

## **BITÁCORA - TALLER 5**

### **Simulador de Caída Libre**

#### **Integrantes:**

- **Abel Albuez**
  - **Ricardo Rivas**
- 

## **1. CONFIGURACIÓN INICIAL DEL ENTORNO**

### **Creación del entorno virtual**

**Primero creamos un entorno virtual de Python para aislar las dependencias del proyecto. Esto evita conflictos con otros proyectos y mantiene el sistema limpio.**

### **Creación del archivo requirements.txt**

**Basándonos en el proyecto de referencia, creamos el archivo requirements.txt con todas las dependencias necesarias: ogre-python, vtk, pybullet, numpy, y algunas herramientas de desarrollo como pylint y black.**

**Tiempo invertido: ~10 minutos**

---

## **2. PROBLEMAS DE INSTALACIÓN**

### **Problema 1: Instalación de paquetes básicos**

**Activamos el entorno virtual e intentamos instalar las dependencias con pip install. La mayoría de paquetes se instalaron sin problemas: ogre-python, vtk, numpy y las herramientas de desarrollo funcionaron perfectamente.**

**Resultado:  Instalación exitosa para estos paquetes**

---

### **Problema 2: PyBullet requiere compilación (ERROR CRÍTICO)**

**Cuando pip intentó instalar PyBullet, apareció un error diciendo que no podía construir las "wheels" del paquete y que requería Microsoft Visual C++ versión 14.0 o superior.**

**¿Qué pasó? PyBullet no tiene versiones precompiladas (wheels) disponibles para Python 3.12 en Windows. El paquete necesita compilarse desde el código fuente, y para compilar extensiones de C++ en Windows se requiere tener instalado el compilador de Microsoft.**

**¿Cómo lo resolvimos?**

- 1. Buscamos en internet sobre el error y confirmamos que necesitábamos las Build Tools de Visual Studio**
- 2. Descargamos el instalador de Microsoft Visual C++ Build Tools desde el sitio oficial de Visual Studio**
- 3. Durante la instalación, seleccionamos la opción "Desarrollo para el escritorio con C++" que incluye:**
  - El compilador MSVC**
  - Windows SDK**
  - Herramientas de CMake**
- 4. La instalación de las Build Tools tomó aproximadamente 20 minutos y ocupó unos 6-7 GB de espacio**
- 5. Reiniciamos la terminal para que tomara las nuevas variables de entorno**
- 6. Volvimos a ejecutar pip install pybullet**
- 7. Esta vez PyBullet se compiló correctamente (tomó unos 3 minutos) y se instaló sin problemas**

**Resultado:  PyBullet instalado después de compilación**

**Tiempo invertido: ~45 minutos (incluyendo descarga e instalación de Build Tools)**

**Lección aprendida: En Windows con Python 3.12, es necesario tener un compilador C++ instalado antes de trabajar con paquetes como PyBullet que requieren compilación desde código fuente.**

---

### **3. CAMBIOS AL CÓDIGO**

**Análisis del código base**

Estudiamos el archivo MovingSpheres.py original que tenía una sola esfera de billar roja cayendo desde una posición fija sobre un plano de tamaño fijo. Necesitábamos modificarlo para cumplir con los requisitos del taller.

#### **Cambio 1: Parámetros configurables**

Agregamos variables de clase para hacer el simulador configurable: número de esferas a generar, altura desde donde caen, tiempo entre la generación de cada esfera, y los límites del plano. También agregamos variables internas para llevar el control del contador de esferas y el tiempo acumulado.

#### **Cambio 2: Plano configurable**

Modificamos la llamada que crea el plano para que en lugar de usar valores fijos use el parámetro configurable de límites del plano. Esto cumple con el requisito del taller de que el plano sea definido por el usuario.

#### **Cambio 3: Método para generar esferas aleatorias**

Creamos un método nuevo llamado `_generarEsfera` que genera una esfera en una posición aleatoria. El método calcula posiciones aleatorias en X y Z dentro de los límites del plano (dejando un margen para que no aparezcan en el borde), selecciona un material aleatorio de los cuatro disponibles, crea la esfera visual en Ogre3D, crea la esfera física en PyBullet, las vincula, y muestra un mensaje informativo en consola.

#### **Cambio 4: Generación automática**

Modificamos el método `frameRenderingQueued` para que genere esferas automáticamente. Agregamos lógica que verifica si aún faltan esferas por generar, acumula el tiempo transcurrido, y cuando se alcanza el intervalo configurado llama al método para generar una nueva esfera.

#### **Cambio 5: Eliminar esfera fija**

Eliminamos todo el código que creaba la esfera roja de billar predefinida, ya que ahora las esferas se generan dinámicamente de forma aleatoria.

#### **Cambio 6: Ajustes visuales y físicos**

Cambiamos la cámara de modo libre a modo órbita para facilitar la visualización, ajustamos la posición inicial de la cámara, cambiamos el fondo a un color azul cielo, y modificamos los parámetros físicos del plano para tener un rebote más realista.

Tiempo invertido: ~1 hora

---

## 4. EXPLORACIÓN Y REFINAMIENTO DEL CÓDIGO

### Experimentación con el comportamiento del simulador

Después de tener la versión inicial funcionando, comenzamos a explorar el código para mejorar el comportamiento del simulador. Primero notamos que aunque habíamos definido el parámetro `limites_plano` con valores de -3 a 3 metros, en la línea donde se creaba el plano visual estaba usando valores fijos de -2 a 2 metros. Esto causaba una inconsistencia: las esferas se generaban en posiciones calculadas según los límites configurables, pero el plano visual era más pequeño, entonces algunas esferas aparecían cayendo fuera del plano visible. Para corregir esto, modificamos la llamada al método `_ground` para que usara `self.limites_plano` en lugar de los valores fijos, asegurando que el plano visual coincidiera exactamente con el área donde se generan las esferas.

Durante las pruebas también exploramos la idea de hacer que las esferas cayeran de manera infinita en lugar de detenerse después de generar diez esferas. Originalmente teníamos una variable `num_esferas` que limitaba la cantidad, y en el método `frameRenderingQueued` había una condición que verificaba si el contador era menor que este número antes de generar más esferas. Para lograr la caída infinita simplemente eliminamos la variable `num_esferas` y quitamos la condición del `if` que verificaba el límite, dejando únicamente la lógica de tiempo que controla el intervalo entre cada generación. De esta manera el simulador continúa generando esferas indefinidamente cada medio segundo, creando una lluvia constante de esferas que caen sobre el plano. Esto hace la simulación más dinámica e interesante visualmente, aunque hay que tener cuidado con el rendimiento si se deja corriendo por mucho tiempo.

---

## 5. PROBLEMA CON RECURSOS DE OGRE

### El error

Al intentar ejecutar el programa por primera vez después de nuestras modificaciones, la aplicación se cerraba inmediatamente con un error diciendo que no podía localizar el archivo `OgreUnifiedShader.h` en el grupo de recursos `OgreInternal`.

¿Qué significaba? `Ogre3D` necesita ciertos archivos de shaders internos para funcionar correctamente. El archivo `resources.cfg` le dice a `Ogre` dónde buscar estos

recursos, pero solo tenía configurada la carpeta de recursos del proyecto, no los recursos internos de Ogre.

#### La solución

Investigamos dónde estaban los recursos internos de Ogre y descubrimos que estaban en la carpeta Media dentro del entorno virtual. Actualizamos el archivo resources.cfg agregando una nueva sección llamada OgreInternal con tres rutas: una para los shaders principales que incluyen OgreUnifiedShader.h, otra para la biblioteca de shaders en tiempo de ejecución, y otra para recursos de terreno.

Resultado:  La aplicación inició correctamente

Tiempo invertido: ~15 minutos

---

## 6. PRUEBAS Y VERIFICACIÓN

### Pruebas funcionales realizadas

Verificamos que las esferas se generaran en posiciones aleatorias diferentes, que los colores fueran aleatorios, que todas las posiciones estuvieran dentro del plano visible, que las esferas cayeran por gravedad y rebotaran al tocar el suelo, y que hubiera interacción física correcta entre las esferas. Con la implementación de caída infinita, observamos que el simulador continuaba generando esferas sin detenerse, creando una acumulación gradual de objetos físicos en el plano.

### Pruebas de configuración

Probamos cambiar el tamaño del plano a valores pequeños y grandes para verificar que las esferas siempre cayeran dentro del área visible después de corregir el error. Ajustamos el intervalo de generación para observar cómo afectaba la densidad de esferas. Al dejar el simulador corriendo durante varios minutos con caída infinita, notamos que después de aproximadamente 100 esferas el rendimiento comenzaba a degradarse notablemente.

### Resultados

Con el intervalo de 0.5 segundos entre esferas, la generación es constante y fluida. El rendimiento se mantiene estable durante los primeros minutos, pero eventualmente la acumulación de objetos físicos causa degradación. Para uso prolongado sería recomendable implementar algún sistema de limpieza de esferas antiguas.

Tiempo invertido: ~30 minutos

---

## 7. RESUMEN FINAL

### Archivos modificados

- **MovingSpheres.py:** Aproximadamente 30 líneas agregadas o modificadas
- **resources.cfg:** 4 líneas agregadas para recursos internos de Ogre

### Requisitos del taller cumplidos

✓ Generación aleatoria de esferas   ✓ Caída libre sobre un plano   ✓ Plano definido por el usuario (configurable)

### Problemas enfrentados y resueltos

1. **PyBullet requiere compilador C++ en Windows - Resuelto instalando Visual C++ Build Tools**
2. **Recursos internos de Ogre no encontrados - Resuelto actualizando resources.cfg**

**Tiempo total invertido: Aproximadamente 2.5 horas**

### Aprendizajes

Hemos visto cómo a medida que avanzamos con los talleres los cambios se vuelven más mínimos y se van abstrayendo las responsabilidades. Cuando solo teníamos VTK había que preocuparse por crear manualmente la geometría completa de cada objeto, calcular las normales para la iluminación, configurar el pipeline de renderizado completo desde cero, manejar la ventana de visualización, configurar la cámara, las luces, y básicamente todo el proceso de llevar los datos geométricos a la pantalla. Era necesario entender profundamente cómo funcionaba cada etapa del pipeline gráfico.

Con Ogre3D nos preocupamos solo por la escena y los objetos visuales. A diferencia de VTK, Ogre ya abstrae todo el pipeline de renderizado y nos proporciona un sistema de escenas con nodos jerárquicos. Ya no tenemos que preocuparnos por configurar el render pipeline manualmente, solo creamos objetos, los adjuntamos a nodos, y Ogre se encarga de renderizarlos eficientemente. Los materiales, shaders, e iluminación ya están manejados por el motor.

Ahora con PyBullet como motor de física para videojuegos, a diferencia de Ogre y VTK, nos preocupamos únicamente por definir las propiedades físicas de los objetos: masa, forma de colisión, fricción y rebote. El motor se encarga automáticamente de

calcular todas las interacciones físicas, las colisiones entre objetos, la respuesta a la gravedad, y actualizar las posiciones frame a frame. No necesitamos implementar detección de colisiones ni resolver las ecuaciones de movimiento, solo sincronizamos las posiciones calculadas por Bullet con la representación visual en Ogre.

Además, el sistema de colisiones de PyBullet funciona de manera muy eficiente y realista. Cada objeto tiene dos representaciones: una visual en Ogre3D (lo que vemos) y una física en PyBullet (cómo interactúa). El motor de física calcula continuamente las colisiones entre las formas geométricas simplificadas, aplica las fuerzas, y resuelve las restricciones físicas. Lo impresionante es que solo necesitamos llamar a `stepSimulation` en cada frame y luego copiar las posiciones y orientaciones resultantes a los nodos visuales de Ogre. Esta separación entre física y gráficos permite que cada sistema haga lo que mejor sabe hacer: Bullet calcula física precisa con formas simplificadas para velocidad, mientras Ogre renderiza geometría detallada para calidad visual.

La clave está en la sincronización: en cada frame primero avanzamos la simulación física, luego obtenemos las nuevas posiciones de PyBullet, y finalmente actualizamos los nodos visuales en Ogre3D. Este patrón de "simular física → obtener resultados → actualizar gráficos" es fundamental en los motores de videojuegos modernos y permite separar las responsabilidades de manera clara y eficiente.