADS`.

La Tabla 7.2 recoge los resultados de este experimento, mostrando el coste medio total por fotograma de las implementaciones con Jacobi y con Grafos a medida que varía `OMP_NUM_THREADS`. Los valores de esta tabla se muestran gráficamente en la Figura 7.21. Como se puede ver, la implementación con el Método Jacobi mejora en eficiencia a medida que aumenta el número de hilos permitidos, mientras que el sistema basado en Grafos produce sus mejores resultados con `OMP_NUM_THREADS` = 5. Conociendo esta discrepancia entre los dos modelos, vamos a tomar la decisión de completar los test de profiling de los apartados siguientes limitando el número de hilos en 5. Lo importante para nuestro estudio es tener datos que comparar bajo las mis-

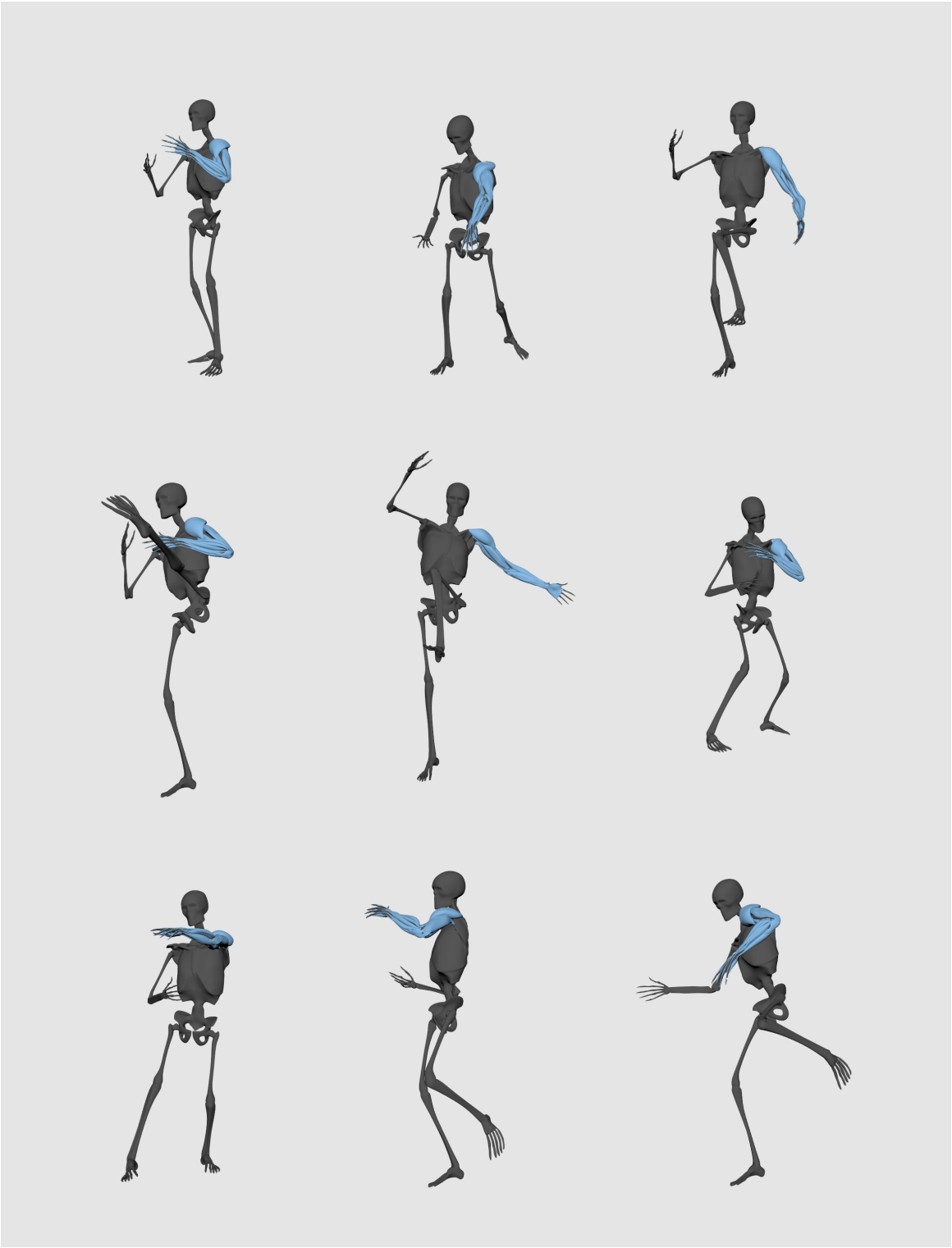

Figura 7.17: Simulación del brazo completo sobre movimientos de lucha con SMT-G (T6).

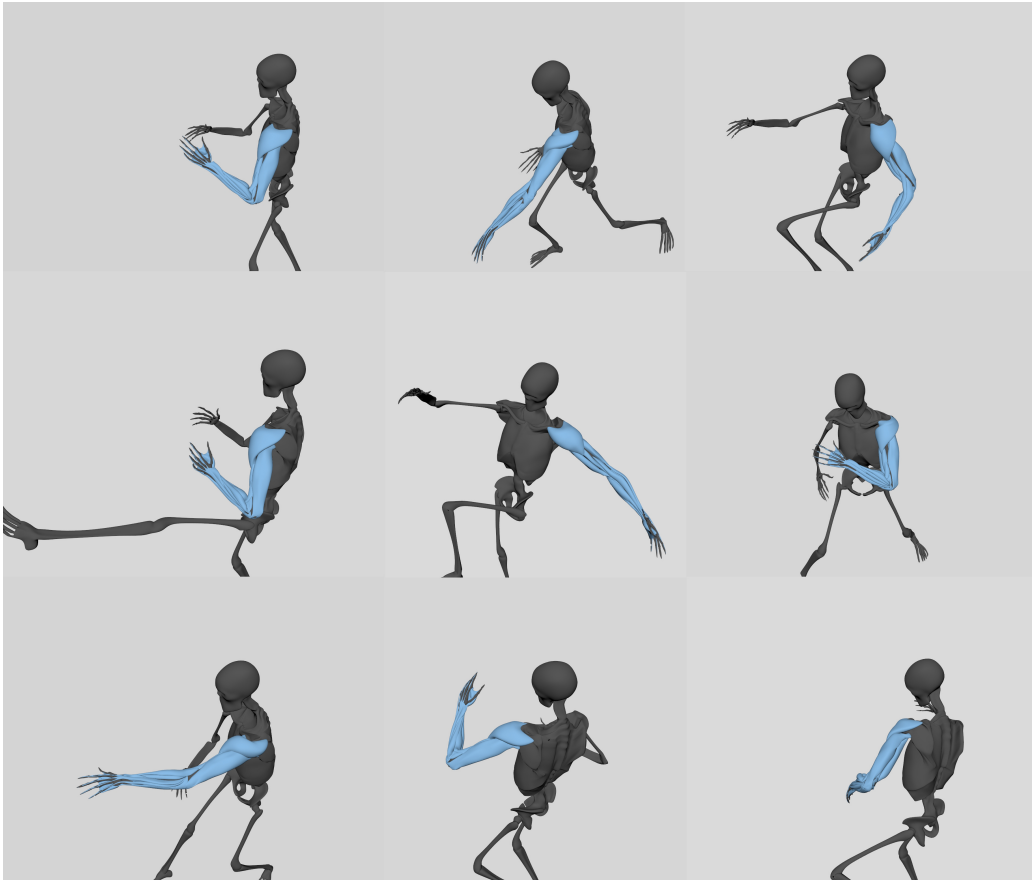

Figura 7.18: Perspectiva detalle de la simulación del brazo completo sobre movimientos de lucha con SMT-G (T6).

mas condiciones. Sabremos que Jacobi puede ofrecernos menores tiempos de cálculo con más cantidad de hilos, pero a nivel de comparativa es más conveniente que la configuración de los sistemas de simulación sea exactamente la misma.

Comenzando con una comparativa de tiempos en el test de profiling P1, se han obtenido los resultados de la Figura 7.22. Por un lado, podemos ver cómo BE2 y BE7 tienen coste mayor en los sistemas multi-threaded. A pesar de que pueda parecer erróneo, este dato tiene su explicación en que P1 simula únicamente un músculo, y por eso, una paralelización a nivel de músculo sólo supone una sobrecarga por la introducción y gestión de `OMP`. Por otro lado, en *Closest Data* (BE5), que es el bloque más costoso en SST, sí se obtiene una mejora considerable tanto con Jacobi como con Grafos: los 6,31ms de SST pasan a 2,28 y 2,07 de SMT-J y SMT-G respectivamente. Por último, la proyección de restricciones BE6 reduce su coste con Jacobi pero empeora gravemente con el Método de Grafos, tomando tiempos muy por encima incluso de la versión secuencial.

En el resto de test de profiling (P2, P3 y P4) se dan situaciones similares (ver Figuras 7.23, 7.24 y 7.25). En estos casos, el bloque BE2 (*Actualizar Colisionadores*) en las versiones

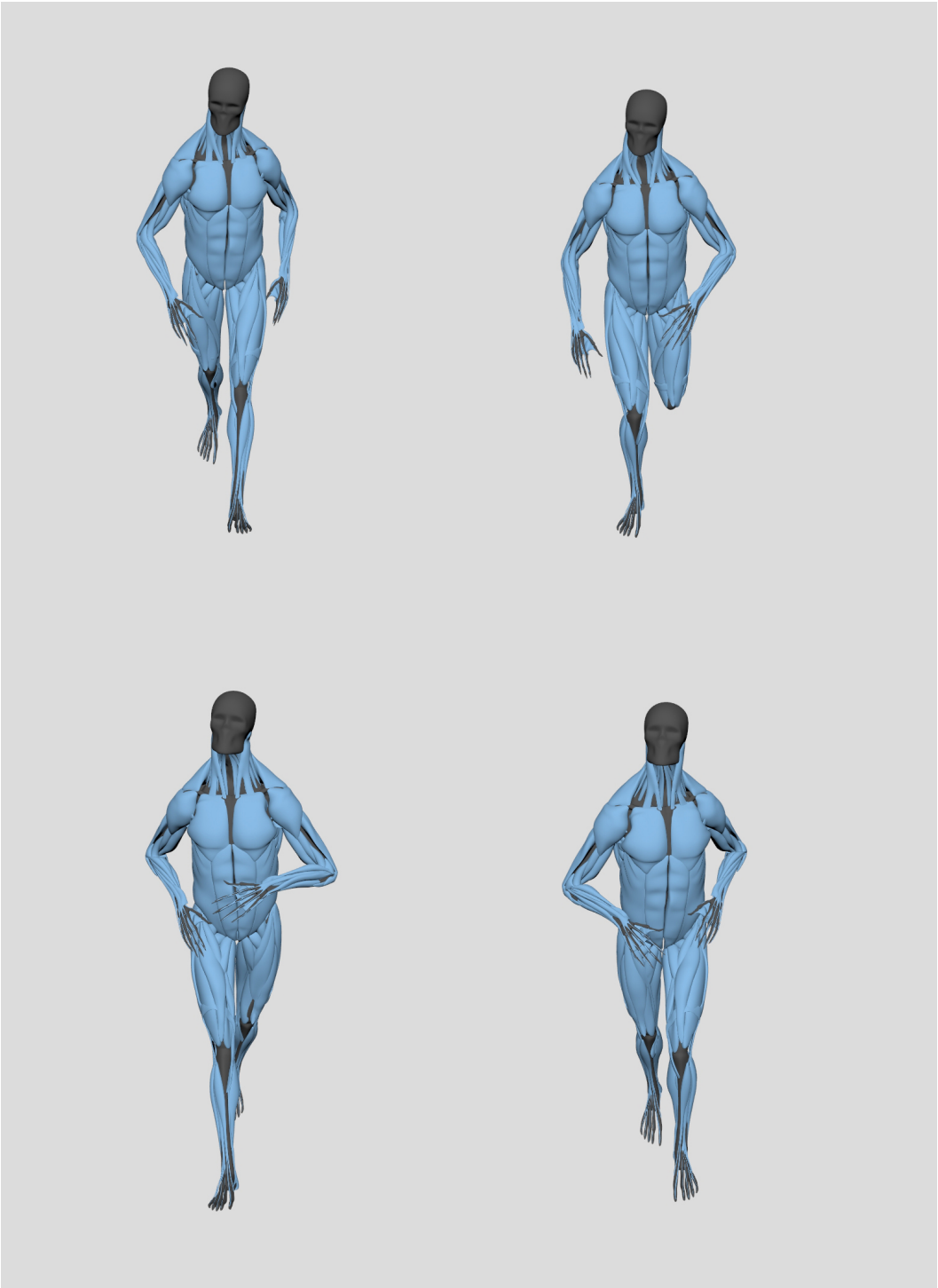

Figura 7.19: Simulación del cuerpo completo caminando y corriendo con SMT-G (T7).

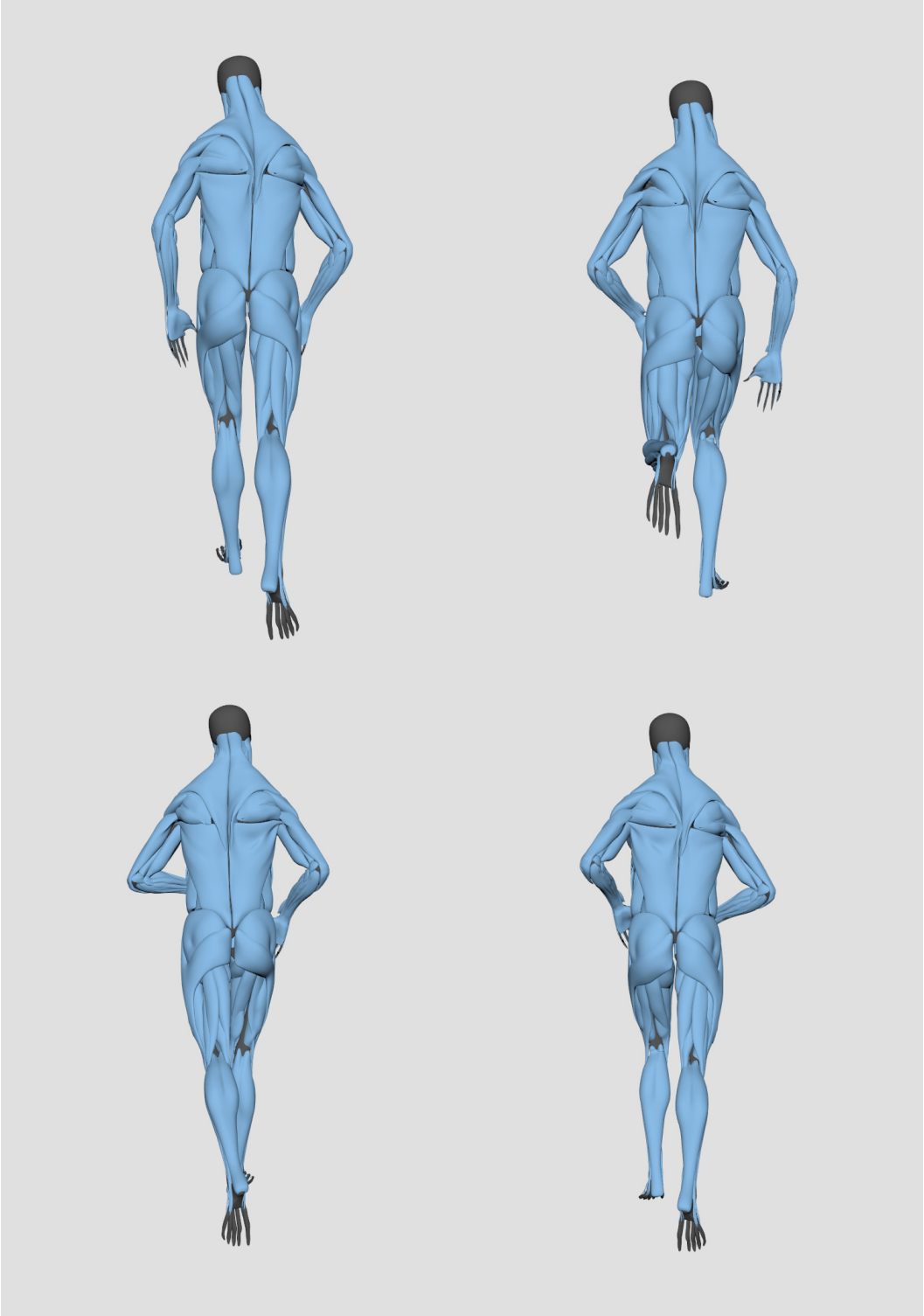

Figura 7.20: Simulación del cuerpo completo caminando y corriendo con SMT-G (T7).

Threads	SMT-J	SMT-G
1	379,02	354,60
2	222,66	222,24
3	171,62	165,90
4	151,70	155,35
5	140,44	154,93
6	136,41	161,14
7	130,49	165,48
8	124,76	168,46

Tabla 7.2: Estudio de tiempos (milisegundos por fotograma) con SMT-J y SMT-G en P1, P2, P3 y P4 en función del número de hilos.

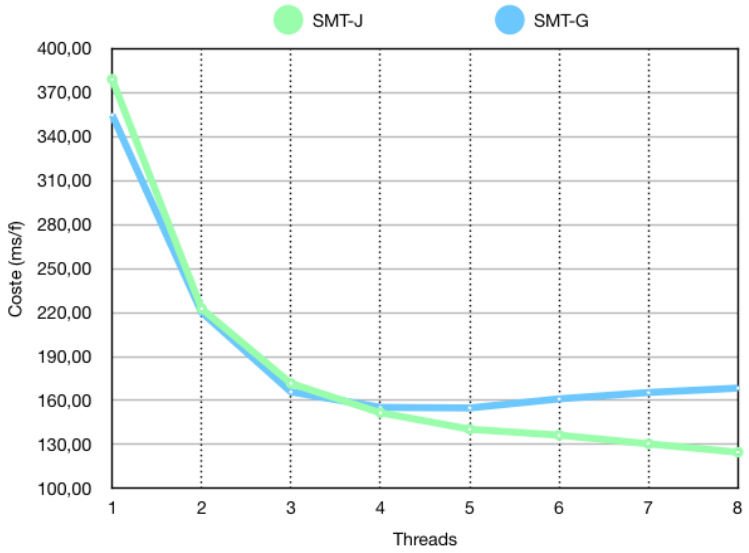

Figura 7.21: Gráfica de costes de cómputo según el número de hilos con SMT-J (verde) y SMT-G (azul) sobre el test T5.

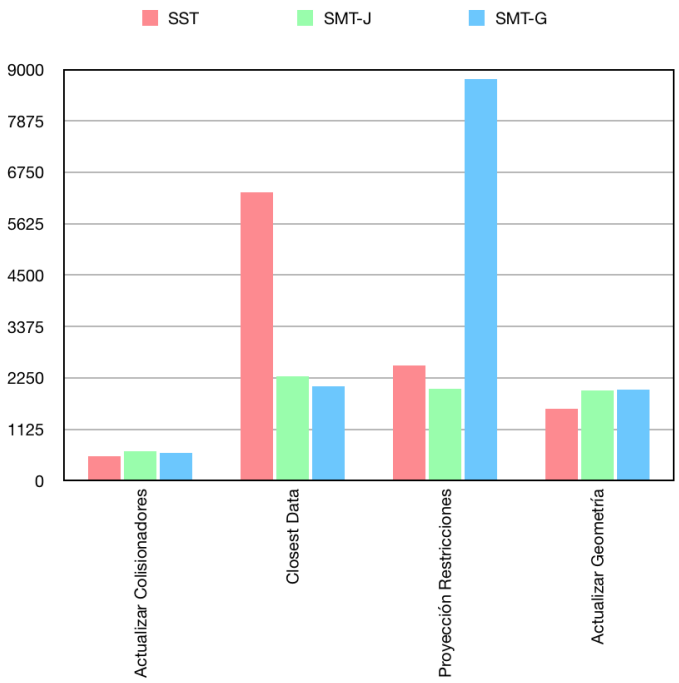

Figura 7.22: Gráfica comparativa de costes de cómputo de los bloques BE2, BE5, BE6 y BE7 con los sistemas SST (rojo), SMT-J (verde) y SMT-G (azul) en el test P1.

optimizadas tiene un coste ligeramente mayor que en la versión secuencial (sólo en P3 se obtiene mejora). Podemos decir que el nivel de complejidad de objetos colisionadores (en nuestro caso, huesos del esqueleto) es muy bajo como para que la paralelización de esta tarea sea beneficiosa. Dicho de otro modo, la sobrecarga asociada al uso de OpenMP no compensa en situaciones como las que manejamos, las cuales tienen uno o dos huesos solamente ya que en nuestro esqueleto se combinan los huesos anatómicos para formar geometrías más sencillas.

La tarea BE7, *Actualizar Geometría*, sí que mejora en las versiones optimizadas. Esta mejora, además, se acentúa en P3 y P4 ya que la cantidad de músculos que se computa es mayor, hasta 17. En este sentido, los resultados invitan a pensar que, a medida que crezca el número de músculos en el sistema, el porcentaje de beneficio también crecerá con SMT-J y SMT-G.

En lo referente al cálculo de *Closest Data* (BE5), la mejora que observábamos en P1 también se observa en P2, P3 y P4. Se trata de la tarea en la que más beneficio se obtiene con las optimizaciones implementadas. No es descabellado pensar que gran parte de la ganancia que obtengamos con SMT-J y SMT-G se deba a la diferencia en lo que a BE5 se refiere.

En las columnas relativas a la *Proyección de Restricciones* podemos ver que Jacobi reduce el tiempo de cómputo respecto a la versión secuencial pero, el método de Grafos sigue siendo mucho peor que SMT-J y también que SST. Por una parte, podemos concluir que la implementación del método Jacobi se ha realizado con éxito a tenor de estos resultados. Y por otra, los

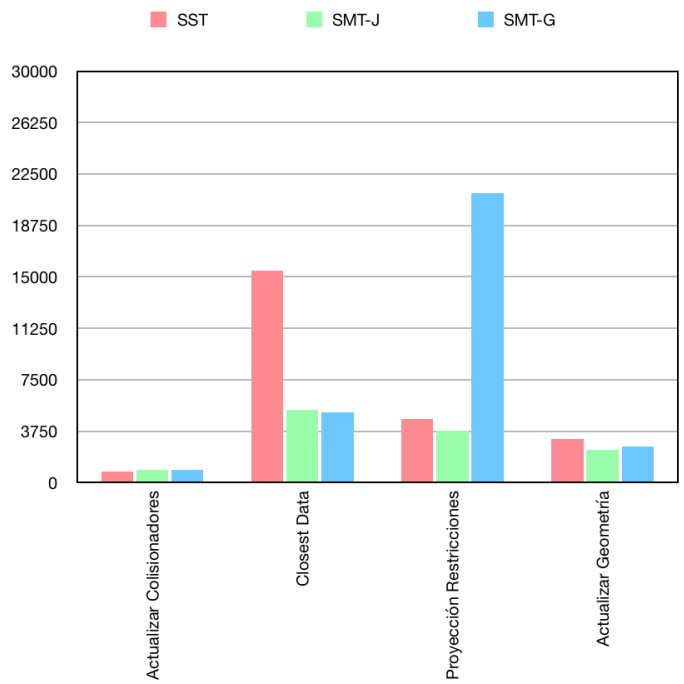

Figura 7.23: Gráfica comparativa de costes de cómputo de los bloques BE2, BE5, BE6 y BE7 con los sistemas SST (rojo), SMT-J (verde) y SMT-G (azul) en el test P2.

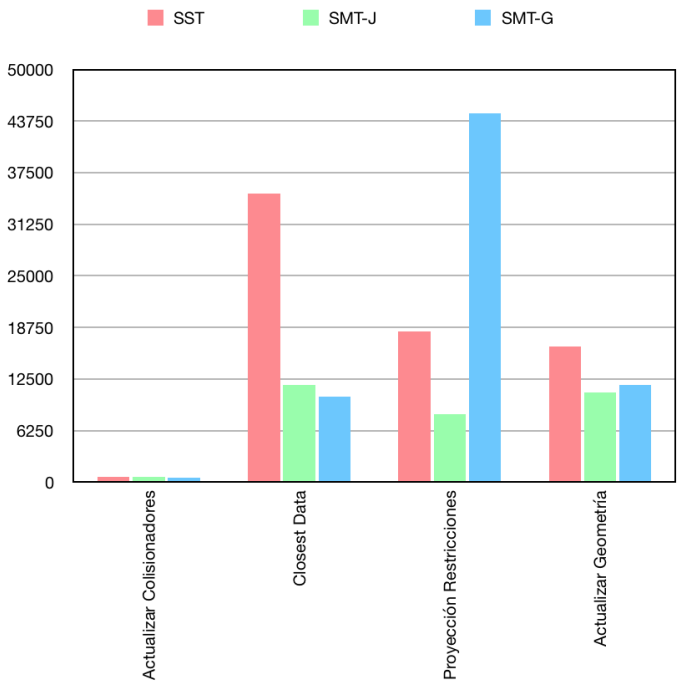

Figura 7.24: Gráfica comparativa de costes de cómputo de los bloques BE2, BE5, BE6 y BE7 con los sistemas SST (rojo), SMT-J (verde) y SMT-G (azul) en el test P3.

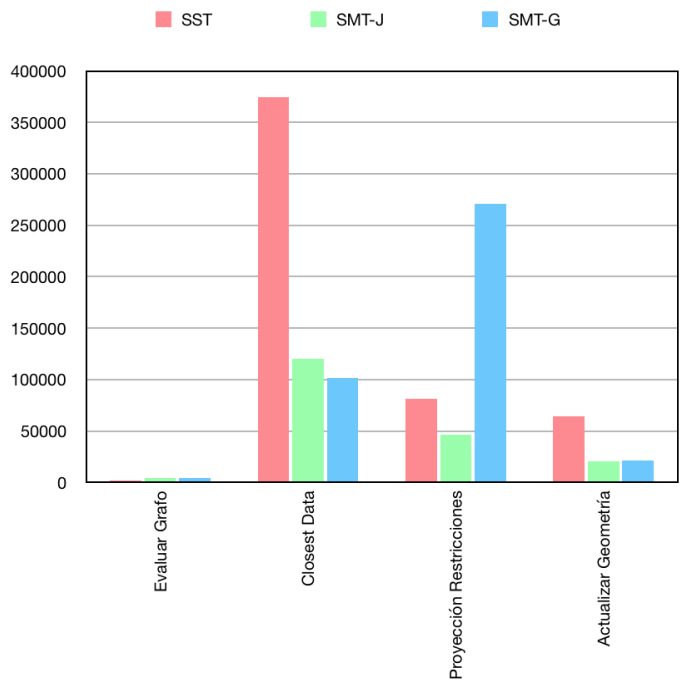

Figura 7.25: Gráfica comparativa de costes de cómputo de los bloques BE2, BE5, BE6 y BE7 con los sistemas SST (rojo), SMT-J (verde) y SMT-G (azul) en el test P4.

Test	Restricciones	Grupos	Promedio
P1	571	22	25,95
P2	1040	56	18,57
P3	4035	116	34,78
P4	18567	713	26,04

Tabla 7.3: Relación de restricciones y grupos generados en SMT-G en P1, P2, P3 y P4 con estructura interna en los músculos.

resultados tan negativos con SMT-G merecen que hagamos un especial análisis. Recordemos que la principal limitación de dicho método es que si el grado de compartición de las restricciones es muy alto, la eficiencia del sistema cae considerablemente porque el número de grupos aumenta y, además, muchos de esos grupos contienen pocas restricciones. La Tabla 7.3 recoge el número de restricciones y de grupos creados en cada test de profiling.

El número de grupos que se obtiene en todos los casos es alto si tenemos en cuenta el número de restricciones. Se han realizado varios análisis a nivel interno de los grupos para entender la composición de los mismos (ver Figura A.1 del Anexo A). Se ha detectado un número muy alto de grupos que tienen menos de cinco restricciones, incluso los hay con tan sólo una. La causa de esta vaga distribución se debe a la alta compartición de vértices en las restricciones de la estructura interna. Pensemos que todos los vértices de la malla de un músculo se conectan con

Test	Restricciones	Grupos	Promedio
P1	469	14	35,4
P2	854	21	40,67
P3	3333	17	196,05
P4	15169	143	106,08

Tabla 7.4: Relación de restricciones y grupos generados en SMT-G* en P1, P2, P3 y P4 sin estructura interna en los músculos.

un punto de la estructura interna. El número de puntos internos es muy pequeño comparado con el número de vértices en la malla, por ejemplo, en el caso del bíceps anatómico tenemos 20 puntos internos por 496 vértices. La consecuencia de esto es que la generación de restricciones relativas a la estructura interna fuerza la generación de un elevado número de grupos ya que cada punto interno puede llegar a conectarse con aproximadamente 30 vértices, es decir, genera 30 restricciones que obligatoriamente deben pertenecer a grupos diferentes por el hecho de afectar al mismo punto interno.

Si eliminamos la estructura interna, la cantidad de grupos que se generan es mucho menor tal y como muestra la Tabla 7.4 (consúltese también un ejemplo de esta otra distribución de grupos en la Figura A.2 del Anexo A). Si prestamos atención a los datos y los comparamos con la Tabla 7.3, podemos ver que los grupos disminuyen en cantidad pero aumentan en tamaño según indica la columna de promedio que divide el número de restricciones entre el número de grupos (téngase en cuenta que es una aproximación, no la distribución real). Esta nueva organización de grupos favorece el balanceo de los cálculos que necesita el Método de Coloreado de Grafos para ofrecer buenos resultados en cuanto a eficiencia.

Llegados a este punto, lo que se debe aclarar es si deshabilitar la estructura interna realmente corrige los malos resultados de coste obtenidos en los test de profiling y si, además, la calidad de la simulación no se ve afectada. En primer lugar, la Figura 7.26 demuestra que la nueva distribución de grupos sí mejora notablemente la eficiencia del método basado en Grafos (utilizaremos SMT-G* para referirnos a la versión sin estructura interna) en los test P3 y P4. De hecho, en esos casos ya produce mejores resultados que SST, corrigiendo los problemas detectados en los test previos.

En segundo lugar, para comprobar que la simulación reproduce dinámicas adecuadas con la versión SMT-G*, se ha simulado de nuevo el test T6. La idea es validar esta versión sobre el test más exigente en términos de convergencia del sistema. La Figura 7.27 nos confirma que así es y que, si deseamos hacer un uso eficiente del Graph-Coloring debemos deshabilitar la generación de estructura interna sin riesgos de perder estabilidad.

Terminando con el cálculo de costes, vamos a comparar el tiempo medio por fotograma en cada uno de los sistemas implementados en este proyecto (ver datos de la Tabla 7.5). La

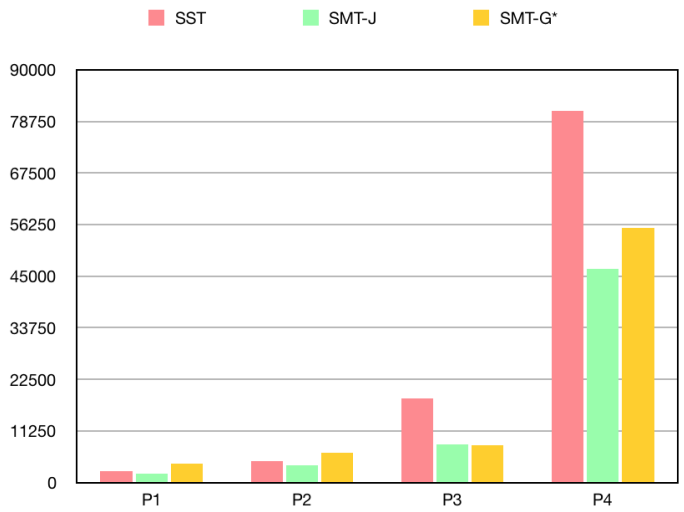

Figura 7.26: Gráfica comparativa de costes de cómputo de la proyección de restricciones con los sistemas SST (rojo), SMT-J (verde) y SMT-G* sin estructura interna (amarillo) en los test P1, P2, P3 y P4.

Profiling	**SST** (ms/f)	**SMT-J** (ms/f)	**SMT-G** (ms/f)	**SMT-G*** (ms/f)
P1	11,32	7,30	14,36	8,28
P2	24,57	12,89	30,80	13,37
P3	71,49	32,74	69,38	26,26
P4	531,92	198,96	406,68	152,49

Tabla 7.5: Tiempos absolutos en milisegundos por fotograma del SST, SMT-J y SMT-G con estructura interna y sin ella (SMT-G*) en P1, P2, P3 y P4.

conclusión más importante que debemos extraer es que el Método con Jacobi ofrece tiempos menores que SST y SMT-G en todos los test, excepto en P3 y P4 si lo comparamos con SMT-G*. Además, recordemos que Jacobi sería más eficiente si aumentáramos el número de hilos disponibles hasta 8 (los valores mostrados aquí se obtuvieron con 5 hilos solamente). Por otra parte, SMT-G (con estructura interna) sólo mejora al sistema secuencial en P4 y muy ligeramente en P3. Por último, cabe destacar que la modificación del sistema basado en grafos sin estructura interna (SMT-G*) es el más eficiente en P3 y P4. Es importante recalcar estos datos puesto que P3 y P4 (simulaciones con músculos anatómicamente correctos) se corresponden con escenarios típicos en los que un sistema de simulación de músculos debe responder bien. La Figura 7.28 representa los datos de la tabla en forma de gráfica.

Se han calculado los porcentajes de mejora respecto a SST de las optimizaciones implementadas, ver Tabla 7.6. Algunos de los datos más llamativos son los valores negativos de SMT-G en P1 y P2. Recordemos que estos dos escenarios son muy simples y hacen uso de pseudo-músculos como la esfera ovalada o el cubo cuyas geometrías no generan un gran número de

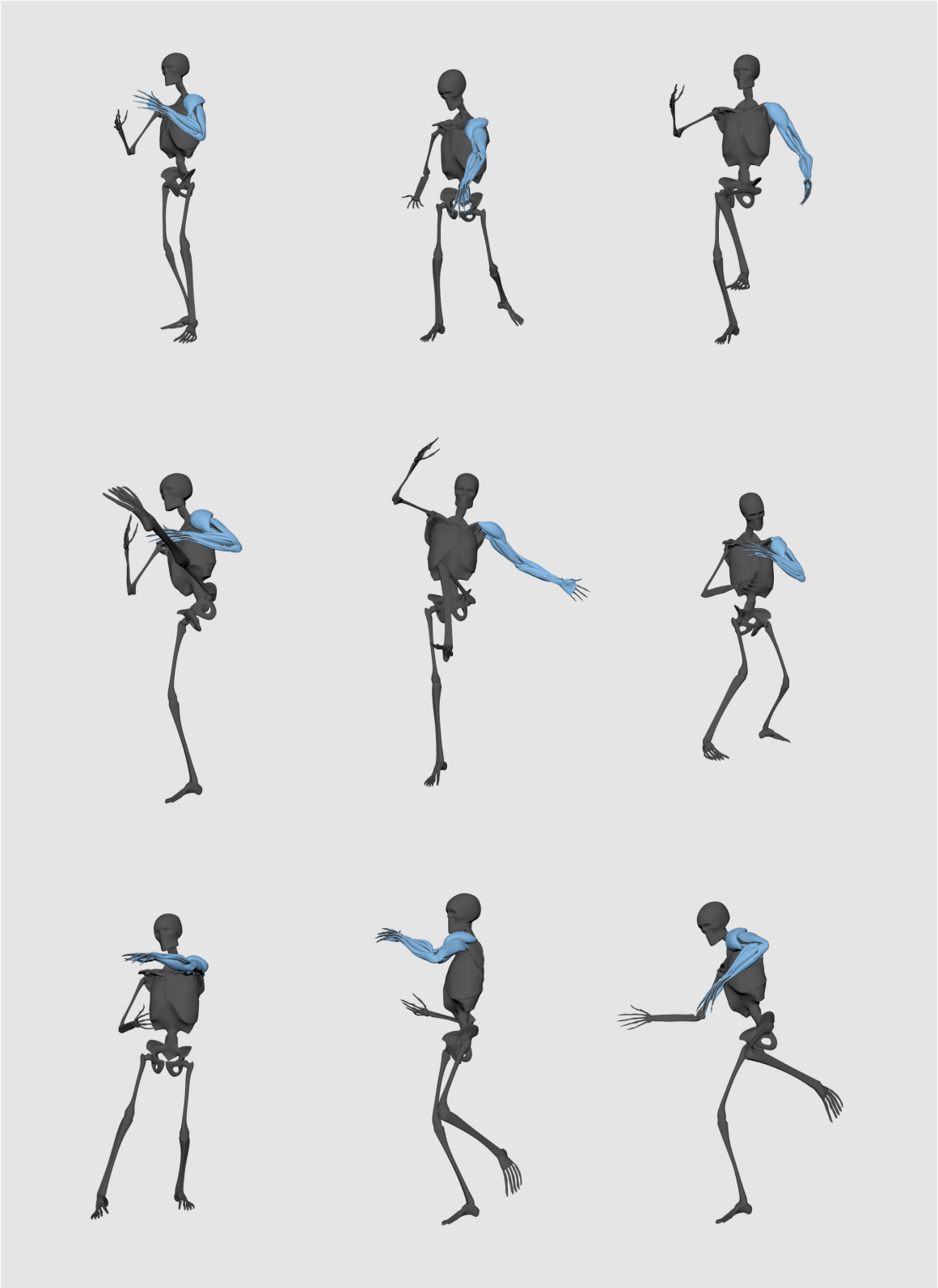

Figura 7.27: Simulación del brazo completo sobre movimientos de lucha con SMT-G sin estructura interna (T6).

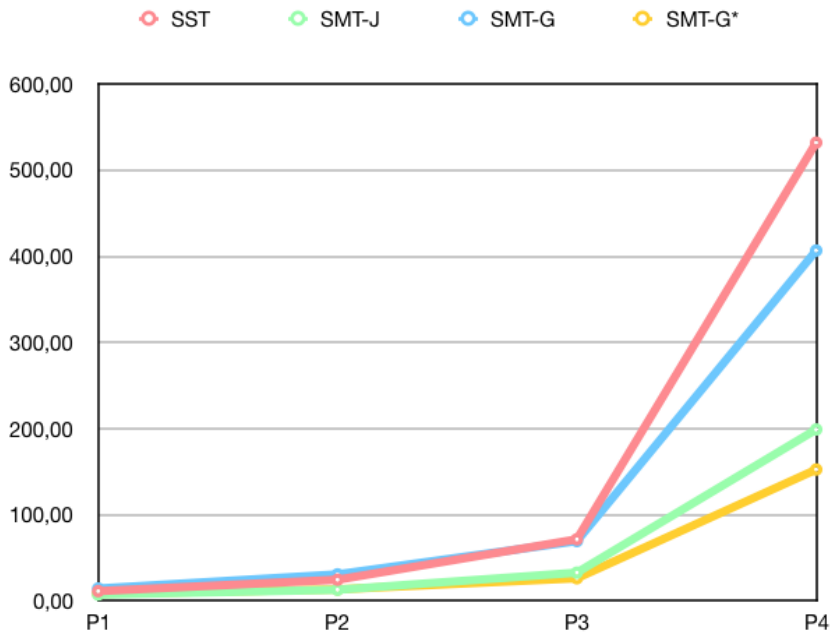

Figura 7.28: Comparativa de la evolución de tiempo medio por fotograma en P1, P2, P3 y P4 con los sistemas SST, SMT-J y SMT-G.

Profiling	**SMT-J** (%)	**SMT-G** (%)	**SMT-G*** (%)
P1	35,54	-26,79	26,88
P2	47,53	-25,34	45,59
P3	54,20	2,95	63,27
P4	62,60	23,55	71,33

Tabla 7.6: Porcentajes de mejora respecto al sistema secuencial en P1, P2, P3 y P4 de SMT-J, SMT-G y SMT-G*.

restricciones. En definitiva, son escenarios adecuados para pruebas pero poco exigentes para el modelo MSXPBD con nuestra optimización. Al margen de esto, lo que debemos destacar de todo el estudio realizado son los porcentajes de la última fila de la tabla relativos al test P4, ya que se trata de una escena con 17 músculos anatómicamente realistas. Precisamente en este test, es donde nuestras implementaciones ofrecen muy buenos resultados. La ganancia que se consigue es, en el peor de los casos, del 23,55 % con SMT-G. El Método Jacobi nos da una mejora del 62,60 % respecto al sistema single-threaded. Y la versión que mayor reducción de cómputo implica es la implementación con el Método de Coloreado de Grafos sin estructura interna, alcanzando el 71,33 % de ganancia en rendimiento.

Para finalizar, se quiere ofrecer una idea más real de cómo cambia la velocidad de cálculo en cada implementación. Para ello, se ha convertido el contenido de la Tabla 7.5 en valores de tasa

Profiling	SST (fps)	SMT-J (fps)	SMT-G (fps)	SMT-G* (fps)
P1	88.34	136.99	69.64	120.77
P2	40.70	77.58	32.47	74.79
P3	13.99	30.54	14.41	38.08
P4	1.88	5.03	2.46	6.56

Tabla 7.7: Tasas de refresco en fotogramas por segundo de SST, SMT-J y SMT-G con estructura interna y sin ella (SMT-G*) en P1, P2, P3 y P4.

de refresco (esto es, fotogramas por segundo), mostrados en la Tabla 7.7. Los test P1 y P2 se han conseguido simular con frame-rates muy altos en todas las versiones, incluyendo el sistema secuencial y destacando los 136,99 fps de SMT-J en P1. En cualquier caso, se trata de tiempos por encima (o muy por encima) del mínimo necesario para considerar que son simulaciones en tiempo real. No obstante, P1 y P2 son test con músculos de geometría sencilla.

El primer profiling que complica la escena es P3, donde tenemos el bíceps y el brachialis anatómicamente correctos. En este caso, SST y SMT-G están por debajo de los 15 fps, tasa que ya no consideramos tiempo real. Sin embargo, el método con Jacobi (30,54 fps) y el método de Grafos sin estructura interna (38,08 fps) sí alcanzan ese ratio deseado. Con todo, es P4 el escenario más adecuado para validar un sistema de estas características, ya que tenemos una cantidad importante de músculos con geometría realista.

De peor a mejor caso, comenzamos con el sistema secuencial que no pasa de los 1,88 fps. Le sigue de cerca la optimización con el método de Grafos con la estructura interna habilitada. A pesar de los problemas de balanceado y coloreado de grupos que tiene esta versión, consigue mejorar la tasa de refresco de la implementación single-threaded, llegando a los 2,46 fps. En tercer lugar se sitúa el sistema con Jacobi que sube hasta los 5,03 fps (de nuevo, téngase en cuenta la ganancia que obtendría esta versión si simulamos con 8 hilos concurrentes). Y finalmente, la optimización basada en Coloreado de Grafos sin estructura interna en los músculos obtiene la tasa de refresco más alta, llegando a los 6,56 fps.

7.3.3. Estudio comparativo con el modelo original

Siguiendo con el planteamiento realizado en el apartado 7.3.3, a continuación vamos comparar las versiones del sistema optimizadas con Jacobi y Grafos respecto al modelo original de Romeo et al. implementado en Python secuencial. Como recordatorio, decir que en el artículo original de MSXPBD se simula un humano completo a una velocidad de 44,04 segundos por fotograma. La configuración del escenario de pruebas, según los autores, consiste en una separación de la simulación en 6 procesos independientes, con un total de 135780 restricciones y 10 iteraciones. Dicho de otro modo, la versión Python del plug-in alcanza una tasa de refresco de

Sistema	Implementación	Tiempo (s/f)	Rendimiento (fps)
Romeo et al.	Python Single-Threaded	44.04	0.023
SST	C++ Single-Threaded	1.414	0.707
SMT-J	C++ Multi-Threaded Jacobi	0.685	1.46
SMT-G	C++ Multi-Threaded Grafos	1.138	0.879

Tabla 7.8: Resumen de costes (en segundos por fotograma y en fps) del modelo original en Python y los sistemas implementados en este proyecto: SST, SMT-J y SMT-G.

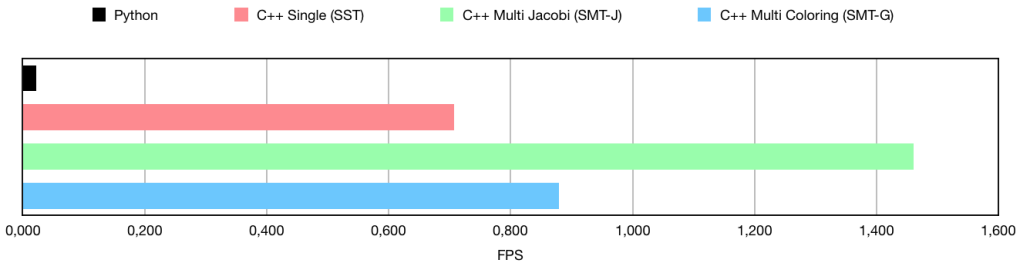

Figura 7.29: Comparativa de tasas de refresco (en fotogramas por segundo) entre las versiones (de arriba a abajo): Python, C++ single-threaded, C++ multi-threaded con el Método Jacobi y C++ multi-threaded con Graph Coloring.

0,023 fps. Recordemos también el equipo sobre el que se realizan esas mediciones, a saber, un Intel Core i7-4790K (4,00GHz) con 32GB de RAM.

En nuestro caso, la escena de simulación para este estudio también separa el humano en 6 partes para computar 10 iteraciones y 127270 restricciones (note el lector la diferencia de esta cantidad respecto al test de Romeo et al.). Bajo estas condiciones, por un lado, nuestro sistema multi-threaded con el Método de Jacobi ha logrado simular cada fotograma en una media de 0,685 segundos, es decir, con un frame-rate de 1,46 fps. Por otro lado, la versión con el Método de Coloreado de Grafos es algo más lenta, con un coste medio por fotograma de 1,138 segundos, esto es, 0,879 fps. Téngase en cuenta que dicha versión del enfoque con Grafos contiene la estructura interna, lo que significa que estamos ante una distribución de los grupos nada favorable para este método. A pesar de todo, tanto Jacobi como Grafos adquieren tasas de refresco muy por encima del modelo con Python. Concretamente, Jacobi es aproximadamente 63 veces más rápido y Coloreado de Grafos unas 38 veces más rápido. Todos estos datos están recopilados en la Tabla resumen 7.8.

Si mostramos los frame-rates en una gráfica de barras, Figura 7.29, podemos hacernos una idea visual de la diferencia de velocidades de cálculo. De arriba a abajo, se muestran las tasas de refresco en fotogramas por segundo de la implementación en Python de Romeo et al., nuestro Sistema single-threaded, nuestra versión multi-threaded con Jacobi y la versión con Graph-Coloring con estructura interna incluida.

Capítulo 8

Cierre del proyecto

8.1. Conclusiones

En la presente sección, exponemos las conclusiones que se extraen del desarrollo de este proyecto desde el punto de vista de los objetivos definidos en el Capítulo 4. En primer lugar, se ha re-diseñado por completo el modelo MSXPBD original de Romeo et al. para implementar una versión en C++11 single-threaded. Los inicios de este proyecto se centraron en lograr una versión compilada del plug-in, en nuestro caso, para Maya 2018 en MacOS. De esta versión, se puede afirmar que los objetivos en lo que a calidad de las simulaciones se refiere, han sido superados satisfactoriamente. En el apartado 6.4.1 se ha demostrado que el sistema es capaz de reproducir comportamientos realistas al simular un brazo completo con hasta 17 músculos anatómicamente correctos y también el cuerpo entero con 132 músculos divididos en 6 escenas de simulación independientes.

Algunas de las premisas marcadas para este desarrollo están ligadas a la aplicabilidad del sistema en la industria de los Efectos Visuales. De hecho, la optimización de MSXPBD perseguida no debía ir en detrimento de estas condiciones. Hablamos de la alta calidad de los resultados, la flexibilidad del sistema, la intuitividad y sencillez de parametrización y la libertad y control artísticos. En este sentido, el plug-in generado ha mantenido la misma configuración en cuanto a interfaz de usuario integrada en Maya que tenía el sistema presentado por Romeo et al. en [16].

Partiendo de la implementación secuencial, SST, se realizó un estudio de costes de cómputo para determinar qué bloques de ejecución tenían mayor impacto sobre el coste total, con el fin de focalizar el proceso de optimización en esos bloques. Dicho estudio confirmó que el cuello de botella se hallaba en el cálculo de información relacionada con las interconexiones entre músculos, esto es, cuestiones de puntos más cercanos, orientación de normales, transformaciones relativas, intersecciones, etc. En definitiva, operaciones geométricas con la API de Maya que se

realizan muchas veces a razón del número de vértices que se está simulando. También era crítica la proyección de restricciones propia del modelo MSXPBD, así como la lectura y escritura de vértices desde un punto de vista de comunicación entre la capa de aplicación y la capa core.

También ha sido necesario un estudio de los principales métodos de paralelización de PBD y XPBD, a saber, el Método de Jacobi y el Método de Coloreado de Grafos. Ambos métodos han sido aplicados sobre la versión secuencial para dar lugar a sendos sistemas multi-threaded (SMT-J y SMT-G). Para ello, se ha aplicado una metodología de análisis, diseño, implementación y pruebas de forma separada para finalmente obtener dos plug-ins compilados independientes con multi-threading. El enfoque de este proyecto basado en la aplicación de técnicas de optimización en el contexto de HPC ha obligado, además, a profundizar en el estudio y uso de OpenMP como herramienta de paralelización.

Si nos centramos en los objetivos exigidos sobre eficiencia, podemos retomar los datos obtenidos en la Tabla 7.6 en términos de costes de cómputo total. La tabla nos dice las reducciones de tiempo de ejecución de las versiones optimizadas respecto a la implementación single-threaded. Lo más importante a destacar es que en escenarios complejos con el brazo entero, es posible conseguir una reducción del 62,60 % con Jacobi y hasta del 71,33 % con el Método de Grafos. Se trata de mejoras en eficiencia con un margen muy amplio que nos permiten validar el desarrollo realizado en este proyecto y a considerar como satisfechos los objetivos de optimización impuestos.

Tanto es así, que si realizamos una comparativa con el sistema en Python de Romeo et al. del que partíamos, obtenemos datos todavía más positivos (recogidos en la Tabla 7.8). Los autores de MSXPBD comentan que consiguen resolver el cuerpo humano completo dividiendo la simulación en 6 procesos que suman un total de 135780 restricciones con 10 iteraciones en el solver en 44,04 segundos por frame, es decir, 0,023 fps. En nuestro caso, 127270 restricciones con 10 iteraciones nos da un coste de 1,414 (con la versión single-threaded), 0,685 (con Jacobi) y 1,138 (con Grafos) segundos por frame, o lo que es lo mismo, 0,707 fps, 1,46 fps y 0,879 fps. En otras palabras, en la simulación de todo el cuerpo hemos logrado un incremento del rendimiento en 30 veces con el sistema secuencial, 63 veces con Jacobi y 38 veces con Graph-Coloring, todas respecto al modelo MSXPBD original en Python.

Además, esta comparativa es injusta con nuestro sistema tal y como se ha explicado con anterioridad, ya que los experimentos están realizados en un equipo con menores prestaciones que el utilizado por Romeo et al. en su artículo. Esto acentúa la valoración positiva de la optimización que se ha completado en este proyecto. Además, en el artículo original se defiende que MSXPBD alcanza tiempos de cálculo equiparables a los de Ziva VFX, que es uno de los sistemas de simulación de músculos basados en FEM de uso más extendido en la industria. Por todo ello, podemos afirmar que las implementaciones llevadas a cabo aquí con Jacobi y

Coloreado de Grafos, se sitúan por delante de otras soluciones actualmente consolidadas en el mercado en términos de eficiencia.

Desde el punto de vista de la calidad de las simulaciones, los resultados obtenidos son comparables a los del modelo original de Romeo et al. Los test han demostrado que la activación de las fibras produce cambios coherentes en la rigidez, forma y volumen de los músculos, dando lugar a dinámicas que podemos percibir como realistas. Sin embargo, el sistema con el Método de Jacobi demostró ser inestable cuando los movimientos del esqueleto son explosivos, a pesar de que en otros escenarios más simples el resultado es bueno. Por su parte, el método basado en Grafos sí está a la altura tanto del modelo MSXPBD original como del sistema single-thread en SST.

Para finalizar la sección, es conveniente comentar cómo ha evolucionado el desarrollo del proyecto en comparación a la planificación realizada en el punto 5.4 de esta memoria (consultar Tabla 5.6). El Bloque 1 de tareas centrado en la definición y diseño del proyecto se completó satisfactoriamente dentro de los plazos establecidos. Por contra, los plazos de los Bloques 2 y 3 sufrieron retrasos. Por un lado, el Bloque 2 (Implementación del sistema en C++ *single-thread*), se dilató hasta cinco semanas, pasando del 31 de marzo previsto al 4 de mayo. El principal motivo fue que, dada la complejidad del sistema, surgieron numerosos problemas y *bugs* que requirieron un esfuerzo extra hasta que quedaron solventados y consolidados. Ese desfase en la Entrega 2, provocó a su vez el desfase en la entrega del Bloque 3 (Implementación del sistema en C++ *multi-thread*): se trasladó del 16 de junio al 30 de junio. Así y todo, lo más destacable es que el límite de fecha final de proyecto ha podido respetarse incluso con cinco días de antelación, esto es, se ha adelantado del 15 de julio previsto al día 10 del mismo mes. Mayoritariamente, este reajuste ha sido posible gracias a que el desarrollo de la documentación, tanto de esta memoria como de los vídeos demostrativos, ha sido realizo de forma paralela a medida que se iban completando las distintas tareas relacionadas.

En resumen, podemos concluir que el desarrollo de este proyecto ha cumplido con los objetivos académicos, de investigación, de resultados y de planificación fijados al comienzo.

8.2. Limitaciones y trabajo futuro

De todo el desarrollo realizado, es justo destacar la gran limitación encontrada en la implementación del Sistema Multi-Threaded con el Método Jacobi en lo que a estabilidad se refiere. Como se ha comentado en apartados anteriores, una de las desventajas del enfoque jacobiano es el exceso de relajación de las deformaciones debido al promedio que se realiza de los deltas de posición en los vértices. Esta aproximación introduce una pérdida de rigidez considerable que es crucial para nuestro sistema de músculos. Dados unos movimientos en el esqueleto de conside-

rable rapidez y desplazamiento, el solver no es capaz de aplicar las correcciones necesarias para mantener a los músculos en su sitio. Esto genera inercias muy altas que terminan provocando la inestabilidad de la simulación y, consecuentemente, un *crash* en la escena de Maya. Ni siquiera el factor de *successive over-relaxation* que se propone en la literatura es capaz de contrarrestar el problema.

Dadas las circunstancias, es posible que el uso del Método Jacobi integrado en MSXPBD requiera de un rediseño del modelo en sí mismo. Por eso, una de las líneas de trabajo futuro se centraría en tratar de generar nuevas estructuras y restricciones que aseguren la fijación de los músculos, al mismo tiempo que permitan dinámicas perceptibles en la superficie, es decir, sin que se conviertan en objetos totalmente rígidos como nos ha ocurrido al introducir las conexiones adicionales de tipo hard.

Por otra parte, en lo relativo al Método de Coloreado de Grafos, se ha detectado una limitación importante que tiene que ver con el balanceado de los grupos de restricciones. Hemos podido ver que el uso de estructura interna produce una elevada compartición de vértices entre restricciones que conlleva a la creación de muchos grupos de pequeño tamaño. Esto es muy perjudicial para la eficiencia de una paralelización basada en grafos como la que se ha planteado. Hemos podido comprobar también que con nuestro diseño se aplica la proyección de restricciones a nivel de músculo, por tanto, ha sido a ese nivel en el que se ha introducido el coloring y la paralelización. Es evidente que, si tenemos en cuenta un sólo músculo, el grado de compartición puede ser muy alto. Por eso, proponemos una vía de investigación que consiste en un rediseño del sistema completo para que la proyección de restricciones se haga a nivel general del solver, es decir, que sea el solver el que controla las restricciones de todos los músculos y así, pueda generar grupos de gran tamaño ya que no habrá compartición entre restricciones de distintos músculos. Esto generaría un mapa de grupos mucho más eficiente y adecuado para computar MSXPBD con paralelización basada en grafos.

En este proyecto, por cuestiones de alcance, no se ha considerado implementar un sistema híbrido que combine Jacobi con Coloreado. Llegados a este punto en el que hemos podido experimentar las ventajas y desventajas de cada uno en el caso práctico que nos ocupa, será muy interesante trabajar en un modelo que haga valer los puntos fuertes de uno y otro, con el fin de lograr una optimización todavía mejor de la obtenida hasta el momento.

Cuestiones que tienen que ver con las características tecnológicas del proyecto, y más concretamente con el hardware utilizado para las pruebas, nos invitan a preguntarnos qué grado de mejora conseguiríamos con la implementación actual si ejecutamos nuestro sistema en una máquina más potente acorde al perfil de prestaciones que se tiene en un estudio de VFX profesional. Será interesante comprobar en qué nivel de velocidad de cálculo nos sitúa respecto a los productos que se utilizan en la industria actualmente.

En el ámbito del software, otra de las líneas a cultivar tiene que ver con la compilación del plug-in para otros sistemas operativos comunes, es decir, Windows y Linux. Asimismo, uno de los motivos por los que este desarrollo ha tratado de separar completamente la capa de aplicación de la capa core es para poder integrar el plug-in en otros programas con relativa facilidad. Es por eso que en el futuro nos ocuparemos de diseñar e implementar nuestro sistema en aplicaciones como Houdini, Unity3D o Unreal Engine.

Y finalmente, desde una perspectiva de investigación y modelado intrínsecos a MSXPBD, heredamos aquí las líneas de trabajo futuro ya sugeridas por Romeo et al. que se centran en el estudio de las restantes capas de tejidos orgánicos que se contemplan en el contexto de la simulación de personajes: las fascia, la grasa y la piel.

Apéndice A

Método de Coloreado de Grafos: Ejemplos de composición de grupos

Esfera		Bíceps		Brachialis	
Grupo	Tamaño	Grupo	Tamaño	Grupo	Tamaño
0	54	0	281	0	95
1	51	1	277	1	93
2	55	2	285	2	95
3	57	3	293	3	98
4	78	4	402	4	147
5	90	5	477	5	165
6	80	6	481	6	164
7	43	7	119	7	44
8	9	8	22	8	7
9	9	9	19	9	7
10	9	10	18	10	7
11	9	11	18	11	5
12	4	12	18	12	5
13	4	13	18	13	5
14	4	14	16	14	5
15	2	15	15	15	5
16	2	16	15	16	5
17	2	17	15	17	5
18	2	18	12	18	5
19	2	19	12	19	5
20	2	20	12	20	4
21	2	21	12	21	4
		22	12	22	4
		23	11	23	4
		24	11	24	3
		25	11	25	3
		26	11	26	3
		27	11	27	3
		28	9	28	3
		29	9	29	2
		30	7	30	2
		31	7	31	2
		32	5	32	2
		33	5	33	2
		34	4	34	2
		35	4	35	2
		36	4	36	2
		37	4	37	2
		38	3	38	1
		39	3	39	1
		40	3	40	1
		41	3	41	1
		42	2	42	1
		43	2	43	1
		44	2	44	1
		45	2	45	1
		46	2	46	1
		47	2	47	1
		48	2	48	1
		49	1	49	1
		50	1	50	1
		51	1		
		52	1		
		53	1		
		54	1		
		55	1		
		56	1		
		57	1		
		58	1		
		59	1		
		60	1		
		61	1		
		62	1		
		63	1		
		64	1		

Figura A.1: Distribución de restricciones por grupo en la esfera, bícceps y brachialis con estructura interna.

Esfera		Bíceps		Brachialis	
Grupo	Tamaño	Grupo	Tamaño	Grupo	Tamaño
0	47	0	266	0	89
1	47	1	266	1	89
2	51	2	271	2	91
3	51	3	280	3	94
4	90	4	417	4	150
5	90	5	486	5	170
6	78	6	481	6	163
7	2	7	13	7	3
8	2	8	2		
9	2				
10	2				
11	2				
12	2				
13	2				

Figura A.2: Distribución de restricciones por grupo en la esfera, bíceps y brachialis sin estructura interna.

Bibliografía

[1] Autodesk. Maya API reference. https://download.autodesk.com/us/maya/2011help/API/main.html. Accessed: 2019-04-27.

[2] Jan Bender, Matthias Müller, and Miles Macklin. Position-based simulation methods in computer graphics. In *EUROGRAPHICS 2015 Tutorials*. Eurographics Association, 2015.

[3] Jan Bender, Matthias Müller, Miguel A. Otaduy, and Matthias Teschner. Position-based methods for the simulation of solid objects in computer graphics. In *EUROGRAPHICS 2013 State of the Art Reports*. Eurographics Association, 2013.

[4] Sean Comer, Jacob Buck, and Brice Criswell. Under the scalpel - ilm's digital flesh workflows. In *ACM SIGGRAPH 2015 Talks*, SIGGRAPH '15, pages 10:1–10:1, New York, NY, USA, 2015. ACM.

[5] M. Fratarcangeli and F. Pellacini. Scalable partitioning for parallel position based dynamics. *Comput. Graph. Forum*, 34(2):405–413, May 2015.

[6] H S Gasser and A V Hill. The dynamics of muscular contraction. *Proceedings of the Royal Society of London B: Biological Sciences*, 96(678):398–437, 1924.

[7] James Jacobs, Jernej Barbic, Essex Edwards, Crawford Doran, and Andy van Straten. How to build a human: Practical physics-based character animation. In *Proceedings of the 2016 Symposium on Digital Production*, DigiPro '16, pages 7–9, New York, NY, USA, 2016. ACM.

[8] M. Kelager, S. M. Niebe, and K. Erleben. *A Triangle Bending Constraint Model for Position-based Dynamics*, pages 31–37. The Eurographics Association, 2010.

[9] Anders Langlands. SIGGRAPH now: War for the planet of the apes. https://youtu.be/txoEDIdbUrg, 2017. Accessed: 2018-11-16.

[10] Dongwoon Lee, Michael Glueck, Azam Khan, Eugene Fiume, and Ken Jackson. Modeling and simulation of skeletal muscle for computer graphics: A survey. *Found. Trends. Comput. Graph. Vis.*, 7(4):229–276, April 2012.

[11] M. Macklin and M. Müller. Position based fluids. *ACM Trans. Graph.*, 32(4):104:1–104:12, July 2013.

[12] Miles Macklin, Matthias Müller, and Nuttapong Chentanez. Xpbd: Position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games*, MIG '16, pages 49–54, New York, NY, USA, 2016. ACM.

[13] Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Trans. Graph.*, 33(4):153:1–153:12, July 2014.

[14] Andy Milne, Mark McLaughlin, Rasmus Tamstorf, Alexey Stomakhin, Nicholas Burkard, Mitch Counsell, Jesus Canal, David Komorowski, and Evan Goldberg. Flesh, flab, and fascia simulation on zootopia. In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, pages 34:1–34:2, New York, NY, USA, 2016. ACM.

[15] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *J. Vis. Comun. Image Represent.*, 18(2):109–118, April 2007.

[16] Marco Romeo, Carlos Monteagudo, and Daniel Sánchez-Quirós. Muscle Simulation with Extended Position Based Dynamics. In Ignacio García-Fernández and Carlos Ureña, editors, *Spanish Computer Graphics Conference (CEIG)*. The Eurographics Association, 2018. CEIG 2018 best paper award.

[17] Jaewoo Seo, Yeongho Seol, Daehyeon Wi, Younghui Kim, and Junyong Noh. Rigging transfer. *Computer Animation and Virtual Worlds*, 21(3-4):375–386, 2010.

[18] SideFX. Houdini fx 16.5. https://www.sidefx.com/products/houdini-fx/. Accessed: 2018-11-16.

[19] Tim Steele, J. C. Leprevost, and Kriss Gossart. A position-based dynamics system for animated character effects. In *ACM SIGGRAPH 2014 Talks*, SIGGRAPH '14, pages 55:1–55:1, New York, NY, USA, 2014. ACM.

[20] J. Teran, S. Blemker, V. Ng Thow Hing, and R. Fedkiw. Finite volume methods for the simulation of skeletal muscle. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 68–74, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[21] Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. Robust quasistatic finite elements and flesh simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, pages 181–190, New York, NY, USA, 2005. ACM.

[22] Fabio Turchet, Marco Romeo, and Oleg Fryazinov. Physics-aided editing of simulation-ready muscles for visual effects. In *ACM SIGGRAPH 2016 Posters*, SIGGRAPH '16, pages 80:1–80:2, New York, NY, USA, 2016. ACM.

[23] Ziva VFX. Ziva VFX. http://www.zivadynamics.com/ziva-vfx. Accessed: 2018-11-16.

[24] Weta. Tissue. https://www.wetafx.co.nz/research-and-tech/technology/tissue/. Accessed: 2018-11-16.

[25] Felix E Zajac and Gordon E Michael. Determining muscle's force and action in multi-articular movement. *Exercise and Sport Sciences Reviews*, 17(1):187–230, January 1989.