Introducción a Phaser 3

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez
ruben.rodriguez@urjc.es
URJC

Natalia Madrueño
natalia.madrueno@urjc.es
URJC

2025-09-09

Universidad Rey Juan Carlos

DSLAB
DATA SCIENCE LABORATORY

Tabla de contenidos

Introducción a Phaser 3

¿Qué es Phaser 3?

Framework open source de HTML5 para videojuegos

- Juegos que se ejecutan **directamente en navegadores web**
- Liberado en **2018** - evolución de versiones anteriores
- Enfocado en **juegos 2D** multiplataforma

Ventajas:

- Sin instalación para usuarios finales
- Multiplataforma (desktop + móvil)
- Distribución simple vía web
- Basado en tecnologías estándar

Tecnologías:

- HTML5 Canvas
- JavaScript/TypeScript
- Canvas API / WebGL

Sistemas de Renderizado

Dos opciones disponibles:

- **Canvas (por defecto)**: Mayor compatibilidad, menos exigente
- **WebGL**: Gráficos avanzados, mejor rendimiento, requiere soporte GPU

¿Cuándo usar cada uno?

- Canvas: Juegos simples, máxima compatibilidad
- WebGL: Muchos objetos, efectos visuales complejos

```
// Configurar tipo de renderizado
const config = {
    type: Phaser.AUTO,  // Elige el mejor disponible
    // type: Phaser.CANVAS,  // Forzar Canvas
    // type: Phaser.WEBGL,   // Forzar WebGL
};
```

Requisitos para Empezar

Navegador moderno:

- Chrome (recomendado - mejores DevTools)
- Firefox, Safari, Edge u Opera

Obtener Phaser 3:

- **CDN** (más rápido)

```
<script src="//cdn.jsdelivr.net/npm/phaser@3.55.2/dist/phaser.js"></script>
```

- **Descarga directa** phaser.io/download/stable
- **GitHub** github.com/photonstorm/phaser
- **Node** `npm install phaser`

Estructura Básica

El Elemento Canvas

¿Qué es el Canvas?

- Etiqueta HTML5 (`<canvas>`) para dibujar gráficos
- Funciona como un **lienzo en blanco**
- Se manipula mediante JavaScript

Sistema de coordenadas:

- Origen **(0,0)** en esquina superior izquierda
- Eje X positivo → derecha
- Eje Y positivo → abajo (inverso a matemáticas)
- Dimensiones por defecto: 300x300 píxeles

```
1 <canvas id="game" width="800" height="600"></canvas>
```

Configuración Inicial

Objeto de configuración define parámetros fundamentales:

- Tipo de renderizado (AUTO/CANVAS/WEBGL)
- Dimensiones del juego
- Motor de físicas y gravedad
- Funciones principales del juego

```
const config = {
    type: Phaser.AUTO,      // Renderizado automático
    width: 800,             // Ancho del canvas
    height: 600,            // Alto del canvas
    physics: {
        default: 'arcade',  // Motor de físicas
        arcade: {
            gravity: { y: 300 }  // Gravedad vertical
        }
    },
    scene: {
        preload: preload,   // Función de carga
        create: create,     // Función de creación
        update: update      // Función de actualización
    }
};
```

Las Tres Funciones Principales

preload() - Cargar recursos

- **Cuándo**: Se ejecuta primero, solo una vez
- **Para qué**: Cargar imágenes, sonidos, sprites
- **Importante**: Phaser espera a que todo se cargue antes de continuar

```
function preload() {
    // Cargar recursos con identificadores únicos
    this.load.image('cielo', 'assets/cielo.png');
    this.load.image('suelo', 'assets/plataforma.png');
    this.load.image('estrella', 'assets/estrella.png');
}
```

Nota

El primer parámetro es el **identificador** que usaremos después

Las Tres Funciones Principales (2)

create() - Crear objetos del juego

- **Cuándo**: Se ejecuta después de preload()
- **Para qué**: Inicializar y posicionar elementos
- **Importante**: El orden de creación determina el orden de renderizado

```
function create() {
    // Añadir imagen de fondo centrada
    this.add.image(400, 300, 'cielo');

    // Crear plataforma con físicas
    this.plataforma = this.physics.add.image(400, 500, 'suelo');
    this.plataforma.setImmovable(true);  // No se mueve
}
```

Las Tres Funciones Principales (3)

update(time, delta) - Bucle del juego

- **Cuándo**: Se ejecuta constantemente (~60 veces/segundo)
- **Para qué**: Lógica que debe actualizarse continuamente
- **Parámetros**: `time` (tiempo total), `delta` (desde último frame)

```
function update(time, delta) {
    // Mover jugador con teclado
    if (this.cursors.left.isDown) {
        this.jugador.x -= 5;  // Mover izquierda
    }
    if (this.cursors.right.isDown) {
        this.jugador.x += 5;  // Mover derecha
    }
}
```

Tip

Usar `delta` para movimiento consistente en diferentes dispositivos

Gestión de Escenas

Concepto de Escena

¿Qué es una escena?

- Pantalla o estado independiente del juego
- Tiene su propio flujo de ejecución (preload, create, update)
- Mantiene sus propios recursos y objetos
- Permite organización modular del código

Ejemplos comunes:

- Menú principal
- Pantalla de gameplay
- Menú de pausa
- Pantalla de game over
- Tutorial
- Pantalla de créditos

Escena Implícita vs Explícita

Forma implícita (simple):

```
// Crea una escena anónima automáticamente
const config = {
    scene: {
        preload: preload,
        create: create,
        update: update
    }
};
```

Forma explícita (recomendada):

```
class MiEscena extends Phaser.Scene {
    preload() { /* ... */ }
    create() { /* ... */ }
    update() { /* ... */ }
}

const config = {
    scene: [MiEscena, OtraEscena]
};
```

Nota

Las funciones preload/create/update **siempre** están dentro de una escena

SceneManager - Métodos Disponibles

Cambiar entre escenas:

- `start()` : Inicia escena y **detiene la actual**
- `launch()` : Inicia escena **en paralelo** (mantiene actual)
- `stop()` : Detiene completamente una escena

Controlar estado:

- `pause()` / `resume()` : Pausar/reanudar (sigue visible)
- `sleep()` / `wake()` : Dormir/despertar (no visible)

Gestionar orden:

- `bringToTop()` / `sendToBack()` : Mover al frente/fondo
- `moveAbove()` / `moveBelow()` : Posicionar relativamente

Configurar Múltiples Escenas

Definir escenas en el array de configuración:

```
// La primera escena del array se inicia automáticamente
const config = {
    scene: [MenuPrincipal, Juego, GameOver]
};
```

Añadir/eliminar dinámicamente:

```
// Añadir nueva escena en tiempo de ejecución
this.scene.add('clave', ConfigEscena, autoStart, datos);

// Eliminar una escena
this.scene.remove('clave');
```

Cambiar Entre Escenas

start() vs launch():

```
// start: Detiene la actual y cambia
this.scene.start('Juego', { nivel: 1 });

// launch: Ejecuta en paralelo (útil para HUDs)
this.scene.launch('MenuPausa', { origenEscena: 'Juego' });

// stop: Detiene completamente
this.scene.stop('MenuPausa');
```

Importante

Diferencia clave: `start` reemplaza, `launch` superpone escenas

Pausar y Reanudar

pause() / resume():

- Detiene `update()` pero sigue renderizando
- Útil para pausar el juego manteniéndolo visible

```
// Pausar (sigue visible, no actualiza)
this.scene.pause('Juego');

// Reanudar
this.scene.resume('Juego');
```

sleep() / wake():

- Detiene `update()` y renderizado
- Más eficiente que pause

```
// Dormir (no visible, no actualiza)
this.scene.sleep('Fondo');

// Despertar
this.scene.wake('Fondo');
```

Sistema de Pausa - Ejemplo

Escena principal del juego:

```
class Juego extends Phaser.Scene {
    constructor() {
        super('Juego');
    }

    create() {
        // Configurar juego...

        // Detectar tecla ESC para pausar
        this.input.keyboard.on('keydown-ESC', () => {
            this.scene.pause();          // Pausar juego
            this.scene.launch('MenuPausa', {
                escenaOrigen: 'Juego'
            });
        });
    }
}
```

Sistema de Pausa - Ejemplo (2)

Escena del menú de pausa:

```
class MenuPausa extends Phaser.Scene {
    constructor() {
        super('MenuPausa');
    }

    create(datos) {
        // Recibir datos de la escena que pausó
        console.log('Pausado desde:', datos.escenaOrigen);

        // Crear fondo semitransparente
        let overlay = this.add.rectangle(400, 300, 800, 600, 0x000000, 0.7);

        // Botón continuar
        let botonContinuar = this.add.text(400, 300, 'Continuar', {
            fontSize: '32px'
        }).setOrigin(0.5);
        botonContinuar.setInteractive();

        botonContinuar.on('pointerdown', () => {
            this.scene.stop();                  // Cerrar menú
            this.scene.resume(datos.escenaOrigen);  // Reanudar juego
        });
    }
}
```

Pasar Datos Entre Escenas

Enviar datos al cambiar de escena:

```
// Pasar datos al iniciar una escena
this.scene.start('SiguienteEscena', {
    puntuacion: 100,
    nivel: 2,
    nombre: 'Jugador1'
});
```

Recibir datos en la escena destino:

```
class SiguienteEscena extends Phaser.Scene {
    create(datos) {
        // Recibir los datos como parámetro
        console.log(datos.puntuacion);  // 100
        console.log(datos.nivel);       // 2
        console.log(datos.nombre);      // 'Jugador1'

        this.add.text(100, 100, `Puntuación: ${datos.puntuacion}`);
    }
}
```

Pasar Datos - Otros Métodos

También funciona con launch() y otros:

```
// Con launch (escena en paralelo)
this.scene.launch('MenuPausa', {
    juegoActual: 'Nivel1',
    tiempoTranscurrido: 120
});

// Con transition
this.scene.transition({
    target: 'GameOver',
    duration: 1000,
    data: {
        puntuacionFinal: this.puntuacion,
        tiempoJugado: this.tiempo
    }
});
```

Transiciones Entre Escenas

Crear efectos visuales suaves:

- `duration` : Tiempo en milisegundos
- `onUpdate` : Callback durante transición
- `progress` : Valor de 0 a 1
- `data` : Objeto con datos para la escena destino

```
this.scene.transition({
    target: 'SiguienteEscena',
    duration: 1000,             // 1 segundo
    moveBelow: true,
    data: {                     // Datos a pasar
        puntuacion: this.puntos,
        nivel: this.nivelActual
    },
    onUpdate: (progress) => {
        // Crear efectos de fundido
        this.cameras.main.setAlpha(1 - progress);
    }
});
```

Reordenar Escenas

Control del orden de renderizado:

- Valores mayores se dibujan encima
- Útil para HUDs y menús superpuestos

```
// Operaciones básicas
this.scene.bringToTop('HUD');
this.scene.sendToBack('Fondo');

// Movimiento relativo
this.scene.moveAbove('A', 'B');   // A encima de B
this.scene.moveBelow('A', 'B');   // A debajo de B

// Movimiento incremental
this.scene.moveUp('Escena');
this.scene.moveDown('Escena');
```

Trabajo con Imágenes

Cargar y Mostrar

Proceso en dos pasos:

1. **Cargar** en `preload()` con identificador único
2. **Mostrar** en `create()` usando el identificador

```
function preload() {
    // Cargar con identificador único
    this.load.image('personaje', 'assets/personaje.png');
    this.load.image('fondo', 'assets/fondo.jpg');
}

function create() {
    // Añadir al canvas en posición (x, y)
    this.add.image(400, 300, 'fondo');
    let sprite = this.add.image(200, 150, 'personaje');
}
```

> ⚠ **Advertencia**
>
> Siempre cargar en `preload()` antes de usar en `create()`

Sistema de Coordenadas y Origen

Origen por defecto: centro de la imagen

- Valores de 0 a 1
- (0, 0) = esquina superior izquierda
- (0.5, 0.5) = centro (defecto)
- (1, 1) = esquina inferior derecha

```
// Cambiar punto de origen
let imagen = this.add.image(100, 100, 'personaje')
    .setOrigin(0, 0);  // Esquina superior izquierda

imagen.setOrigin(0.5, 1);  // Centro inferior
imagen.setOrigin(1, 0);     // Esquina superior derecha
```

Tip

Cambiar origen útil para alinear objetos o rotaciones específicas

Transformaciones Básicas

Escalar, voltear y rotar:

```
let jugador = this.add.image(400, 300, 'personaje');

// Escalar uniformemente
jugador.setScale(1.5);      // 150% del tamaño

// Escalar independientemente
jugador.setScale(2, 0.5);   // Ancho x2, alto x0.5

// Voltear
jugador.flipX = true;       // Espejo horizontal
jugador.flipY = true;       // Espejo vertical

// Rotar (en radianes)
jugador.rotation = Math.PI / 4;  // 45 grados
```

Nota

π radianes = 180 grados

Profundidad y Capas

Controlar qué se dibuja encima:

- La propiedad `depth` funciona como capas
- Valores mayores se dibujan sobre valores menores
- Por defecto todas tienen depth = 0

```
let fondo = this.add.image(400, 300, 'cielo');
fondo.depth = 0;  // Atrás

let jugador = this.add.image(400, 300, 'personaje');
jugador.depth = 10;  // Encima del fondo

let hud = this.add.image(400, 50, 'interfaz');
hud.depth = 100;  // Encima de todo
```

Motores de Físicas

Tres Motores Disponibles

Arcade Physics:

- Más simple y rápido
- Solo rectángulos y círculos
- Ideal para juegos arcade/plataformas

Impact Physics:

- Soporta pendientes en tiles
- Más complejo que Arcade
- Ideal para plataformas con terreno inclinado

Matter Physics:

Configurar Arcade Physics

Configuración en el objeto config:

```
const config = {
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 300 },  // Gravedad en píxeles/s²
            debug: false          // Mostrar contornos físicos
        }
    }
};
```

Tip

Activar `debug: true` durante desarrollo para ver colisiones

Añadir Físicas a Objetos

Diferencia entre objetos con y sin físicas:

```
1 // Sin físicas (imagen estática)
2 this.add.image(400, 300, 'fondo');
3
4 // Con físicas (puede moverse, colisionar, etc.)
5 this.jugador = this.physics.add.image(100, 450, 'personaje');
```

Importante

Usar `physics.add` en lugar de solo `add` para habilitar físicas

Propiedades Físicas

Configurar comportamiento físico:

```
this.jugador = this.physics.add.image(100, 450, 'personaje');

// No salir de los límites del canvas
this.jugador.setCollideWorldBounds(true);

// Rebote al chocar (0 = no rebota, 1 = rebote perfecto)
this.jugador.setBounce(0.3);

// Velocidad inicial (píxeles/segundo)
this.jugador.setVelocity(100, -50);

// Aceleración (píxeles/segundo²)
this.jugador.setAcceleration(50, 0);
```

Sistema de Entrada

Teclado - Eventos

Detectar teclas específicas:

```
function create() {
    // Detectar cuando se presiona
    this.input.keyboard.on('keydown-SPACE', () => {
        this.jugador.setVelocityY(-330); // Saltar
    });

    // Detectar cuando se suelta
    this.input.keyboard.on('keyup-SPACE', () => {
        console.log('Tecla soltada');
    });
}
```

Nota

`keydown` se dispara al presionar, `keyup` al soltar

Teclado - Modificadores

Detectar combinaciones de teclas:

```
this.input.keyboard.on('keydown-A', (event) => {
    if (event.ctrlKey) {
        console.log('CTRL + A');
    } else if (event.shiftKey) {
        console.log('SHIFT + A');
    } else if (event.altKey) {
        console.log('ALT + A');
    } else {
        console.log('Solo A');
    }
});
```

Teclado - Cursores

Objeto para flechas, espacio y shift:

```
function create() {
    // Crear objeto de cursores
    this.cursors = this.input.keyboard.createCursorKeys();
}

function update() {
    // Resetear velocidad
    this.jugador.setVelocityX(0);

    // Comprobar teclas presionadas
    if (this.cursors.left.isDown) {
        this.jugador.setVelocityX(-160);
    } else if (this.cursors.right.isDown) {
        this.jugador.setVelocityX(160);
    }

    // Saltar solo si está en el suelo
    if (this.cursors.up.isDown && this.jugador.body.touching.down) {
        this.jugador.setVelocityY(-330);
    }
}
```

Teclado - Combos

Detectar secuencias de teclas:

- Útil para códigos secretos o trucos
- Detecta automáticamente la secuencia correcta

```
function create() {
    // Combo con letras
    let combo1 = this.input.keyboard.createCombo('KONAMI');

    // Combo Konami: ↑ ↑ ↓ ↓ ← → ← →
    let combo2 = this.input.keyboard.createCombo(
        [38, 38, 40, 40, 37, 39, 37, 39]
    );

    // Detectar cuando se completa
    this.input.keyboard.on('keycombomatch', (combo) => {
        console.log('¡Código secreto desbloqueado!');
        this.activarModoEspecial();
    });
}
```

Ratón y Táctil

API unificada para ratón y pantallas táctiles:

```
function create() {
    // Detectar clic/toque
    this.input.on('pointerdown', (pointer) => {
        if (pointer.leftButtonDown()) {
            console.log('Clic izquierdo:', pointer.x, pointer.y);
        }
        if (pointer.rightButtonDown()) {
            console.log('Clic derecho');
        }
    });

    // Detectar cuando se suelta
    this.input.on('pointerup', (pointer) => {
        console.log('Botón soltado');
    });

    // Detectar movimiento
    this.input.on('pointermove', (pointer) => {
        console.log('Posición:', pointer.x, pointer.y);
    });
}
```

Detección de Colisiones

Collider vs Overlap

Dos formas de detectar interacciones:

Collider:

- Detecta **y resuelve** colisiones físicamente
- Separa objetos automáticamente
- Aplica masa, velocidad, rebote

Overlap:

- Solo **detecta** superposición espacial
- Los objetos pueden atravesarse
- Sin resolución física

> **Tip**
>
> Collider para plataformas, Overlap para coleccionables

Ejemplos de Uso

Collider para plataformas:

```
function create() {
    let jugador = this.physics.add.image(100, 450, 'personaje');

    let plataforma = this.physics.add.image(400, 500, 'suelo');
    plataforma.setImmovable(true);  // No se mueve al impacto

    // Colisión con resolución física
    this.physics.add.collider(jugador, plataforma);
}
```

Overlap para coleccionables:

```
let estrella = this.physics.add.image(200, 200, 'estrella');

this.physics.add.overlap(jugador, estrella, (j, e) => {
    e.destroy();  // Eliminar estrella
    this.puntuacion += 10;
});
```

Grupos Estáticos

Para objetos inmóviles (plataformas, paredes):

- Más eficientes que objetos dinámicos
- No se mueven ni responden a físicas
- Ideales para nivel/escenario

```
function create() {
    // Crear grupo estático
    let plataformas = this.physics.add.staticGroup();

    // Añadir plataformas
    plataformas.create(400, 568, 'suelo')
        .setScale(2)
        .refreshBody();  // Actualizar tras modificar

    plataformas.create(600, 400, 'suelo');
    plataformas.create(50, 250, 'suelo');

    // Colisión con todas
    this.physics.add.collider(this.jugador, plataformas);
}
```

⚠ **Advertencia**

Grupos Dinámicos

Para objetos con físicas completas:

- Se mueven y responden a gravedad
- Pueden colisionar entre sí
- Más costosos computacionalmente

```
// Crear 12 estrellas espaciadas
let estrellas = this.physics.add.group({
    key: 'estrella',
    repeat: 11,  // 1 + 11 = 12 total
    setXY: { x: 12, y: 0, stepX: 70 }  // Cada 70px
});

// Aplicar propiedades a cada una
estrellas.children.iterate((estrella) => {
    estrella.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));
});

// Overlap para recogerlas
this.physics.add.overlap(this.jugador, estrellas, (j, e) => {
    e.disableBody(true, true);  // Desactivar
    this.puntuacion += 10;
});
```

Callbacks de Colisión

Ejecutar código cuando ocurre colisión:

```
// Callback básico
this.physics.add.collider(jugador, enemigo, (j, e) => {
    // Se ejecuta cada vez que colisionan
    j.setTint(0xff0000);      // Jugador en rojo
    j.setVelocityX(-200);     // Retroceder
});
```

Process callback (condición):

- Decide si procesar la colisión
- Retorna `true` para procesar, `false` para ignorar

```
this.physics.add.collider(
    jugador,
    enemigo,
    this.dañar,              // Si procesa
    this.puedeRecibirDaño,   // Condición
    this
);
```

Sistema de Invulnerabilidad

Ejemplo: no recibir daño temporalmente

```
function puedeRecibirDaño(jugador, enemigo) {
    // Solo procesar si no es invulnerable
    return !jugador.invulnerable;
}

function dañar(jugador, enemigo) {
    jugador.invulnerable = true;
    jugador.setTint(0xff0000);  // Rojo

    // Volver vulnerable tras 2 segundos
    this.time.delayedCall(2000, () => {
        jugador.invulnerable = false;
        jugador.clearTint();
    });
}
```

Detectar Contacto con Superficies

Propiedades útiles para mecánicas:

- `touching.down` : Tocando suelo
- `touching.up` : Tocando techo
- `touching.left/right` : Tocando paredes
- `blocked.*` : Similar pero solo objetos inmóviles

```
function update() {
    // Verificar si toca el suelo
    if (this.jugador.body.touching.down) {
        console.log('En el suelo');
    }

    // Saltar solo si está en el suelo
    if (this.cursors.up.isDown &&
        this.jugador.body.touching.down) {
        this.jugador.setVelocityY(-330);
    }

    // Detectar si está contra pared
    if (this.jugador.body.touching.left) {
        console.log('Pared izquierda');
    }
}
```

Patrón Command

¿Qué es el Patrón Command?

Problema en juegos multijugador:

- Capturar acciones del jugador (input)
- Ejecutar localmente (respuesta inmediata)
- Transmitir por red a otros jugadores
- Reproducir acciones recibidas

Solución: Patrón Command

- Encapsula cada acción como un objeto
- Separa input, lógica y networking
- Facilita testing y debugging

Beneficios del Patrón

Ventajas principales:

- **Separación de responsabilidades**: Input, lógica y red independientes
- **Testabilidad**: Probar comandos sin juego completo
- **Flexibilidad**: Añadir comandos sin modificar código existente
- **Networking transparente**: Mismo comando para local y red
- **Debugging**: Registrar/reproducir todas las acciones
- **Predicción cliente**: Respuesta inmediata + reconciliación servidor

Estructura Base - Command

Clase abstracta con interfaz común:

```
class Command {
    execute() {
        // Implementar en subclases
        // Ejecuta la acción en el juego
    }

    serialize() {
        // Convertir a JSON para enviar por red
        return {};
    }

    getPlayer() {
        // Retorna la entidad asociada
        return null;
    }
}
```

Nota

Esta es la base que heredarán todos los comandos específicos

Comando Concreto - Constructor

Definir el comando específico:

```
1  class MovePaddleCommand extends Command {
2      constructor(paddle, direction) {
3          super();
4          this.paddle = paddle;         // Referencia al objeto
5          this.direction = direction;  // 'up', 'down', 'stop'
6      }
7
8      // Métodos en siguientes slides...
9  }
```

Tip

Cada comando guarda toda la información necesaria para ejecutarse

Comando Concreto - Ejecutar

Implementar la lógica del comando:

```
class MovePaddleCommand extends Command {
    // ... constructor ...

    execute() {
        const speed = 300;  // Píxeles por segundo

        if (this.direction === 'up') {
            this.paddle.setVelocityY(-speed);
        } else if (this.direction === 'down') {
            this.paddle.setVelocityY(speed);
        } else {  // 'stop'
            this.paddle.setVelocityY(0);
        }
    }

    // Métodos serialize() y getPlayer() en siguiente slide...
}
```

Comando Concreto - Serializar

Preparar para transmisión por red:

```
class MovePaddleCommand extends Command {
    // ... constructor y execute() ...

    serialize() {
        // Convertir a formato JSON simple
        return {
            type: 'MOVE_PADDLE',
            playerId: this.paddle.id,
            direction: this.direction
        };
    }

    getPlayer() {
        return this.paddle;
    }
}
```

Nota

Solo se envían datos esenciales, no referencias a objetos

CommandProcessor - Estructura

Coordina comandos locales y remotos:

```
class CommandProcessor {
    constructor() {
        this.players = new Map();  // Registro de jugadores
        this.network = null;       // Gestor de red
    }

    setNetwork(networkManager) {
        this.network = networkManager;
    }

    // Métodos process() y receiveCommand() en siguientes slides...
}
```

CommandProcessor - Procesar Local

Ejecutar comandos del jugador local:

```
class CommandProcessor {
    // ... constructor ...

    process(command) {
        const player = command.getPlayer();

        // Solo ejecutar si es jugador local
        if (player && player.authority === 'LOCAL') {
            command.execute();  // Respuesta inmediata

            // Transmitir a otros jugadores
            if (this.network && this.network.isConnected()) {
                this.network.send(command.serialize());
            }
        }
    }
}
```

Importante

La autoridad `LOCAL` evita ejecutar el comando dos veces

CommandProcessor - Recibir Remoto

Procesar comandos de otros jugadores:

```
class CommandProcessor {
    // ... process() ...

    receiveCommand(data) {
        const player = this.players.get(data.playerId);

        // Solo ejecutar si es jugador remoto
        if (player && player.authority === 'REMOTE') {
            const command = this.deserialize(data, player);
            if (command) {
                command.execute();
            }
        }
    }
}
```

Nota

Comandos remotos solo se ejecutan al recibirlos por red

CommandProcessor - Deserializar

Reconstruir comandos desde JSON:

```
class CommandProcessor {
    // ... receiveCommand() ...

    deserialize(data, player) {
        switch(data.type) {
            case 'MOVE_PADDLE':
                return new MovePaddleCommand(player, data.direction);

            case 'SHOOT':
                return new ShootCommand(player, data.x, data.y);

            default:
                console.warn('Comando desconocido:', data.type);
                return null;
        }
    }
}
```

Integración en Phaser - Create

Configurar escena con autoridad:

```
class GameScene extends Phaser.Scene {
    constructor() {
        super('GameScene');
        this.commandProcessor = new CommandProcessor();
    }

    create() {
        // Paleta local (controlada por este jugador)
        this.localPaddle = this.physics.add.image(50, 300, 'paddle');
        this.localPaddle.id = 'player1';
        this.localPaddle.authority = 'LOCAL';
        this.localPaddle.setCollideWorldBounds(true);

        // Continúa en siguiente slide...
    }
}
```

Integración en Phaser - Create (2)

Configurar paleta remota y registrar:

```
create() {
    // ... localPaddle ...

    // Paleta remota (controlada por otro jugador)
    this.remotePaddle = this.physics.add.image(750, 300, 'paddle');
    this.remotePaddle.id = 'player2';
    this.remotePaddle.authority = 'REMOTE';
    this.remotePaddle.setCollideWorldBounds(true);

    // Registrar ambos jugadores
    this.commandProcessor.players.set('player1', this.localPaddle);
    this.commandProcessor.players.set('player2', this.remotePaddle);

    // Configurar input
    this.cursors = this.input.keyboard.createCursorKeys();
}
```

Integración en Phaser - Update

Convertir input a comandos:

```
update() {
    // Crear comando según tecla presionada
    let command;

    if (this.cursors.up.isDown) {
        command = new MovePaddleCommand(this.localPaddle, 'up');
    } else if (this.cursors.down.isDown) {
        command = new MovePaddleCommand(this.localPaddle, 'down');
    } else {
        command = new MovePaddleCommand(this.localPaddle, 'stop');
    }

    // Procesar (ejecuta local y envía a red)
    this.commandProcessor.process(command);
}
```

Flujo Completo del Patrón

Jugador Local:

1. Presiona tecla en `update()`
2. Crea `MovePaddleCommand`
3. `CommandProcessor.process()` ejecuta inmediatamente
4. Serializa y envía por red

Jugador Remoto:

1. Recibe datos JSON por red
2. `CommandProcessor.receiveCommand()` deserializa
3. Crea `MovePaddleCommand` con datos recibidos
4. Ejecuta el comando

Ventajas en Arquitectura

Transparencia de red:

- Mismo código para local y remoto
- Solo cambia la autoridad (LOCAL/REMOTE)

Preparado para evolución:

- Fácil cambiar de REST a WebSockets
- Solo modificar NetworkManager
- Comandos permanecen iguales

Escalabilidad:

- Añadir nuevos comandos = nueva clase
- No modificar código existente
- Principio Open/Closed

Resumen

Conceptos Clave de Phaser

Estructura básica:

- Canvas con coordenadas (0,0 arriba-izquierda)
- Tres funciones: `preload` , `create` , `update` (dentro de escenas)
- SceneManager para múltiples pantallas
- Pasar datos entre escenas con parámetro en `create()`

Físicas:

- Arcade Physics (simple y rápido)
- Collider (con física) vs Overlap (sin física)
- Grupos estáticos y dinámicos

Input:

Conceptos Clave del Patrón Command

Arquitectura:

- Command: Encapsula cada acción
- CommandProcessor: Coordina local y red
- Autoridad LOCAL/REMOTE

Beneficios:

- Separación input/lógica/red
- Testing simplificado
- Debugging con registro de comandos
- Preparado para networking