

Animación y Simulación Avanzada

Practica 4: Elementos Finitos

Esta práctica consiste en crear una simulación completa utilizando el simulador FEM Unity3D que hemos desarrollado en clase. La escena simulada será de elección libre, pero la simulación deberá contar con las siguientes características:

- Malla de simulación gruesa, ~500 tetrahedros
- Malla de simulación fina embebida en la anterior
- Detección de colisiones de la malla embebida

Estas tres características se corresponden con tres partes de la práctica.

Parte 1: Generar malla de simulación e inicializar simulador (2 puntos)

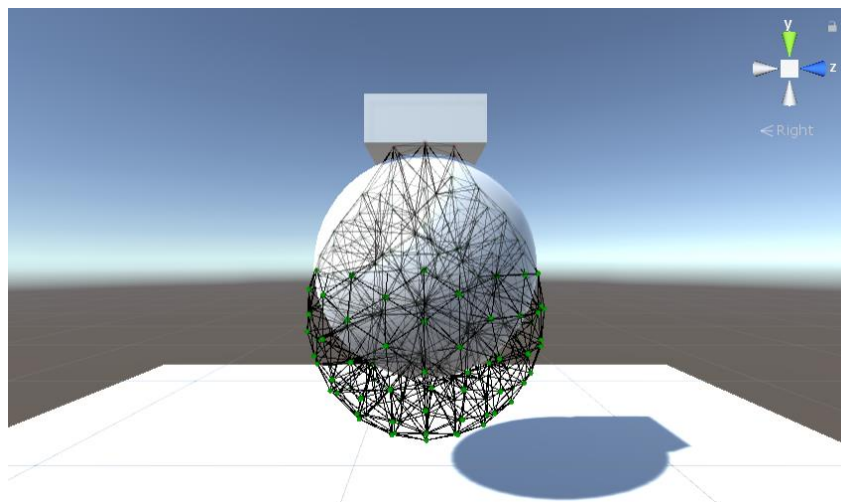
Partiendo de una superficie 3D cualquiera, habrá que utilizar el software TetGen para crear la correspondiente malla volumétrica de simulación. Se proporciona sendos ejecutables de TetGen y TetView para 32/64bit en la carpeta Assets/Resources. Los archivos deberán situarse en la carpeta Assets/Resources siguiendo el siguiente criterio de nombrado:

- [nombre]_viz.obj: malla superficial del objeto
- [nombre]_sim.1.node: nodos de simulación generados por TetGen
- [nombre]_sim.1.ele: tetraedros de simulación generados por TetGen

Una vez obtenidos los archivos, será necesario parsearlos y crear los nodos, tetraedros y el objeto simulable correspondientes. Esto se llevará a cabo completando tres métodos de la clase ObjectLoader:

- CreateSimulable: crea y devuelve el objeto simulable
- ParseTetGenNodes: parsea las líneas del archivo para extraer una lista de vértices
- ParseTetGenTetras: parsea las líneas del archivo para extraer una lista de índices

Finalizada esta parte, deberá ser posible ejecutar ya la escena Loading. La simulación resultante no tiene en cuenta la malla embebida, y solo deforma la malla de simulación. En la parte 2 resolveremos este problema.



Parte 2: Incrustar malla detallada en la malla de simulación (3 puntos)

En esta parte de la práctica se completarán tres métodos que permiten calcular y almacenar las coordenadas paramétricas de cada vértice embebido e interpolar su posición en función de las posiciones de los nodos de su tetraedro. Para ello se utilizará la clase `EmbeddedPoint`, que relaciona la coordenada iso-paramétrica de cada punto embebido con su posición deformada:

```
/// <summary>
/// Stores the iso-parametric coordinate of an embedded point
/// together with the current deformed position. The deformed
/// position gets updated whenever the deformation of the
/// tetrahedras changes.
/// </summary>
public class EmbeddedPoint
{
    public VectorXD isoParam;
    public VectorXD position;
}
```

Será necesario completar tres métodos de la clase `Tetrahedron` y la clase `FEMSystem`.

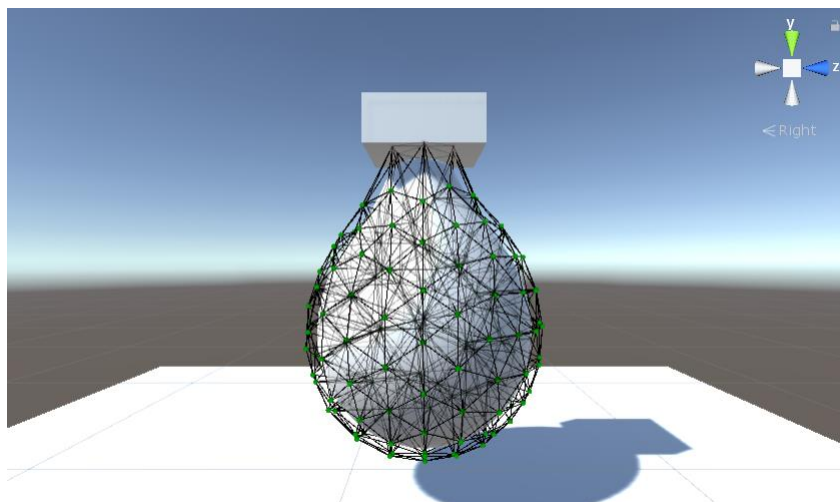
En la clase `FEMSystem`:

- `EmbedMesh`. Este método se invoca durante la inicialización. Para cada uno de los vértices de la malla de visualización proporcionada como parámetro, busca el punto embebido correspondiente e inicializa la coordenada iso-paramétrica. Almacena el resultado en la lista que se dispone. Debe utilizar el método `Tetrahedron.TryAddEmbedded`.

En la clase `Tetrahedron`:

- `TryAddEmbedded`. Este método se debe invocar desde `FEMSystem.EmbedMesh`. Comprueba si el vértice que se pasa como parámetro está dentro del tetraedro, en cuyo caso, inicializa su coordenada paramétrica y almacena el punto en la variable lista `Tetrahedron.embeddedPoints`.
- `UpdateEmbedded`. Este método se invoca al final de cada paso de simulación. Actualiza la posición deformada de cada vértice embebido en el tetraedro almacenado en la variable `Tetrahedron.embeddedPoints`.

Una vez completada esta parte, al ejecutar la simulación se observará como la malla detallada se deforma con la malla de simulación como resultado de interpolar sus posiciones usando las funciones de forma.



Parte 3: Implementar contacto plano-malla embebida (2 puntos)

En esta parte de la práctica se implementarán colisiones a nivel de malla embebida. La clase FEMSystem guarda una lista de obstáculos en forma de objetos de la clase GameObject. Se considerará que estos obstáculos son simplemente planos, por lo que se puede obtener su posición y normal simplemente accediendo al objeto Transform. Los vértices embebidos en el objeto pueden potencialmente colisionar con cualquiera de estos planos. Se debe detectar esta colisión y generar la correspondiente respuesta en forma de fuerzas y Jacobianas. Esto se llevará a cabo completando 4 métodos:

En la clase Tetrahedron:

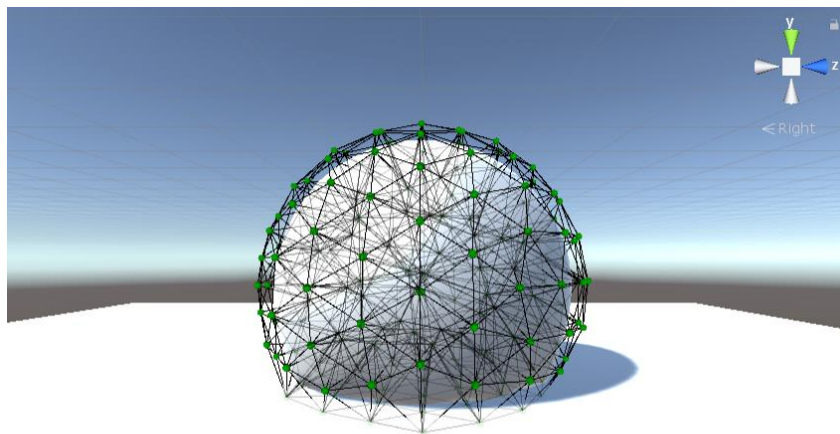
- AddCollisionForceEmbedded: calcula la fuerza de la colisión correspondiente a los vértices embebidos en el tetraedro y la añade al vector global de fuerzas del sistema. Se proporcionan punto del plano, normal del plano y rigidez de la colisión.
- AddCollisionJacobianEmbedded: lo mismo que el método anterior, pero con las Jacobianas. Fijarse en que la Jacobiana debe considerar como cambia la fuerza en un nodo debido al efecto de cualquier otro nodo del tetraedro.

En la clase FEMSystem:

- AddCollisionForceEmbedded: recorre los tetraedros ensamblando fuerzas
- AddCollisionJacobianEmbedded: recorre los tetraedros ensamblando Jacobianas

Opcionalmente se podrán completar también las versiones de AddCollisionJacobianEmbedded que aparecen sugeridas tanto en Tetrahedron.cs como en FEMSystem.cs, correspondientes a la implementación sparse.

Una vez completada esta parte, se deberá cambiar la variable CollType de ObjectLoader al valor EmbeddedNode. Entonces, será posible simular un objeto con contacto a nivel de malla embebida y visualizar correctamente la deformación del objeto mediante las funciones de forma.



Parte 4: Extensiones opcionales (3 puntos)

A partir de aquí, es posible añadir otras características a la escena o al simulador para optar a la mejor nota posible. Se deja a elección libre de cada uno, pero hacer (o intentar) alguna extensión se valorará positivamente. Algunas sugerencias:

- **Mejorar rendimiento:** esta implementación es un FEM lineal, eso implica que algunas de las magnitudes utilizadas pueden ser precomputadas y almacenadas en vez de recalculadas. Modificar el código a tal efecto.
- **Colisiones más complejas:** la versión básica solo permite colisiones contra planos. Resulta muy fácil calcular colisiones con respecto a otras superficies paramétricas como esferas o cápsulas, dado que se puede detectar la colisión y calcular penetraciones y normales usando fórmulas cerradas.
- **Colisiones arbitrarias:** aunque laborioso, se podrían incluir colisiones contra elementos del entorno más complejos. Una malla 3D es una colección de secciones de planos (caras). Implementar esta característica, potencialmente añadiendo alguna estructura de almacenamiento espacial para acelerar la búsqueda.
- **Simulación de varios objetos:** las prácticas desarrolladas en la asignatura correspondientes a la parte de masa-muelle y sólido rígido utilizan una estructura simulable-manager muy similar a la de esta práctica. El código está casi preparado para poder simular en la misma escena (y con un mismo solve lineal) varios objetos distintos. Modificar el código a tal efecto y mostrar una escena de ejemplo.
- **Factores estéticos:** usar la simulación dentro de una animación 3D que tenga sentido.

Si realizas alguna de estas extensiones, adjunta con la solución el proyecto completo de Unity3D, junto con un pequeño documento describiendo lo que se ha hecho (realmente pequeño, estrictamente lo que sea necesario).

El entorno

El entorno en el que se realiza la práctica es muy similar al visto en otras prácticas de la asignatura. La clase PhysicsManager se encarga de almacenar los objetos simulables y resolver cada step de simulación.

Objetos a cargar, contienen ObjectLoader ←

GameObjects que se considerarán obstáculos ←

Cajas utilizadas para fijar nodos en su interior ←

Opciones del solver. Desmarcar Pause para iniciar la simulación. Marcar Step para dar solo un paso. ←

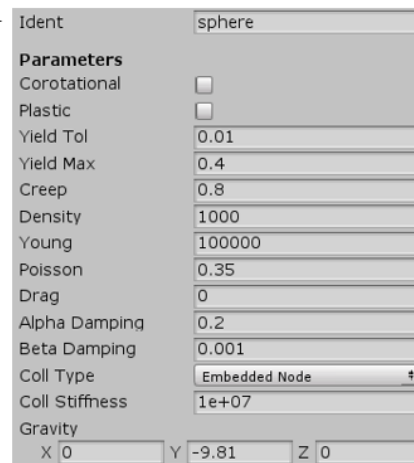
Si CreateGrid está marcado, en lugar de leer los objetos con ObjectLoader, se genera una simulación de un grid X·Y·Z. Funciona sin hacer ningún cambio al código. ←

The screenshot shows the Unity Inspector for a PhysicsManager component. The 'Loaded Objects' section has a dropdown menu showing 'Object0 (ObjectLoader)'. The 'Coll Obstacles' section has a dropdown menu showing 'Plane'. The 'Node Fixers' section has a dropdown menu showing '0'. The 'Integration' section has a 'Paused' checkbox checked, a 'Step' checkbox unchecked, a 'Time Step' field set to '0.05', an 'Integration Method' dropdown set to 'Implicit', and a 'Solver Type' dropdown set to 'Dense'. The 'Debug grid' section has a 'Create Grid' checkbox unchecked, a 'Fix Points' checkbox checked, and various numerical fields for 'Dims', 'Gravity', 'Young', 'Poisson', 'Density', 'Drag', 'Alfa Damping', 'Beta Damping', 'Corotational', 'Plastic', 'Yield Tol', 'Yield Max', 'Creep', and 'Coll Stiffness'.

La principal diferencia en esta práctica es que para cada objeto simulable se debe crear un objeto que contenga el componente ObjectLoader. Este componente busca en el disco las mallas de simulación y visualización generadas.

Nombre de las mallas a cargar. Es decir, el valor de [Nombre] en la explicación que figura en la Parte 1.

Parámetros de simulación explicados durante las clases. Los valores default deberían generar buenas simulaciones.



The image shows a software interface for configuring simulation parameters for an object named 'sphere'. The interface is divided into two main sections: 'Ident' and 'Parameters'. The 'Ident' section contains a text field with the value 'sphere'. The 'Parameters' section contains a list of simulation parameters, each with a corresponding input field or checkbox. The parameters and their values are: Corotational (checkbox, unchecked), Plastic (checkbox, unchecked), Yield Tol (0.01), Yield Max (0.4), Creep (0.8), Density (1000), Young (100000), Poisson (0.35), Drag (0), Alpha Damping (0.2), Beta Damping (0.001), Coll Type (Embedded Node), Coll Stiffness (1e+07), and Gravity (X: 0, Y: -9.81, Z: 0).

Parameter	Value
Ident	sphere
Corotational	<input type="checkbox"/>
Plastic	<input type="checkbox"/>
Yield Tol	0.01
Yield Max	0.4
Creep	0.8
Density	1000
Young	100000
Poisson	0.35
Drag	0
Alpha Damping	0.2
Beta Damping	0.001
Coll Type	Embedded Node
Coll Stiffness	1e+07
Gravity	X: 0, Y: -9.81, Z: 0

Entrega

- Parte obligatoria: Archivos de texto Tetrahedron.cs, FEMSystem.cs y ObjectLoader.cs
- Extensiones: .zip con la carpeta Assets completa y pequeño documento explicativo.

--

¡Buena suerte! 😊