Universidad Zaragoza
1542

Trabajo Fin de Grado

Modelado de la física de objetos blandos en tiempo real basado en el uso del SDK Bullet.

Autor/es

Ignacio Ruiz Martín

Director/es

Francisco José Serón Arbeloa

Escuela de Ingeniería y Arquitectura / Universidad de Zaragoza

2014

Modelado de la física de objetos blandos en tiempo real basado en el uso del SDK Bullet.

RESUMEN

El trabajo realizado ha consistido en el estudio de una librería de renderizado gráfico llamada Ogre3D y una librería de simulación física llamada BulletPhysics para su posterior aplicación al mundo de los videojuegos.

El resultado final muestra una escena que podría ser de cualquier videojuego, renderizada en Tiempo Real. En ella se muestran varios objetos entre los que se encuentran el suelo, el cielo, el personaje principal y varios objetos con comportamientos físicos diferentes. Todos ellos interactúan físicamente de manera real.

El estudio y posterior desarrollo e implementación del trabajo ha estado centrado en crear y gestionar objetos con comportamiento físico de objetos blandos.

A mis padres y mi hermano, por hace esto posible

y aguantarme estos cuatro años.

A África, porque me dio el empujón necesario.

Este Trabajo Fin de Grado corresponde al plan de Estudios del Grado de Ingeniería Informática de la Universidad de Zaragoza.

El número de créditos definidos en el plan de estudios es de 12 créditos.

1. Índice

2. Introducción

2.1 Ámbito de conocimientos

El mundo de los videojuegos está en constante cambio y evolución. No es difícil ver las diferencias entre los videojuegos de hace veinte años y los actuales. Se puede decir que el mayor cambio que estos han sufrido está relacionado con los gráficos, y con la jugabilidad.

El consumidor de videojuegos, además de la diversión inherente a un buen diseño del producto, busca una apariencia cada vez más real que le proporcione una experiencia de inmersión en sus horas de juego.

En relación a ello, es fundamental la simulación de fenómenos físicos que haga creíble aquello que vemos en la pantalla. No es extraño ver en un videojuego caídas de objetos o choques entre ellos que nos parecen irreales, que no se ajustan a lo que nuestro cerebro suele procesar en la realidad; esto es, principalmente, porque muchas veces las físicas están hechas por artistas, animadores, y no simuladas teniendo en cuenta buenos modelos.

Los productos de software que proporcionan simulaciones físicas en los videojuegos se denominan motores de físicas y son los encargados de especificar un modelo físico creíble del mundo.

Como bien sabemos, el mundo de los videojuegos es Tiempo Real, por ello el tiempo del que disponemos para calcular cada *frame* está muy limitado, y además, gran parte de éste, es consumido por el propio motor gráfico. Por lo tanto, es crucial una gestión eficiente de los recursos para consumir el menor tiempo posible a la hora de simular las físicas.

2.2 Objetivo

El objetivo principal de este trabajo fin de grado consiste en simular cuerpos rígidos y blandos sometidos a la acción de fuerzas externas y gravedad.

Para ello, se han elegido dos herramientas de código libre, con las ventajas que esto supone: su uso es gratuito, existe la posibilidad de modificar el código para cubrir alguna necesidad, y tienen una comunidad muy activa dispuesta a resolver cualquier problema que surja.

Para la renderización de los gráficos se utiliza Ogre3D[1]. Ogre3D es una interfaz orientada a objetos escrita en el lenguaje de programación C++. Esta interfaz es capaz

de establecer un bucle de renderización que mostrará por pantalla con una calidad suficientemente real objetos que añadamos a la escena.

Para el análisis de la herramienta, así como el aprendizaje de la misma, se ha consultado la página principal (1-Sitio web de Ogre3D), la wiki (2-Wiki de Ogre3D), el foro (3-Foro de Ogre3D) y la documentación de la API (4-API de Ogre3D) Una explicación más detallada sobre Ogre3D puede ser encontrada en el anexo I.

Para la simulación de físicas se ha utilizado BulletPhysics. Ésta es, igualmente, una librería orientada a objetos escrita en C++. Es capaz, mediante la utilización de objetos virtuales, de simular un el mundo físico, detectando colisiones entre los objetos, así como otras interacciones físicas con el medio, como fricción o gravedad.

Con el fin de estudiar las posibilidades de la librería BulletPhysics y aprender a usarla, así como consultar errores que han aparecido durante la realización del proyecto, se ha consultado la página web de la librería (5-Sitio web de BulletPhysics), la wiki (6-Wiki de BulletPhysics), el manual de uso (7-Manual de BulletPhysics) y el foro (8-Foro de BulletPhysics).

Con todo esto en mente, los objetivos particulares de este proyecto son:

1. Estudio de la librería de código abierto para física en tiempo real, BulletPhysics.
2. Estudio de la librería de código abierto para renderizado de objetos 3D en tiempo real, Ogre3D.
3. Desarrollo e implementación de un módulo de integración de BulletPhysics en Ogre3D.
4. Análisis de sus posibilidades desde el punto de vista de la simulación fenomenológica.
5. Estudio e implementación de la simulación de cuerpos blandos en videojuegos usando BulletPhysics + Ogre3D.
6. Análisis de su eficiencia.

3 Trabajo desarrollado

El trabajo ha consistido en el desarrollo e implementación necesarios para la presentación de una escena renderizada en tiempo real en la que se simulan fenómenos físicos, cuerpos rígidos y blandos sometidos a fuerzas externas y fuerza gravedad.

En la figura 1 podemos ver una captura de pantalla tomada de la aplicación final. En ella, se pueden apreciar los diferentes tipos de objetos que incluimos en el sistema. Además, podemos comprender que existe una cámara desde la que se observa la escena y una pantalla en la que se renderiza la escena.

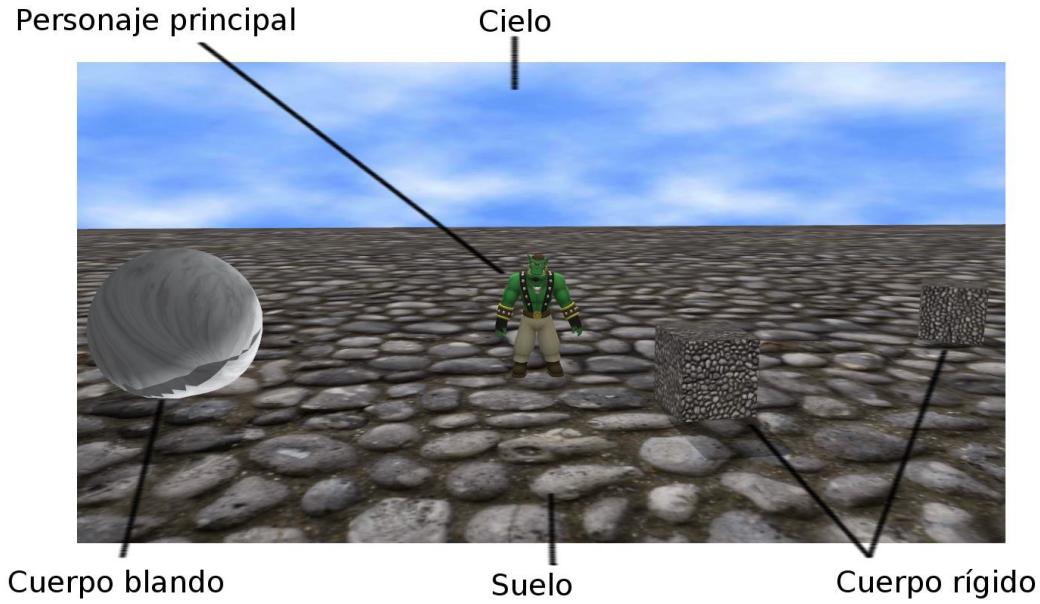


Figura 1: Captura de la aplicación finalizada

3.1 Aplicación vista desde alto nivel

Para alcanzar el objetivo del proyecto, se ha programado una aplicación que se centra en dos pasos fundamentales.

- Creación de la escena y adición de nodos visibles y físicos a la misma.
- Control del bucle de renderización donde se actualizan todas las informaciones, *frame* a *frame.*

Cada una de las clases que se estudiarán más detalladamente a lo largo de la memoria se puede agrupar en una de estas dos funciones, o en las dos.

Para entender cuáles son los componentes de la aplicación, obsérvese la figura 2, en ella se puede ver el proceso que se sigue en cada *frame* desde que se comienza hasta que se muestra por pantalla.

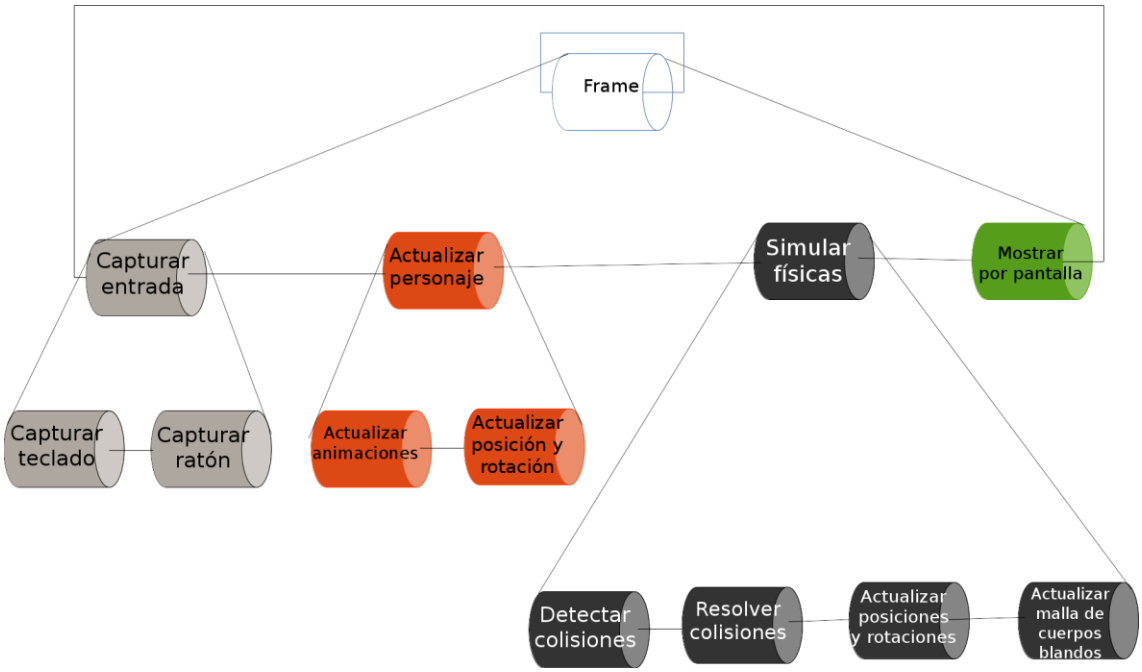


Figura 2: Esquema del proceso de renderización de un *frame*.

Las clases que se verán a continuación implementan este esquema de renderizado,. Bien creando los objetos que se utilizarán, o bien siendo las encargadas de actualizarlos.

3.2 Diagrama de clases

La arquitectura de la aplicación desarrollada se puede observar en la figura 3. En los apartados siguientes se explica de manera detallada cuál es la función de cada clase y se ha implementado.

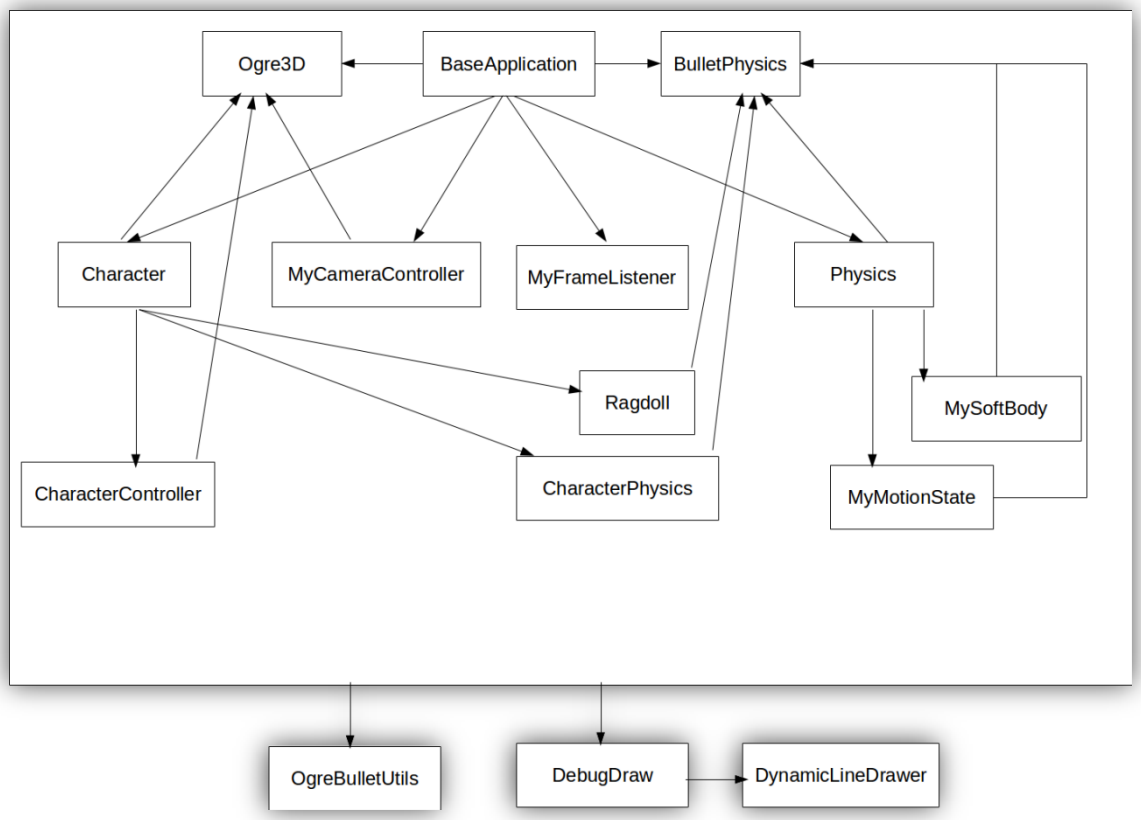
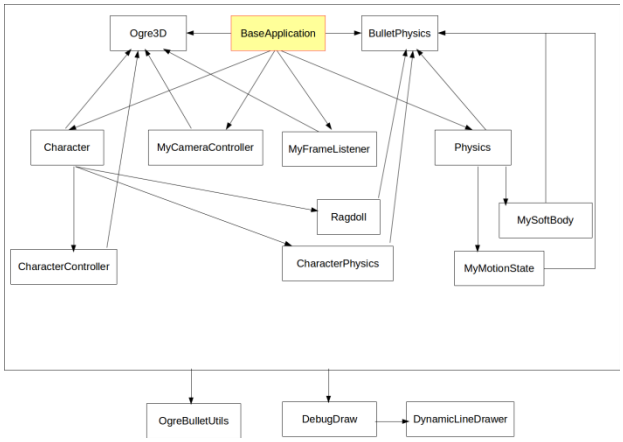


Figura 3: Diagrama de clases de la aplicación

Todas ellas salvo Ogre3D y BulletPhysics han sido implementadas en última instancia por el autor. Los trozos de código que han sido reutilizados de otros usuarios son debidamente nombrados y enlazados en los comentarios del código.

En el apéndice III se puede encontrar la estructura completa de la aplicación desarrollada, así como la documentación del código.

3.3 BaseApplication



Esta es la clase principal del sistema. Se encargar de la inicialización de los diferentes componentes, así como de la creación de la escena en Ogre3D.

Para que este motor gráfico funcione correctamente necesita que se inicialicen y configuren los siguientes componentes:

- Primero, crear el *Ogre::Root*, éste es el punto de acceso del sistema. Con ello, Ogre3D cargará las configuraciones necesarias. Es el primer objeto que debe ser creado, y el último que debe ser destruido.
- Segundo, se cargan los recursos (mallas de polígonos, texturas, etc.) mediante el uso de un objeto *Ogre::ResourceGroupManager*.
- Tercero, se crea el objeto *Ogre::SceneManager*, que consiste en el grafo de la escena. Ver figura 4.
- Cuarto, se crea la cámara con el objeto *Ogre::Camera* y el *viewport*, usando *Ogre::Viewport*, que define la parte de la pantalla donde se renderizarán objetos.

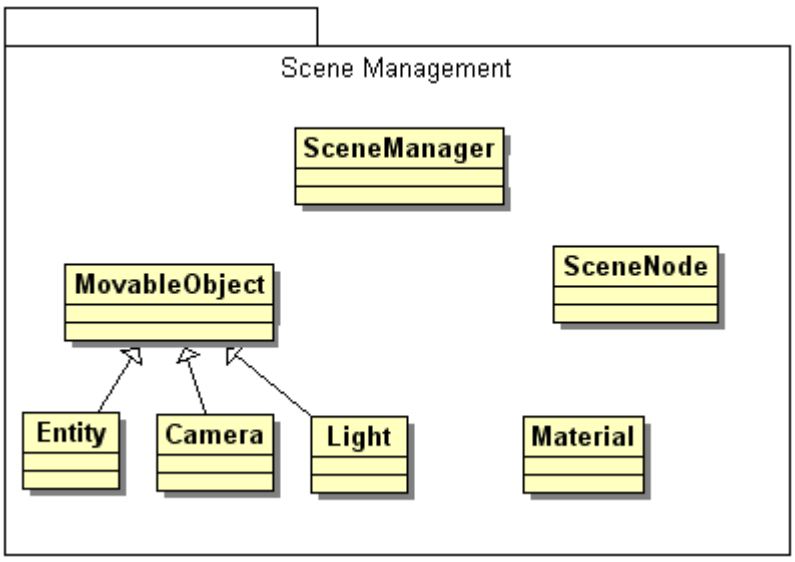


Figura 4: Grafo de escena de Ogre3D

La clase BaseApplication también se encarga de inicializar los demás componentes principales que ha sido necesario programar para la realización de este proyecto y que serán explicados con detalle posteriormente en la clase que corresponda, estos son:

- *MyFrameListener:* Es la clase encargada de controlar lo que debe de ocurrir en cada *frame*.
- *Physics*: Objeto cuya función es la de llevar a cabo en cada *frame* la simulación fenomenológica de la física.
- *MyCameraController:* Se encargará de hacer los movimientos necesarios de la cámara de manera que se puedan observar bien lo que ocurre en la escena.
- *Character*: Gestiona la creación y el movimiento del personaje principal de la aplicación.

3.3.1 Creación de la escena

Ogre3D funciona mediante la creación de objetos 3D que son añadidos a una escena. Posteriormente, en su bucle de renderización, el motor recorrerá todos los objetos que existen, decidirá cuáles son los que se tienen que ver en ese momento y los mostrará en la pantalla.

Las escenas son creadas y gestionadas mediante el grafo de la escena, al que Ogre3D nos proporciona acceso utilizando el objeto *Ogre::SceneManager*.

Además, los objetos 3D en Ogre3D están compuestos de nodos y entidades. Los nodos son la posición y la rotación en la escena. Las entidades son las encargadas de darle forma y apariencia concreta a cada nodo.

La escena que se presenta en esta aplicación consta de los siguientes objetos:

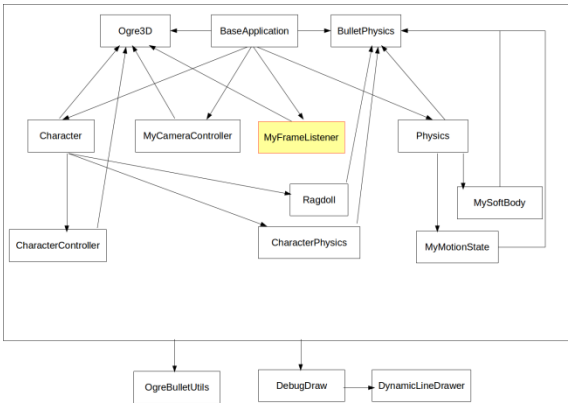
- El cielo: Está creado mediante el uso de una herramienta de Ogre3D llamada *SkyDome* a la que se puede asignar una textura y la presenta como una cúpula. En este caso, se ha texturizado con una textura dinámica, de manera que las nubes parecen estar en movimiento.
- El personaje principal: Hablaremos de él más adelante en la clase que se encarga de gestionarlo.

- El suelo: Éste es en principio un objeto *Plane* de Ogre3D con una textura, pero de cara a que la textura parezca más real, se le ha aplicado un BumMapping[2]. Esta técnica consiste en utilizar un mapa de normales en la etapa de iluminado para darle aspecto de rugosidad a la superficie sin necesidad de modificar su malla poligonal. [3]

[2] Página de la Wikipedia de BumpMapping: http://en.wikipedia.org/wiki/Bump_mapping
[3] Tutorial de implementación de Bum Mapping: http://www.ogre3d.org/tikiwiki/Materials#Advanced_Materials

- El objeto blando: Es un objeto cuyo comportamiento fenomenológico es el de un cuerpo blando.
- Objetos rígidos: Son objetos cuyo comportamiento fenomenológico es el de cuerpos rígidos.

3.4 MyFrameListener



Ogre3D tiene un sistema de trabajo que está basado en eventos. Se producen eventos cuando un frame va a ser renderizado, cuando está siendo renderizado, cuando ha acabado, cuando se pulsa una tecla, cuando se mueve el ratón, etc.

Esta clase es la encargada de recibir todos estos eventos y procesarlos.

Es una clase que hereda de *Ogre::FrameListener*, *Ogre::WindowEventListener*, *OIS::KeyListener* y *OIS::MouseListener*.

OIS[4] es una librería orientada a objetos que controla los dispostivos de entrada.

Todo ello asegura que, implementando los métodos predefinidos que se han seleccionado es posible gestionar los eventos.

3.4.1 Eventos cada frame

Evento de inicio de renderización de un *frame*: Cuando Ogre3D comienza a renderizar un *frame*, éste invoca el método *frameStarted*() indicándole mediante un evento cuánto tiempo ha transcurrido desde la renderización del último *frame*. Dentro de este método se realizan llamadas a las funciones de las clases que controlan los periféricos de entrada y salida de OIS con objeto de capturar sus cambios.

Evento de renderización de un *frame:* Cuando Ogre3D está renderizando un frame envía un evento, éste es recogido en el método *frameRenderingQueued()*, que actualiza los siguientes componentes:

[4] Explicación de OIS: http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Using+OIS

- La actualización del personaje principal se realiza mediante la llamada a la función:

Character->**updateCharacter**(timeSinceLastFrame)

- La actualización del mundo físico se realiza llamando a:

Physics->**stepSimulation**(timeSinceLastFrame)

- La actualización de la malla del objeto blando en función de lo que haya informado el motor de físicasse realiza llamando a:

Physics->updateSoftBodies()

Evento de finalización de renderizado de un *frame:* Cuando Ogre3D termina de renderizar un *frame*, el método *frameEnded()* es llamado, este caso, únicamente utilizamos éste para motivos de *debug* y mediciones de tiempo.

3.4.2 Eventos de periféricos de entrada y salida

En este proyecto las pulsaciones del teclado pueden se controlan de manera asíncrona utilizando los siguientes métodos:

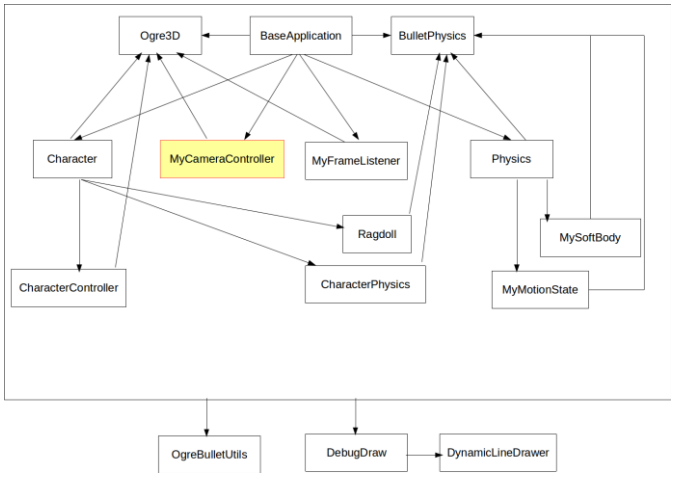
- *keyPressed():* Se llama a este método cuando se pulsa una tecla para detectar qué tecla es la que ha activado el evento. En esta aplicación, se utiliza este evento para:
  - Activar el *debug* de las físicas pulsando la tecla L.
  - Lanzar un cubo pulsando la tecla F.
  - Cerrar la aplicación pulsando la tecla ESC.
  - Mandarle el evento a la clase *Character* que moverá el personaje.
- *keyReleased():* Se llama a este método cuando se deja de pulsar una tecla que había sido pulsada previamente. En este se le manda el evento a la clase *Character* para la gestión del movimiento del personaje.

El ratón se controla mediante la conjunción de los métodos *mousePressed(), mouseReleased() y mouseMoved().* En este proyecto, se han utilizado para rotar la cámara y hacer zoom. En concreto, se rota la cámara si al mover el ratón uno de los dos botones está pulsado, y se hace zoom cuando se mueve la ruleta.

3.4.3 Eventos relacionados con la gestión de ventanas

El método *windowClosed()* se utiliza para capturar cuando se ha cerrado la ventana y así destruir los objetos necesarios y *windowsResized()* se utiliza para recalcular tamaños y posiciones del ratón.

3.5 MyCameraController



Para observar lo que ocurre en la escena, lo más cómodo es utilizar una cámara en tercera persona que se pueda manejar con el ratón.

Esta clase es la encargada de gestionar los movimientos de cámara. Lo que hace es establecer tres nodos. El nodo *mainNode* es el nodo donde está el personaje, el nodo *sightNode* es el nodo hacia donde la cámara tiene que mirar y el nodo *desiredCameraNode* es el nodo que se establece como objetivo en las traslaciones de la cámara.

La cámara es actualizada en cada *frame* utilizando el método *update()* y, en función del tiempo que haya pasado entre *frames*, actualiza su posición para acercarse a la del nodo *desiredCameraNode.* Obsérvese en la figura 5 el funcionamiento del movimiento de los tres nodos.

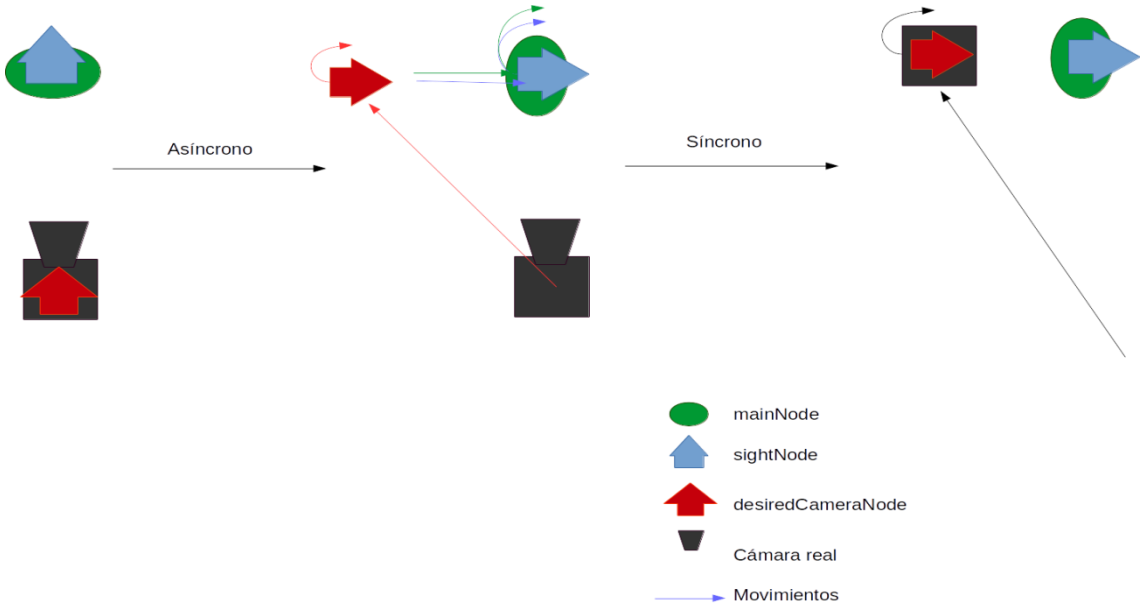


Figura 5: Movimiento de cámara que sigue al personaje principal

El nodo *desiredCameraNode* está emparentado con el nodo *sightNode*, de manera que se mueve y mantiene su posición relativa siempre que el padre lo hace. Así, sólo hay que mover el nodo *sightNode* cuando el personaje se mueve, para que la cámara actualice su posición y lo siga en el siguiente frame.

Además, los métodos *injectMouseMove()* y *adjustZoom()* de la clase se encargan de gestionar la rotación en los ejes X e Y y la posición en el eje Z, respectivamente. Estos movimientos en X e Y se producirán cuando se mueva el ratón y actúan en girando el nodo *sightNode*, mientras que el nodo *desiredCameraNode* mantiene su posición relativa, ver figura 6. El zoom se produce cuando se mueve la ruleta del ratón y lo que hace es disminuir la distancia entre el nodo *desiredCameraNode* y el *sightNode*. Obsérvese en la figura 6 su funcionamiento.

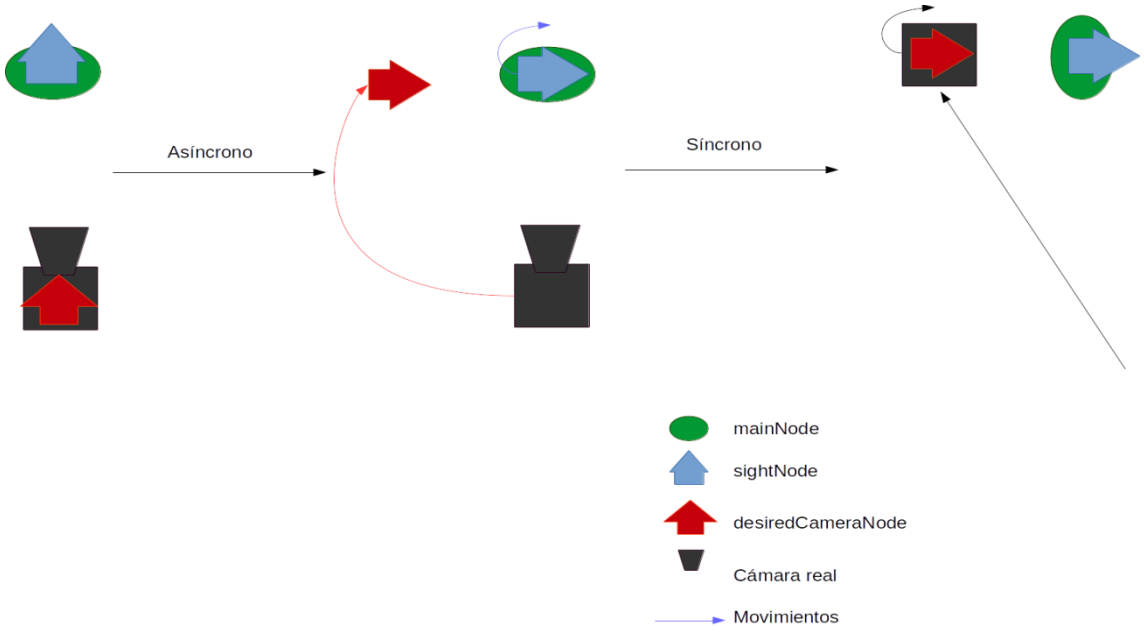
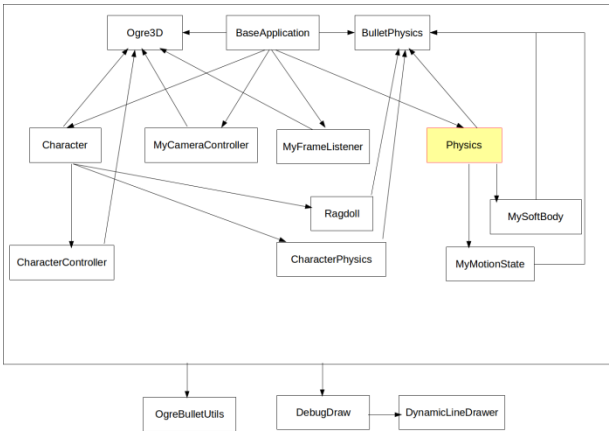


Figura 6: Rotación de cámara

En la figura 7, se muestra una captura de pantalla de la aplicación, obsérvese cómo se puede ver el personaje desde cierto ángulo gracias al funcionamiento de la cámara.

Figura 7: Personaje principal visto por delante

3.6 Physics



Como ya hemos dicho anteriormente, esta clase será la encargada de implementar el motor de físicas, para ello hace uso de la librería BulletPhysics.

La clase se tiene que encargar tanto de añadir objetos al mundo físico, como de actualizar su posición y forma (en caso de que sea blando) tras cada *frame*.

Lo primero que hace es inicializar y configurar las físicas. En este momento se caracteriza el mundo físico, para ello es necesario decidir lo siguiente:

- El tipo de colisiones que tendrá el mundo se genera creando un objeto que herede de btDefaultCollisionConfiguration de Bullet, para el proyecto se ha elegido btSoftBodyRigidBodyCollisionConfiguration porque en nuestra escena habrá objetos sólidos y blandos.
- Para el cálculo rápido de una primera aproximación de los pares de objetos que pueden estar colisionando, hay que seleccionar un algoritmo de *BroadPhase*. La utilidad de este algoritmo está explicada más extensamente en el anexo II.
- Algoritmo de detección de colisiones (*Collision Dispatcher)*: Éste es el algoritmo que, de los pares que obtiene por parte de la etapa de *BroadPhase,* calcula exactamente aquellos pares entre los que se está produciendo una colisión.
- De cara a resolver las colisiones, necesitamos seleccionar un algoritmo de *Solver.*

La clase pone a disposición del que la use un serie de métodos que son los que se encargarán de añadir los objetos al mundo físico. Los métodos, aunque no todos utilizados en el producto final del proyecto, son los siguientes.

- addRigidBody: Añade un objeto rígido genérico, especificándole todos los parámetros que son necesarios. Este método es principalmente usado por el resto de los que añaden cuerpos rígidos (suelo, cubo y volúmenes envolventes del personaje principal).

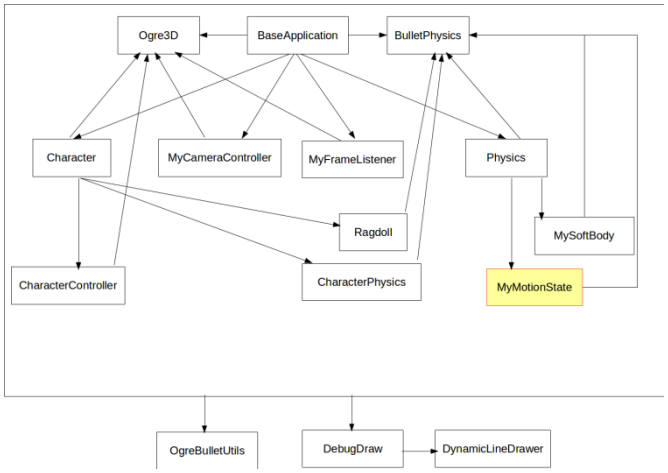
- addStaticPlane: Añade un plano. Recibe el nodo de Ogre3D al que se quiere vincular el cuerpo rígido.
- addCube: Añade un cubo. Recibe el nodo de Ogre3D al que se quiere vincular el cuerpo rígido.
- AddImpulsedCube: Lanza un cubo en la dirección que recibe. También recibe el nodo de Ogre3D al que se quiere vincular el cuerpo rígido.
- AddSoftFromEntity: Añade un cuerpo blando que tiene la forma de la malla que se proporciona. Además, recibe el nodo de Ogre3D al que se quiere vincular el cuerpo blando.
- addSoftSphere: Añade una esfera blanda con presión interna. Recibe el nodo de Ogre3D al que se quiere vincular el cuerpo blando.

Como vemos, todos los métodos requieren un objeto nodo de Ogre3D. Esto se debe a que en cada *frame* el motor de físicas debe actualizar también la posición de estos nodos para que se el motor de render los muestre por pantalla.

La forma en la que se ha implementado en este proyecto la actualización de los nodos de Ogre3D pasa por crear una clase que hereda de *btDefaultMotionState* de BulletPhysics. Esta clase se llama *MyMotionState* y será explicada más adelante.

Sin embargo, actualizar con *Motion States* sólo funciona para los objetos rígidos, por lo tanto hay que buscar otra solución para los objetos blandos ya que stos necesitan, además de cambiar la posición y la rotación, modificar la malla del objeto 3D para que muestre las deformaciones que ha sufrido. Esta implementación está hecha en la clase *MySoftBody* que encapsula el objeto blando para poder tratarlo fácilmente y de manera que el motor de físicas sólo tenga que llamar al método *updateOgreMesh()* para que su comportamiento sea el esperado.

3.7 MyMotionState



Como se ha indicado anteriormente, ésta es la forma de que los nodos visibles de Ogre3D se vean actualizados en función de lo que les ocurra cuando el mundo físico realiza un paso temporal en la simulación.

MyMotionState hereda de una clase de BulletPhysics que se llama *btDefaultMotionState.* Los objetos de esta clase se instancian a la vez que los cuerpos rígidos para que, en cada paso de simulación, proporcionen la posición y la rotación a los cuerpos rígidos.

Sin embargo, es posible extender dicha funcionalidad, de manera que, además de actualizar su posición y rotación en el mundo físico, se actualicen también la posición y la rotación del nodo de Ogre3D al que están vinculados.

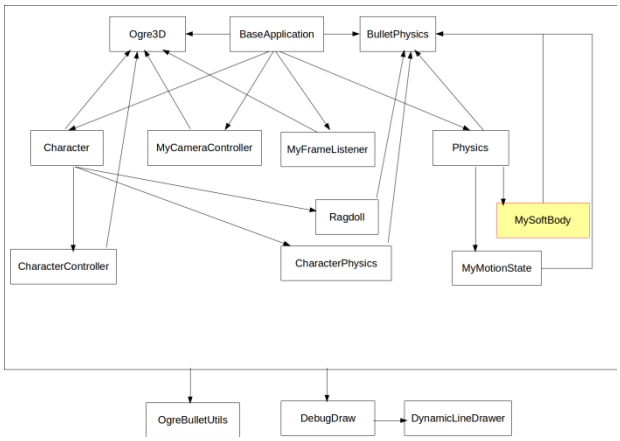
Esto se hace sobrescribiendo el método al que sabemos que BulletPhysics llama cuando quiere actualizar un objeto, *setWorldTransform*, de manera que se y añade dicha funcionalidad. Se puede ver un ejemplo en el código siguiente.

```
void MyMotionState::setWorldTransform(const btTransform & worldTrans)
{
if (mVisibleObj == NULL)
    return;
mTransform = worldTrans;
btTransform transform = mTransform * mCOM;
btQuaternion rot = transform.getRotation();
btVector3 pos = transform.getOrigin();
mVisibleObj->setOrientation(rot.w(), rot.x(), rot.y(), rot.z());
mVisibleObj->setPosition(pos.x(), pos.y(), pos.z());
}
```

En el código se ve cómo se recibe la transformación y, además de actualizarla en el objeto principal que es *mTransform*, actualiza con *setOrientation()* y *setPosition()* el nodo de Ogre3D.

Utilizando de esta manera *Motion States* no es necesario más que dar un paso más en la simulación consiguiendo de paso que los elementos de la escena también se actualicen en el motor gráfico.

3.8 MySoftBody

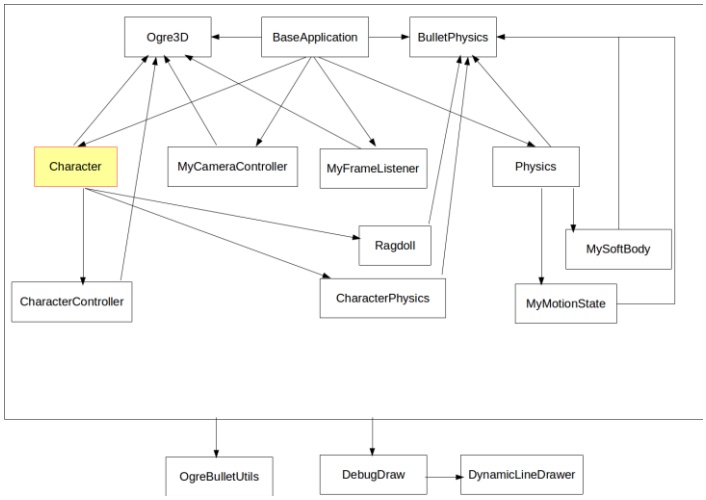


Ésta clase encapsula un objeto blando de BulletPhysics y el nodo en Ogre3D al que está vinculado.

El único método que tiene es *updateOgreMesh()* cuyo funcionamiento no es trivial y consiste en iterar sobre todos los vértices del cuerpo blando en el mundo físico y copiar su información de rotación y traslación al nodo del motor de renderización.

La transformación que sufren los vértices es relativa a la posición del cuerpo, por lo tanto, también hay que actualizar la posición y rotación global del objeto.

3.9 Character



Esta clase es la que se encarga de gestionar todo el movimiento del personaje por pantalla. Hace uso principalmente de otras dos, que son *CharacterController* y *CharacterPhysics.*

Tiene principalmente dos funciones, la primera es recibir la entrada del teclado y coordinar el funcionamiento de las dos clases anteriores para mover el personaje y la segunda es actualizar la posición y rotación del personaje en cada *frame*.

La primera función se resuelve mediante los métodos *injectKeyUp()* e *injectKeyDown().* Estos dos métodos se llaman por la clase *MyFrameListener* cuando una tecla es pulsada. Lo que hacen es, en función de la tecla que ha sido pulsada, actualizan los valores de las variables de dirección de movimiento o indican a *CharacterPhysics* y *CharacterController* que el personaje debe saltar, si puede hacerlo.

Con respecto a la segunda, la clase tiene un método llamado *updateCharacter()* que recibe el tiempo entre *frame* y *frame,* y se encarga precisamente de actualizar la posición y el estado del personaje.

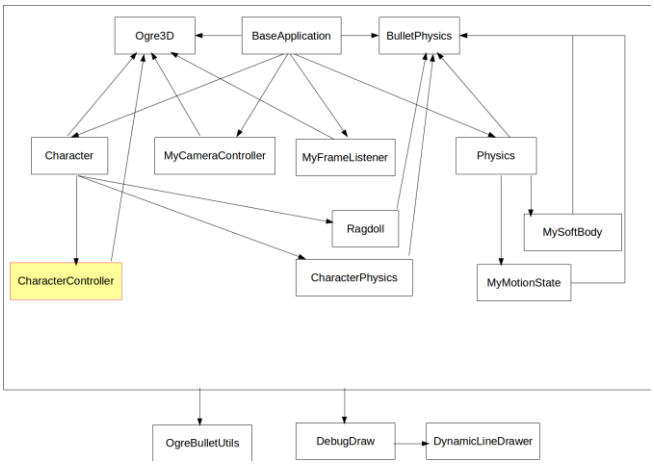
En función de la variable de dirección de movimiento, la velocidad de movimiento y la dirección a la que mira la cámara, establece la posición final del personaje en el tiempo transcurrido. Además, si el motor de físicas ha actualizado la posición del objeto que simula al personaje y éste ya no se encuentra en el mismo sitio, lo mueve para que concuerde.

Cuando el personaje está realizando algún tipo de movimiento, llama a la clase *CharacterController* para activar la animación que corresponda en ese momento.

Con la clase *Character*, obtenemos el movimiento del personaje controlado por entrada y salida de teclado y los movimientos físicos que le afectan, que pueden ser caídas y saltos.

En el proyecto se ha considerado que el personaje no reaccione ante las colisiones para evitar que el personaje saliese despedido cuando choca contra algo.

3.10 CharacterController



Esta clase utilizada por *Character* es la que se encarga de gestionar el apartado visual del personaje. Es decir, se encarga de crear el nodo de Ogre3D, asignarle las texturas, etc.

El modelo, texturizado y animaciones del personaje se han descargado de la web de Ogre3D[5] y se han modificado mínimamente para que se adaptasen mejor a este proyecto.

El principal cometido de esta clase es el de crear la estructura para activar y desactivar las animaciones del personaje cuando se requiera.

Para ello implementa un conjunto de métodos que se utilizarán para que el personaje comience o termine una animación. Los métodos son los siguientes:

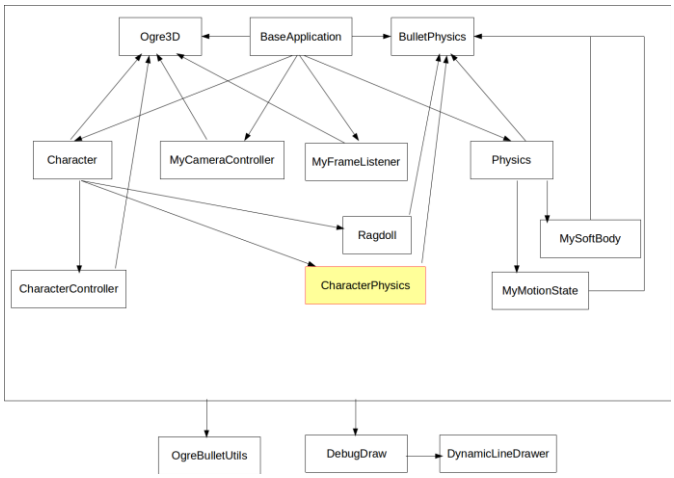
- *animRunStart() y animRunEnd():* El personaje activa y desactiva la animación de correr hacia cualquier dirección.
- *animJumpStart(), animJumpEnd()* y *animJumpLoop()*: Se encargarán de empezar y terminar la animación de salto, además, se tiene la animación en la que el personaje está en el aire.
- *animSliceStart*() y *animSliceEnd*(): Son las animaciones de golpear del personaje, que se activan cuando la tecla "1" es pulsada.

El otro método principal que esta clase implementa es *updateAnimations()*. Se llama cada vez que una animación tiene que actualizarase, es decir, cada *frame*. Y actualiza la animación en función del tiempo que haya pasado y la termina si es

[5] Sitio web del modelo: http://www.ogre3d.org/tikiwiki/Sinbad+Model

necesario; además, en caso de tratar con la animación de salto, cambia a *animJumpLoop* si *animJumpStart* ya ha finalizado.

3.11 CharacterPhysics



Esta clase utilizada por *Character* se encargará de simular el comportamiento físico del personaje.

El personaje se modela de tres formas diferente. Una, la malla más precisa, para renderizar. Otra, para el movimiento y la caída, para lo que el personaje se engloba una cápsula aproximadamente de su tamaño. Y la tercera para las colisiones con objetos, en este caso, el personaje se modela con más precisión mediante el uso de varias cápsulas.

En la figura 8, las líneas blancas se corresponden con el segundo modelado y las verdes con el tercero.

Figura 8: Personaje principal con modo debug activo

CharacterPhysics es la clase que utiliza la segunda aproximación geométrica.

Simula el personaje no como un cuerpo rígido normal, sino como un *btPairCachingGhostObject*, es decir, un cuerpo fantasma.

Estos cuerpos no reaccionan ante las colisiones y tampoco las provocan de la manera habitual, por lo tanto no afectarán al mundo, pero utilizando otro tipo de algoritmos, permiten calcular qué objetos están colisionando con él.

Para simplificar la programación del control del personaje, la clase *CharacterPhysics* hereda de la clase *btCharacterControllerInterface* de BulletPhysics, que es una interfaz que establece un orden de llamadas a métodos que recibirá cualquier implementación de la misma cuando se añada a un mundo.

La asignación de *CharacterPhysics* como controlador de personaje se ha hecho en la clase principal de la aplicación, *BaseApplication* en la siguiente línea de código:

Physics->**getDynamicsWorld**()->**addAction**(mCharacterController);

El método al que Bullet Physics llama en cada *frame* para actualizar *CharacterPhysics* es *updateAction(),* donde se describe y calcula todo lo que va a pasar en un *frame.* En este caso, por simplicidad de código, se ha subdividido este método en los siguientes métodos:

1. preStep():

   En este método se comprueba el estado actual del personaje en función de lo que el motor de físicas le haya comunicado que ha ocurrido.

2. playerStep()

   1. Llamada al método setRBForceImpulseBasedOnCollision():

      Una vez detectado que hay colisiones, lo que se hace en este método es realizar la reacción física asociada. Los cuerpos que han colisionado con el personaje se obtienen haciendo uso de una estructura de datos que ha sido rellenada por BulletPhysics.

   2. Se actualiza la velocidad de salto si la clase *Character* comunica que el jugador ha pulsado la barra espaciadora o si lo se ha hecho en *frames* anteriores y el personaje aún está en mitad de un salto.

   3. Llamada al método *stepUp():*

      Mediante la llamada a un *convexSweepTest* se comprueba si hay algo justo encima del personaje que le impida saltar. Si lo hay, no se cambia la posición del nodo verticalmente.

      Además, se comprueba que el salto no haya terminado en función de la fuerza del salto y el tiempo que el personaje lleva saltando. Si ha terminado, se cambia de signo para que comience la caída.

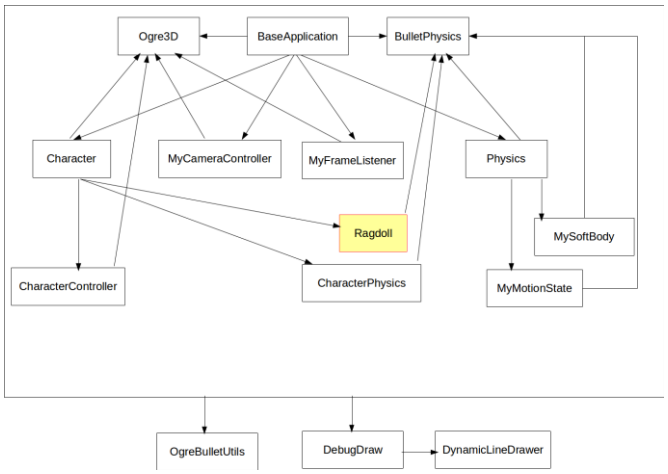
   4. Llamada al método *stepForwardAndStrafe()* si y solo si la clase *Character()* ha comunicado que el jugador ha pulsado una de las teclas de movimiento y por lo tanto se han actualizado la velocidad y la dirección del mismo:

Mediante la llamada a un *convexSweepTest* se comprueba si hay algo justo delante en la dirección del personaje en la que el personaje tiene que moverse y si hay algo, no se modifica su posición.

   5. Se actualiza la velocidad de caída si, por un salto o por un movimiento, el personaje está cayendo.
   6. Llamada al método *stepDown():*

Mediante la llamada a un *convexSweepTest* se comprueba si hay algo justo debajo del personaje que indique que ha tocado suelo. Si lo hay, no se cambia la posición del nodo verticalmente y se le comunica a la clase *Character* para que se actualicen las animaciones; se pone a cero la velocidad vertical.

3.12 Ragdoll



Esta clase utilizada por *CharacterPhysics* es la que se encarga del modelado geométrico del tercer tipo para poder calcular el comportamiento físico del personaje en relación con la colisión con los objetos blandos, de manera que estos se deformen de forma precisa. El resultado de esta clase se puede ver gráficamente en la figura 8, son las líneas verdes.

Para entender el funcionamiento de esta clase es fundamental conocer la existencia de los huesos en los modelos 3D. Cuando se crea un personaje en 3D el primer paso es hacer la forma del mismo, modelarlo moviendo vértices, etc. A continuación, se texturiza para que parezca real. Luego, de cara a la animación, que sería el último paso, se realiza el llamado proceso de *rigging*. Éste consiste en la creación de un sistema de esqueleto no renderizable, por debajo de la malla del personaje que ayudará a los animadores a mover las partes de la malla del cuerpo de manera realista.

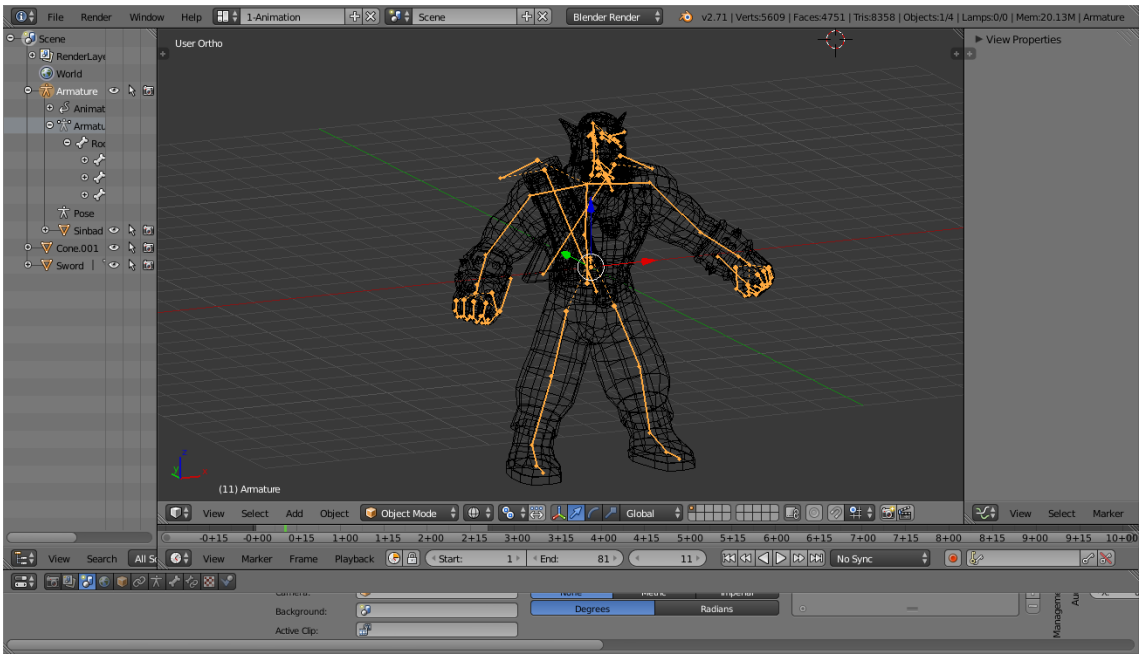


Figura 9: *Rigging* del personaje principal

En la figura 9 se puede ver una captura de pantalla de Blender, el software que se ha utilizado para modificar el personaje principal, donde se pueden observar estos huesos.

Ogre3D pone a disposición del programador un sistema para acceder a la posición de estos huesos.

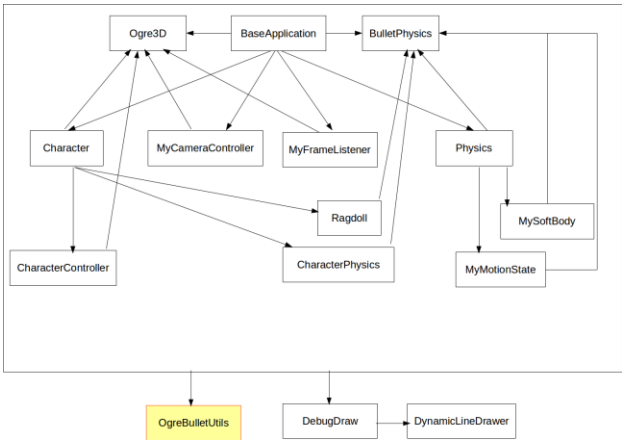
La clase *Ragdoll* lo que hace recorrer un subconjunto de los huesos del esqueleto que se ha considerado que modelarían con suficiente precisión el cuerpo físico del personaje y crear objetos rígidos de masa 0 (cápsulas), para que no se vean afectados por la gravedad.

El personaje principal está modelado, en total, por diecisiete objetos rígidos. Uno en la cabeza, dos en el torso, dos en cada brazo, uno en cada mano, dos en cada pierna, uno en cada pie y dos en las vainas de la espalda.

De esta manera, las cápsulas producen movimiento en otros objetos, en concreto, deformaciones en los cuerpos blandos.

La clase, además, se encarga de, en cada *frame* modificar la posición y rotación de estas cápsulas para adaptarlas a la que actualmente tienen los huesos. De esta manera, con las animaciones de las que ya se ha hablado, también movemos el exoesqueleto de frprmado por las cápsulas.

3.13 OgreBulletUtils



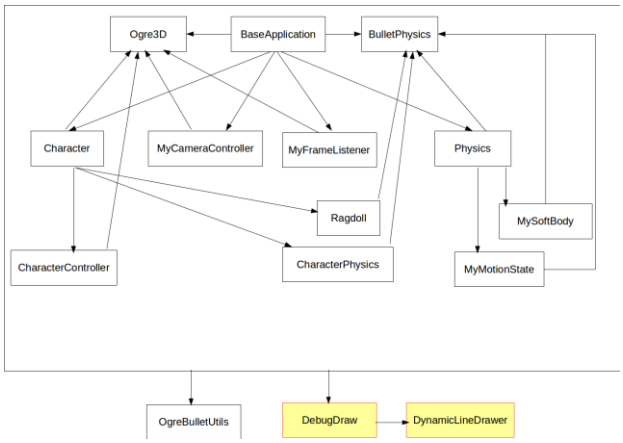
Ésta es una de las dos clases auxiliares que tiene el proyecto.

En concreto, esta clase implementa una serie de métodos que hacen más fácil la comunicación entre BulletPhysics y Ogre3D.

Principalmente se encarga de la conversión de unidades de Ogre3D a BulletPhysics y viceversa. En concreto, de los vectores y los cuaterniones.

Además, tiene implementados otros métodos de objetos complejos de BulletPhysics desde una malla de Ogre3D.

3.14 DebugDraw y DynamicLineDrawer



Ésta es la otra clase auxiliar presente en el sistema. Es la encargada de dibujar las líneas que vemos en la figura 5 y que, como veremos en las capturas del final del proyecto, están presentes en todos los objetos. Muestra por pantalla, renderizado gráficamente, el mundo físico de BulletPhysics.

En concreto, DebugDraw hereda del sistema de debug de BulletPhysics y DynamicLineDrawer hereda de un objeto renderizable de Ogre3D. De esta manera, en conjunción, es posible renderizar en Ogre3D los objetos de BulletPhysics.

3.15 Análisis de eficiencia

Como ya hemos explicado, Ogre3D ofrece acceso al programador a su bucle de renderización mediante los eventos producidos en la clase *MyFrameListener*, de esta manera, ha resultado muy simple calcular el tiempo que le cuesta a la aplicación el cálculo de cada *frame,* así como el cálculo de cada una de sus partes.

Los datos medios para un *frame* de cada una de las partes, son:

Tiempo total (100%): 6.307 ms.
Tiempo de captura de periféricos de entrada (0,06%): 0.004 ms.
Tiempo de actualización de personaje y animaciones (1%): 0,057 ms.
Tiempo de cálculo de físicas (21%): 1.347 ms.
Tiempo de renderización gráfica (77,1%): 4.863 ms.

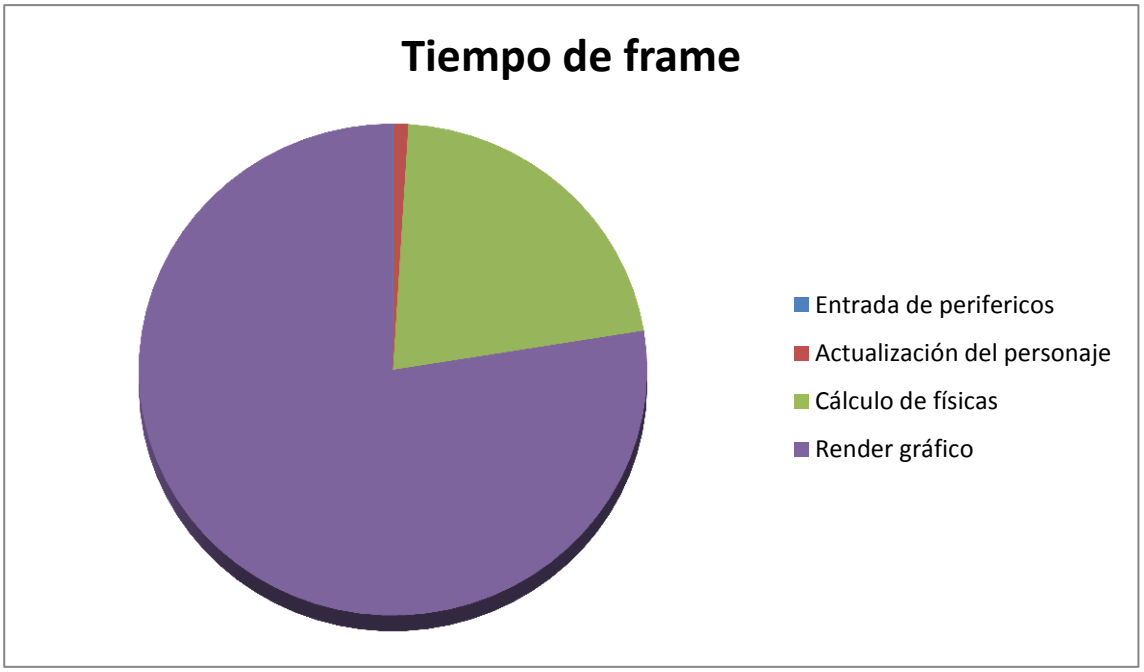


Figura 10: Gráfico del tiempo medio de *frame*

Estas pruebas han sido ejecutadas sobre una tarjeta gráfica integrada de un procesador Intel modelo i7-4770K

El chipset gráfico tiene las siguientes especificaciones:

Gráficos de procesador	Intel® HD Graphics 4600
Frecuencia base de los gráficos	350 MHz
Frecuencia dinámica máxima de los gráficos	1.25 GHz
Memoria máxima de vídeo de los gráficos	1.7 GB

4. Resultados (salidas gráficas)

Figura 11: Interacción objetos rígidos 1.

Figura 12: Interacción objetos rígidos 2.

Figura 13: Deformación cuerpo blando 1.

Figura 14: Deformación cuerpo blando 2.

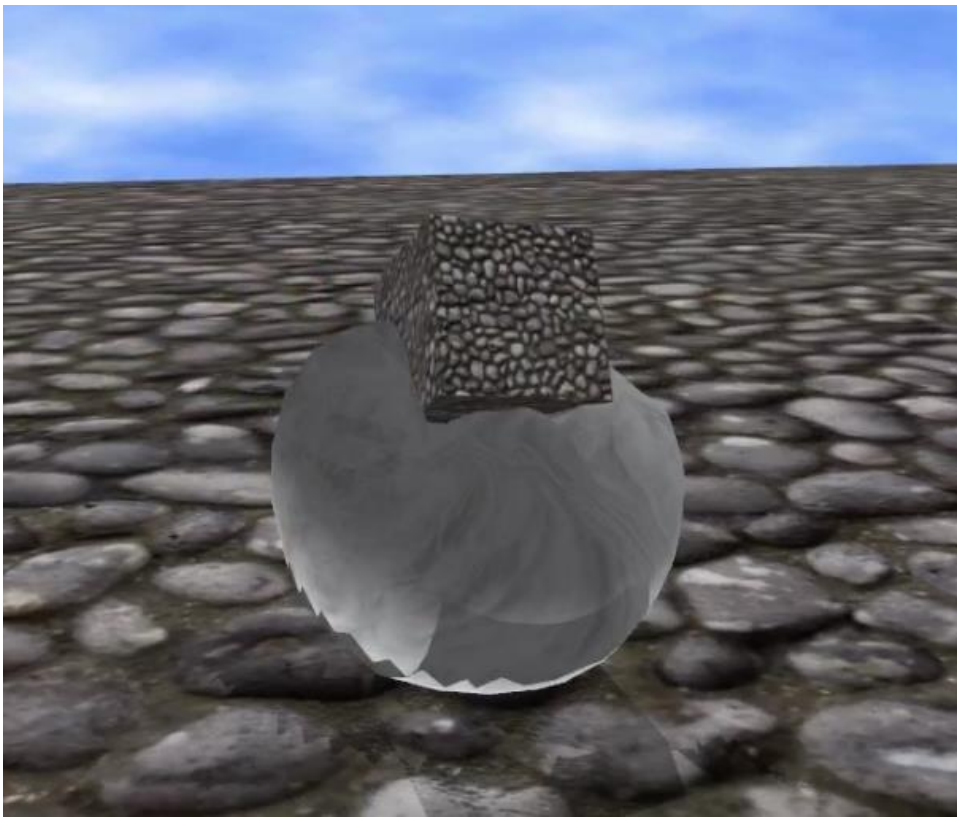


Figura 15: Deformación cuerpo blando 3.

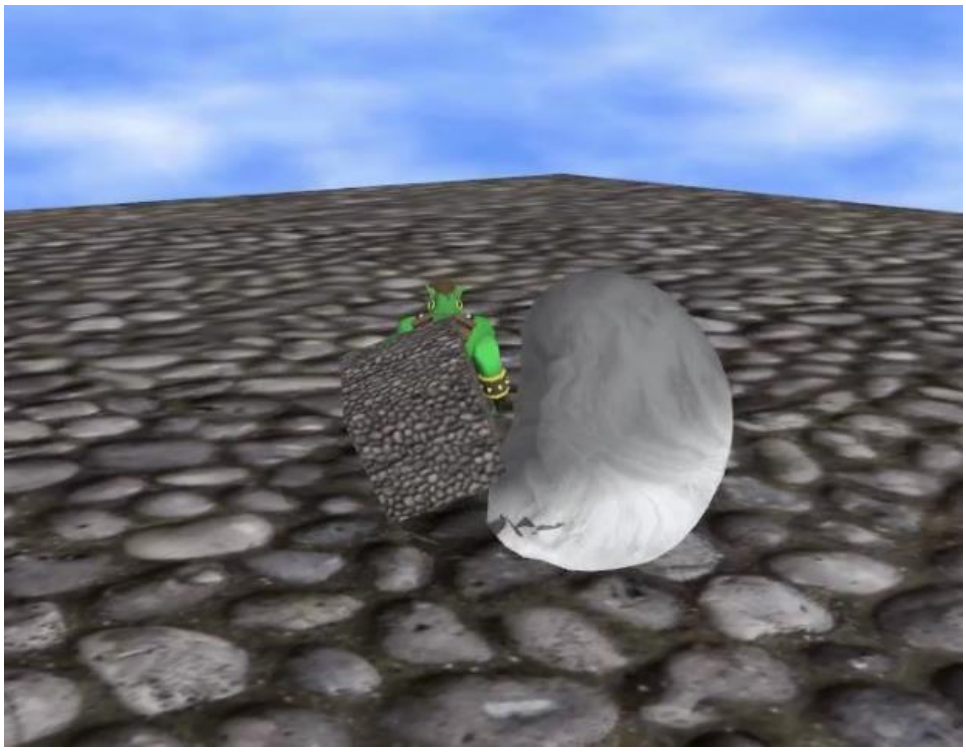


Figura 16: Deformación cuerpo blando 4.

5. Conclusiones y trabajo futuro

El objetivo principal de este Trabajo Fin de Grado era simular cuerpos rígidos y blandos sometidos a la acción de fuerzas externas y gravedad. (Ver apartado 2).

Para ello se ha hecho una aplicación completa formada por trece clases diseñadas por el autor. (Ver apartado 3).

Se han utilizado dos librerías de código libre. Ogre3D y BulletPhysics.

La primera de ellas ofrece un sistema capaz de renderizar objetos 3D en Tiempo Real, así como de configurar todos los aspectos que tienen que ver con el motor gráfico de un videojuego.

La segunda de ellas ofrece un motor capaz de simular físicas en tiempo real, así como de configurar todos los aspectos que tienen que ver con el motor de físicas de un videojuego.

Como se ha demostrado de forma visual mediante imágenes estáticas en el apartado 4, y se verá mediante vídeo o ejecución en tiempo real el día de la defensa, el objetivo propuesto ha sido plenamente cumplido.

Las líneas futuras de trabajo pasar por programar una librería genérica de uso de BulletPhysics sobre Ogre3D que se despegue de la programación de este ejemplo en concreto. Para ello, sirve la misma estructura de clases que se ha planteado, únicamente es necesaria la extensión de las mismas.

6. Diagrama de tiempos

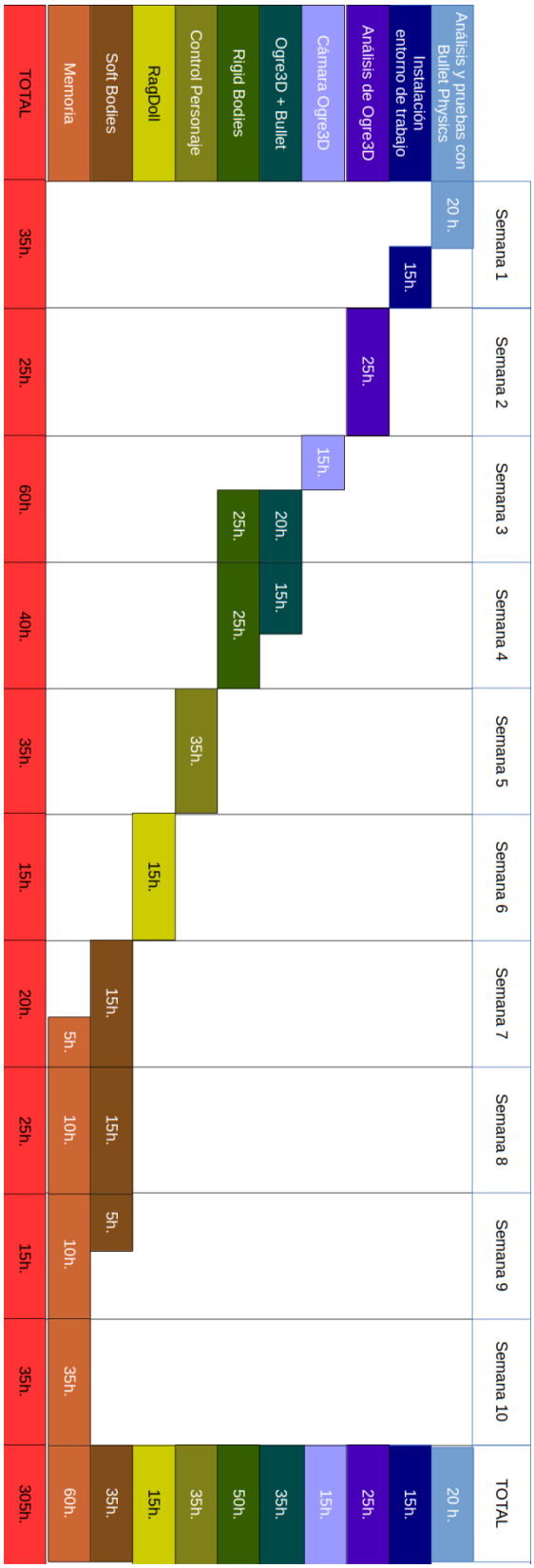


Figura 17: Diagrama de tiempos.

7. Bibliografía

1 - *Sitio web de Ogre3D*. http://www.ogre3d.org/

2 - *Wiki de Ogre3D*. Obtenido de http://www.ogre3d.org/tikiwiki/tiki-index.php

3 - *Foro de Ogre3D*. http://www.ogre3d.org/forums/

4 - *API de Ogre3D*. http://www.ogre3d.org/docs/api/1.9/

5 - *Sitio web de BulletPhysics*. Obtenido de http://www.bulletphysics.org/

6 - *Wiki de BulletPhysics*. http://bulletphysics.org/mediawiki-1.5.8/index.php/Main_Page

7 - BulletPhysics. *Bullet 2.82 Physics SDK Manual. Apéndice III*

8 - *Foro de BulletPhysics*. http://www.bulletphysics.org/Bullet/phpBB3/

9-Nvidia CG tutorials. http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html

10 - Wikipedia. https://www.wikipedia.org/