Ejercicios guiados de programación en CHAI3D para renderizado háptico

Dispositivos Hápticos para Tacto Virtual

Laura Raya González
Carlos Garre del Olmo

1. Presentación de la práctica

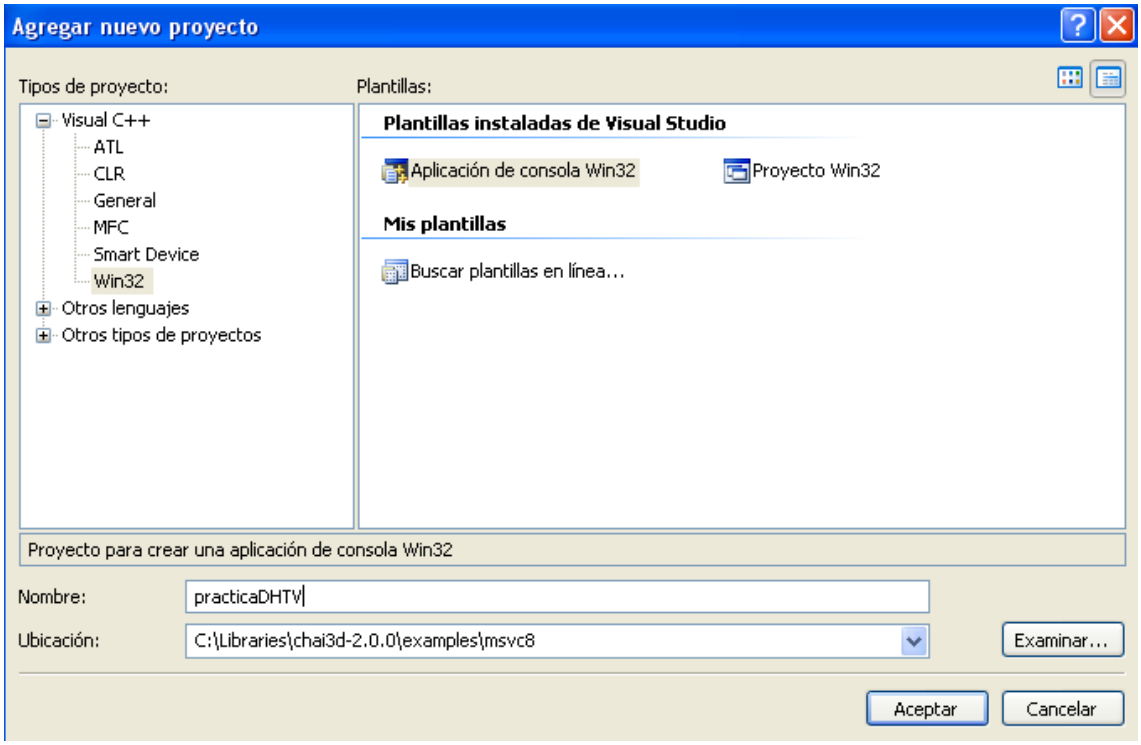
En esta práctica el alumno tendrá que seguir los pasos explicados durante el enunciado con el objetivo de desarrollar los ejercicios propuestos. Cada uno de los ejercicios pretende explicar de manera detallada algunos de los conceptos más importantes en el manejo de los dispositivos hápticos. El alumno deberá ir poniendo el código explicado en un fichero cpp (añadir lo que sea necesario) y hacerlo ejecutar con el fin de que el resultado de su ejecución sea similar a las imágenes presentadas en esta práctica.

La presente práctica podrá entregarse por parejas. Será imprescindible entregar vía Campus Virtual un fichero .pdf más cuatro ficheros .cpp con cada uno de los ejercicios.

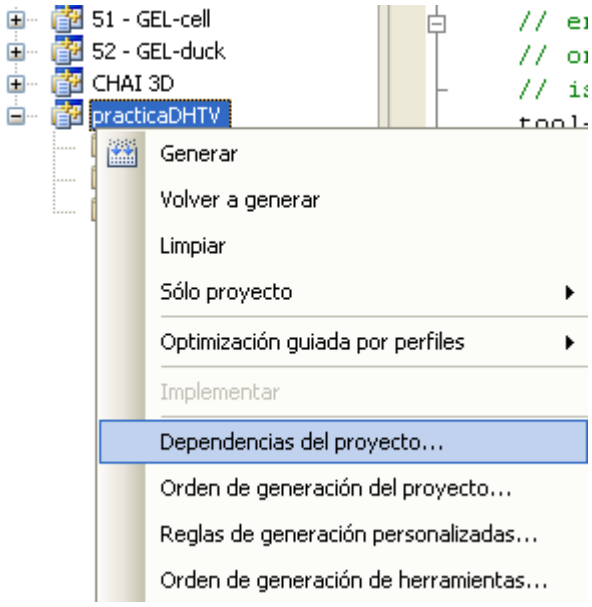
2. Creación del proyecto

La forma más fácil de crear el proyecto para la práctica es reutilizar la solución ya existente en los ejemplos de código de CHAI3D (chai3d.sln).
Una vez abierta la solución:

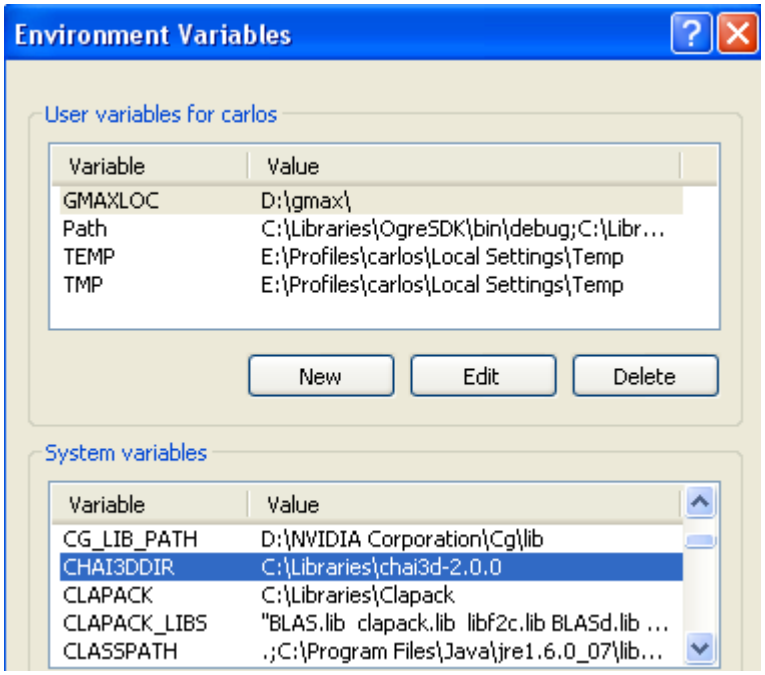
1. Añadir nuevo proyecto de tipo Win32 Console.



3. Añadir al nuevo proyecto dependencia del proyecto CHAI3D.



4. Crear la variable de entorno CHAI3DDIR, apuntando a la ruta en la que tenemos los ficheros de CHAI3D (en Windows, Mi Pc → Propiedades → Opciones avanzadas → Variables de Entorno).



5. Configurar propiedades del proyecto. En caso de duda, comparar con las propiedades de los proyectos de ejemplo de la librería:

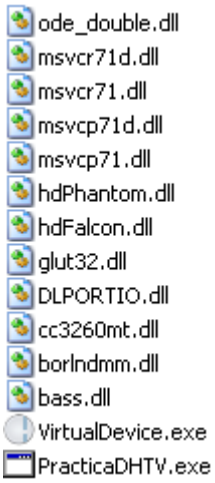
C++:

- General→Additional include directories: $(CHAI3DDIR)/src;$(CHAI3DDIR)/external/OpenGL/msvc
- Preprocessor→Preprocessor definitions: WIN32;_DEBUG;_MSVC

Linker:

- General→Additional Library directories:
$(CHAI3DDIR)/lib/msvc8;$(CHAI3DDIR)/external/OpenGL/msvc
- Input→Additional dependencies:
winmm.lib opengl32.lib chai3d-debug.lib glu32.lib odbc32.lib odbccp32.lib
Nota: en modo *Release*, utilizar chai3d-release.lib.

6. **¡Importante!** Copiar al directorio de trabajo las DLL y, en caso de que no tengamos dispositivo háptico, también el fichero VirtualDevice.exe.



El directorio donde tenemos que copiar todo esto es el que tengamos especificado en: Depuración→Directorio de trabajo.

Ejercicio 1: El "Hola Mundo" háptico

Podemos partir del ejemplo **Devices** de CHAI3D, que es el más sencillo de los que incluye la librería.

Objetivo:

Visualizar una pantalla negra en la que haya una pequeña esfera que se mueva siguiendo al dispositivo háptico. El objetivo es simplificar al máximo el código del ejemplo (devices) para que queden el menor número de líneas de código.

Desarrollo del ejercicio:

Partimos del código del ejemplo y nos fijamos en las partes principales que lo componen. Comentamos aquí sólo lo más importante, que nos servirá de base para crear nuestro ejemplo con el menor número de líneas de código posible.

Variables globales:

En primer lugar tenemos las relacionadas con la creación del grafo de escena:

```
cWorld* world;
cCamera* camera;
cLight* light;
```

Y a continuación las relacionadas con el control del dispositivo háptico:

```
cHapticDeviceHandler* handler;
cGenericHapticDevice* hapticDevice;
```

En nuestro ejercicio no tendremos un vector de dispositivos, sino un único dispositivo hapticDevice.

El handler es único y permite tener acceso a todos los dispositivos conectados al sistema. Utilizamos el handler una única vez para obtener un puntero al primer dispositivo conectado, y ya no volveremos a utilizarlo.

Por último, el único objeto de nuestra escena será una esfera, que es el cursor que sigue los movimientos del dispositivo:

```
cShapeSphere* cursor;
```

Funciones:

Hay tres funciones principales a implementar:

- **main**: crea el grafo de escena, configura el dispositivo háptico y lanza los dos threads (gráfico y háptico).
- **updateGraphics**: código del thread gráfico.
- **updateHaptics**: código del thread háptico.

Además, podemos definir las clásicas funciones keyboard, resize, mouse, etc, típicas en cualquier aplicación gráfica. Es también importante que definamos una función **close** en la que finalicemos la conexión con el dispositivo y cerremos los threads.

Función main:

1. Crear grafo de escena

Dejamos al alumno que analice este código por su cuenta.

2. Crear herramienta háptica.

```
handler = new cHapticDeviceHandler();
handler->getDevice(hapticDevice, 0);
```

Hemos solicitado al handler un puntero para acceder al primer dispositivo conectado (dispositivo 0). A partir de ahora, podemos controlar el dispositivo a través de la variable global hapticDevice.

```
hapticDevice->open();
hapticDevice->initialize();
```

Con estas dos líneas abrimos la conexión con el dispositivo y lo calibramos (funciones dependientes de cada dispositivo).

```
cursor = new cShapeSphere(0.01);
world->addChild(cursor);
```

Por último, creamos la esfera que nos servirá como cursor que seguirá al dispositivo.

3. Inicialización de GLUT

Como tenemos grafo de escena, la inicialización consiste en poco más que crear la ventana, definir las funciones keyboard, reshape, etc… En este punto, lo más importante es definir qué función va a ejecutar nuestro bucle gráfico:

```
glutDisplayFunc(updateGraphics);
```

Como el bucle gráfico se implementa usando Glut, el código de updateGraphics no tendrá que incluir el propio bucle, sino sólo una iteración.

4. Comenzar simulación:

```
simulationRunning = true;
```

Esta variable indica que la simulación está activa. En el momento en que el usuario cierre la aplicación (típicamente con una tecla), esta variable se pondrá a false, señalizando así al bucle háptico que debe detenerse. Es, por tanto, una sencilla forma de comunicación entre el thread gráfico y el háptico.

```
cThread* hapticsThread = new cThread();
hapticsThread->set(updateHaptics, CHAI_THREAD_PRIORITY_HAPTICS);
```

Por último, lanzamos el bucle gráfico simplemente llamando al loop de Glut:

```
glutMainLoop();
```

Función updateGraphics:

En este ejemplo no hay ningún tipo de interacción con la escena, por lo que sólo tenemos que pedir a la cámara que renderice la escena:

```
camera->renderView(displayW, displayH);
```

Y preparar la próxima iteración del bucle gráfico como se hace siempre con Glut:

```
glutSwapBuffers();
if (simulationRunning)
   glutPostRedisplay();
```

Función updateHaptics:

En general, un bucle háptico siempre consiste en lo siguiente:

1. Leer posición/orientación del dispositivo.
2. Calcular la fuerza/par en base a:
    a. Posición/orientación leída del dispositivo.
    b. Última configuración de contacto calculada en el bucle gráfico.
3. Enviar la fuerza/par calculada al dispositivo.

La condición de salida del bucle háptico va a ser que desde el bucle gráfico le señalicen que queremos terminar la simulación. Esto sucede típicamente porque el usuario ha pulsado una tecla para cerrar la aplicación. S importante señalizar esto mediante la variable simulationRunning.

```
while(simulationRunning)
{
    // 1. Read position of haptic device
    cVector3d newPosition;
    hapticDevice->getPosition(newPosition);

    // Update position of cursor
    cursor->setPos(newPosition);

    // 2. Compute a reaction force
    cVector3d newForce (0,0,0);

    // 3. Send computed force to haptic device
    hapticDevice->setForce(newForce);
 }

// exit haptics thread
simulationFinished = true;
```

En nuestro caso tan sencillo, incluimos en el propio bucle háptico la actualización de posición del cursor, cuya nueva posición será exactamente la leída desde el dispositivo.

La variable simulationFinished sirve para señalizar al bucle gráfico que el bucle háptico ha terminado. Podemos resumir la señalización entre los threads de la siguiente manera:

1. Se inicia bucle gráfico. (simulationRunning = true).
2. Se inicia bucle háptico (simulationFinished = false).
3. …Simulación…
4. El usuario pulsa la tecla para salir de la aplicación (simulationRunning = false).
5. El bucle háptico termina su última iteración (simulationFinished = false).
6. Ya se puede terminar el programa.

Función close:

La podemos activar por ejemplo, desde un evento de teclado.

```
    simulationRunning = false;
    while (!simulationFinished) { cSleepMs(100); }
    hapticDevice->close();
```

La primera línea señaliza al bucle háptico que debe finalizar. La segunda línea es una espera activa a que termine el bucle háptico. Por último, podemos enviar los comandos al dispositivo para finalizar la comunicación.

PREGUNTAS PUNTO 1

1.- Del trozo de código para lanzar el bucle háptico,
¿Qué significa la opción CHAI_THREAD_PRIORITY_HAPTICS?

2.- Del trozo de código donde se lee la posición del probe:

¿En qué tipo de coordinadas estará la variable `newPosition`?

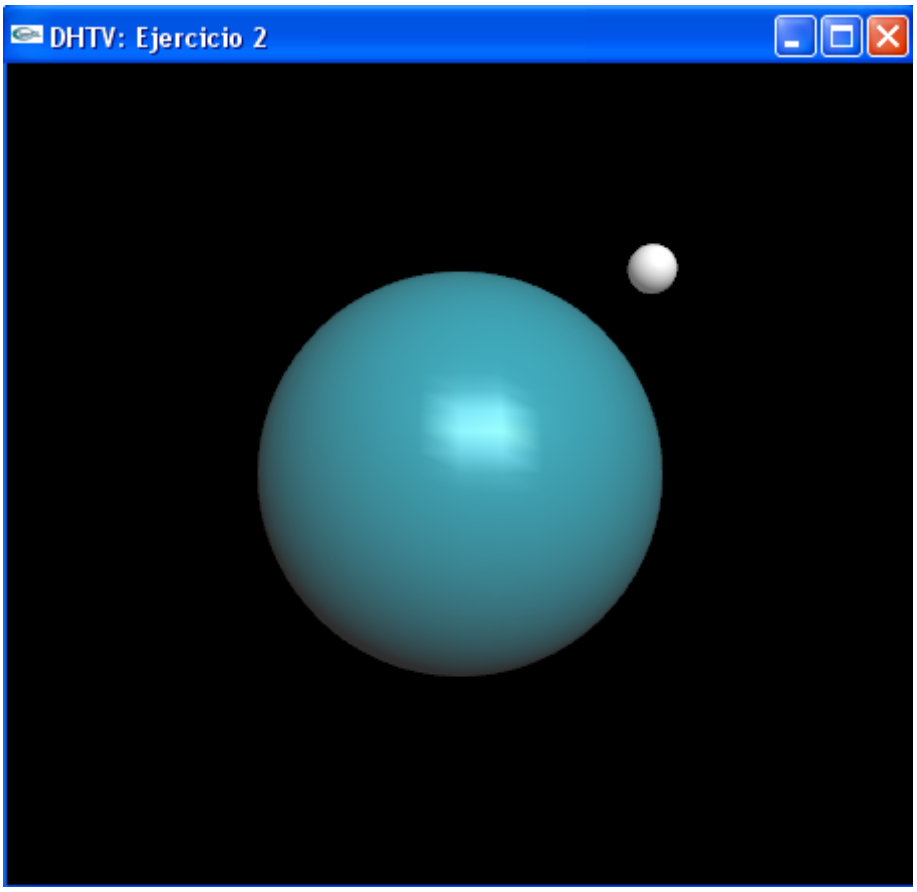
3.- Haz varias capturas del resultado de ejecutar tu código. Prueba a cambiar algunos parámetros (como el radio) sobre él.

Ejercicio 2: Tocar una esfera

Partimos del ejercicio anterior, para ir añadiendo un poco más de código.

Objetivo:

Visualizar una pantalla negra en la que hay una esfera grande que podemos tocar con nuestro cursor.



Desarrollo del ejercicio:

Al ser nuestro objeto una esfera utilizaremos las funciones predefinidas de CHAI3D para crear esferas, lo que nos permite calcular la fuerza utilizando campos potenciales (potential fields). Por ello, por el momento, no necesitamos implementar un virtual proxy.

Aún así, utilizaremos la clase cGeneric3dofPointer, que implementa automáticamente el seguimiento del dispositivo y el cálculo de fuerzas.

Variables globales:

Cambiaremos el cursor por:

```
cGeneric3dofPointer* tool;
```

Y añadiremos el objeto a tocar:

```
cShapeSphere* object;
```

Función main:

Cambiamos la parte que inicializa el dispositivo háptico:

```
handler = new cHapticDeviceHandler();
handler->getDevice(hapticDevice, 0);

// create a 3D tool and add it to the world
tool = new cGeneric3dofPointer(world);
world->addChild(tool);

// connect the haptic device to the tool
tool->setHapticDevice(hapticDevice);

// initialize tool by connecting to haptic device
tool->start();
```

Con esto hemos asociado el tool al dispositivo, para que lo siga automáticamente. El comando start internamente lanzará los comandos necesarios para conectar con el dispositivo, por lo que ya no es necesario llamar a open e initialize.

En el caso de que queramos escalar las posiciones leídas desde el dispositivo a un movimiento más amplio o más reducido del tool, podemos utilizar el método setWorkspaceRadius. Por defecto, su valor es 1.0, y cuanto menor lo hagamos menos espacio abarcará el movimiento del tool.

```
tool->setWorkspaceRadius(1.0);
```

Podemos, también, cambiar la apariencia del propio tool, configurando el radio de la esfera que lo representa:

```
tool->setRadius(0.01);
```

Si vamos a añadir objetos que podamos tocar, es importante que podamos definir la rigidez de cada objeto, para poder sentir objetos con diferentes propiedades hápticas. El problema es que necesitamos saber cuál es el rango de rigideces que podemos configurar.

La rigidez se expresa (en sistema internacional) en Newtons/metro, por lo que va a depender: por una parte de cuántos Newtons podamos llegar a generar con nuestro dispositivo háptico y, por otra parte, de en qué escala esté nuestra escena (metros, milímetros…).

Por tanto, para saber la máxima rigidez que podemos asignar a un material:

```
double workspaceScaleFactor = tool->getWorkspaceScaleFactor();
cHapticDeviceInfo info;
info = hapticDevice->getSpecifications();
```

```
double maxStiffness = info.m_maxForceStiffness/workspaceScaleFactor;
```

Con la primera línea obtenemos el factor de escala (con respecto a metros) de nuestra escena. A continuación, leemos de las especificaciones del dispositivo cuál es la máxima fuerza que puede proporcionar. Por último, la máxima rigidez se obtiene dividiendo estos dos valores.

De ahora en adelante, siempre que queramos asignar una rigidez a un objeto, lo haremos entre 0 y maxStiffness.

Vamos a crear ahora el objeto a tocar:

```
object = new cShapeSphere(0.1);
world->addChild(object);

// set the position of the object at the center of the world
object->setPos(0.0, 0.0, 0.0);

// set the material stiffness to 100% of the maximum
object->m_material.setStiffness(1.0 * maxStiffness);
```

Este objeto está ahora incluido en la escena, pero no se puede tocar hápticamente. Para ello tenemos que asociarle un efecto háptico que, en este caso, será un efecto de superficie. Es decir, asociamos a su superficie el efecto de poder ser tocada hápticamente:

```
cEffectSurface* newEffect = new cEffectSurface(object);
object->addEffect(newEffect);
```

Aunque el efecto cEffectSurface sea el más habitual, no siempre queremos que la interacción háptica con un objeto consista en tocar su superficie. Si, por ejemplo, definimos un objeto que simule un fluido, podemos querer que tenga un efecto háptico de viscosidad en su interior, pudiendo atravesar sin problemas su superficie.

Función updateHaptics:

```
// 1. Read device and update position of tool
tool->updatePose();

// 2. Compute interaction forces
tool->computeInteractionForces();

// 3. Send forces to device
tool->applyForces();
```

Tenemos de forma muy sencilla los tres pasos básicos de todo bucle háptico (una línea por cada paso), pero en este caso utilizando los métodos que nos proporciona la clase cGeneric3dofPointer.

Función close:
En lugar de acceder directamente al dispositivo (método close), cerraremos la comunicación a través del método stop del tool:

```
tool->stop();
```

PREGUNTAS PUNTO 2

1.- ¿Qué es un Potencial field?

2.- ¿Por qué no es necesario la implementación de un virtual Proxy en este caso?

3.- ¿En qué tipo de entornos puede ser útil tener un valor más grande en el parámetro de este comando? ¿Y más pequeño?

4.- Haz varias capturas del resultado de ejecutar tu código. Prueba a cambiar algunos parámetros (como el color de la esfera, su material, su radio, etc) y muestralo con capturas de pantalla.

Ejercicio 3: Virtual Proxy

Partimos del ejercicio anterior, para ir añadiendo sólo el mínimo código necesario.

Objetivo:

Visualizar una pantalla negra en la que hay un objeto con cualquier forma que podemos tocar con nuestro cursor.

Desarrollo del ejercicio:

En este caso nuestro objeto no será ni una esfera ni un toro, sino un objeto cargado desde fichero (por ejemplo, generado con 3DStudio). Esto implica que la fuerza ya no se calculará mediante campos potenciales, sino utilizando el clásico virtual proxy.

Configuración del proxy:

En el ejercicio anterior nuestro cursor atravesaba la esfera cuando la tocábamos, debido a que no estábamos visualizando el proxy sino el HIP (Haptic Interface Point). Si queremos ver sólo el proxy:

```
tool->m_deviceSphere->setShowEnabled(false);
```

Y, además, podemos definir su radio, igual que con el HIP:

```
tool->m_proxyPointForceModel->setProxyRadius(0.01);
```

El proxy está restringido a permanecer en la superficie de los objetos. Esto supone un problema cuando se utilizan mallas de triángulos para definir a los objetos (como es habitual), ya que los objetos son huecos y tienen dos superficies (una interna y otra externa). Cabe la posibilidad de que al iniciar nuestra simulación, el dispositivo se encuentre en una posición que haga que nuestro proxy aparezca inicialmente en el interior de un objeto. Si esto ocurriese, el proxy no podría escapar jamás, ya que estaría restringido a la superficie interna del objeto. Para evitar esto, podemos configurar el proxy para que sólo esté restringida por la superficie externa:

```
tool->m_proxyPointForceModel->
m_collisionSettings.m_checkBothSidesOfTriangles = false;
```

Carga de la malla del objeto:

Definiremos el objeto como un cMesh:

```
cMesh* object;
```

Y cargaremos su malla que, en este ejemplo, está en el archivo coche.obj. Si el alumno desea importar otro tipo de malla, podrá hacerlo sin problemas:

```
object = new cMesh(world);
world->addChild(object);
object->loadFromFile("./coche.obj");
```

Al no tener una forma básica, como una esfera, necesitamos precalcular la estructura de datos que se utilizará para la detección de colisiones (jerarquía de cajas si utilizamos AABB, o jerarquía de esferas si utilizamos Sphere Tree). En este ejemplo utilizaremos AABB (Axis Aligned Bounding Boxes):

```
object->createAABBCollisionDetector(1.01 * proxyRadius, true, false);
```

El primer argumento hace que la caja no encaje exactamente con los bordes del objeto, sino que lo cubra con un margen de tolerancia adicional. Como mínimo, este margen debe ser tanto como el radio del proxy (que en nuestro caso era 0.01), ya que el proxy en realidad siempre se simula como un punto y no una esfera. De no dejar este margen, veríamos cómo el proxy penetra siempre hasta la mitad de la esfera en los objetos. Además, dejamos un pequeño margen adicional (1.01) para evitar problemas numéricos al ajustarnos exactamente al borde entre la esfera del proxy y el objeto.

El siguiente parámetro (true) indica que, en caso de que el objeto tenga hijos en el grafo de escena, propagaremos la jerarquía de cajas también a sus hijos. El último parámetro (false) sirve para crear una lista de vecindades que no vamos a utilizar.

Por último, especificamos las características hápticas del objeto. El objeto tendrá por defecto el efecto háptico de superficie, pero añadiremos además características de fricción:

```
object->setStiffness(1.0*maxStiffness, true);
object->setFriction(0.1, 0.2, true);
```

Problemas de escala:

Es posible que el modelo que cargamos esté en unas unidades de medida que resulten demasiado grandes o demasiado pequeñas para nuestra escena. Podemos ajustar el tamaño del objeto al de la escena con el siguiente código:

```
object->computeBoundaryBox(true);

double size = cSub(object->getBoundaryMax(),
object->getBoundaryMin()).length();

object->scale(tool->getWorkspaceRadius()/size);
```

Esto lo haremos justo antes de crear la estructura de datos para la detección de colisiones (antes de createAABBCollisionDetector).

Añadir textura al objeto:

Si cargamos mallas de 3DStudio (.3ds) la textura debería cargarse automáticamente. Si utilizamos mallas en formato OBJ, simplemente tenemos que añadir este código:

```
cTexture2D* texture = new cTexture2D();
texture->loadFromFile("./textura.bmp");
object->setTexture(texture, true);

object->setUseTexture(true);
object->m_material.m_ambient.set(1.0, 1.0, 1.0);
object->m_material.m_diffuse.set(1.0, 1.0, 1.0);
object->m_material.m_specular.set(1.0, 1.0, 1.0);
```

PREGUNTAS PUNTO 3

1.- ¿Cuál es la diferencia entre los algoritmos AABB y los Octel Tree?

2.- Explica los parámetros del comando utilizado para simular la fricción setfriction.

3.- Haz varias capturas del resultado de ejecutar tu código. Prueba a cambiar algunos parámetros (como el modelo importado, el material, etc) y muéstralo en capturas de pantalla.

Ejercicio 4: Objetos móviles

Partimos del ejercicio anterior, para ir añadiendo sólo el mínimo código necesario.

Objetivo:

Ahora nuestro objeto no estará fijo, sino que podremos desplazarlo al empujarlo con el tool háptico. Al tocar el objeto le imprimiremos una velocidad que se verá frenada por un coeficiente de rozamiento, por lo que el objeto no se desplazará hasta el infinito si no le aplicamos más fuerza. El objeto sólo se va a trasladar aplicando una fuerza sobre su centro de masas, por lo que su orientación no cambiará.

Desarrollo del ejercicio:

En primer lugar, en la configuración del proxy debemos añadir una línea:

```
tool->m_proxyPointForceModel->m_useDynamicProxy = true;
```

Esta línea es necesaria siempre que haya objetos móviles que puedan interactuar con el proxy. Si todos los objetos son estáticos, esta línea no es necesaria y se utilizará un modelo de proxy simplificado.

A continuación, vamos a definir una serie de parámetros físicos para la simulación. Los podemos declarar como variables globales o localmente dentro de updateHaptics:

```
cVector3d velocity;
velocity.zero();

double timestep = 0.001;
double mass = 1.0;

double friction = 0.0001;
double pushGain = 1.0;
```

El vector *velocity* es la velocidad actual del objeto móvil (el coche), que empieza siendo cero. La variable *timestep* representa el paso de tiempo en segundos (1ms en este caso, ya que el bucle háptico funciona a 1KHz) para la integración numérica. La variable *mass* representa la masa del objeto móvil. La variable *friction* representa una hipotética resistencia o fricción con el aire, entre 0 y 1. La variable *pushGain* representa una ganancia en la fuerza con la que empuja el proxy al objeto.

A continuación mostramos el nuevo código para **updateHaptics**:

```
while(simulationRunning)
{
   // compute global reference frames for each object
   world->computeGlobalPositions(true);

   // 1. Read device and update position of tool
   tool->updatePose();
```

```
    // 2. Compute interaction forces
    tool->computeInteractionForces();

    // 3. Send forces to device
    tool->applyForces();

    // 4. Collision response

    // 4.1. check if the tool is touching something
    if (tool->m_proxyPointForceModel->getNumContacts() > 0)
    {
        // 4.2. get last force computed on tool
        cVector3d force = cNegate(tool->m_lastComputedGlobalForce);
        force = pushGain * force;

        // 4.3. compute acceleration
        cVector3d acceleration = (1.0/mass)*force;

        // 4.4. integrate velocity
        velocity = velocity + timestep*acceleration;
    }

    // 4.5. Apply friction
    if (velocity.length() > CHAI_SMALL)
        velocity = (1-friction)*velocity;
    else
        velocity.zero();

    // 4.6. Integrate position
    cVector3d objectPos = object->getGlobalPos();
    objectPos = objectPos + timestep*velocity;
    object->setPos(objectPos);
}

// exit haptics thread
simulationFinished = true;
```

La primera línea que aparece nueva es en la que se llama a computeGlobalPositions. Esto es necesario siempre que haya objetos móviles, ya que por motivos de rendimiento CHAI3D no actualiza las posiciones y orientaciones de los objetos del mundo a no ser que se le diga explícitamente (cuando nos pueda hacer falta tener una posición actualizada para, por ejemplo, interactuar con un objeto).

Además, hemos añadido al bucle háptico la respuesta a colisiones que, en este caso, consiste en mover el objeto al empujarlo con el proxy. Normalmente la respuesta a colisiones se hace en el bucle gráfico, pero en este caso tan sencillo no hay problema en procesarla en menos de 1ms y, por tanto, podemos incluirla en el bucle háptico para simplificar.

La respuesta a colisiones en este caso consta de seis pasos:

1. Comprobar si hay colisión entre el proxy y el objeto. Si la hay, continuamos con el paso 2. Si no, vamos directamente al paso 5.
2. Obtener la fuerza con la que empuja el proxy. En primer lugar, obtenemos el último valor de fuerza del virtual coupling que ha computado internamente el modelo de virtual proxy de CHAI3D (tool->m_lastComputedGlobalForce). A continuación, simplemente multiplicamos este valor por nuestro parámetro pushGain, por si queremos que el proxy empuje al objeto con más fuerza (otra forma de conseguir esto sería reduciendo la masa del objeto).
3. Como ya tenemos la fuerza, podemos obtener directamente la aceleración, dividiendo la fuerza por la masa (2ª ley de Newton).
4. Teniendo la aceleración, podemos obtener la nueva velocidad del objeto mediante una sencilla aproximación por Taylor de primer orden. En realidad estamos haciendo la integración explícita más básica.
5. Tanto si ha habido colisión como si no, tenemos que actualizar la posición del objeto que puede estar moviéndose debido a la inercia de una colisión anterior. En primer lugar, aplicaremos el rozamiento con el aire para contrarrestar la inercia del objeto.
6. Por último, integramos la posición a partir de la velocidad. Aplicamos esta nueva posición al objeto con object->setPos.

Nota: los valores de los parámetros físicos que se muestran en este documento como ejemplo están ajustados para funcionar bien en modo Debug. Para que el movimiento no sea demasiado brusco en modo Release conviene ajustar estos valores.