UNITY 6.3 · UNIVERSAL 2D · DESDE CERO

De una nave a un juego

Extrejam 2026 · Sesión 2 · Programación de videojuegos

Vas a construir una arena en la que recoges cristales y esquivas meteoritos. No necesitas haber programado antes: primero verás una acción en el juego, después localizarás el código que la provoca y, finalmente, cambiarás una cosa para comprobar que la entiendes.

05 puntos de control **05 apoyos visuales** **01 juego que puedes explicar**

01 · Tu mapa de la sesión

La meta: poder explicar qué hace cada uno de los cinco scripts y modificar una regla pequeña por tu cuenta. Si el juego funciona y puedes explicar tu cambio, has completado la práctica.

Punto de control	Qué puedes hacer al terminar	Scripts de juego
PC1 · Arena	Reconocer objetos, componentes y campos del Inspector	Ninguno
PC2 · Movimiento	Mover la nave con WASD o las flechas	PlayerController
PC3 · Física	Chocar con los muros y recoger cristales	+ Crystal
PC4 · Meteoritos	Crear enemigos desde un prefab y recibir impactos	+ Meteor y MeteorSpawner
PC5 · Partida	Jugar, perder, reiniciar y conservar el récord	+ GameManager

Trabajamos con **Unity 6000.3.27f1 y Universal 2D**. Descarga los puntos de control desde extrejam.com/unity. Extrae el ZIP completo en una carpeta propia, añade esa carpeta de proyecto desde Unity Hub y espera a que termine la importación. Si necesitas recuperar el ritmo, abre el punto de control del bloque anterior; no mezcles sus carpetas Assets entre sí.

Entra en Play para probar y sal antes de editar la escena. Los cambios hechos durante Play normalmente no se conservan. Haz clic dentro de Game para que el teclado llegue al juego. Para trabajar en los retos, utiliza una copia de tu proyecto.

Esta guía funciona sin Internet. La navegación permite ir al bloque actual; el modo de exposición muestra las ideas centrales. Las pistas se abren de una en una y las soluciones quedan al final de cada reto. El PDF contiene también las pistas y las soluciones: evita avanzar hasta ellas si quieres intentarlo primero.

02 · PC1: objetos que tienen componentes

Una **escena** reúne los objetos del nivel. Un **GameObject** es un objeto al que añadimos componentes: Transform coloca la nave; SpriteRenderer la dibuja; más adelante un script decidirá cómo se mueve. Un script guardado en Project todavía no está actuando sobre ningún objeto.

En Hierarchy seleccionas el objeto; en Inspector ves sus componentes y propiedades. Project contiene los archivos reutilizables. Scene permite montar el nivel y Game muestra lo que ve la cámara. Un **prefab** será un modelo guardado en Project del que podremos crear varias copias durante la partida.

La arena contiene una cámara, un fondo, cuatro muros, Player y Crystal. Guarda la escena como `Assets/Scenes/Game.unity`. La cámara es ortográfica, está en (0, 0, −10) y tiene Size 6.2. Player empieza en (0, 0, 0) y Crystal en (3, 1, 0).

Elemento	Posición X, Y	Tamaño del SpriteRenderer Tiled y BoxCollider2D
Muro izquierdo / derecho	−8.25, 0 / 8.25, 0	0.5 × 9.5
Muro superior / inferior	0, 4.75 / 0, −4.75	16 × 0.5
Fondo	0, 0	24 × 15; orden −10

Los muros no son triggers y no necesitan Rigidbody2D. Mantén escala (1,1,1) y rotación cero. Usa los sprites incluidos en PC1; no hace falta importar bibliotecas adicionales. La nave tiene orden de dibujo 2 y el cristal 1. En este punto **no hay movimiento programado**.

Prueba visible: selecciona Player y cambia su posición; comprueba que Transform cambia dónde está, sin alterar su imagen. Devuélvelo al origen y guarda.

03 · PC2: convertir teclas en movimiento

Crea `Assets/Scripts/PlayerController.cs` y añade el componente a Player. El nombre de la clase y el del archivo deben coincidir. El archivo completo comentado está en el apéndice y en `CodigoComentado/PC2/PlayerController.cs` del paquete de materiales. Deja Speed en 5.

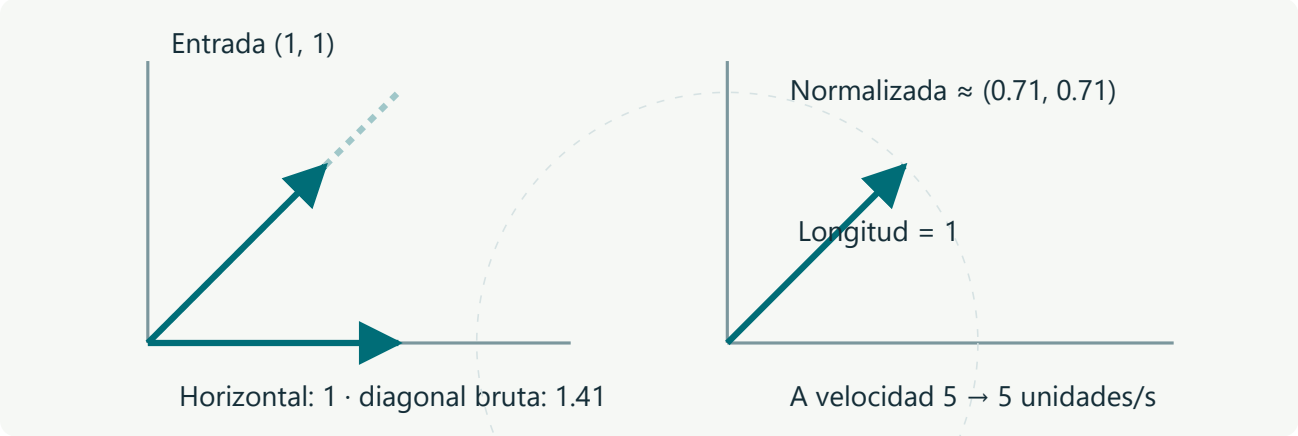
`Update` se ejecuta cada fotograma mientras el componente está activo. Lee las teclas, construye un `Vector2` y llama a `Move`. Un vector reúne dos números: X indica derecha o izquierda; Y indica arriba o abajo. La tecla derecha suma 1 a X; izquierda resta 1. Si pulsas ambas, se compensan.

```
transform.position += (Vector3)direction.normalized * speed * Time.deltaTime;
```

La línea suma un desplazamiento a la posición actual. `normalized` conserva la dirección con longitud 1; el vector cero sigue siendo cero. `speed` expresa unidades por segundo. `Time.deltaTime` es el tiempo transcurrido desde el fotograma anterior: convierte esa velocidad en el desplazamiento de este fotograma.

APOYO 1 / 5

Una diagonal sin ventaja



Sin normalizar: 5 × √2 ≈ 7.07 unidades/s. Normalizada: 5 unidades/s. El vector cero sigue siendo cero.

Predice antes de probar: si quitas `normalized`, ¿la nave recorrerá la misma distancia en un segundo al ir hacia arriba que al ir en diagonal? Abre la explicación después de responder.

▸ **Ver explicación**

No. (1,1) tiene longitud √2, aproximadamente 1.41. A velocidad 5 recorrería unas 7.07 unidades por segundo en diagonal. Normalizar mantiene la velocidad en 5, cualquiera que sea la dirección.

Prueba visible: WASD y flechas mueven la nave; al soltar se detiene. En PC2 atraviesa los muros: todavía cambiamos Transform directamente. La física del bloque siguiente resolverá ese comportamiento.

04 · PC3: física y recogida

Añade a Player un Rigidbody2D **Dynamic**, Gravity Scale 0, Collision Detection Continuous y Freeze Rotation Z. Añade CircleCollider2D de radio 0.38, sin Is Trigger. En PlayerController añade la referencia `body` y arrastra el Rigidbody2D del propio Player a Body.

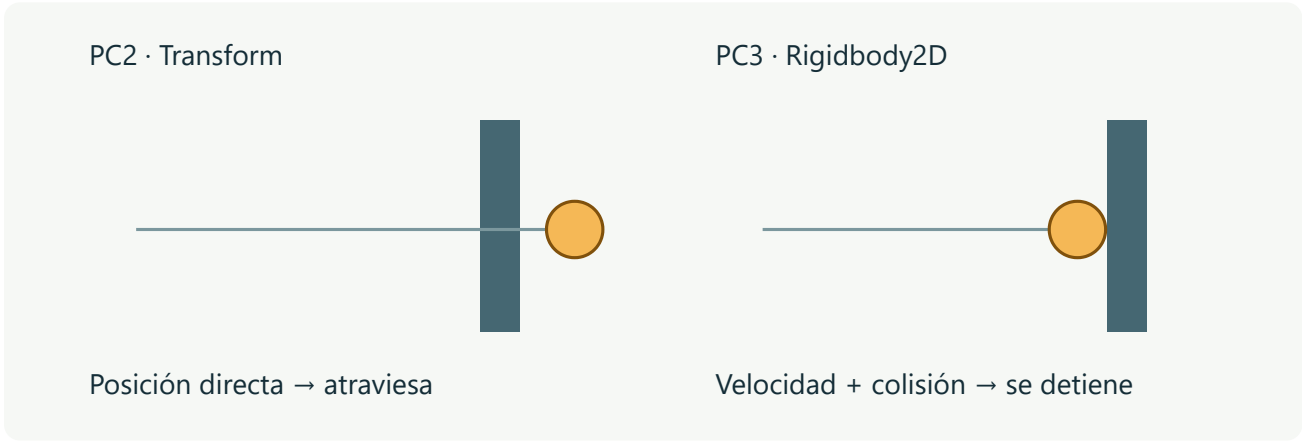
Sustituye el contenido de Move por esta línea; el resto de la lectura de teclas se mantiene. La versión completa está en `CodigoComentado/PC3`.

```
body.linearVelocity = direction.normalized * speed;
```

Ahora entregamos a la física una **velocidad en unidades por segundo**. Aquí no multiplicamos por deltaTime: Rigidbody2D ya integra la velocidad durante la simulación. El cambio se centra en quién mueve la nave; no es una regla universal de que todo código de física deba ir en Update.

APOYO 2 / 5

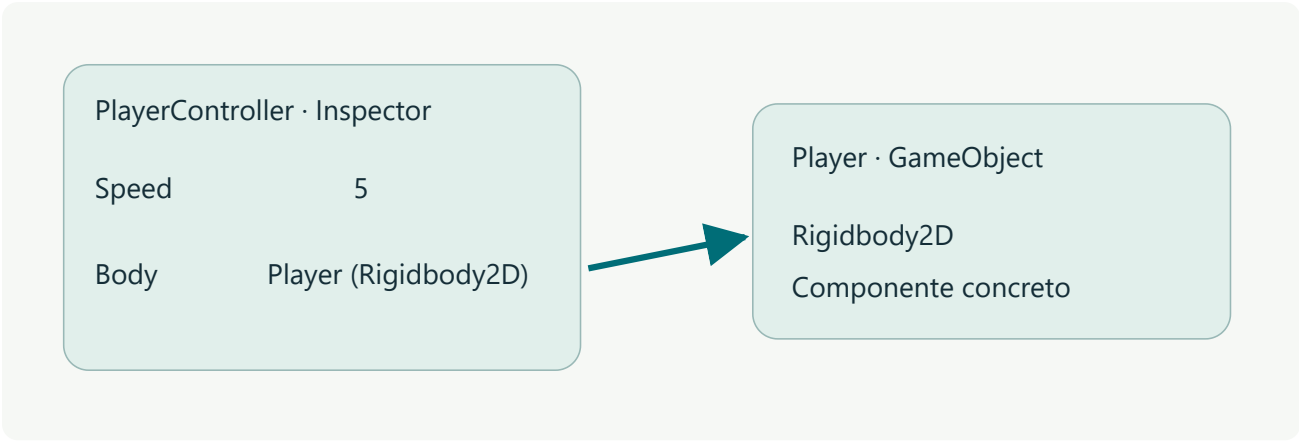
Quién decide el movimiento



Esquema del ejercicio con los componentes descritos. Transform cambia la posición; Rigidbody2D deja que la física resuelva el contacto.

APOYO 4 / 5

Un número y una referencia



Speed guarda un valor. Body guarda el enlace a un componente existente. None significa que no está asignado; declarar el campo no crea ese enlace.

Un valor numérico como Speed guarda un número. Body guarda una **referencia a un componente concreto**. Declarar el campo no lo enlaza automáticamente: cuando aparece None, falta arrastrar el componente. `[SerializeField]` permite guardar y editar ese campo privado en Inspector; `private` limita el acceso desde otras clases.

Para Crystal, crea y asigna la etiqueta `Crystal`, añade CircleCollider2D con Is Trigger y radio 0.23, y añade `Crystal.cs`, con Limits (7.2, 3.7). Un trigger detecta el contacto sin actuar como una pared. PlayerController usa `OnTriggerEnter2D`, comprueba la etiqueta, suma un punto y llama a `Relocate` del cristal. Esa función elige otra posición dentro de los límites.

Prueba visible: los cuatro muros detienen la nave. Al tocar el cristal, cambia de posición y Console muestra el contador. De momento no hay marcador en pantalla. No necesitas aprender un sistema de trazas para seguir la práctica.

▸ **Si no ocurre nada: dos comprobaciones rápidas**

Selecciona Player. ¿Ves PlayerController como componente y está activado? Tener el archivo en Project no basta. Después comprueba que Body contiene su Rigidbody2D, no None. Conecta la referencia fuera de Play y vuelve a probar. Si la causa es otra, pide ayuda o recupera el punto de control.

05 · PC4: un modelo, muchos meteoritos

Crea Meteor con el sprite `meteorBrown_big1`, etiqueta `Meteor` y orden de dibujo 3. Añade CircleCollider2D trigger de radio 0.42 y Rigidbody2D Kinematic con gravedad 0. Añade `Meteor.cs`, enlaza su Body, Speed 3 y Limits (10, 6.5). Arrástralo a `Assets/Prefabs/Meteor.prefab` y retira la instancia de la escena.

Crea un objeto vacío MeteorSpawner y añade `MeteorSpawner.cs`. Enlaza Meteor Prefab con el archivo de Project y Player con la nave de Hierarchy. Deja Interval 1.5 y Edges (9, 5.5). El temporizador acumula deltaTime y, al alcanzar el intervalo, crea un meteorito en uno de los cuatro bordes.

```
meteor.GetComponent<Meteor>().Launch(
    ((Vector2)player.position - position).normalized);
```

Restar la posición de salida a la posición del jugador produce la dirección hacia él. La calculamos **al crear el meteorito** y se la entregamos a Launch. No la recalculamos continuamente.

APOYO 3 / 5

Apuntar una vez no es perseguir



Dirección = (posición del jugador al crear − posición de salida).normalized. Launch recibe ese vector una vez; no hay seguimiento continuo.

Predice: si te mueves después del disparo, ¿el meteorito gira para perseguirte? No: sigue la dirección fijada al salir. Puedes esquivarlo desplazándote. Es una decisión de diseño, no un fallo de seguimiento.

PlayerController empieza con tres vidas, resta una por contacto con un meteorito y destruye ese meteorito. Meteor se destruye también cuando sale de sus límites. El prefab guarda sus propios componentes; el spawner le proporciona la dirección que depende de la escena.

Prueba visible: aparecen meteoritos desde los bordes, avanzan en línea recta y cada impacto resta una vida en Console. PC4 todavía no termina la partida al llegar a cero; ese cierre pertenece a PC5. En Assets/Prefabs encontrarás además HUD y EventSystem preparados para el siguiente bloque: son recursos guardados y aún no están en la escena. Sus botones todavía no tienen acciones asignadas.

06 · PC5: reglas, interfaz y final de partida

GameManager reúne el estado de esta partida: puntos, vidas, dificultad, comienzo y derrota. PlayerController conserva entrada, movimiento y contactos; Crystal cambia de posición; Meteor se mueve y se retira; MeteorSpawner controla cuándo y dónde nace. **Cada script tiene una responsabilidad que puedes explicar en una frase.** No añadimos todos los comportamientos al script de la nave.

En una copia de PC4, utiliza las versiones de `CodigoComentado/PC5`: PlayerController y MeteorSpawner reciben una referencia a GameManager. Añade GameManager a un objeto vacío del mismo nombre y arrástralo a esos dos campos. Añade AudioSource, desactiva Play On Awake y deja Volume 0.65. Usa los sonidos Kenney incluidos: `sfx_twoTone.ogg` para recogida y `sfx_shieldDown.ogg` para impacto.

Arrastra `Assets/Prefabs/HUD.prefab` a Hierarchy y llama Canvas a la instancia. Arrastra también `EventSystem.prefab` una sola vez: ya utiliza InputSystemUIInputModule. Los recursos esenciales de TextMeshPro están incluidos. El HUD es una interfaz preparada para que puedas dedicar este bloque a entender el código y conectar sus referencias.

En Canvas, arrastra Main Camera al campo Render Camera. Ya están preparados Screen Space - Camera, Plane Distance 5, orden 20 y Canvas Scaler con Scale With Screen Size, 1920 × 1080 y Match 0.5. No tienes que crear de nuevo los objetos de la tabla: úsala para reconocer y revisar lo que contiene el prefab.

En todos los RectTransform siguientes, usa anclas y pivote centrales (0.5, 0.5). Las posiciones son relativas al centro, no píxeles absolutos de pantalla. Los textos van centrados y con Raycast Target desactivado; los botones y fondos de panel sí reciben raycasts.

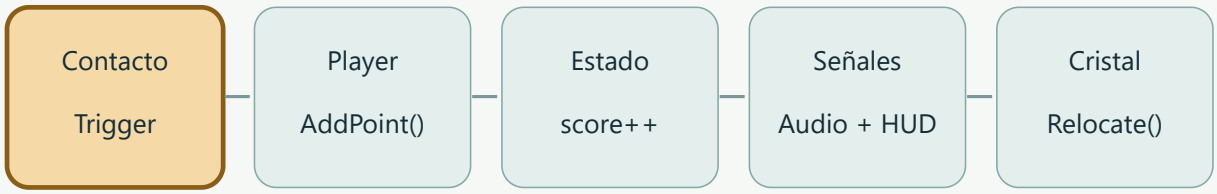
Objeto	Padre	Posición / tamaño	Fuente o función
Score / Lives	Canvas	(−570,470) / (570,470); 600 × 75	34
TitlePanel / DefeatPanel	Canvas	(0,0); 1920 × 1080	Fondo oscuro opaco al 94%
Title	TitlePanel	(0,150); 1600 × 160	76; nombre del juego
Instructions	TitlePanel	(0,10); 1600 × 90	30; mover, recoger y esquivar
PlayButton	TitlePanel	(0,−140); 440 × 92	Jugar; On Click → GameManager.Play
DefeatTitle	DefeatPanel	(0,180); 1600 × 120	64; fin de partida
Result	DefeatPanel	(0,35); 1000 × 130	38
RestartButton	DefeatPanel	(0,−140); 440 × 92	Volver a jugar → GameManager.Restart

Las etiquetas de los botones usan 420 × 80 y fuente 36. Deja DefeatPanel inactivo. En el componente GameManager, conecta por orden: Score Text con Canvas/Score; Lives Text con Canvas/Lives; Result Text con Canvas/DefeatPanel/Result; Title Panel y Defeat Panel con sus paneles; Audio Source con el AudioSource de GameManager; Collect Sound y Hit Sound con los dos clips. Los botones llegan con On Click vacío: añade una entrada, arrastra GameManager y elige Play para Jugar y Restart para Volver a jugar. No pulses Play hasta terminar los cinco scripts y estas conexiones. Incluye Game como escena inicial en la lista del perfil de compilación para que Restart pueda recargarla.

Al recoger, PlayerController llama a AddPoint y después a Relocate. Dentro de AddPoint se incrementa score, se reproduce el sonido y se actualiza el texto. El siguiente esquema explica qué recibe el jugador; el sonido se solicita antes de actualizar el HUD en el código, prácticamente en el mismo instante.

APOYO 5 / 5

Recoger: regla y señales



AddPoint: sumar → solicitar sonido → actualizar HUD. Luego: recolocar.

El HUD y el sonido confirman el mismo cambio. Las señales se solicitan en el mismo contacto; el esquema separa responsabilidades para explicarlas.

```
return Mathf.Max(0.4f, 1.5f - score * 0.05f);
```

Este intervalo controla la dificultad: con 0 puntos aparecen meteoritos cada 1.5 segundos; con 10, cada 1 segundo; con 22 o más, cada 0.4. Un intervalo menor significa más apariciones, no más velocidad. `Max` impide que el intervalo se vuelva cero o negativo.

En el título, `Time.timeScale = 0` pausa el tiempo del juego. Jugar lo devuelve a 1. La derrota guarda el récord mediante PlayerPrefs, enseña el resultado y pausa otra vez. Reiniciar recarga la escena; reaparece el título y hay que pulsar Jugar. PlayerPrefs guarda una preferencia local, no un ranking de Internet.

Prueba visible: título → Jugar → recoger → recibir tres impactos → resultado y récord → Volver a jugar → título → Jugar. La nueva partida empieza con cero puntos y tres vidas; el récord permanece.

07 · Hacer que una acción se entienda

En la primera sesión hablamos del feedback: cómo comunica el juego lo que acaba de pasar. Aquí conectamos una regla con sus señales. Recoger aumenta el marcador y reproduce un sonido; recibir un impacto reduce las vidas y puede añadir un destello breve. La señal debe ayudarte a entender el cambio.

Compara PC5 con el ejecutable con pulido de «Órbita Cero», que puedes descargar en extrejam.com/unity. Observa qué acción acaba de suceder y qué señal te lo confirma. El brillo, las partículas y la sacudida no deben cambiar puntos, vidas ni colisiones. Los paquetes comerciales del pulido no son necesarios para completar estas prácticas y no forman parte de sus materiales.

Mini ejercicio, sólo si queda tiempo: cambia una señal cada vez y compara dos partidas cortas. ¿Puedes distinguir recogida e impacto sin mirar el marcador? ¿Sigues entendiendo ambos con el audio silenciado? Si el destello tapa al siguiente meteorito o la sacudida dificulta leer, reduce su intensidad. No dependas únicamente del color: conserva texto, forma o sonido complementario.

08 · Programar en una jam

La sesión del 27 de octubre terminó con la parte 5 y su ronda. La parte 6, que trataba de cómo se trabaja cuando el reloj manda, no llegó a impartirse por falta de tiempo. Este capítulo la recoge completa para que puedas seguirla por tu cuenta, hayas asistido o no. No necesitas ningún proyecto nuevo: basta con tu copia de PC5 y el ejecutable con pulido de la página de la sesión.

Una jam te da muy poco tiempo para hacer un juego entero. En ese contexto hay atajos que son perfectamente legítimos y que conviene conocer antes de llegar: usar un asistente de IA para cambios pequeños, invertir unos minutos en pulido y aprovechar recursos que no has creado tú. Los tres atajos tienen reglas, y este capítulo explica cuáles son y por qué existen.

La IA como herramienta: pide poco, lee antes de aplicar y compruébalo jugando

Un asistente de IA puede escribir código por ti, y en una jam se usa. Lo útil no es pedirle el juego entero, sino un cambio concreto que puedas revisar en un minuto. Tres reglas resumen la forma de trabajar.

La primera regla es **pedir poco**. Un encargo acotado dice qué archivo se toca, qué tiene que cambiar y qué debe seguir igual. Si pides «mejora el juego», la respuesta puede reescribir cinco scripts y no sabrás cuál de los cambios rompió algo.

La segunda regla es **leer antes de aplicar**. Una respuesta que suena convincente puede estar mal. Antes de aceptar un cambio, localiza cada línea nueva en tu archivo y explica qué hace. Si no entiendes una línea, pide al asistente que la explique en relación con tu código. Si la respuesta toca algo que no pediste, recházalo y vuelve a acotar el encargo.

La tercera regla es **comprobarlo jugando**. El código que compila no tiene por qué hacer lo que querías. La prueba es observable: un valor concreto en el Inspector, una partida corta y un resultado que puedes ver.

Cómo era la demostración prevista. El ponente iba a abrir una copia de PC5 y a pedir a un agente de IA dentro de VS Code, en modo de aprobación antes de cada edición, que hiciera configurables las vidas iniciales entre 1 y 5, con 3 por defecto, tocando sólo `GameManager`. La respuesta esperada es pequeña: un campo `startingLives` visible en el Inspector y una línea en `Start` que lo limita con `Mathf.Clamp`. La demostración consistía en leer el cambio en voz alta antes de aceptarlo, guardar la escena después de cambiar el valor fuera de Play y probar tres casos: con una vida, un impacto termina la partida; con cinco, empiezas con cinco; y tras reiniciar, sigues con cinco. Si la respuesta era correcta, no había que inventar un fallo para dramatizarla. Puedes reproducir exactamente esa demostración con la actividad de IA de este material (11), que incluye el encargo, la respuesta de referencia y las comprobaciones.

Elige dónde trabajas con la IA. Un chat en el navegador te obliga a copiar el código y a pegar el cambio a mano, y eso convierte la lectura en un paso literal. Un agente dentro del editor es más rápido y te enseña el cambio como una diferencia entre versiones, pero sólo es seguro si apruebas cada edición y no le dejas ejecutar órdenes por su cuenta. En los dos casos, trabaja sobre una copia del proyecto o con control de versiones, para poder volver atrás.

En la Extrejam la IA no está prohibida: se declara qué se ha hecho con ella. El detalle de cómo se declara estará en las bases de la jam. Mientras tanto, acostúmbrate a anotar mientras trabajas en qué partes del juego has usado IA (código, gráficos, sonido o texto) y para qué. Reconstruir esa lista el domingo por la mañana es mucho más difícil que llevarla al día.

> «Mi opinión, para terminar. La IA es el futuro y, queramos o no, vamos a tener que aprender a usarla o nos quedaremos atrás. No viene a reemplazarnos: viene a darnos herramientas muy potentes, y va a abrir el desarrollo de videojuegos a mucha más gente. Programadores y artistas la usaremos como ayudante técnico, para trabajar mejor y más rápido que antes. Pero el alma de un juego, lo que lo hace único, la sigue poniendo quien lo diseña.» — Carlos Rubiales, ponente de la sesión

El pulido: el mismo juego, otra sensación

Pulir es añadir señales que hacen que cada acción se sienta, sin tocar las reglas. En la parte 6 el ponente iba a montar en unos cuatro minutos una capa de pulido sobre el juego de PC5. Esa capa ya está montada en el ejecutable de Windows que puedes descargar en la página de la sesión, con el título votado por la sala, «Órbita Cero».

Juega una partida a PC5 y otra al ejecutable. Las reglas son exactamente las mismas: los mismos puntos, las mismas tres vidas, el mismo intervalo de aparición y las mismas colisiones. Lo que cambia es lo que percibes:

Momento	En PC5	En el ejecutable con pulido
Recoger un cristal	Sube el marcador y suena un tono	El cristal brilla de forma pulsante, y al recogerlo se produce un destello de partículas y un sonido propio
Recibir un impacto	Bajan las vidas y suena un aviso	La nave destella en blanco, se produce una explosión en el punto del impacto, la cámara tiembla 0,15 segundos y el juego se congela 0,04 segundos
Interfaz	Textos y paneles sencillos	Marcos y botones con estilo propio
Ambiente	Silencio entre sonidos	Música en bucle a volumen bajo

La pausa de 0,04 segundos es la técnica que se conoce como *hit stop*: detener el juego un instante casi imperceptible hace que el golpe pese. La sacudida y la pausa son cortas a propósito. Si duran demasiado, tapan el siguiente meteorito y empeoran el juego en vez de mejorarlo.

Cómo se monta, en pocas líneas. El proyecto del ponente llegaba con todo preparado salvo lo que se iba a hacer en directo: arrastrar dos grupos de efectos como hijos de la nave, conectar el destello al shader de la nave y añadir a `GameManager` dos campos y dos llamadas. Una llamada dispara el efecto de recogida justo después de sumar el punto y la otra dispara el de impacto justo después de restar la vida. Nada más cambia en el código. Esto es posible porque `GameManager` ya tenía un sitio claro para cada regla; en un código desordenado, añadir esas dos líneas habría obligado a buscar en varios archivos.

Por qué no tienes ese proyecto. El pulido usa siete paquetes de la Unity Asset Store comprados por el estudio: Feel, All In 1 Sprite Shader, 2D Explosion URP, Ultimate Loot VFX Pack, Cartoon GUI Pack, Ultimate Game Music Collection y Universal Sound FX. Su licencia permite publicar un juego terminado que los incluya, y por eso puedes descargar el ejecutable. No permite repartir el proyecto con esos archivos dentro. La sección siguiente explica la diferencia.

Pulido con lo que ya tienes. No necesitas comprar nada para practicar. En tu copia de PC5 puedes cambiar durante un instante el color del `SpriteRenderer` de la nave al recibir un impacto, añadir un `Particle System` de Unity al recoger un cristal o usar otros sonidos de Kenney. Aplica la misma disciplina que el ponente: un efecto cada vez, sin tocar las reglas, y comparando dos partidas cortas. La sección 07 · «Hacer que una acción se entienda» de este material propone cómo evaluar cada señal.

Qué puede usar y compartir un equipo

Todo lo que no has creado tú tiene una licencia, y la licencia dice qué puedes hacer con ello. En una jam importan sobre todo dos casos, que son los que has visto en esta sesión.

Los recursos de Kenney tienen licencia CC0. CC0 significa que el autor renuncia a sus derechos: puedes usar esos gráficos y sonidos, modificarlos y compartirlos, también dentro de un proyecto, sin pedir permiso. Por eso van incluidos en los cinco puntos de control. Dar crédito no es obligatorio con CC0, pero sí es una buena costumbre, y en una jam forma parte de declarar lo que no es tuyo.

Los paquetes de la Asset Store usan normalmente la licencia estándar de Unity. Esa licencia te permite incluir el paquete dentro de un juego terminado y publicarlo. No te permite distribuir los archivos del paquete por separado, ni en un proyecto compartido del que cualquiera pueda extraerlos. En la práctica, eso afecta a los equipos de tres maneras. Si un compañero compró un paquete, la licencia es suya y no del equipo, porque un grupo de personas que se junta para una jam no es una misma empresa. Los paquetes de tipo herramienta, como Feel, necesitan además un puesto por cada persona que trabaja en el proyecto donde están instalados. Y subir a un repositorio público un proyecto con paquetes comerciales dentro equivale a redistribuirlos. Algunos paquetes tienen licencias propias, distintas de la estándar, así que lee la de cada uno antes de meterlo en el proyecto común.

Esta sección resume la información pública de la Asset Store para orientarte; no es asesoramiento legal. Ante la duda con un paquete concreto, consulta a su autor o elige un recurso con licencia libre.

Atajos legítimos para una jam

Estos atajos son legales en una jam y te ahorran horas:

1. **Assets gratuitos, declarados.** Usa recursos gratuitos con licencia que lo permita, como los de Kenney, y anota de dónde sale cada uno.
2. **Tu propia plantilla, preparada antes.** Puedes llegar con un proyecto base hecho en casa, con la estructura de carpetas, los ajustes y las piezas que siempre repites. Lo que se valora es el juego que haces durante la jam, no el tiempo que pierdes configurando Unity.
3. **Paquetes de la Asset Store, con su licencia.** Si los usas, respeta lo que explica la sección anterior.

Lo que se valora es que el juego sea tuyo. Por eso todo lo que no sea tuyo, incluida la IA, se declara.

Código limpio y modular: un script, una responsabilidad

El hábito que más te ayudará en la jam es dar a cada script una sola tarea que puedas explicar en una frase. El juego de la sesión tiene cinco scripts y cada uno hace una cosa:

Script	Su responsabilidad
`PlayerController`	Mueve la nave y avisa cuando toca algo
`Crystal`	Se recoloca al ser recogido
`Meteor`	Vuela recto y desaparece al salir de la arena
`MeteorSpawner`	Decide cuándo y desde dónde sale cada meteorito
`GameManager`	Arbitra la partida: puntos, vidas, dificultad, derrota y reinicio

Esta separación es la razón de tres cosas que has visto. Cada parte de la sesión añadió una pieza sin romper las anteriores. El cambio de vidas iniciales que se pide a la IA toca un único archivo. Y el pulido se enchufa con dos líneas en `GameManager`, porque ese script ya era el sitio de los puntos y de las vidas. En un equipo, la misma separación permite que cada persona trabaje en una pieza sin pisar el trabajo de las demás.

Para mantenerla, pon nombres que expliquen la intención y desconfía de un script que crece sin parar. Si para añadir un efecto visual estás copiando movimiento, daño y aparición en el mismo archivo, estás mezclando responsabilidades.

Para llevar a la jam

Antes de llegar, comprueba que tienes instalada la versión de Unity que vas a usar y una plantilla propia que abre sin errores. Durante la jam, lleva al día una lista con el origen y la licencia de cada recurso que no sea tuyo y con lo que hayas hecho con IA. Pide a la IA cambios pequeños, lee cada uno antes de aplicarlo y pruébalo jugando. Deja el pulido para cuando las reglas funcionen, y añádelo de efecto en efecto.

Para seguir practicando, elige uno de los retos de este material (09) y comparte el resultado en Discord con dos frases: «He cambiado…» y «Sé que funciona porque…».

09 · Cinco retos para seguir en casa

Escoge **uno**, no los cinco a la vez. Reserva entre 30 y 60 minutos como orientación; los retos 4 y 5 exigen más montaje. Si te atascas, abre primero una pista y prueba de nuevo. El objetivo es poder enseñar el resultado y explicar qué cambiaste. Las soluciones completas de los retos con scripts están en `Retos/` del paquete.

DESDE PC1 · 30 MIN · MONTAJE

1 · Dos obstáculos

Añade dos obstáculos interiores sin escribir código. Queremos entender que imagen y colisión son componentes diferentes.

Funciona cuando: Se ven dos obstáculos y, a partir de PC3, la nave se detiene al intentar atravesarlos. En PC1 sólo puedes comprobar su geometría.

▸ **Pista 1 · Dónde mirar**

Un muro ya tiene el aspecto y el collider necesarios. Duplicarlo conserva sus componentes.

▸ **Pista 2 · Cómo avanzar**

Cambiar la imagen no cambia necesariamente el tamaño de la colisión. Ajusta SpriteRenderer y BoxCollider2D.

▸ **Solución · Abrir después de intentarlo**

Duplica un muro como Obstacle1 en (−3,1,0), escala (1,1,1), rotación cero. Usa Draw Mode Tiled, tamaño (2,0.6), y el mismo tamaño en BoxCollider2D, sin Is Trigger ni Rigidbody2D. Duplica como Obstacle2 en (3,−2,0). Mantén beamLong1 y el color azul de los muros. Si combinas el reto del cristal, crea la capa física Obstacles y asígnala a ambos. Prueba sus colisiones en una copia de PC3 o posterior.

DESDE PC2 · 30 MIN · PRIMERA LÍNEA

2 · Una velocidad con límite

Permite experimentar con Speed sin superar ocho unidades por segundo ni mover la nave al revés al escribir un número negativo.

Funciona cuando: Con Speed 500, la rapidez queda en 8, también en diagonal. Con Speed −5, la nave no se mueve.

<details>
<summary>Pista 1 · Dónde mirar</summary>

No cambies cada tecla. Todas acaban llamando a Move: ese es el único lugar que necesitas modificar.
</details>

<details>
<summary>Pista 2 · Cómo avanzar</summary>

Mathf.Clamp(valor, mínimo, máximo) devuelve el valor limitado entre esos dos extremos.
</details>

<details>
<summary>Solución · Abrir después de intentarlo</summary>

En PC2 sustituye sólo la línea de Move por la primera de abajo. En PC3–PC5 usa la segunda, sin deltaTime. Son alternativas según el punto de control, no dos líneas para ejecutar juntas. Deja Speed en 5 después de experimentar.

```
transform.position += (Vector3)direction.normalized * Mathf.Clamp(speed, 0f, 8f) * Time.deltaTime;

// Alternativa para PC3, PC4 o PC5:
body.linearVelocity = direction.normalized * Mathf.Clamp(speed, 0f, 8f);
```
</details>

DESDE PC3 · 45–60 MIN · REFERENCIAS Y CONDICIONES

3 · Un cristal que deja espacio

Al recogerlo, haz que el cristal aparezca a dos unidades o más de la nave. Conserva su posición dentro de la arena.

Funciona cuando: Tras varias recogidas, incluida una cerca de las esquinas, el cristal queda dentro de la arena y separado de la nave.

▸ **Pista 1 · Dónde mirar**

Crystal necesita saber dónde está Player. Añade una referencia Transform y enlázala en Inspector.

▸ **Pista 2 · Cómo avanzar**

Genera una posición y comprueba Vector2.Distance antes de aceptarla. Limita los intentos para no quedarte en un bucle sin salida.

▸ **Solución · Abrir después de intentarlo**

Sustituye Crystal.cs por la solución de Retos/Crystal.cs y enlaza Player. Deja Limits (7.2,3.7), Minimum Distance 2 y Obstacles Nothing si no añadiste obstáculos. Si hiciste el reto 1, selecciona sólo la capa física Obstacles. La solución prueba 20 posiciones y descarta las cercanas o solapadas; después utiliza la esquina opuesta. Esa salida vale para esta arena y estos valores, no para cualquier distribución de muros o distancia mínima. Revisa el diseño si los cambias.

Solución completa · Crystal.cs

```
// EXTREJAM 2026 · Reto opcional · Crystal.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class Crystal : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private Vector2 limits = new Vector2(7.2f, 3.7f);
    [SerializeField] private Transform player;
    [SerializeField] private float minimumDistance = 2f;
    [SerializeField] private LayerMask obstacles;

    // El cristal decide su nueva posición; el jugador sólo le pide que se recoloque.
    public void Relocate()
    {
        // Acotar los intentos evita bloquear el juego si cambia la arena.
        for (int attempt = 0; attempt < 20; attempt++)
        {
            Vector2 position = new Vector2(Random.Range(-limits.x, limits.x), Random.Range(-limits.y,
limits.y));
            if (Vector2.Distance(position, player.position) < minimumDistance) continue;
            if (Physics2D.OverlapCircle(position, 0.4f, obstacles) != null) continue;
            transform.position = position;
            return;
        }
        // En esta arena las cuatro esquinas interiores están libres de obstáculos.
        transform.position = new Vector2(player.position.x >= 0 ? -limits.x : limits.x,
            player.position.y >= 0 ? -limits.y : limits.y);
    }
}
```

DESDE PC4 · 45–60 MIN · PREFABS

4 · Tres tipos de meteorito

Haz que cada aparición elija entre un meteorito grande, mediano o pequeño, con distinto tamaño de colisión y rapidez.

Funciona cuando: Aparecen las tres variantes, sus colliders encajan y mantienen la dirección inicial aunque tengan diferente rapidez.

<details>
<summary>Pista 1 · Dónde mirar</summary>

El prefab es el lugar común a todas las apariciones. Un componente separado evita agrandar MeteorSpawner.
</details>

<details>
<summary>Pista 2 · Cómo avanzar</summary>

Elige un índice y úsalo para imagen, radio y rapidez. No cambies la dirección que ya calculó Launch.
</details>

<details>
<summary>Solución · Abrir después de intentarlo</summary>

Añade Retos/MeteorVariation.cs al prefab Meteor. Importa meteorBrown_med1.png y meteorBrown_small1.png de RecursosCC0 como Sprite Single, 100 PPU, Full Rect, Bilinear, Compression None y sin mipmaps. Enlaza Sprite Renderer, Circle y Body con los componentes del prefab. Sprites lleva exactamente tres entradas: big1, med1 y small1. Con Speed 3 de Meteor, resultan radios 0.42, 0.21 y 0.12, y velocidades 2.4, 3.3 y 4.5. Start elige después de Launch; si más adelante reutilizas objetos, deberás ejecutar la elección en cada nuevo lanzamiento.
</details>

Solución completa · MeteorVariation.cs

```
// EXTREJAM 2026 · Reto opcional · MeteorVariation.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class MeteorVariation : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private SpriteRenderer spriteRenderer;
    [SerializeField] private CircleCollider2D circle;
    [SerializeField] private Rigidbody2D body;
    [SerializeField] private Sprite[] sprites;

    // Unity llama a Start antes del primer Update del componente. No se repite por cada reactivación.
    private void Start()
    {
        // Launch ya ha fijado la dirección; sólo cambiamos la rapidez.
        int variant = Random.Range(0, sprites.Length);
        spriteRenderer.sprite = sprites[variant];
        circle.radius = variant == 0 ? 0.42f : variant == 1 ? 0.21f : 0.12f;
        body.linearVelocity *= variant == 0 ? 0.8f : variant == 1 ? 1.1f : 1.5f;
    }
}
```

DESDE PC5 · 45–60 MIN · CONECTAR COMPORTAMIENTOS

5 · Un escudo de un impacto

Añade una recogida que aparece tras 12 segundos y protege durante 5 segundos o un solo impacto, lo que ocurra antes.

Funciona cuando: El primer impacto con escudo no resta vida, el siguiente sí. También caduca si no recibes impactos. Después espera 12 segundos para reaparecer.

▸ **Pista 1 · Dónde mirar**

Separa la aparición y duración del escudo en su propio componente. GameManager sólo necesita preguntar si bloquea el daño.

▸ **Pista 2 · Cómo avanzar**

Desactivar todo el objeto impide que su Update avance el temporizador. Oculta sólo el sprite y desactiva su collider mientras espera.

▸ **Solución · Abrir después de intentarlo**

Importa powerupBlue_shield.png y shield3.png de RecursosCC0 con los mismos ajustes del reto 4. Crea ShieldPickup en (0,2,0), sprite powerupBlue_shield, orden 2, CircleCollider2D trigger de radio 0.3 y sin Rigidbody2D. Añade Retos/ShieldController.cs. Desactiva sólo SpriteRenderer.enabled y Collider.enabled; conserva el GameObject activo. Crea ShieldBubble como hijo de Player, posición local cero, escala uno, sprite shield3, orden 4 y objeto inactivo. En ShieldController enlaza Player, Pickup (su SpriteRenderer), Pickup Collider y Bubble (ShieldBubble); Interval 12 y Duration 5. Añade el campo siguiente a GameManager y enlázalo con ShieldPickup. Dentro del TakeDamage existente, después de la comprobación de vidas cero y antes de lives--, añade la condición indicada. El meteorito se destruye por el contacto incluso si se bloquea el daño. La pausa del juego detiene los temporizadores.

```
[SerializeField] private ShieldController shield;

// Dentro de TakeDamage, después de if (lives == 0) return;
// y antes de lives--:
if (shield.TryBlock()) return;
```

Solución completa · ShieldController.cs

```
// EXTREJAM 2026 · Reto opcional · ShieldController.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class ShieldController : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private Transform player;
    [SerializeField] private SpriteRenderer pickup;
    [SerializeField] private Collider2D pickupCollider;
    [SerializeField] private GameObject bubble;
    [SerializeField] private float interval = 12f;
    [SerializeField] private float duration = 5f;
    private float timer;
    private float remaining;

    // Unity llama a Update cada fotograma con este componente activo. Revisa el estado o la entrada para
decidir si actuar.
    private void Update()
    {
        if (remaining > 0)
        {
            remaining -= Time.deltaTime;
            if (remaining <= 0) bubble.SetActive(false);
            return;
        }
        if (pickup.enabled) return;
        // Acumulamos tiempo del juego; con timeScale a cero deltaTime no hace avanzar el contador.
        timer += Time.deltaTime;
        if (timer < interval) return;
        timer = 0;
        pickup.enabled = true;
        pickupCollider.enabled = true;
    }

    // Unity avisa al entrar en un trigger 2D. other es el collider del otro objeto, no una copia de él.
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.transform != player) return;
        pickup.enabled = false;
        pickupCollider.enabled = false;
        remaining = duration;
        bubble.SetActive(true);
    }

    // Un bool expresa sí o no. Consumir la protección ahora impide que bloquee dos impactos.
    public bool TryBlock()
    {
        if (remaining <= 0) return false;
        // Consumirlo antes del siguiente contacto impide bloquear dos impactos.
        remaining = 0;
        bubble.SetActive(false);
        return true;
    }
}
```

10 · Si ya sabes programar: dos ampliaciones

A · Dificultad configurable · 30–45 minutos. Parte de una copia de PC5. Separa tres datos de la fórmula: intervalo inicial, intervalo mínimo y reducción por punto. Mantén el cálculo en GameManager y la creación de meteoritos en MeteorSpawner. No necesitas un menú ni una nueva clase para probar la idea.

> ▸ **Pista 1 · Qué cambia y qué permanece**
>
> Los valores 1.5, 0.4 y 0.05 son parámetros. La operación Max y la relación con score expresan la regla. Cambia los datos desde Inspector, manteniendo el comportamiento original como valores iniciales.

> ▸ **Pista 2 · Evitar configuraciones incoherentes**
>
> El intervalo mínimo debe ser positivo y no superar al inicial. La reducción no debe ser negativa. Normaliza esos datos al calcular el resultado, sin añadir lógica de dificultad a Meteor.

> ▸ **Solución de referencia · Sustituir GetSpawnInterval y añadir estos campos**

```
[SerializeField] private float initialInterval = 1.5f;
[SerializeField] private float minimumInterval = 0.4f;
[SerializeField] private float reductionPerPoint = 0.05f;

public float GetSpawnInterval()
{
    float initial = Mathf.Max(0.05f, initialInterval);
    float minimum = Mathf.Clamp(minimumInterval, 0.05f, initial);
    float reduction = Mathf.Max(0f, reductionPerPoint);
    return Mathf.Max(minimum, initial - score * reduction);
}
```

> Es un reemplazo del método existente, no un segundo método con el mismo nombre. Con los valores iniciales, 0, 10 y 22 puntos deben producir 1.5, 1 y 0.4 segundos. Prueba una configuración suave cambiando un dato cada vez. Esta ampliación es una propuesta para tu copia, no está aplicada a PC5.

B · Reutilizar meteoritos · 45–60 minutos para quien ya tenga base. Construye un conjunto de meteoritos que puedan prestarse y devolverse. El objetivo es entender su ciclo de vida; no asumimos que este juego necesite optimización ni prometemos una mejora de rendimiento sin medirla.

> ▸ **Pista 1 · Busca las entradas y salidas**
>
> El spawner usa Instantiate. Hay dos salidas: Meteor se destruye al abandonar la arena y PlayerController destruye el meteorito al tocarlo. Las dos deben devolverlo al conjunto. Si cambias sólo una, seguirás perdiendo objetos.

▸ Pista 2 · Un objeto reutilizado tiene memoria

Antes de cada préstamo reinicia posición, velocidad y cualquier apariencia temporal. Start no se repite al reactivar un objeto ya iniciado. Si combinaste el reto de tamaños, traslada su selección a una operación que llames en cada lanzamiento.

▸ Solución de diseño · Contrato entre componentes

```
MeteorPool: mantiene disponibles y sabe qué meteoritos ha prestado.
Pedir(posición, dirección):
    usar uno disponible; si no hay, crear otro y registrar su propietario
    preparar su estado en inactivo; activar; lanzar con la nueva dirección
Devolver(meteorito):
    si ya estaba disponible, no devolverlo una segunda vez
    detener su velocidad; desactivar; añadir a disponibles
MeteorSpawner: pide al conjunto en lugar de Instantiate.
Meteor: devuelve al salir de los límites.
PlayerController: tras resolver el daño, devuelve al tocarlo.
```

Este bloque es pseudocódigo para implementar, no C# listo para pegar. Usa referencias explícitas al propietario, evita un gestor global y encapsula la devolución en Meteor para no acoplar PlayerController a la estructura del conjunto. Si el conjunto está vacío, hacerlo crecer mantiene las apariciones; omitir el meteorito cambiaría la dificultad. Comprueba que una nueva aparición sale del borde correcto, que no conserva velocidad anterior y que un impacto no devuelve dos veces el mismo objeto. La solución no está incorporada ni verificada en los proyectos base.

11 · IA: pedir un cambio que puedas explicar

Actividad · 30 minutos. Trabaja en una copia de PC5 y utiliza el asistente que tengas disponible. El objetivo es hacer configurable el número de vidas iniciales entre 1 y 5, manteniendo 3 como valor predeterminado. Sólo cambia GameManager. No necesitas instalar herramientas de IA dentro de Unity.

Antes de consultar, explica dónde están las vidas y cuándo comienza la partida. Entrega al asistente el GameManager actual y este encargo. Lee su propuesta antes de pegarla; si no entiendes una línea, pide que la explique en relación con tu archivo.

```
Este es mi GameManager de un ejercicio de Unity 6000.3.27f1.
Quiero editar las vidas iniciales desde Inspector entre 1 y 5,
con 3 por defecto. Al cargar la escena, lives debe tomar ese valor.
Conserva la puntuación, el récord, el audio, la pausa y el reinicio.
Modifica sólo GameManager y usa un campo privado serializado.
Protege también valores fuera de rango en código.
Devuelve un cambio pequeño indicando dónde va cada línea,
explica por qué se inicializa antes de UpdateHud y propone
tres comprobaciones observables. No añadas paquetes ni gestores.
A continuación pego mi archivo completo:
```

Decide: ¿la respuesta respeta un solo archivo? ¿Inicializa vidas antes de mostrar el HUD? ¿Conserva el valor predeterminado y las reglas? Si propone reescribir cinco scripts o generar todo el juego, vuelve a acotar la petición. El aspecto convincente de una respuesta no demuestra que funcione.

▸ **Ayuda · Una respuesta pequeña que puedes revisar sin conexión**

Añade este campo a GameManager, junto al resto de campos. Conserva `private int lives = 3;`.

```
[SerializeField] private int startingLives = 3;
```

Como primera instrucción dentro del Start existente, antes de UpdateHud, añade:

```
lives = Mathf.Clamp(startingLives, 1, 5);
```

No crees un segundo Start. La función limita el valor usado durante la partida; el campo del Inspector puede seguir mostrando el número que escribiste. Esta es una referencia para revisar la propuesta de la IA, no un cambio aplicado al punto de control.

Fuera de Play, escribe 1 en Starting Lives, guarda la escena y juega: un impacto debe terminar la partida. Escribe 5 fuera de Play, guarda la escena y comprueba que empiezas con cinco; reiniciar debe mostrar de nuevo el título con cinco y cero puntos. Por último prueba 0 y 99, guardando la escena antes de cada entrada en Play: el juego debe usar 1 y 5 respectivamente. Devuelve el valor a 3 al terminar y verifica que recogida, audio y récord siguen funcionando.

Entrega al chat: «Pedí…, acepté estas dos líneas porque…, lo comprobé haciendo…». Si una respuesta no funcionó, explica qué observaste y qué corregiste. Puedes completar la actividad con la respuesta de referencia si no tienes acceso a un asistente.

12 · Tu siguiente sesión de práctica

Elige el reto que se ajuste a tu punto de partida. Si empiezas desde cero, obstáculos o límite de velocidad. Si ya te orientas con componentes, distancia del cristal. Si manejas referencias y eventos, variantes o escudo. Si ya programas, dificultad configurable o conjunto reutilizable. Son alternativas: no necesitas terminar todas para seguir la formación.

En Discord comparte una captura o un vídeo corto, el punto de control del que partiste y dos frases: «He cambiado…» y «Sé que funciona porque…». No hace falta un informe técnico. Si necesitas ayuda, indica el objeto seleccionado y el campo o función que estás cambiando.

Un hábito para quedarte: pon nombres que expliquen la intención y da a cada función una tarea concreta. Si para añadir una señal visual estás copiando movimiento, daño y aparición al mismo archivo, revisa qué responsabilidad estás mezclando. Puedes pedir a la IA que explique una función o que proponga un cambio acotado; tú decides si conserva las reglas.

13 · Código comentado para consultar

Estos archivos reproducen el código de los puntos de control con comentarios didácticos adicionales. Usa **la carpeta de tu PC**: las versiones son alternativas, no scripts que debas añadir todos a la vez. Sustituye el archivo con el mismo nombre en tu copia del proyecto; no crees dos clases PlayerController. Un comentario explica el propósito, pero no se ejecuta.

Lee primero los comentarios del script que estás modificando. No necesitas memorizar el apéndice ni estudiar todas las versiones durante el directo. Las palabras nuevas aparecen junto a su uso: campo, referencia, condición, método y parámetro.

▸ PC2 · PlayerController.cs

```
// EXTREJAM 2026 · PC2 · PlayerController.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;
using UnityEngine.InputSystem;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class PlayerController : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private float speed = 5f;

    // Unity llama a Update cada fotograma con este componente activo. Revisa el estado o la entrada para decidir
si actuar.
    private void Update()
    {
        Vector2 direction = Vector2.zero;
        // Una variable local sólo se usa en esta llamada. Puede no haber teclado: la siguiente condición lo
comprueba.
        Keyboard keyboard = Keyboard.current;
        if (keyboard == null) return;
        if (keyboard.wKey.isPressed || keyboard.upArrowKey.isPressed) direction.y += 1;
        if (keyboard.sKey.isPressed || keyboard.downArrowKey.isPressed) direction.y -= 1;
        if (keyboard.dKey.isPressed || keyboard.rightArrowKey.isPressed) direction.x += 1;
        if (keyboard.aKey.isPressed || keyboard.leftArrowKey.isPressed) direction.x -= 1;
        // Normalizar evita que una diagonal sea más rápida.
        Move(direction);
    }

    // Un parámetro recibe el dato de la llamada. Esta función concentra la decisión de cómo mover la nave.
    public void Move(Vector2 direction)
    {
        // deltaTime mantiene la velocidad al cambiar los fotogramas por segundo.
        transform.position += (Vector3)direction.normalized * speed * Time.deltaTime;
    }
}
```

▸ PC3 · Crystal.cs

```
// EXTREJAM 2026 · PC3 · Crystal.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class Crystal : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private Vector2 limits = new Vector2(7.2f, 3.7f);

    // El cristal decide su nueva posición; el jugador sólo le pide que se recoloque.
    public void Relocate()
    {
        // El margen mantiene el sprite lejos de los muros.
        transform.position = new Vector2(Random.Range(-limits.x, limits.x), Random.Range(-limits.y, limits.y));
    }
}
```

▸ PC3 · PlayerController.cs

```
// EXTREJAM 2026 · PC3 · PlayerController.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;
using UnityEngine.InputSystem;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class PlayerController : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private float speed = 5f;
    [SerializeField] private Rigidbody2D body;
    private int score;

    // Unity llama a Update cada fotograma con este componente activo. Revisa el estado o la entrada para decidir
si actuar.
    private void Update()
    {
        Vector2 direction = Vector2.zero;
        // Una variable local sólo se usa en esta llamada. Puede no haber teclado: la siguiente condición lo
comprueba.
        Keyboard keyboard = Keyboard.current;
        if (keyboard == null) return;
        if (keyboard.wKey.isPressed || keyboard.upArrowKey.isPressed) direction.y += 1;
        if (keyboard.sKey.isPressed || keyboard.downArrowKey.isPressed) direction.y -= 1;
        if (keyboard.dKey.isPressed || keyboard.rightArrowKey.isPressed) direction.x += 1;
        if (keyboard.aKey.isPressed || keyboard.leftArrowKey.isPressed) direction.x -= 1;
        // Normalizar evita que una diagonal sea más rápida.
        Move(direction);
    }

    // Un parámetro recibe el dato de la llamada. Esta función concentra la decisión de cómo mover la nave.
    public void Move(Vector2 direction)
    {
        // Velocidad en unidades/segundo: la física integra el movimiento, por eso aquí no multiplicamos por
deltaTime.
        body.linearVelocity = direction.normalized * speed;
    }

    // Unity avisa al entrar en un trigger 2D. other es el collider del otro objeto, no una copia de él.
    private void OnTriggerEnter2D(Collider2D other)
    {
        // if ejecuta el bloque sólo cuando se cumple esta condición. La etiqueta debe existir y estar asignada.
        if (other.CompareTag("Crystal"))
        {
        // ++ suma exactamente uno. El punto y coma termina la instrucción.
            score++;
            Debug.Log("Cristales: " + score);
            other.GetComponent<Crystal>().Relocate();
        }
    }
}
```

▸ PC4 · Crystal.cs

```
// EXTREJAM 2026 · PC4 · Crystal.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class Crystal : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private Vector2 limits = new Vector2(7.2f, 3.7f);

    // El cristal decide su nueva posición; el jugador sólo le pide que se recoloque.
    public void Relocate()
    {
        // El margen mantiene el sprite lejos de los muros.
        transform.position = new Vector2(Random.Range(-limits.x, limits.x), Random.Range(-limits.y, limits.y));
    }
}
```

▸ PC4 · Meteor.cs

```
// EXTREJAM 2026 · PC4 · Meteor.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class Meteor : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private Rigidbody2D body;
    [SerializeField] private float speed = 3f;
    [SerializeField] private Vector2 limits = new Vector2(10f, 6.5f);

    // Recibe una dirección ya calculada. Guardar la velocidad una vez produce un trayecto recto.
    public void Launch(Vector2 direction)
    {
        // La dirección se fija al nacer: el meteorito no persigue después.
        body.linearVelocity = direction * speed;
    }

    // Unity llama a Update cada fotograma con este componente activo. Revisa el estado o la entrada para decidir
si actuar.
    private void Update()
    {
        if (Mathf.Abs(transform.position.x) > limits.x || Mathf.Abs(transform.position.y) > limits.y)
            Destroy(gameObject);
    }
}
```

▸ PC4 · MeteorSpawner.cs

```
// EXTREJAM 2026 · PC4 · MeteorSpawner.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class MeteorSpawner : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private GameObject meteorPrefab;
    [SerializeField] private Transform player;
    [SerializeField] private float interval = 1.5f;
    [SerializeField] private Vector2 edges = new Vector2(9f, 5.5f);
    private float timer;

    // Unity llama a Update cada fotograma con este componente activo. Revisa el estado o la entrada para decidir
si actuar.
    private void Update()
    {
        // Acumulamos tiempo del juego; con timeScale a cero deltaTime no hace avanzar el contador.
        timer += Time.deltaTime;
        if (timer < interval) return;
        timer = 0;
        Spawn();
    }

    // Crear un meteorito y darle sus datos es responsabilidad del spawner, no del jugador.
    public void Spawn()
    {
        Vector2 position = new Vector2(Random.Range(-edges.x, edges.x), Random.Range(-edges.y, edges.y));
        int edge = Random.Range(0, 4);
        if (edge == 0) position.x = -edges.x;
        if (edge == 1) position.x = edges.x;
        if (edge == 2) position.y = -edges.y;
        if (edge == 3) position.y = edges.y;
        GameObject meteor = Instantiate(meteorPrefab, position, Quaternion.identity);
        // El prefab no puede guardar una referencia a la nave de esta escena.
        meteor.GetComponent<Meteor>().Launch(((Vector2)player.position - position).normalized);
    }
}
```

▸ PC4 · PlayerController.cs

```
// EXTREJAM 2026 · PC4 · PlayerController.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;
using UnityEngine.InputSystem;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class PlayerController : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private float speed = 5f;
    [SerializeField] private Rigidbody2D body;
    private int score;
    private int lives = 3;

    // Unity llama a Update cada fotograma con este componente activo. Revisa el estado o la entrada para decidir
si actuar.
    private void Update()
    {
        Vector2 direction = Vector2.zero;
        // Una variable local sólo se usa en esta llamada. Puede no haber teclado: la siguiente condición lo
comprueba.
        Keyboard keyboard = Keyboard.current;
        if (keyboard == null) return;
        if (keyboard.wKey.isPressed || keyboard.upArrowKey.isPressed) direction.y += 1;
        if (keyboard.sKey.isPressed || keyboard.downArrowKey.isPressed) direction.y -= 1;
        if (keyboard.dKey.isPressed || keyboard.rightArrowKey.isPressed) direction.x += 1;
        if (keyboard.aKey.isPressed || keyboard.leftArrowKey.isPressed) direction.x -= 1;
        // Normalizar evita que una diagonal sea más rápida.
        Move(direction);
    }

    // Un parámetro recibe el dato de la llamada. Esta función concentra la decisión de cómo mover la nave.
    public void Move(Vector2 direction)
    {
        // Velocidad en unidades/segundo: la física integra el movimiento, por eso aquí no multiplicamos por
deltaTime.
        body.linearVelocity = direction.normalized * speed;
    }

    // Unity avisa al entrar en un trigger 2D. other es el collider del otro objeto, no una copia de él.
    private void OnTriggerEnter2D(Collider2D other)
    {
        // if ejecuta el bloque sólo cuando se cumple esta condición. La etiqueta debe existir y estar asignada.
        if (other.CompareTag("Crystal"))
        {
        // ++ suma exactamente uno. El punto y coma termina la instrucción.
            score++;
            Debug.Log("Cristales: " + score);
            other.GetComponent<Crystal>().Relocate();
        }
        // if ejecuta el bloque sólo cuando se cumple esta condición. La etiqueta debe existir y estar asignada.
        if (other.CompareTag("Meteor"))
        {
            lives--;
            Debug.Log("Vidas: " + lives);
            Destroy(other.gameObject);
        }
    }
}
```

▸ PC5 · Crystal.cs

```
// EXTREJAM 2026 · PC5 · Crystal.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class Crystal : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private Vector2 limits = new Vector2(7.2f, 3.7f);

    // El cristal decide su nueva posición; el jugador sólo le pide que se recoloque.
    public void Relocate()
    {
        // El margen mantiene el sprite lejos de los muros.
        transform.position = new Vector2(Random.Range(-limits.x, limits.x), Random.Range(-limits.y, limits.y));
    }
}
```

▸ PC5 · GameManager.cs

```
// EXTREJAM 2026 · PC5 · GameManager.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;
using UnityEngine.SceneManagement;
using TMPro;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class GameManager : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private TMP_Text scoreText;
    [SerializeField] private TMP_Text livesText;
    [SerializeField] private TMP_Text resultText;
    [SerializeField] private GameObject titlePanel;
    [SerializeField] private GameObject defeatPanel;
    [SerializeField] private AudioSource audioSource;
    [SerializeField] private AudioClip collectSound;
    [SerializeField] private AudioClip hitSound;
    private int score;
    private int lives = 3;

    // Unity llama a Start antes del primer Update del componente. No se repite por cada reactivación.
    private void Start()
    {
        Time.timeScale = 0;
        titlePanel.SetActive(true);
        defeatPanel.SetActive(false);
        UpdateHud();
    }

    // El botón Jugar llama a este método público. Restablecemos el avance del tiempo de juego.
    public void Play()
    {
        titlePanel.SetActive(false);
        Time.timeScale = 1;
    }

    // El estado de la partida y sus señales se actualizan juntos. No duplicamos el contador en el jugador.
    public void AddPoint()
    {
        if (lives == 0) return;
        // ++ suma exactamente uno. El punto y coma termina la instrucción.
        score++;
        audioSource.PlayOneShot(collectSound);
        UpdateHud();
    }

    // Devuelve un número; no crea meteoritos. Un intervalo menor aumenta la frecuencia de aparición.
    public float GetSpawnInterval()
    {
        return Mathf.Max(0.4f, 1.5f - score * 0.05f);
    }

    // La regla de daño vive aquí. Comprobar vidas cero evita seguir resolviendo daño después de perder.
    public void TakeDamage()
    {
        if (lives == 0) return;
        lives--;
        audioSource.PlayOneShot(hitSound);
        UpdateHud();
        if (lives == 0) EndGame();
    }

    // Los textos muestran el estado actual. Cambiar un texto no debe convertirse en otra fuente de puntos o
vidas.
```

```
    private void UpdateHud()
    {
        scoreText.text = "CRISTALES  " + score;
        livesText.text = "VIDAS  " + lives;
    }

    // Cerramos la partida, guardamos el récord local y enseñamos el resultado antes de pausar.
    private void EndGame()
    {
        int record = Mathf.Max(score, PlayerPrefs.GetInt("Record", 0));
        PlayerPrefs.SetInt("Record", record);
        // Guardamos ahora para conservar el récord aunque se cierre la ventana.
        PlayerPrefs.Save();
        resultText.text = "Cristales: " + score + "\nRécord: " + record;
        defeatPanel.SetActive(true);
        Time.timeScale = 0;
    }

    // Recargar la escena vuelve a ejecutar su inicialización; se mostrará de nuevo el título.
    public void Restart()
    {
        // La pausa pertenece a la partida que termina, no al proceso de carga.
        Time.timeScale = 1;
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
}
```

▸ PC5 · Meteor.cs

```
// EXTREJAM 2026 · PC5 · Meteor.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class Meteor : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private Rigidbody2D body;
    [SerializeField] private float speed = 3f;
    [SerializeField] private Vector2 limits = new Vector2(10f, 6.5f);

    // Recibe una dirección ya calculada. Guardar la velocidad una vez produce un trayecto recto.
    public void Launch(Vector2 direction)
    {
        // La dirección se fija al nacer: el meteorito no persigue después.
        body.linearVelocity = direction * speed;
    }

    // Unity llama a Update cada fotograma con este componente activo. Revisa el estado o la entrada para decidir
si actuar.
    private void Update()
    {
        if (Mathf.Abs(transform.position.x) > limits.x || Mathf.Abs(transform.position.y) > limits.y)
            Destroy(gameObject);
    }
}
```

▸ PC5 · MeteorSpawner.cs

```
// EXTREJAM 2026 · PC5 · MeteorSpawner.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class MeteorSpawner : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private GameObject meteorPrefab;
    [SerializeField] private Transform player;
    [SerializeField] private GameManager gameManager;
    [SerializeField] private Vector2 edges = new Vector2(9f, 5.5f);
    private float timer;

    // Unity llama a Update cada fotograma con este componente activo. Revisa el estado o la entrada para decidir
si actuar.
    private void Update()
    {
        // Acumulamos tiempo del juego; con timeScale a cero deltaTime no hace avanzar el contador.
        timer += Time.deltaTime;
        if (timer < gameManager.GetSpawnInterval()) return;
        timer = 0;
        Spawn();
    }

    // Crear un meteorito y darle sus datos es responsabilidad del spawner, no del jugador.
    public void Spawn()
    {
        Vector2 position = new Vector2(Random.Range(-edges.x, edges.x), Random.Range(-edges.y, edges.y));
        int edge = Random.Range(0, 4);
        if (edge == 0) position.x = -edges.x;
        if (edge == 1) position.x = edges.x;
        if (edge == 2) position.y = -edges.y;
        if (edge == 3) position.y = edges.y;
        GameObject meteor = Instantiate(meteorPrefab, position, Quaternion.identity);
        // El prefab no puede guardar una referencia a la nave de esta escena.
        meteor.GetComponent<Meteor>().Launch(((Vector2)player.position - position).normalized);
    }
}
```

▸ PC5 · PlayerController.cs

```
// EXTREJAM 2026 · PC5 · PlayerController.cs
// Versión didáctica: sólo se añaden comentarios al código de referencia.
// Usa esta versión en su punto de control; no añadas varias clases con el mismo nombre.
using UnityEngine;
using UnityEngine.InputSystem;

// Esta clase hereda de MonoBehaviour para poder añadirse como componente a un GameObject.
public class PlayerController : MonoBehaviour
{
    // Campos: datos que conserva este componente. SerializeField permite configurarlos desde Inspector.
    // Si el tipo es un componente u objeto, arrastra su referencia; escribir el campo no realiza el enlace.
    [SerializeField] private float speed = 5f;
    [SerializeField] private Rigidbody2D body;
    [SerializeField] private GameManager gameManager;

    // Unity llama a Update cada fotograma con este componente activo. Revisa el estado o la entrada para decidir
si actuar.
    private void Update()
    {
        Vector2 direction = Vector2.zero;
        // Una variable local sólo se usa en esta llamada. Puede no haber teclado: la siguiente condición lo
comprueba.
        Keyboard keyboard = Keyboard.current;
        if (keyboard == null) return;
        if (keyboard.wKey.isPressed || keyboard.upArrowKey.isPressed) direction.y += 1;
        if (keyboard.sKey.isPressed || keyboard.downArrowKey.isPressed) direction.y -= 1;
        if (keyboard.dKey.isPressed || keyboard.rightArrowKey.isPressed) direction.x += 1;
        if (keyboard.aKey.isPressed || keyboard.leftArrowKey.isPressed) direction.x -= 1;
        // Normalizar evita que una diagonal sea más rápida.
        Move(direction);
    }

    // Un parámetro recibe el dato de la llamada. Esta función concentra la decisión de cómo mover la nave.
    public void Move(Vector2 direction)
    {
        // Velocidad en unidades/segundo: la física integra el movimiento, por eso aquí no multiplicamos por
deltaTime.
        body.linearVelocity = direction.normalized * speed;
    }

    // Unity avisa al entrar en un trigger 2D. other es el collider del otro objeto, no una copia de él.
    private void OnTriggerEnter2D(Collider2D other)
    {
        // if ejecuta el bloque sólo cuando se cumple esta condición. La etiqueta debe existir y estar asignada.
        if (other.CompareTag("Crystal"))
        {
            gameManager.AddPoint();
            other.GetComponent<Crystal>().Relocate();
        }
        // if ejecuta el bloque sólo cuando se cumple esta condición. La etiqueta debe existir y estar asignada.
        if (other.CompareTag("Meteor"))
        {
            gameManager.TakeDamage();
            Destroy(other.gameObject);
        }
    }
}
```

14 · Material y créditos

El paquete del alumnado incluye esta guía HTML, el PDF, las versiones comentadas de los scripts, las tres soluciones de retos y cuatro sprites adicionales con su licencia CC0 de Kenney. No es un proyecto Unity completo: abre el punto de control que descargues de extrejam.com/unity y consulta los archivos que correspondan.

Los recursos gráficos de las prácticas proceden de Kenney Space Shooter Redux y los sonidos básicos de Kenney Sci-fi Sounds. Se conserva la licencia de los recursos adicionales junto a ellos. El código didáctico es material propio de la formación de Viral Studios S.L. Los assets comerciales que se muestran en el pulido pertenecen a esa demostración y no son necesarios para practicar. El ejecutable con pulido de «Órbita Cero» es un juego compilado: incluye los paquetes comerciales dentro del juego, como permite su licencia, pero no sus archivos fuente.

Preparado para Extrejam 2026 · Versión didáctica del 28 de octubre de 2026, posterior a la sesión.

Extrejam 2026 · Viral Studios S.L. · Material para el alumnado · 28/10/2026