Grado en Ingeniería Informática

Computación

Proyecto de Fin de Grado

Simulación de telas mediante Sistemas de Partículas

Autor

Ibon Merino Bermejo

Directoras:

María del Carmen Hernandez Gomez

Itziar Baragaña Garate

2017

Agradecimientos

Para empezar con este apartado me gustaría agradecer a mis dos directoras de proyecto Carmen Hernández e Itziar Baragaña todo el apoyo que me han ofrecido a lo largo del proyecto y en las distintas asignaturas que me han impartido clase. Por todo esto y mucho más, gracias.

Quisiera agradecer el apoyo que me han dado mis padres y mi hermano durante todos los años de mi carrera, tanto en los estudios como en mi vida personal.

También quisiera agradecer a todos los profesores que de una forma u otra han influido en mi desarrollo como profesional y como persona.

Por último, quiero agradecer a todas las personas que de alguna manera han estado presentes en mi vida, como mis amigos, compañeros de clase, compañeros de trabajo...

A todos y cada uno de ellos gracias.

Resumen

El objetivo principal de este proyecto es la creación de un sistema que simule de forma realista el comportamiento de telas basándose en un sistema de partículas. Para ello, se estudiarán las distintas técnicas de modelado y las propiedades físicas necesarias para el correcto funcionamiento de este.

The main objective of this project is to create a system to simulate in a realistic way the behavior of cloth based on a particle system. For that, the different modelling techniques and the physical properties necessary for the proper running will be studied.

Proiektu honen helburu nagusia ehunen portaera simulatzen duen partikula sistema batean oinarritutako sistema egitea da. Hori egiteko, modelatze teknika ezberdinak eta hauek bidezko portaera edukitzeko behar dituzten propietate fisikoak ikertu egingo dira.

Índice general

Agradecimientos I

Resumen III

Índice general V

Índice de figuras IX

Indice de tablas XI

1. Introducción 1

1.1. Contexto . . . 1
1.2. Propuesta . . . 2
1.3. Motivación personal . . . 2
1.4. Estado del arte . . . 2

2. Documento de los objetivos del proyecto 5

2.1. Objetivos del proyecto (Alcance y exclusiones) . . . 5
2.2. Herramientas utilizadas . . . 5
2.3. Aprendizaje . . . 9
2.4. Planificación . . . 9
2.5. Análisis de riesgos . . . 9
2.6. Análisis de factibilidad . . . 10

3. Desarrollo del proyecto 11

3.1. Modelo geométrico 11

3.2. Modelo dinámico 13

3.2.1. ¿Qué es una partícula? 15

3.2.2. Descripción de las características de una partícula 15

3.2.3. Ciclo de vida de una partícula 16

3.2.4. Relaciones entre partículas 20

4. Diseño 23

4.1. Inicialización 23

4.1.1. Escena 24

4.1.2. Objetos 24

4.1.3. Herramientas de control y ayuda 26

4.2. Animación 28

4.2.1. Simulación 28

4.2.2. Renderizado 29

4.3. Librerías propias 29

4.3.1. Particle.js 30

4.3.2. Cloth.js 32

4.3.3. Cloth.html 36

5. Análisis del proyecto 39

5.1. Complicaciones 39

5.2. Gestión del proyecto 40

5.3. Resultados 40

6. Conclusiones 45
6.1. Conclusiones generales del proyecto 45
6.2. Líneas futuras 46
Anexos
A. Experimento de las masas de las partículas 49
A.1. Peso fijo 49
A.2. Pesos siguiendo una sinusoidal 49
A.3. Pesos aleatorios 49
A.4. Conclusión 50
B. Implementación 51
B.1. Particle.js 51
B.2. Cloth.js 54
B.3. Cloth.html 60
C. Simulación de una bandera 75
C.1. Particle.js 76
C.2. Cloth.js 79
C.3. Cloth.html 86
Bibliografía 101

Índice de figuras

2.1. Ejemplo Three.js. 8
2.2. Tom's Planner. 9

3.1. Geometría de la tela. 11
3.2. Caras. 12
3.3. Triángulos de la tela. 13
3.4. Sistema de partículas 14
3.5. Vecindario. 14
3.6. Vecindad de orden *n*. 15
3.7. Relaciones entre partículas. 21

4.1. Arquitectura del PFG. 23
4.2. Estructura de la inicialización 24
4.3. La tela 25
4.4. La bola 25
4.5. Estadísticas de la escena 26
4.6. El menú 27
4.7. Los ejes 27

5.1. Un instante en la simulación de la tela en el aire 41

5.2. Otro instante en la simulación de la tela en el aire 41

5.3. Un instante en la simulación de la tela en colisión con la bola 42

5.4. Un instante en la simulación de la tela en colisión con el suelo 42

5.5. Zoom al instante en la simulación de la tela en colisión con el suelo . . . 43

C.1. Foto a la simulación de la bandera 76

Indice de tablas

3.1. Caras por vértice 12

4.1. Nomenclatura 29

5.1. Imputación de horas en el proyecto 40

1. CAPÍTULO

Introducción

1.1. Contexto

La animación por computador, animación digital o animación por ordenador es la técnica que consiste en crear imágenes en movimiento mediante el uso de un ordenador. Durante los primeros 20 años, década de los 60 y 70, se usaron por primera vez imágenes creadas por ordenador. Este hecho no tuvo una gran repercusión en la industria del cine y la televisión ya que esta tecnología aún era muy rudimentaria. Una década más tarde, en los 80, los primeros largometrajes, *Tron* (1982) y *The Last Starfighter* (1984), ya incluían planos creados por ordenador.

Toy Story de *Pixar*, estrenado en 1995, fue el primer largometraje totalmente generado por ordenador. Fue en este momento cuando los distintos estudios de animación empezaron a incorporar todas las técnicas que se habían estado investigando hasta la fecha pero que no se podían implementar por falta medios físicos; es decir, los ordenadores no eran lo suficientemente robustos y rápidos para procesar esa gran cantidad de datos.

Ya en 2001 se intentó revolucionar el mundo de la animación creando una película realista usando sólo imágenes generadas por ordenador, sin actores. La película, *Final Fantasy: The Spirits Within* fue producida por *Square Pictures*. Desgraciadamente la película no tuvo el éxito esperado en taquilla lo que llevó al cierre de este estudio. Aún así sirvió como prólogo a la película *The matrix reloaded*.

1.2. Propuesta

Nuestra propuesta se centra en un área de los gráficos de computador muy concreta que abarca tanto el modelado gráfico como la animación: el modelado de ropa. Se trata de un proyecto que englobará lo aprendido en un corto periodo de aprendizaje y se utilizarán conceptos asimilados en las asignaturas de la rama de Computación. Para ello, se creará un modelo virtual de una tela con distintas interacciones con elementos como pueden ser la gravedad, colisiones con una bola...

Para la creación de este modelo físico y la definición de sus consiguientes propiedades dinámicas, se realizará un estudio de los distintos tipos de modelado, tras lo cual se decidirá el tipo de modelado a implementar en este proyecto.

1.3. Motivación personal

Desde pequeño me ha fascinado la animación y los videojuegos. Podía pasarme horas viendo películas de animación y jugando a videojuegos. Mucha gente considera eso una pérdida de tiempo, pero creo que gracias a todo ese tiempo dedicado a verlas se estimuló mi creatividad. Por eso pienso aprovechar esta oportunidad para demostrar que esos mundos fantásticos y ficticios sirven para algo y para que más personas disfruten de ellas como yo lo hice antaño y continúo haciéndolo.

Por lo tanto, decidí hacer mi Proyecto de Fin de Grado en algo relacionado con los gráficos por ordenador, para profundizar más en los conocimientos adquiridos en la carrera en la especialidad de Computación.

1.4. Estado del arte

El modelado y la simulación de ropa han ido ganando consistencia a medida que la tecnología avanzaba gracias al gran desarrollo del hardware y del software tanto en la simulación como en el modelado. En la década de los 80, los primeros y más simples modelos fueron apareciendo. La primera aplicación fue en 1987 [Terzopoulos et al., 1987] y se basó en las ecuaciones de Lagrange de la dinámica y de las superficies de energía elástica. Esto, por ejemplo, permitió simular el movimiento de una bandera.

En cuanto al modelado de la ropa es imprescindible citar el artículo [Ng and Grimsdale, 1996] y el libro [House and Breen, 2000]. El primero de ellos hace un resumen de las 19 técnicas de modelado de ropa que había hasta el momento y que aún se consideran de las más importantes. Esas técnicas se categorizan en geométricas, que no tienen en cuenta las propiedades físicas de la ropa y sólo se basan en la apariencia; físicas, que se fundamentan en las propiedades físicas de la ropa tales como la elasticidad, la gravedad...; o híbridas, que combinan técnicas geométricas y físicas.

Tras años de investigación y la llegada de las potentes máquinas que tenemos hoy en día, el estudio actual se centra en las colisiones, el contacto y la fricción de la animación de ropa; es decir, en el modelado físico, o más bien híbrido ya que también se busca una apariencia realista de la ropa. Se han hecho grandes avances gracias a [Bridson et al., 2002] con su eficaz tratamiento de las colisiones y la posterior aportación y mejora de éste por parte de [Selle et al., 2009] con el robusto modelado de ropa de alta definición usando paralelismo, las colisiones basadas en la historia y la fricción precisa.

2. CAPÍTULO

Documento de los objetivos del proyecto

2.1. Objetivos del proyecto (Alcance y exclusiones)

El alcance del proyecto se recoge en varios apartados: objetivos de aprendizaje y objetivos de diseño.

Los objetivos de aprendizaje son:

1. Aprender a utilizar la librería de Javascript **Three.js**.
2. Estudiar los distintos tipos de modelado.
3. Estudiar la matemática y la física que hay tras la animación textil.

Los objetivos de diseño son:

1. Elección del modelo matemático (tipo de modelado).
2. Elección de las distintas herramientas para la visualización del modelo.
3. Implementación de la física y visualización de la animación resultante.

2.2. Herramientas utilizadas

En este proyecto se han utilizado dos herramientas: *Three.js* y *Tom's planner*.

2.2.0.1. Three.js

Es una biblioteca liviana escrita en JavaScript para crear y mostrar gráficos animados por ordenador en 3D en un navegador Web que puede ser utilizada en conjunción con el elemento canvas de HTML5, SVG o WebGL. El código fuente está alojado en un repositorio con dirección Three.js:Github.

La página oficial donde se pueden encontrar la documentación y una gran cantidad de ejemplos en Three.js es: http://threejs.org/

Three.js es una herramienta muy sencilla de utilizar que otorga gran facilidad al usuario para crear animaciones 3D. A continuación, se explicarán los pasos a seguir para crear una escena sencilla:

1. Incluir la librería.

```
<script src="../libs/three.js"></script>
```

2. Crear la escena: Objeto de Three.js que guarda todos los elementos de la escena a visualizar.

```
scene = new THREE.Scene();
```

3. Crear la cámara: Objeto de Three.js mediante el cual veremos la escena.

```
camera = new    THREE.PerspectiveCamera( 30, window.innerWidth /
    window.innerHeight, 1, 1000 );
camera.position.x = -20;
camera.position.y = 30;
camera.position.z = 40;
camera.lookAt(new THREE.Vector3(10, 0, 0));
```

   En este caso hemos creado una cámara en perspectiva.

4. Añadir iluminación a la escena: Objetos de Three.js que permiten iluminar la escena.

```
ambientLight = new THREE.AmbientLight( 0x666666 );
spotLight = new THREE.SpotLight(0xffffff);
spotLight.position.set(-40, 60, -10);
```

5. Añadir los objetos que se van a visualizar: Objetos básicos de Three.js (primitivas) tales como una esfera, un cubo... u objetos generados a partir de la combinación y transformación de estas primitivas.

```
ballGeometry = new THREE.SphereGeometry( 4, 20, 20 );
ballMaterial = new THREE.MeshPhongMaterial( { color: 0xaaaaaa } )
    ;
ballMesh = new THREE.Mesh( ballGeometry, ballMaterial );
```

No debemos olvidar que debemos añadir todo elemento creado a la escena:

```
scene.add(camera);
scene.add(ambientLight);
scene.add(spotLight);
scene.add(ballMesh);
```

6. Crear el *renderer*: Objeto de Three.js que se encarga de visualizar la escena.

```
renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setPixelRatio( window.devicePixelRatio );
renderer.setSize( window.innerWidth, window.innerHeight );
renderer.shadowMap.enabled = true;
```

Hay que añadir al cuerpo del documento html el código HTML del renderer.

```
document.body.appendChild(renderer.domElement);
```

7. Crear la función *render*: Mediante esta función, que se llamará constantemente, se dibujan los objetos de la escena desde el punto de vista de la cámara en cada instante de tiempo (*frame*).

```
function render() {
  requestAnimationFrame(render);
  renderer.render(scene, camera);
}
```

8. Crear un controlador del tamaño de la ventana.

```
function handleResize() {
```

```
  camera.aspect = window.innerWidth / window.innerHeight;
  camera.updateProjectionMatrix();
  renderer.setSize(window.innerWidth, window.innerHeight);
}
window.addEventListener('resize', handleResize, false);
```

9. Lanzar la inicialización de todos los objetos cuando la ventana esté cargada.

```
window.onload = init;
```

Una vez realizados estos pasos, veremos una escena con una esfera como puede observarse en la figura 2.1

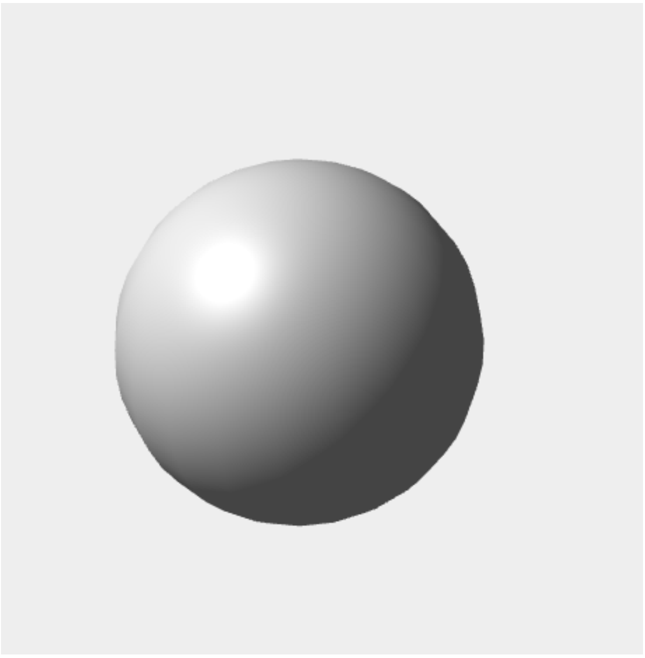

Figura 2.1: Ejemplo Three.js.

2.2.0.2. **Tom's planner**

Es una herramienta *web* para planificar proyectos. Gracias a esta herramienta se ha podido plasmar la planificación del proyecto y llevar a cabo un seguimiento de forma colaborativa (directoras y alumno).

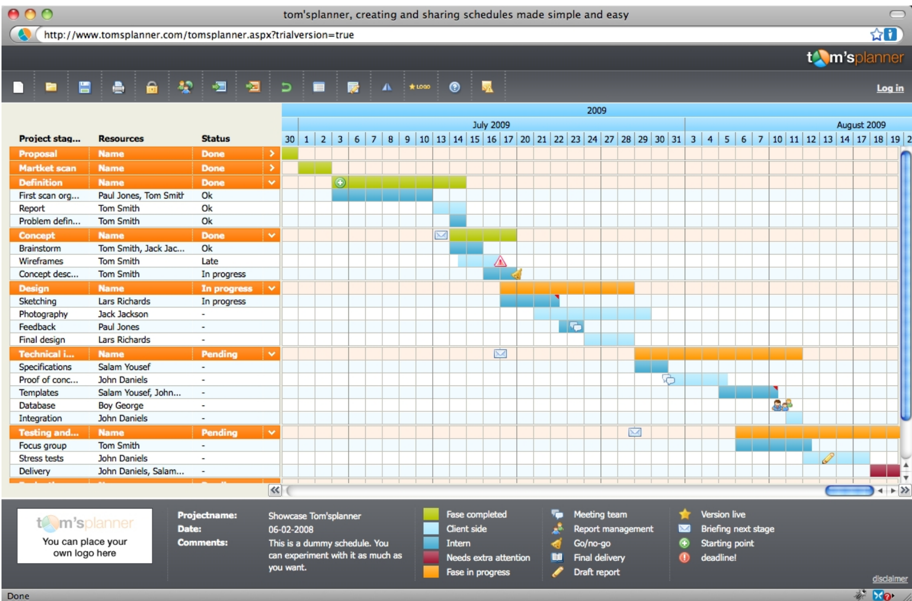

Figura 2.2: Tom's Planner.

2.3. Aprendizaje

Las primeras semanas se dedicaron a la formación y manejo de Three.js ya que era una librería desconocida para mí. Para ello, se realizaron distintos tutoriales on-line y ejemplos aportados por las directoras de este proyecto con lo que se adquirió destreza suficiente para poner manejar esta herramienta.

2.4. Planificación

Se ha realizado una planificación dinámica, esto es, hemos ido modificando una planificación base inicial a medida que surgían imprevistos. Toda la planificación está recogida en el siguiente documento: `http://www.tomsplanner.es/public/pfgibonmerino`.

2.5. Análisis de riesgos

Existen dos factores de riesgo que pueden surgir en el proyecto:

1. Vacaciones de Semana Santa: En estas dos semanas no se han planificado horas de trabajo, pero si fuera necesario se podrían utilizar para el proyecto.

2. Exámenes en la segunda evaluación:

   No existe mayor problema en esta semana si se planifica y organiza bien.

2.6. Análisis de factibilidad

Este proyecto es factible debido a los pocos riesgos que existen y a la poca repercusión que estos pueden tener. Además, se cuenta con medios y horas de sobra para poder realizarlo aunque surjan imprevistos.

3. CAPÍTULO

Desarrollo del proyecto

Nuestra propuesta incluye dos modelos, uno dinámico con el cual se calcula la dinámica de la tela, o sea, su movimiento; y el modelo geométrico que es el modelo que se visualiza.

3.1. Modelo geométrico

Este modelo está formado por vértices y caras. La geometría de la tela se puede observar en la figura 3.1. La tela está formada por una superficie 3D plana que se compone de los vértices y caras anteriormente mencionados.

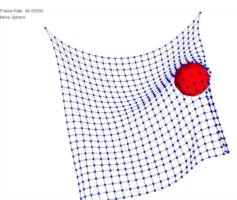

Figura 3.1: Geometría de la tela.

Antes de avanzar más en la explicación de este modelo voy a explicar muy brevemente cómo se renderizan los objetos. La geometría de un objeto está formada por vértices, aristas y caras (en nuestro caso triángulos). Los vértices y las aristas forman esas caras. De esta manera el motor gráfico dibujará cada una de las caras de un objeto hasta formar el objeto en su totalidad.

Continuando con nuestro modelo, las caras se forman siguiendo la geometría de la figura 3.2, es decir, dado el vértice en la posición (i, j), éste formará triángulos con los vértices $(i-1, j-1)$, $(i-1, j)$, $(i, j-1)$, $(i, j+1)$, $(i+1, j)$ y $(i+1, j+1)$, siendo i la posición de las columnas y j la posición de las filas. En la tabla 3.1 se puede ver qué vértices forman cada cara.

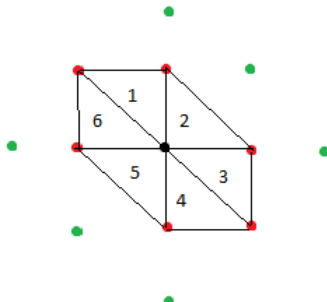

Figura 3.2: Caras.

triángulo 1	$(i-1, j-1)$	$(i, j-1)$	(i, j)
triángulo 2	$(i, j-1)$	$(i+1, j)$	(i, j)
triángulo 3	$(i+1, j)$	$(i+1, j+1)$	(i, j)
triángulo 4	$(i+1, j+1)$	$(i, j+1)$	(i, j)
triángulo 5	$(i, j+1)$	$(i-1, j)$	(i, j)
triángulo 6	$(i-1, j)$	$(i-1, j-1)$	(i, j)

Tabla 3.1: Caras por vértice

En la figura 3.3 podemos ver cómo están formadas todas las caras gracias a la visualización de la geometría en modo *wireframe* o alambre.

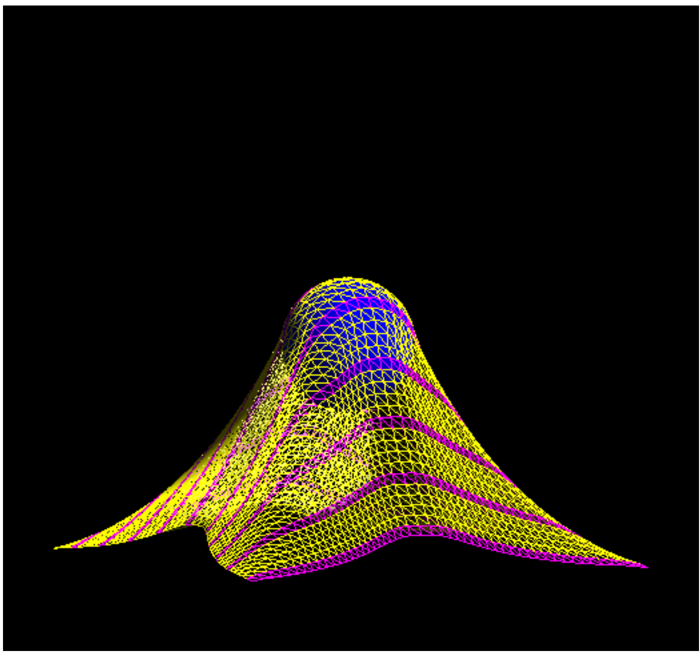

Figura 3.3: Triángulos de la tela.

Para que no se vean los triángulos y parezca de verdad una tela, aplicamos un material.

3.2. Modelo dinámico

Para diseñar el modelo dinámico nos hemos basado en un sistema de partículas llamado *mass-spring* que consiste en un conjunto de partículas unidas entre sí mediante un enlace de tipo "muelle".

El sistema de partículas se representa como un array bidimensional $(fila, columna)$ en el que, en cada posición, se encuentra una partícula.

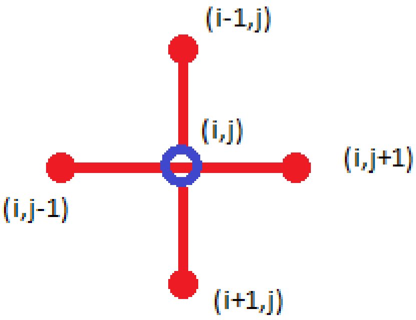

Figura 3.4: Sistema de partículas

Para una mejor comprensión del modelo dinámico primero definiremos un par de conceptos:

1. **Vecindario**:

   El vecindario de una partícula son aquellas partículas colindantes a ésta. Se les llama vecinos cercanos o inmediatos a aquellos vecinos con los que comparte geometría; es decir, las partículas vecinas con las que forma alguna cara, y vecinos lejanos a aquellos vecinos que no comparten una geometría pero sí propiedades físicas.

   En la figura 3.5 se pueden ver los vecinos de una partícula. Los vecinos rojos son los cercanos y los verdes los lejanos.

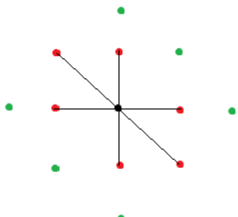

Figura 3.5: Vecindario.

2. **Vecindario de orden** n:

   Se define vecindad de orden n de una partícula p, $V_n(p)$, al conjunto de 4 vértices colocados en cierta posición con respecto a ella. En nuestro caso sólo existen los vecindarios de orden 1, 2 y 3 descritos como:

   a) Para $n = 1 : [(i-1, j), (i, j-1), (i, j+1), (i+1, j)]$

b) Para $n = 2 : [(i-1, j-1), (i-1, j+1), (i+1, j-1), (i+1, j+1)]$

c) Para $n = 3 : [(i-2, j), (i, j-2), (i, j+2), (i+2, j)]$

siendo (i, j) la posición de la partícula p. En la figura 3.6 las partículas rojas son el vecindario de orden 1, las azules de orden 2 y las verdes de orden 3.

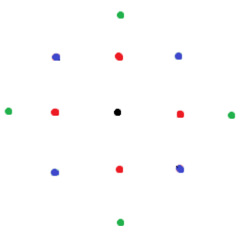

Figura 3.6: Vecindad de orden n.

En cuanto a la distribución de las partículas, éstas están repartidas uniforme y equidistantemente a lo largo del plano.

3.2.1. ¿Qué es una partícula?

Una partícula no es más que otra forma de representar a un vértice del modelo. La dinámica del sistema de partículas aporta, por tanto, un mecanismo de modificación de la posición de estos vértices del modelo geométrico.

Para que se entienda mejor, pensemos en que los vértices del modelo geométrico son las distintas partes de una marioneta, los hilos son las partículas y el marionetista representa la dinámica del sistema de partículas. Para que la marioneta se mueva de una determinada manera, el marionetista tendrá que mover cada uno de los hilos de una manera específica. Es decir, la dinámica del sistema de partículas calculará los movimientos de las partículas para que, de esta manera, la tela simule el efecto deseado.

3.2.2. Descripción de las características de una partícula

A continuación, describiremos las distintas características de cada partícula así como las fuerzas tanto internas como externas que influirán en su comportamiento.

- Masa: Es la masa de cada partícula. Para crear un efecto realista, se le asignan pesos ligeramente diferentes a cada partícula. La distribución de esos pesos puede ser muy

diversa y muchas de esas distribuciones pueden no generar el efecto deseado. Por ello, se ha realizado un pequeño experimento en el que se aplican distintos criterios en la elección de los pesos de cada partícula. Este experimento se encuentra descrito en el anexo A en la página 49.

- Posición Actual: Posición de la partícula en el instante actual.
- Posición Anterior: Posición de la partícula en el instante anterior.
- Posición Original: Posición donde se encontraba la partícula cuando se creó.
- Fuerza: Es el vector resultante del sumatorio de las fuerzas que se ejercen sobre la partícula.
- Aceleración: Es la aceleración que tiene la partícula en función de las fuerzas que se ejercen sobre ella y la masa de esta. Se calcula mediante la segunda ley de Newton:

$$\sum \vec{F} = m \cdot \vec{a} \Longrightarrow \vec{a} = \frac{\sum \vec{F}}{m},$$

  donde $\vec{a}$ es la aceleración, $\vec{F}$ es la fuerza y m es la masa.

- Velocidad: Velocidad actual de la partícula. Esta se obtiene a partir de la aceleración.

3.2.3. Ciclo de vida de una partícula

Las partículas tienen un comportamiento que se podría asemejar al de un ser vivo: nacen, viven y mueren.

3.2.3.1. Nacimiento

El nacimiento se considera la inicialización de la partícula; o sea, cómo se crea la partícula y cuáles son sus características iniciales. Una vez generada la geometría del objeto, se crean las partículas cuyo número dependerá de dicha geometría. Se podría hacer un estudio de cuál sería el número óptimo de partículas para cada caso concreto.

Las características iniciales son:

1. Posición actual = posición anterior = posición original: Posición de cada partícula en el instante t_0.
2. Fuerza: La fuerza que se ejerce sobre la partícula en el instante t_0 es nula.
3. Aceleración: La aceleración de la partícula en el instante t_0 es nula.
4. Velocidad: La velocidad de la partícula en el instante t_0 es nula.

3.2.3.2. Vida

La vida de la partícula es la interacción de esta con su entorno y consigo misma; esto es, cómo cambia su posición, su color u otras características del instante t al $t+1$.

En nuestro modelo, una partícula se ve afectada por fuerzas como la gravedad y el viento, por el estado del resto de partículas y por otros objetos como el suelo u obstáculos que se puede encontrar a lo largo de su vida.

1. Interacción con el **viento**:

   Si el viento está activo se ha de calcular la fuerza que se aplica sobre cada una de las caras que forman la geometría de la tela y, por tanto, sobre las partículas asociadas a dicha cara. Esto es, la fuerza que el viento ejerce sobre una cara viene dada por el producto escalar entre la normal de la cara y la fuerza del viento:

   $$\vec{F_c} = \frac{\vec{N_c}}{||N_c||} \cdot (\vec{N_c} \cdot \vec{F_v}),$$

   siendo $\vec{F_c}$ la fuerza que ejerce el viento en la cara, $\vec{N_c}$ la normal de la cara y $\vec{F_v}$ la fuerza del viento.

   Esta fuerza resultante es la que hay que aplicar a cada una de las partículas asociadas a dicha cara.

2. Interacción con la **gravedad**:

   Como cada partícula tiene una masa, sobre éstas se ejerce una fuerza potencial gravitatoria.

   $$\vec{F_g} = m \cdot \vec{g},$$

   siendo $\vec{F_g}$ la fuerza gravitatoria que se ejerce sobre la partícula, m la masa de la partícula y $\vec{g}$ la intensidad del campo gravitatorio terrestre o gravedad, la cual se

puede modificar para generar distintos efectos como simular la caída de un objeto en el espacio donde esta intensidad es menor que en la Tierra. Este es un caso particular de la segunda Ley de Newton, ya que la gravedad es una aceleración.

3. Interacción con **objetos externos**:

   Los objetos externos que interactúan con la tela son estáticos, inanimados, por lo que si una partícula colisiona con un objeto externo habrá que calcular la intersección entre ambos objetos.

   Para calcular si dos objetos colisionan entre sí, hay que saber qué dos tipos de objetos colisionan, ya que se utiliza una técnica u otra para ello. En nuestro caso, van a colisionar un punto (la partícula) con una esfera (la bola). Esta colisión entre objetos es bastante sencilla ya que sólo necesitamos la posición y el radio de la esfera y la posición del punto.

   El punto colisiona con la esfera si:

   $$|posicionEsfera - posicionPunto| < radioEsfera$$

   Por otro lado, también puede colisionar un punto (la partícula) con el suelo (un plano 3D). Para ello no hay más que comprobar el valor de la coordenada y de la posición de la partícula y la coordenada y del suelo.

   El punto colisiona con el plano si:

   $$posicionPunto.y < posicionSuelo.y$$

4. Interacción entre las **partículas**:

   a) Elasticidad: Para el cálculo de la elasticidad se utilizará la Ley de Hooke, la cual establece que el alargamiento unitario que experimenta un material elástico es directamente proporcional a la fuerza $\vec{F}$ aplicada sobre el mismo:

   $$\varepsilon = \frac{\delta}{L} = \frac{\vec{F}}{AE},$$

   siendo δ el alargamiento, L la longitud original, E el módulo de Young y A la sección transversal de la pieza estirada.

A partir de la ecuación anterior, despejando $\vec{F}$,

$$\vec{F} \cdot L = \delta AE,$$

$$\vec{F} = \frac{AE}{L}\delta,$$

se obtiene la forma más común de representar matemáticamente esta ley, la llamada ecuación del muelle. En esta ecuación se relaciona la fuerza $\vec{F}$ ejercida por el resorte en sentido opuesto a la fuerza externa aplicada al extremo del mismo que genera la elongación δ.

$$\vec{F} = -k\delta,$$

siendo k la constante elástica del resorte o lo que es lo mismo $\frac{-AE}{L}$ y δ su elongación.

b) Restricciones de distancia entre partículas: Entre cada par de partículas existe una distancia máxima que no pueden superar para generar un modelo realista, ya que si las partículas se alejan demasiado unas de otras esto puede dar origen a un muchos efectos raros en la tela debido a vibraciones.

Por otra parte, vamos a usar distintos métodos de integración para ver las diferencias que surgen al utilizar uno u otro método.

1. Euler explícito: Es el método de integración numérica más sencillo. Consiste en calcular el valor siguiente en función del valor anterior y un incremento, partiendo de que se conocen los valores iniciales.

$$\vec{x}(t+dt) = \vec{x}(t) + dt * \vec{x}'(t),$$

siendo $\vec{x}(t)$ la posición y $\vec{x}'(t)$ la velocidad de la partícula.

2. Verlet explícito: Es un método de integración numérica utilizado en las ecuaciones del movimiento de Newton. El método de Verlet se ha empleado tradicionalmente para el cálculo de trayectorias de partículas, ya que no requiere del cálculo de las velocidades si la fuerza no depende de la velocidad, lo cual lo hace el integrador idóneo para nuestro modelo.

Este método usa la aproximación de la diferencia central a la segunda derivada, la aceleración, como vemos a continuación:

$$\frac{\Delta^2 \vec{x}_n}{\Delta t^2} = \frac{\frac{\vec{x}_{n+1}-\vec{x}_n}{\Delta t} - \frac{\vec{x}_n-\vec{x}_{n-1}}{\Delta t}}{\Delta t} = \frac{\vec{x}_{n+1} - 2\vec{x}_n + \vec{x}_{n-1}}{\Delta t^2} = \vec{a}_n,$$

siendo $\vec{x}_n$ la posición de la partícula en el instante de tiempo n, $\vec{a}_n$ la aceleración de la partícula en el instante de tiempo n y Δt el incremento de tiempo entre el instante n y el $n+1$.

Despejando la nueva posición, obtenemos:

$$\vec{x}_{n+1} = 2\vec{x}_n - \vec{x}_{n-1} + \vec{a}_n \Delta t^2$$

3.2.3.3. Muerte

Existen diversas formas en las que una partícula puede morir: que su tiempo de vida alcance un valor límite, que su posición supere una posición límite o cualquier otra condición sobre alguna de sus características. En nuestro caso no hay un tiempo límite de vida de la partícula ni ninguna restricción en cuanto a la vida de las partículas. Nuestras partículas mueren cuando finaliza o se reinicia la animación. En ese caso, se eliminan todas las partículas y se generan otras nuevas para la siguiente animación.

3.2.4. Relaciones entre partículas

En nuestro caso vamos a utilizar lo que se llama un *Shearing spring* y un *Bending spring* que son un tipo de *mass-spring*[1]. Estos modelos no sólo unen los vecinos de orden 1 (restricciones estructurales, en color verde en la figura 3.7) sino también los vecinos de orden 2 en el caso del *Shearing spring* (en color rojo) y los de orden 3 en el caso del *Bending spring* (en color azul). Gracias al *Shearing spring* se logra tener el efecto que tiene la ropa al cortarla al bies[2] (mejor caída, mayor elasticidad y mayor fluidez de la tela).

[1] Ver link.

[2] Corte al bies: Este corte se realiza en diagonal a la dirección del hilo y a la trama de la tela.

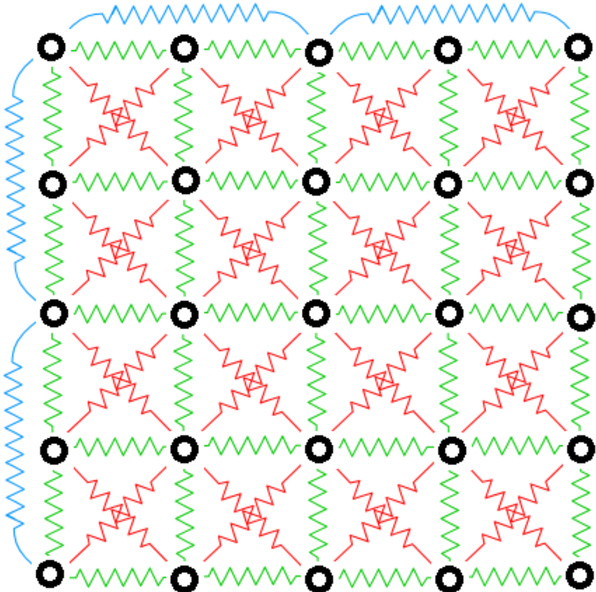

Figura 3.7: Relaciones entre partículas.

Estas relaciones se pueden tratar como si fueran unos muelles entre las partículas por lo que generarán una fuerza elástica contraria al movimiento cuando se muevan las partículas. Además, una de las características más importantes de estas relaciones es que existe una distancia máxima entre partículas; es decir, lo máximo que se pueden separar dos partículas relacionadas.

4. CAPÍTULO

Diseño

El proyecto tiene como objetivo la simulación del comportamiento de una tela.

Como se puede observar en la figura 4.1 el proyecto consta de dos partes principales: la inicialización y la animación.

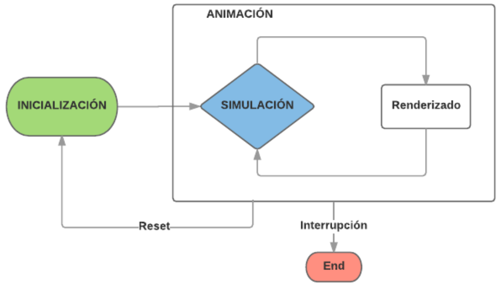

Figura 4.1: Arquitectura del PFG.

4.1. Inicialización

En la inicialización se prepara todo lo necesario para la simulación de la tela. Para ello hay que generar la escena, los objetos incluidos en ella y las herramientas de ayuda y control (Ver Figura 4.2).

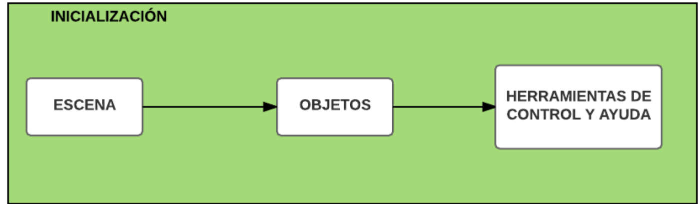

Figura 4.2: Estructura de la inicialización

4.1.1. Escena

La escena es como el lienzo donde se colocan todos los objetos que van a ser visualizados y el usuario ve dicha escena gracias a la cámara y al visualizador.

La escena está compuesta por:

- La cámara: Se utiliza una cámara en perspectiva.
- El visualizador (*renderer*): El visualizador es el que nos permite ver una imagen en cada instante de tiempo; es decir, es el motor subyacente que genera las imágenes consecutivas que forman la simulación.
- La iluminación: La iluminación es necesaria para poder visualizar los objetos. En nuestro caso, se utilizan 2 luces: una luz ambiental y una direccional.

Además, se añade el efecto niebla (*fog*) a la escena a fin de crear una sensación de difuminación o borrosidad de los objetos en función de su cercanía o lejanía a la cámara de la escena.

4.1.2. Objetos

Una vez generada la escena ya se pueden crear y colocar los objetos en ella. En nuestro modelo hemos incluido los siguientes objetos:

- La tela:

  Este es el objeto principal de la escena y el que tiene mayor complejidad. Como se especificó en el capítulo 3, este objeto está compuesto por dos modelos, uno

geométrico y otro dinámico. El geométrico es el que se visualiza y el dinámico es el que moldea el modelo geométrico (Figura 4.3).

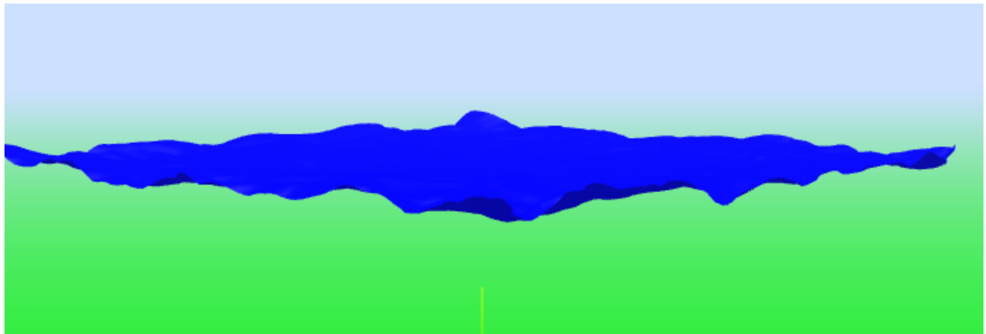

Figura 4.3: La tela

El modelo geométrico es un plano 3D definido por su ecuación paramétrica.

El modelo dinámico, en cambio, viene definido por un array de partículas y sus relaciones: estructurales[1], *Shear*[2]y *Bend*[3].

- La bola:

  Es un objeto 3D esférico que se utiliza para visualizar el efecto que tiene la tela al caer sobre otro objeto y cómo colisiona con él (Figura 4.4).

Figura 4.4: La bola

- El suelo:

  El suelo es un objeto 3D plano que sirve para limitar la caída de la tela y como base de nuestra escena.

[1] *Relaciones estructurales*: Relaciones de partículas con sus vecinos de orden 1.
[2] *Shear*: Relaciones de partículas con sus vecinos de orden 2.
[3] *Bend*: Relaciones de partículas con sus vecinos de orden 3.

4.1.3. Herramientas de control y ayuda

Estas herramientas sirven para darnos una mejor comprensión de la escena o para controlarla.

- Estadísticas del renderizado:

  Estas estadísticas nos sirven para poder comprobar la velocidad de la simulación; es decir, los *frames*[4] por segundo (FPS).

Figura 4.5: Estadísticas de la escena

- Controlador de órbita:

  El controlador de órbita nos permite movernos más fácilmente por la escena. La cámara no tiene un movimiento libre sino que orbita sobre el origen de coordenadas.

- Menú:

  Este menú nos permite modificar distintas variables de la escena así como algunas características de los objetos.

[4] *Frames*: Fotogramas

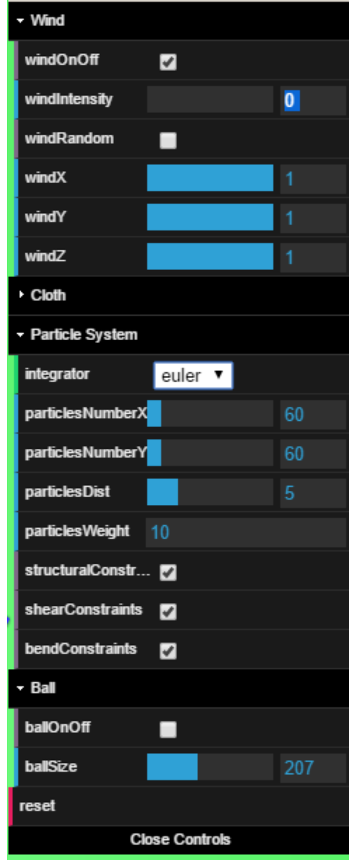

Figura 4.6: El menú

- Ejes:

  Son los ejes de coordenadas 3D que nos ayudan a orientarnos en la escena y saber cuál es cada dirección. En la figura 4.7 el eje rojo es el eje x, el eje azul es el z y el verde es el eje y.

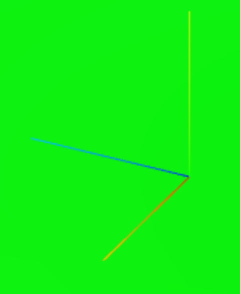

Figura 4.7: Los ejes

4.2. Animación

La animación de la tela es la parte principal del proyecto. Ésta no es más que un bucle continuo donde se calcula la posición de cada uno de los puntos de la tela (partículas), explicado en el apartado 4.2.1; y posteriormente se dibujan, explicado en el apartado 4.2.2.

4.2.1. Simulación

En la simulación toma parte toda la física explicada en el capítulo 3. En los siguientes apartados se explican uno por uno estos comportamientos. Están ordenados según el orden de evaluación.

4.2.1.1. Viento

El viento está compuesto por su dirección y su intensidad. La dirección se utiliza para comprobar cuánta de esa fuerza se aplica sobre cada cara de la tela.

4.2.1.2. Fuerza gravitatoria

La fuerza gravitatoria tiene en cuenta el peso del objeto y la intensidad del campo gravitatorio terrestre el cual es constante y tiene un valor de $9,8m/s^2$.

4.2.1.3. Fuerza elástica

Como las telas tienen un comportamiento elástico, en mayor o menor medida, se define una constante elástica que sirva para medir cuán de elástica es la tela.

4.2.1.4. Integración del movimiento

Se dispone de dos integradores numéricos: el de Euler y el de Verlet. El resultado visual para los dos es muy parecido pero es mucho más eficiente el integrador de Verlet.

4.2.1.5. Restricciones internas

La restricción más importante es la de satisfacer la distancia máxima entre partículas. Para ello se aplica una corrección si esta es mayor que la permitida.

4.2.1.6. Colisiones con la bola

Para el cálculo de las colisiones de una tela con una bola, se utilizan la posición de la bola, su radio y la posición de cada partícula.

4.2.1.7. Colisiones con el suelo

Para la colisión con el suelo, sólo se utilizan la posición en el eje y del suelo y la posición de cada partícula.

4.2.2. Renderizado

Para renderizar la escena, primero se actualizan las posiciones del modelo geométrico de la tela a partir de las posiciones calculadas en el modelo dinámico, se recalculan las caras y las normales y se toma una instantánea de la escena mediante la cámara y el visualizador.

4.3. Librerías propias

En esta sección se explicarán cada una de las librerías generadas específicamente para este proyecto (*Particle.js* y *Cloth.js*) así como el archivo principal programado en HTML (*Cloth.html*). Para que se entienda en todo momento de qué se está hablando la tabla 4.1 tiene una descripción de la nomenclatura utilizada.

Constantes	**EN MAYÚSCULAS Y NEGRITA**
Atributos y variables	**lowerCamelCase**[5]**y en negrita**
Funciones	**lowerCamelCase y en cursiva**

Tabla 4.1: Nomenclatura

[5]**lowerCamelCase**: Estilo de escritura que se aplica a frases o palabras compuestas en la que la primera palabra empieza con minúscula y las demás en mayúsculas como por ejemplo, miPrimeraFuncion().

4.3.1. Particle.js

En esta librería Javascript de diseño propio están implementadas las funciones y variables de cada partícula, tales como, añadir fuerza, añadir elasticidad y los distintos integradores.

4.3.1.1. Constantes

Las constantes que afectan directamente a las partículas están definidas en este archivo.

- **DAMPING** o amortiguador: Esta es una constante auxiliar que sirve para suavizar el movimiento de las partículas. Su valor en nuestro caso es 0.03.
- **DRAG** o arrastre: Esta es la constante que se usa para suavizar el movimiento. Su valor es $1 - \mathbf{DAMPING}$.
- **MASS** o masa: Es la masa base de cada partícula. Su valor es 0.1.
- **GRAVITY** o gravedad: Es el valor de la constante de gravitación de la escena. En nuestro caso, como en la tierra, 981 (está multiplicado por 100 para equiparar unidades).
- **TIMESTEP**: Tiempo entre dos cálculos sucesivos de la posición de las partículas En nuestro caso, es de 0.018s.
- **TIMESTEPSQ**: Es el cuadrado del tiempo entre dos cálculos sucesivos de la posición de las partículas, ya que teniendo este valor el cómputo es más eficiente.

4.3.1.2. Variables

Variables globales utilizadas por las funciones de las partículas:

- **restDistance**: Es la distancia entre partículas. El valor inicial es 5.
- **xSegs**: Número de columnas de partículas en el eje x. El valor inicial es 100.
- **ySegs**: Número de filas de partículas en el eje y. El valor inicial es 100.
- **weight**: Es la variabilidad de la masa, es decir, cuán de dispares van a ser las masas de las partículas. El valor inicial es 10.
- **elasticity**: Es la elasticidad de las uniones entre partículas. El valor inicial es 5.

4.3.1.3. Funciones

- *clothPlane (u, v)*: Dada la localización de una partícula mediante los índices *u* y *v* que representan la fila y la columna del array de partículas, esta función devuelve las coordenadas *x*, *y* y *z* de la posición asociada a dicha partícula en el plano 3D que representa la tela.

- *configureParticle(object3D, x, y, z, mass)*: Dado un objeto 3D, las coordenadas *x*, *y* y *z*, y la masa base de la partícula, esta función configura al objeto 3D de manera que tenga las propiedades de una partícula con los atributos inicializados de esta manera:

  1. **pos**: Posición en el plano 3D.
  2. **previous**: Posición en el plano 3D en el instante anterior.
  3. **original**: Posición en el plano 3D al inicio.
  4. **acceleration**: Vector 3D nulo.
  5. **velocity**: Vector 3D nulo.
  6. **mass**: Valor generado aleatoriamente a partir de la masa base.

  $$mass \cdot (numeroRandom^{6} \cdot weight + 1)$$

  7. **invMass**: $1/$**mass**

- *addForce(particle, force)*

  Dada una partícula y una fuerza, calcula la aceleración de la partícula al aplicarle dicha fuerza.

- *addElasticity(particle1, particle2)*

  Dadas dos partículas vecinas, ejerce una fuerza elástica sobre cada partícula. Cuanto más se alejen, mayor será la fuerza que ejerzan.

- *plane(width, height)*

  Esta función genera un plano 3D de anchura y altura igual a sus parámetros.

[6] numeroRandom: Número racional aleatorio entre 0 y 1

4.3.1.4. Integration methods

Los métodos de integración sirven para calcular la posición de cada partícula a lo largo del tiempo.

- *euler(particle, time)*

  Para implementar el método de integración de Euler utilizamos la aceleración para calcular la velocidad y dicha velocidad para calcular la posición de la partícula.

- *verlet(particle, timesq)*

  Para el método de integración de Verlet se utiliza directamente la aceleración y la constante TIMESTEPSQ[7] para minimizar la cantidad de operaciones. Además se tiene en cuenta la anterior posición para calcular la nueva y que de esta manera el movimiento sea más suave y, en definitiva, más realista.

4.3.2. Cloth.js

En esta librería se recopilan las variables de todos los elementos que aparecen en la escena así como las variables necesarias para visualizar la tela (exceptuando las variables auxiliares). Además también contiene las funciones de inicialización de los elementos y las funciones necesarias para simular el movimiento de la tela.

4.3.2.1. Variables

- Variables de la tela:
  - **clothGeometry**: Geometría de la tela.
  - **clothMaterial**: Material de la tela.
  - **clothMesh**: El objeto tela formado por su geometría y su material.
- Variables de la bola:
  - **ballGeometry**: Geometría de la bola.
  - **ballMaterial**: Material de la bola.
  - **ballMesh**: El objeto bola compuesto por su geometría y su material.

[7] TIMESTEPSQ: El cuadrado del tiempo entre cálculos sucesivos de la posición.

- **ballPositionOnSystemParticle**: Vector tridimensional con la posición de la bola.
- **ballSize**: Tamaño de la bola.

- Variables del suelo:
  - **groundGeometry**: Geometría del suelo.
  - **groundMaterial**: Material del suelo.
  - **groundMesh**: El objeto suelo dado por su geometría y su material.
- Sistema de partículas:
  - **particleSystem**: Array unidimensional de objetos 3D configurados como partículas.
- Variables de la escena:
  - **scene**: Variable donde se guardarán todos los elementos que están presentes en la escena. Es como el lienzo donde vamos a poner nuestra animación.
  - **camera**: Cámara con la cual vamos a visualizar la escena.
  - **renderer**: Visualizador que nos permite renderizar la escena.
- Variables de iluminación y materiales:
  - **ambientLight**: Luz ambiente de la escena.
  - **directionalLight**: Luz direccional de la escena.
- Variables de ayuda y control:
  - **stats**: Elemento de ayuda que sirve para ver el rendimiento de la animación (FPS-*Frames* por segundo).
  - **orbitControler**: Herramienta utilizada para poder moverse alrededor de la escena.
  - **container**: Contenedor HTML que sirve para guardar todos los elementos en el código HTML.
  - **axis**: Ejes de ayuda para obtener una mejor orientación de las direcciones.
- Variables del menú:

  - **controls**: Contenedor del menú.
  - **clothColor**: Variable que guarda la elección del color de la tela del menú.
  - **integrator**: Variable que guarda la elección del tipo de integrador del menú.
  - **structural**: Variable booleana que guarda si se utiliza una relación *structural* entre las partículas.
  - **shear**: Variable booleana que guarda si se utiliza una relación *Shear* entre las partículas.
  - **bend**: Variable booleana que guarda si se utiliza una relación *Bend* entre las partículas.

- Variables del viento:

  - **wind**: Variable booleana que indica si el viento está activo o no.
  - **windStrength**: Variable que guarda el valor de la fuerza del viento.
  - **windForce**: Variable que guarda el vector tridimensional con la dirección del viento.

4.3.2.2. Funciones

- *initScene()*

  Crea una nueva escena y activa el efecto niebla (*fog*).

- *initCamera()*

  Esta función crea una cámara y la coloca en una posición donde se pueda ver bien la escena.

- *createAxis()*

  Crea los ejes de la escena.

- *ilumination()*

  Crea la iluminación con dos luces:

  1. Una luz ambiental que ilumina toda la escena con poca intensidad.
  2. Una luz direccional que ilumina desde la parte superior de la escena.

- *createCloth()*

  Esta función crea la tela a partir de su geometría y su material y activa el atributo **castShadow** que indica que este objeto va a generar sombras en la escena.

- *createClothMaterial()*

  Crea el material de la tela.

- *createClothGeometry()*

  Crea la geometría de la tela mediante la ecuación paramétrica del plano descrito en la variable general *clothPlane*.

- *createGround()*

  Crea el suelo a partir de su geometría y su material y activa el atributo **receiveShadow** que indica que se van a generar sombras sobre este objeto.

- *createGroundMaterial()*

  Crea el material del suelo.

- *createGroundGeometry()*

  Crea el suelo como un objeto geométrico de tipo plano.

- *createBall()*

  Crea la bola a partir de su geometría y su material y activa el atributo **receiveShadow** que indica que se van a generar sombras sobre esta bola.

- *createBallMaterial()*

  Crea el material de la bola.

- *createBallGeometry()*

  Crea la geometría de una bola (una esfera).

- *initRender()*

  Esta función crea el visualizador (*renderer*) y ajusta sus parámetros iniciales.

- *initControls()*

  Inicializa los controles de navegación en órbita, es decir, aquellos controles que nos permiten movernos por la escena.

- *initStats()*

  Inicializa las estadísticas que nos permiten observar el rendimiento.

- *initWind()*

  Inicializa el objeto viento y le asigna una fuerza y una dirección.

- *satisfyConstraints(p1, p2, distance)*

  Dadas dos partículas, esta función corrige la distancia entre ellas de forma que siempre se mantengan a una distancia menor que la distancia máxima.

4.3.3. Cloth.html

Este es el programa principal el cual incluye funciones recogidas en los otros dos archivos ya comentados (librerías propias) así como otras librerías muy utilizadas tales como:

1. THREE.js
2. Detector.js
3. OrbitControls.js
4. stats.min.js
5. dat.gui.js

4.3.3.1. Funciones

- *init()*

  Inicializa la simulación.

- *initCloth()*

  Crea el modelo dinámico de la tela.

- *onWindowResize()*

  Controlador del tamaño de la ventana. Ajusta la resolución de la escena a la de la ventana.

- *initMenu()*

  Inicializa el menú que nos permite cambiar la configuración de la simulación.

- *animate()*

  Función que se encarga del bucle continuo en el que se llaman a las funciones *simulate* y *render* continuamente.

- *render()*

  Función que se encarga de renderizar la escena.

- *simulate(time)*

  Dado un instante de tiempo, realiza la simulación para pasar al siguiente instante de tiempo.

- *Cloth(w,h)*

  Función que dadas la altura y la anchura devuelve el el modelo dinámico de la tela.

- *reset()*

  Resetea la escena para colocar la tela en su posición inicial.

5. CAPÍTULO

Análisis del proyecto

En este capítulo se expondrá el análisis del proyecto, es decir, las complicaciones que han surgido, los cambios que se han hecho respecto a la planificación original y los resultados obtenidos.

5.1. Complicaciones

Durante este proyecto no han surgido muchas complicaciones ya que se estructuró bien desde un primer momento y eso facilitó la realización de este. Aún así surgieron un par de problemas que comentaré a continuación:

- Problemas con la implementación de los sistemas de referencia en Three.js:

  A la hora de calcular la posición de cada una de las partículas en el sistema de referencia del mundo hay que tener en cuenta las transformaciones que tiene cada una de las partículas y las que tiene toda la tela en general, es decir, el padre de las partículas en su jerarquía. Three.js ofrece unos métodos que calculan la posición de los hijos en el sistema de referencia del mundo pero que al no ajustarse a la jerarquía de mi modelo no se han utilizado y se han usado funciones propias.

5.2. Gestión del proyecto

El proyecto ha seguido su curso con normalidad sin ningún tipo de riesgo y siguiendo la planificación establecida al inicio del mismo.

Las horas totales empleadas en el proyecto están plasmadas en la tabla 5.1.

Diseño	Implementación	Documentación	Aprendizaje	Totales
25	77	192	16	312

Tabla 5.1: Imputación de horas en el proyecto

5.3. Resultados

Los resultados obtenidos en las simulaciones han sido bastante satisfactorios. El rendimiento de la simulación ha sido óptimo (60 FPS en el ordenador en el que se han realizado las pruebas) mientras el número de partículas no superase las 4300. No es un gran número de partículas, pero parece suficiente para la cantidad de condicionantes que tienen (aire, colisiones con objetos, gravedad...).

Como valorar una simulación es bastante subjetivo, no existe una medida cuantitativa de cuán de bien está la simulación. Bajo mi criterio, realiza una simulación bastante realista.

Las siguiente figuras son fotografías tomadas durante una simulación para comprobar las distintas propiedades de la tela.

Viento: En las figuras 5.1 y 5.2 se puede observar la simulación de la tela cuando está suspendida en el aire y es movida por el viento. En este tipo de simulaciones, se forman arrugas en la tela lo que proporciona un efecto más realista.

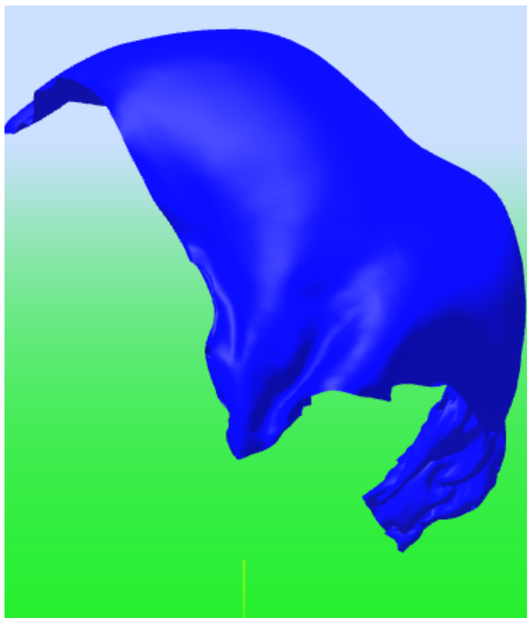

Figura 5.1: Un instante en la simulación de la tela en el aire

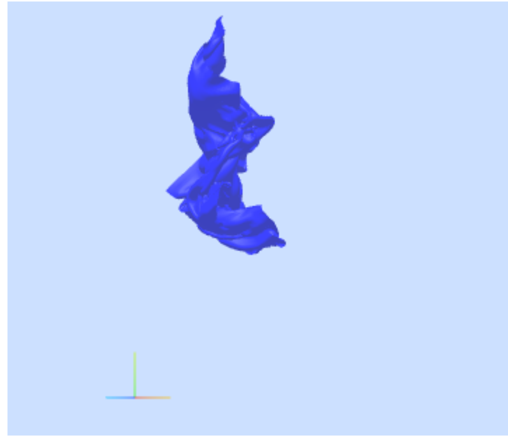

Figura 5.2: Otro instante en la simulación de la tela en el aire

Colisiones: En las figuras 5.3 y 5.4 se pueden ver las simulaciones de las colisiones con una bola y con el suelo respectivamente.

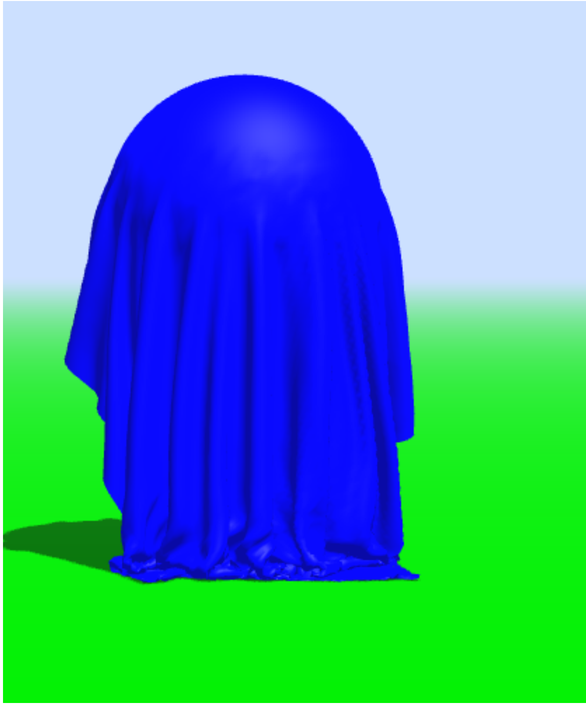

Figura 5.3: Un instante en la simulación de la tela en colisión con la bola

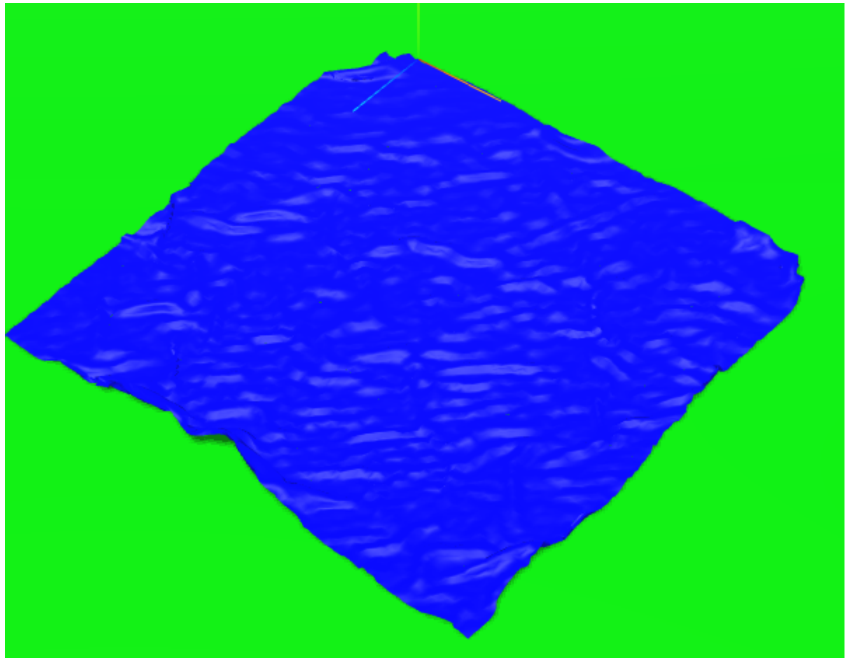

Figura 5.4: Un instante en la simulación de la tela en colisión con el suelo

En la figura 5.5 se ha realizado un zoom para apreciar mejor las arrugas que se generan al

colisionar con el suelo.

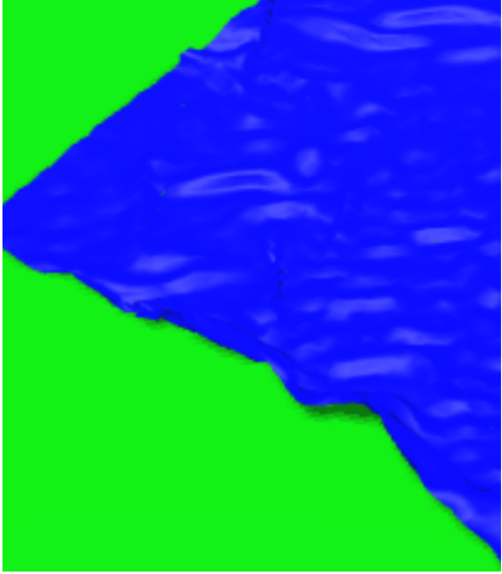

Figura 5.5: Zoom al instante en la simulación de la tela en colisión con el suelo

6. CAPÍTULO

Conclusiones

Como capítulo final se presentan las conclusiones generales del proyecto y las líneas futuras.

6.1. Conclusiones generales del proyecto

Los objetivos principales de este proyecto han sido cumplidos. Se ha logrado implementar un modelo matemático robusto y eficaz que permite simular el comportamiento de una tela.

Se ha conseguido aprender el manejo de la librería Three.js de manera que sirva como preámbulo a futuros proyectos en este campo utilizando esta potente librería.

Gracias a las asignaturas de la rama de Computación se ha logrado comprender el funcionamiento interno de los motores gráficos y utilizar modelos matemáticos con el fin de simular objetos reales. Esto ha permitido una mayor fluidez a la hora de diseñar el modelo matemático del proyecto y la implementación de este.

En general, ha sido un proyecto muy enriquecedor cuyo producto final puede servir como base a futuros proyectos aun mayores.

6.2. Líneas futuras

Tras el estudio realizado al inicio del proyecto se pensó en realizar dos tipos de modelos dinámicos para comprobar su eficiencia por separado y un tercer modelo que juntase ambos para ganar eficiencia. Estos modelos eran un sistema de partículas y una superficie NURBS. Dada la gran cantidad de trabajo que supuso realizar un modelo eficiente y que simulase de manera realista el comportamiento de una tela mediante el sistema de partículas, se decidió no realizar los otros dos modelos dinámicos. Por ello, esas líneas quedan abiertas para el futuro.

La idea de utilizar superficies NURBS para modelar el comportamiento de una tela puede ser interesante debido a la poca cantidad de puntos de control (comparada con el número de partículas del sistema de partículas) necesarios para controlar el comportamiento de esta superficie. El único inconveniente que encontramos fue el poder simular colisiones ya que, en principio, sería bastante más complicado que con el sistema de partículas.

A raíz de la problemática de las colisiones se nos ocurrió juntar ambos modelos, es decir, mientras la tela no colisionase con ningún objeto sería simulada mediante una superficie NURBS, pero en el momento en el que colisionara pasaría a simularse mediante un sistema de partículas. Esto aumentaría el rendimiento de la simulación.

Otra forma de mejorar este proyecto sería aumentar el rendimiento de la simulación mediante procesos concurrentes o en paralelo. Esta mejora sería bastante laboriosa pero podría aumentar mucho el número de partículas con las que se podría hacer una simulación en tiempo real con un rendimiento óptimo.

Anexos

A. ANEXO

Experimento de las masas de las partículas

Este experimento busca encontrar una distribución de pesos de las partículas para lograr un efecto realista de la caída de la tela. Para ello se han realizado simulaciones con diferentes tipos de técnicas y estos han sido los resultados obtenidos:

A.1. Peso fijo

Utilizando el mismo peso para todas las partículas, sólo se logra tener un efecto realista si activamos el viento de forma que surjan algunas arrugas ya que si no, la tela cae plana.

A.2. Pesos siguiendo una sinusoidal

Distribuyendo los pesos según una sinusoidal, el efecto no es nada realista pues se nota una periodicidad en la ondulación de la tela que no se presenta en la realidad.

A.3. Pesos aleatorios

Utilizando pesos aleatorios comprendidos entre 0.1 y 1, se logra simular de forma bastante realista la caída de una tela ya que se crean arrugas en la tela por la pequeña variación en los pesos.

A.4. Conclusión

De entre todos los experimentos realizados hemos observado que los mejores resultados se han logrado con una inicialización aleatoria de los pesos de las partículas, por lo que es el método que se va a implementar en nuestra simulación.

B. ANEXO

Implementación

B.1. Particle.js

```
//Particle.js
//this paticle system has been created for cloth simulation

//CONSTANTS
var DAMPING = 0.03;
var DRAG = 1 - DAMPING;
var MASS = 0.1;

var GRAVITY = 981;

var TIMESTEP = 18 / 1000;
var TIMESTEP_SQ = TIMESTEP * TIMESTEP;

//Variables
var restDistance = 5;

var xSegs = 100;
var ySegs = 100;

```

```
20 var weight = 10;
21
22 var gravityV = new THREE.Vector3( 0,  GRAVITY, 0 ).multiplyScalar(
     MASS );
23
24 var elasticity = 5;
25
26 //Cloth structure
27 var clothPlane = plane( restDistance * xSegs, restDistance * ySegs );
28
29 //Particle functions
30
31 //Configures a THREE.Object3D with a particle characteristics
32 function configureParticle( object3D, x, y, z, mass ){
33   object3D.pos = clothPlane(x, z);
34   object3D.previous = clothPlane(x, z);
35   object3D.original = clothPlane(x, z);
36   object3D.acceleration = new THREE.Vector3( 0, 0, 0 );
37   object3D.velocity = new THREE.Vector3( 0, 0, 0 );
38   object3D.mass = mass * Math.floor((Math.random() * weight) + 1);
39   object3D.invMass = (1 / object3D.mass);
40 }
41
42 //Adds a force to that particle to increase it's acceleration
43 function addForce ( particle, force ) {
44   var tmpF = new THREE.Vector3(force.x,force.y,force.z);
45   particle.acceleration.sub( tmpF.multiplyScalar( particle.invMass ) )
     ;
46
47 }
48
49 //Adds elasticity to the constraint between particles
50 function addElasticity ( particle1, particle2 ){
51   var distNow = new THREE.Vector3(), distPrev = new THREE.Vector3();
52   distNow.subVectors( particle1.pos, particle2.pos );
53   distPrev.subVectors( particle1.previous, particle2.previous );
```

```
54   var dif = distNow.length() - distPrev.length();
55   if (dif>0){
56     addForce(particle1, - distNow.normalize().multiplyScalar(dif/2)*
    elasticity);
57     addForce(particle2,  distNow.normalize().multiplyScalar(dif/2)*
    elasticity);
58   }
59 }
60
61
62 //Cloth functions
63 function plane( width, height ) {
64
65   return function( u, v ) {
66
67     var x = ( u - 0.5 ) * width;
68     var y = 0;
69     var z = ( v + 0.5 ) * height;
70
71     return new THREE.Vector3( x, y, z );
72
73   };
74
75 }
76
77 //Integration methods
78 //Euler
79 function euler( particle, time ) {
80   particle.previous.copy(particle.pos);
81   particle.velocity.add(particle.acceleration.multiplyScalar( timesq )
    );
82   particle.pos.add(particle.velocity.multiplyScalar( timesq ));
83   particle.acceleration.set( 0, 0, 0 );
84
85 };
86
```

```
87  //Verlet
88  function verlet( particle, timesq ){
89    var tmp = new THREE.Vector3();
90    var newPos = tmp.subVectors( particle.pos, particle.previous );
91    newPos.multiplyScalar( DRAG ).add( particle.pos );
92    newPos.add( particle.acceleration.multiplyScalar( timesq ) );
93  
94    particle.previous = particle.pos;
95    particle.pos = newPos;
96    particle.acceleration.set( 0, 0, 0 );
97  }
```

B.2. Cloth.js

```
1   //Cloth variables
2   var clothGeometry, clothMaterial, clothMesh;
3   
4   //Ball
5   var ballGeometry, ballMaterial, ballMesh, ballOnOff=false;
6   var ballPositionOnSystemParticle, ballSize;
7   
8   //Ground
9   var groundGeometry, groundMaterial, groundMesh;
10  
11  //Particle system
12  var particleSystem;
13  
14  //Scene
15  var scene;
16  
17  //Camera
18  var camera;
19  
20  //Renderer
21  var renderer;
```

```
22
23 //Ilumination and material
24 var ambientLight, directionalLight;
25
26 //Stats
27 var stats;
28
29 //Orbit Controler
30 var orbitControler;
31
32 //HTML container
33 var container;
34
35 //Axis
36 var axis;
37
38 //Controler
39 var controls;
40
41 //Controls parameters
42 var clothColor = 0x0000FF,integrator = 'verlet', structural=true,
       shear=true, bend=true;
43
44 //Wind
45 var wind, windStrength , windForce;
46
47
48
49 //Functions
50
51 function initScene(){
52   scene = new THREE.Scene();
53   scene.fog = new THREE.Fog(0xcce0ff, 500, 10000);
54 }
55
56 function initCamera(){
```

```
57    camera = new THREE.PerspectiveCamera( 30, window.innerWidth /
       window.innerHeight, 1, 10000 );
58    camera.position.x = 1000;
59    camera.position.y = 50;
60    camera.position.z = 1500;
61    scene.add(camera);
62  }
63
64  function createAxis(){
65    axis = new THREE.AxisHelper( 75 );
66    scene.add(axis);
67
68  }
69
70  function ilumination(){
71    ambientLight = new THREE.AmbientLight( 0x666666 );
72    scene.add(ambientLight);
73
74    directionalLight = new THREE.DirectionalLight( 0xdfebff, 1.75 );
75    directionalLight.position.set( 50, 200, 100 );
76    directionalLight.position.multiplyScalar( 1.3 );
77
78    directionalLight.castShadow = true;
79
80    directionalLight.shadow.mapSize.width = 1024;
81    directionalLight.shadow.mapSize.height = 1024;
82
83    var d = 1000;
84
85    directionalLight.shadow.camera.left = - d;
86    directionalLight.shadow.camera.right = d;
87    directionalLight.shadow.camera.top = d;
88    directionalLight.shadow.camera.bottom = - d;
89
90    directionalLight.shadow.camera.far = 5000;
91
```

```
92    scene.add(directionalLight);
93  }
94
95  function createCloth(){
96    createClothMaterial();
97    createClothGeometry();
98    clothMesh = new THREE.Mesh( clothGeometry, clothMaterial );
99    clothMesh.castShadow = true;
100
101 }
102
103 function createClothMaterial(){
104   clothMaterial = new THREE.MeshPhongMaterial( {
105     specular: 0x030303,
106     color: clothColor,
107     side: THREE.DoubleSide,
108     alphaTest: 0.5
109   } );
110 }
111
112 function createClothGeometry(){
113   clothGeometry = new THREE.ParametricGeometry( clothPlane,xSegs,
        ySegs );
114   clothGeometry.dynamic = true;
115 }
116
117 function createGround(){
118   createGroundMaterial();
119   createGroundGeometry();
120   groundMesh = new THREE.Mesh( groundGeometry, groundMaterial );
121   groundMesh.position.y = - 250;
122   groundMesh.rotation.x = - Math.PI / 2;
123   groundMesh.receiveShadow = true;
124   scene.add( groundMesh );
125 }
126
```

```
127 function createGroundMaterial(){
128   groundMaterial = new THREE.MeshPhongMaterial( {
129     color:"green",
130     specular: 0x111111
131   } );
132 }
133
134 function createGroundGeometry(){
135   groundGeometry = new THREE.PlaneBufferGeometry( 20000, 20000 );
136 }
137
138 function createBall(){
139   ballSize = 100;
140   createBallMaterial();
141   createBallGeometry();
142   ballMesh = new THREE.Mesh( ballGeometry, ballMaterial );
143   ballMesh.castShadow = true;
144   ballMesh.position.set( 0, 0, 0 );
145   ballMesh.receiveShadow = true;
146   scene.add( ballMesh );
147   ballMesh.visible = ! true;
148 }
149
150 function createBallMaterial(){
151   ballMaterial = new THREE.MeshPhongMaterial( { color: 0xaaaaaa } );
152 }
153
154 function createBallGeometry(){
155   ballGeometry = new THREE.SphereGeometry( ballSize, 20, 20 );
156 }
157
158 function initRenderer(){
159   renderer = new THREE.WebGLRenderer( { antialias: true } );
160   renderer.setPixelRatio( window.devicePixelRatio );
161   renderer.setSize( window.innerWidth, window.innerHeight );
162   renderer.setClearColor( scene.fog.color );
```

```
163 
164   renderer.gammaInput = true;
165   renderer.gammaOutput = true;
166   renderer.shadowMap.enabled = true;
167 }
168 
169 function initControls(){
170   orbitControler = new THREE.OrbitControls( camera,
      renderer.domElement );
171   orbitControler.maxPolarAngle = Math.PI * 0.5;
172   orbitControler.minDistance = 1000;
173   orbitControler.maxDistance = 7500;
174 }
175 
176 function initStats(){
177   stats = new Stats();
178 }
179 
180 function initWind(){
181   wind = true;
182   windStrength = 100;
183   windForce = new THREE.Vector3( 0, 0, 0 );
184 }
185 
186 function satisifyConstraints( p1, p2, distance ) {
187   var diff = new THREE.Vector3();
188   diff.subVectors( p2.pos, p1.pos );
189   var currentDist = diff.length();
190   if ( currentDist === 0 ) return; // prevents division by 0
191   var correction = diff.multiplyScalar( 1 - distance / currentDist );
192   var correctionHalf = correction.multiplyScalar( 0.5 );
193   p1.pos.add( correctionHalf );
194   p2.pos.sub( correctionHalf );
195 
196 }
```

B.3. Cloth.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>three.js webgl - cloth simulation</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
    user-scalable=no, minimum-scale=1.0, maximum-scale=1.0">
    <style>
      body {
        font-family: Monospace;
        background-color: #000;
        color: #000;
        margin: 0px;
        overflow: hidden;
      }
    </style>
  </head>

  <body>
    <script src="../libs/three.js"></script>
    <script src="../libs/Detector.js"></script>
    <script src="../libs/OrbitControls.js"></script>
    <script src="../libs/stats.min.js"></script>
    <script src="../libs/dat.gui.js"></script>

    <script src="../libs/Particle.js"></script>
    <script src="../libs/Cloth.js"></script>

    <script >
      init();
      animate();

      //Initializes every object in the scene
```

```
33      function init(){
34        container = document.createElement( 'div' );
35        document.body.appendChild( container );
36
37        initScene();
38        initCamera();
39        createAxis();
40        ilumination();
41
42        createCloth();
43        initCloth();
44
45        createGround();
46
47        ballPositionOnSystemParticle = new THREE.Vector3( 0, -200,
    restDistance*ySegs );
48        createBall();
49
50        initRenderer();
51
52        container.appendChild( renderer.domElement );
53
54        initControls();
55
56        initStats();
57
58        container.appendChild( stats.dom );
59
60        initWind();
61
62        window.addEventListener( 'resize', onWindowResize, false );
63
64        initMenu()
65      }
66
67      //Initializes the cloth dynamic model
```

```
68        function initCloth(){
69          particleSystem = new Cloth(xSegs, ySegs);
70          clothMesh.position.set(0, 200, -restDistance*ySegs);
71          scene.add( clothMesh );
72        }
73
74        //Windows resize handler
75        function onWindowResize() {
76
77          camera.aspect = window.innerWidth / window.innerHeight;
78          camera.updateProjectionMatrix();
79
80          renderer.setSize( window.innerWidth, window.innerHeight );
81
82        }
83
84        //Initializes the control menu
85        function initMenu(){
86          controls = new function(){
87            this.windOnOff = true;
88            this.windIntensity = 100;
89            this.windRandom = false;
90            this.windX = 0;
91            this.windY = 0;
92            this.windZ = 0;
93            this.integrator = 'verlet';
94            this.particlesNumberX = 100;
95            this.particlesNumberY = 100;
96            this.particlesDist = 5;
97            this.particlesWeight = 10;
98            this.clothColor = 0x0000FF;
99            this.clothElasticity = 5;
100           this.structuralConstraints = true;
101           this.shearConstraints = true;
102           this.bendConstraints = true;
103           this.ballOnOff = false;
```

```
104         this.ballSize = 100;
105         this.reset = reset;
106       }
107 
108       var gui = new dat.GUI();
109 
110       //Wind controls
111       var windGui = gui.addFolder('Wind');
112       windGui.add(controls,'windOnOff').onFinishChange(function(){
113         wind = controls.windOnOff;
114       });
115       windGui.add(controls,'windIntensity',0, 500);
116       windGui.add(controls,'windRandom');
117       windGui.add(controls,'windX',-1,1).step(1);
118       windGui.add(controls,'windY',-1,1).step(1);
119       windGui.add(controls,'windZ',-1,1).step(1);
120 
121       //Cloth controls
122       var clothGui = gui.addFolder('Cloth');
123       clothGui.addColor(controls,'clothColor').onFinishChange(
      function(){
124         clothColor=controls.clothColor;
125         reset();
126       });
127       clothGui.add(controls,'clothElasticity',0,10).onFinishChange(
      function(){
128         elasticity = controls.clothElasticity;
129       });
130 
131       //Particle System controls
132       var particleSystemGui = gui.addFolder('Particle System');
133       particleSystemGui.add(controls,'integrator', ['verlet', 'euler
      ']).onFinishChange(function(){
134         integrator = controls.integrator;
135         reset();
136       });
```

```
137     particleSystemGui.add(controls,'particlesNumberX', 0, 500).
    step(5).onFinishChange(function(){
138         xSegs = controls.particlesNumberX;
139         reset();
140     });
141     particleSystemGui.add(controls,'particlesNumberY', 0, 500).
    step(5).onFinishChange(function(){
142         ySegs = controls.particlesNumberY;
143         reset();
144     });
145     particleSystemGui.add(controls,'particlesDist', 0.1, 20).
    onFinishChange(function(){
146         restDistance = controls.particlesDist;
147         reset();
148     });
149     particleSystemGui.add(controls,'particlesWeight').min(0).
    onFinishChange(function(){
150         weight = controls.particlesWeight;
151         reset();
152     });
153     particleSystemGui.add(controls,'structuralConstraints').
    onFinishChange(function(){
154         structural = controls.structuralConstraints;
155         reset();
156     });
157     particleSystemGui.add(controls,'shearConstraints').
    onFinishChange(function(){
158         shear = controls.shearConstraints;
159         reset();
160     });
161     particleSystemGui.add(controls,'bendConstraints').
    onFinishChange(function(){
162         bend = controls.bendConstraints;
163         reset();
164     });
165
```

```
166         //Ball controls
167         var ballGui = gui.addFolder('Ball');
168         ballGui.add(controls,'ballOnOff').onFinishChange(function(){
169           ballOnOff = controls.ballOnOff;
170           ballMesh.visible = ballOnOff;
171         });
172         ballGui.add(controls,'ballSize', 10, 500).onChange(function(){
173           var scale = controls.ballSize / ballSize;
174           ballSize = controls.ballSize;
175           ballMesh.geometry.scale(scale,scale,scale);
176         });
177 
178         //Reset button
179         gui.add(controls, 'reset');
180 
181         gui.remember(controls);
182 
183       }
184 
185       //Controls the animation
186       function animate() {
187 
188         requestAnimationFrame( animate );
189 
190           var time = Date.now();
191 
192         if(controls.windRandom){
193           windStrength = Math.cos( time / 7000 ) * 20 + 200;
194           windForce.set( Math.sin( time / 2000 ), Math.sin( time /
    3000 ), Math.sin( time / 1000 ) ).normalize().multiplyScalar(
    windStrength );
195         }else{
196           windStrength = controls.windIntensity;
197           windForce.set( -controls.windX, -controls.windY,
    -controls.windZ ).normalize().multiplyScalar( windStrength );
198         }
```

```
199
200         simulate( time );
201         render();
202         stats.update();
203
204
205
206       }
207
208       //Controls the rendering
209       function render() {
210
211         for ( var i = 0, il = particleSystem.particles.length; i < il;
     i ++ ) {
212
213           clothMesh.geometry.vertices[ i ].copy(
    particleSystem.particles[ i ].pos );
214
215         }
216
217
218         clothMesh.geometry.computeFaceNormals();
219         clothMesh.geometry.computeVertexNormals();
220
221         clothMesh.geometry.normalsNeedUpdate = true;
222         clothMesh.geometry.verticesNeedUpdate = true;
223
224         camera.lookAt( scene.position );
225
226         renderer.render( scene, camera );
227
228       }
229
230       //Controls the cloth simulation
231       function simulate ( time ){
232
```

```
233        var i, il, particles, particle, pt, constraints, constraint;
234        var diff = new THREE.Vector3();
235        var tmpForce = new THREE.Vector3();
236        particles = particleSystem.particles;
237
238          if ( wind ) {
239
240            var face, faces = clothMesh.geometry.faces, normal;
241
242            for ( i = 0, il = faces.length; i < il; i ++ ) {
243
244              face = faces[ i ];
245              normal = face.normal;
246
247              tmpForce.copy( normal ).normalize().multiplyScalar(
    normal.dot( windForce ) );
248              addForce( particles[ face.a ], tmpForce);
249              addForce( particles[ face.b ], tmpForce);
250              addForce( particles[ face.c ], tmpForce);
251            }
252
253          }
254
255          // Gravity force
256          for ( i = 0, il = particles.length; i < il; i ++ ) {
257
258            particle = particles[ i ];
259            addForce( particle, gravityV);
260
261          }
262
263          // Elasticity force
264          constraints = particleSystem.constraints;
265          il = constraints.length;
266
267          for ( i = 0; i < il; i ++ ) {
```

```
268
269           constraint = constraints[ i ];
270           addElasticity(constraint[0],constraint[1]);
271
272         }
273
274         //Particles movement integration
275         particles = particleSystem.particles;
276         for ( i = 0, il = particles.length; i < il; i ++ ) {
277           particle = particles[ i ];
278           if(integrator = 'verlet'){
279             verlet( particle, TIMESTEP_SQ );
280           }else if(integrator = 'euler'){
281             euler(particle,TIMESTEP);
282           }
283
284
285         }
286
287         // Start Constraints
288         constraints = particleSystem.constraints;
289         il = constraints.length;
290         for ( i = 0; i < il; i ++ ) {
291
292           constraint = constraints[ i ];
293           satisifyConstraints( constraint[ 0 ], constraint[ 1 ],
  constraint[ 2 ] );
294
295         }
296
297         // Ball Constraints
298         if ( ballOnOff ) {
299           for ( i = 0, il = particles.length; i < il; i ++ ) {
300
301             particle = particles[ i ];
302             pos = particle.pos;
```

```
303             diff.subVectors( pos, ballPositionOnSystemParticle );
304             if ( diff.length() < ballSize ) {
305               diff.normalize().multiplyScalar( ballSize+1 );
306               pos.copy( ballPositionOnSystemParticle ).add( diff );
307 
308             }
309 
310           }
311 
312         }
313 
314 
315         // Floor Constraints
316         for ( i = 0, il = particles.length; i < il; i ++ ) {
317 
318           particle = particles[ i ];
319           pos = particle.pos;
320           if ( pos.y <  -450 ) {
321             //addFriction( particle );
322             pos.y = -449;
323 
324           }
325 
326         }
327     }
328 
329     //Dymic model
330     function Cloth( w, h ) {
331       w = w || 10;
332       h = h || 10;
333       this.w = w;
334       this.h = h;
335 
336       var particles = [];
337       var constraints = [];
338       var particle;
```

```
339
340         var u, v;
341
342         // Create particles
343         for ( v = 0; v <= h; v ++ ) {
344
345           for ( u = 0; u <= w; u ++ ) {
346             particle = new THREE.Object3D( );
347             configureParticle(particle, u / w, 0, v / h, MASS )
348             particles.push( particle );
349
350           }
351
352         }
353
354         // Structural
355         if(structural){
356           for ( v = 0; v < h; v ++ ) {
357
358             for ( u = 0; u < w; u ++ ) {
359
360               constraints.push( [
361                 particles[ index( u, v ) ],
362                 particles[ index( u, v + 1 ) ],
363                 restDistance
364               ] );
365
366               constraints.push( [
367                 particles[ index( u, v ) ],
368                 particles[ index( u + 1, v ) ],
369                 restDistance
370               ] );
371
372             }
373
374           }
```

```
375
376            for ( u = w, v = 0; v < h; v ++ ) {
377
378              constraints.push( [
379                particles[ index( u, v ) ],
380                particles[ index( u, v + 1 ) ],
381                restDistance
382
383              ] );
384
385            }
386
387            for ( v = h, u = 0; u < w; u ++ ) {
388
389              constraints.push( [
390                particles[ index( u, v ) ],
391                particles[ index( u + 1, v ) ],
392                restDistance
393              ] );
394
395            }
396          }
397
398          // Shear
399           if(shear){
400             var diagonalDist = Math.sqrt(restDistance * restDistance *
    2);
401             for (v=0;v<h;v++) {
402              for (u=0;u<w;u++) {
403
404                constraints.push([
405                  particles[index(u, v)],
406                  particles[index(u+1, v+1)],
407                  diagonalDist
408                ]);
409
```

```
410                 constraints.push([
411                   particles[index(u+1, v)],
412                   particles[index(u, v+1)],
413                   diagonalDist
414                 ]);
415 
416               }
417             }
418           }
419 
420 
421 
422            //BENDING SPRING
423            if(bend){
424             var bendingDist = restDistance * 2 ;
425             for (v=0;v<h-2;v++){
426               for (u=0;u<w-2;u++){
427                 constraints.push([
428                   particles[index(u, v)],
429                   particles[index(u+2, v)],
430                   bendingDist
431                 ]);
432 
433                 constraints.push([
434                   particles[index(u, v)],
435                   particles[index(u, v+2)],
436                   bendingDist
437                 ]);
438               }
439              }
440            }
441 
442           this.particles = particles;
443           this.constraints = constraints;
444 
445           function index( u, v ) {
```

```
446
447              return u + v * ( w + 1 );
448
449            }
450
451            this.index = index;
452          }
453
454          //Resets the scene
455          function reset(){
456            scene.remove(clothMesh);
457            delete clothMesh;
458            delete clothMaterial;
459            delete clothGeometry;
460            delete particleSystem;
461            clothPlane = plane( restDistance * xSegs, restDistance * ySegs
        );
462            ballPositionOnSystemParticle = new THREE.Vector3( 0, -200,
      restDistance*ySegs );
463            createCloth();
464            initCloth();
465          }
466
467
468        </script>
469      </body>
470    </html>
```

C. ANEXO

Simulación de una bandera

Para realizar la simulación de una bandera se puede utilizar la tela que hemos diseñado ya que tiene todas las propiedades necesarias para poder realizar una simulación de este tipo.

Para ello, no hay más que rotar la tela de forma que quede en posición vertical, ajustar la dirección del viento, crear el asta de la bandera y sujetar la bandera a dicho asta.

En la figura C.1 se puede observar una foto de esta simulación.

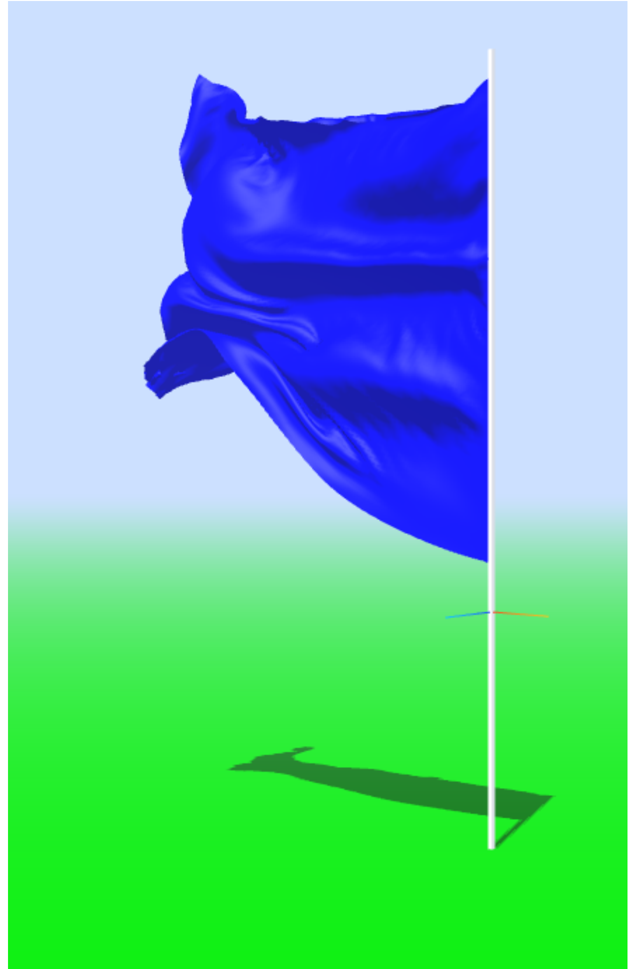

Figura C.1: Foto a la simulación de la bandera

A continuación, presentamos el código de esta simulación.

C.1. Particle.js

```
//Particle.js
//this paticle system has been created for cloth simulation

//CONSTANTS
var DAMPING = 0.03;
var DRAG = 1 - DAMPING;
var MASS = 0.1;
```

```
8
9  var GRAVITY = 981;
10
11 var TIMESTEP = 18 / 1000;
12 var TIMESTEP_SQ = TIMESTEP * TIMESTEP;
13
14 //Variables
15 var restDistance = 5;
16
17 var xSegs = 100;
18 var ySegs = 100;
19
20 var weight = 10;
21
22 var gravityV = new THREE.Vector3( 0,  GRAVITY, 0 ).multiplyScalar(
       MASS );
23
24 var elasticity = 5;
25
26 //Cloth structure
27 var clothPlane = plane( restDistance * xSegs, restDistance * ySegs );
28
29 //Particle functions
30
31 //Configures a THREE.Object3D with a particle characteristics
32 function configureParticle( object3D, x, y, z, mass ){
33   object3D.pos = clothPlane(x, z);
34   object3D.previous = clothPlane(x, z);
35   object3D.original = clothPlane(x, z);
36   object3D.acceleration = new THREE.Vector3( 0, 0, 0 );
37   object3D.velocity = new THREE.Vector3( 0, 0, 0 );
38   object3D.mass = mass * Math.floor((Math.random() * weight) + 1);
39   object3D.invMass = (1 / object3D.mass);
40 }
41
42 //Adds a force to that particle to increase it's acceleration
```

```
43 function addForce ( particle, force ) {
44   var tmpF = new THREE.Vector3(force.x,force.y,force.z);
45   particle.acceleration.sub( tmpF.multiplyScalar( particle.invMass ) )
       ;
46 
47 }
48 
49 //Adds elasticity to the constraint between particles
50 function addElasticity ( particle1, particle2 ){
51   var distNow = new THREE.Vector3(), distPrev = new THREE.Vector3();
52   distNow.subVectors( particle1.pos, particle2.pos );
53   distPrev.subVectors( particle1.previous, particle2.previous );
54   var dif = distNow.length() - distPrev.length();
55   if (dif>0){
56     addForce(particle1, - distNow.normalize().multiplyScalar(dif/2)*
      elasticity);
57     addForce(particle2,  distNow.normalize().multiplyScalar(dif/2)*
      elasticity);
58   }
59 }
60 
61 
62 //Cloth functions
63 function plane( width, height ) {
64 
65   return function( u, v ) {
66 
67     var x = ( u - 0.5 ) * width;
68     var y = ( v + 0.5 ) * height;
69     var z = 0;
70 
71     return new THREE.Vector3( x, y, z );
72 
73   };
74 
75 }
```

```
//Integration methods
//Euler
function euler( particle, time ) {
  particle.previous.copy(particle.pos);
  particle.velocity.add(particle.acceleration.multiplyScalar( timesq )
    );
  particle.pos.add(particle.velocity.multiplyScalar( timesq ));
  particle.acceleration.set( 0, 0, 0 );

};

//Verlet
function verlet( particle, timesq ){
  var tmp = new THREE.Vector3();
  var newPos = tmp.subVectors( particle.pos, particle.previous );
  newPos.multiplyScalar( DRAG ).add( particle.pos );
  newPos.add( particle.acceleration.multiplyScalar( timesq ) );

  particle.previous = particle.pos;
  particle.pos = newPos;
  particle.acceleration.set( 0, 0, 0 );
}
```

C.2. Cloth.js

```
//Cloth variables
var clothGeometry, clothMaterial, clothMesh;

//Particle system
var particleSystem;

//Ball
var ballGeometry, ballMaterial, ballMesh, ballOnOff=false;

```

```
10 //Ground
11 var groundGeometry, groundMaterial, groundMesh;
12
13 //Pole
14 var poleGeometry, poleMaterial, poleMesh;
15
16 //Scene
17 var scene;
18
19 //Camera
20 var camera;
21
22 //Renderer
23 var renderer;
24
25 //Ilumination and material
26 var ambientLight, directionalLight;
27
28 //Stats
29 var stats;
30
31 //Orbit Controler
32 var orbitControler;
33
34 //Controler
35 var controls;
36
37 //Controls parameters
38 var clothColor = 0x0000FF,integrator = 'verlet', structural=true,
       shear=true, bend=true;
39
40 //HTML container
41 var container;
42
43 //Axis
44 var axis;
```

```
45
46  //Wind
47  var wind=true , windStrength , windForce;
48
49  var ballPosition, ballSize;
50
51  //Last time
52  var lastTime;
53
54
55  //Functions
56
57  function initScene(){
58    scene = new THREE.Scene();
59    scene.fog = new THREE.Fog(0xcce0ff, 500, 10000);
60  }
61
62  function initCamera(){
63    camera = new THREE.PerspectiveCamera( 30, window.innerWidth /
        window.innerHeight, 1, 10000 );
64    camera.position.x = 1000;
65    camera.position.y = 50;
66    camera.position.z = 1500;
67    scene.add(camera);
68  }
69
70  function createAxis(){
71    axis = new THREE.AxisHelper( 75 );
72    scene.add(axis);
73
74  }
75
76  function ilumination(){
77    ambientLight = new THREE.AmbientLight( 0x666666 );
78    scene.add(ambientLight);
79
```

```
80    directionalLight = new THREE.DirectionalLight( 0xdfebff, 1.75 );
81    directionalLight.position.set( 50, 200, 100 );
82    directionalLight.position.multiplyScalar( 1.3 );
83
84    directionalLight.castShadow = true;
85
86    directionalLight.shadow.mapSize.width = 1024;
87    directionalLight.shadow.mapSize.height = 1024;
88
89    var d = 1000;
90
91    directionalLight.shadow.camera.left = - d;
92    directionalLight.shadow.camera.right = d;
93    directionalLight.shadow.camera.top = d;
94    directionalLight.shadow.camera.bottom = - d;
95
96    directionalLight.shadow.camera.far = 5000;
97
98    scene.add(directionalLight);
99  }
100
101 function createCloth(){
102   createClothMaterial();
103   createClothGeometry();
104   clothMesh = new THREE.Mesh( clothGeometry, clothMaterial );
105   clothMesh.castShadow = true;
106
107 }
108
109 function createClothMaterial(){
110   clothMaterial = new THREE.MeshPhongMaterial( {
111     specular: 0x030303,
112     color: clothColor,
113     side: THREE.DoubleSide,
114     alphaTest: 0.5
115   } );
```

```
}

function createClothGeometry(){
  clothGeometry = new THREE.ParametricGeometry( clothPlane,xSegs,
    ySegs );
  clothGeometry.dynamic = true;
}

function createGround(){
  createGroundMaterial();
  createGroundGeometry();
  groundMesh = new THREE.Mesh( groundGeometry, groundMaterial );
  groundMesh.position.y = - 250;
  groundMesh.rotation.x = - Math.PI / 2;
  groundMesh.receiveShadow = true;
  scene.add( groundMesh );
}

function createGroundMaterial(){
  groundMaterial = new THREE.MeshPhongMaterial( {
    color:"green",
    specular: 0x111111
  } );
}

function createGroundGeometry(){
  groundGeometry = new THREE.PlaneBufferGeometry( 20000, 20000 );
}

function createBall(){
  ballSize = 100;
  ballPositionOnSystemParticle = new THREE.Vector3( 0, -200, 500 );
  createBallMaterial();
  createBallGeometry();
  ballMesh = new THREE.Mesh( ballGeometry, ballMaterial );
  ballMesh.castShadow = true;
```

```
151   ballMesh.position.set( 0, 0, 0 );
152   ballMesh.receiveShadow = true;
153   scene.add( ballMesh );
154   ballMesh.visible = ! true;
155 }
156 
157 function createBallMaterial(){
158   ballMaterial = new THREE.MeshPhongMaterial( { color: 0xaaaaaa } );
159 }
160 
161 function createBallGeometry(){
162   ballGeometry = new THREE.SphereGeometry( ballSize, 20, 20 );
163 }
164 
165 function createPole(){
166   createPoleMaterial();
167   createPoleGeometry();
168   poleMesh = new THREE.Mesh( poleGeometry, poleMaterial );
169   poleMesh.castShadow = true;
170   scene.add(poleMesh);
171 
172 }
173 
174 function createPoleMaterial(){
175   poleMaterial = new THREE.MeshPhongMaterial( {
176     color: 0xffffff,
177     specular: 0x111111,
178     shininess: 100
179   } );
180 }
181 
182 function createPoleGeometry(){
183   poleGeometry = new THREE.BoxGeometry( 5, 1150, 5 );
184 }
185 
186 function initRenderer(){
```

```
  renderer = new THREE.WebGLRenderer( { antialias: true } );
  renderer.setPixelRatio( window.devicePixelRatio );
  renderer.setSize( window.innerWidth, window.innerHeight );
  renderer.setClearColor( scene.fog.color );
  renderer.gammaInput = true;
  renderer.gammaOutput = true;
  renderer.shadowMap.enabled = true;
}

function initControls(){
  orbitControler = new THREE.OrbitControls( camera,
    renderer.domElement );
  orbitControler.maxPolarAngle = Math.PI * 0.5;
  orbitControler.minDistance = 1000;
  orbitControler.maxDistance = 7500;
}

function initStats(){
  stats = new Stats();
}

function initWind(){
  wind = true;
  windStrength = 100;
  windForce = new THREE.Vector3( 0, 0, 0 );
}

function satisifyConstraints( p1, p2, distance ) {
  var diff = new THREE.Vector3();
  diff.subVectors( p2.pos, p1.pos );
  var currentDist = diff.length();
  if ( currentDist === 0 ) return; // prevents division by 0
  var correction = diff.multiplyScalar( 1 - distance / currentDist );
  var correctionHalf = correction.multiplyScalar( 0.5 );
  p1.pos.add( correctionHalf );
  p2.pos.sub( correctionHalf );
```

```
222
223 }
```

C.3. Cloth.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <title>three.js webgl - cloth simulation</title>
5      <meta charset="utf-8">
6      <meta name="viewport" content="width=device-width,
     user-scalable=no, minimum-scale=1.0, maximum-scale=1.0">
7      <style>
8        body {
9          font-family: Monospace;
10         background-color: #000;
11         color: #000;
12         margin: 0px;
13         overflow: hidden;
14       }
15     </style>
16   </head>
17
18   <body>
19     <script src="../libs/three.js"></script>
20     <script src="../libs/Detector.js"></script>
21     <script src="../libs/OrbitControls.js"></script>
22     <script src="../libs/stats.min.js"></script>
23     <script src="../libs/dat.gui.js"></script>
24
25     <script src="../libs/Particle_Bandera.js"></script>
26     <script src="../libs/Cloth_Bandera.js"></script>
27
28     <script >
29       init();
```

```
      animate();

      //Initializes every object in the scene
      function init(){
        container = document.createElement( 'div' );
        document.body.appendChild( container );

        initScene();
        initCamera();
        createAxis();
        ilumination();

        createCloth();
        initCloth();

        createGround();

        createBall();

        createPole();

        initRenderer();

        container.appendChild( renderer.domElement );

        initControls();

        initStats();

        container.appendChild( stats.dom );

        initWind();

        window.addEventListener( 'resize', onWindowResize, false );

        initMenu()
```

```
66      }
67
68      //Initializes the cloth dynamic model
69      function initCloth(){
70        particleSystem = new Cloth(xSegs, ySegs);
71        clothMesh.position.set(250, -200, 0);
72        scene.add( clothMesh );
73      }
74
75      //Windows resize handler
76      function onWindowResize() {
77
78        camera.aspect = window.innerWidth / window.innerHeight;
79        camera.updateProjectionMatrix();
80
81        renderer.setSize( window.innerWidth, window.innerHeight );
82
83      }
84
85      //Initializes the control menu
86      function initMenu(){
87        controls = new function(){
88          this.windOnOff = true;
89          this.windIntensity = 100;
90          this.windRandom = false;
91          this.windX = 1;
92          this.windY = -1;
93          this.windZ = 0;
94          this.integrator = 'verlet';
95          this.particlesNumberX = 100;
96          this.particlesNumberY = 100;
97          this.particlesDist = 5;
98          this.particlesWeight = 10;
99          this.clothColor = 0x0000FF;
100         this.clothElasticity = 5;
101         this.structuralConstraints = true;
```

```
102         this.shearConstraints = true;
103         this.bendConstraints = true;
104         this.ballOnOff = false;
105         this.ballSize = 100;
106         this.reset = reset;
107       }
108 
109       var gui = new dat.GUI();
110 
111       //Wind controls
112       var windGui = gui.addFolder('Wind');
113       windGui.add(controls,'windOnOff').onFinishChange(function(){
114         wind = controls.windOnOff;
115       });
116       windGui.add(controls,'windIntensity',0, 500);
117       windGui.add(controls,'windRandom');
118       windGui.add(controls,'windX',-1,1).step(1);
119       windGui.add(controls,'windY',-1,1).step(1);
120       windGui.add(controls,'windZ',-1,1).step(1);
121 
122       //Cloth controls
123       var clothGui = gui.addFolder('Cloth');
124       clothGui.addColor(controls,'clothColor').onFinishChange(
    function(){
125         clothColor=controls.clothColor;
126         reset();
127       });
128       clothGui.add(controls,'clothElasticity',0,10).onFinishChange(
    function(){
129         elasticity = controls.clothElasticity;
130       });
131 
132       //Particle System controls
133       var particleSystemGui = gui.addFolder('Particle System');
134       particleSystemGui.add(controls,'integrator', ['verlet', 'euler
    ']).onFinishChange(function(){
```

```
135            integrator = controls.integrator;
136            reset();
137        });
138        particleSystemGui.add(controls,'particlesNumberX', 0, 500).
   step(5).onFinishChange(function(){
139            xsegs = controls.particlesNumberX;
140            reset();
141        });
142        particleSystemGui.add(controls,'particlesNumberY', 0, 500).
   step(5).onFinishChange(function(){
143            ysegs = controls.particlesNumberY;
144            reset();
145        });
146        particleSystemGui.add(controls,'particlesDist', 0.1, 20).
   onFinishChange(function(){
147            restDistance = controls.particlesDist;
148            reset();
149        });
150        particleSystemGui.add(controls,'particlesWeight').min(0).
   onFinishChange(function(){
151            weight = controls.particlesWeight;
152            reset();
153        });
154        particleSystemGui.add(controls,'structuralConstraints').
   onFinishChange(function(){
155            structural = controls.structuralConstraints;
156            reset();
157        });
158        particleSystemGui.add(controls,'shearConstraints').
   onFinishChange(function(){
159            shear = controls.shearConstraints;
160            reset();
161        });
162        particleSystemGui.add(controls,'bendConstraints').
   onFinishChange(function(){
163            bend = controls.bendConstraints;
```

```
164          reset();
165        });
166
167        //Ball controls
168        var ballGui = gui.addFolder('Ball');
169        ballGui.add(controls,'ballOnOff').onFinishChange(function(){
170          ballOnOff = controls.ballOnOff;
171          ballMesh.visible = ballOnOff;
172        });
173        ballGui.add(controls,'ballSize', 10, 500).onChange(function(){
174          var scale = controls.ballSize / ballSize;
175          ballSize = controls.ballSize;
176          ballMesh.geometry.scale(scale,scale,scale);
177        });
178
179        //Reset button
180        gui.add(controls, 'reset');
181
182        gui.remember(controls);
183
184      }
185
186      //Controls the animation
187      function animate() {
188
189        requestAnimationFrame( animate );
190
191          var time = Date.now();
192
193        if(controls.windRandom){
194          windStrength = Math.cos( time / 7000 ) * 20 + 200;
195          windForce.set( Math.sin( time / 2000 ), Math.sin( time /
   3000 ), Math.sin( time / 1000 ) ).normalize().multiplyScalar(
   windStrength );
196        }else{
197          windStrength = controls.windIntensity;
```

```
198         windForce.set( controls.windX*(1+Math.sin( time / 1000 )),
     controls.windY, controls.windZ*Math.sin( time / 1000 ) ).normalize
     ().multiplyScalar( windStrength );
199       }
200 
201       simulate( time );
202       render();
203       stats.update();
204 
205 
206 
207     }
208 
209     //Controls the rendering
210     function render() {
211 
212       for ( var i = 0, il = particleSystem.particles.length; i < il;
      i ++ ) {
213 
214         clothMesh.geometry.vertices[ i ].copy(
     particleSystem.particles[ i ].pos );
215 
216       }
217 
218 
219       clothMesh.geometry.computeFaceNormals();
220       clothMesh.geometry.computeVertexNormals();
221 
222       clothMesh.geometry.normalsNeedUpdate = true;
223       clothMesh.geometry.verticesNeedUpdate = true;
224 
225       camera.lookAt( scene.position );
226 
227       renderer.render( scene, camera );
228 
229     }
```

```
230
231       //Controls the cloth simulation
232       function simulate ( time ){
233         if ( ! lastTime ) {
234           lastTime = time;
235           return;
236
237         }
238
239         var i, il, particles, particle, pt, constraints, constraint,
     pins;
240         var diff = new THREE.Vector3();
241         var tmpForce = new THREE.Vector3();
242         particles = particleSystem.particles;
243
244         if ( wind ) {
245
246           var face, faces = clothMesh.geometry.faces, normal;
247
248           for ( i = 0, il = faces.length; i < il; i ++ ) {
249
250             face = faces[ i ];
251             normal = face.normal;
252
253             tmpForce.copy( normal ).normalize().multiplyScalar(
     normal.dot( windForce ) );
254             addForce( particles[ face.a ], tmpForce);
255             addForce( particles[ face.b ], tmpForce);
256             addForce( particles[ face.c ], tmpForce);
257           }
258
259         }
260
261         // Gravity force
262         for ( i = 0, il = particles.length; i < il; i ++ ) {
263
```

```
264             particle = particles[ i ];
265             addForce( particle, gravityV);
266 
267         }
268 
269         // Elasticity force
270         constraints = particleSystem.constraints;
271         il = constraints.length;
272 
273         for ( i = 0; i < il; i ++ ) {
274 
275             constraint = constraints[ i ];
276             addElasticity(constraint[0],constraint[1]);
277 
278         }
279 
280         //Particles movement integration
281         particles = particleSystem.particles;
282         for ( i = 0, il = particles.length; i < il; i ++ ) {
283             particle = particles[ i ];
284             if(integrator = 'verlet'){
285               verlet( particle, TIMESTEP_SQ );
286             }else if(integrator = 'euler'){
287               euler(particle,TIMESTEP);
288             }
289 
290 
291         }
292 
293         // Start Constraints
294         constraints = particleSystem.constraints;
295         il = constraints.length;
296         for ( i = 0; i < il; i ++ ) {
297 
298             constraint = constraints[ i ];
299             satisifyConstraints( constraint[ 0 ], constraint[ 1 ],
```

```
        constraint[ 2 ] );
300
301         }
302
303         // Ball Constraints
304         if ( ballOnOff ) {
305           for ( i = 0, il = particles.length; i < il; i ++ ) {
306
307             particle = particles[ i ];
308             pos = particle.pos;
309             diff.subVectors( pos, ballPositionOnSystemParticle );
310             if ( diff.length() < ballSize ) {
311               diff.normalize().multiplyScalar( ballSize+1 );
312               pos.copy( ballPositionOnSystemParticle ).add( diff );
313
314             }
315
316           }
317
318         }
319
320
321         // Floor Constraints
322         for ( i = 0, il = particles.length; i < il; i ++ ) {
323
324           particle = particles[ i ];
325           pos = particle.pos;
326           if ( pos.y <  -449 ) {
327             //addFriction( particle );
328             pos.y = -449;
329
330           }
331
332         }
333         //Pole Constraints
334
```

```
335
336         // Pin Constraints
337         pins = particleSystem.pins;
338         for ( i = 0, il = pins.length; i < il; i ++ ) {
339
340           var xy = pins[ i ];
341           var p = particles[ xy ];
342           p.pos.copy( p.original );
343           p.previous.copy( p.original );
344
345         }
346       }
347
348       //Dymic model
349       function Cloth( w, h ) {
350         w = w || 10;
351         h = h || 10;
352         this.w = w;
353         this.h = h;
354
355         var particles = [];
356         var constraints = [];
357         var particle;
358         var pins =[];
359
360         var u, v;
361         // Create particles
362         for ( v = 0; v <= h; v ++ ) {
363
364           for ( u = 0; u <= w; u ++ ) {
365             particle = new THREE.Object3D( );
366             configureParticle(particle, u / w, 0, v / h, MASS )
367             particles.push( particle );
368
369           }
370           pins.push(index(0,v));
```

```
371
372         }
373
374         // Structural
375         if(structural){
376           for ( v = 0; v < h; v ++ ) {
377
378             for ( u = 0; u < w; u ++ ) {
379
380               constraints.push( [
381                 particles[ index( u, v ) ],
382                 particles[ index( u, v + 1 ) ],
383                 restDistance
384               ] );
385
386               constraints.push( [
387                 particles[ index( u, v ) ],
388                 particles[ index( u + 1, v ) ],
389                 restDistance
390               ] );
391
392             }
393
394           }
395
396           for ( u = w, v = 0; v < h; v ++ ) {
397
398             constraints.push( [
399               particles[ index( u, v ) ],
400               particles[ index( u, v + 1 ) ],
401               restDistance
402
403             ] );
404
405           }
406
```

```
407         for ( v = h, u = 0; u < w; u ++ ) {
408 
409           constraints.push( [
410             particles[ index( u, v ) ],
411             particles[ index( u + 1, v ) ],
412             restDistance
413           ] );
414 
415         }
416       }
417 
418       // Shear
419        if(shear){
420          var diagonalDist = Math.sqrt(restDistance * restDistance *
    2);
421          for (v=0;v<h;v++) {
422           for (u=0;u<w;u++) {
423 
424             constraints.push([
425               particles[index(u, v)],
426               particles[index(u+1, v+1)],
427               diagonalDist
428             ]);
429 
430             constraints.push([
431               particles[index(u+1, v)],
432               particles[index(u, v+1)],
433               diagonalDist
434             ]);
435 
436           }
437         }
438       }
439 
440 
441 
```

```
442         //BENDING SPRING
443         if(bend){
444          var bendingDist = restDistance * 2 ;
445          for (v=0;v<h-2;v++){
446            for (u=0;u<w-2;u++){
447              constraints.push([
448                particles[index(u, v)],
449                particles[index(u+2, v)],
450                bendingDist
451              ]);
452
453              constraints.push([
454                particles[index(u, v)],
455                particles[index(u, v+2)],
456                bendingDist
457              ]);
458            }
459          }
460         }
461
462        this.particles = particles;
463        this.constraints = constraints;
464        this.pins = pins;
465
466        function index( u, v ) {
467
468          return u + v * ( w + 1 );
469
470        }
471
472        this.index = index;
473        console.log(this);
474      }
475
476      //Resets the scene
477      function reset(){
```

```
478            scene.remove(clothMesh);
479            delete clothMesh;
480            delete clothMaterial;
481            delete clothGeometry;
482            delete particleSystem;
483            createCloth();
484            initCloth();
485          }
486
487
488      </script>
489    </body>
490  </html>
```

Bibliografía

[Bridson et al., 2002] Bridson, R., Fedkiw, R., and Anderson, J. (2002). Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph.*, 21(3):594–603.

[House and Breen, 2000] House, D. H. and Breen, D. E., editors (2000). *Cloth Modeling and Animation*. A. K. Peters, Ltd., Natick, MA, USA.

[Ng and Grimsdale, 1996] Ng, H. N. and Grimsdale, R. L. (1996). Computer graphics techniques for modeling cloth. *IEEE Comput. Graph. Appl.*, 16(5):28–41.

[Selle et al., 2009] Selle, A., Su, J., Irving, G., and Fedkiw, R. (2009). Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):339–350.

[Terzopoulos et al., 1987] Terzopoulos, D., Platt, J., Barr, A., and Fleischer, K. (1987). Elastically deformable models. *SIGGRAPH Comput. Graph.*, 21(4):205–214.