

# Introducción a Phaser 3

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez Natalia Madrueño

[ruben.rodriguez@urjc.es](mailto:ruben.rodriguez@urjc.es)

URJC

[natalia.madrueno@urjc.es](mailto:natalia.madrueno@urjc.es)

URJC

2025-09-09



# Tabla de contenidos

- [Introducción a Phaser 3](#)
- [Estructura Básica](#)
- [Gestión de Escenas](#)
- [Trabajo con Imágenes](#)
- [Motores de Físicas](#)
- [Sistema de Entrada](#)
- [Detección de Colisiones](#)
- [Patrón Command](#)
- [Resumen](#)

# 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

```
1 // Configurar tipo de renderizado
2 const config = {
3   type: Phaser.AUTO, // Elige el mejor disponible
4   // type: Phaser.CANVAS, // Forzar Canvas
5   // type: Phaser.WEBGL, // Forzar WebGL
6 };
```

# Requisitos para Empezar

## Navegador moderno:

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

## Obtener Phaser 3:

- **CDN** (más rápido)

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

- **Descarga directa** [phaser.io/download/stable](https://phaser.io/download/stable)
- **GitHub** [github.com/photonstorm/phaser](https://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

```
1 const config = {  
2     type: Phaser.AUTO,          // Renderizado automático  
3     width: 800,                // Ancho del canvas  
4     height: 600,               // Alto del canvas  
5     physics: {  
6         default: 'arcade',    // Motor de físicas  
7         arcade: {  
8             gravity: { y: 300 } // Gravedad vertical  
9         }  
10    },  
11    scene: {  
12        preload: preload,    // Función de carga  
13        create: create,      // Función de creación  
14        update: update       // Función de actualización  
15    }  
16};
```

# 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

```
1 function preload() {  
2     // Cargar recursos con identificadores únicos  
3     this.load.image('cielo', 'assets/cielo.png');  
4     this.load.image('suelo', 'assets/plataforma.png');  
5     this.load.image('estrella', 'assets/estrella.png');  
6 }
```



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

```
1 function create() {  
2     // Añadir imagen de fondo centrada  
3     this.add.image(400, 300, 'cielo');  
4  
5     // Crear plataforma con físicas  
6     this.plataforma = this.physics.add.image(400, 500, 'suelo');  
7     this.plataforma.setImmovable(true); // No se mueve  
8 }
```

# 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)

```
1 function update(time, delta) {  
2     // Mover jugador con teclado  
3     if (this.cursors.left.isDown) {  
4         this.jugador.x -= 5; // Mover izquierda  
5     }  
6     if (this.cursors.right.isDown) {  
7         this.jugador.x += 5; // Mover derecha  
8     }  
9 }
```



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):

```
1 // Crea una escena anónima automáticamente
2 const config = {
3     scene: {
4         preload: preload,
5         create: create,
6         update: update
7     }
8 };
```

## Forma explícita (recomendada):

```
1 class MiEscena extends Phaser.Scene {
2     preload() { /* ... */ }
3     create() { /* ... */ }
4     update() { /* ... */ }
5 }
6
7 const config = {
8     scene: [MiEscena, OtraEscena]
9 };
```



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:

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

# Configurar Múltiples Escenas

## Definir escenas en el array de configuración:

```
1 // La primera escena del array se inicia automáticamente
2 const config = {
3   scene: [MenuPrincipal, Juego, GameOver]
4 };
```

## Añadir/eliminar dinámicamente:

```
1 // Añadir nueva escena en tiempo de ejecución
2 this.scene.add('clave', ConfigEscena, autoStart, datos);
3
4 // Eliminar una escena
5 this.scene.remove('clave');
```

# Cambiar Entre Escenas

## start() vs launch():

```
1 // start: Detiene la actual y cambia
2 this.scene.start('Juego', { nivel: 1 });
3
4 // launch: Ejecuta en paralelo (útil para HUDs)
5 this.scene.launch('MenuPausa', { origenEscena: 'Juego' });
6
7 // stop: Detiene completamente
8 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

```
1 // Pausar (sigue visible, no actualiza)
2 this.scene.pause('Juego');
3
4 // Reanudar
5 this.scene.resume('Juego');
```

## sleep() / wake():

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

```
1 // Dormir (no visible, no actualiza)
2 this.scene.sleep('Fondo');
3
4 // Despertar
5 this.scene.wake('Fondo');
```

# Sistema de Pausa - Ejemplo

## Escena principal del juego:

```
1 class Juego extends Phaser.Scene {  
2     constructor() {  
3         super('Juego');  
4     }  
5  
6     create() {  
7         // Configurar juego...  
8  
9         // Detectar tecla ESC para pausar  
10        this.input.keyboard.on('keydown-ESC', () => {  
11            this.scene.pause();           // Pausar juego  
12            this.scene.launch('MenuPausa', {  
13                escenaOrigen: 'Juego'  
14            });  
15        });  
16    }  
17}
```

# Sistema de Pausa - Ejemplo (2)

## Escena del menú de pausa:

```
1 class MenuPausa extends Phaser.Scene {  
2     constructor() {  
3         super('MenuPausa');  
4     }  
5  
6     create(datos) {  
7         // Recibir datos de la escena que pausó  
8         console.log('Pausado desde:', datos.escenaOrigen);  
9  
10        // Crear fondo semitransparente  
11        let overlay = this.add.rectangle(400, 300, 800, 600, 0x000000, 0.7);  
12  
13        // Botón continuar  
14        let botonContinuar = this.add.text(400, 300, 'Continuar', {  
15            fontSize: '32px'  
16        }).setOrigin(0.5);  
17        botonContinuar.setInteractive();  
18  
19        botonContinuar.on('pointerdown', () => {  
20            this.scene.stop();           // Cerrar menú  
21            this.scene.resume(datos.escenaOrigen); // Reanudar juego  
22        });  
23    }  
24 }
```

# Pasar Datos Entre Escenas

## Enviar datos al cambiar de escena:

```
1 // Pasar datos al iniciar una escena
2 this.scene.start('SiguienteEscena', {
3     puntuacion: 100,
4     nivel: 2,
5     nombre: 'Jugador1'
6});
```

## Recibir datos en la escena destino:

```
1 class SiguienteEscena extends Phaser.Scene {
2     create(datos) {
3         // Recibir los datos como parámetro
4         console.log(datos.puntuacion); // 100
5         console.log(datos.nivel); // 2
6         console.log(datos.nombre); // 'Jugador1'
7
8         this.add.text(100, 100, `Puntuación: ${datos.puntuacion}`);
9     }
10 }
```

# Pasar Datos - Otros Métodos

También funciona con launch() y otros:

```
1 // Con launch (escena en paralelo)
2 this.scene.launch('MenuPausa', {
3     juegoActual: 'Nivel1',
4     tiempoTranscurrido: 120
5 });
6
7 // Con transition
8 this.scene.transition({
9     target: 'GameOver',
10    duration: 1000,
11    data: {
12        puntuacionFinal: this.puntuacion,
13        tiempoJugado: this.tiempo
14    }
15});
```

# 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

```
1 this.scene.transition({
2   target: 'SiguienteEscena',
3   duration: 1000,           // 1 segundo
4   moveBelow: true,
5   data: {                  // Datos a pasar
6     puntuacion: this.puntos,
7     nivel: this.nivelActual
8   },
9   onUpdate: (progress) => {
10     // Crear efectos de fundido
11     this.cameras.main.setAlpha(1 - progress);
12   }
13});
```

# Reordenar Escenas

## Control del orden de renderizado:

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

```
1 // Operaciones básicas
2 this.scene.bringToFront('HUD');
3 this.scene.sendToBack('Fondo');
4
5 // Movimiento relativo
6 this.scene.moveAbove('A', 'B');    // A encima de B
7 this.scene.moveBelow('A', 'B');    // A debajo de B
8
9 // Movimiento incremental
10 this.scene.moveUp('Escena');
11 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

```
1 function preload() {  
2     // Cargar con identificador único  
3     this.load.image('personaje', 'assets/personaje.png');  
4     this.load.image('fondo', 'assets/fondo.jpg');  
5 }  
6  
7 function create() {  
8     // Añadir al canvas en posición (x, y)  
9     this.add.image(400, 300, 'fondo');  
10    let sprite = this.add.image(200, 150, 'personaje');  
11 }
```

## ⚠ 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

```
1 // Cambiar punto de origen
2 let imagen = this.add.image(100, 100, 'personaje')
3   .setOrigin(0, 0); // Esquina superior izquierda
4
5 imagen.setOrigin(0.5, 1); // Centro inferior
6 imagen.setOrigin(1, 0); // Esquina superior derecha
```



Tip

Cambiar origen útil para alinear objetos o rotaciones específicas

# Transformaciones Básicas

## Escalar, voltear y rotar:

```
1 let jugador = this.add.image(400, 300, 'personaje');
2
3 // Escalar uniformemente
4 jugador.setScale(1.5);      // 150% del tamaño
5
6 // Escalar independientemente
7 jugador.setScale(2, 0.5);   // Ancho x2, alto x0.5
8
9 // Voltear
10 jugador.flipX = true;     // Espejo horizontal
11 jugador.flipY = true;     // Espejo vertical
12
13 // Rotar (en radianes)
14 jugador.rotation = Math.PI / 4; // 45 grados
```



$\pi$  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`

```
1 let fondo = this.add.image(400, 300, 'cielo');
2 fondo.depth = 0; // Atrás
3
4 let jugador = this.add.image(400, 300, 'personaje');
5 jugador.depth = 10; // Encima del fondo
6
7 let hud = this.add.image(400, 50, 'interfaz');
8 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:

```
1 const config = {  
2     physics: {  
3         default: 'arcade',  
4         arcade: {  
5             gravity: { y: 300 }, // Gravedad en píxeles/s2  
6             debug: false      // Mostrar contornos físicos  
7         }  
8     }  
9 };
```



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:

```
1 this.jugador = this.physics.add.image(100, 450, 'personaje');  
2  
3 // No salir de los límites del canvas  
4 this.jugador.setCollideWorldBounds(true);  
5  
6 // Rebote al chocar (0 = no rebota, 1 = rebote perfecto)  
7 this.jugador.setBounce(0.3);  
8  
9 // Velocidad inicial (pixeles/segundo)  
10 this.jugador.setVelocity(100, -50);  
11  
12 // Aceleración (pixeles/segundo2)  
13 this.jugador.setAcceleration(50, 0);
```

# Sistema de Entrada

# Teclado - Eventos

## Detectar teclas específicas:

```
1 function create() {  
2     // Detectar cuando se presiona  
3     this.input.keyboard.on('keydown-SPACE', () => {  
4         this.jugador.setVelocityY(-330); // Saltar  
5     });  
6  
7     // Detectar cuando se suelta  
8     this.input.keyboard.on('keyup-SPACE', () => {  
9         console.log('Tecla soltada');  
10    });  
11}
```



keydown se dispara al presionar,keyup al soltar

# Teclado - Modificadores

Detectar combinaciones de teclas:

```
1 this.input.keyboard.on('keydown-A', (event) => {
2   if (event.ctrlKey) {
3     console.log('CTRL + A');
4   } else if (event.shiftKey) {
5     console.log('SHIFT + A');
6   } else if (event.altKey) {
7     console.log('ALT + A');
8   } else {
9     console.log('Solo A');
10  }
11});
```

# Teclado - Cursos

## Objeto para flechas, espacio y shift:

```
1 function create() {
2     // Crear objeto de cursores
3     this.cursors = this.input.keyboard.createCursorKeys();
4 }
5
6 function update() {
7     // Resetear velocidad
8     this.jugador.setVelocityX(0);
9
10    // Comprobar teclas presionadas
11    if (this.cursors.left.isDown) {
12        this.jugador.setVelocityX(-160);
13    } else if (this.cursors.right.isDown) {
14        this.jugador.setVelocityX(160);
15    }
16
17    // Saltar solo si está en el suelo
18    if (this.cursors.up.isDown && this.jugador.body.touching.down) {
19        this.jugador.setVelocityY(-330);
20    }
21 }
```

# Teclado - Combos

## Detectar secuencias de teclas:

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

```
1 function create() {
2     // Combo con letras
3     let combo1 = this.input.keyboard.createCombo('KONAMI');
4
5     // Combo Konami: ↑ ↑ ↓ ↓ ← → ← →
6     let combo2 = this.input.keyboard.createCombo(
7         [38, 38, 40, 40, 37, 39, 37, 39]
8     );
9
10    // Detectar cuando se completa
11    this.input.keyboard.on('keycombomatch', (combo) => {
12        console.log('¡Código secreto desbloqueado!');
13        this.activarModoEspecial();
14    });
15 }
```

# Ratón y Táctil

## API unificada para ratón y pantallas táctiles:

```
1 function create() {
2     // Detectar clic/toque
3     this.input.on('pointerdown', (pointer) => {
4         if (pointer.leftButtonDown()) {
5             console.log('Clic izquierdo:', pointer.x, pointer.y);
6         }
7         if (pointer.rightButtonDown()) {
8             console.log('Clic derecho');
9         }
10    });
11
12    // Detectar cuando se suelta
13    this.input.on('pointerup', (pointer) => {
14        console.log('Botón soltado');
15    });
16
17    // Detectar movimiento
18    this.input.on('pointermove', (pointer) => {
19        console.log('Posición:', pointer.x, pointer.y);
20    });
21 }
```

# 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 colecciónables

# Ejemplos de Uso

## Collider para plataformas:

```
1 function create() {  
2     let jugador = this.physics.add.image(100, 450, 'personaje');  
3  
4     let plataforma = this.physics.add.image(400, 500, 'suelo');  
5     plataforma.setImmovable(true); // No se mueve al impacto  
6  
7     // Colisión con resolución física  
8     this.physics.add.collider(jugador, plataforma);  
9 }
```

## Overlap para colecciónables:

```
1 let estrella = this.physics.add.image(200, 200, 'estrella');  
2  
3 this.physics.add.overlap(jugador, estrella, (j, e) => {  
4     e.destroy(); // Eliminar estrella  
5     this.puntuacion += 10;  
6 });
```

# Grupos Estáticos

Para objetos inmóviles (plataformas, paredes):

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

```
1 function create() {
2     // Crear grupo estático
3     let plataformas = this.physics.add.staticGroup();
4
5     // Añadir plataformas
6     plataformas.create(400, 568, 'suelo')
7         .setScale(2)
8         .refreshBody(); // Actualizar tras modificar
9
10    plataformas.create(600, 400, 'suelo');
11    plataformas.create(50, 250, 'suelo');
12
13    // Colisión con todas
14    this.physics.add.collider(this.jugador, plataformas);
15 }
```

# Grupos Dinámicos

Para objetos con físicas completas:

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

```
1 // Crear 12 estrellas espaciadas
2 let estrellas = this.physics.add.group({
3   key: 'estrella',
4   repeat: 11, // 1 + 11 = 12 total
5   setXY: { x: 12, y: 0, stepX: 70 } // Cada 70px
6 });
7
8 // Aplicar propiedades a cada una
9 estrellas.children.iterate((estrella) => {
10   estrella.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));
11 });
12
13 // Overlap para recogerlas
14 this.physics.add.overlap(this.jugador, estrellas, (j, e) => {
15   e.disableBody(true, true); // Desactivar
16   this.puntuacion += 10;
17 });
```

# Callbacks de Colisión

Ejecutar código cuando ocurre colisión:

```

1 // Callback básico
2 this.physics.add.collider(jugador, enemigo, (j, e) => {
3     // Se ejecuta cada vez que colisionan
4     j.setTint(0xff0000);          // Jugador en rojo
5     j.setVelocityX(-200);        // Retroceder
6 });

```

Process callback (condición):

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

```

1 this.physics.add.collider(
2     jugador,
3     enemigo,
4     this.dañar,                // Si procesa
5     this.puedeRecibirDaño,    // Condición
6     this
7 );

```

# Sistema de Invulnerabilidad

## Ejemplo: no recibir daño temporalmente

```
1 function puedeRecibirDaño(jugador, enemigo) {  
2     // Solo procesar si no es invulnerable  
3     return !jugador.invulnerable;  
4 }  
5  
6 function dañar(jugador, enemigo) {  
7     jugador.invulnerable = true;  
8     jugador.setTint(0xff0000); // Rojo  
9  
10    // Volver vulnerable tras 2 segundos  
11    this.time.delayedCall(2000, () => {  
12        jugador.invulnerable = false;  
13        jugador.clearTint();  
14    });  
15 }
```

# 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

```
1 function update() {
2     // Verificar si toca el suelo
3     if (this.jugador.body.touching.down) {
4         console.log('En el suelo');
5     }
6
7     // Saltar solo si está en el suelo
8     if (this.cursors.up.isDown &&
9         this.jugador.body.touching.down) {
10        this.jugador.setVelocityY(-330);
11    }
12
13    // Detectar si está contra pared
14    if (this.jugador.body.touching.left) {
15        console.log('Pared izquierda');
16    }
17 }
```

# 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:

```
1 class Command {  
2     execute() {  
3         // Implementar en subclases  
4         // Ejecuta la acción en el juego  
5     }  
6  
7     serialize() {  
8         // Convertir a JSON para enviar por red  
9         return {};  
10    }  
11  
12    getPlayer() {  
13        // Retorna la entidad asociada  
14        return null;  
15    }  
16 }
```



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:

```
1 class MovePaddleCommand extends Command {  
2     // ... constructor ...  
3  
4     execute() {  
5         const speed = 300; // Píxeles por segundo  
6  
7         if (this.direction === 'up') {  
8             this.paddle.setVelocityY(-speed);  
9         } else if (this.direction === 'down') {  
10            this.paddle.setVelocityY(speed);  
11        } else { // 'stop'  
12            this.paddle.setVelocityY(0);  
13        }  
14    }  
15  
16    // Métodos serialize() y getPlayer() en siguiente slide...  
17 }
```

# Comando Concreto - Serializar

Preparar para transmisión por red:

```
1 class MovePaddleCommand extends Command {  
2     // ... constructor y execute() ...  
3  
4     serialize() {  
5         // Convertir a formato JSON simple  
6         return {  
7             type: 'MOVE_PADDLE',  
8             playerId: this.paddle.id,  
9             direction: this.direction  
10        };  
11    }  
12  
13    getPlayer() {  
14        return this.paddle;  
15    }  
16 }
```



Solo se envían datos esenciales, no referencias a objetos

# CommandProcessor - Estructura

Coordina comandos locales y remotos:

```
1 class CommandProcessor {  
2     constructor() {  
3         this.players = new Map(); // Registro de jugadores  
4         this.network = null;    // Gestor de red  
5     }  
6  
7     setNetwork(networkManager) {  
8         this.network = networkManager;  
9     }  
10  
11    // Métodos process() y receiveCommand() en siguientes slides...  
12 }
```

# CommandProcessor - Procesar Local

## Ejecutar comandos del jugador local:

```
1 class CommandProcessor {  
2     // ... constructor ...  
3  
4     process(command) {  
5         const player = command.getPlayer();  
6  
7         // Solo ejecutar si es jugador local  
8         if (player && player.authority === 'LOCAL') {  
9             command.execute(); // Respuesta inmediata  
10  
11             // Transmitir a otros jugadores  
12             if (this.network && this.network.isConnected()) {  
13                 this.network.send(command.serialize());  
14             }  
15         }  
16     }  
17 }
```

### ! Importante

La autoridad `LOCAL` evita ejecutar el comando dos veces

# CommandProcessor - Recibir Remoto

Procesar comandos de otros jugadores:

```
1 class CommandProcessor {  
2     // ... process() ...  
3  
4     receiveCommand(data) {  
5         const player = this.players.get(data.playerId);  
6  
7         // Solo ejecutar si es jugador remoto  
8         if (player && player.authority === 'REMOTE') {  
9             const command = this.deserialize(data, player);  
10            if (command) {  
11                command.execute();  
12            }  
13        }  
14    }  
15 }
```



Comandos remotos solo se ejecutan al recibirlas por red

# CommandProcessor - Deserializar

Reconstruir comandos desde JSON:

```
1 class CommandProcessor {  
2     // ... receiveCommand() ...  
3  
4     deserialize(data, player) {  
5         switch(data.type) {  
6             case 'MOVE_PADDLE':  
7                 return new MovePaddleCommand(player, data.direction);  
8  
9             case 'SHOOT':  
10                return new ShootCommand(player, data.x, data.y);  
11  
12             default:  
13                 console.warn('Comando desconocido:', data.type);  
14                 return null;  
15         }  
16     }  
17 }
```

# Integración en Phaser - Create

## Configurar escena con autoridad:

```
1 class GameScene extends Phaser.Scene {  
2     constructor() {  
3         super('GameScene');  
4         this.commandProcessor = new CommandProcessor();  
5     }  
6  
7     create() {  
8         // Paleta local (controlada por este jugador)  
9         this.localPaddle = this.physics.add.image(50, 300, 'paddle');  
10        this.localPaddle.id = 'player1';  
11        this.localPaddle.authority = 'LOCAL';  
12        this.localPaddle.setCollideWorldBounds(true);  
13  
14        // Continúa en siguiente slide...  
15    }  
16 }
```

# Integración en Phaser - Create (2)

## Configurar paleta remota y registrar:

```
1 create() {  
2     // ... localPaddle ...  
3  
4     // Paleta remota (controlada por otro jugador)  
5     this.remotePaddle = this.physics.add.image(750, 300, 'paddle');  
6     this.remotePaddle.id = 'player2';  
7     this.remotePaddle.authority = 'REMOTE';  
8     this.remotePaddle.setCollideWorldBounds(true);  
9  
10    // Registrar ambos jugadores  
11    this.commandProcessor.players.set('player1', this.localPaddle);  
12    this.commandProcessor.players.set('player2', this.remotePaddle);  
13  
14    // Configurar input  
15    this.cursors = this.input.keyboard.createCursorKeys();  
16 }
```

# Integración en Phaser - Update

## Convertir input a comandos:

```
1 update() {  
2     // Crear comando según tecla presionada  
3     let command;  
4  
5     if (this.cursors.up.isDown) {  
6         command = new MovePaddleCommand(this.localPaddle, 'up');  
7     } else if (this.cursors.down.isDown) {  
8         command = new MovePaddleCommand(this.localPaddle, 'down');  
9     } else {  
10        command = new MovePaddleCommand(this.localPaddle, 'stop');  
11    }  
12  
13    // Procesar (ejecuta local y envía a red)  
14    this.commandProcessor.process(command);  
15 }
```

# 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