Game Design Document
Bolotro

Grupos 3 y 4

Jorge Fernández Marín (839113)

Javier Julve Yubero (840710)

Marcos García Ortega (844419)

Saul Caballero Luca (848431)

Aroa Redondo Zamora (851769)

11 de julio de 2025

Índice

1. Descripción del juego

Bolotro es un juego individual que busca replicar el conocido juego de los bolos. para ello el jugador debe intentar derribar los 10 bolos que hay sobre la pista con la bola de bolos que puede lanzar. El objetivo del juego será la de conseguir la máxima cantidad de puntos posibles.
El jugador dispondrá de 2 tiros por cada ronda, y un total de 10 rondas. En cada una de estas, podrá obtener diferentes multiplicadores de puntuación en relación a cuantos bolos consiga derribar. Al final de la partida, se podrá observar la puntuación total del juego con el objetivo de competir contra sí mismo o contra sus amigos.

2. Implementación

El juego sufrió una refactorización del código a mitad del proceso de desarrollo con el objetivo de modularizar lo máximo posible el proyecto. El proyecto se compone de los siguientes archivos:

- `main.js`
  Es el archivo principal del juego. Inicializa los componentes clave como la escena y la cámara, y arranca el bucle de animación.
- `Camera.js`
  Define la cámara del juego y su comportamiento.
- `InputManager.js`
  Se encarga de gestionar las respuestas a las pulsaciones de teclado del usuario.
- `Geometría.js`
  Define la geometría de los cubos, bolas y planos del juego.
- `SceneManager.js`
  Administra los objetos de la escena, y coordina la renderización de los mismos junto al `WebGLManager`.
- `WebGLManager.js`
  Maneja directamente la inicialización y configuración del contexto WebGL, incluyendo shaders, buffers y texturas necesarias para el renderizado.
- `Objects.js`
  Define las entidades del juego (bola de bolos, bolos y suelo), junto a sus atributos y comportamiento.
- `PhysicsManager.js`
  Controla la física de la simulación, como movimiento, colisiones y fuerzas. Vease la sección motor de físicas para entender el comportamiento en detalle.
- `ForceBar.js`
  Controla el comportamiento visual de la barra de fuerza y la fuerza aplicada a la bola una vez que esta es lanzada.
- `MassManager.js`
  Gestiona la masa que tiene la bola antes de que la misma sea lanzada. El como afecta a la jugabilidad puede ser consultado en la sección motor de físicas.
- `ScoreManager.js`
  Gestiona el sistema de puntos, manejando tanto los bonificadores y turno extra en la última jugada en caso de pleno, como la tabla visual de puntuaciones.
- `OBJLoader.js`
  Es un parseador de OBJs que consiste en leer los vértices y normales de un OBJ para poder transformarlo en un modelo 3D en el juego ya renderizado.

3. Gameplay e interfaz

- **Controles**.
  - **Flechas** ← →: Permiten mover la cámara con la bola a la izquierda o a la derecha.
  - **Shift + Flechas** ← →: Permiten rotar la cámara con la bola a la izquierda o a la derecha.
  - **Flecha** ↑: Permite poner una vista de águila en cualquier momento para ver cuantos bolos quedan en la pista.
  - **R**: Pasa de tirada sin esperar a que el turno acabe.
  - **Espacio**: Al mantenerlo se empieza a cargar y descargar una barra de fuerza, cuando lo soltemos la bola de bolos será lanzada con una fuerza proporcional a la de la barra.
- **Tabla de puntuación**. Esta tabla va marcando los bolos derribados que llevas y la puntuación total de cada ronda siguiendo las reglas oficiales de los bolos y aplicando sus multiplicadores.

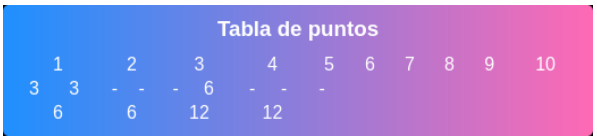


Figura 1: Tabla de puntuación

- **GAME OVER**. Al acabar el juego saldrá un cuadro de texto que indica que el juego acabó, nuestra puntuación y un botón para volver a jugar.

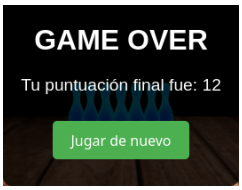


Figura 2: GAME OVER

- **Indicador de fuerza**. Este indicador consiste en una barra horizontal que indica la potencia de la fuerza de lanzamiento, cuanto más pulse el usuario, más irá aumentando la barra, lo que se simboliza con un degradado de verde a rojo, siendo el lanzamiento más débil el verde y el más fuerte el rojo.

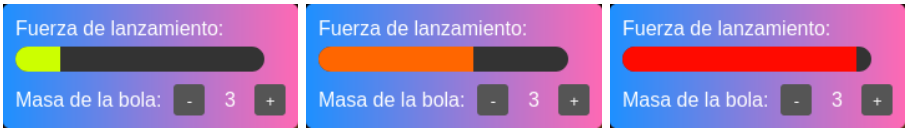


Figura 3: Diferentes cargas de la barra de fuerza

- **Selector de masa**. Justo debajo del indicador de fuerza se encuentra el selector de la masa de la bola. Esta masa afectará al comportamiento del lanzamiento puesto que el motor de físicas se ha implementado de manera realista como se detalla en el apartado `motor de físicas`.

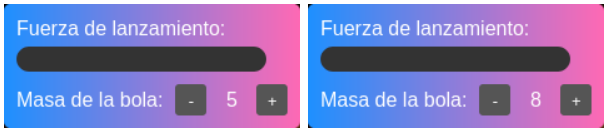


Figura 4: Diferentes masas de la bola

- **Botones de música y sonido**. Estos botones se encuentran también en la pantalla donde se desarrolla el gameplay.
    - **Botón de música**: Al pulsar este botón podremos iniciar y pausar la música del juego.

Figura 5: Botón de música

    - **Botón de sonido**: Al pulsar este botón se activará el sonido del juego, que principalmente está presente cuando derribamos algún bolo.

Figura 6: Botón de sonido

    - **Botón de cambio de fondo de pista**: Podemos pulsar este botón para cambiar el fondo de la pista, que variará entre una selección de imágenes que incluyen una selección de sorpresas.

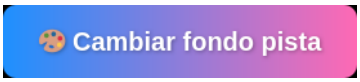


Figura 7: Botón de fondo de pista

    - **Botón ocultar flecha**: Este botón nos sirve para eliminar la flecha que nos indica la dirección del lanzamiento, si el usuario quisiera una experiencia más desafiante.

Figura 8: Botón ocultar flecha

- **Menús HTML**.
    - **Menú Principal**: Con botones de 'Jugar' y 'Controles'.
    - **Pantalla de Controles**: Descripción de las teclas.
    - **Pantalla de juego**: Es donde se desarrolla el juego en sí.

4. Motor de físicas

Con el objetivo de conseguir un gameplay realista, se creó un motor de físicas que simula el comportamiento de colisiones de cuerpos sólidos. El motor se divide en 2 partes fundamentales las cuales son la detección de colisiones y la resolución de las mismas.

4.1. Introduccion al sistema de físicas

Antes de empezar a hablar del sistema del motor de físicas, debemos de saber que existen 3 tipos de objetos en nuestro juego, los cuales son:

- La **bola de bolos**.
  Es el unico objeto que el jugador controla, y la resolución de sus colisiones es sencilla pues tiene una geometría esferica sencilla.

- Los **bolos**.
  Son los objetos que el jugador tiene que derribar y tienen una geometría compleja, es por ello que se ha optado por sustituir su comportamiento por cubos. En el juego, todos los cálculos de colisiones se le aplican a los cubos y luego los datos de posicion y rotación se copian al objeto del bolo con el objetivo de simplificar los y resolución de colisiones.
  Véase la Figura 8.

- El **plano**. Representa el suelo del juego, la geometría es simple y su comportamiento es distinto a los de los demás objetos de la escena pues este es inamovible.

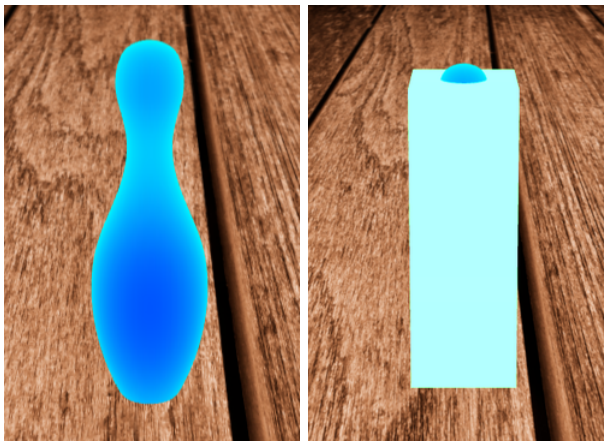
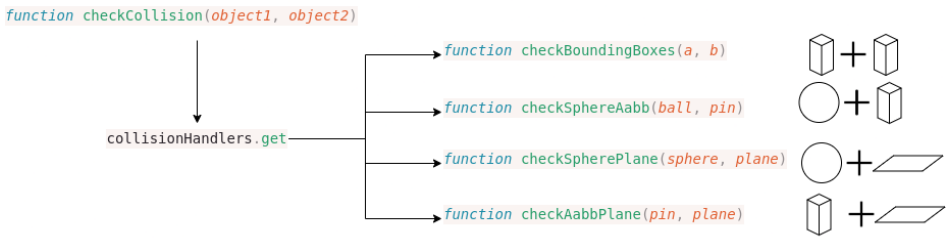


Figura 9: Diferencia entre la representación visual y matemática del bolo

4.2. Comprobacion de colisiones

La resolución de colisiones se implementa en el archivo `PhysicsEngine.js`, mediante la función `checkCollision`. Esta función `checkCollision` está dividida en 2 partes, la primera siendo identificar cuales son los objetos en los que queremos calcular y la segunda es saltar a la función especifica que calcula si ha habido colisión entre los objetos. Las colisiones se detectan por metodos triviales sencillos para cajas y esferas como aligned axis boxes(Aabb). Vease la siguiente figura.



4.3. Resolución de las colisiones

La implementación de resolución de colisiones fue un tema debatido en el grupo, pero se acabo apostando por implementar un modelo de **fisicas de objetos rigidos** simplificada, haciendo referencia a las ecuaciones que modelan los cuerpos solidos en la vida real. Una vez que sabemos que 2 objetos han colisionado, la resolución se realiza en 2 partes.

- La **resolución de momento linear**.
  Se define como la velocidad en línea recta de un punto a otro, o lo que es lo mismo, el cambio en la

posición de un objeto con el tiempo en una ruta rectilínea. La función `resolveLinearMomentum(dt, obj1, obj2)` es la que se encarga de resolver para 2 objetos la velocidad linear despues de su impacto y realiza los siguientes cálculos:

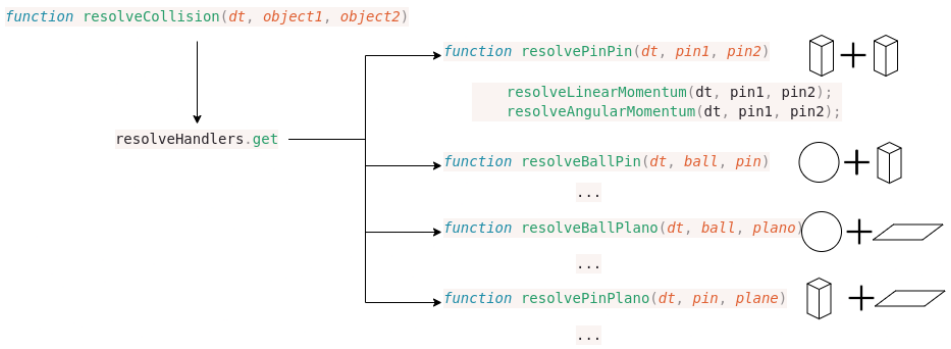
1. Calcular el vector normal $\vec{n} = \text{normalize}(\vec{p}_1 - \vec{p}_2)$
2. Obtener las componentes normales $v_{1n} = \vec{v}_1 \cdot \vec{n}$, $v_{2n} = \vec{v}_2 \cdot \vec{n}$
3. Calcular la nueva velocidad normal de $\vec{v}_1$:

$$v'_{1n} = \frac{(m_1 - m_2)v_{1n} + 2m_2 v_{2n}}{m_1 + m_2}$$

4. Sumar la componente tangencial original $\vec{v}_{1t}$ y la nueva normal para obtener $\vec{v}'_1$
5. Actualizar la posición: $\vec{p}'_1 = \vec{p}_1 + \Delta t \cdot \vec{v}'_1$

- La **resolución de momento angular**.
  La velocidad angular es una medida de la velocidad de rotación al rededor de su propio eje.
  La función `resolveAngularMomentum(dt, obj1, obj2)` es la que se encarga de resolver para 2 objetos la velocidad angular despues de su impacto y realiza los siguientes cálculos:

1. Calcular el vector desde el centro del segundo objeto al primero $\vec{r} = \vec{p}_1 - \vec{p}_2$
2. Calcular la fuerza de impacto $\vec{F} = -100 \cdot m_2 \cdot \vec{v}_2$
3. Obtener el torque $\vec{\tau} = \vec{r} \times \vec{F}$
4. Calcular la aceleración angular $\vec{\alpha} = \vec{\tau}$
5. Actualizar la velocidad angular: $\vec{\omega}'_1 = \vec{\omega}_1 + \Delta t \cdot \vec{\alpha}$



5. Representación gráfica

- **Shaders** (`Bolotro.html`). Nuestros shaders se aplican a los 3 objetos principales (esfera, plano y bolo) y su función es generar un mapa de normales u, v para que luego podamos usar dichas normales a la hora de aplicar las diferentes texturas a los objetos.

- **Texturas**. Las texturas son simplemente imágenes planas en una resolución cuadrada que envuelven el objeto al que va asociado y usando los mapas de normales generados por los shaders.

- **OBJs**. Se ha creado un importador de OBJs para poder importar modelos 3D externos que no fueran figuras geométricas básicas para que los bolos tuvieran dicha forma.

- **Buffers y atributos**. Los objetos en la escena se guardan en `objectsToDraw`. Todos los objetos en dicho array tienen el su programInfo correspondiente al tipo de objeto que son, `pointsArray`, que contienen los puntos en el espacio 3D y el orden en el que se han de unir para formar los triángulos necesarios para su representación, además los bolos poseen un segundo array llamado `visualPointsArray` que sirve para almacenar allí el modelo visual del OBJ importado ya que en pointsArray se almacena su hitbox, la cual es invisible en el juego.

- **Matrices**.
  - `model`, `view`, `projection` son calculadas con utilidades de `MVnew.js`.
  - Transformaciones de posición aplicadas con `translate` y multiplicaciones de matrices, además las del bolo se realizan en ambos array simultáneamente para que lo visual concuerde con lo interactuable.

6. Información del desarrollo

6.1. Código base

A continuación se muestra un resumen de las métricas de desarrollo del proyecto:

- **Commits totales**: 38
- **Cambios en el tiempo (ctimes) totales**: 609
- **Archivos totales**: 32
- **Líneas de código (LOC) totales**: 4166

Autor	LOC	Commits	Archivos	Distribución (LOC/Commits/Archivos)
DonJulve	3036	25	22	72.9 % / 65.8 % / 68.8 %
Jorge Fernandez	1130	13	10	27.1 % / 34.2 % / 31.2 %

Cuadro 1: Estadísticas de contribución por autor

6.2. Cronograma y reparto de tareas

En total se calculan las siguientes horas de trabajo:

- **Jorge Fernández**. 40 Horas.
  Estudio de la nueva arquitectura de proyecto, refactorización del código base, estudio de físicas de cuerpos solidos, implementación del motor de físicas y propiedades de objetos y creación de la memoria.
- **Javier Julve**. 54 Horas.
  Diseño e implementación de las mecánicas del juego, así como del gameplay, creación de todos los menús y botones junto a sus funciones, diseño e implementación de shaders, manejo de texturas y de modelos 3D importados, creación de la memoria y elaboración del tráiler.

6.3. Problemas encontrados

- **Refactorización del código base**.
  A mitad de la etapa de desarrollo, se cambiaron los roles de trabajo Jorge Fernández y Marcos García. En esa étapa de cambio de roles se decidió por seccionar el código en módulos más pequeños puesto que el proyecto estaba compactado en un solo archivo y trabajar en él de 0 era una tarea de extrema complejidad. El refactor tardó al rededor de 1 semana y media y permitió trabajar sobre el código de manera más sencilla y permitiendo un aumento en el rendimiento de trabajo de los integrantes.
- **Creación del sistema de físicas**.
  Se optó por implementar físicas realistas para el comportamiento del juego, lo que obligó a estudiar el funcionamiento de la física de cuerpos sólidos con el objetivo de usar las formulas que modelan el comportamiento real en el juego.

- **Uso de ficheros OBJ**.
  Como se ha explicado en el problema anterior, el sistema de físicas estaba implementado para funcionar con figuras geométricas simples, con lo que el modelo de bolos fue importado más tarde y se tuvo que entender como funciona un OBJ por dentro para hacer un parseador y así poder usar el modelo en el juego. Cabe destacar además la complejidad encontrada a la hora de almacenarlo en un segundo array dentro del objeto y el aplicarle las mismas operaciones de transformación a este OBJ y al modelo del bolo antiguo que ahora actua de hitbox invisible.

- **Cambio de número de integrantes del grupo**.
  Durante el desarrollo del juego, el grupo sufrió por la falta de trabajo de uno de los integrantes, debido a esto, se tomó la drástica decisión de expulsarle del grupo ya que a 2 semanas de la entrega, sus contribuciones en el proyecto habían sido mínimas.

- **Monolitización del proyecto**.

  Durante el desarrollo del proyecto, se decidió crear el juego en base a una arquitectura modular con el objetivo de separar segmentos de código en archivos distintos, lo que acarrea el problema de que cuando se llama al módulo principal, este tendrá que hacer peticiones HTTP / HTTPS a los distintos módulos para poder usarlos. Esto hace que cuando queramos jugar al juego, debamos levantar un servidor local, en nuestro caso, con `python`. Al intentar monolitizar el proyecto, nos encontramos con que el rendimiento caía de tal manera, que el juego no se comportaba como estaba planteado, es por eso, por lo que descartamos la opción de monolitizar nuestro juego.