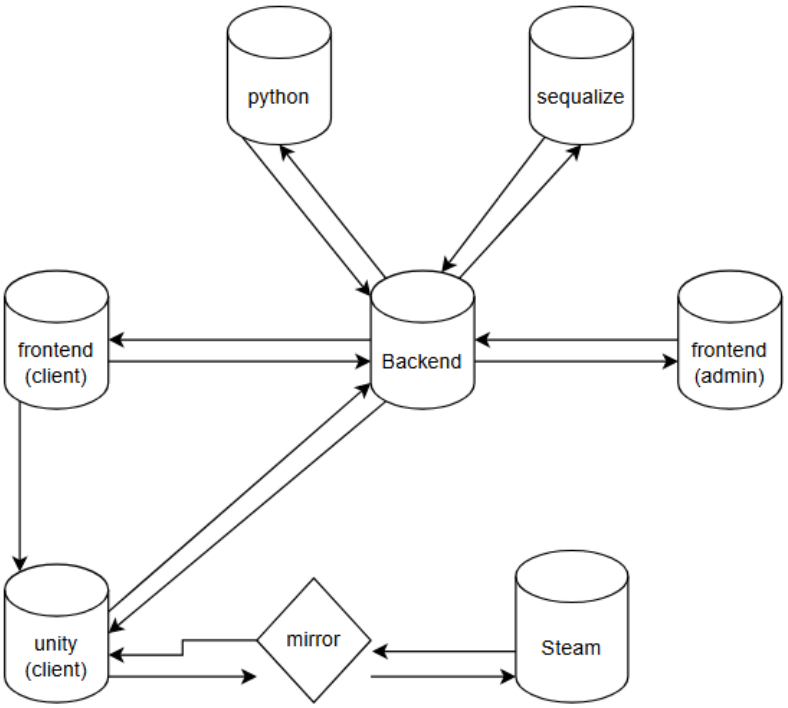
1. Arquitectura de l’aplicació:



2. Rutes de l'aplicació:

Servicio de Bosses (Puerto: 3008)

```
// Obtiene todos los bosses
app.get('/bosses', async (req, res)
// Parámetros: No requiere
// Retorna: Array de objetos Boss
// {
//   id_boss: number,
//   boss_name: string,
//   boss_max_health: number,
//   move_speed: number,
//   attack_range: number,
//   attack_cooldown: number,
//   attack1_damage: number,
//   attack2_damage: number,
//   attack3_damage: number
// }

// Obtiene un boss específico por ID
app.get('/bosses/:id', async (req, res)
// Parámetros URL:
//   - id: number (ID del boss)
// Retorna: Objeto Boss (misma estructura que arriba)

// Actualiza estadísticas de un boss
app.put('/bosses/:id', async (req, res)
// Parámetros URL:
//   - id: number (ID del boss)
// Body: {
//   boss_name?: string,
//   boss_max_health?: number,
//   move_speed?: number,
//   attack_range?: number,
//   attack_cooldown?: number,
//   attack1_damage?: number,
//   attack2_damage?: number,
//   attack3_damage?: number
```

```
// }
// Retorna: Objeto Boss actualizado
```

Servicio de Enemies (Puerto: 3007)

```
// Obtiene todos los enemigos
app.get('/enemies', async (req, res)
// Parámetros: No requiere
// Retorna: Array de objetos Enemy
// {
//   id_enemy: number,
//   enemy_name: string,
//   enemy_max_health: number,
//   move_speed: number,
//   attack_range: number,
//   attack_cooldown: number,
//   attack_damage: number
// }

// Obtiene un enemigo específico
app.get('/enemies/:id', async (req, res)
// Parámetros URL:
//   - id: number (ID del enemigo)
// Retorna: Objeto Enemy (misma estructura que arriba)

// Actualiza estadísticas de un enemigo
app.put('/enemies/:id', async (req, res)
// Parámetros URL:
//   - id: number (ID del enemigo)
// Body: {
//   enemy_name?: string,
//   enemy_max_health?: number,
//   move_speed?: number,
//   attack_range?: number,
//   attack_cooldown?: number,
//   attack_damage?: number
// }
// Retorna: Objeto Enemy actualizado
```

Servicio de Game (Puerto:3002)

```
// Crea una nueva partida
router.post('/newGame', async (req, res)
```

```
// Body: {
//   nickname: string,       // Nickname del jugador
//   game_name: string,      // Nombre de la partida
//   game_status?: string,   // Estado (default: 'active')
//   total_progress?: number, // Progreso total (default: 0.00)
//   time_played?: number,    // Tiempo jugado (default: 0)
//   position_x?: number,     // Posición X (default: 0)
//   position_y?: number,     // Posición Y (default: 0)
//   health?: number,         // Salud (default: 100)
//   coins?: number          // Monedas (default: 0)
// }
// Retorna: {
//   message: string,
//   game: Object           // Datos de la partida creada
// }

// Obtiene información de la partida actual
router.get('/game', async (req, res)
// Query params:
//   - nickname: string    // Nickname del jugador
// Retorna: {
//   id_game: number,
//   game_name: string,
//   game_status: string,
//   total_progress: number,
//   time_played: number,
//   position_x: number,
//   position_y: number,
//   health: number,
//   coins: number,
//   id_player: number
// }
// Carga una partida específica por nickname
router.get('/loadGame/:nickname', async (req, res)
// Parámetros URL:
//   - nickname: string    // Nickname del jugador
// Retorna: Objeto Game con todos los datos de la partida

// Obtiene la última partida de un jugador
router.get('/lastGame/:nickname', async (req, res)
// Parámetros URL:
//   - nickname: string    // Nickname del jugador
// Retorna: {
```

```
//   game: {
//     id_game: number,
//     game_name: string,
//     game_status: string,
//     total_progress: number,
//     last_save_date: date,
//     position_x: number,
//     position_y: number,
//     health: number,
//     coins: number,
//     inventory_id: number
//   }
// }

// Elimina una partida
router.delete('/deleteGame/:nickname', async (req, res)
// Parámetros URL:
//   - nickname: string    // Nickname del jugador
// Retorna: {
//   message: string       // Mensaje de confirmación
// }

// Actualiza una partida
router.put('/updateGame/:nickname', async (req, res)
// Parámetros URL:
//   - nickname: string    // Nickname del jugador
// Body (opcional):
//   - game_name?: string
//   - game_status?: string
//   - total_progress?: number
//   - time_played?: number
//   - position_x?: number
//   - position_y?: number
//   - health?: number
//   - coins?: number
// Retorna:
// {
//   message: string,       // Mensaje de confirmación
//   game: Object           // Partida actualizada
// }
```

Servicio de Player (Puerto:3001)

```
// Crea un nuevo jugador
router.post('/player', async (req, res)
// Body: {
//   nickname: string      // Nickname único del jugador
// }
// Retorna: {
//   message: string,
//   player: Object        // Datos del jugador creado
// }

// Login de jugador
router.post('/loginPlayer', async (req, res)
// Body: {
//   nickname: string     // Nickname del jugador
// }
// Retorna: {
//   id_player: number,
//   nickname: string,
//   created_at: date,
//   updated_at: date
// }

// Obtiene todos los jugadores
router.get('/players', async (req, res)
// Parámetros: No requiere
// Retorna: Array de objetos Player

// Elimina un jugador
router.delete('/player/:id', async (req, res)
// Parámetros URL:
//   - id: number         // ID del jugador
// Retorna: {
//   message: string     // Mensaje de confirmación
// }
```

Servicio de Inventory (Puerto:3003)

```
// Crea un inventario inicial
router.post('/createInventory', async (req, res)
// Body: {
//   player_id: number    // ID del jugador
// }
// Retorna: {
```

```
//   message: string,
//   id_inventory: number,
//   items: Array         // Items iniciales añadidos
// }

// Obtiene el inventario de un jugador
router.get('/inventory/:nickname', async (req, res)
// Parámetros URL:
//   - nickname: string   // Nickname del jugador
// Retorna: Array de {
//   id_item: number,
//   item_name: string,
//   item_description: string,
//   item_type: string,
//   value: number,
//   rarity: string,
//   item_image: string,
//   quantity: number
// }

// Añade un ítem al inventario
router.post('/addItem', async (req, res)
// Body: {
//   player_id: number,   // ID del jugador
//   item_id: number,     // ID del ítem
//   quantity?: number    // Cantidad (default: 1)
// }
// Retorna: {
//   message: string
// }

// Elimina un ítem del inventario
router.delete('/removeItem', async (req, res)
// Body: {
//   player_id: number,
//   item_id: number
// }
// Retorna: {
//   message: string
// }

// Obtiene información de un ítem específico
router.get('/item/:itemId', async (req, res)
```

```
// Parámetros URL:
//   - itemId: number    // ID del ítem
// Retorna:
//   - Objeto Item con todos sus campos
//   - 404 si no se encuentra el ítem
//   - 500 en caso de error del servidor
```

Servicio de EnemyDeathStats (Puerto:3006)

```
// Sirve la imagen de estadísticas de muertes de enemigos
router.get('/enemy-death-stats/image', (req, res)
// Parámetros: No requiere
// Retorna:
//   - Archivo de imagen PNG generado previamente
//   - 404 si la imagen no se encuentra
```

Servicio de User (Puerto:3004)

```
// Registra un nuevo usuario
router.post('/register', async (req, res)
// Body: {
//   username: string,
//   email: string,
//   password: string
// }
// Retorna: {
//   message: string,
//   user: Object           // Datos del usuario creado (sin
password)
// }

// Login de usuario
router.post('/login', async (req, res)
// Body: {
//   email: string,
//   password: string
// }
// Retorna: {
//   token: string,        // JWT token
//   user: Object          // Datos del usuario (sin password)
// }

// Obtiene todos los usuarios
app.get('/users', async (req, res)
// Parámetros: No requiere
```

```
// Retorna: Array de objetos User
//   - 500 en caso de error del servidor


// Obtiene un usuario por ID
app.get('/users/:id', async (req, res)
// Parámetros URL:
//   - id: number     // ID del usuario
// Retorna: Objeto User
//   - 404 si no se encuentra el usuario
//   - 500 en caso de error del servidor


// Crea un nuevo usuario
app.post('/users', async (req, res)
// Body:
//   - name: string
//   - email: string
//   - password: string
// Retorna: Objeto User creado
//   - 400 si hay error de validación
```

Servicio de Items (Puerto:3005)

```
// Obtiene todos los items
router.get('/items', async (req, res)
// Parámetros: No requiere
// Retorna: Array de {
//   id_item: number,
//   item_name: string,
//   item_description: string,
//   item_type: string,
//   value: number,
//   rarity: string,
//   item_image: string
// }

// Obtiene un item específico
router.get('/items/:id', async (req, res)
// Parámetros URL:
//   - id: number        // ID del item
// Retorna: Objeto Item (misma estructura que arriba)

// Actualiza un item
```

```
router.put('/items/:id', async (req, res)
// Parámetros URL:
//   - id: number       // ID del item
// Body: {
//   item_name?: string,
//   item_description?: string,
//   item_type?: string,
//   value?: number,
//   rarity?: string,
//   item_image?: string
// }
// Retorna: {
//   message: string,
//   item: Object       // Item actualizado
// }

// Elimina un item
router.delete('/items/:id', async (req, res)
// Parámetros URL:
//   - id: number      // ID del item
// Retorna: {
//   message: string   // Mensaje de confirmación
// }
```

Servicio de Shop (Puerto:3009)

```
// Subir imágenes de una nueva skin (8 skins + 1 portada)
app.post('/shop/upload-images', upload.array('images', 9), async
(req, res)
// Body:
//   - targetFolder: string
// Form-data:
//   - images[]: 9 archivos (8 skins + 1 portada JPG)
// Retorna: { message: string }
//   - 400 si faltan imágenes o el targetFolder
//   - 500 en caso de error

// Obtener todas las skins
app.get('/shop', async (req, res)
// Parámetros: ninguno
// Retorna: Array de objetos Shop
//   - 500 en caso de error
```

```
// Obtener una skin por ID
app.get('/shop/:id', async (req, res)
// Parámetros URL:
//   - id: number
// Retorna: objeto Shop
//   - 404 si no se encuentra
//   - 500 en caso de error

// Crear una nueva skin
app.post('/shop', async (req, res)
// Body: datos de la skin (ej. skin_name, precio, etc.)
// Retorna: objeto Shop creado
//   - 500 si falla la creación

// Actualizar una skin
app.put('/shop/:id', async (req, res)
// Parámetros URL:
//   - id: number
// Body: campos actualizados (incluye cambio de nombre)
// Efectos: renombra archivos e imágenes si cambia el nombre
// Retorna: objeto actualizado
//   - 404 si no se encuentra
//   - 500 en caso de error

// Actualizar imágenes de una skin existente
app.post('/shop/update-images/:id', upload.array('images', 9),
async (req, res)
// Parámetros URL:
//   - id: number
// Body:
//   - targetFolder: string
//   - oldSkinName: string
// Form-data:
//   - images[]: hasta 9 imágenes (skins y/o portada)
// Retorna: { message: string, newImageUrl: string }
//   - 400 si falta targetFolder
//   - 404 si la skin no existe
//   - 500 en caso de error

// Eliminar una skin y sus imágenes
app.delete('/shop/:id', async (req, res)
// Parámetros URL:
//   - id: number
```

```
// Efectos: elimina la skin, su carpeta de imágenes y su portada
// Retorna: { message: string }
//   - 404 si no existe
//   - 500 en caso de error

// Descargar ZIP con imágenes de una skin
app.get('/shop/zip/:skinName', (req, res)
// Parámetros URL:
//   - skinName: string
// Retorna: archivo ZIP descargable con las imágenes de la skin
//   - 404 si la carpeta no existe
//   - 500 si falla la compresión
```

Servicio de Control (Puerto: 3000)

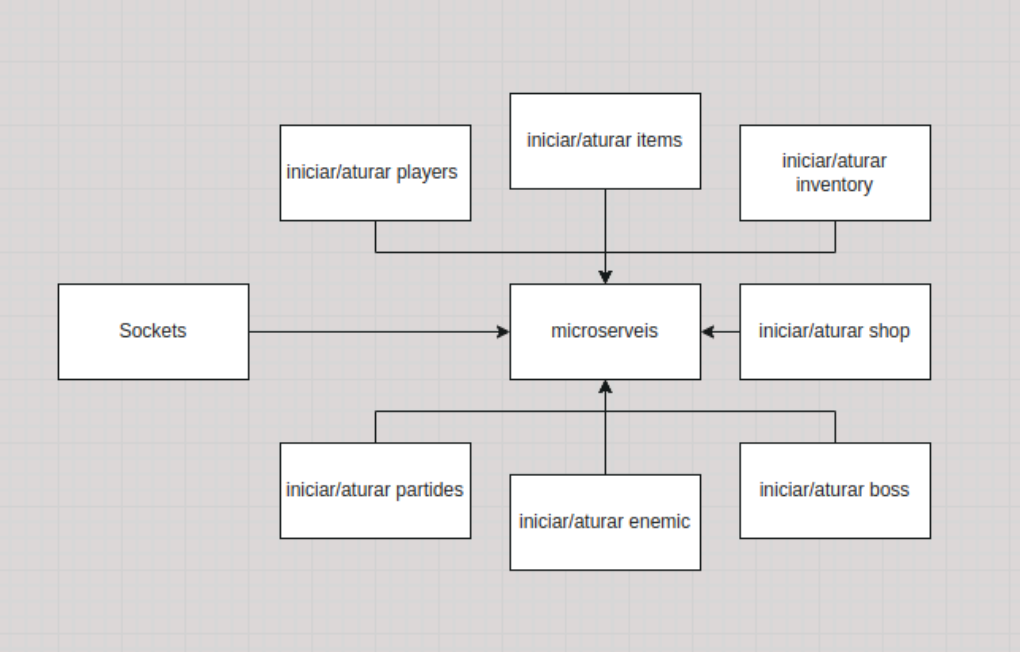
```
// Obtener estado de todos los servicios
app.get('/services', async (req, res)
// Parámetros: ninguno
// Retorna:
//   - success: boolean
//   - services: array con el estado de cada servicio
//   - 500 en caso de error general

// Iniciar un servicio específico
app.post('/services/:serviceType/start', checkServiceExists,
async (req, res)
// Parámetros URL:
//   - serviceType: string (ej. 'shop', 'stats', etc.)
// Middleware: checkServiceExists (verifica que el servicio
existe)
// Retorna:
//   - success: boolean
//   - message: string (confirmación)
//   - service: objeto con estado del servicio
//   - 500 si falla al iniciar el servicio

// Detener un servicio específico
app.post('/services/:serviceType/stop', checkServiceExists, async
(req, res)
// Parámetros URL:
//   - serviceType: string
// Middleware: checkServiceExists
// Retorna:
//   - success: boolean
```

```
//    - message: string (confirmación)
//    - service: objeto con estado actualizado
//    - 500 si falla al detener el servicio
```

3. Esquema d'esdeveniments de comunicació (sockets):

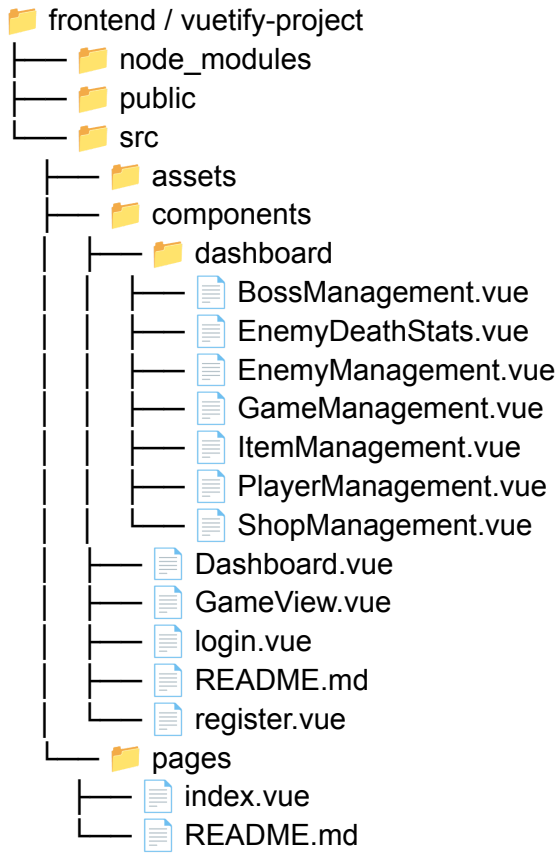


4. Base de dades

```
/**
* Modelo Boss - Representa los jefes del juego
* Este modelo define la estructura y comportamiento de los jefes
en la base de datos
*/
const Boss = sequelize.define('Boss', {
 // Identificador único del jefe
 id_boss: {
     type: DataTypes.INTEGER,
     primaryKey: true,
     autoIncrement: true
 },
 // Nombre del jefe
 boss_name: {
     type: DataTypes.STRING(50),
     allowNull: false
 },
 // Puntos de vida máximos del jefe
 boss_max_health: {
```

```
        type: DataTypes.INTEGER,
        allowNull: false
    },
    // Velocidad de movimiento del jefe
    move_speed: {
        type: DataTypes.FLOAT,
        allowNull: false
    },
    // Rango de ataque del jefe
    attack_range: {
        type: DataTypes.FLOAT,
        allowNull: false
    },
    // Tiempo de recarga entre ataques
    attack_cooldown: {
        type: DataTypes.FLOAT,
        allowNull: false
    },
    // Daño del primer ataque
    attack1_damage: {
        type: DataTypes.INTEGER,
        allowNull: false
    },
    // Daño del segundo ataque
    attack2_damage: {
        type: DataTypes.INTEGER,
        allowNull: false
    },
    // Rango de visión del jefe
    vision_range: {
        type: DataTypes.FLOAT,
        allowNull: false
    },
    // ID del ítem que deja al ser derrotado
    reward_item: {
        type: DataTypes.INTEGER
    }
});
```

5. Esquema de components de frontend (VUE)



Descripció dels Components

Components Principals

- **Dashboard.vue**: Component principal del panell d'administració que actua com a contenidor per als components específics del dashboard.
- **GameView.vue**: Component per visualitzar l'estat actual del joc.
- **login.vue**: Gestiona l'autenticació d'usuaris al sistema.
- **register.vue**: Gestiona el registre de nous usuaris.

Components del Dashboard

El dashboard està dividit en diversos components especialitzats per gestionar diferents aspectes del joc:

BossManagement.vue:
  - Gestió dels enemics tipus "boss" al joc
  - Configuració de les seves característiques i comportaments

EnemyDeathStats.vue:
  - Visualització d'estadístiques sobre la mort dels enemics
  - Gràfics i anàlisis de patrons de defunció

EnemyManagement.vue:
- Administració dels diferents tipus d'enemics
- Configuració de característiques, habilitats i comportaments

GameManagement.vue:
- Control de paràmetres globals del joc

ItemManagement.vue:
- Administració d'ítems i objectes dins del joc

PlayerManagement.vue:
- Gestió dels jugadors i les seves característiques
- Administració de perfils, progressió i habilitats

ShopManagement.vue:
- Control de la botiga virtual dins del joc
- Gestió d'articles, preus i disponibilitat de productes (incloent skins)

Relacions entre Components

Jerarquia de Components:
- `Dashboard.vue` actua com a contenidor principal que carrega els components específics del dashboard segons la navegació de l'usuari.
- Els components dins de la carpeta `dashboard` són subcomponents especialitzats que s'integren dins del dashboard principal.

Flux de Navegació:
- Els usuaris/administradors probablement comencen a `login.vue` o `register.vue`
- Després de l'autenticació, els administradors accedeixen a `Dashboard.vue` i el jugador/usuari accedeix al `GameView.vue` que es on pot iniciar el joc i començar a jugar.
- Des del dashboard, poden navegar als diferents components de gestió

Integració amb Vuetify:
- El nom del directori principal "vuetify-project" indica que el projecte utilitza Vuetify com a framework d'UI
- Els components probablement aprofiten els elements d'UI de Vuetify per mantenir una experiència d'usuari coherent

Gestió d'Estats

Encara que no es mostra explícitament en l'estructura de fitxers, el projecte probablement utilitza:

Vuex o **Pinia** per a la gestió d'estat global
Props i Events per a la comunicació entre components pare-fill
Vue Router per a la navegació entre components i pàgines

6. Documentació de codi frontend (VUE):

Plantilla Principal

```
<v-container fluid class="medieval-theme">
<v-card class="parchment-card" elevation="8">
  <!-- Contingut principal -->
</v-card>
</v-container>
```

La plantilla està estructurada amb Vuetify, un framework de components per a Vue basat en Material Design, però personalitzat amb un estil medieval.

Seccions Clau

1. Capçalera del Component

```
<v-card-title class="card-title d-flex align-center">
 <div class="title-wrapper">
   <v-icon large class="mr-2 crown-icon">mdi-shopping</v-icon>
   <span class="title-text">ROYAL SKIN EMPORIUM</span>
   <v-icon large class="ml-2 crown-icon">mdi-shopping</v-icon>
 </div>
 <v-spacer></v-spacer>
 <v-btn color="accent" class="refresh-btn"
@click="openCreateDialog">
   <span class="btn-text">NEW SKIN</span>
   <v-icon right>mdi-plus</v-icon>
 </v-btn>
</v-card-title>
```

La capçalera mostra el títol de l'aplicació i un botó per crear nous skins.

2. Visualització de Skins

```
<div class="items-container pa-4">
 <!-- Estat de càrrega -->
 <v-row v-if="loading" justify="center"
class="loading-container">
   <!-- Indicador de càrrega -->
 </v-row>

 <!-- Estat buit -->
 <v-row v-else-if="skins.length === 0" justify="center"
class="empty-container">
   <!-- Missatge d'estat buit -->
 </v-row>
```

```
 <!-- Llista de skins -->
 <v-row v-else>
   <v-col v-for="skin in skins" :key="skin.id" cols="12" sm="6"
md="4" lg="3">
     <!-- Targeta de skin individual -->
   </v-col>
 </v-row>
</div>
```

Aquesta secció gestiona tres estats possibles:

Càrrega: Es mostra un indicador de càrrega circular
Llista buida: Es mostra un missatge quan no hi ha skins disponibles
Llista de skins: Mostra els skins en una graella responsiva

3. Diàleg de Creació/Edició

```
<v-dialog v-model="dialog" max-width="600px">
 <v-card class="parchment-card dialog-card">
   <!-- Formulari de creació/edició de skins -->
 </v-card>
</v-dialog>
```

Aquest diàleg conté un formulari extens per a la creació o edició de skins, amb camps per a:

Nom del skin
Preu
Descripció
Imatges per a diferents animacions (Attack, Dash, Death, Fall, Idle, Run, Jump, TakeHit)
Imatge de portada

4. Diàleg de Confirmació d'Eliminació

```
<v-dialog v-model="dialogDelete" max-width="500px">
 <v-card class="parchment-card dialog-card">
   <!-- Confirmació d'eliminació -->
 </v-card>
</v-dialog>
```

Finestra modal que demana confirmació abans d'eliminar un skin.

Gestió de Dades

```
data() {
   return {
     skins: [],
     dialog: false,
     dialogDelete: false,
```

```
    valid: false,
    loading: false,
    editedIndex: -1,
    isEditing: false,
    oldSkinName: null,
    animationFiles: {
      Attack: null,
      Dash: null,
      Death: null,
      Fall: null,
      Idle: null,
      Run: null,
      Jump: null,
      TakeHit: null
    },
    coverFile: null,
    previewCover: null,
    snackbar: {
      show: false,
      text: '',
      color: '',
      icon: ''
    },
    editedItem: {
      id: null,
      name: '',
      price: 0,
      description: '',
      imageUrl: ''
    },
    defaultItem: {
      id: null,
      name: '',
      price: 0,
      description: '',
      imageUrl: ''
    },
    imageRules: [
      v => !v || v.size < 2000000 || 'Image size should be less
than 2 MB!'
    ]
  }
}
```

Mètodes Principals

Càrrega de Skins

```
async loadSkins() {
     this.loading = true;
     try {
       const response = await
fetch(`${import.meta.env.VITE_SHOP_API_URL}shop`)
       if (!response.ok) throw new Error('Error loading skins')
       const rawSkins = await response.json()
       this.skins = rawSkins.map(skin => ({
         ...skin,
         imageUrl:
`${import.meta.env.VITE_SHOP_API_URL}imagenes/shop/Portadas/${ski
n.skin_name.toLowerCase()}.jpg`
       }))


       this.showNotification('Skins summoned successfully',
'success', 'mdi-check-circle');
     } catch (error) {
       console.error('Error loading skins:', error)
       this.showNotification('Failed to summon skins', 'error',
'mdi-alert');
     }
     this.loading = false;
   }
```

Aquest mètode carrega la llista de skins des del servidor i configura les URLs de les imatges.

Creació de Skin

```
async createSkin() {
     // Validar que todas las imágenes estén presentes (solo para
creación)
     const requiredKeys =
['Attack','Dash','Death','Fall','Idle','Run','Jump','TakeHit'];
     for (const key of requiredKeys) {
       if (!this.animationFiles[key]) {
         this.showNotification(`Missing image: ${key}`, 'error',
'mdi-alert');
         throw new Error(`Missing image: ${key}`);
       }
     }
```

```
      if (!this.coverFile) {
        this.showNotification('Missing cover image', 'error',
'mdi-alert');
        throw new Error('Missing cover image');
      }

      const formData = new FormData();
      formData.append('targetFolder', this.editedItem.name);

      // Ordenar las imágenes según los sufijos del backend
      requiredKeys.forEach(key => {
        formData.append('images', this.animationFiles[key]);
      });
      formData.append('images', this.coverFile); // La portada
debe ir la última

      const response = await
fetch(`${import.meta.env.VITE_SHOP_API_URL}shop/upload-images`, {
        method: 'POST',
        body: formData
      });

      if (!response.ok) throw new Error('Error uploading images');
      this.showNotification('Skin images uploaded successfully',
'success', 'mdi-check-circle');

      // Ahora crear la skin en la base de datos
      const skinData = {
        skin_name: this.editedItem.name,
        price: this.editedItem.price,
        description: this.editedItem.description,
        image_url:
`/imagenes/shop/Portadas/${this.editedItem.name}.jpg`,
        is_available: true
      };

      const dbResponse = await
fetch(`${import.meta.env.VITE_SHOP_API_URL}shop`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(skinData)
      });
```

```
      if (!dbResponse.ok) throw new Error('Error creating skin in
DB');
      this.showNotification('New skin crafted successfully',
'success', 'mdi-check-circle');
    }
```

Aquest mètode gestiona la creació d'un nou skin, enviant primer les imatges i després creant el registre a la base de dades.

Actualització de Skin

```
async updateSkin() {
      // Verificar qué imágenes se han cambiado
      const requiredKeys =
['Attack','Dash','Death','Fall','Idle','Run','Jump','TakeHit'];
      const hasChangedImages =
Object.values(this.animationFiles).some(file => file !== null) ||
this.coverFile !== null;

      // Si hay cambios en las imágenes, subir las nuevas
      if (hasChangedImages) {
        const formData = new FormData();
        formData.append('targetFolder', this.editedItem.name);
        formData.append('oldSkinName', this.oldSkinName);

        // Añadir las imágenes que se han cambiado
        let filesCount = 0;
        requiredKeys.forEach(key => {
          if (this.animationFiles[key]) {
            formData.append('images', this.animationFiles[key]);
            filesCount++;
          }
        });

        // Añadir la portada si se ha cambiado
        if (this.coverFile) {
          formData.append('images', this.coverFile);
          filesCount++;
        }

        // Solo enviar la petición si hay archivos para actualizar
        if (filesCount > 0) {
          const response = await
fetch(`${import.meta.env.VITE_SHOP_API_URL}shop/update-images/${t
his.editedItem.id}`, {
```

```
            method: 'POST',
            body: formData
          });

          if (!response.ok) throw new Error('Error updating
images');

          const data = await response.json();
          if (data.newImageUrl) {
            this.editedItem.imageUrl = data.newImageUrl;
          }

          this.showNotification('Skin images updated
successfully', 'success', 'mdi-check-circle');
        }
      }

      // Actualizar los datos en la base de datos
      const skinData = {
        skin_name: this.editedItem.name,
        price: this.editedItem.price,
        description: this.editedItem.description,
        image_url: this.editedItem.imageUrl
      };

      // Si el nombre ha cambiado, actualizar la URL de la imagen
      if (this.oldSkinName !== this.editedItem.name) {
        skinData.image_url =
`/imagenes/shop/Portadas/${this.editedItem.name}.jpg`;
      }

      const dbResponse = await
fetch(`${import.meta.env.VITE_SHOP_API_URL}shop/${this.editedItem
.id}`, {
        method: 'PUT',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(skinData)
      });

      if (!dbResponse.ok) throw new Error('Error updating skin in
DB');
      this.showNotification('Skin updated successfully',
'success', 'mdi-check-circle');
```

```
    }
```

Aquest mètode gestiona l'actualització d'un skin existent, actualitzant només les imatges que s'han canviat.

Eliminació de Skin

```
async deleteItemConfirm() {
     try {
       const response = await
fetch(`${import.meta.env.VITE_SHOP_API_URL}shop/${this.editedItem
.id_skin}`, {
         method: 'DELETE'
       })
       if (!response.ok) throw new Error('Error deleting skin')
       this.showNotification('Skin banished successfully',
'success', 'mdi-check-circle');
       this.loadSkins()
     } catch (error) {
       console.error('Error deleting skin:', error)
       this.showNotification('Failed to banish skin', 'error',
'mdi-alert');
     }
     this.closeDelete()
   }
```

Aquest mètode elimina un skin del sistema.

Característiques Visuals

El component utilitza un disseny medieval amb:

Fons de pergamí: Textura de paper antic
Vores adornades: Colors i estils que recorden a l'època medieval
Tipografia temàtica: 'Cinzel' i 'Philosopher' per donar l'aspecte medieval
Indicadors de raresa: Colors diferents per mostrar la raresa dels skins segons el preu
Animacions: Transicions suaus en passar el cursor per sobre de les targetes

```
css.parchment-card {
 background-color: #2c2421 !important;
 background-image:
url('https://www.transparenttextures.com/patterns/parchment.png')
;
 border: 3px solid #704214 !important;
 border-radius: 8px !important;
}

.rarity-legendary {
 background: linear-gradient(90deg, #ff8c00, #ffd700, #ff8c00);
```

```
 background-size: 200% 100%;
 animation: gradient-shift 2s ease infinite;
}
```

Funcionalitats Destacades

Gestió Completa de Skins:

Creació, lectura, actualització i eliminació (CRUD)
Visualització en una interfície atractiva

Gestió d'Imatges:

Pujada de múltiples imatges per a diferents animacions
Previsualització d'imatges abans de guardar
Actualització selectiva d'imatges (només les que canvien)

Validació de Formulari:

Comprovació de camps obligatoris
Validació de mida de fitxers

Interfície Responsiva:

Adaptació a diferents mides de pantalla amb el sistema de graella de Vuetify

Notificacions:

Sistema de notificacions per informar a l'usuari sobre l'estat de les operacions

7. Documentació de codi backend:

1. Arquitectura General

La aplicación implementa una arquitectura de microservicios donde cada funcionalidad principal está separada en servicios independientes, cada uno corriendo en su propio puerto. El serviceController.js actúa como un orquestador de servicios.

Estructura de Servicios

```
const services = {
   player: {
       name: 'Servicio de Jugadores',
       port: 3001,
```

```
        script: 'services/playerService.js'
    },
    game: {
        name: 'Servicio de Partidas',
        port: 3002,
        script: 'services/gameService.js'
    },
    // ... más servicios
};
```

2. Componentes Principales

2.1 Gestión de Procesos

```
const { spawn } = require('child_process');
const runningServices = {};  // Mantiene registro de servicios activos
```

Utiliza child_process.spawn para crear procesos hijo independientes
Cada servicio corre en su propio proceso
Permite gestión independiente de cada servicio

2.2 Sistema de Logging

```
function logMessage(serviceType, message) {
 const timestamp = new Date().toISOString();
 const logFile = path.join(LOG_DIR, `${serviceType}.log`);
 const logEntry = `${timestamp} - ${message}\n`;

 fs.appendFileSync(logFile, logEntry);
 console.log(`[${serviceType}] ${message}`);
}
```

Logging centralizado para todos los servicios
Archivos de log separados por servicio
Timestamps para cada entrada

2.3 Gestión de Estados

```
async function checkServiceStatus(serviceType) {
 if (!services[serviceType]) {
   throw new Error(`Servicio ${serviceType} no encontrado`);
 }

 const serviceConfig = services[serviceType];
 let status = 'OFFLINE';
```

```
 if (runningServices[serviceType]?.pid) {
   try {
     process.kill(runningServices[serviceType].pid, 0);
     status = 'ONLINE';
   } catch {
     runningServices[serviceType] = null;
   }
 }

 if (status === 'OFFLINE') {
   const inUse = await isPortInUse(serviceConfig.port);
   if (inUse) status = 'ONLINE';
 }

 return {
   id: Object.keys(services).indexOf(serviceType) + 1,
   name: serviceConfig.name,
   status,
   port: serviceConfig.port,
   serviceType
 };
}
```

Monitoreo en tiempo real del estado de cada servicio
Verificación de procesos activos
Verificación de puertos en uso

3. Funcionalidades Clave

3.1 Inicio de Servicios

```
async function startService(serviceType) {
 const serviceConfig = services[serviceType];
 const status = await checkServiceStatus(serviceType);

 if (status.status === 'ONLINE') return status;

 if (!fs.existsSync(serviceConfig.script)) {
   throw new Error(`El archivo ${serviceConfig.script} no
existe`);
 }

 const child = spawn('node', [serviceConfig.script], {
   env: { ...process.env, [`${serviceType.toUpperCase()}_PORT`]:
serviceConfig.port },
```

```
    stdio: 'pipe',
    detached: true
  });

  runningServices[serviceType] = child;

  child.stdout.on('data', (data) => logMessage(serviceType,
data.toString().trim()));
  child.stderr.on('data', (data) => logMessage(serviceType,
`ERROR: ${data.toString().trim()}`));
  child.on('close', (code) => {
    logMessage(serviceType, `Proceso terminado con código
${code}`);
    runningServices[serviceType] = null;
    // Emitir cambio de estado al cerrarse el proceso
    checkServiceStatus(serviceType).then(status => {
      io.emit('service-status-changed', status);
    });
  });

  logMessage(serviceType, `Servicio iniciado en puerto
${serviceConfig.port}`);
  await new Promise(resolve => setTimeout(resolve, 1000));
  const newStatus = await checkServiceStatus(serviceType);
  io.emit('service-status-changed', newStatus); // Emitir cambio
de estado
  return newStatus;
}
```

Inicio asíncrono de servicios
Inyección de variables de entorno
Manejo de salida estándar y errores
Detección automática de cierre de servicios

3.2 Detención de Servicios

```
async function stopService(serviceType) {
  const process = runningServices[serviceType];
  if (!process) {
    const status = await checkServiceStatus(serviceType);
    io.emit('service-status-changed', status);
    return status;
  }
  return new Promise((resolve, reject) => {
    process.once('close', async () => {
```

```
      runningServices[serviceType] = null;
      const status = await checkServiceStatus(serviceType);
      io.emit('service-status-changed', status);
      resolve(status);
    });
    try {
      process.kill('SIGTERM');
    } catch (err) {
      reject(err);
    }
  });
}
```

Detención graciosa de servicios
Limpieza de recursos
Notificación de cambios de estado

3.3 Comunicación en Tiempo Real

```
const server = http.createServer(app);
const io = new Server(server, { cors: { origin: "*" } });

// Al conectar un cliente, enviar el estado de todos los
servicios
io.on('connection', async (socket) => {
 const statuses = await Promise.all(
   Object.keys(services).map(serviceType =>
checkServiceStatus(serviceType))
 );
 socket.emit('all-services-status', statuses);
});
```

Implementación de WebSockets para actualizaciones en tiempo real
Notificaciones de cambios de estado
Soporte CORS para comunicación cross-origin

4. API REST para Gestión

// Obtener estado de todos los servicios

```
app.get('/services', async (req, res) => {
```

// Iniciar un servicio

```
app.post('/services/:serviceType/start', checkServiceExists,
async (req, res) => {
```

// Detener un servicio

```
app.post('/services/:serviceType/stop', checkServiceExists, async
(req, res) => {
```

5. Características de Alta Disponibilidad

·Monitoreo Continuo
·Verificación constante del estado de los servicios
·Detección automática de fallos
·Gestión de Errores
·Manejo robusto de excepciones
·Logging detallado de errores
·Reinicio automático de servicios caídos
·Escalabilidad
·Servicios independientes que pueden escalar horizontalmente
·Puertos configurables mediante variables de entorno
·Esta implementación permite una arquitectura flexible, mantenible y escalable, donde cada servicio puede ser desarrollado, desplegado y escalado de manera independiente.

8. Desplegament Contenidors

```
version: "3.8"

services:

 # Interfaz web para gestionar bases de datos (similar a phpMyAdmin)
 adminer:
   image: adminer
   container_name: trFinal_adminer
   restart: always
   ports:
     - "8080:8080"  # Puerto para acceder desde el navegador:
http://localhost:8080
   networks:
     - trFinal_network

 # Contenedor de la base de datos MySQL 8.0
 mysql:
   image: mysql:8.0
   container_name: mysql
   restart: always
   environment:
     MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}    # Contraseña del
root
     MYSQL_DATABASE: ${MYSQL_DATABASE}              # Nombre de la base
de datos
```

```
    MYSQL_USER: ${MYSQL_USER}                    # Usuario de la base
de datos
    MYSQL_PASSWORD: ${MYSQL_PASSWORD}            # Contraseña del
usuario
  ports:
    - "3310:3306"  # Mapea el puerto 3306 del contenedor al 3310 del
host
  volumes:
    - ./mysql_data:/var/lib/mysql  # Persistencia de datos de MySQL
  networks:
    - trFinal_network

 # Backend de la aplicación (Node.js + librerías de Python)
 backend:
  image: node:18-alpine
  container_name: trFinal_backend
  restart: always
  ports:
    - "3000:3000"
    - "3001:3001"
    - "3002:3002"
    - "3003:3003"
    - "3004:3004"
    - "3005:3005"
    - "3006:3006"
    - "3007:3007"
    - "3008:3008"
    - "3009:3009"
    - "3010:3010"
    # Se exponen múltiples puertos, posiblemente para microservicios o
módulos independientes
  working_dir: /app/backend  # Directorio de trabajo dentro del
contenedor
  volumes:
    - ../backend:/app/backend           # Código fuente del backend
    - /app/backend/node_modules         # Evita conflictos con
dependencias del host
  depends_on:
    - mysql  # Espera a que la base de datos esté disponible
  command: >
    sh -c "
      apk add --no-cache python3 py3-pip py3-numpy py3-matplotlib
py3-pandas py3-seaborn py3-setuptools py3-wheel py3-pymysql &&
```

```
      sleep 5 &&
      npm install &&
      npm run dev
    "  # Instala dependencias Python, luego ejecuta el backend en modo
desarrollo
  networks:
    - trFinal_network

 # Frontend de la aplicación (Vue.js con Vuetify)
 frontend:
  image: node:18-alpine
  container_name: trFinal_frontend
  restart: always
  ports:
    - "4000:4000"  # Puerto del servidor de desarrollo de Vue
  working_dir: /app/frontend
  volumes:
    - ../frontend/vuetify-project:/app/frontend  # Código fuente del
frontend
    - /app/frontend/node_modules                 # Evita conflictos
con dependencias del host
  depends_on:
    - backend  # Espera a que el backend esté activo
  command: >
    sh -c "
      sleep 6 &&
      npm install &&
      npm run dev
    "  # Instala dependencias de Node y ejecuta el frontend en modo
desarrollo
  networks:
    - trFinal_network

# Red compartida entre todos los servicios
networks:
 trFinal_network:
  driver: bridge

# Volumen para persistencia de datos de la base de datos MySQL
volumes:
 mysql_data:
```

9. Unity

Documentació Unity con deepwiki:
Descripció de l'estructura tècnica del videojoc i dels seus components principals:

Documentació Unity completa: **DeepWiki**

Aquest apartat inclou una descripció detallada d'alguns dels scripts més rellevants utilitzats en el desenvolupament del joc. S'hi explica la seva funcionalitat, com s'integren dins l'arquitectura general del projecte i el paper que tenen en la jugabilitat o el funcionament intern. L'objectiu és proporcionar una visió clara de la lògica del joc i facilitar-ne tant la comprensió com el manteniment.

1. PlayerController

Aquest script controla tot el comportament del personatge jugable en un joc 2D fet amb Unity, integrant moviment, accions, combat, ús de màgia, sons i gestió de la UI. El codi utilitza el sistema nou d'inputs d'Unity, permetent que el jugador es mogui, salti, ataqui amb espasa, faci dash i llanci boles de foc (que consumeixen manà). La classe gestiona variables internes com la salut, el manà, les monedes recollides i l'estat del personatge (viu, mort, atordit, fent dash, etc.), actualitzant constantment tant les barres de vida i manà com la UI de monedes.

El moviment es calcula amb física (Rigidbody2D) i s'adapta a l'orientació del personatge, activant animacions i sons segons si corre, salta, cau o ataca. El salt només es pot fer si el personatge està a terra, i el dash és una acceleració ràpida amb efecte visual i sonor, limitada per cooldown. L'atac activa una hitbox en la direcció correcta i pot danyar enemics, caps o trencar parets falses, tot gestionant les col·lisions via Physics2D. Els atacs i la màgia tenen cooldown per evitar abús.

Quan el jugador rep dany, es redueix la seva vida i, si arriba a zero, es desactiven els controls, es mostra la pantalla de mort i, després d'uns segons, es reestableix la posició, la salut i el manà, amagant la UI de mort i tornant el control al jugador. El sistema de manà es regenera automàticament, però llençar boles de foc el consumeix, impedint-ne l'ús si no n'hi ha prou.

El codi també gestiona la recollida de monedes i la seva visualització, la interacció amb trampes (com punxes) i l'empenta del personatge quan rep cops forts. Tot plegat, l'script sincronitza estats, animacions, efectes i dades per garantir una experiència fluida i coherent, centralitzant tot el cicle de vida i accions del jugador dins el joc.

Document: **playerController.js**

2. BossTutorial

Aquest script és el que dona vida als enemics finals d'escena ("boss") del meu joc. M'encarrego que pugui moure's, detectar el jugador, atacar-lo de dues maneres diferents segons la seva vida, mostrar la barra de vida i el nom a la UI, i fins i tot actualitzar les seves estadístiques des d'un servidor web sense necessitat de recompilar el joc. Quan el jugador entra dins el seu radi de visió, el boss comença a moure's cap a ell, activa la música especial i mostra totes les UI corresponents.

Si el jugador s'apropa prou, el boss l'ataca (amb animacions, hitbox i dany real), i si li queda poca vida canvia el tipus d'atac. He volgut fer-ho molt visual: la barra de vida es va actualitzant, el nom canvia de color segons la salut, i quan el boss mor, es reprodueix una animació, es registra la seva derrota a una API externa, i es desintegra gradualment abans de desaparèixer del joc. També he afegit la possibilitat de reiniciar la seva salut per a noves partides. Tot plegat fa que el boss sigui molt dinàmic, reactiu i fàcilment configurable sense tocar el codi base.

Document: BossTutorial.js

3. EnemyAI

En aquest script vaig crear la base per a la intel·ligència artificial dels enemics del meu joc. Aquesta classe abstracta `EnemyAI` defineix el comportament genèric de tots els enemics: poden detectar el jugador, moure's cap a ell, atacar-lo quan estan a prop i rebre danys o morir. La configuració d'estadístiques (velocitat, vida, abast de seguiment i atac, etc.) és fàcilment editable des de l'Inspector o a través d'un objecte de dades.

El sistema controla animacions segons l'estat de l'enemic (moviment, rebre dany, mort), i gestiona la física i la col·lisió quan moren. Quan la vida arriba a zero, es reprodueix l'animació de mort, es desactiven components físics, es registra la mort a una API i, passats uns segons, l'enemic es desactiva automàticament. També he afegit funcions per reiniciar l'enemic, útil per a partides o nivells nous. Aquesta estructura em permet crear fàcilment subclasses per a tipus d'enemics amb comportaments d'atac especials, mantenint el codi flexible i escalable.

Document: EnemyAI.js

4. BarraVida

Aquest script gestiona la vida del jugador en un joc fet amb Unity, mostrant visualment la barra de vida, permetent la curació mitjançant pocions i sincronitzant l'estat amb una base de dades externa. Controla tant la part visual (barra de vida), com els efectes (partícules i so) i la connexió amb el backend per validar i consumir objectes del jugador.

En iniciar, l'script recupera el player_id i el nickname del jugador mitjançant PlayerPrefs, assigna la vida màxima, i inicialitza la barra de vida. A cada frame (Update), detecta si el jugador prem la tecla 1 per intentar utilitzar una poció de vida.

Quan es detecta aquesta acció, l'script fa una petició GET a un servidor local (http://localhost:3003/inventory/{nickname}) per comprovar si el jugador té una poció (ítem amb ID 2). Si existeix, es fa una petició DELETE (/removeItem) amb dades JSON per eliminar una unitat de la poció del seu inventari. Si tot va bé, es curen 50 punts de vida, amb límit de vida màxima.

El procés de curació inclou:

L'activació d'un sistema de partícules (healingEffect)

La reproducció d'un efecte sonor (healSound)

L'actualització visual de la barra (healthBar) en funció de la nova vida

Tot el sistema està dissenyat perquè sigui reactiu i modular, connectant aspectes visuals, de jugabilitat i de comunicació amb el servidor. També inclou una classe auxiliar JsonData1 per estructurar correctament les dades a enviar via JSON.

Document: barraVida.js

5. Fireball

Aquest script controla el comportament d'una bola de foc llançada pel jugador dins d'un joc en 2D amb Unity. Integra moviment, col·lisions, efectes visuals i sonors, i interacció amb enemics i caps. És un sistema modular i reutilitzable que permet configurar la bola de foc des del PlayerController o qualsevol altre component.

Quan es genera una bola de foc, aquesta es mou en línia recta (sense gravetat) en la direcció establerta (dreta o esquerra), amb una velocitat configurable. El Rigidbody2D està configurat com a kinematic per controlar el moviment manualment, i el collider està configurat com a trigger per detectar col·lisions sense activar la física de rebot. Després d'un temps (lifetime), la bola de foc es destrueix automàticament si no ha col·lisionat amb res.

L'script permet la inicialització dinàmica amb paràmetres com la direcció, la velocitat i el dany. A més, la bola de foc pot ser disparada mitjançant el mètode estàtic ShootFireball, que genera una instància de la prefab i l'inicialitza amb la direcció en què està mirant el jugador (isFacingRight).

Quan la bola impacta amb un objecte:

Si té el tag "Enemy" o "Boss", li aplica dany mitjançant la interfície IDamageable o cridant directament el mètode TakeDamage.

Si col·lideix amb el terra ("Ground") o un altre objecte sòlid, es destrueix sense aplicar dany.

No afecta el jugador ni altres triggers.

A l'impactar, la bola de foc:

Reprodueix un efecte visual (impactEffectPrefab)

Emet un so (impactSound)

Es destrueix automàticament

Aquest script facilita la reutilització mitjançant una interfície (IDamageable) per permetre a diversos enemics rebre dany de forma genèrica. També assegura una sincronització fluida entre moviment, col·lisions, i efectes visuals i auditius, enriquint l'experiència de combat.

Document: Fireball.cs

6. Checkpoint

Aquest script gestiona el sistema de punts de control (checkpoints) dins d'un joc 2D fet amb Unity. Està associat a una "hoguera" (campfire) que, en interactuar-hi, permet al jugador guardar la partida manualment prement la tecla C quan es troba a prop. El sistema integra l'estat del jugador, la seva posició, salut, monedes, l'escena actual i l'estat dels enemics en el moment de la interacció.

Quan el jugador entra a l'àrea de col·lisió de la hoguera, la variable isNearCheckpoint s'activa, i si es prem la tecla C, es crida el mètode SaveGame(), que:

Regenera la salut del jugador al màxim.

Guarda la posició actual del jugador (PlayerPrefs.SetFloat("position_x/y")).

Desa la salut, monedes i nom de l'escena actual.

Desa l'estat de tots els enemics (IsDead i posició inicial).

Encén l'animació de la hoguera (si encara no estava encesa) i guarda el seu estat encès.

Desa tota la informació a través de PlayerPrefs, i opcionalment a través del script GuardarPartida, si existeix a l'escena.

També té un sistema de restauració dels enemics, reposicionant-los i reiniciant-ne l'estat mitjançant el mètode ResetEnemies() cada vegada que es guarda.

A més, detecta l'entrada i sortida del jugador a la zona de col·lisió de la hoguera a través dels mètodes OnTriggerEnter2D i OnTriggerExit2D, que activen o desactiven la possibilitat de guardar.

Aquest sistema ofereix una forma eficient de gestionar els punts de control, essencial per a la persistència en jocs d'acció o plataformes, combinant elements visuals, sonors i de jugabilitat.

Document: Checkpoint.cs

10. Python

Entorn

L'script està pensat per executar-se en un entorn Python i utilitza diverses llibreries:

- pandas: per a la manipulació i anàlisi de dades.
- seaborn i matplotlib: per a la generació de gràfics.
- sqlalchemy: per a la connexió i consulta a la base de dades MySQL.
- os i datetime: per a la gestió de rutes i dates.
- Les credencials de la base de dades s'han de definir com a variables d'entorn (DB_NAME, DB_USER, DB_PASS, DB_HOST, DB_PORT).

Connexió a la base de dades

La connexió es realitza mitjançant SQLAlchemy, utilitzant un motor MySQL. Les credencials es recuperen de les variables d'entorn i es construeix la cadena de connexió. L'script executa una consulta SQL per obtenir tots els registres de la taula enemy_death_stat i desa el resultat en un DataFrame de pandas.

Tractament de les dades

- Es consulta la taula enemy_death_stat i es desa en un DataFrame.
- L'script comprova si el DataFrame té dades. Si està buit, genera una imatge indicant que no hi ha dades.
- Si hi ha dades, es processen les columnes rellevants (per exemple, noms d'enemics, dates de mort, noms de caps) per generar diferents tipus de gràfics.
- Es busquen columnes de data amb noms habituals (fecha, date, created_at, death_time) i es converteixen a tipus datetime per a l'anàlisi temporal.

Tipus de gràfics

L'script genera diversos tipus de gràfics i els desa com a imatges:

- Gràfic de barres: nombre de morts per cada tipus d'enemic.
- Gràfic de línia: morts per dia.
- Gràfic de pastís: morts de cada usuari.
- Gràfic de barres horitzontals: morts per Boss.

Cada gràfic es desa a la carpeta stat_images i l'script imprimeix la ruta on s'ha desat la imatge.

```
import os

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from datetime import datetime

from sqlalchemy import create_engine

# Load DB credentials from environment variables
```

```
DB_NAME = os.getenv('DB_NAME')

DB_USER = os.getenv('DB_USER')

DB_PASS = os.getenv('DB_PASS')

DB_HOST = os.getenv('DB_HOST')

DB_PORT = int(os.getenv('DB_PORT', 3306))

# Output directory for generated images

OUTPUT_DIR = os.path.join(os.path.dirname(__file__), '../stat_images')

os.makedirs(OUTPUT_DIR, exist_ok=True)

# Create SQLAlchemy engine for MySQL

engine =
create_engine(f"mysql+pymysql://{DB_USER}:{DB_PASS}@{DB_HOST}:{DB_PORT}
/{DB_NAME}")

# Query enemy death stats

df = pd.read_sql('SELECT * FROM enemy_death_stat', engine)

print(f"[enemy-death-stats] Filas encontradas en enemy_death_stat:
{len(df)}")

img_path = os.path.join(OUTPUT_DIR, 'deaths_per_enemy.png')

# Example plot: deaths per enemy

if not df.empty:

    plt.figure(figsize=(16, 9))

    ax = sns.countplot(data=df, x='enemy_name',
order=df['enemy_name'].value_counts().index)

    plt.title('Muertes por enemigo')

    plt.ylabel('Número de muertes')

    plt.xlabel('Enemigo')

    plt.xticks(rotation=45, ha='right', fontsize=13)

    plt.subplots_adjust(bottom=0.28)

    plt.tight_layout()
```

```
    plt.savefig(img_path, bbox_inches='tight')
    plt.close()
    print(f"[enemy-death-stats] Imagen guardada en: {img_path}")
else:
    # Guardar imagen vacía para comprobar que la ruta funciona
    plt.figure(figsize=(10, 5))
    plt.text(0.5, 0.5, 'No hay datos', fontsize=22, ha='center',
va='center')
    plt.axis('off')
    plt.tight_layout()
    plt.savefig(img_path)
    plt.close()
    print(f"[enemy-death-stats] No hay datos. Imagen vacía guardada en:
{img_path}")
# Gráfico 2: Muertes por día (línea)
if not df.empty:
    date_col = None
    for col in ['fecha', 'date', 'created_at', 'death_time']:
        if col in df.columns:
            date_col = col
            break
    if date_col:
        plt.figure(figsize=(10, 6))
        df[date_col] = pd.to_datetime(df[date_col])
        deaths_per_day = df.groupby(df[date_col].dt.date).size()
        import matplotlib.dates as mdates
        plt.figure(figsize=(16, 9))
        # Formatea las fechas como 'dd/mm' para el eje X
```

```
        formatted_dates = [d.strftime('%d/%m') for d in
deaths_per_day.index]
        plt.plot(formatted_dates, deaths_per_day.values, marker='o',
color='crimson', linewidth=2)
        plt.title('Muertes por día')
        plt.ylabel('Número de muertes')
        plt.xlabel('Día')
        plt.grid(True, linestyle='--', alpha=0.5)
        # Solo muestra cada N etiqueta, dependiendo de cuántas fechas
haya
        N = max(1, len(formatted_dates)//8)
        for i, label in enumerate(plt.gca().get_xticklabels()):
            if i % N != 0:
                label.set_visible(False)
        plt.xticks(rotation=45, ha='right', fontsize=13)
        plt.subplots_adjust(bottom=0.28)
        plt.tight_layout()
        img_path_day = os.path.join(OUTPUT_DIR, 'deaths_per_day.png')
        plt.savefig(img_path_day, bbox_inches='tight')
        plt.close()
        print(f"[enemy-death-stats] Imagen guardada en:
{img_path_day}")
    else:
        print("[enemy-death-stats] No se encontró columna de fecha para
muertes por día.")
# Gráfico 3: Muertes por usuario/jugador (tarta/pie)
if not df.empty:
    user_col = None
    for col in ['user_id', 'player_name', 'jugador', 'usuario',
```

```
'player_nickname']:
        if col in df.columns:
            user_col = col
            break
    if user_col:
        plt.figure(figsize=(8, 8))
        death_counts = df[user_col].value_counts()
        plt.figure(figsize=(12, 12))
        plt.pie(death_counts.values, labels=death_counts.index,
autopct='%1.1f%%', startangle=140, colors=sns.color_palette('pastel'),
textprops={'fontsize': 15})
        plt.title('Proporción de muertes por usuario/jugador')
        plt.tight_layout()
        img_path_user = os.path.join(OUTPUT_DIR, 'deaths_per_user.png')
        plt.savefig(img_path_user, bbox_inches='tight')
        plt.close()
        print(f"[enemy-death-stats] Imagen guardada en:
{img_path_user}")
    else:
        print("[enemy-death-stats] No se encontró columna de
usuario/jugador para muertes por usuario.")
# Gráfico 4: Muertes por boss (barras horizontales)
if not df.empty:
    boss_col = None
    for col in ['boss_name', 'jefe']:
        if col in df.columns:
            boss_col = col
            break
```

```
    if boss_col:
        plt.figure(figsize=(10, 6))
        death_counts = df[boss_col].value_counts()
        plt.figure(figsize=(16, 9))
        plt.barh(death_counts.index, death_counts.values,
color=sns.color_palette('muted'))
        plt.title('Muertes por boss')
        plt.xlabel('Número de muertes')
        plt.ylabel('Boss')
        plt.yticks(fontsize=13)
        plt.subplots_adjust(left=0.28)
        plt.tight_layout()
        img_path_boss = os.path.join(OUTPUT_DIR, 'deaths_per_boss.png')
        plt.savefig(img_path_boss, bbox_inches='tight')
        plt.close()
        print(f"[enemy-death-stats] Imagen guardada en:
{img_path_boss}")
    else:
        print("[enemy-death-stats] No se encontró columna de boss para
muertes por boss.")
```