Manual de Pilas Engine 2

Contents

Primeros pasos

En esta sección me gustaría describir los pasos iniciales para comenzar a hacer juegos.

Pilas ha sido diseñada especialmente para que todas las personas puedan aprender a programar realizando videojuegos, de forma sencilla y totalmente en español.

Esta es la pantalla inicial que aparecerá cuando abras pilas por primera vez:

Figure 1: portada

Desde aquí se pueden acceder a todas las secciones de pilas, te recomendamos inspeccionarlas al menos una vez para familiarizarte con el entorno.

Ahora bien, la parte más interesante de la herramienta se puede acceder pulsando el botón que dice "**Abrir el editor**"

Desde esta sección, vas a ver una escena principal y varios paneles:

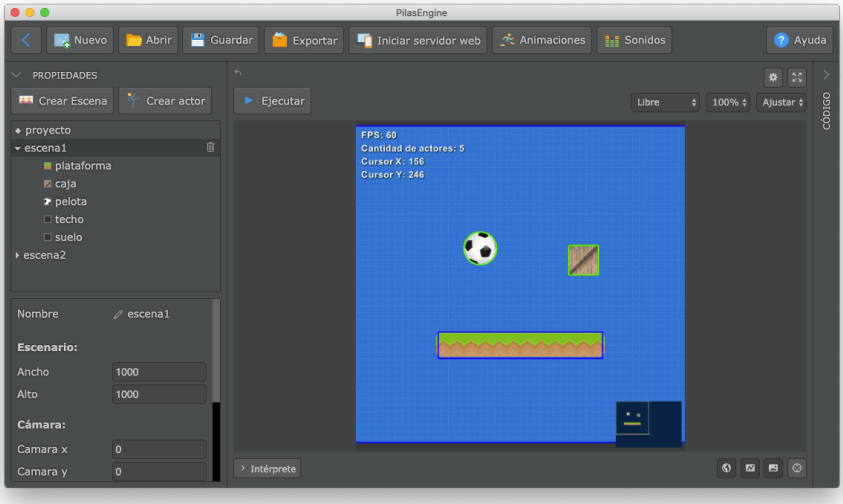


Figure 2: editor-1

Al principio puede parecer un poco abrumador... pero vamos a ir paso a paso. Hay dos partes importantes en esta pantalla:

- A la izquierda está el panel de actores y propiedades. Donde vamos a ver listados todos los actores que aparecen en la pantalla. Los actores en este caso son: una plataforma, una pelota, una caja, un techo y suelo.
- Luego a la derecha de la pantalla tenemos el area del editor:

Hay otras cosas importantes en la interface, pero vamos a prestarle mayor atención a estas dos partes.

Sigamos:

¡Pongamos el juego en funcionamiento!

El botón que aparece arriba del area de juego es muy importante, porque nos permite poner en funcionamiento el juego y hacer una prueba real.

Así que hagamos el intento, pulsa el botón ejecutar una vez:

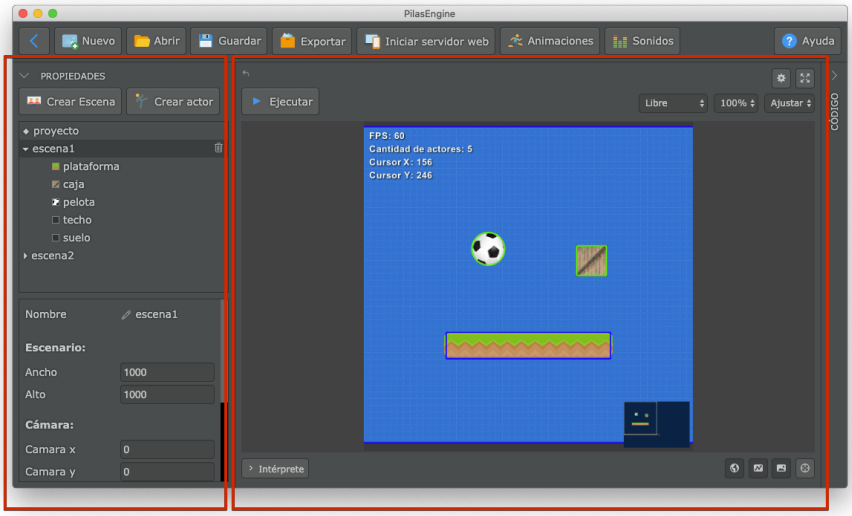


Figure 3: editor-partes

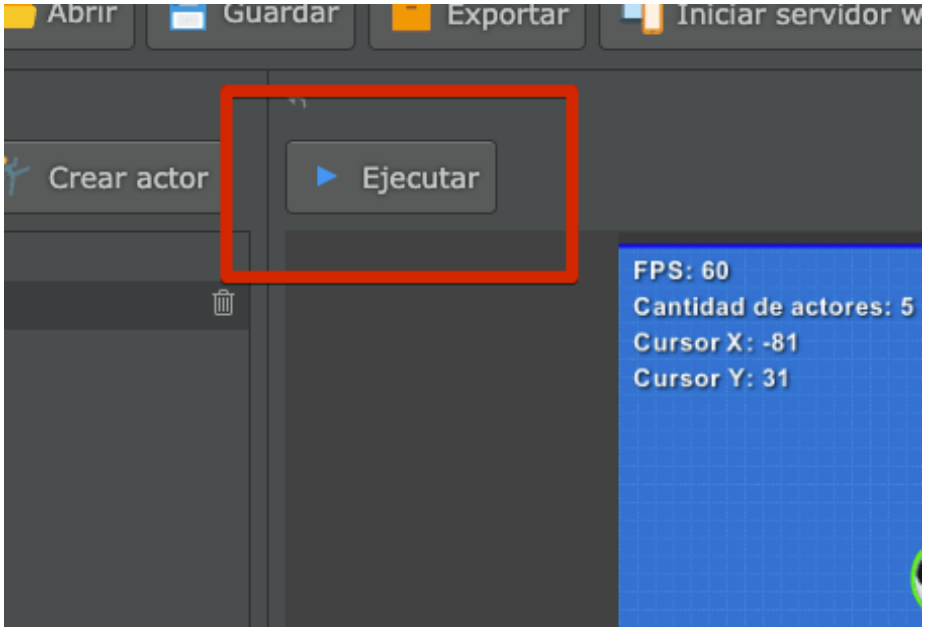


Figure 4: boton-ejecutar

Deberías ver cómo dos de los actores comienzan a rebotar en la plataforma:

Figure 5: primer-ejecucion

Cuando pulsas el botón “ejecutar” además de ponerse en funcionamiento el juego sucede algo más: el editor por completo ingresará en un modo llamado “ejecución”, así que no vamos a poder editar el código o cambiar la escena. Todo lo que suceda en ese momento es parte de la experiencia de usuario de nuestros juegos. Una vez que exportemos el juego, nuestros usuarios solo van a ver el juego, no el editor.

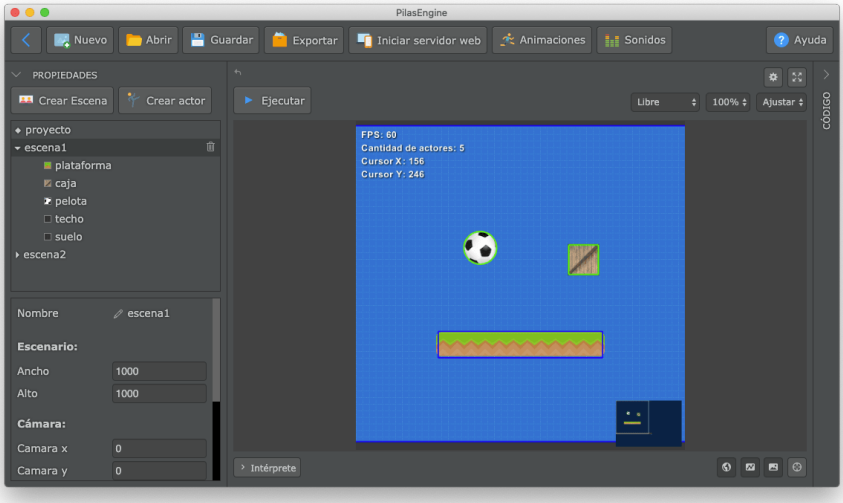
Esto es importante porque nos lleva a pensar en el diseño del juego y hacernos preguntas interesantes: ¿Qué queremos que el usuario haga dentro de nuestro juego?, ¿qué elementos le vamos a mostrar?, ¿cuándo gana y cuando pierde?.

Pero claro, por el momento solo podemos hacernos las preguntas, porque lo que tenemos en el editor es muy poquito, el usuario no puede hacer nada muy interesante aún... solo ver cómo rebotan esos dos objetos en la plataforma :|

Diseñando nuestro primer juego

Para nuestro primer juego vamos a pensar en algo más interesante, queremos que el fondo del juego sea un cielo lleno de estrellas, que el usuario pueda mover una nave con el teclado y que luego de unos segundos comiencen a aparecer algunos enemigos.

Pulsá el botón que dice "detener" en el editor (o pulsá la tecla `Esc`), tendrías que volver a ver el resto de los elementos del editor nuevamente habilitados.



Borremos cada uno de los actores que aparecen en la pantalla, selecciona al actor pelota y luego pulsa el botón del cesto de basura tal y como muestra esta imagen:

Repetí esos pasos hasta que la escena quede limpia, sin ningún actor:

Ahora bien, con la escena completamente limpia, agreguemos un actor para representar a nuestro protagonista:

Pulsa el botón "Crear actor" y luego selecciona la nave:

Una vez que selecciones el actor vas a verlo formar parte de la escena así:

Ahora pulsemos de nuevo el botón "ejecutar" y pulsa las flechitas del teclado para mover la nave por el escenario (¡y con la barra espaciadora vas a poder disparar!):

Ambientando la escena en el espacio

Al principio te mencioné que queríamos hacer que la nave pareciera estar en el espacio, sin embargo aún tenemos ese fondo azul que no se parece mucho al espacio...

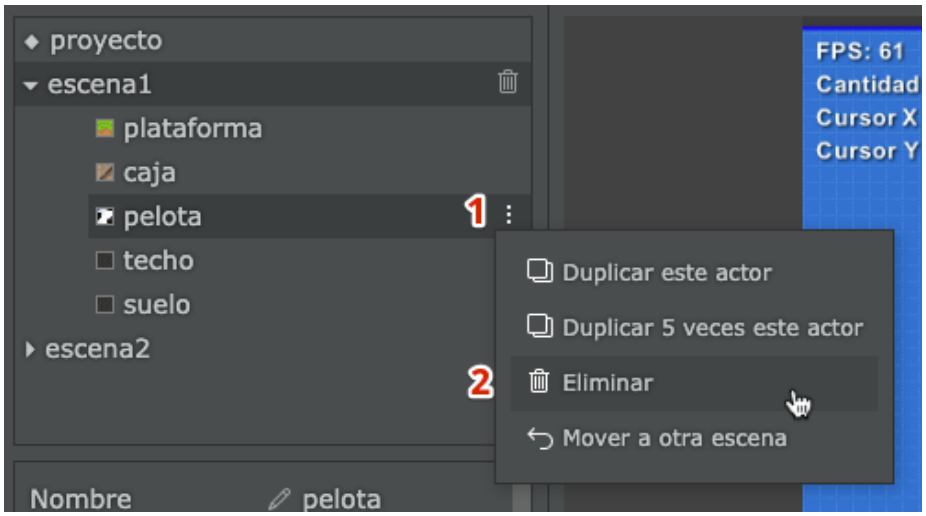


Figure 6: borrar

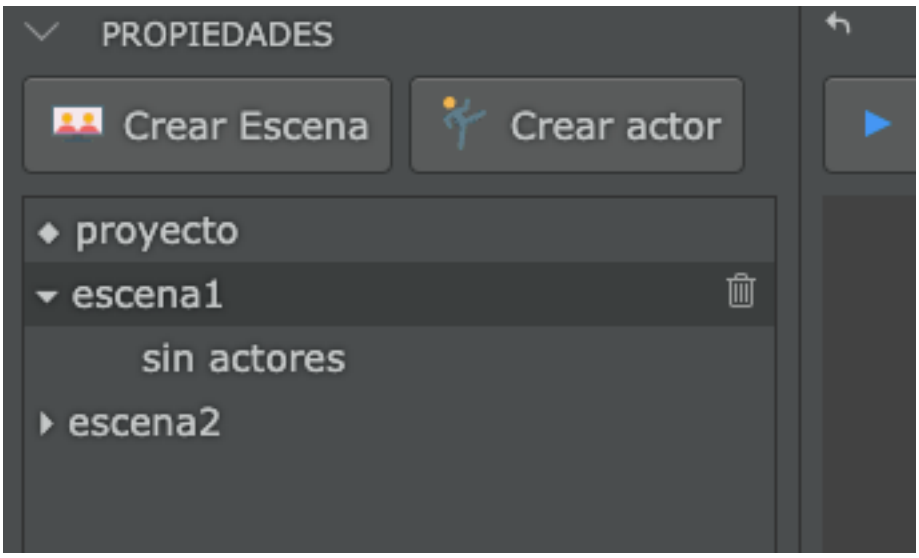


Figure 7: image-20200601230444058

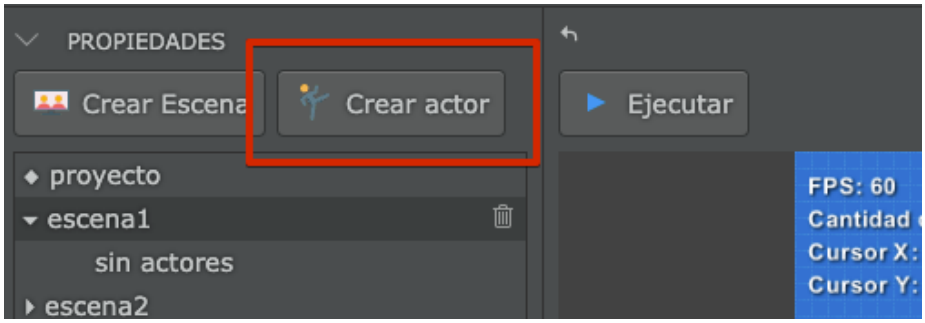


Figure 8: crear-actor-1

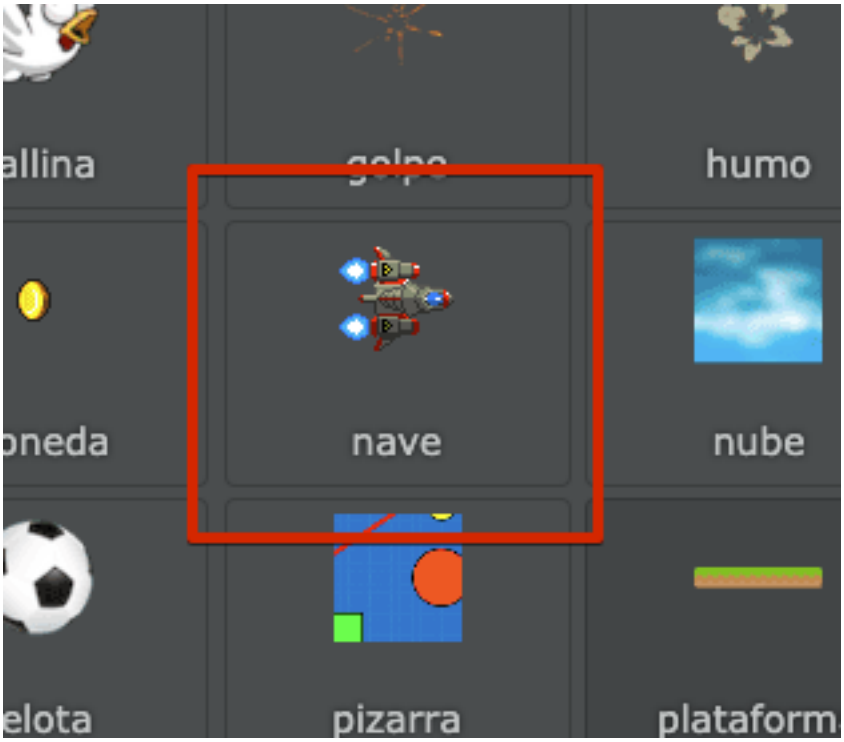


Figure 9: crear-actor-2

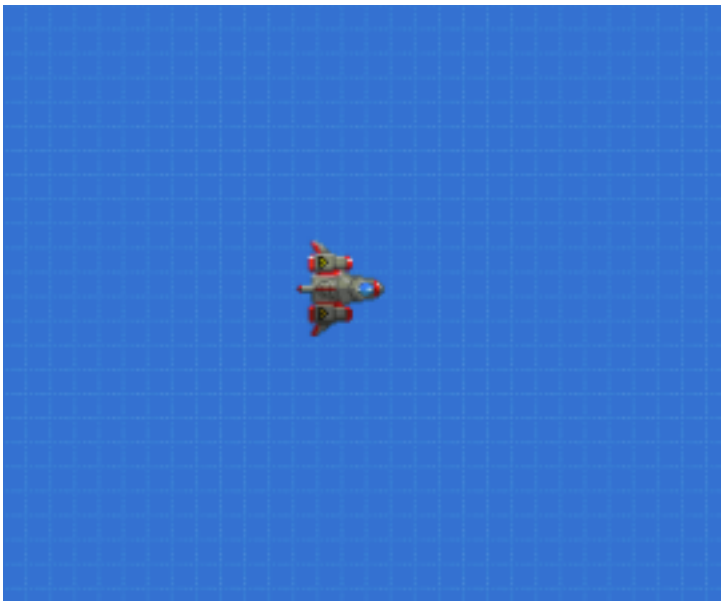


Figure 10: creacion-de-la-nave

Así que hagamos un cambio, selecciona la escena del panel izquierdo, vas a ver que la parte inferior del panel va a cambiar:

Esa sección que marcamos con el número “2” en la imagen contiene todas las propiedades de la escena. Si moves la barra de desplazamiento hacia abajo vas a ver una propiedad llamada fondo:

Hace click sobre esa propiedad, donde está el ícono y vas a ver que pilas te propone varias imágenes para sustituir el fondo:

Selecciona la del espacio:

Vas a notar que el fondo de la pantalla va a cambiar por completo:

Creando más actores

Ahora vamos a darle un poco mas de vida al juego: Nuestra nave está en el espacio pero no parece haber un motivo por el cual está ahí, pensemos una historia:

“A causa de un experimento que se salió de control en la tierra, algunos objetos llegaron al espacio: bananas, manzanas y otras frutas comenzaron a ensuciar el espacio flotando por ahí. Nuestra misión es comandar una nave que tiene como objetivo”limpiar" el desorden de la basura espacial lo antes posible !!!"

Figure 11: nave

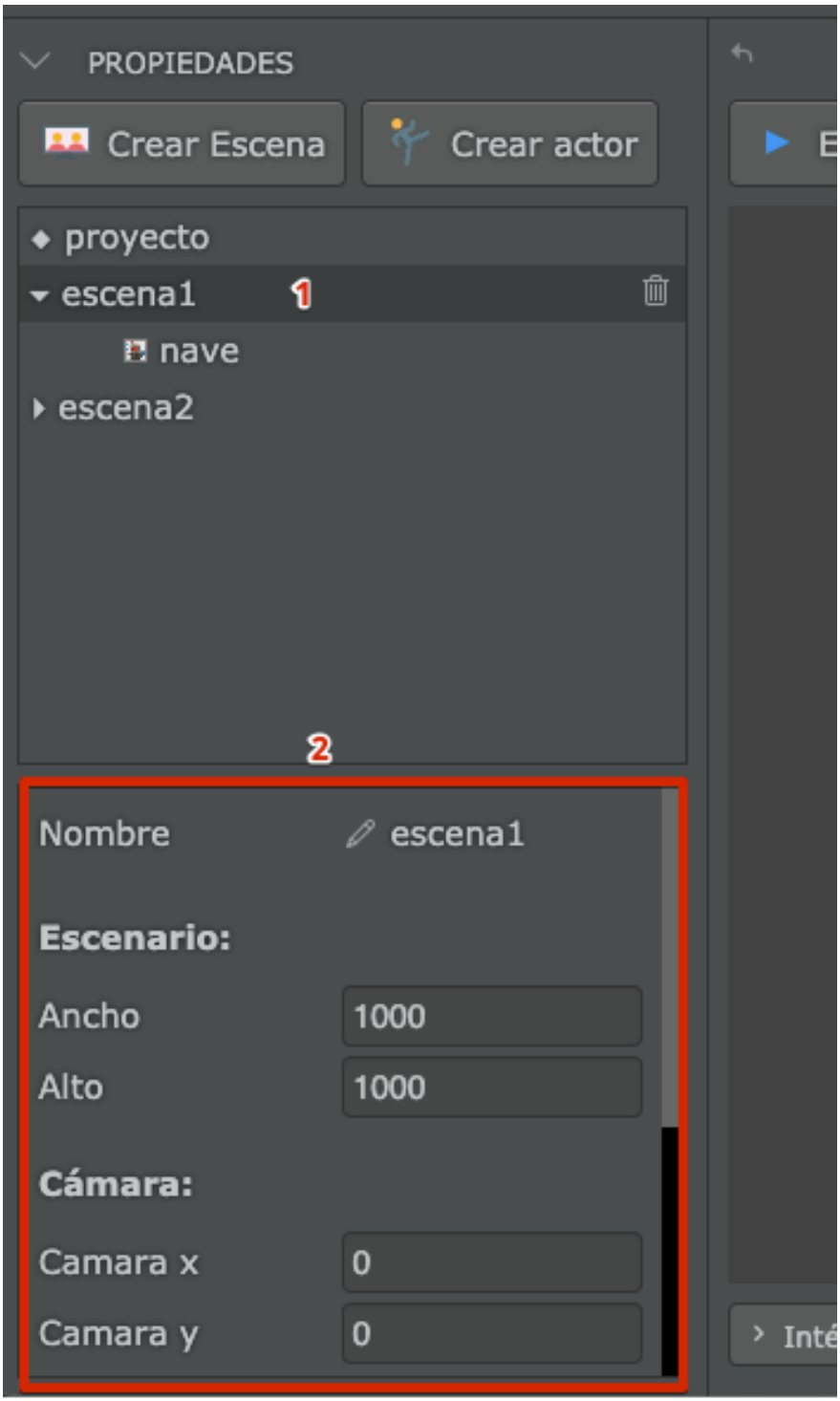


Figure 12: panel-de-escena

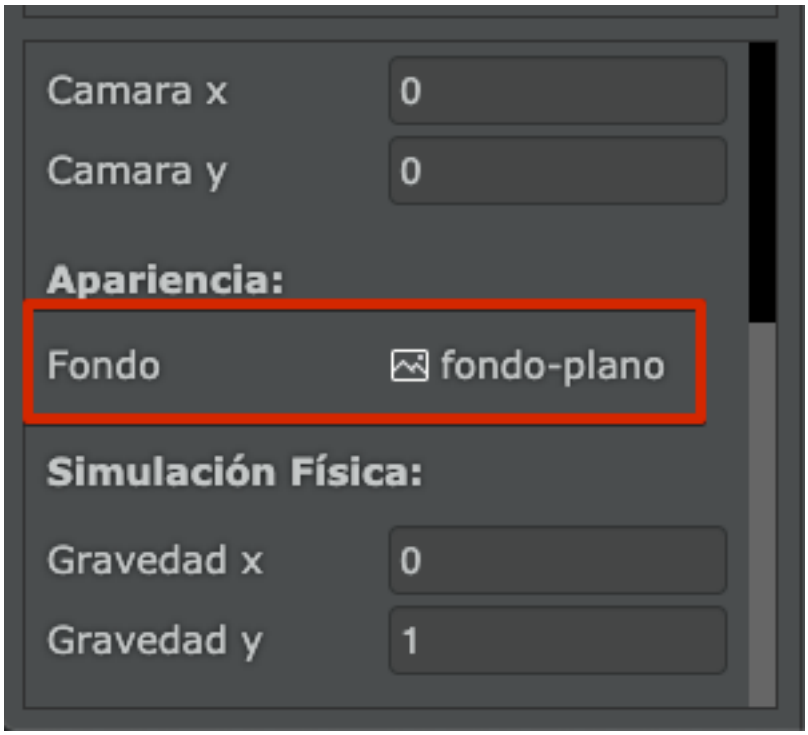


Figure 13: propiedad-fondo

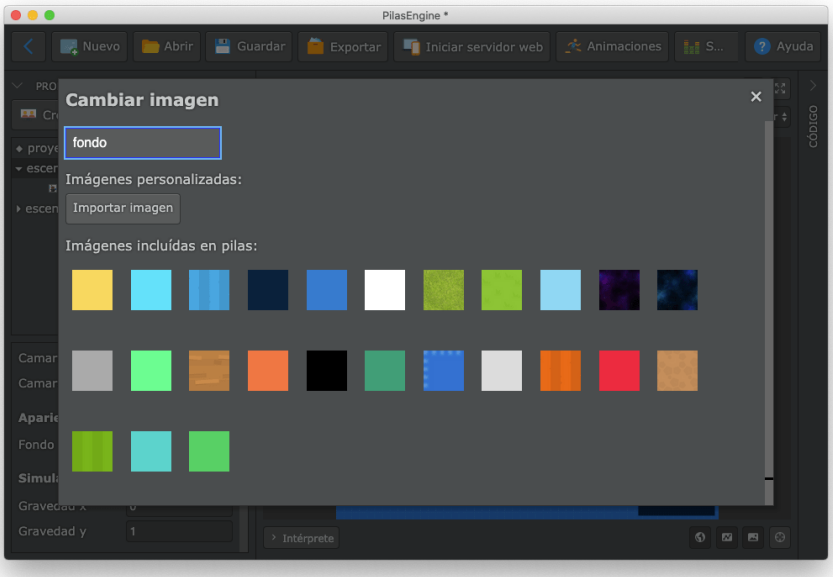


Figure 14: cambiar-fondo-1

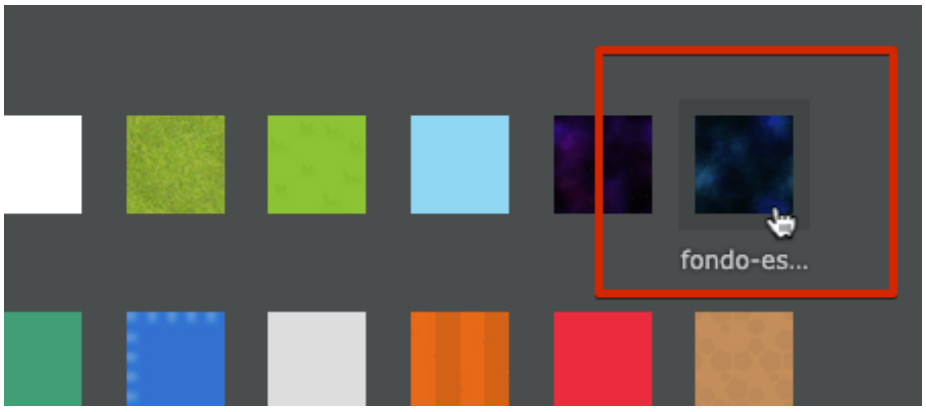


Figure 15: fondo-espacio

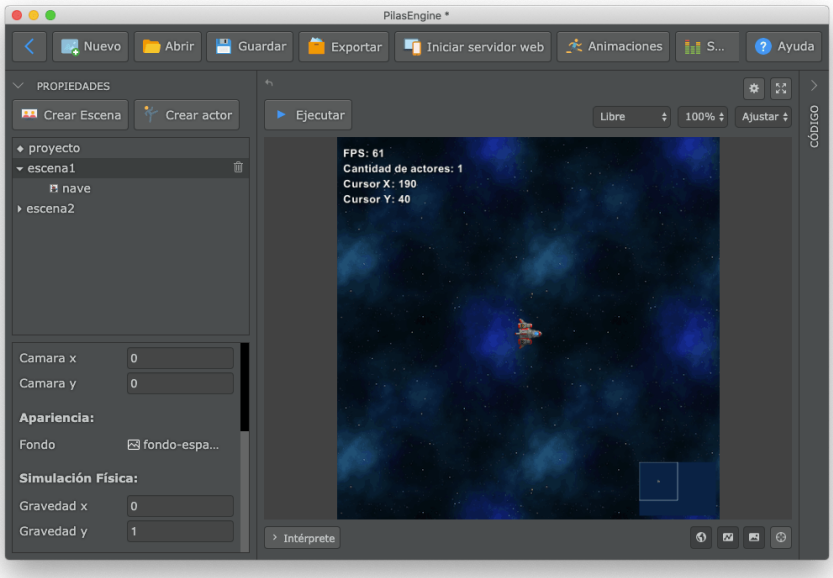


Figure 16: fondo-espacial-en-la-escena

Bueno, no es una gran idea... pero es algo, algo que podemos construir, jugar y divertidos. ¡así que vamos a intentarlo!

Pulsá el botón "Crear actor" nuevamente:

Pero a diferencia de antes, que elegimos la nave, ahora elegí el actor que no tiene una imagen asignada:

Luego, puedes mover este actor usando el mouse, así no tapa a la nave:

Ahora cambiemos la apariencia del actor. Este paso es muy similar al que hicimos antes, tendrías que seleccionar al actor, ir al panel de propiedades pero ahora pulsar la propiedad "imagen":

y luego, selecciona alguna de las opciones. Por ejemplo una manzana:

y te debería quedar así:

Agregando colisiones

Vamos a hacer que la manzana se pueda destruir con los disparos de la nave.

Para eso tenemos que volver a seleccionar el actor con apariencia de manzana y asignarle una figura física así:

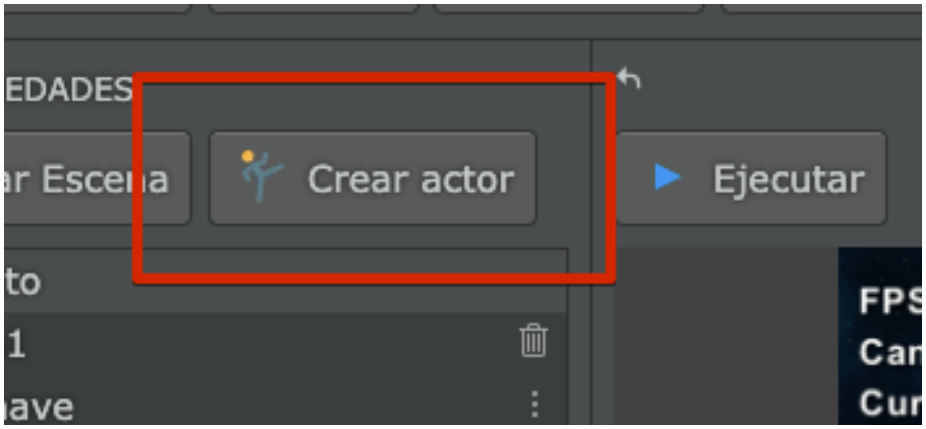


Figure 17: crear-actor

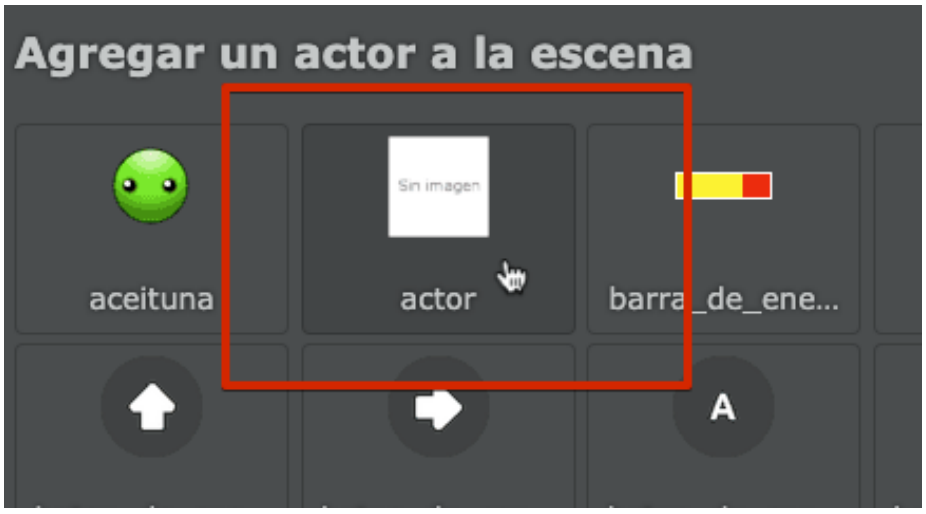


Figure 18: actor-sin-imagen

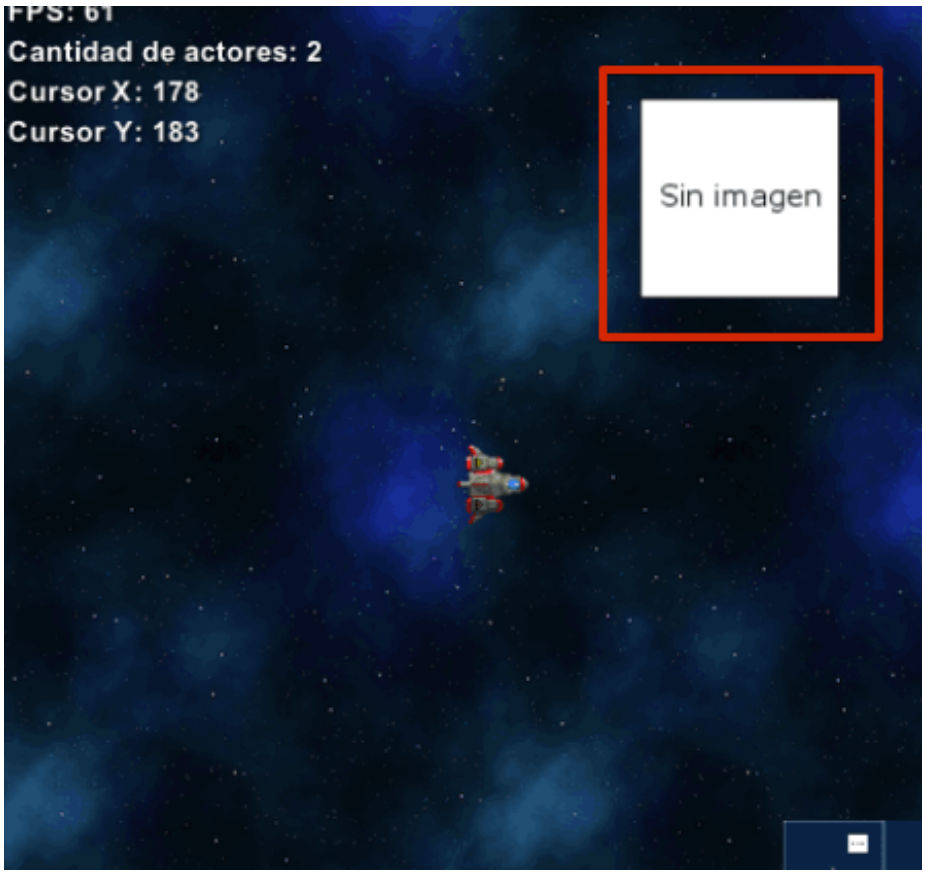


Figure 19: mover-actor

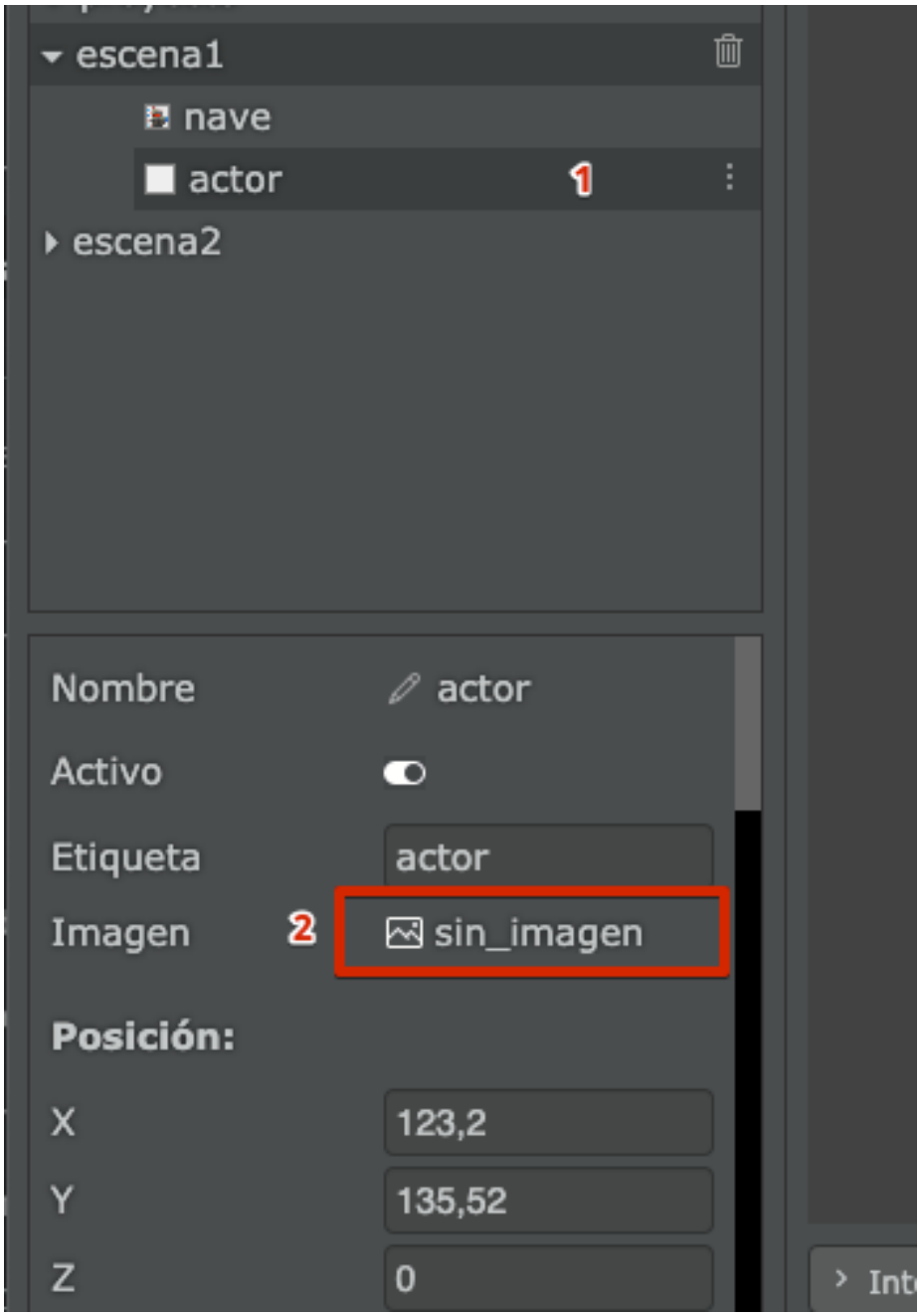


Figure 20: cambiando-imagen

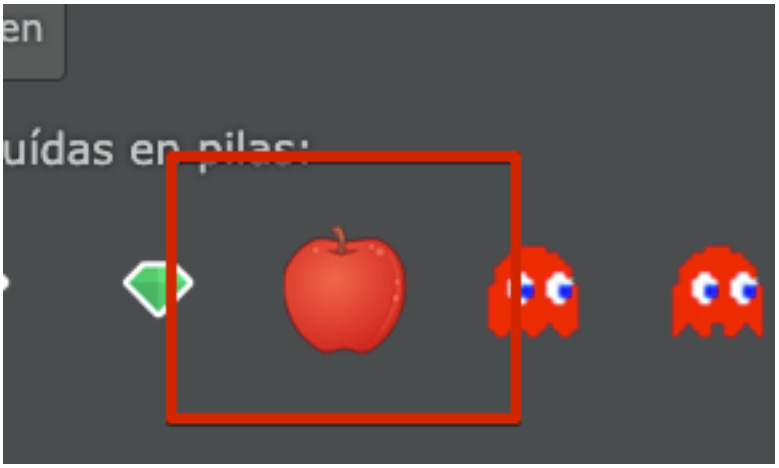


Figure 21: manzana

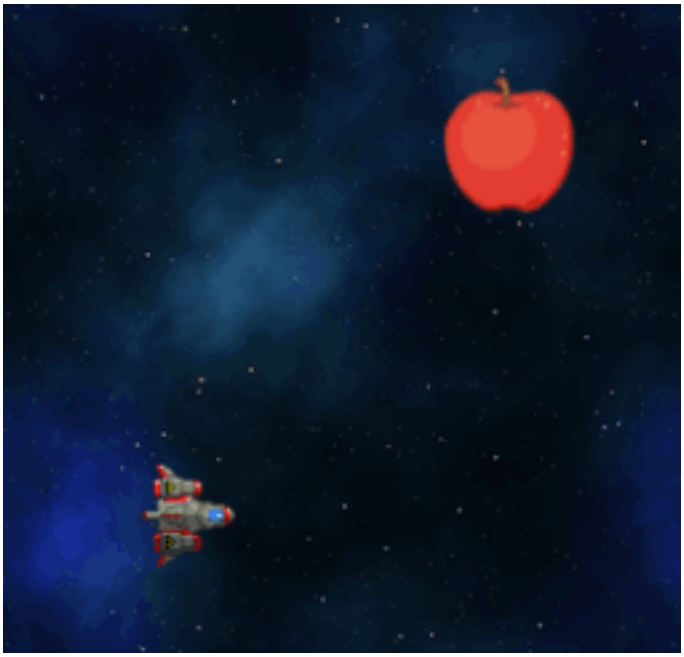


Figure 22: manzana-en-el-espacio

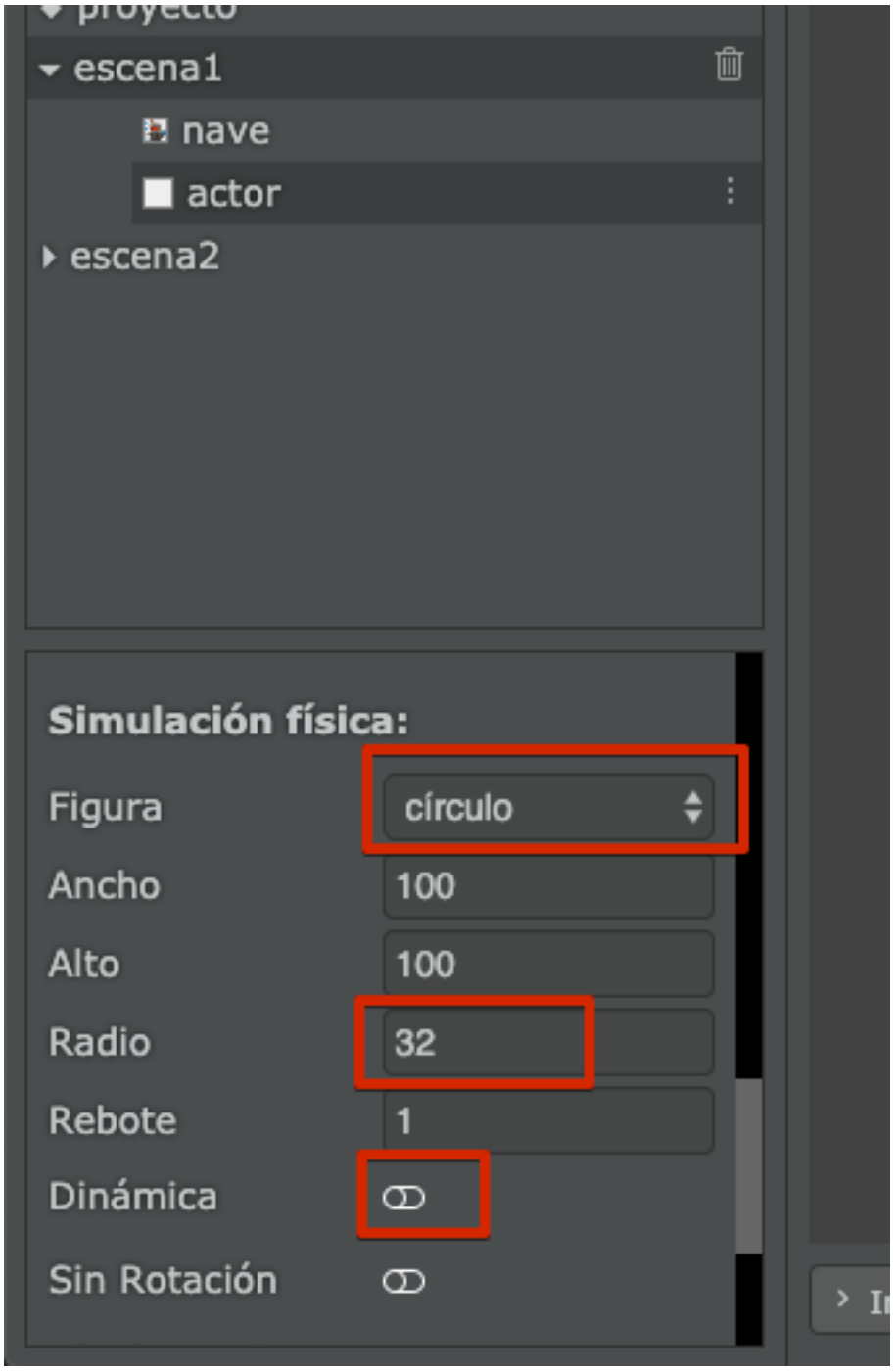


Figure 23: propiedades-de-manzana

Estos parámetros admiten muchas combinaciones, pero te recomiendo que la figura sea "círculo", el radio de "32" y desactives la opción "Dinámica".

Vas a notar que en la pantalla aparecerá un círculo azul al rededor del actor nuevo. Esto significa que el actor tiene un area de contacto para poder interactuar con el resto de los actores:

Reaccionando a las colisiones

Para hacer que el actor manzana reaccione tenemos que escribir un poquito de código.

Selecciona nuevamente al actor y pulsa la pestaña que aparece en la parte derecha de la pantalla que tiene el texto "Código":

Cuando pulses esa pestaña, vas a ver que se abre un panel nuevo; un panel en donde tenemos código y un botón que dice "Recetas":

El código es una de las cosas más importantes que incluye pilas, ya que el código nos permite darle órdenes a la computadora para que haga cosas, como eliminar actores, reaccionar al movimiento del mouse, emitir sonidos y todo lo que se te ocurra.

Sin embargo para empezar vamos a tomar un atajo, vamos a pedirle a pilas que nos ayude a escribir el código para que la manzana se pueda eliminar fácilmente.

Pulsa el botón que dice "Recetas":

y luego selecciona la opción que dice "Cuando colisiona explotar":

Cuando hagas eso, vas a notar que pilas escribió por nosotros una porción de código que hace algunas cosas por nosotros:

Mas adelante en el manual vamos a escribir algo de código sin usar recetas, desde cero y comprendiendo cada función y expresión, por ahora lo dejaremos ahí.

Ahora pulsa "Ejecutar" y corrobora cómo los disparos pueden limpiar el espacio de frutas:

Es solo el principio

Pilas es una herramienta super completa, y este mini-tutorial es solo el comienzo de una gran aventura. Seguí leyendo este manual para conocer muchas más cosas que incluye pilas o explora nuestro sitio web, vas a encontrar tutoriales, videos, juegos y muchas cosas más.

¡Te damos la bienvenida a el mundo de la programación!

Figure 24: image-20200601235938682

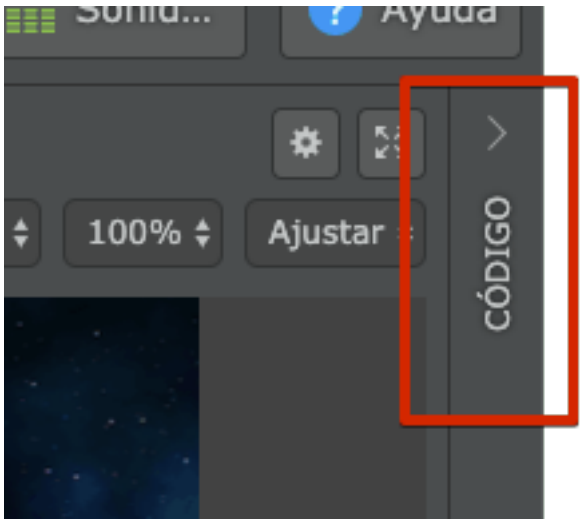


Figure 25: solapa-codigo

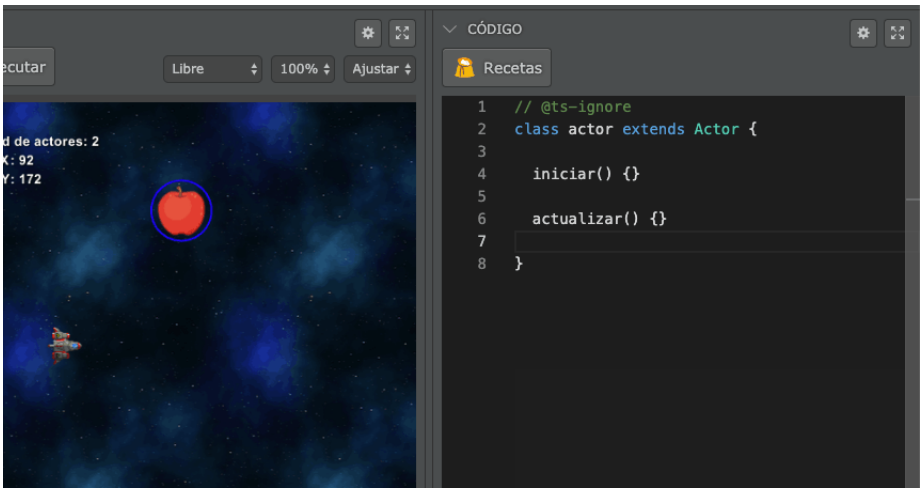


Figure 26: codigo-primera-vista

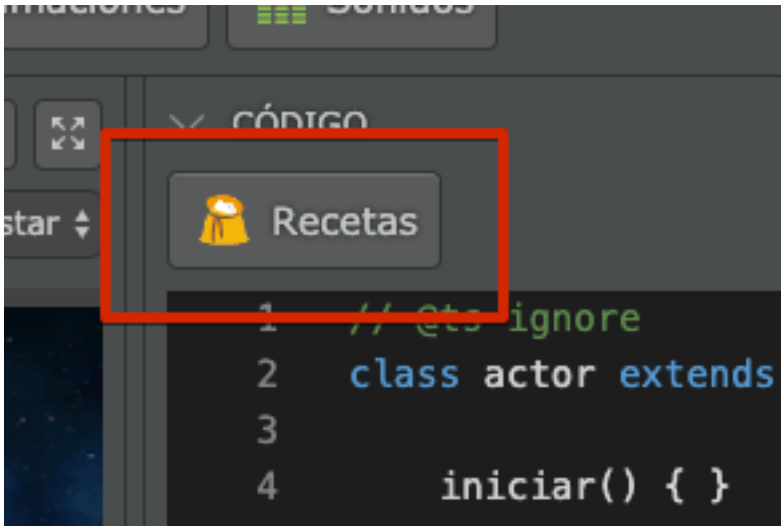


Figure 27: receta-1

Figure 28: receta-2

CÓDIGO

Recetas

```
1   // @ts-ignore
2   class actor extends Actor {
3
4       iniciar() { }
5
6       actualizar() { }
7
8       cuando_colisiona(otro_actor: Actor) {
9           this.eliminar();
10
11          let explosion = this.pilas.actores.explosion();
12          explosion.x = this.x;
13          explosion.y = this.y;
14      }
15
16  }
```

Figure 29: codigo-de-la-receta

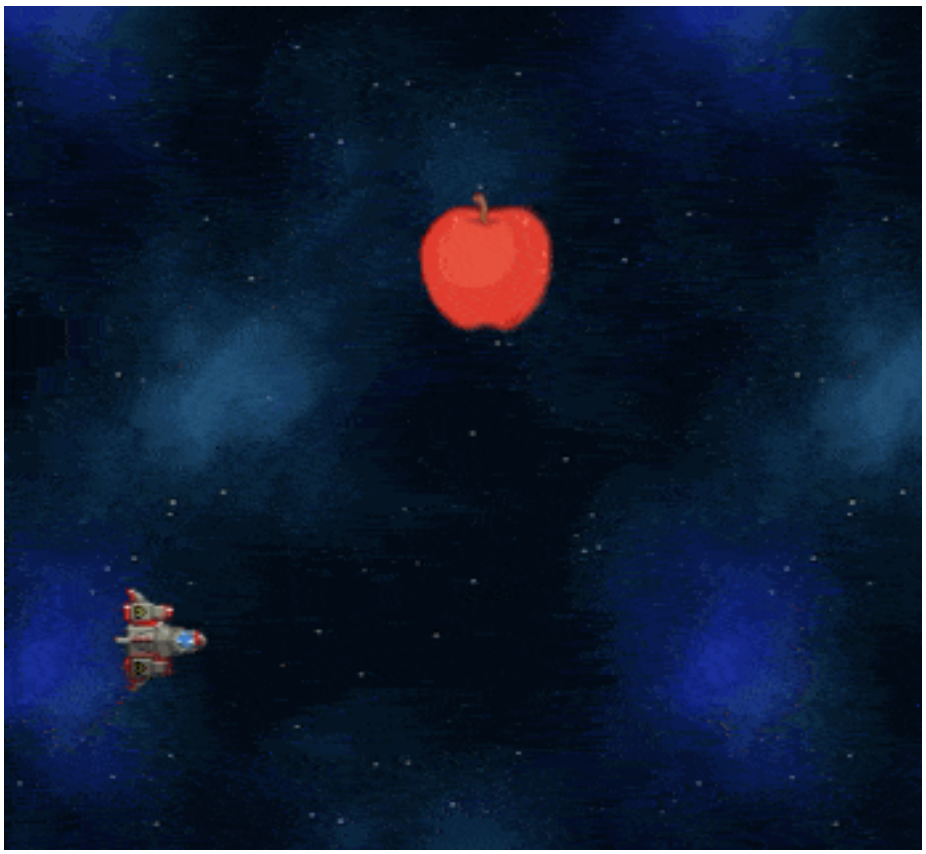
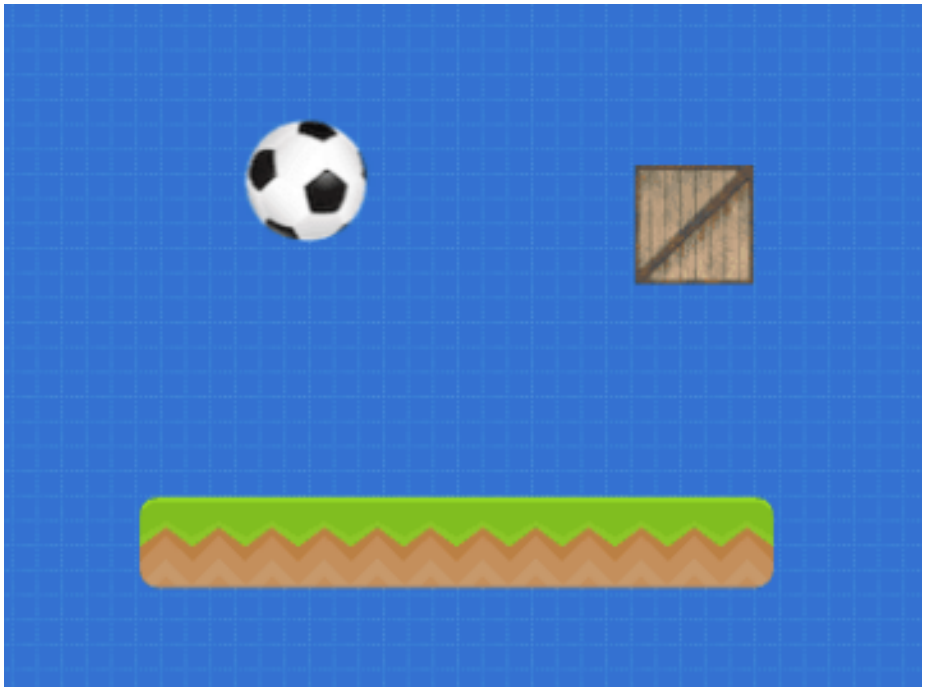


Figure 30: explosion

Actores

En pilas llamamos “actores” a los objetos que aparecen en la pantalla y pueden interactuar con el escenario.

Cuando abrís pilas por primera vez vas a notar que en la pantalla hay tres actores principales, cada uno de estos actores tiene una imagen, posición y comportamiento diferente:



Sin embargo, lo más interesante de estos actores es que los podemos personalizar tanto como queramos, podemos hacer que reaccionen al teclado, que se muevan por si solos, que emitan sonidos o tengan diferentes apariencias.

Cómo crear un actor nuevo

Para crear un actor simplemente hay que pulsar el botón “crear actor” del editor:

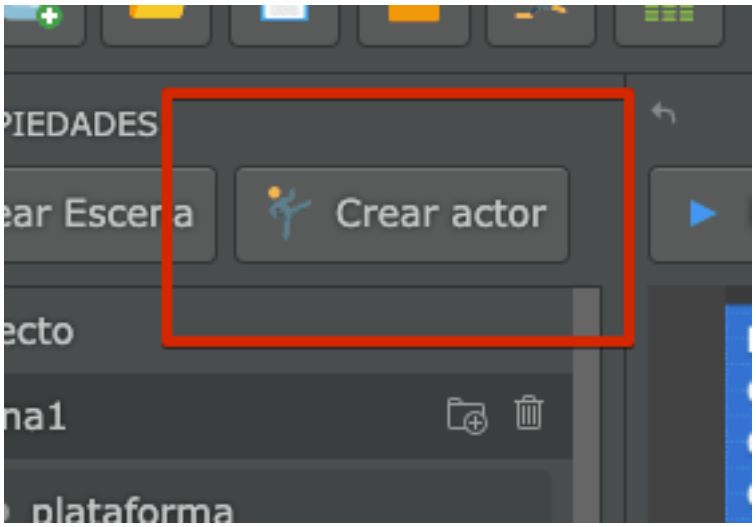


Figure 31: crear-actor-4420826

Y ahí se va a desplegar una ventana con varios actores pre diseñados para incluir en tu juego:

Cada uno de estos actores viene con algún comportamiento para que puedas experimentar e investigar. Ten en cuenta que cada actor que agregues se va a poder cambiar por completo.

Un actor básico

Si bien pilas trae varios actores completos, cada juego es diferente y único, así que el actor mas conveniente para todo juego es un actor mínimo que podemos personalizar desde cero.

Este actor aparece como un rectángulo blanco que dice “sin imagen”:

Ahora si bien es el actor más discreto de todos, lo vamos usar para hacer algunas pruebas y mostrarte conceptos de la herramienta.

Pulsa el botón “crear actor” y luego selecciona a este actor de modo que te quede en la escena de esta forma (los otros actores los puedes eliminar por el momento):

A partir de ese momento, el actor formará parte de la escena, así que vas a ver su nombre en el panel de la escena y una serie de propiedades en el panel izquierdo:

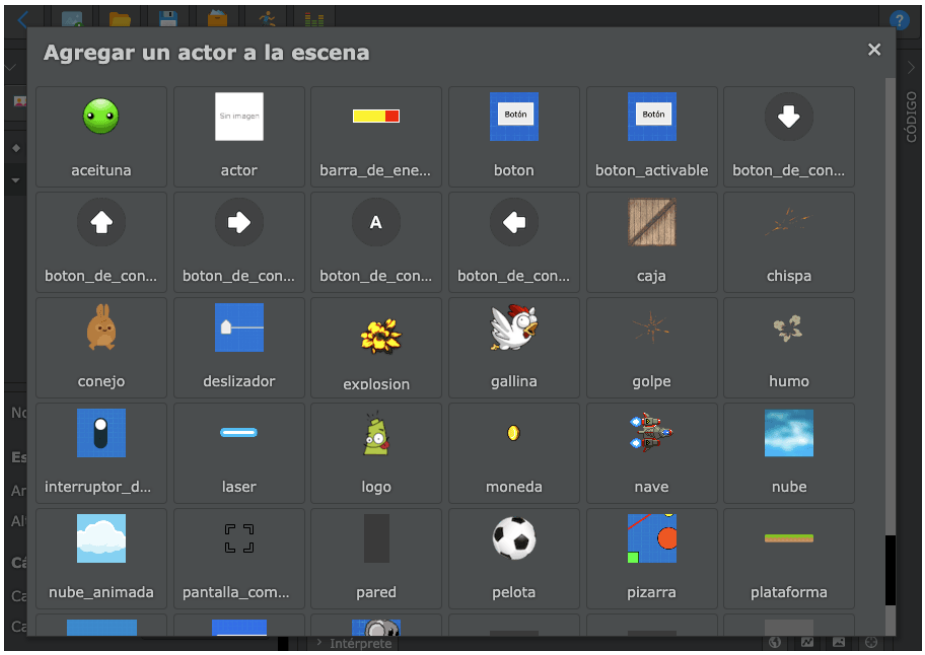


Figure 32: actores-predisenados

Figure 33: actor-sin-imagen

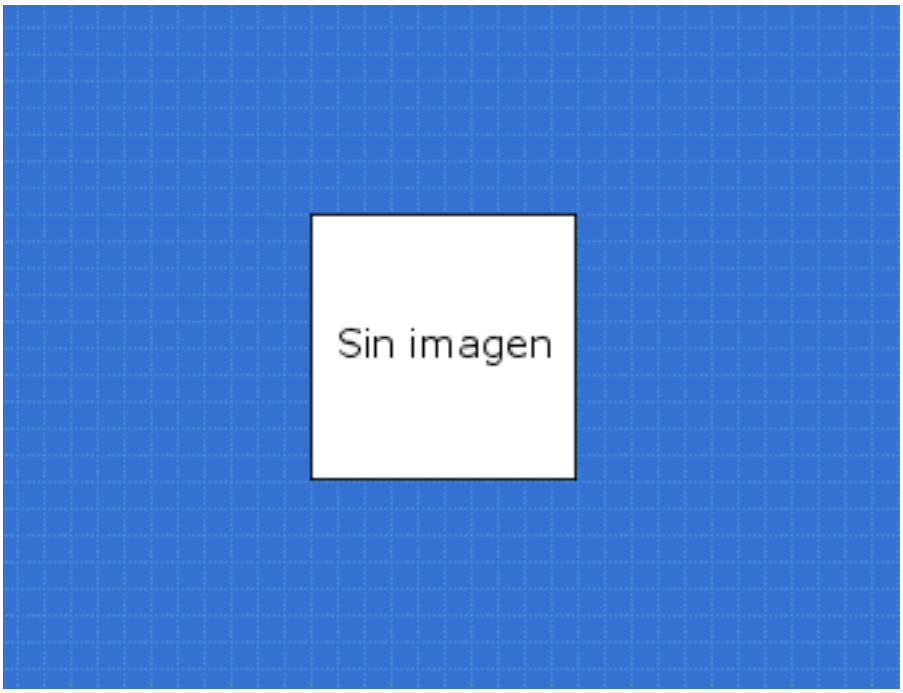


Figure 34: actor-sin-imagen-en-la-escena

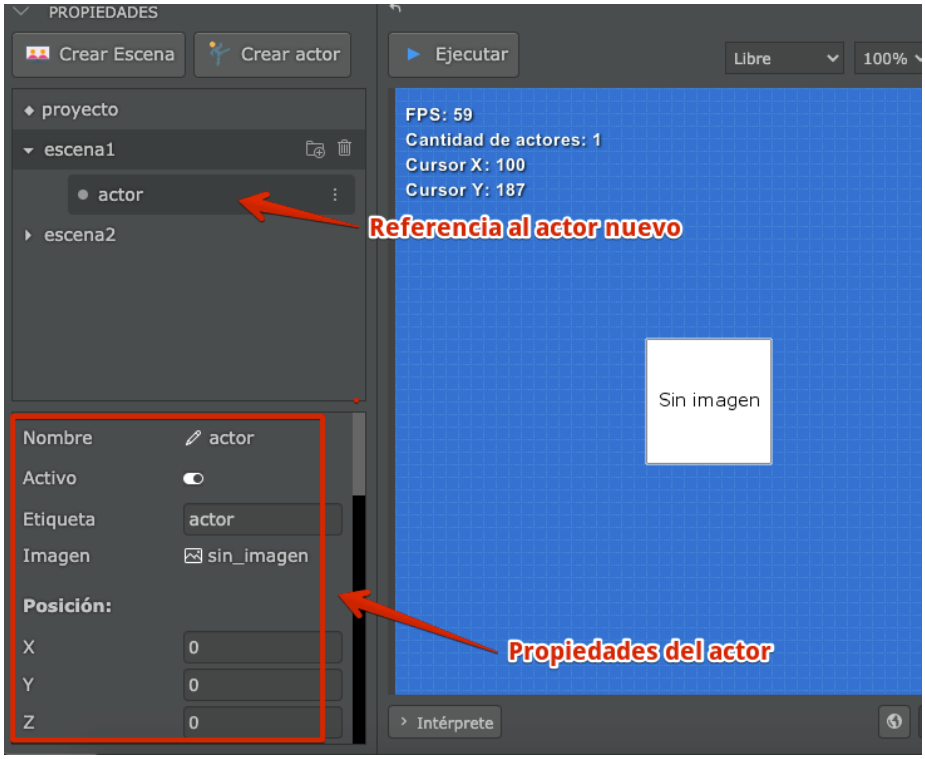


Figure 35: propiedades

Sentite libre de cambiar esas propiedades para experimentar cómo se puede personalizar a un actor.

La propiedad más utilizada es `imagen`, que cuando la pulses te va a dar una serie imágenes alternativas para seleccionar:

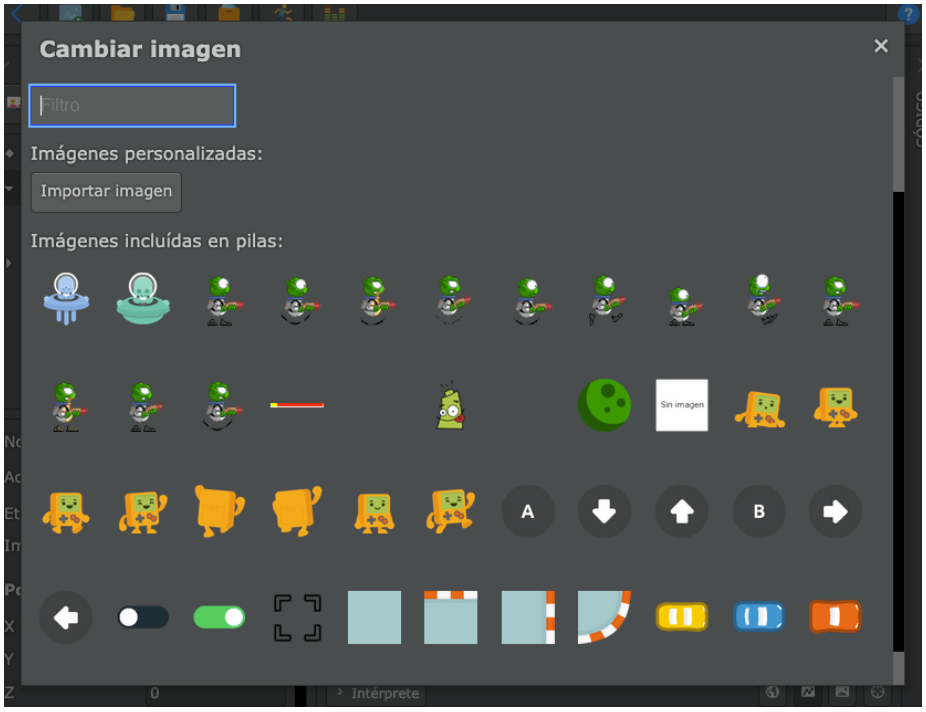


Figure 36: imagenes

Por ejemplo, si elijo la imagen del mono sonriendo mi actor va a tener una apariencia completamente diferente:

Todas las propiedades se pueden ajustar usando el teclado o incluso realizando movimientos con el mouse:

Recortar imágenes de actores

Si estás buscando crear efectos o juegos de piezas, ten en cuenta que existe un método llamado `recortar` que te permite seleccionar una porción de la imagen del actor y mostrarla en pantalla.

Por ejemplo, si tenemos un actor como este:

Podemos crear un recorte a la mitad mediante este código:

```
this.recortar(0, 0, 125, 146);
```

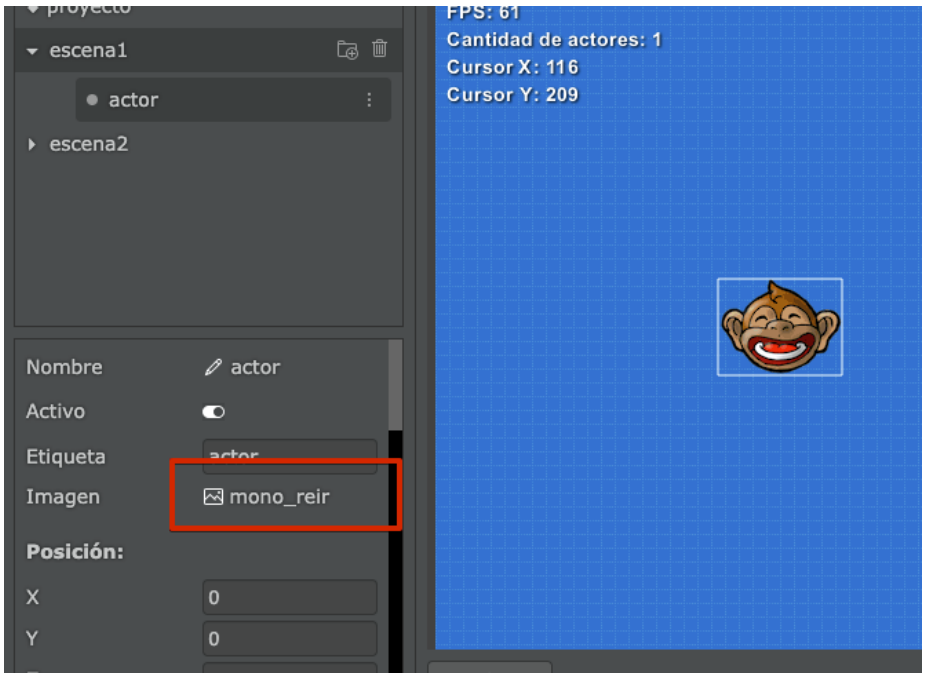


Figure 37: actor-mono

Figure 38: animacion

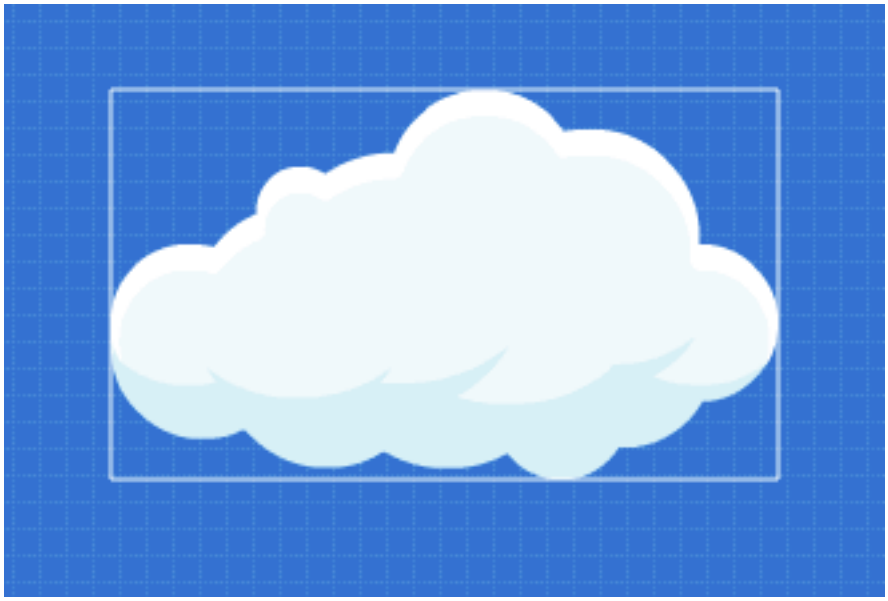


Figure 39: nube

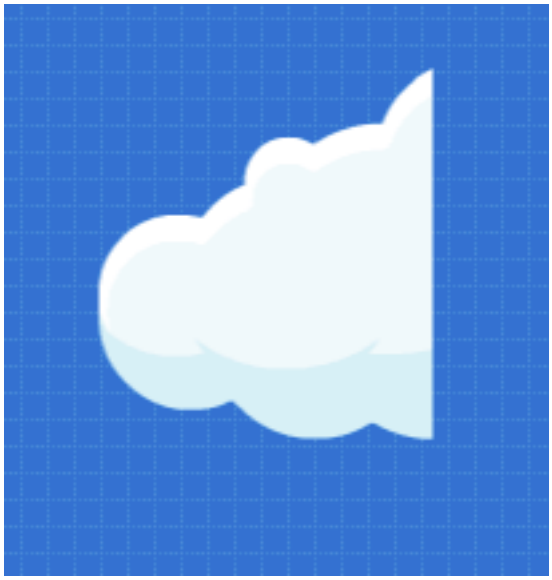


Figure 40: nube-recortada

Donde los valores `0, 0, 125, 146` son las coordenadas `izquierda`, `arriba`, `ancho` y `alto` :

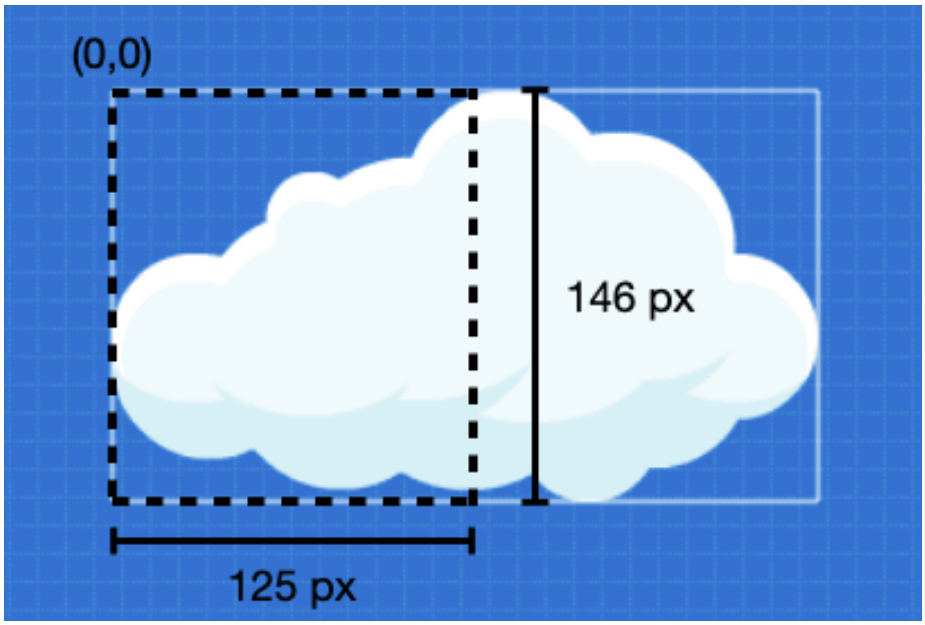


Figure 41: recorte-

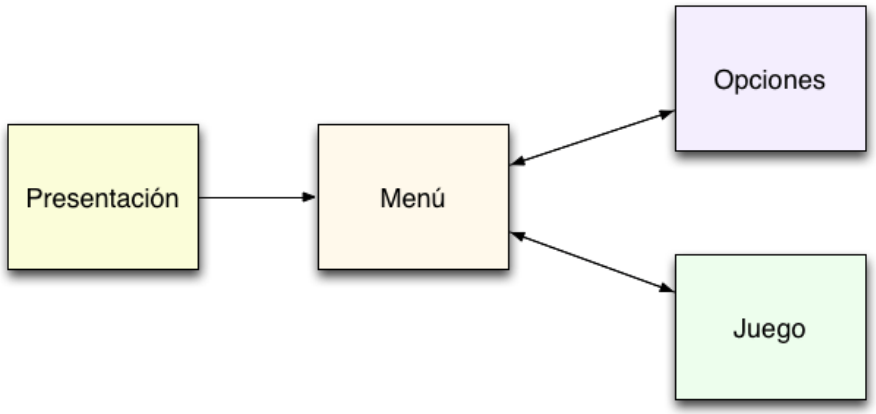
y cuando quieras volver a la imagen original, simplemente puede llamar a la función:

```
this.eliminar_recortado();
```

Escenas

Las escenas te permiten dividir el juego en partes reconocibles y que interactúan de manera diferente con el usuario.

Un juego típico tendrá al menos una escena como el menú principal, una presentación y una pantalla de juego.



Cambiar o reiniciar escenas

Para cambiar la escena actual se puede llamar a la función `cambiar_escena` indicando el nombre de la escena que se quiere cambiar. Por ejemplo:

```
pilas.cambiar_escena("escena2");
```

También se puede reiniciar la escena actual llamando a esta función:

```
pilas.reiniciar_escena();
```

Gravedad y simulación física

Cada escena tiene asociado su propio entorno de simulación de física. Así que si estás haciendo un juego que involucra física es probable que quieras ajustar algunos parámetros como `gravedad_x` y `gravedad_y`. Estos parámetros se encuentran en el inspector de la escena y también se pueden cambiar desde el código así:

```
pilas.fisica.gravedad_x = 1;
pilas.fisica.gravedad_y = 1;
```

o bien:

```
pilas.escena.gravedad_x = 0;
pilas.escena.gravedad_y = -2;
```

El intérprete

Programar se parece un poco a mantener una conversación con la computadora, le enviamos mensajes y esperamos una devolución de su parte. Por ese motivo, para poder hacer un juego que pueda ejecutar una computadora podemos usar un "lenguaje de programación" que entendamos tanto personas como computadoras.

Aquí es donde entra en juego una pieza de Pilas llamada "Intérprete". El intérprete es como un *chat* en donde podemos comunicarnos directamente con la computadora y esperar respuestas.

Para mostrar brevemente lo que puede hacer el intérprete pulsá el botón "Ejecutar", y luego hace click en ese botón que aparece debajo del area de juego a la izquierda:

Cuando pulses ese botón debería abrirse un area rectangular de esta forma:

Lo primero que aparece en esta pantalla son dos mensajes que nos envía pilas avisando que ingresamos en el modo ejecución, es decir, el juego está en funcionamiento. Y por el otro lado nos dice que hay dos variables para usar "pilas" y "actores".

Pero antes de hablar de variables, vamos a probar exactamente qué nos permite hacer el intérprete.

En el intérprete podemos escribir código y la computadora nos va a responder.

Hagamos una prueba, escribí `2+2` en el intérprete así probamos la interacción con la computadora.

En area en donde se puede escribir es la que está marcada con ese cursor `>`:

Luego pulsa la tecla "Enter" y la computadora te va a responder:

El intérprete va a responder a cada una de las cuentas con el resultado, como si fuera una calculadora aritmética.

Sin embargo, hace más que eso, si le pedimos incorrectamente algo nos dará un mensaje de error:

```
2 + numero;
```

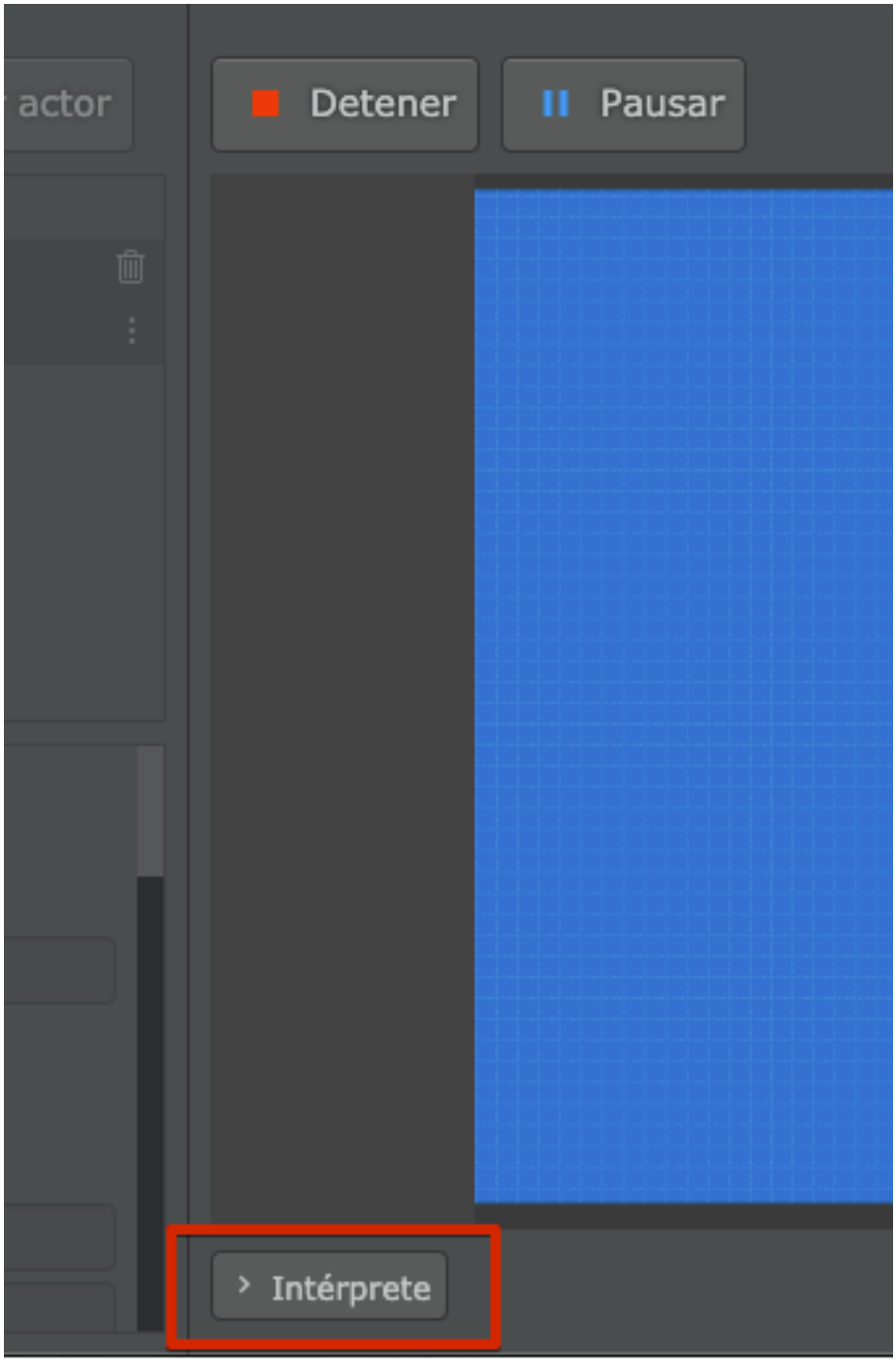


Figure 42: boton-interprete

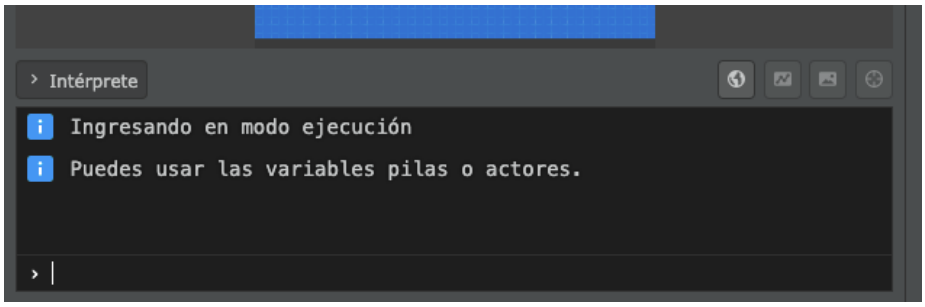


Figure 43: interprete-inicial

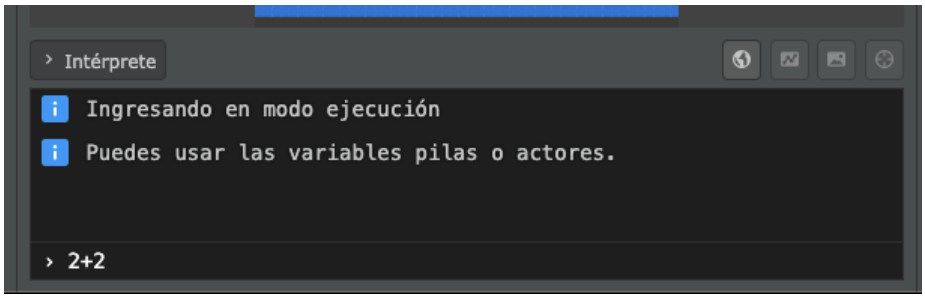


Figure 44: suma

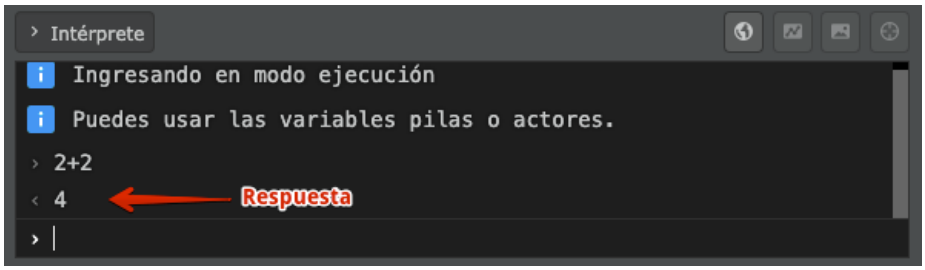


Figure 45: respuesta

Figure 46: error

Y esto está bien, si bien el texto rojo parece algo grave, no lo es. La computadora nos explica que no sabe de que se trata "numero" y por lo tanto no sabría decirnos el resultado de esa cuenta.

Si le indicamos que el "numero" es `10` sí va a poder decirnos el resultado:

```
numero = 10;
2 + numero;
```

Figure 47: resultado-con-variable

¡Claro!, el resultado es `12` ya que creamos una variable llamada "numero" con el valor de 10.

Accediendo a las variables pilas y actores

Ahora bien, el intérprete no solo puede realizar cuentas aritméticas, también nos da la posibilidad de realizar pruebas, crear o modificar los actores en la pantalla y varias cosas mas.

Al principio de esta sección mencioné que pilas nos avisaba que había dos variables listas para utilizar "pilas" y "actores":

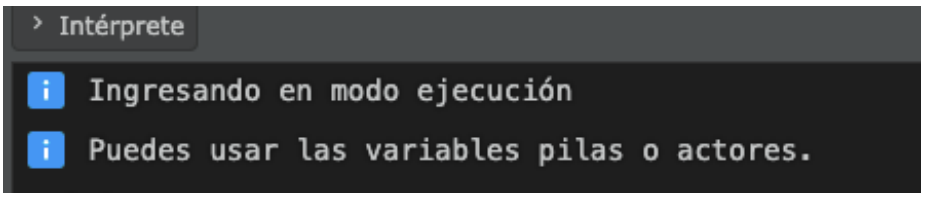


Figure 48: variables

Estas dos palabras se denominan variables, porque representan un nombre que almacena un valor asignado con anterioridad. En nuestro caso, "numero" también es una variable, porque básicamente es el nombre que le dimos al valor `10`.

Ahora bien, ¿qué podemos hacer con estas variables?.

La variable `pilas` nos permite acceder a todas las funcionalidades del motor. Si escribimos algo como:

```
pilas.actores.nave()
```

Vas a notar que aparecerá una nave en la pantalla como respuesta al código que escribimos:

También vas a notar que en el intérprete aparece el mensaje `<nave en (0, 0)>`, esto es así para indicarnos que obedeció a nuestras orden. Pilas creó el actor nave y lo dejó en la posición `(0, 0)` de la pantalla.

Si haces click en la pantalla, vas a notar que la nave está "viva", se puede mover y disparar usando el teclado.

Ahora bien, aquí es donde entra en juego la otra variable que te comenté llamada "actores". Esta variable va a permitirnos referirnos a cualquier actor dentro de la escena.

Escribí estas lineas de código en el intérprete para ver el resultado:

```
nave.escala = 2
nave.rotacion = 90
nave.decir("hola")
```

Lo que deberías ver es que la nave cambia de tamaño, apunta hacia arriba y además muestra un mensaje:

Esto es bastante útil, porque el código que escribimos en el intérprete también se puede trasladar al editor. Si tenemos dudas sobre alguna función o forma de hacer algo en pilas podemos probar en el intérprete, y si todo sale bien, llevarlo al editor y que forma parte de nuestro juego.

Algunos consejos

El intérprete es algo bastante original en los motores de video juegos, pero se utiliza muchísimo en ambientes de desarrollo web y sistemas complejos.

Y como varias de las personas que usamos pilas venimos de ese mundo, el intérprete está diseñado en base a la experiencia de usar intérpretes durante muchos años... Un ejemplo de esto es el sistema de auto completado:

Siempre que escribas una variable conocida el intérprete va a intentar ayudarte a escribir código de esta forma:

Figure 49: image-20200604161609718

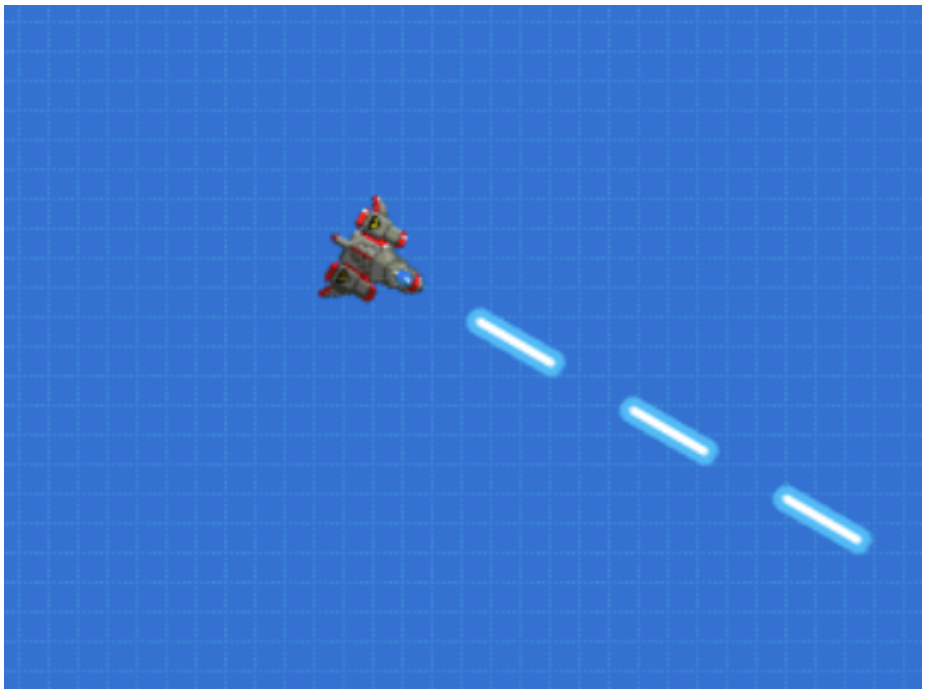


Figure 50: image-20200604161800642

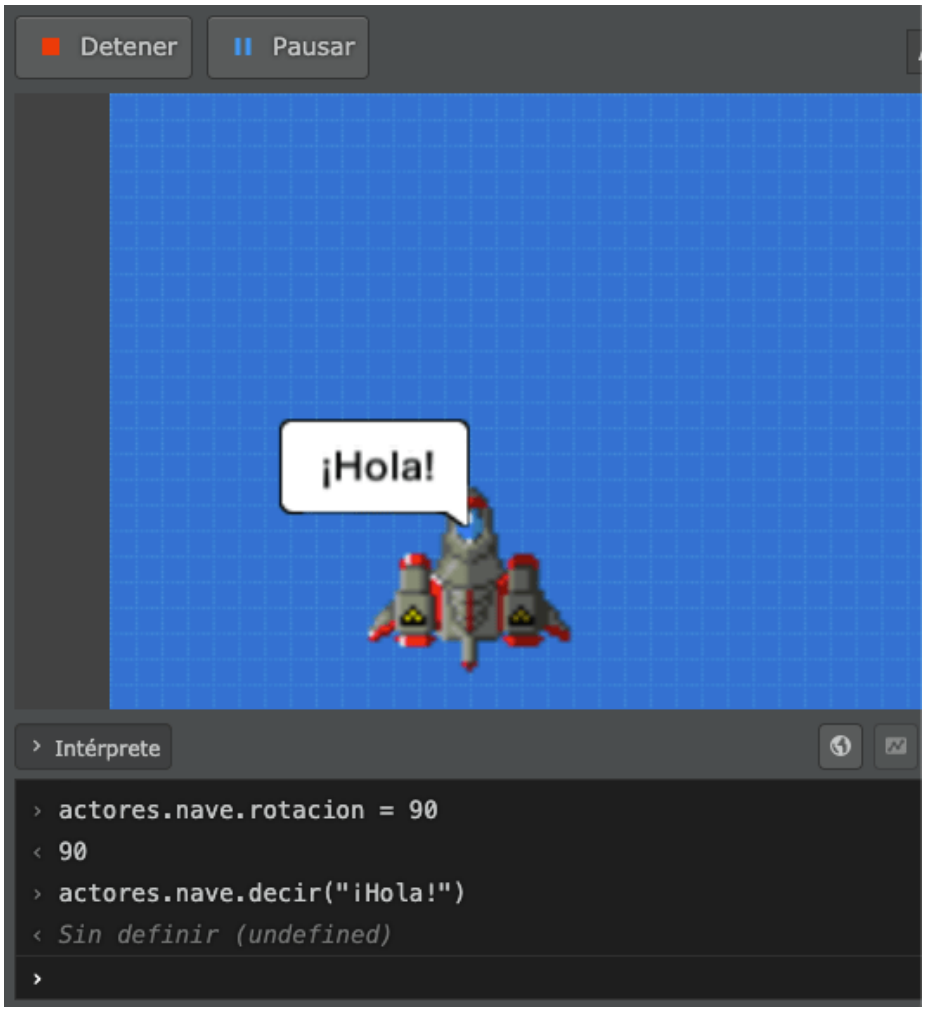


Figure 51: nave-dice-hola

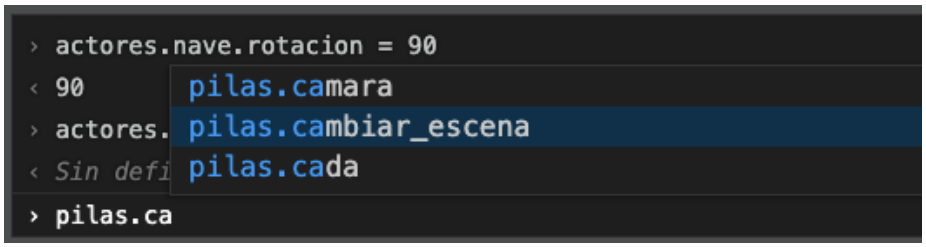


Figure 52: autocompletado

Otra necesidad muy común es querer volver a repetir una instrucción anterior, para eso puedes pulsar las flechitas del teclado hacia arriba y abajo para navegar:

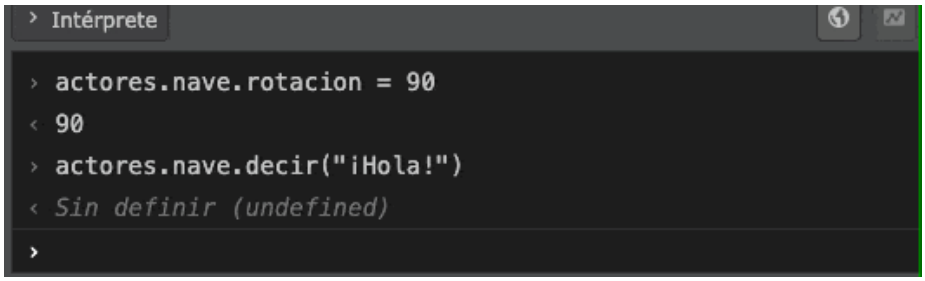
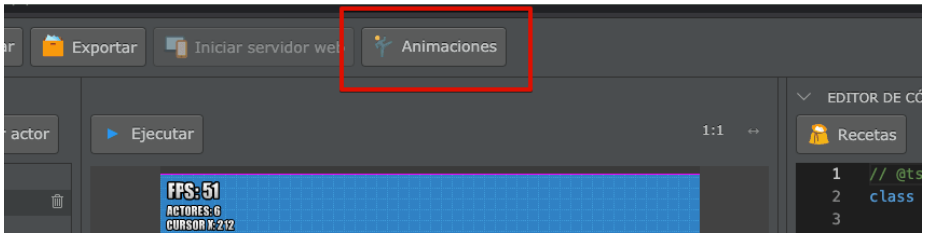


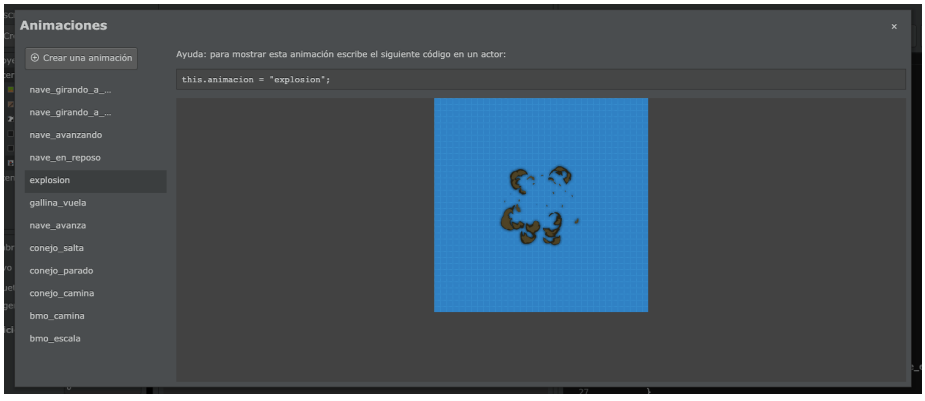
Figure 53: autocompletado-historico

Animaciones

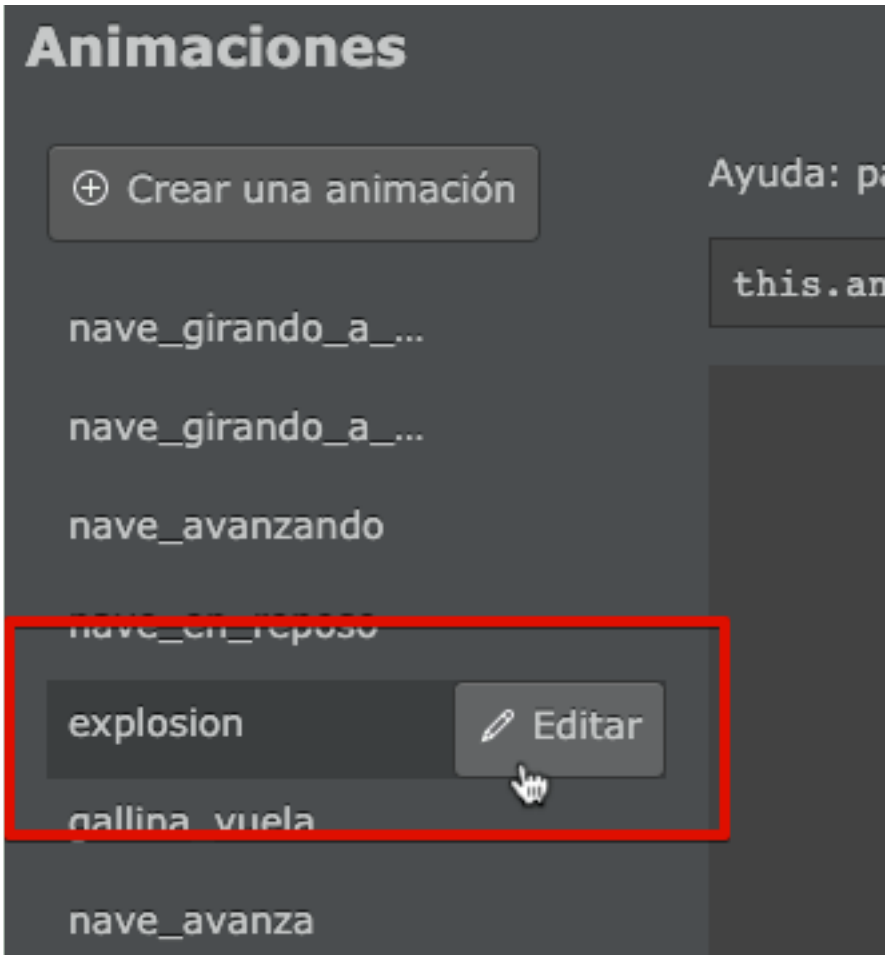
Para crear animaciones se tiene que utilizar el editor que aparece en la parte superior de la pantalla:



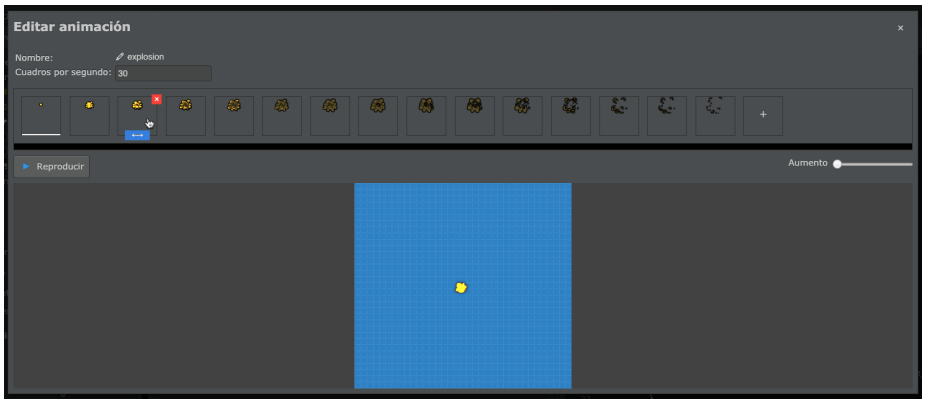
Este botón abrirá una ventana en donde se pueden previsualizar todas la animaciones del proyecto:



Estas animaciones también se pueden editar fácilmente, solo tienes que pasar el mouse sobre el nombre de la animación y pulsar el botón “editar”:

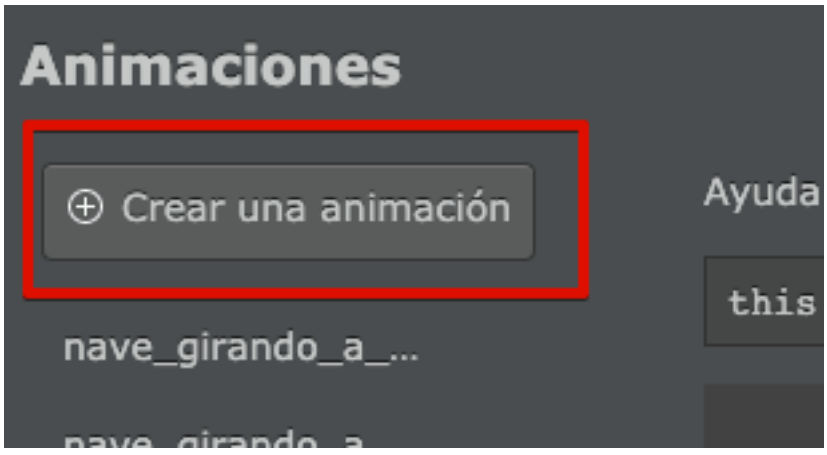


Las animaciones son simplemente una lista de imágenes que pilas mostrará una detrás de la otra, a determinada velocidad. Vas ver toda esta información en el editor de animaciones:

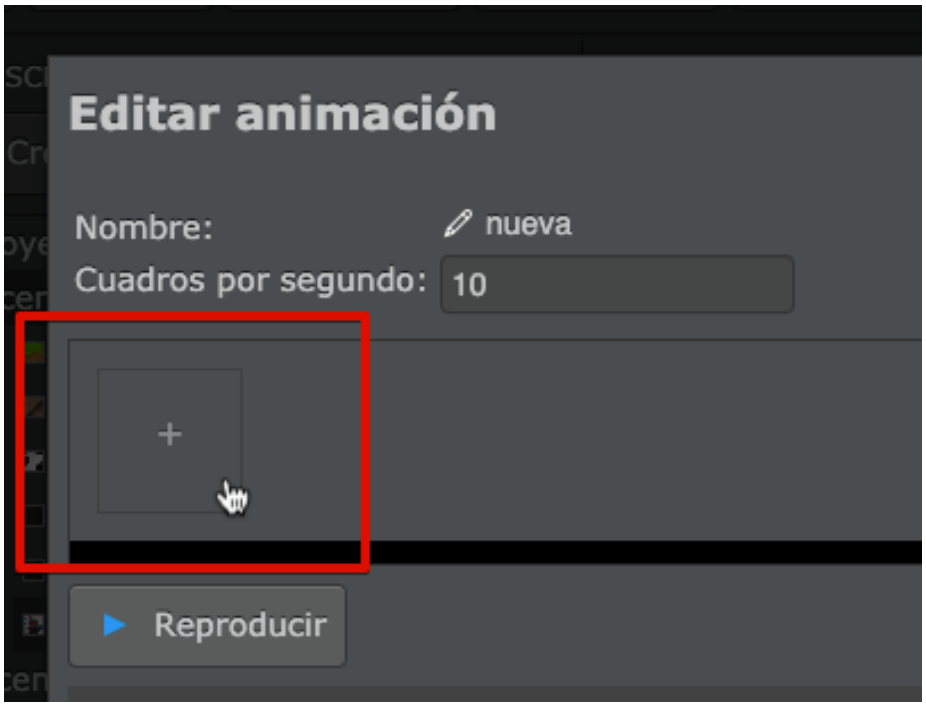


Crear animaciones desde el editor

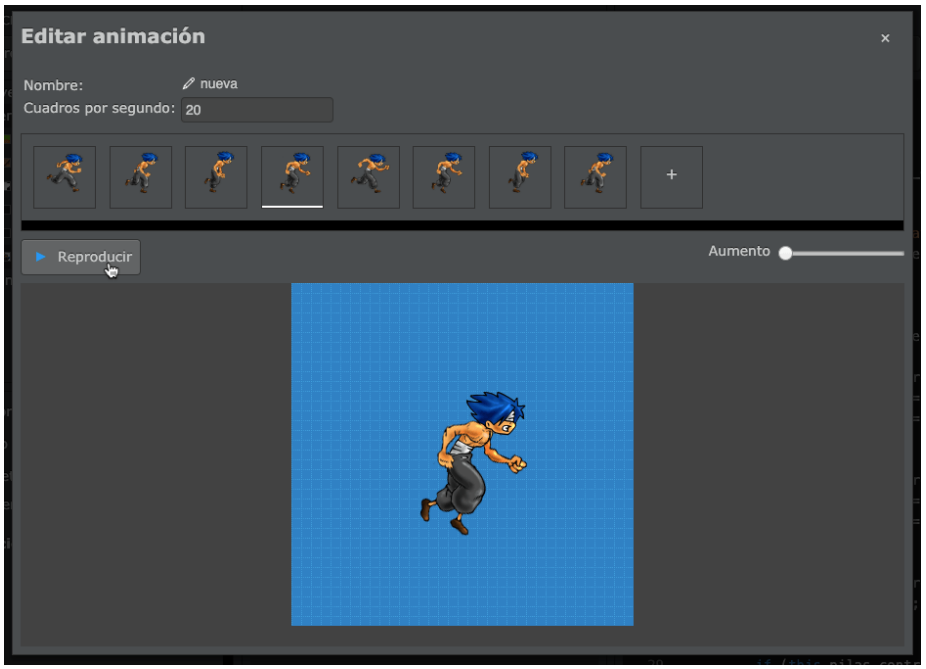
Para crear animaciones hay que pulsar el botón “Crear una animación”:



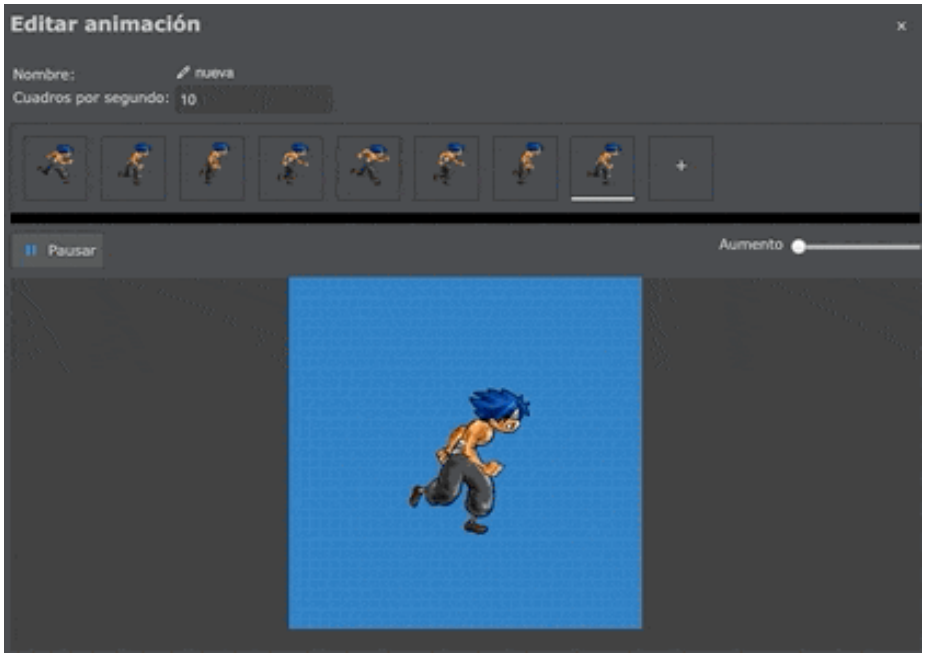
y luego cargar cada uno de los cuadros de animación pulsando el botón “+” que aparece en la parte superior de la ventana:



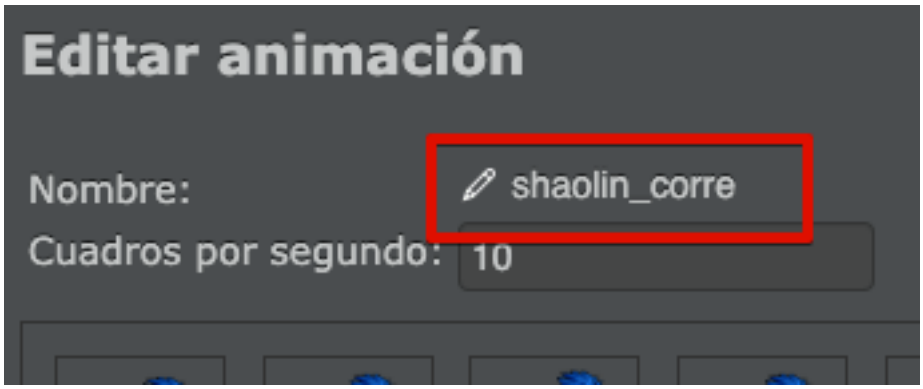
Por ejemplo, aquí incluí algunos cuadros de animación de un personaje corriendo:



Puedes utilizar el botón “Reproducir” y la propiedad “Cuadros por segundo” para ajustar la velocidad de la animación y dejarla como quieras:



Por último es muy importante que le asignes un nombre a la animación, por ejemplo “shaolin_corre”:

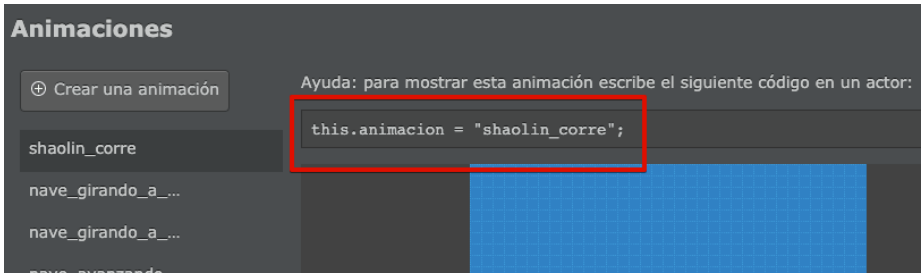


Esto es muy importante porque tu juego puede tener un montón de animaciones, y el nombre que le asignes será la única forma de identificar cada una de las animaciones.

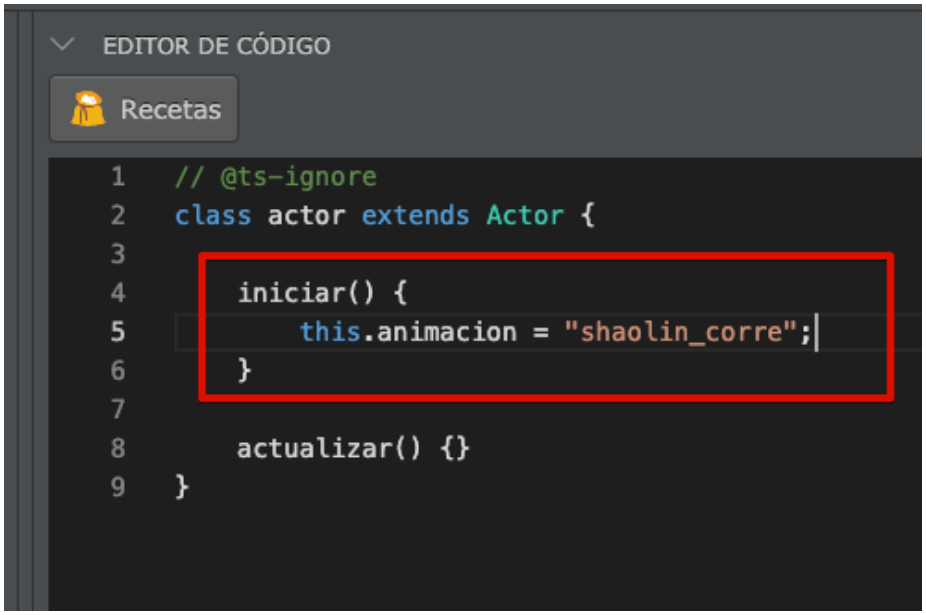
Ahora sí, puedes cerrar la ventana del editor y continuar con la siguiente sección.

Cómo usar las animaciones

Una vez que tienes creada la animación, lo único que hace falta es copiar el código que aparece como ayuda en el visor de animaciones dentro del código:



Ese código, sirve para indicarle al actor qué animación tiene que reproducir. Por ejemplo, si quieres que el actor muestre esta animación al comenzar deberías colocarlo dentro de la función “iniciar” así:



Detectar la finalización de las animaciones

Las animaciones siempre se muestran de forma cíclica, es decir, cuanto terminan vuelven a empezar desde cero. Si tu animación es tradicional, como un personaje caminando, no tienes que hacer nada adicional para que continue la animación automáticamente.

Si quieres detectar el momento exacto cuando la animación llega a su final,

deberías crear un método llamado `cuando_finaliza_animacion` y colocar ahí algún código para reaccionar ante la finalización de la animación. Por ejemplo, el actor "explosión" se elimina de la pantalla automáticamente cuando finaliza su animación:

```
class explosion extends Actor {
  // Otros métodos

  cuando_finaliza_animacion(nombre: string) {
    this.eliminar();
  }
}
```

Controlando animaciones desde el código

Para mostrar una animación en un actor tenemos que asignar un valor al atributo animación así:

```
iniciar() { // iniciar, u otro método.
  this.animacion = "personaje_caminando";
}
```

Esto hará que la animación se muestre de forma contínua. Si queremos hacer un uso manual, primero tenemos que pausar la animación así:

```
this.pausar_animacion();
```

Una vez ahí, podemos hacer que la animación avance manualmente llamando al método "actualizar_animacion":

```
this.actualizar_animacion();
```

Este método también admite un parámetro para controlar la velocidad, estos son algunos ejemplos de invocación:

```
// para reproducir la animación al doble de velocidad:
this.actualizar_animacion(2);

// para reproducir la animación a la mitad de velocidad:
this.actualizar_animacion(1/2);

// para reproducir la animación en reversa
this.actualizar_animacion(-1);

// para reproducir la animación en reversa muy lentamente, 5
// veces más lento de lo normal
this.actualizar_animacion(-1/5);
```

Una vez finalizado el manejo de las animaciones de forma manual, se puede llamar al siguiente método para continuar con la animación desde donde quedó:

```
this.continuar_animacion();
```

Animación de propiedades

Todos los actores tienen una serie de propiedades como `x`, `y`, `rotacion` y `transparencia`. En total hay mas de 10 propiedades listas para modificar, tanto desde el editor como desde el código.

Ahora bien, si cambiamos progresivamente estas propiedades podemos lograr animaciones interesantes. Por ejemplo podríamos cambiar progresivamente la rotación de un actor para que parezca estar en movimiento así:

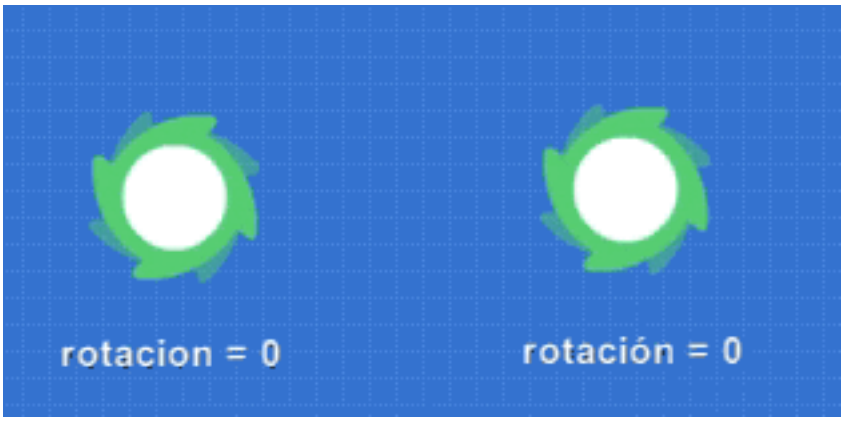


Figure 54: rotacion

Y lo mismo con las otras propiedades, si queremos que un actor se mueva hacia la derecha de la pantalla podemos cambiar muchas veces su propiedad `x` para que parezca "moverse" hacia un costado de la pantalla.

Sin embargo, hacer animaciones cambiando propiedades una a una se puede volver tedioso si realmente queremos hacer animaciones completas, así que pilas tienen una forma de simplificar esto.

Usando la función “animar”

Todos los actores tienen una función llamada `animar` parar facilitar la creación de animaciones, o movimientos, mediante el cambio de propiedades de un actor.

Si queremos lograr que un actor gire infinitas veces como mostramos en la animación de más arriba podemos escribir lo siguiente:

Figure 55: image-20200617173113502

La función `this.animar` iniciará una animación sobre las propiedades del actor. Primero espera dos argumentos:

- El tipo de animación que se realizará, esta puede ser `Tipo.lineal`, `Tipo.desborde`, `Tipo.suave`, `Tipo.elastico`, `Tipo.rebote`. Vamos a ver esto en la siguiente sección.
- La cantidad de veces que se ejecutará la animación: aquí se puede poner un número como 5 para que la animación de rotación se haga `5` veces, o como valor especial se puede colocar `-1` para que la animación se haga por siembre.

Lo siguiente que se coloca después de la llamada a `this.animar()` es una secuencia de animaciones a realizar.

Por ejemplo, en este ejemplo se le pide al actor que se mueva a la derecha, gire 180 grados y luego regrese al punto `(0, 0)` una vez:

```
this.animar(Tipo.suave, 1)
  .rotar(180)
  .mover_x(200)
  .rotar(180)
  .mover_hasta(0, 0);
```

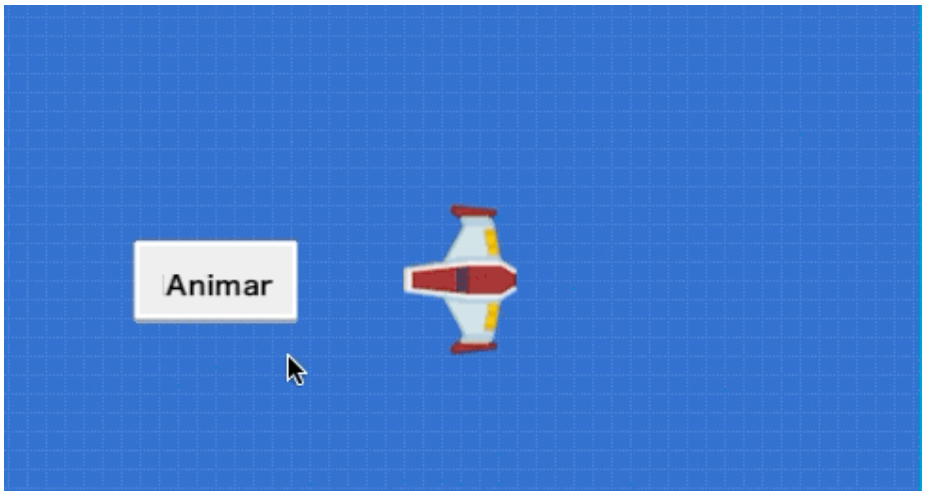


Figure 56: animacion-de-nave

Animaciones soportadas

Existen varias propiedades que se pueden animar, como la posición de los actores, la rotación, transparencia etc...

Para ver el listado completo de estas animaciones te recomiendo mirar la ayuda que te ofrece el editor justo después cerrar el paréntesis que inicia la animación:

Algunas animaciones esperan parámetros, como el valor esperado para la propiedad y la duración total de la animación, así que nuevamente ten en cuenta que el editor te ayudará a conocer los parámetros que espera cada una de las animaciones.

Tipos de animaciones

El primer argumento de la función `this.animar` pude ser uno de estos:

- `Tipo.lineal`
- `Tipo.suave`
- `Tipo.desborde`
- `Tipo.rebote`
- `Tipo.elastico`

Ahora bien, ¿qué significan estos valores?, ¿En qué se diferencian?: Las animaciones son básicamente transformaciones que se le hacen a las propiedades a través del tiempo, así que estos tipos de animaciones se diferencian en cómo la animación progresará en el tiempo.

Veamos el tipo de animación `Tipo.lineal` primero, que es uno de los más

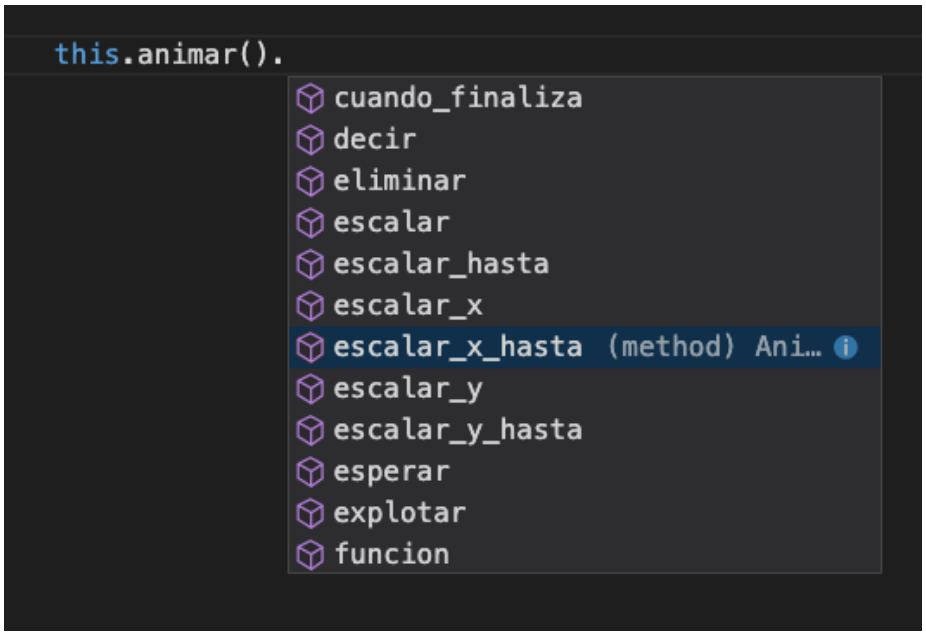


Figure 57: animacion-de-nave

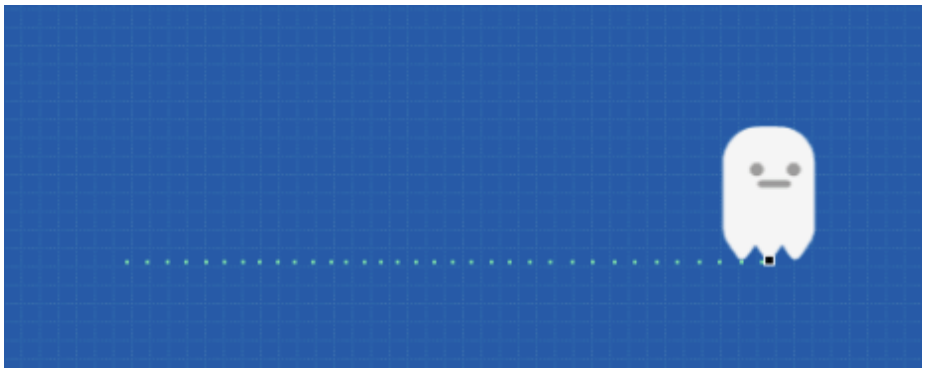
sencillos:

Tipo.Lineal

Imagina que tenemos un actor y le pedimos hacer un movimiento en el eje `x` de forma lineal con este código:

```
this.animar(Tipo.lineal).mover_x(350);
```

Si dibujamos cómo se mueve un actor en el tiempo usando una animación de tipo `lineal` vamos a ver que el movimiento se hace de forma constante, avanzando siempre a la misma velocidad:

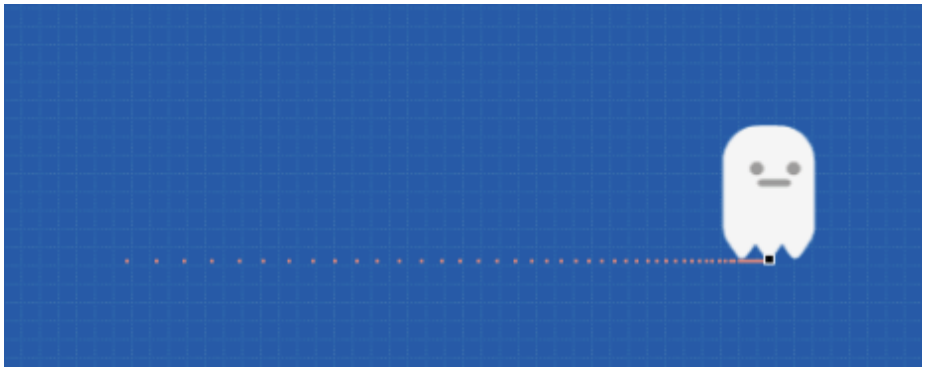


¿Y cómo son el resto de las animaciones?.

Tipo.Suave

La animación de tipo `suave` es similar a la lineal, pero hace que el actor se mueva con cierta desaceleración:

```
this.animar(Tipo.suave).mover_x(350);
```



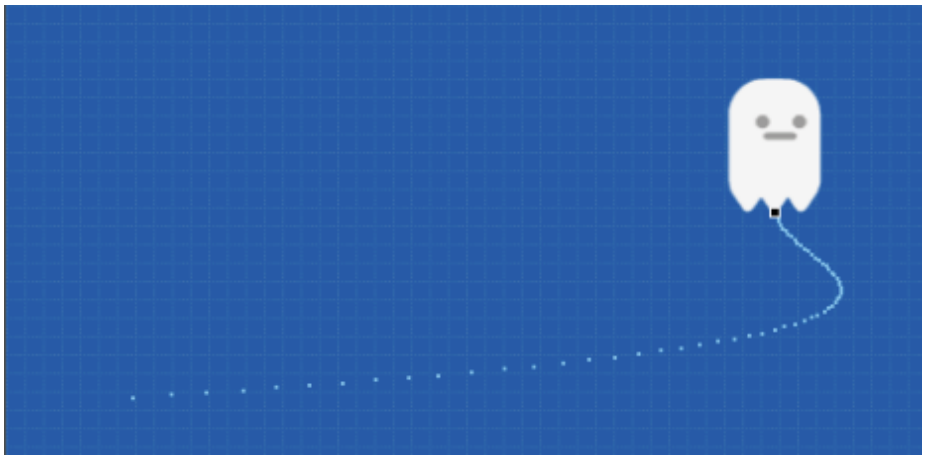
Esto se puede notar porque los puntos por los que pasó el actor se ven cada vez más juntos.

Tipo.desborde

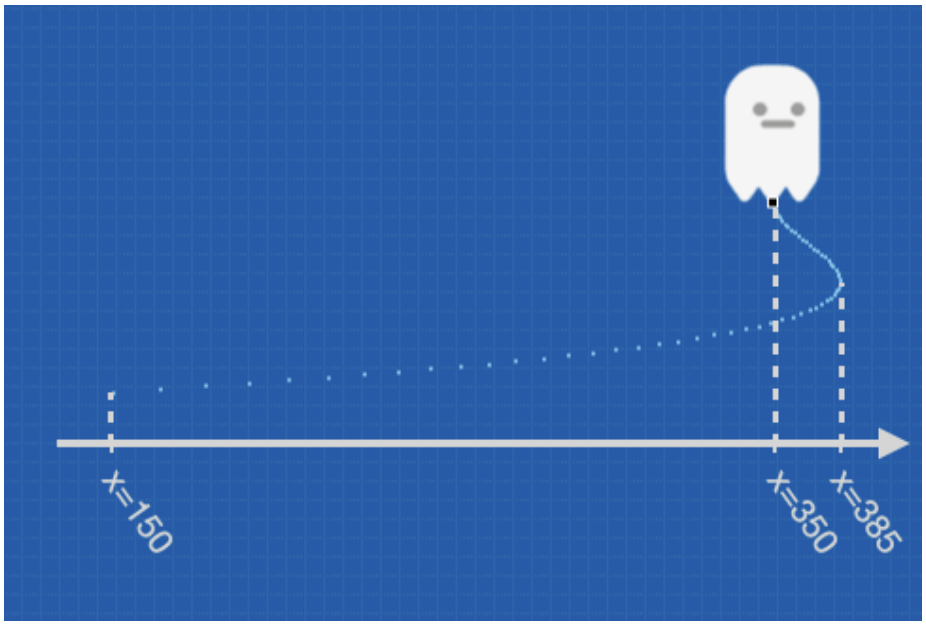
La animación de tipo `desborde` superará el punto de llegada, pero retomará la posición para corregir el desvío.

```
this.animar(Tipo.desborde).mover_x(350);
```

Para que este gráfico se vea un poco mejor, se combinó con un movimiento vertical:



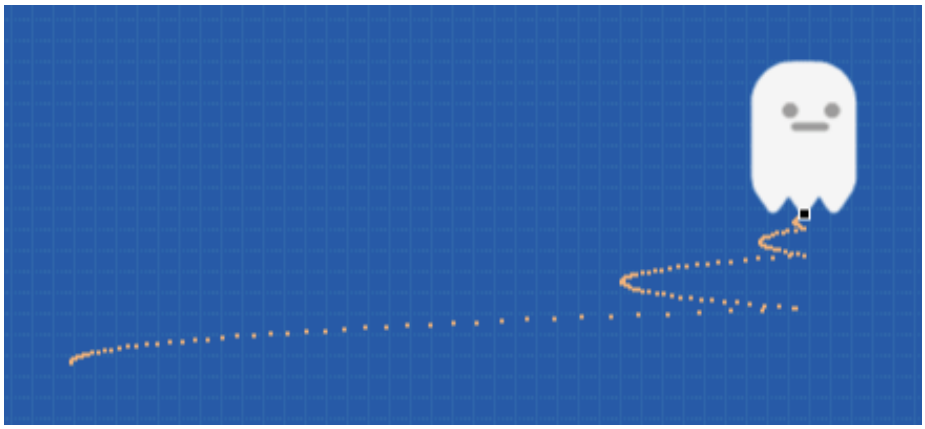
Observa como cerca del punto de llegada el personaje superó la coordenada 350, a la que le habíamos pedido que se mueva, pero suavemente retomó para corregir el desborde:



Tipo.rebote

La animación de tipo rebote llegará al punto solicitado pero va a regresar y corregir la posición varias veces:

```
this.animar(Tipo.rebote).mover_x(350);
```

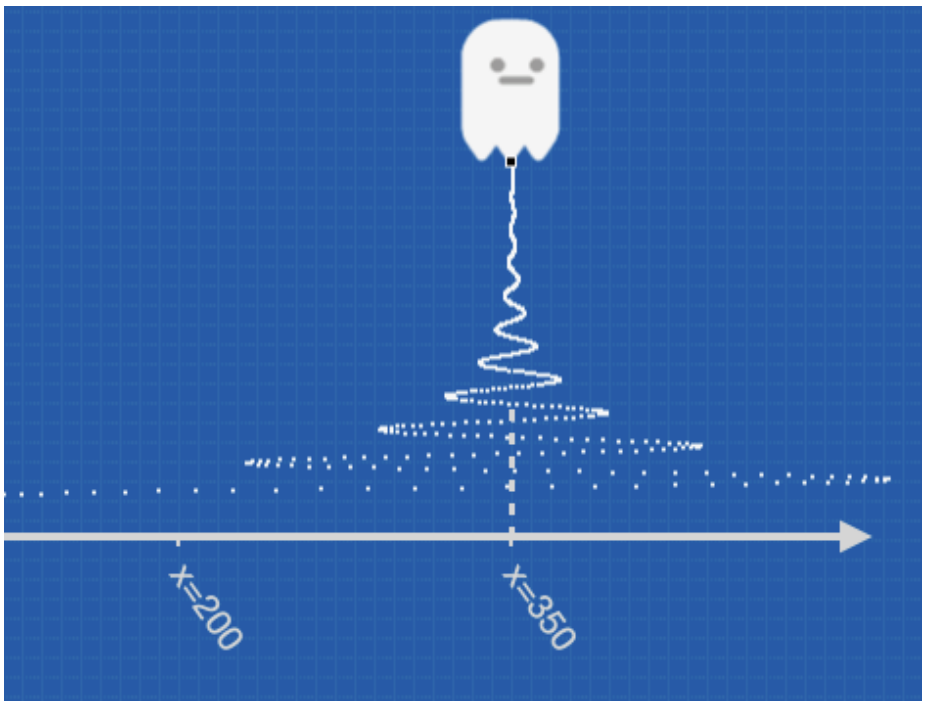


Es ideal para movimiento que aparenten ser mecánicos, como giros de engranajes o palancas.

Tipo.elastico

Las animaciones elásticas son las más llamativas cuando se utilizan en movimientos y rotaciones:

```
this.animar(Tipo.elastico).mover_x(350);
```



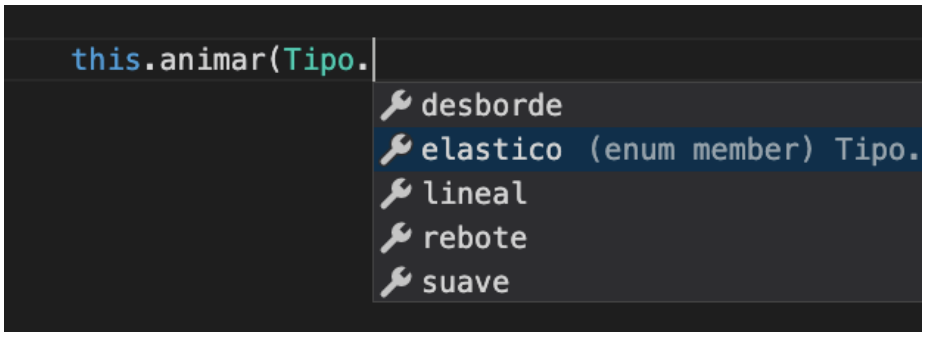
Este tipo de animaciones también se ven muy bien cuando se aplican a escalas, por ejemplo si quieres hacer que un actor parezca “esponjoso” podrías lograrlo combinando dos animaciones así:

```
this.escala_x = 0.6;
this.escala_y = 1.4;
this.animar(Tipo.elastico, 1, 5).escalar_x_hasta(1);
this.animar(Tipo.elastico, 1, 5).escalar_y_hasta(1);
```

Comparando tipos de animaciones

Una forma de ayudarte a reconocer los tipos de animaciones es probar el ejemplo “tipos de animación” de los ejemplos de pilas:

Sin embargo, mi recomendación es que pruebes qué tipo de animación funciona mejor en el juego que estés haciendo, recuerda que cuando escribas el código el propio editor te guiará por todas las opciones disponibles:



Músicas y sonidos

La música y el sonido de un videojuego es clave para lograr que el juego sea divertido y genere reacción en las personas que lo juegan, por ese motivo pilas también incluye un gestor de sonidos incluir audio en tus videojuegos.

Para gestionar los sonidos del proyecto tienes que pulsar el botón “Sonidos” de la interfaz:

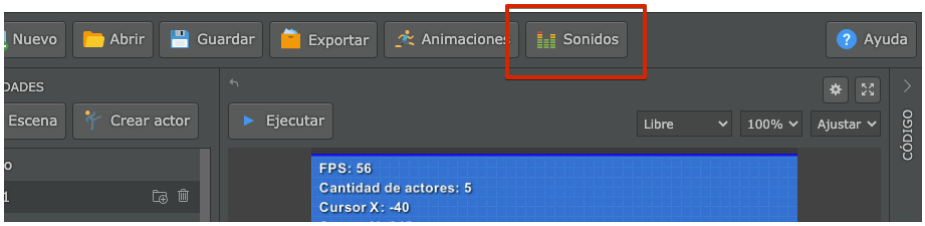


Figure 58: sonidos

Luego vas a ver una ventana con varios sonidos pre cargados, la posibilidad de subir sonidos personalizados y algunos códigos de ejemplo para reproducir estos sonidos desde el código:

Cómo reproducir sonidos desde el código

Cada sonido tiene su propio nombre dentro del gestor de sonidos. Este nombre es super importante porque es el que nos va a permitir distinguir los diferentes sonidos, o músicas, que incluye el juego.

¿Qué diferencia la música de los sonidos?

Tal vez habrás notado que para pilas la música y los sonidos se cargan como archivos `.mp3` en la misma ventana, y de hecho comparten el mismo listado.

Sin embargo hay una diferencia cómo pilas gestionan c estos dos tipos de archivos de audio:

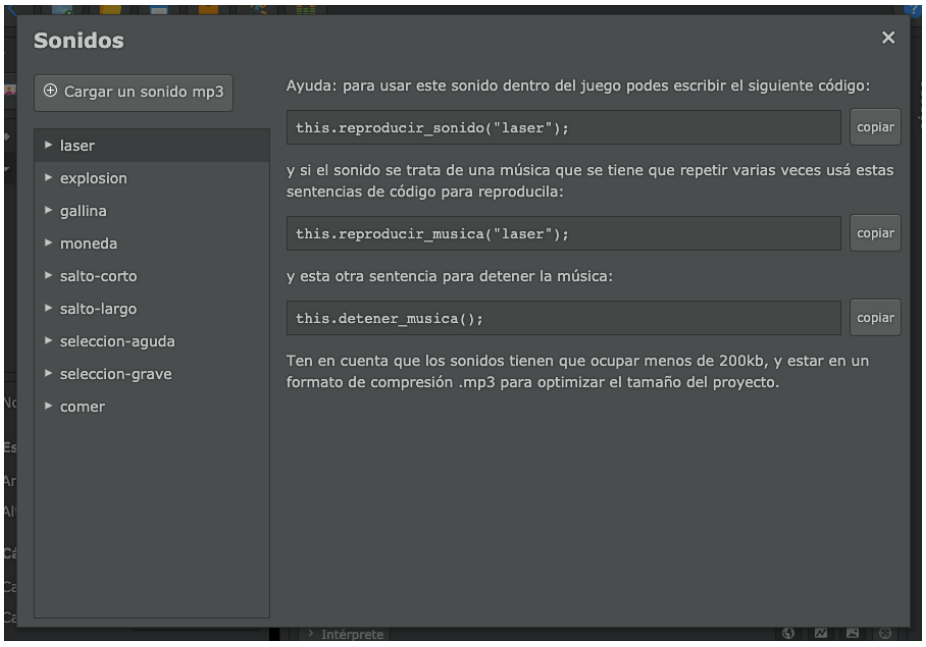


Figure 59: dialogo

- La música se puede reproducir una a la vez. Y cuando el audio de la música finaliza vuelve a comenzar. Esto es ideal para músicas estilo "loop", que se pueden repetir todo el tiempo.
- Los sonidos se pueden reproducir en simultáneo, tantas veces como se quiera y junto a otros sonidos.

Ahora bien, esta distinción la tenemos que expresar en el código: cada vez que queramos reproducir una música, debemos llamar a la función:

```
this.reproducir_musica(nombre);
```

lo que hará que comience a reproducirse ese audio (infinitas veces) y se detendrá cualquier otra música.

Mientras que si queremos reproducir un sonido podemos llamar a la función:

```
this.reproducir_sonido(nombre);
```

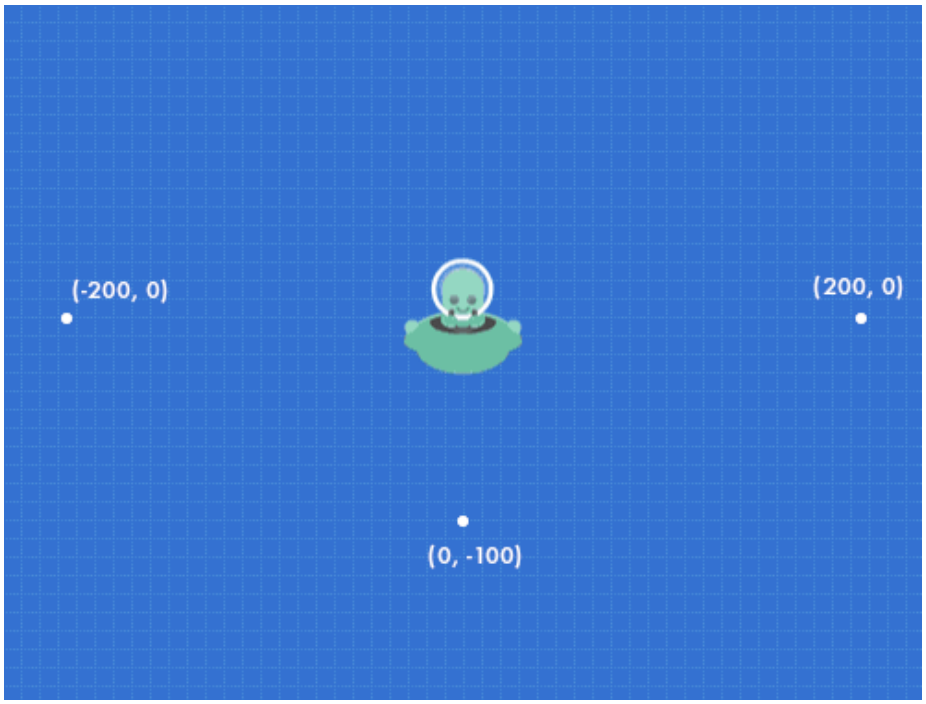
Recorridos

Los recorridos nos permiten realizar movimientos de actores de manera muy simple, lo que tenemos que hacer es enumerar todos los puntos "x" e "y" que queremos recorrer y llamar a una sola función.

Usando la función hacer_recorrido

Por ejemplo, imagina que tenemos un enemigo de juego de naves, un ovni con un alien, y queremos que se mueva por la pantalla de izquierda a derecha.

Primero tenemos que tener en cuenta qué puntos de la pantalla queremos que recorra. Por ejemplo, que primero visite la posición `(x= -200, y=0)`, luego hacia abajo a la posición `(x=0, y=-100)`, luego a la derecha `(x=200, y=0)` y por último que regrese al punto de origen `(x=0, y=0)`:



Y por último, tenemos que llamar a la función `hacer_recorrido` de esta forma dentro de una actor:

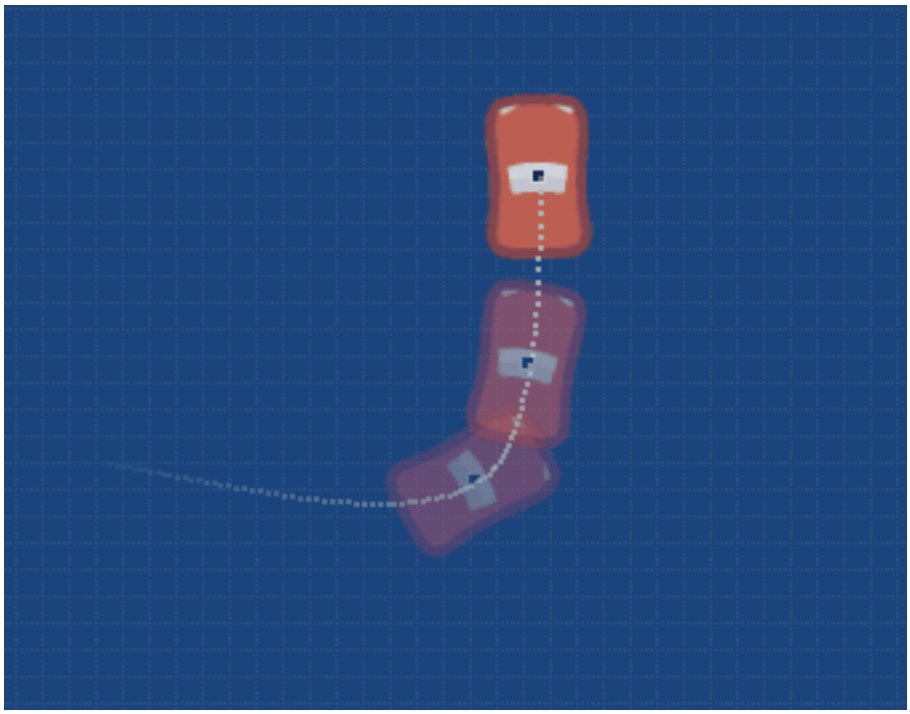
```
this.hacer_recorrido([-200, 0, 0, -100, 200, 0, 0, 0], 7, 1, false);
```

El primer parámetro tiene que ser una lista con todas las coordenadas que queremos que recorra. Tenemos que ingresar el número de la coordenada `x`, luego en de la coordenada `y` y seguir así. Esta lista tiene que tener la forma `[x_punto_1, y_punto_1, x_punto_2, y_punto_2 ....]`

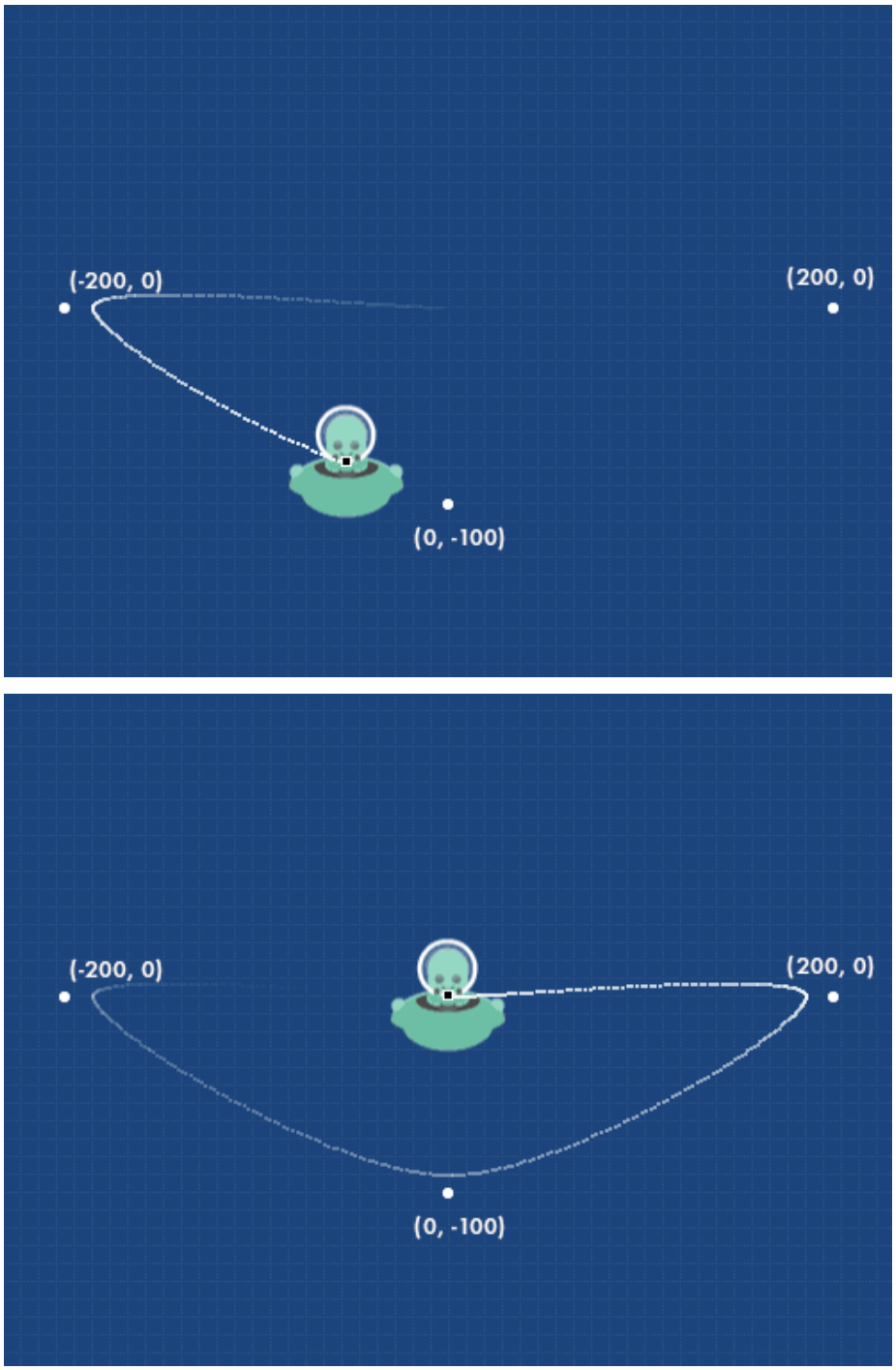
El segundo argumento, que en este ejemplo es `7`, tiene que ser un número indicando cuantos segundos tiene que durar el movimiento. En este caso se pide que el movimiento dure 7 segundos.

El ante último argumento es el número que indicará las repeticiones de esta animación, si le pasamos `0` el movimiento se va a repetir constantemente.

Y el último argumento indica si el actor deber rotar en dirección al camino o no. Es útil poner este argumento en `true` cuando el actor es una nave vista desde arriba o un automóvil:



Ten en cuenta que el actor va a visitar todos los puntos pero de manera gradual, siguiendo el recorrido pero realizando una curva de movimiento muy suave:


(-200, 0)
(200, 0)
(0, -100)
(-200, 0)
(200, 0)
(0, -100)

Cámara

Si bien el juego tendrá un tamaño fijo en la pantalla, pilas incorpora una cámara que nos permite realizar desplazamientos para mostrar otras partes de la escena.

En esta sección vamos a controlar cámara de la escena tanto desde el editor como desde el juego en ejecución.

Mover la cámara desde el editor

Para desplazar la cámara simplemente tenemos que pulsar y arrastrar con el mouse sobre el fondo de la escena:

Vas a notar que en la escena el movimiento de la cámara se refleja en los atributos "Cámara x" y "Cámara Y".

Esos dos atributos también se utilizan para definir la posición inicial de la cámara cuando comienza a ejecutarse el juego.

También vas a observar otros dos atributos llamados "Ancho" y "Alto". Estos dos atributos van a definir el tamaño total del escenario. Podes cambiar esos parámetros para hacer que el escenario total sea más grande o más chico.

Mover la cámara mientras se ejecuta el juego

Para mover la cámara podemos acceder a los atributos de posición directamente, por ejemplo si quieres mover constantemente la cámara hacia la derecha, podrías escribir este código en la escena:

```
class escena1 extends Escena {
  actualizar() {
    this.camara.x += 1;
  }
}
```

O bien, si quieres que un actor se mueva libremente por el escenario y la cámara la siga todo el tiempo podrías usar el método `seguir_al_actor` así:

```
class nave extends Actor {
  actualizar() {
    this.camara.seguir_al_actor(this, 10, true);
  }
}
```

Ten en cuenta que esta función se tiene que llamar todo el tiempo, por eso la tenemos que incluir dentro del método actualizar.

También ten en cuenta que la función lleva 3 parámetros:

```
seguir_al_actor(actor, suavidad, ignorar_bordes);
```

- `actor`: el actor que la cámara tiene que seguir.
- `suavidad`: tiene que ser un número indicando cuán gradual tiene que ser el movimiento de la cámara, cuanto menor sea el número mas rápido y brusco será el movimiento de la cámara.
- `ignorar_bordes`: el último parámetro tiene que ser `true` o `false`. Si se envía `true` la cámara seguirá al actor incluso si sale del area del escenario, esto es ideal para juegos con area de movimiento infinita. Si se coloca `false` la cámara se detendrá en los bordes del escenario.

Zoom o aumento

La cámara de pilas también permite hacer acercamientos, o zoom, usando un atributo llamado `escala`.

Por ejemplo, si queremos aumentar el acercamiento de la cámara podemos usar el atributo `escala` con valores superiores a `1`:

```
this.camara.escala = 2.5;
```

O bien regresar al zoom original escribiendo:

```
this.camara.escala = 1;
```

Fijar actores a la pantalla

Hay casos en donde queremos que algún actor permanezca fijo en su posición, independiente del movimiento de la cámara.

Por ejemplo, el puntaje del jugador no es algo que queremos que salga de la pantalla cuando la cámara se mueve, tampoco los controles, ni el contador de vidas.

Para que este tipo de actores quede visible en pantalla tenemos tenemos que definirles la propiedad `fijo` de esta forma:

```
this.fijo = true;
```

Esa propiedad en valor `true` le indica a pilas que este actor no se tiene que desplazar respetando la cámara, sino que debe permanecer fijo a la pantalla en todo momento.

Autómatas y estados

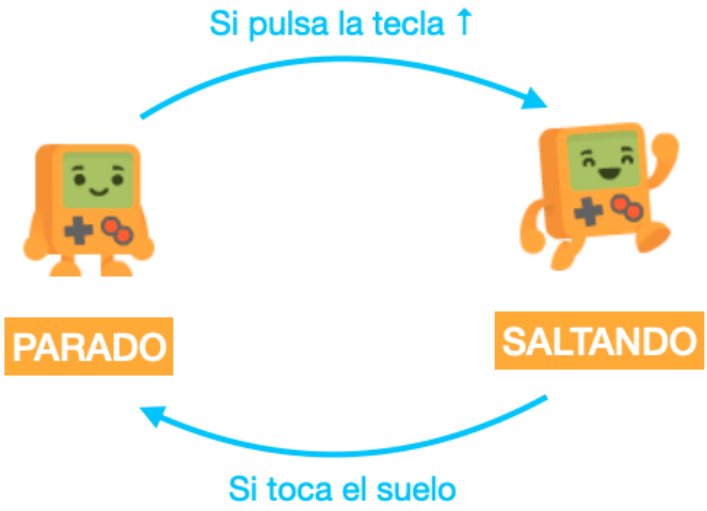
Un desafío muy común en el desarrollo de juegos es lograr que los actores puedan mostrar animaciones y reaccionar de acuerdo a modos particulares: saltar, caminar, quedarse parado, perdiendo, etc...

Para estas cosas Pilas incorpora un mecanismo de autómatas y estados.

Un actor puede estar en un estado a la vez, y solo se puede mover a otros estados a través de transiciones.

Por ejemplo, imaginemos un actor de un juego de plataformas que solo puede hacer dos cosas, saltar y quedarse parado en el lugar:

Si queremos llevar este personaje a un juego nos conviene tener una técnica para permitirle al usuario hacer cualquiera de esas dos acciones pero bajo determinadas condiciones. Solo podría saltar cuando está con los pies en el suelo (es decir, si está en estado "parado") y una vez que está saltando deja de hacerlo cuando toca el suelo:



A este tipo de gráficos lo llamamos diagrama de estados, o autómata, y si bien puede parece un simple modelo gráfico, lo cierto es que resulta muy útil a la hora de escribir el código en nuestro juego.

Implementando estados en pilas

Siguiendo con nuestro ejemplo, el personaje tiene que tener dos estados. Comencemos por el primero, el estado "parado":

Para crear un estado simplemente tenemos que editar el código de un actor y asignarle un valor a la propiedad "estado" y luego crear dos métodos para que pilas sepa a qué métodos llamar cuando el actor esté en ese estado:

```
class MiActor extends Actor {
    iniciar() {
        this.estado = "parado";
    }

    actualizar() {
    }

    parado_iniciar() {
        this.animacion = "parado";
    }

    parado_actualizar() {
```

```
        // código que se ejecutará 60 veces por segundo
      // cuando el actor esté en el estado "parado".
    }
}
```

Observá que si creamos el estado "parado", los métodos que tenemos que agregar a la clase deben llamarse `parado_iniciar` y `parado_actualizar`, es decir: `nombre del estado` + `_` + `iniciar` o `actualizar`.

Ahora, para el siguiente estado llamado "saltando" tendríamos que agregar este código:

```
class MiActor extends Actor {
    // [... código anterior ...]

    saltando_iniciar() {
        this.animacion = "saltando";
    }

    saltando_actualizar() {
    }
}
```

Con esos dos estados, "parado" y "saltando", podríamos permitirle al usuario pasar de un estado a otro bajo una condición.

Por ejemplo, si queremos que el personaje pueda "saltar", pero solo cuando está "parado" y se pulsa la flechita hacia arriba del teclado podemos escribir lo siguiente:

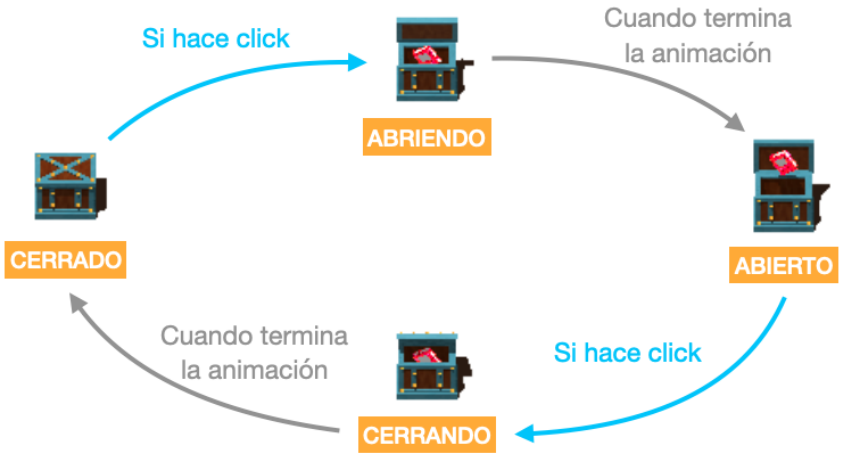
```
class MiActor extends Actor {
    // [... código anterior ...]

    parado_actualizar() {
      if (this.control.arriba) {
        this.estado = "saltando";
      }
    }

}
```

Es decir, con solo asignarle un valor a la variable `estado` le estaremos ordenando a pilas que realice una transición del estado "parado" a "saltando".

Más referencias

Te recomendamos mirar alguno de los ejemplos que incluye pilas sobre autómatas, el más sencillo se llama "automata-cofre", que simplemente muestra una actor

tipo cofre con 4 estados:



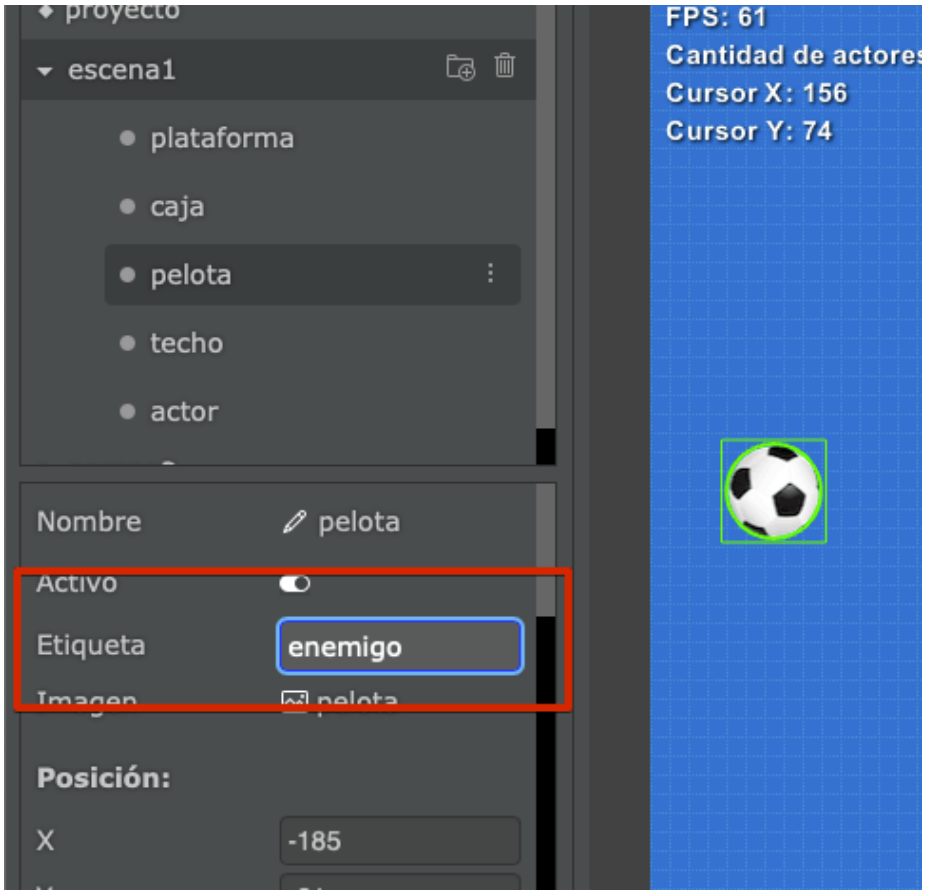
Otro ejemplo muy interesante se llama "automata-contra-calaveras", donde el jugador puede controlar un personaje que tiene solo 3 estados: "parado", "caminando" y "golpeando":

Etiquetas

Las etiquetas son útiles para clasificar actores y simplificar interacciones.

Por ejemplo, en un juego podríamos tener 10, 20 o 30 actores diferentes que tengan el rol de "enemigos". En cuyo caso simplemente podríamos asignarle a todos la misma etiqueta para distinguirlos de los demás.

Las etiquetas se pueden definir directamente en el inspector de pilas:



O bien, también se puede directamente desde el editor código usando el atributo **etiqueta** así:

```
class Caja extends ActorBase {
  iniciar() {
    this.etiqueta = "enemigo";
  }

  actualizar() {}
}
```

Etiquetas para distinguir colisiones

Vamos describir un uso típico de las etiquetas: las etiquetas son muy útiles para distinguir colisiones.

Cuando pilas detecta una colisión entre dos actores llamará a la función *cuando_comienza_una_colision* y enviará el parámetro *actor* como referencia

al actor que entró en contacto.

Así que esto es lo que se suele hacer, imaginemos que nuestro actor *protagonista* pueda capturar monedas pero debe perder vidas si toca un enemigo. Podríamos hacer algo así:

```
class Protagonista extends Actor {
  // otras funciones...

  cuando_comienza_una_colision(actor) {
    if (actor.etiqueta === "moneda") {
      actor.eliminar();
      this.pilas.reproducir_sonido("moneda");
      // sumar puntaje, emitir particulas etc...
    }

    if (actor.etiqueta === "enemigo") {
      this.estado = "perder";
      this.vidas -= 1;
      // emitir sonido de game over, mostrar textos etc...
    }
  }
}
```

Ten en cuenta que en la función `cuando_comienza_una_colision` también se puede cancelar una colisión retornando `true`. Por ejemplo si quieres que un actor pueda sobrepasar a otro.

Habilidades

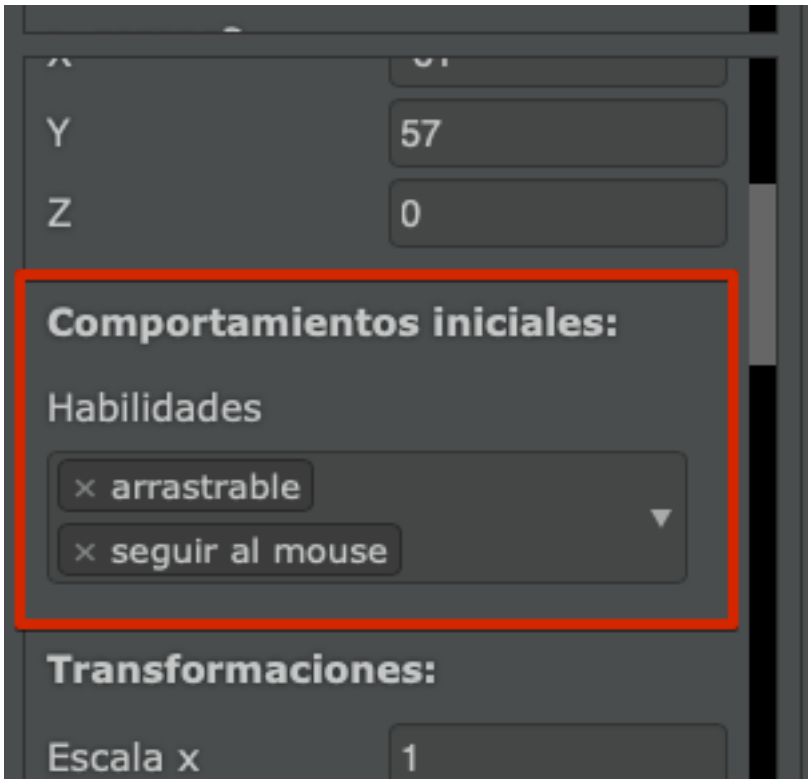
Cuando realizamos juegos hay ciertos comportamientos que son tan comunes que ni siquiera necesitamos programarlos escribiendo código.

Un ejemplo de estos comportamientos comunes incluyen “hacer que un actor se mueva con el teclado”, “poder arrastrar y soltar un actor con el mouse” etc...

A todos estos comportamientos los llamamos “habilidades”, y lo interesante es que podemos tomar a cualquier actor y “enseñarle” cualquier de las habilidades para mejorar nuestro juego.

Desde el editor

Una opción para incorporarle habilidades a un actor es usar el inspector del actor. Hay una propiedad llamada “Habilidades” en donde cargar una o más habilidades que busquemos enseñarle a un actor al momento de iniciar la escena.



Desde el código

Otra opción, es enseñarle una habilidad al actor desde el código.

Imaginá que buscamos crear un juego de cartas, donde el usuario tiene que poder mover una carta por la pantalla usando el mouse o gestos del dispositivo mobile. El primer paso es crear el actor, con la imagen que nos interesa:

```
let mi_actor = pilas.actores.actor();
mi_actor.imagen = "imagenes:cartas/carta";
```

Y aquí lo interesante, para hacer que el usuario pueda mover este actor por la pantalla simplemente tenemos que "enseñarle" al actor la habilidad arrastrable así:

```
mi_actor.aprender("arrastrable");
```

La lista de habilidades disponibles dentro de pilas es la siguiente:

- "arrastrable"
- "mover con el teclado"
- "oscilar rotacion"
- "oscilar transparencia"

- “oscilar verticalmente”
- “rotar constantemente”
- “seguir al mouse lentamente”
- “seguir al mouse”

Olvidar o eliminar habilidades

Si en algún momento quieres eliminar una habilidad de un actor, simplemente se puede llamar a la función `olvidar` así:

```
mi_actor.olvidar("arrastrable");
```

Habilidades personalizadas

Pilas también te permite crear tus propias habilidades.

Primero, deberías crear una clase que herede de `Habilidad` e implementar los métodos `iniciar`, `actualizar` y `terminar`.

Por ejemplo, aquí tenemos el código de una habilidad que se debería colocar en la parte inferior de cuaquier actor:

```
class GirarMuyRapido extends Habilidad {
  iniciar() {
    // aquí puedes acceder a this.actor
  }

  actualizar() {
    this.actor.rotacion += 15;
  }

  terminar() {
    // aquí puedes acceder a this.actor
  }
}
```

Luego, puedes tomar algún actor de tu juego y enseñarle esta habilidad usando dos sentecias como estas:

```
this.pilas.habilidades.vincular("girar muy rapido", GirarMuyRapido);
this.aprender("girar muy rapido");
```

Lo bueno de estas habilidades es que una vez que las declaras se pueden usar en cualquier otro actor, lo único que deberías hacer es agregar este código al método `iniciar` del actor:

```
this.aprender("girar muy rapido");
```

Comportamientos

En el desarrollo de videojuegos es conveniente tener una forma de indicarle a los actores una rutina o tarea para que la realicen.

En pilas usamos el concepto de comportamiento. Un comportamiento es un objeto que simboliza una acción a realizar por un actor.

La utilidad de usar componentes es que puedes asociarlos y intercambiarlos libremente para lograr efectos útiles.

Por ejemplo: un guardia de un juego de acción puede ir de un lado a otro en un pasillo:

- caminar hacia la izquierda hasta el fin del pasillo.
- dar una vuelta completa.
- caminar hacia la derecha hasta el fin del pasillo.
- dar una vuelta completa.
- etc...

En este caso hay 4 comportamientos, y queda en nuestro control si queremos que luego de los 4 comportamientos comience nuevamente.

Realizando comportamientos

Para indicarle a un actor que realice un comportamiento tenemos que llamar a la función `hacer` indicándole el nombre del comportamiento:

```
let mi_actor = pilas.actores.actor();
mi_actor.hacer("aparecer");
```

Ten en cuenta que los comportamientos se pueden encadenar para lograr efectos o animaciones una detrás de otra. Por ejemplo, si queremos mostrar y ocultar un actor podemos hacerlo así:

```
let mi_actor = pilas.actores.actor();
mi_actor.hacer("aparecer");
mi_actor.hacer("desaparecer");
```

Incluso, algunas habilidades reciben parámetros para indicar velocidad, posición etc...

Por ejemplo, si queremos hacer que el actor desaparezca y aparezca varias veces muy rápidamente podemos escribir algo así:

```
let mi_actor = pilas.actores.actor();

for (i=0; i<50; i++) {
  mi_actor.hacer("aparecer", {velocidad: 10});
  mi_actor.hacer("desaparecer", {velocidad: 10});
}
```

Comportamientos incluidos en pilas

Esta es la lista de comportamientos que incluye pilas:

- desaparecer (parámetros: velocidad)
- aparecer (parámetros: velocidad)
- eliminar (sin parámetros)
- mover (parámetros: x, y, demora)

Comportamientos personalizados

Si quieres crear tus propios comportamientos podrías abordarlo de la siguiente forma:

Primero, deberías crear una clase que herede de `Comportamientos` y tenga al menos un método iniciar y actualizar así:

```
class MiComportamiento extends Comportamiento {

  iniciar(argumentos) {
  }

  actualizar() {
    // retornar true para detener el comportamiento.
  }

  terminar() {}
}
```

Luego, deberías vincularlo al módulo de comportamientos:

```
pilas.comportamientos.vincular("mi comportamiento", MiComportamiento);
```

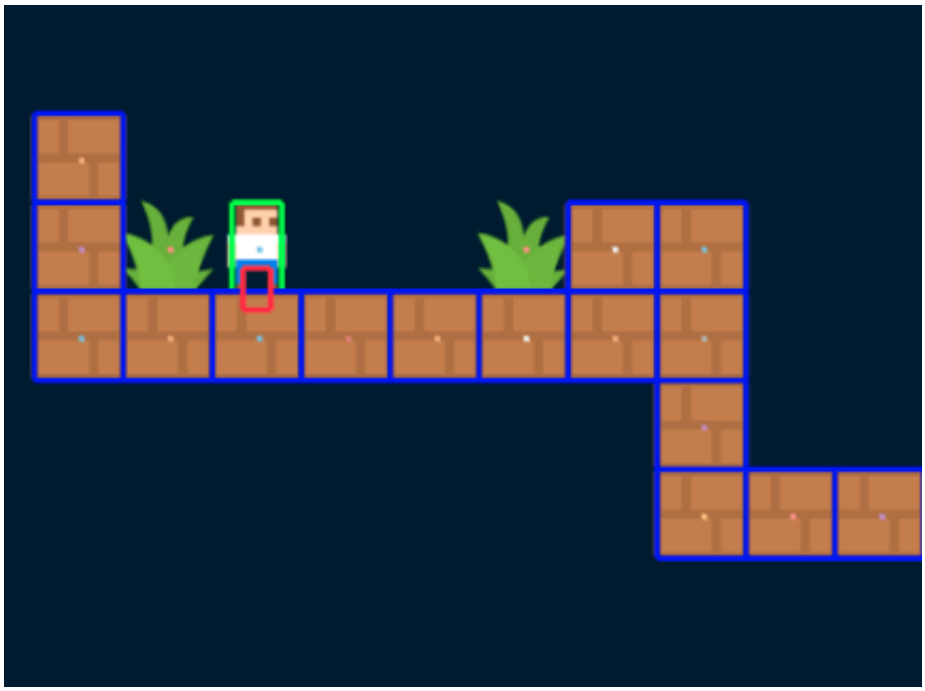
y por último asignárselo a un actor:

```
mi_actor.hacer("mi_comportamiento");
```

Mapas y niveles

Los escenarios de muchos videojuegos se construyen utilizando una técnica llamada tiles, que consiste en utilizar pequeños bloques llamados "tiles" para crear niveles grandes.

Por ejemplo, en esta imagen se puede ver un nivel diseñado usando esta técnica:



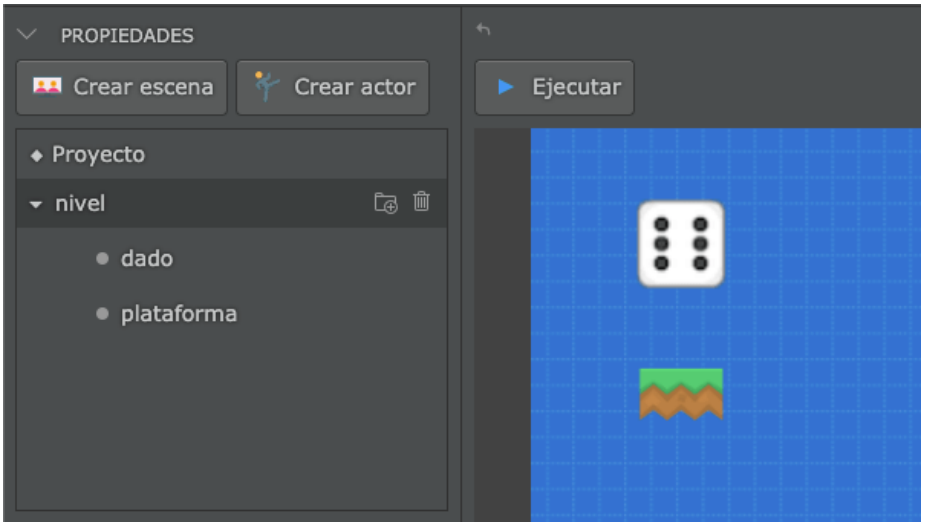
Esta técnica es muy útil porque permite crear escenarios de manera sencilla. Simplemente repitiendo bloques y reutilizando actores.

Pilas incluye dos funciones para que podamos hacer niveles de esta forma. Por un lado tenemos una función para definir los bloques que vamos a utilizar y luego una función para dibujar el escenario en pantalla.

Cómo dibujar un mapa en pantalla

El primer paso es definir qué actores vamos a utilizar como bloques, tenemos que tener esos actores en cualquiera de las escenas.

Por ejemplo, aquí tengo dos actores, uno llamado “dado” y el otro “plataforma”:

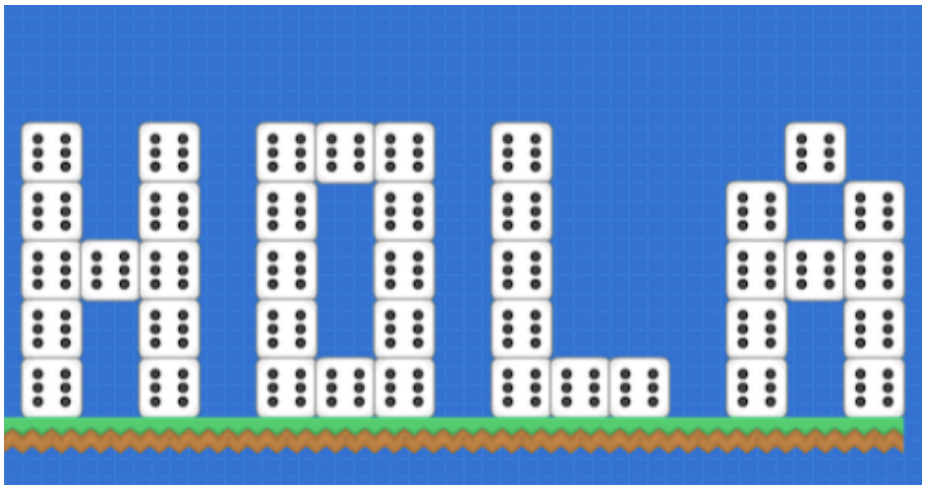


Si quiero crear un nivel usando estos actores tengo que indicarle a pilas cómo llamar a estos actores en la definición de un mapa:

```
this.pilas.definir_mapa({
  "d": "dado",
  "p": "plataforma"
});
```

De esta forma, pilas sabrá que cada que vez que usemos la letra `d` en un mapa nos estaremos refiriendo a un dado y cada vez que usemos la letra `p` estaremos refiriéndonos a una plataforma.

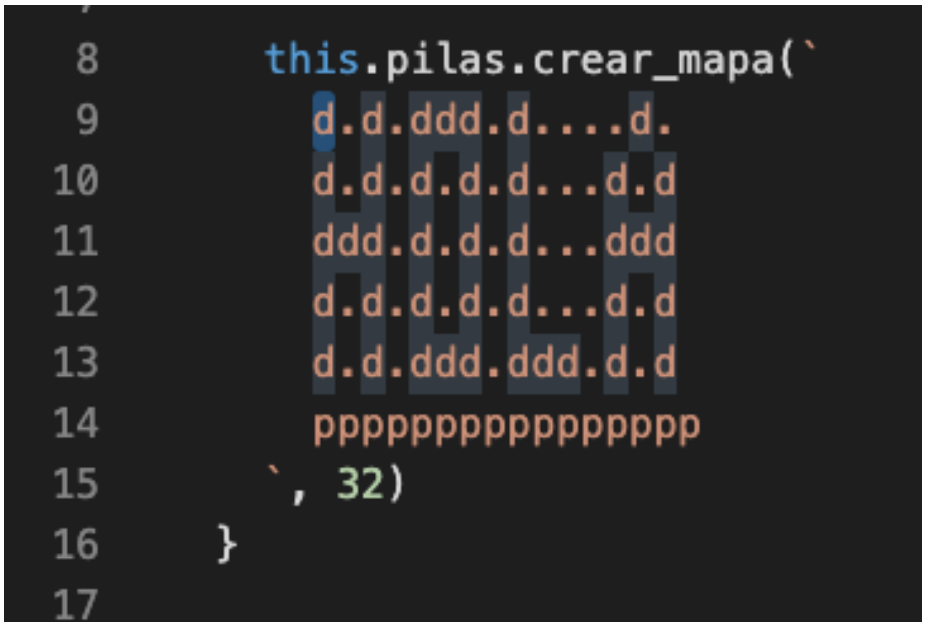
Ahora, imagina que queremos dibujar el mensaje “HOLA” usando dados y luego poner una plataforma debajo del mensaje así:



Para esto podemos usar la función `pilas.crear_mapa` y las teclas `d` y `p` que definimos antes así:

```
this.pilas.crear_mapa(`
      d.d.ddd.d....d.
      d.d.d.d.d...d.d
      ddd.d.d.d...ddd
      d.d.d.d.d...d.d
      d.d.ddd.ddd.d.d
      pppppppppppppppp
`, 32)
```

Oh, un pequeño consejo: suele ayudarte a leer mejor el mapa en texto si seleccionar el caracter que estás ingresando con el editor:



También puede ayudarte a mejorar la legibilidad si usas separadores como ".", "-" o espacios en aquellos lugares donde no necesitas dibujar un actor.

Azar o cálculos aleatorios

Una utilidad muy utilizada en los juegos son las funciones para obtener números aleatorios. Pilas incluye una función muy simple para esto llamada `pilas.azar`:

La función `pilas.azar` retorna un número aleatorio entre dos números. Por ejemplo, si buscas que pilas retorne un número al hacer entre 1 y 5 podrías llamar a la función así:

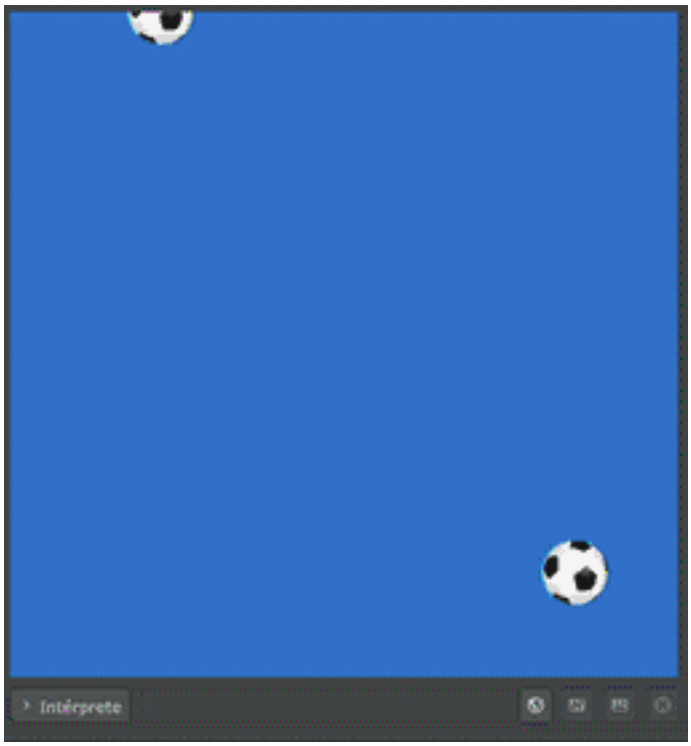
```
pilas.azar(1, 5);
```

Es importante mencionar que el rango de valores incluye los extremos. En el caso anterior la función podría retorna 1, 2, 3, 4 o 5.

Un ejemplo simple

La función `pilas.azar` se podría utilizar para definir la posición inicial de un actor de modo tal que sea impredecible para el usuario.

Por ejemplo, imagina que queremos crear actores de tipo `Pelota` que caigan desde la parte superior de la pantalla así:



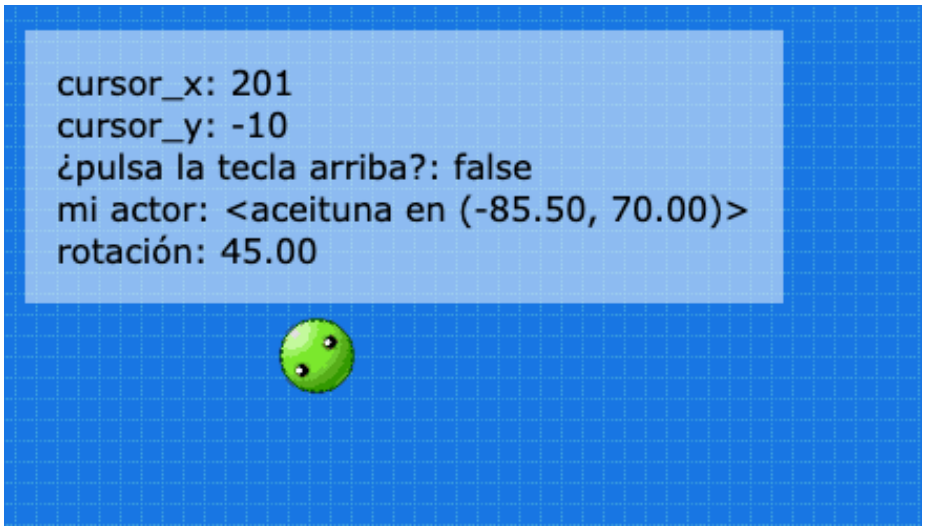
Para esto podemos crear los actores desde la función `cada_segundo` de la escena. Y a su vez, a cada actor que creamos deberíamos asignarle una posición `x` aleatoria así:

```
class escena1 extends Escena {
  iniciar() {}

  actualizar() {}

  cada_segundo() {
    let pelota = this.pilas.actores.pelota();
    pelota.y = 250;
    pelota.x = this.pilas.azar(-200, 200);
  }
}
```

Observables

Los observables te permiten visualizar rápidamente en pantalla qué valor tienen los atributos, variables o expresiones dentro de un juego.

Por ejemplo, imagina que tu juego tiene un personaje que interactúa con la posición `x` e `y` del cursor del mouse. Mediante observables vas a poder agregar un visor de variables como este:



Es super importante poder visualizar rápidamente variables en un juego, no solo para informarse sino también para detectar errores o incluso saber si estamos accediendo a la información correcta.

Los observables se pueden crear mediante una llamada a la función `pilas.observable` enviando un nombre (a elección tuya) y una variable para observar, por ejemplo:

```
actualizar() {
  this.pilas.observar("cursor_x", this.pilas.cursor_x);
  this.pilas.observar("cursor_y", this.pilas.cursor_y);
```

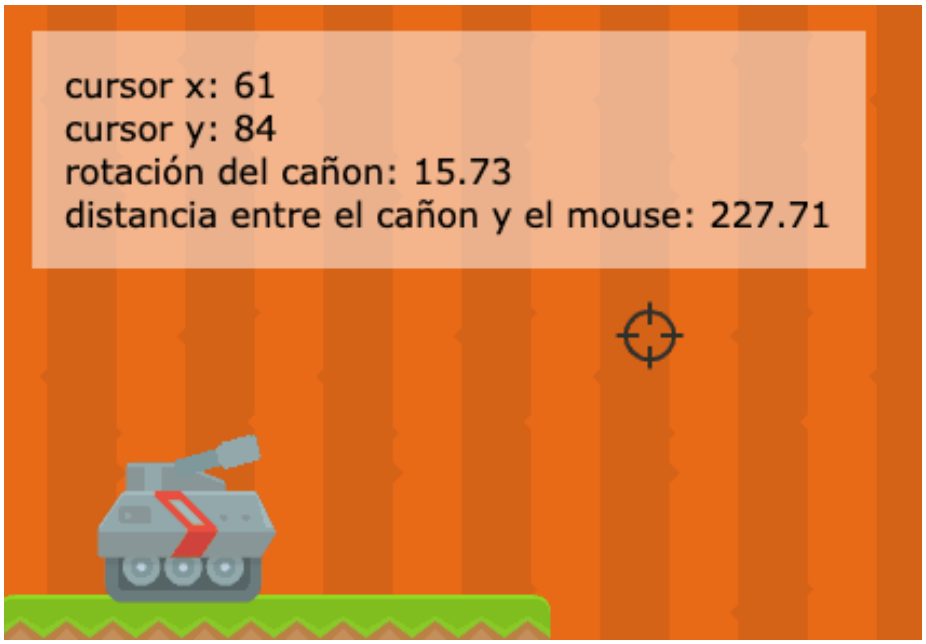
```
  this.pilas.observar("¿pulsa la tecla arriba?", this.pilas.control.arriba);
}
```

Ten en cuenta que los observables se tienen que crear siempre desde funciones `actualizar`, ya sea de actores o de la escena, porque esa es la única forma de que pilas pueda saber el valor más reciente de esas variables.

Otro ejemplo útil, sería poder saber la posición exacta de un actor que hemos creado. Incluso podemos imprimir otros atributos menos visibles como la rotación o la velocidad vertical:

```
class un_actor extends Actor {
  iniciar() {
    this.x = -100;
    this.y = 70;
  }

  actualizar() {
    this.pilas.observar("mi actor", this)
    this.pilas.observar("rotación", this.rotacion)
  }
}
```

Si buscas un ejemplo aplicado de observables abrí el ejemplo angulos:



Manejo de tiempo

Pilas ofrece varias formas de ejecutar funciones cada determinado tiempo. Esto

es útil para crear enemigos, crear relojes para hacer más desafiante un juego o incluso para aumentar la dificultad de un juego.

Funciones manejar el tiempo

Hay dos funciones principales en el temporizador de pilas:

- `pilas.luego(cantidad_de_segundos_a_esperar, función)`
- `pilas.cada(segundos_para_el_intervalo, función, veces_a_repetir: opcional)`

La primer función permite ejecutar una función luego de un periodo de tiempo. Por ejemplo, si queremos hacer que un actor diga algo luego de 3 segundos una vez que se coloca en pantalla podemos hacerlo escribiendo esta sentencia dentro del método `iniciar` del actor así:

```
iniciar() {

  this.pilas.luego(3, () => {
    this.decir("¡Han pasado 3 segundos!");
  });

}
```

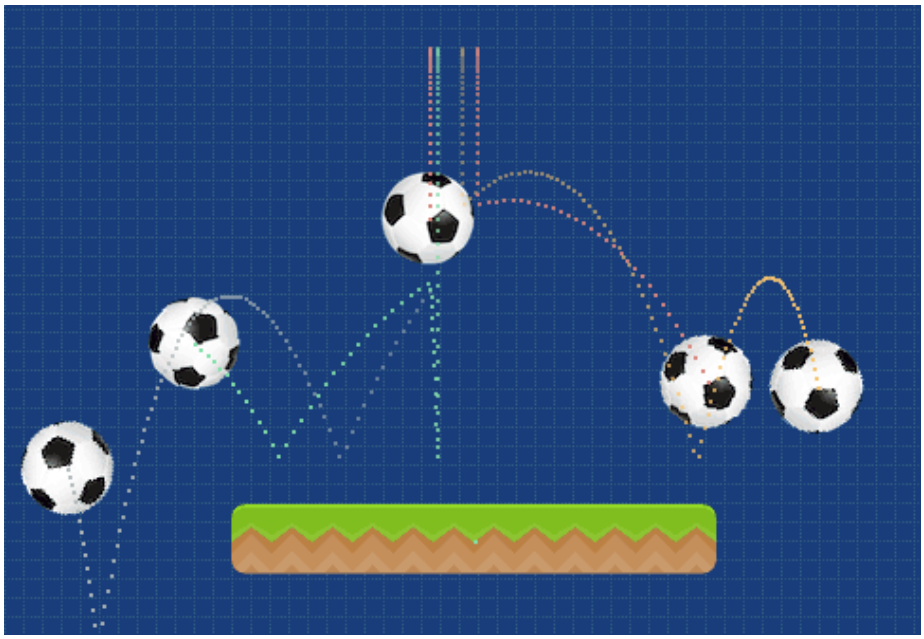
Hay que tener en cuenta que esta función se ejecutará una sola vez.

La segunda función, llamada `pilas.cada`, permite invocar una función cada determinada cantidad de segundos, muy similar a la anterior, pero invoca la función que le pidamos varias veces.

Por ejemplo, imagina que queremos crear actores `pelota` cada medio segundo, podemos hacer algo así:

```
pilas.cada(0.5, () => {
  var pelota = pilas.actores.pelota();
  pelota.y = 100;
  pelota.x = pilas.azar(-200, 200);
});
```

Si ejecutas esas instrucciones, vas a ver en pantalla algo así. Cada medio segundo se creará un actor pelota en una posición diferente:



Cancelar repeticiones o eliminar temporizadores

Si utilizas la función `pilas.cada` para repetir una acción varias veces, es probable que ante una determinada condición quieras dejas de repetir la ejecución de la función.

Una forma de lograr esto es diciéndole a pilas cuantas veces tiene que repetir la función, por ejemplo, para llamar 5 veces una función cada dos segundos deberías llamar a la función así:

```
var actor = pilas.actores.aceituna();

pilas.cada(
  2,
  () => {
    actor.decir("prueba");
  },
  5
);
```

Otra opción, es detener el temporizador directamente desde la función retornando `true`:

Por ejemplo, imagina que tienes un actor que se mueve 50 pixeles a la derecha cada 2 segundos, pero que quieres que deje de moverse cuando llegue a la posición x igual a 200. Podrías implementarlo de esta forma:

```
var actor = pilas.actores.aceituna();

pilas.cada(2, () => {
  actor.x += 50;

  if (actor.x > 200) {
    // retornar este valor 'true' hace que
    // la función deje de llamarse cada
    // dos segundos.
    return true;
  }
});
```

Métodos especiales

También existe un método llamado `cada_segundo` que los actores pueden utilizar para disparar una acción cada vez que transcurre un segundo en el juego. Por ejemplo, si queremos crear un actor que aumente su tamaño cada un segundo podemos definir el método así:

```
class caja extends Actor {
  cada_segundo() {
    this.escala += 0.5;
  }
}
```

O bien, si ya tenemos la referencia al actor también sería válido escribir algo así:

```
var mi_actor = pilas.obtener_actor_por_nombre("caja");

mi_actor.cada_segundo = () => {
  mi_actor.escala += 0.5;
  mi_actor.decir("¡pasó un segundo!");
};
```

Ángulos y distancias

Pilas trae varias funciones para realizar cálculos geométricos sencillos, como obtener la distancia entre actores o incluso el ángulo entre ellos.

Ángulo entre actores o puntos

Un ejemplo práctico sería poder mirar en dirección a donde está el puntero del mouse: Observa uno de los ejemplos que trae pilas, el usuario puede mover el puntero del mouse y observar cómo el cañon ajusta su rotación correctamente:

Este ejemplo hace uso de la función `obtener_angulo_entre_puntos` así:

```
let x = actor_cañon.x;
let y = actor_cañon.y;
let cx = pilas.cursor_x;
let cy = pilas.cursor_y;

actor_cañon.rotacion = pilas.obtener_angulo_entre_puntos(x, y, cx, cy);
```

La función `obtener_angulo_entre_puntos` espera que le enviemos 4 parámetros,

correspondientes a los dos puntos que nos interesan. El resutado de la función es el ángulo en grados esperado.

También existe una función llamada `obtener_angulo_entre_actores`, que hace algo muy similar, solamente que espera que le enviemos dos actores en lugar de 4 números.

Un ejemplo que muestra esto es `angulo-entre-actores`, que nos muestra una nave mirando constantemente al centro de la pantalla, donde está el actor planeta:

Si observas el código, vas a encontrar una llamada a `obtener_angulo_entre_actores` similar a la siguiente:

```
let nave = pilas.actores.nave();
let planeta = pilas.obtener_actor_por_nombre("planeta");
nave.rotacion = pilas.obtener_angulo_entre_actores(nave, planeta);
```

Distancia entre puntos y actores

Otras dos funciones útiles en esta categoría son las funciones `obtener_distancia_entre_puntos` y `obtener_distancia_entre_actores`. Ambas reciben los argumentos de manera muy similar a las funciones de ángulos anteriores. Aquí unos ejemplos:

```
let actor = pilas.actores.aceituna();
let cx = pilas.cursor_x;
```

```
let cy = pilas.cursor_y;

let distancia = pilas.obtener_distancia_entre_puntos(actor.x, actor.y, cx, cy);
actor.decir(`la distancia al mouse es ${distancia}`);
```

o entre actores:

```
let actor = pilas.actores.aceituna();
actor.x = 100;

let nave = pilas.actores.nave();

let distancia = pilas.obtener_distancia_entre_actores(actor, nave);
nave.decir(`la distancia al otro actor es ${distancia}`);
```

Colisiones y física

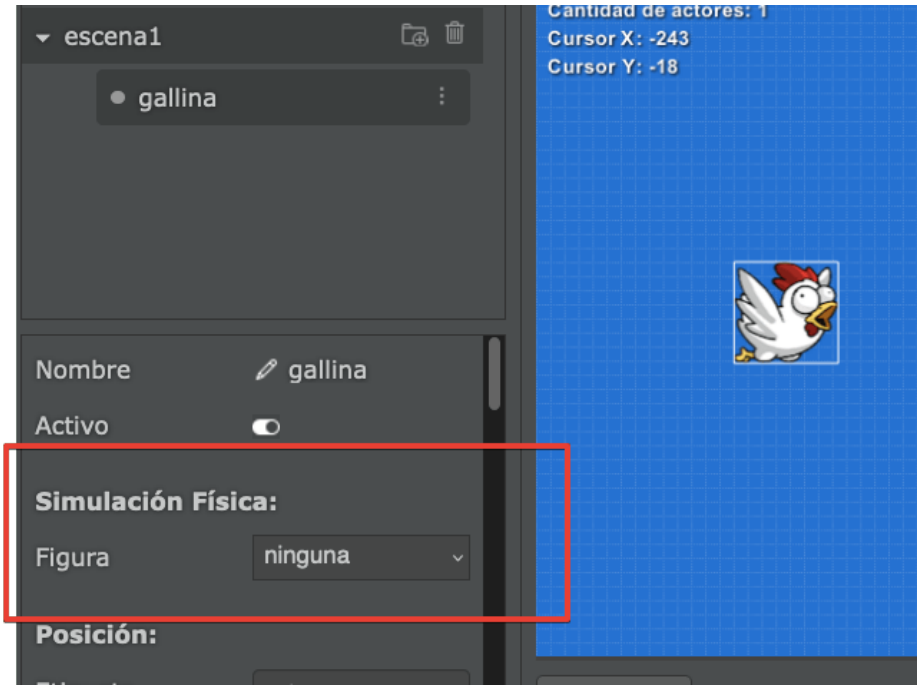
Pilas incluye un motor de física para hacer que los actores puedan colisionar, rebotar entre sí, ser lanzados y reaccionar la aceleración gravitatoria del escenario.

Este motor de física tiene muchas variantes, así que vamos a explorar todas esas oportunidades de configuración en esta sección.

Parámetros principales

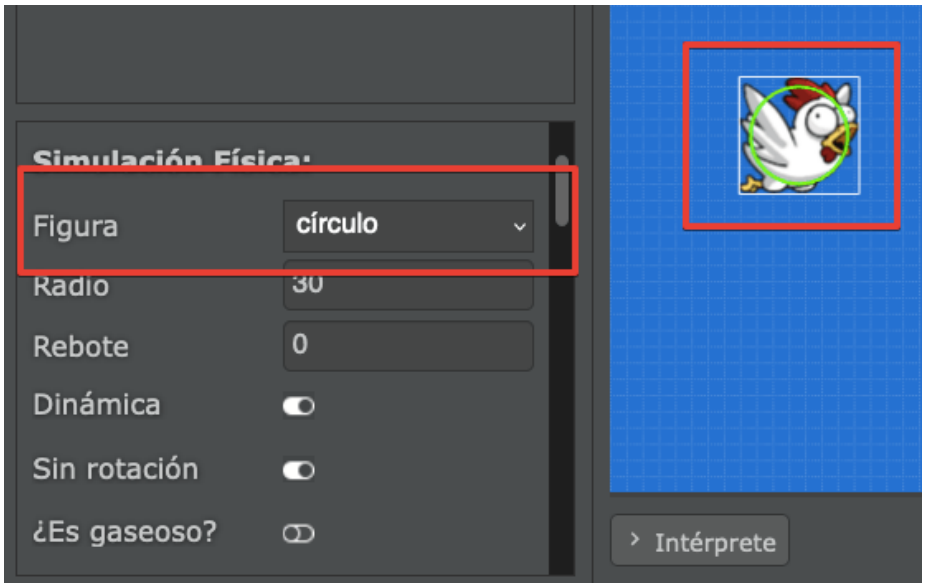
Para que un actor reaccione a las colisiones o se mueva como un objeto físico tenemos que activarle una figura desde las propiedades del actor.

En el panel de propiedades vas a ver una sección llamada "Simulación Física" y dentro de esa sección una propiedad llamada "Figura":

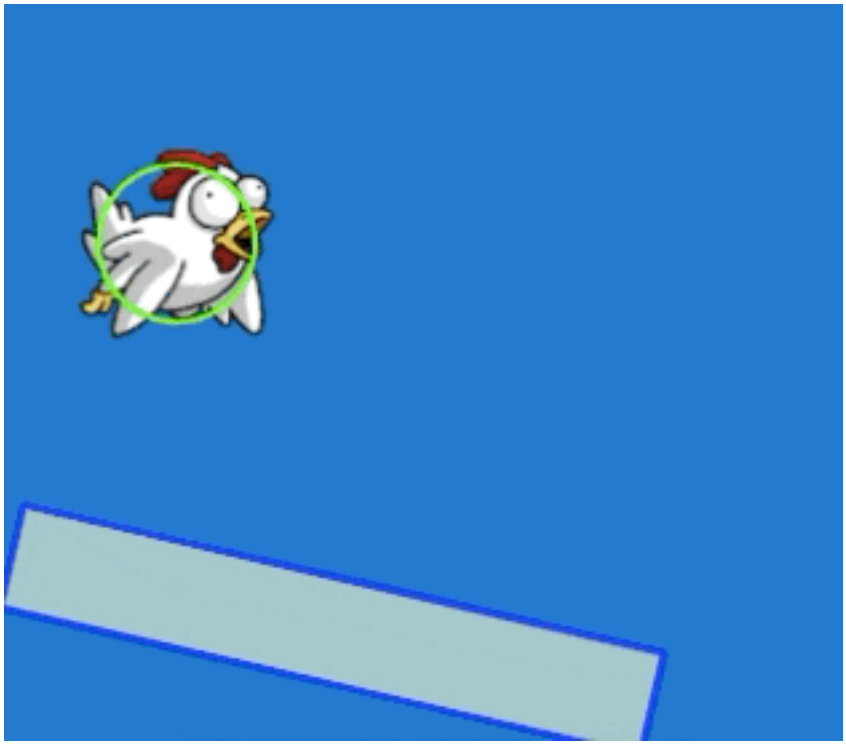


Si esa propiedad tiene el valor “ninguna”, el actor no podrá colisionar ni moverse usando el botón de física, para la computadora será solamente una imagen a mostrar en la pantalla.

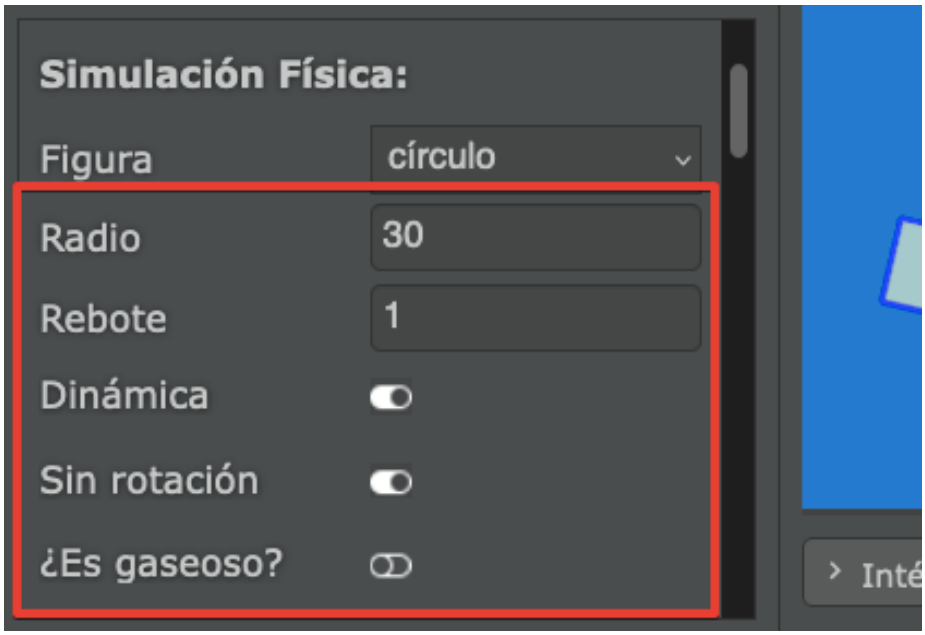
Si la propiedad tiene el valor “círculo” o “rectángulo” veremos que el actor tendrá una forma y tamaño en color verde:



Con esta propiedad, el actor va a reaccionar a la física del escenario, por ejemplo si colocas una plataforma y pulsas el botón ejecutar el actor comenzará a rebotar así:



También vas a notar que al elegir una figura aparecerán más propiedades como radio, rebote, dinámica etc. . . :



Propiedad Radio

La propiedad `radio` nos indica el tamaño de colisión si el actor tiene la figura círculo:

Es aconsejable hacer que el tamaño de esta figura coincida con la imagen del actor, porque la figura física será invisible para quienes jueguen a tu juego, los jugadores solo van a ver la imagen del actor.

Propiedad ancho y alto

Estas propiedades son similares a la propiedad `radio`, pero solo estarán disponibles si la figura es un `rectángulo`:

Pilas incluye estas figuras porque son las más simples y útiles de todas, si bien

Propiedad rebote

La propiedad `rebote` indicará de qué forma se tiene que impulsar el actor cuando entre en contacto con otro. Si el valor de rebote es muy bajo, por ejemplo `0`, el golpe va a hacer que el objeto se detenga por completo. Mientras que valores como `1.5` harán que el objeto se impulse con más fuerza.

Ten en cuenta que la reacción del actor dependerá del valor que tenga la propiedad `rebote` en ambos actores.

Propiedad Dinámica

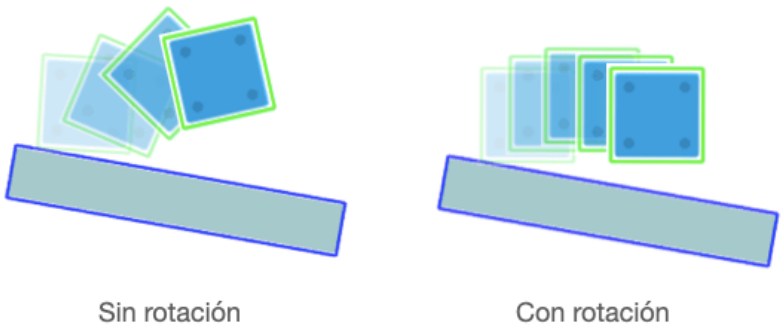
La propiedad `dinámica` puede tener los valores `si` y `no`. Una figura con `dinámica=si` rebotará y se moverá de acuerdo a la aceleración gravitatoria del escenario.

Si la figura tiene la propiedad `dinamica=no` el actor quedará rígido sin moverse ni reaccionar a la aceleración gravitatoria. Esto es ideal para hacer paredes,

plataformas u obstáculos del escenario que no queremos que se muevan por ningún motivo.

Propiedad Sin rotación

La propiedad `sin_rotacion` sirve para fijar la rotación de un actor como se muestra en la siguiente imagen:



Esta propiedad es ideal para personajes, ya que no queremos que ante una colisión aparezcan inclinados.

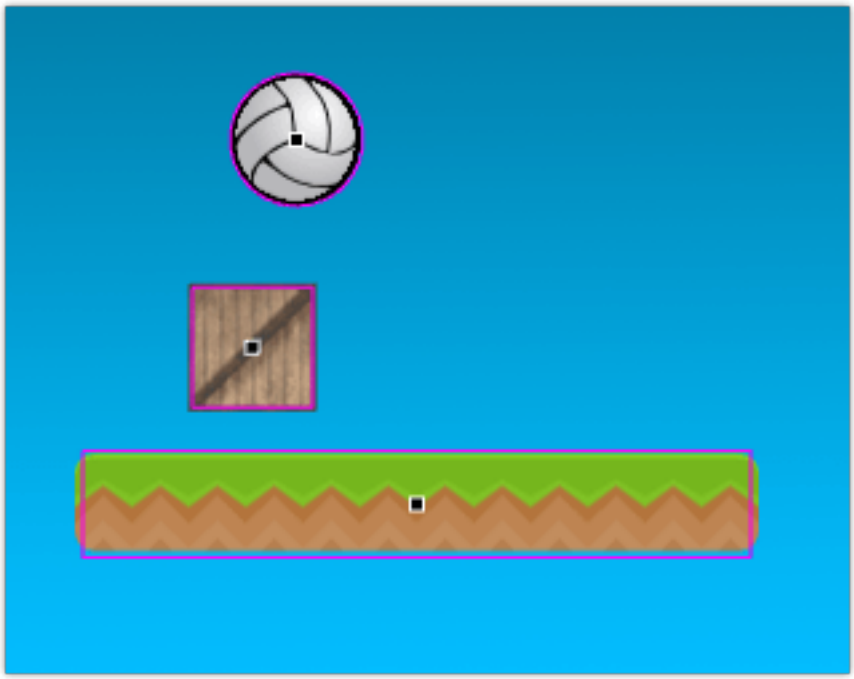
Propiedad gaseoso

La propiedad `¿Es gaseoso?` se utiliza para hacer que un actor se comporte como un fantasma, y que ninguna figura pueda chocarla o chocar a otros actores.

Colisiones

Las colisiones permiten ejecutar código como respuesta al contacto entre diferentes actores. Las funciones se pueden personalizar para hacer casi cualquier cosa: reproducir un sonido para magnificar el impacto, eliminar alguno de los actores en contacto, emitir efectos etc. . .

Por ejemplo, imagina que tenemos estos tres actores:



Cuando el juego se ejecute, la plataforma va a quedar fija en pantalla. Mientras que la pelota y la caja van a moverse hacia abajo y colisionarán.

Pilas va a llamar automáticamente a la función `cuando_comienza_una_colision` ni bien entren en contacto dos actores. Por ejemplo en este caso, pilas va a llamar a la función `cuando_comienza_una_colision` cuando la pelota colisione con la caja, y luego la plataforma.

```
class pelota extends Actor {

  cuando_comienza_una_colision(actor: Actor) {
    if (actor.etiqueta === "caja") {
      return true;
    }

    if (actor.etiqueta === "plataforma") {
      this.decir("Oh, colisioné con una plataforma!");
    }
  }
}
```

En el código hay dos cosas interesantes, tenemos la función `cuando_comienza_una_colision` dentro de la clase "pelota" para detectar colisiones y además intentamos distinguir contra qué actores se produce la colisión usando etiquetas. :

- Si la pelota colisiona con una caja, le indicamos a pilas que ignore la colisión, y continúe. Esto se hace simplemente retornando `true`.
- Si la pelota colisiona con una plataforma, emitimos un mensaje para que el usuario pueda reconocer que la pelota detectó la colisión.

Tipos de colisiones

Hay 3 instantes muy importantes cuando se producen colisiones:

- Cuando se detecta el contacto inicial.
- Cuando los dos actores permanecen en contacto prolongado. Por ejemplo cuando un actor se posa sobre una plataforma.
- El instante en donde la colisión desaparece porque los actores dejan de estar en contacto. Por ejemplo cuando un actor posando sobre una plataforma "salta" y deja de estar en contacto.

Para distinguir estos casos pilas llamará a las tres funciones de forma diferente. Este es un ejemplo de cómo se declaran esas funciones en el código de un actor:

```
class mi_actor extends Actor {
  cuando_comienza_una_colision(actor: Actor) {
    if (actor.etiqueta === "moneda") {
      this.pilas.reproducir_sonido("moneda");
      actor.eliminar();
    }
  }

  cuando_se_mantiene_una_colision(actor: Actor) {}

  cuando_termina_una_colision(actor: Actor) {}
}
```

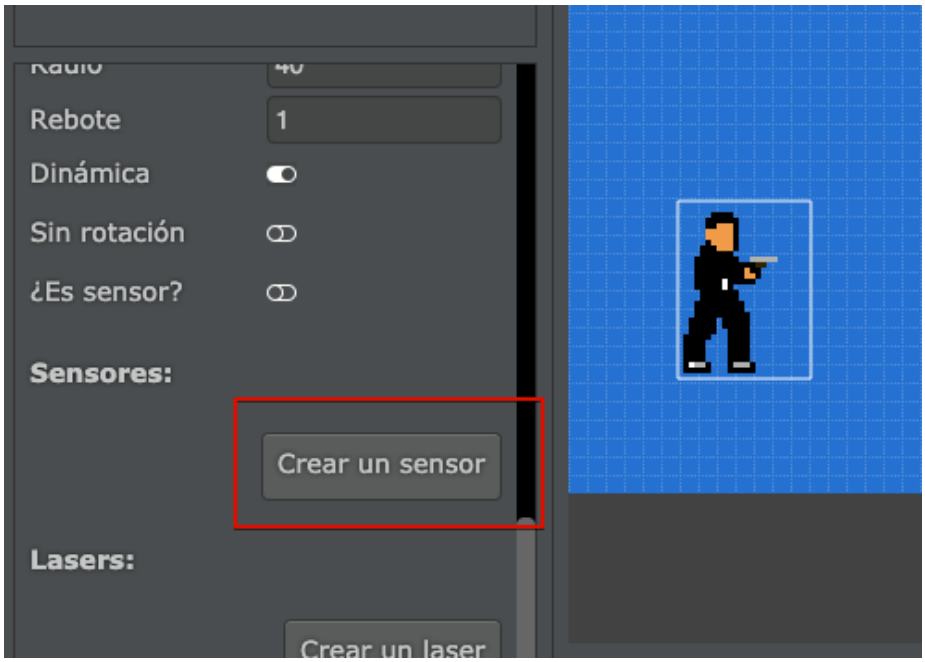
Sensores

Los sensores nos permiten detectar colisiones entre un actor y otros mediante areas rectangulares.

Se pueden utilizar en muchas situaciones diferentes, podrían ser útiles para detectar si una puerta se tiene que abrir ante la cercanía de un actor, detectar si un actor está pisando el suelo o saltando, hacer que un soldado detecte a un enemigo y varias cosas más.

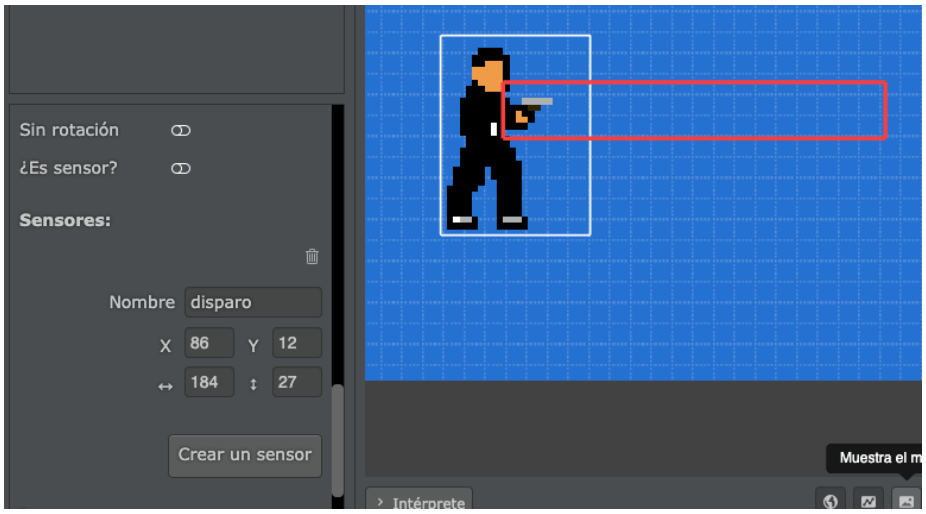
Crear sensores desde el editor

El primer paso para crear un sensor es tener seleccionado un actor y luego añadir el sensor al final de la lista de propiedades:



Una vez que pulsamos ese botón, vamos a poder cambiar las coordenadas del sensor y ajustarlo a nuestro gusto.

En este caso, hice que el actor tenga un sensor para detectar si se encuentra frente a algún actor:



Acceder a los sensores desde el código

Lo interesante de los sensores es cuando podemos interactuar con ellos desde el código, ya que desde ahí vamos a poder saber si el sensor detecta colisión con otros actores.

Tomemos como ejemplo el actor de la imagen anterior, el sensor que creamos se llama "disparo" así que para poder consultarle cosas necesitamos vincularlo a la clase del actor así:

```
2  class actor extends Actor {
3    disparo: Sensor;
4
5    iniciar() {
6      this.disparo = this.obtener_sensor("disparo");
7    }
```

A partir de ese momento, vamos a poder acceder a toda la información del sensor así:

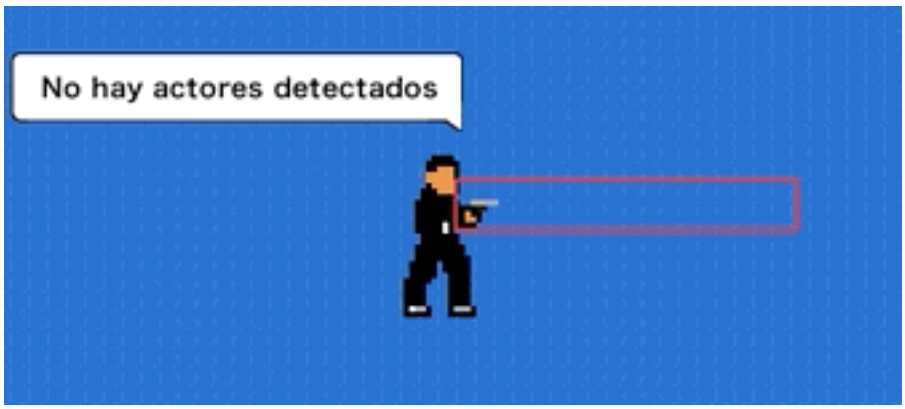
```
  actualizar() {
    this.disparo.
  }
}
```

```
cantidad_de_coli…  (property) Se…
cantidad_de_colisiones_con_la_e…
colisiona_con_etiqueta
colisiones
colisiones_con_la_etiqueta
```

Por ejemplo, si queremos hacer que el actor nos diga en todo momento si hay algún actor frente a su vista podemos escribir algo así:

```
 9    actualizar() {
10
11      if (this.disparo.cantidad_de_colisiones > 0) {
12        this.decir("Mi sensor detectó un actor");
13      } else {
14        this.decir("No hay actores detectados");
15      }
16
17    }
```

y como resultado, tendríamos que observar algo así:

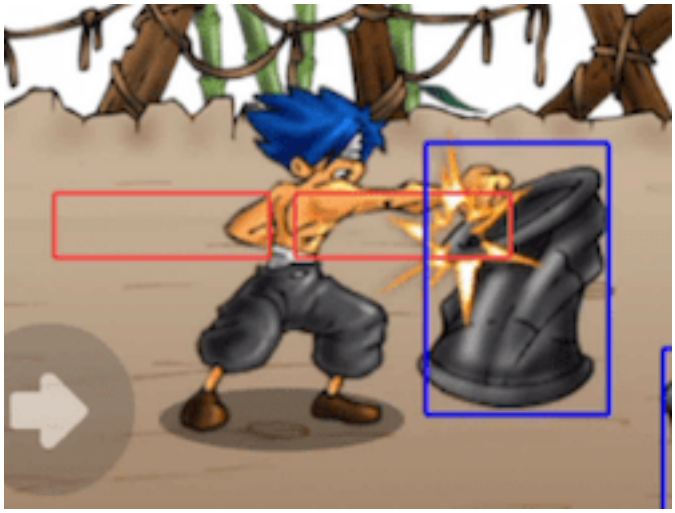


Es decir, cada vez que usemos el código `this.disparo.cantidad_de_colisiones` podremos obtener el número de actores colisionando con ese sensor.

Evaluando colisiones con etiquetas

Mediante código también se puede saber con qué actores entra en contacto un sensor. Se puede utilizar la función `colisiona_con_etiqueta(etiqueta)` que retorna `true` o `false` o incluso la función `colisiones_con_la_etiqueta(etiqueta)` que retornará una lista de todos los actores que colisionan con el sensor.

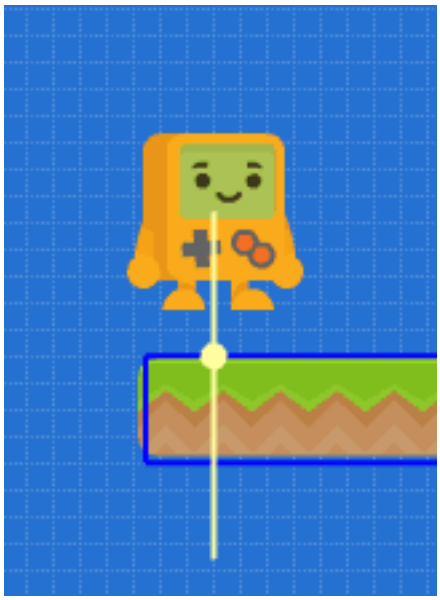
Si quieres ver un caso más completo de esto te recomendamos ver este minijuego dentro de la sección de ejemplos, en donde se usan sensores para detectar que actores se pueden golpear:



Lasers

Los lasers nos permiten detectar distancias y colisiones entre un actor y otros mediante rectas.

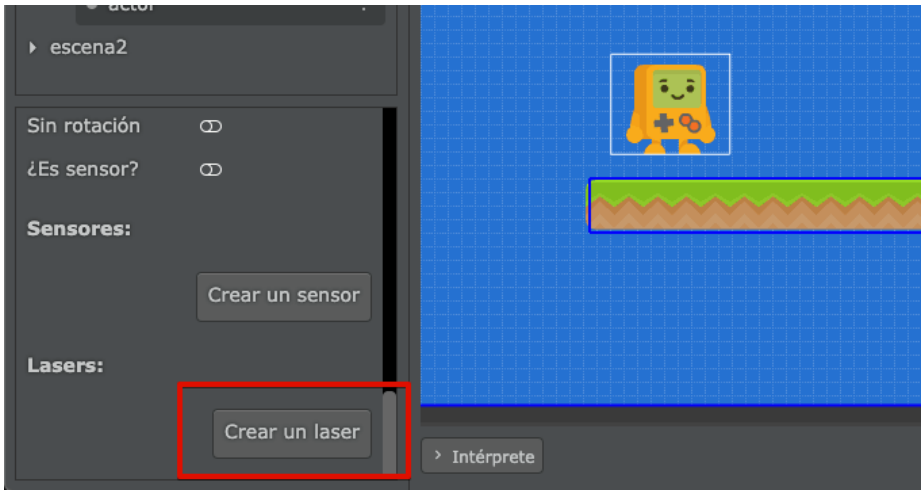
Por ejemplo, en la siguiente imagen tenemos un actor con un laser para detectar si existen plataformas debajo de sus pies:



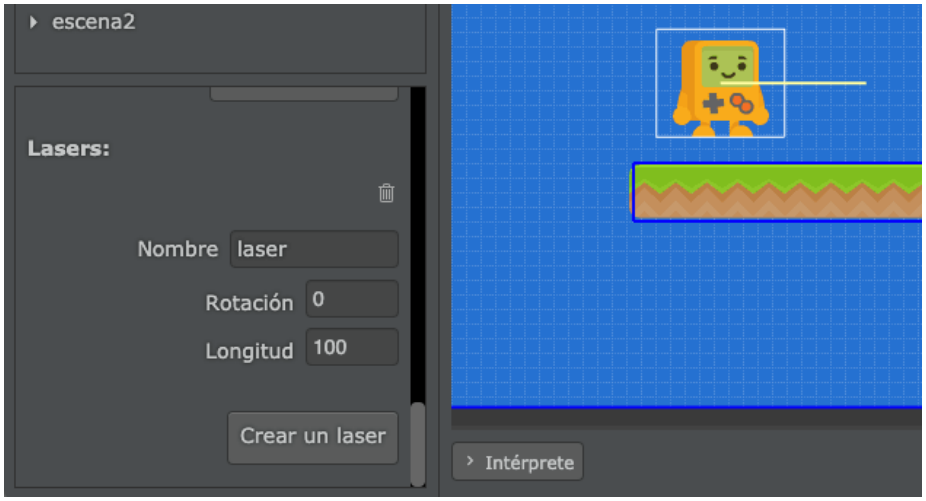
Cada actor puede tener tantos lasers como quiera, la idea de esta característica de pilas es que podamos permitirle a los actores detectar distancias y futuras colisiones para reaccionar a ellas.

Cómo crear lasers

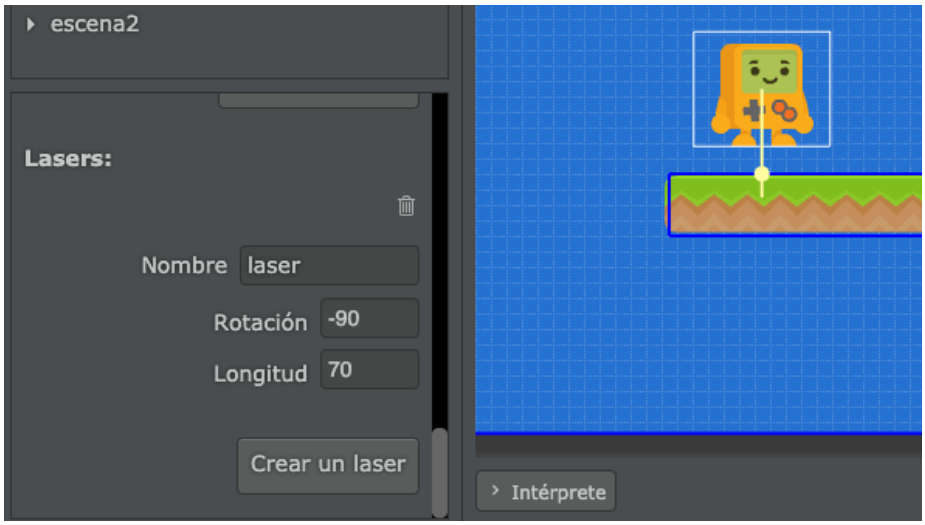
Los lasers se pueden crear directamente desde el editor, tenemos que seleccionar un actor y luego pulsar el botón “crear laser” dentro del panel de propiedades:



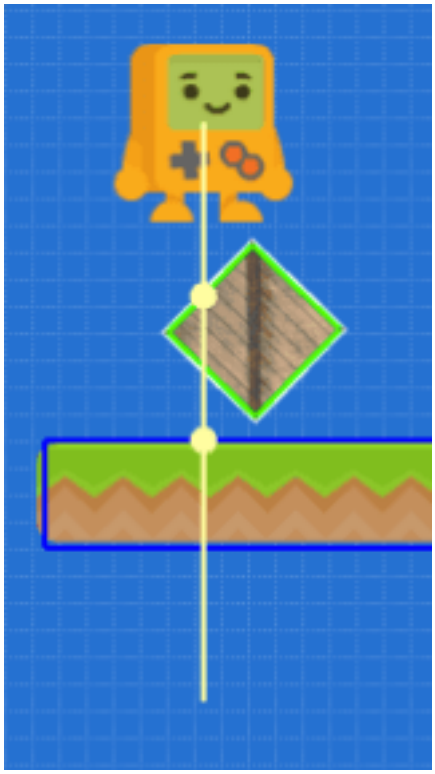
Una vez creado el laser, se tiene que ajustar la longitud y la rotación:



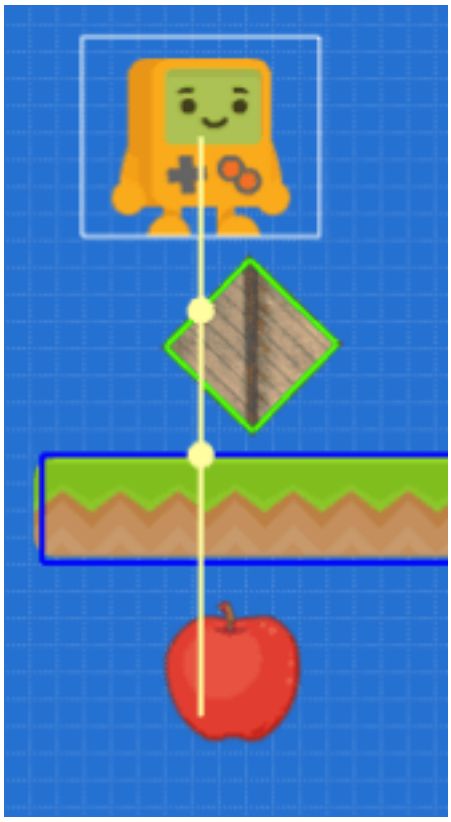
Por ejemplo, si nos interesa saber si el actor está cerca de tocar el suelo o no, podemos reducir la longitud y colocar la rotación en `-90 grados` para que el laser apunte hacia abajo:



Vas a notar que los lasers reaccionan rápidamente a las colisiones con otros actores dentro del editor. La linea del laser mostrará círculos en cada contacto que realice contra otros actores:



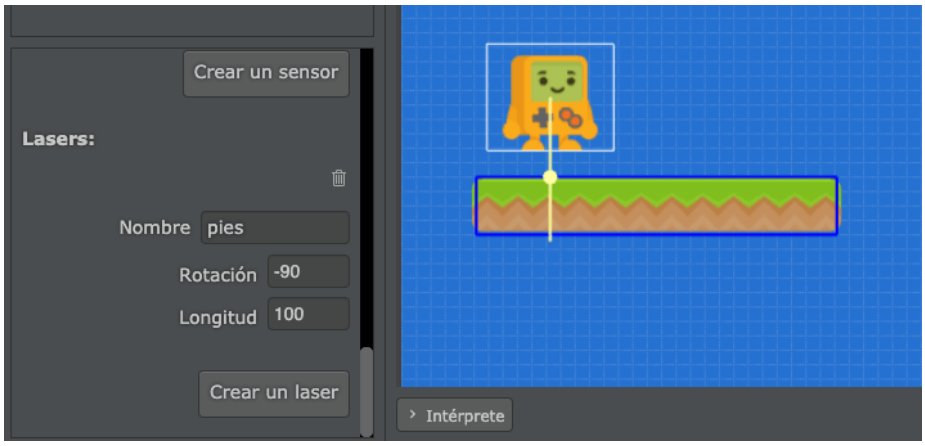
Eso sí, el laser solo detectará el contacto con aquellos actores que tengan una figura física asignada. Por ejemplo aquí se reconocen colisiones con una caja y la plataforma, pero no con la manzana (ya que no tiene figura física).



Accediendo a los lasers desde el código

Una vez que creamos el laser desde el editor, podemos escribir código para comenzar a interactuar con los lasers cuando el juego esté en funcionamiento.

Veamos esto con un ejemplo, queremos que nuestro personaje detecte si está sobre una plataforma o no. Así que podemos crear un laser llamado “pies” y colocarlo en dirección al suelo así:



Ahora podemos ir al código del actor, crear un atributo llamado `pies` y luego vincularle el laser llamando a la función `obtener_laser` así:

```
2  class actor extends Actor {
3    pies: Laser;
4
5    iniciar() {
6      this.pies = this.obtener_laser("pies");
7    }
```

Y con ese código, ahora vamos a poder hacerle consultas en cualquier momento al laser que vinculamos. Estas son algunas de las funciones que incluyen los lasers:

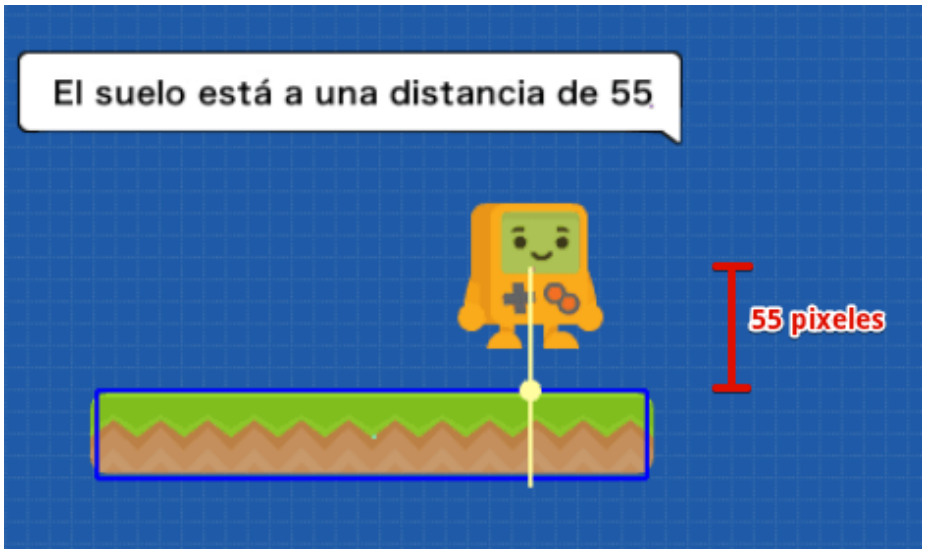
```
 9    actualizar() {
10      this.pies.
11                 actor                (property) L…
12                 colisiona_con_un_actor_de_etiq…
13                 distancia_al_actor_con_etiqueta
14                 distancia_al_actor_mas_cercado
15                 longitud
16                 nombre
17                 obtener_actor_mas_cercano
18                 obtener_colisiones
19                 rotacion
```

Tenemos funciones para consultar cuántos actores se encuentran en el camino del laser, cual es el más cercano, a qué distancia está cada actor y muchas cosas más.

Por ejemplo, podríamos hacer que el actor diga en todo momento a qué distancia está el suelo usando esta función dentro del método `actualizar`:

```
2  class actor extends Actor {
3    pies: Laser;
4
5    iniciar() {
6      this.pies = this.obtener_laser("pies");
7    }
8
9    actualizar() {
10     let distancia = this.pies.distancia_al_actor_mas_cercado();
11     this.decir("El suelo está a una distancia de " + distancia);
12   }
13 }
```

Y lo que deberíamos ver en la pantalla del juego, o ingresando en modo pausa, es que el laser nos informa que hay `55 pixeles` entre el actor (desde donde comienza el laser) hasta la primer colisión con la plataforma:



Ten en cuenta que la función `distancia_al_actor_mas_cercado` del laser, siempre nos retornará un número. Y en caso de que no exista ningún actor cercano, el número que nos retornará será igual a la longitud del laser.

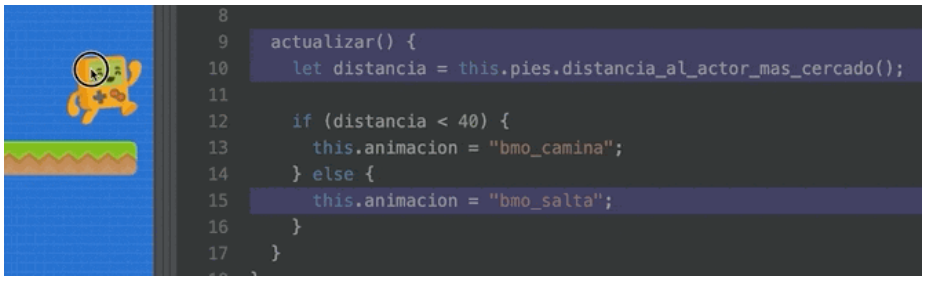
En este caso, como la longitud del laser es `100`, si no encuentra colisión con nada nos retornará ese valor:

Al principio parece raro que nos retorne la longitud del laser, pero de hecho es algo muy práctico, porque casi siempre las distancias se van a evaluar usando la palabra reservada `if` y con alguna comparación.

Observá este ejemplo, si queremos animar al actor de acuerdo a la distancia con la plataforma podemos escribir un código en el método `actualizar` como este:

```
actualizar() {
  let distancia = this.pies.distancia_al_actor_mas_cercado();

  if (distancia < 40) {
    this.animacion = "bmo_camina";
  } else {
    this.animacion = "bmo_salta";
  }
}
```

y el resultado va a ser similar a este:



Es decir, gracias a que el laser nos retorna la distancia como un número, podemos

decirle al actor que si la distancia es menor a `40` queremos que muestre la animación "correr" y si esa distancia es mayor a `40` queremos que muestre la animación de "salto".

Avanzado: Lasers instantáneos desde el código

Los lasers que creamos desde el editor son ideales para visualizar, configurar y comprender fácilmente de qué se trata este concepto de distancia y colisiones en linea recta.

Sin embargo, hay casos muy puntales en donde nos interesa calcular distancias o colisiones pero directamente desde el código. Para estos casos pilas incorpora lasers "instantáneos".

Los lasers "instantáneos" se pueden crear con la función `laser`, que espera la referencia a un actor, un punto inicial y uno final. Y como resultado nos retornará una lista con todas las colisiones que se produzcan en el camino del laser:

```
let resultado = this.pilas.laser(this, this.x, this.y, this.x + 400, this.y);

this.pilas.observar("resultado", resultado);
this.pilas.observar("cantidad de colisiones", resultado.length);
```

También hay otra función muy similar llamada `laser_al_primer_actor` que además acepta una etiqueta, para que podamos detectar distancias contra un tipo de actor en especial.

Estas dos funciones son bastante avanzadas, si queres saber más acerca de ellas te recomendamos ver el glosario de funciones.

Mensajes entre actores y escenas

Dentro de un juego se puede producir varias comunicaciones entre actores. Si un personaje protagonista pierde, tal vez queremos que el contador de puntajes se entere y cambie de color, o que los enemigos conjuntamente dejen de seguir al protagonista; tal vez incluso vamos a querer que la escena se notifique de esto y ponga un cartel en pantalla o pase a otra escena.

Los mensajes sirven para implementar este tipo de comunicaciones entre distintos objetos de nuestro juego. Y tienen una ventaja, porque podemos hacer que un actor no tenga que conocer todo lo que lo rodea, sino que simplemente emita un mensaje "y quién escuche puede hacer algo al respecto".

Enviar mensajes globalmente

Un ejemplo típico que se suele dar de mensajes son los mensajes globales, estos mensajes se pueden enviar desde un actor a todos los demás, estos mensajes globales incluso llegan a la escenas.

Para enviar un mensaje global tenemos que escribir este código en el actor emisor:

```
this.pilas.enviar_mensaje_global("mi_mensaje");
```

Y para que otros actores puedan atender este mensaje simplemente tenemos que crear un método que se llame "cuando*llega_el_mensaje*" seguido del nombre del mensaje así:

```
cuando_llega_el_mensaje_mi_mensaje() {
  this.decir("Algún actor envió el mensaje 'mi_mensaje'");
}
```

opcionalmente, si queremos un actor o escena que capture todos los mensajes podemos usar este método:

```
cuando_llega_un_mensaje(mensaje:string) {
  this.decir("llegó el mensaje " + mensaje);
```

```
}
```

Mensajes dirigidos a actores particulares

Si no queremos enviar el mensajes a todos los actores y escenas, lo que podemos hacer es obtener la referencia a un actor en particular y enviarle el mensaje de forma directa:

```
let puntaje = this.pilas.obtener_actor_por_nombre("puntaje");
puntaje.enviar_mensaje("aumentar");
```

Y de forma similar a como mencionamos antes, el puntaje debería tener un método llamado `cuando_llega_el_mensaje_aumentar` para reaccionar al mensaje:

```
cuando_llega_el_mensaje_aumentar() {
  this.puntaje_acumulado += 1;
}
```

Mensajes con parámetros

A veces junto con el mensaje necesitamos enviar parámetros para ser más específicos. Para eso podemos usar el segundo argumento de las funciones de mensajes.

Por ejemplo, imagina que tenemos un actor que cuando colisiona con una moneda sume 5 puntos, pero cuando colisiona con un diamante sume 10 puntos.

Por un lado, cuando se produce la colisión podemos emitir el mensaje "aumentar_puntaje" con la cantidad de puntos que le corresponden:

```
cuando_colisiona(actor: Actor) {
  if (actor.tiene_etiqueta("moneda")) {
    actor.eliminar();
    this.pilas.enviar_mensaje_global("aumentar_puntaje", {cantidad: 5});
  }

  if (actor.tiene_etiqueta("diamante")) {
    actor.eliminar();
    this.pilas.enviar_mensaje_global("aumentar_puntaje", {cantidad: 10});
  }
}
```

Luego, del lado del receptor (o los receptores) del mensaje, tenemos que leer los parámetros adicionales así:

```
cuando_llega_el_mensaje_aumentar_puntaje(datos) {
  this.puntaje_acumulado += datos.cantidad;
}
```

Código de proyecto

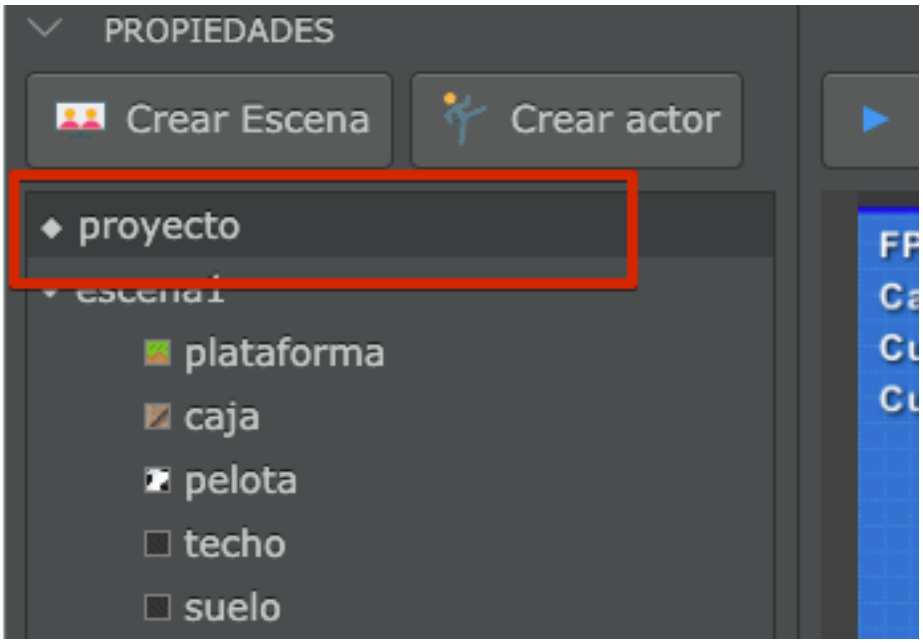
En un proyecto de pilas hay varios elementos que pueden contener su propio código: Tanto las escenas como los actores tienen su propio código asociado.

Ahora bien, en alguna situaciones necesitamos poder guardar variables en algún lugar común que sobreviva más allá de los actores y la escena actual. Por ejemplo si queremos almacenar la cantidad de vidas disponibles que tiene un actor cuando pasa de una escena a otra.

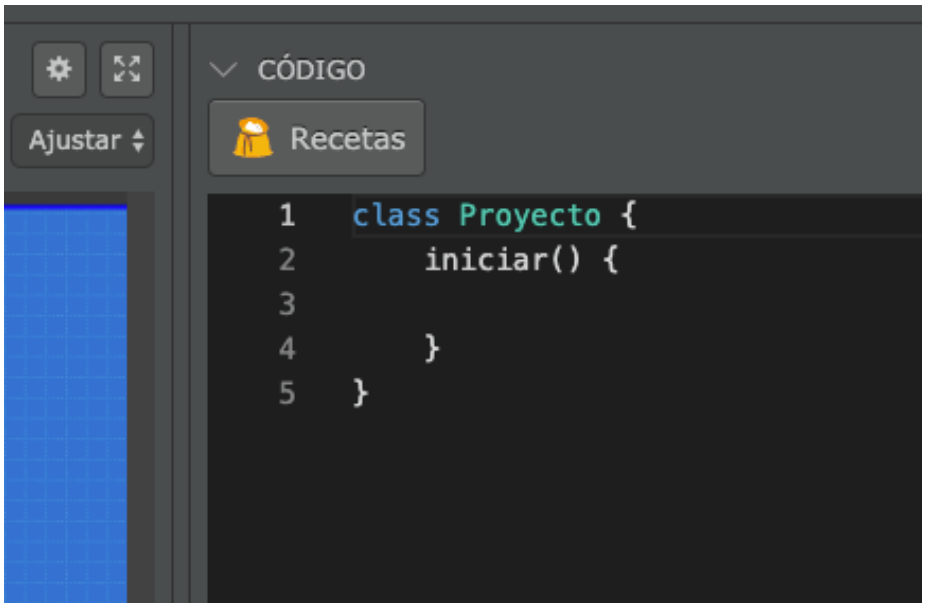
Un ejemplo breve

Para ejemplificar cómo podemos usar el código de la escena voy a dar un ejemplo. Imagina que tenemos un juego en el que tenemos que recolectar monedas con el mouse, pero queremos que la cantidad de monedas persista entre un nivel y otro.

Primero tenemos que seleccionar el proyecto desde el panel de propiedades:



Luego, vamos a ver que en el panel de código aparece el código del proyecto:



El código está casi vacío, así que vamos a crear el contador de monedas recolectadas y vamos a darle un valor inicial:

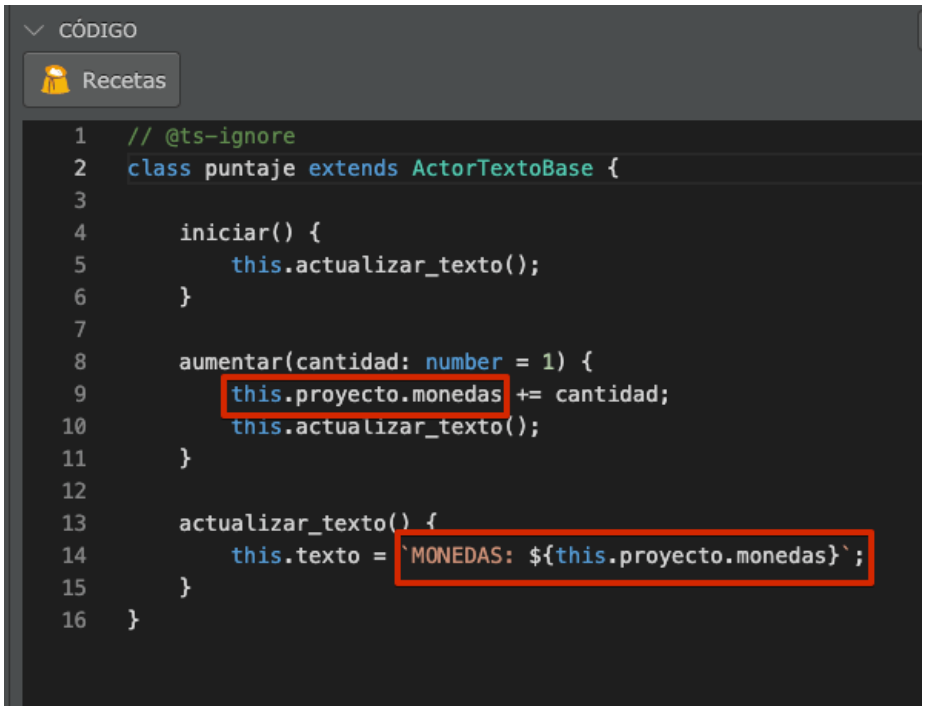
```
class Proyecto {
  monedas: number;
```

```
  iniciar() {
    this.monedas = 0;
  }
}
```

Ahora vamos a crear un actor para "visualizar" ese contador directamente en pantalla.

Pulsá el botón "Crear actor" y luego seleccioná al actor "puntaje":

Ahora editemos el código para que este actor pueda acceder a la variable puntaje del proyecto. Tenemos que cambiar estas lineas de código:



Observá que desde un actor podemos acceder directamente al proyecto usando el código `this.proyecto`, y por supuesto vamos a poder acceder a todas las variables o métodos del proyecto directamente.

El código completo debería quedarte así:

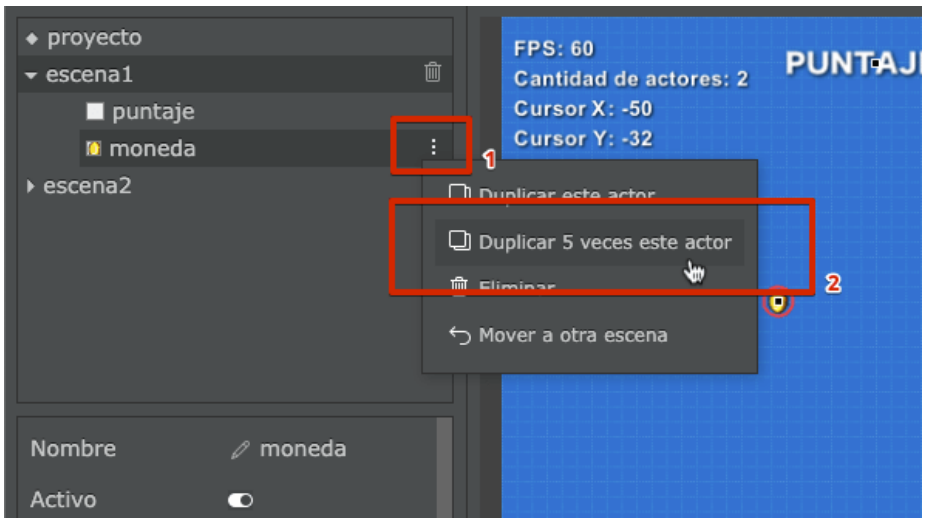
```
// @ts-ignore
class puntaje extends ActorTextoBase {
  iniciar() {
    this.actualizar_texto();
  }

  aumentar(cantidad: number = 1) {
    this.proyecto.monedas += cantidad;
    this.actualizar_texto();
  }

  actualizar_texto() {
    this.texto = `MONEDAS: ${this.proyecto.monedas}`;
  }
}
```

y una vez que ejecutamos el juego debería verse así:

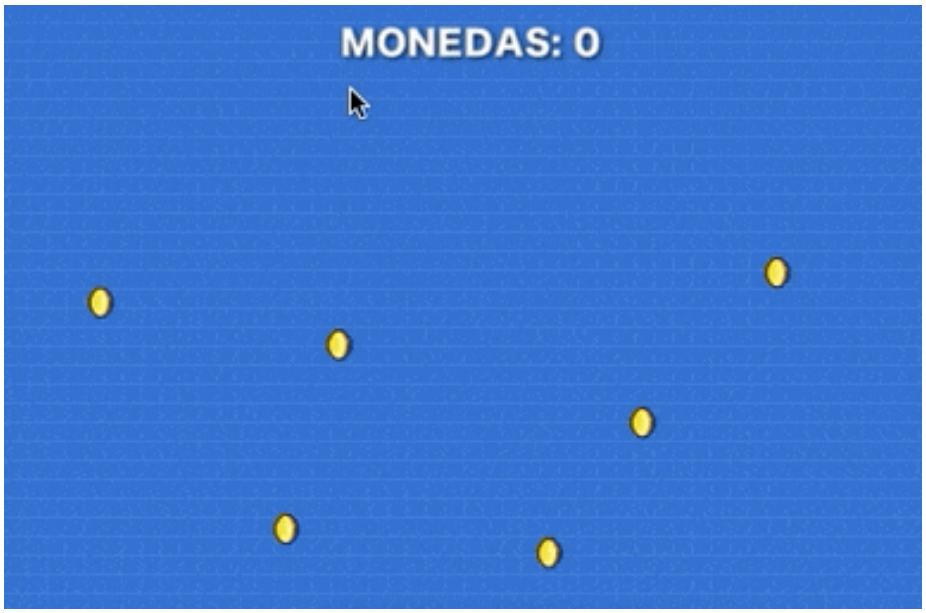
Ahora creá un actor de tipo moneda, con este código:

```
// @ts-ignore
class moneda extends Actor {
  cuando_hace_click() {
    this.proyecto.monedas += 1;
    this.eliminar();
  }
}
```

y luego duplica ese actor unas 5 veces desde el menú:



Y ya casi lo tenemos listo, si ejecutas el proyecto vas a ver cómo el contador de monedas almacena y muestra la variable `monedas` directamente desde el proyecto:

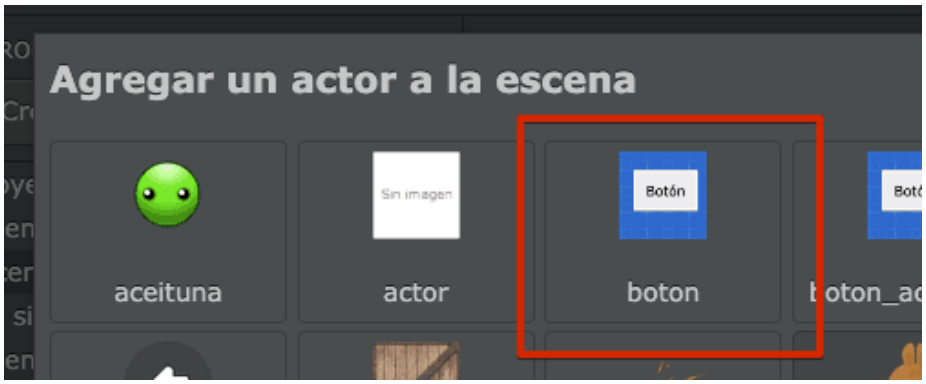


Ahora bien, lo más interesante de almacenar variables en el proyecto es que conservarán su valor incluso si cambiamos de una escena a otra.

Hagamos la prueba: pulsa el botón "crear escena" y en el código de la escena coloca este código para que el usuario pueda ver el puntaje cuando ingresa a esta escena:

```
class escena2 extends Escena {
  iniciar() {
    this.pilas.clonar("puntaje");
  }
}
```

y luego, en la escena principal, pulsa el botón "crear actor" para crear un actor de tipo "botón":



y por último, cambia el código de este actor para que nos permita pasar a la siguiente escena cuando se pulsa:

```
// @ts-ignore
class boton extends ActorTextoBase {
  cuando_hace_click() {
    this.pilas.cambiar_escena("escena2");
  }
}
```

Si bien este ejemplo es muy simple, es un punto de partida para poder mantener los datos del jugador durante toda su partida.

Te recomiendo mirar el proyecto "mantener-valores-entre-escenas" de la sección "ejemplos" de pilas para que veas cómo se puede mejorar este mismo proyecto que realizamos acá.

Eventos del mouse

En pilas llamamos "eventos" a las señales que emiten desde dispositivos como el mouse o teclado. Por ejemplo, un "click" del mouse es un evento al igual que la pulsación de una tecla.

Y lo interesante de los eventos es que podemos capturarlos y disparar alguna acción dentro del juego para responder. Por ejemplo, en un juego de acción, el "click" del mouse podría realizar una explosión o hacer que un personaje salte.

Antes de empezar, el caso más común

Si bien esta sección habla de eventos y cómo hacer uso por completo del mouse, casi siempre se quiere saber la posición del mouse para mover actores o crear objetos. Si ese es tu caso, te comentamos que pilas tiene 4 atributos para conocer en qué posición se encuentra el cursor del mouse en todo momento:

- pilas.cursor_x: posición horizontal del mouse.
- pilas.cursor_y: posición vertical del mouse.
- pilas.cursor_x_absoluta: posición horizontal del mouse pero sin tener en cuenta la posición de la cámara. La coordenada corresponderá a la coordenada física de la ventana.
- pilas.cursor_y_absoluta: posición vertical del mouse pero sin tener en cuenta la posición de la cámara.

Y estos atributos, se puede usar generalmente directamente desde la función actualizar de los actores:

```
class actor extends Actor {
  iniciar() {}

  actualizar() {
    // la sentencias a continuación hacen que el actor siga la
    // posición del cursor del mouse en todo momento
    this.x = this.pilas.cursor_x;
    this.y = this.pilas.cursor_y;
```

```
  }
}
```

¿Cómo capturar un evento del mouse?

Para capturar un evento desde el mouse simplemente hay que declarar alguna de estas funciones en el código:

- cuando_hace_click(x, y, evento_original)
- cuando_termina_de_hacer_click(x, y, evento_original)
- cuando_mueve(x, y, evento_original)
- cuando_sale(x, y, evento_original)

Estas funciones se pueden crear en el código de una escena o de un actor. La diferencia es que en las escenas el "click" o el movimiento se van a detectar en toda la pantalla, mientras que en el código del actor solo se detectarán si el mouse apunta al actor.

Si desde un actor necesitas detectar estos eventos del mouse pero sobre toda la pantalla deberías usar alguno de estos otros métodos:

- cuando_hace_click_en_la_pantalla(x, y, evento_original)
- cuando_mueve_sobre_la_pantalla(x, y, evento_original)
- cuando_termina_de_hacer_click_en_la_pantalla(x, y, evento_original)

Veamos un ejemplo, imaginá que queremos crear actores de la clase "Pelota" cada vez que el usuario hace "click" sobre la pantalla. Podríamos hacerlo colocando este código en la escena:

```
class escena2 extends Escena {
  iniciar() {}

  actualizar() {}

  cuando_hace_click(x, y, evento_original) {
    let pelota = this.pilas.actores.pelota();
    pelota.x = x;
    pelota.y = y;
  }
}
```

¿Cómo desactivar clicks?

Si quieres que un actor deje de responder a clicks, no hace falta eliminar la función `cuando_hace_click` sino que puedes llamar a la siguiente función:

```
this.desactivar_clicks();
```

Cuando a un actor se le pide que desactive sus clicks, los eventos `click` pasarán a actor que esté detrás de él. No se llamará a la función `cuando_hace_click` ni se podrá arrastrar y soltar a ese actor.

Luego, si quieres volver a activar los clicks deberías llamar a esta función:

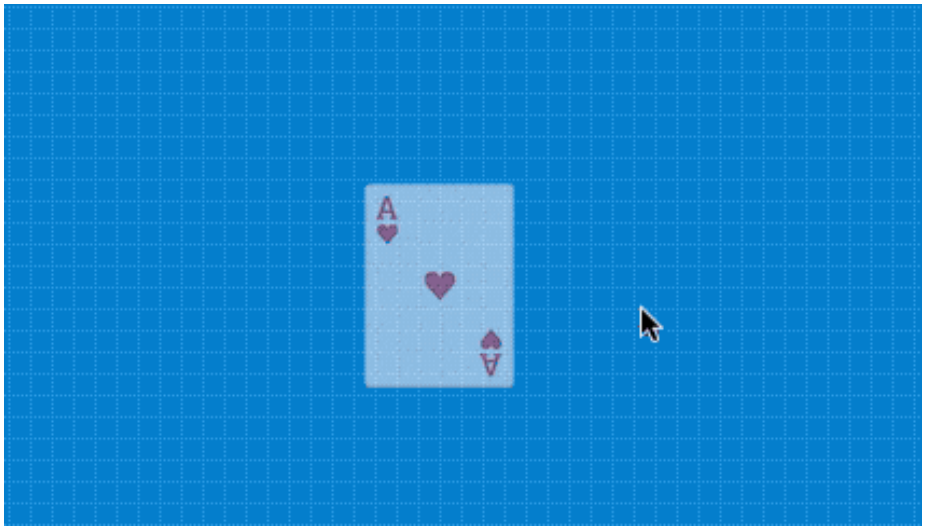
```
this.activar_clicks();
```

Hay un ejemplo llamado `desactivar_clicks` en donde se muestra cómo usar esta función.

¿Cómo capturar eventos del mouse en un actor?

Para capturar eventos en el contexto de un actor, tenemos que usar las mismas funciones, pero declarándolas dentro del código del actor.

Por ejemplo, imaginá que estamos haciendo un juego de cartas y queremos que la carta se pueda hacer girar con un "click" de mouse y que cambie de transparencia cuando el mouse esté sobre ella:



El código del actor debería quedarnos algo similar a lo siguiente:

```
class actor extends Actor {
  propiedades = {};

  iniciar() {
    this.transparencia = 50;
  }

  actualizar() {}
```

```
  cuando_mueve(x, y, evento_original) {
    this.escala = 1.2;
    this.transparencia = 0;
  }

  cuando_sale(x, y, evento_original) {
    this.transparencia = 50;
    this.escala = 1;
  }

  cuando_hace_click(x, y, evento_original) {
    this.rotacion = [360];
  }

  cuando_termina_de_hacer_click(x, y, evento_original) {}
}
```

Los manejadores de eventos como `cuando_mueve`, `cuando_sale` y `cuando_hace_click` van a ser llamados internamente cuando se produzcan esos eventos sobre el actor.

Capturar eventos de forma global, con manejadores

Otra forma de capturar eventos más sofisticada es utilizando el módulo `eventos` de la escena.

```
pilas.eventos.conectar("mueve_mouse", (x, y) => {
  console.log("mueve", x, y);
});

pilas.eventos.conectar("click_de_mouse", (x, y) => {
  console.log("mueve", x, y);
});

pilas.eventos.conectar("termina_click", (x, y) => {
  console.log("mueve", x, y);
});
```

Esta es la lista de eventos completa:

- `mueve_mouse`: x, y
- `click_de_mouse`: x, y, boton_izquierdo, boton_derecho
- `termina_click`: x, y, boton_izquierdo, boton_derecho

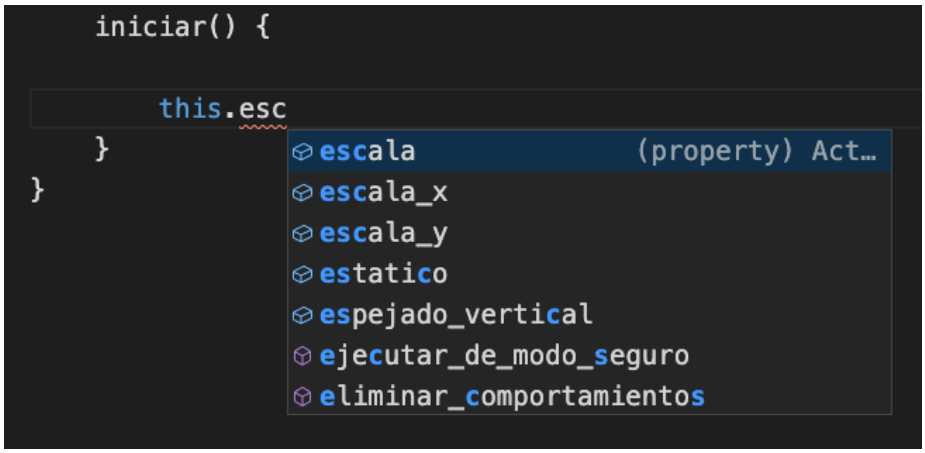
Consejos para usar el editor de código

El editor de código tiene varias funcionalidades para ayudarte a escribir código de manera rápida y cómoda.

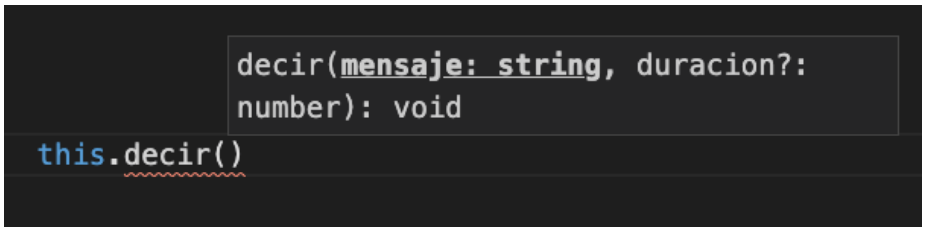
En esta sección te vamos a resumir algunas de esas funcionalidades:

Autocompletado

Siempre que escribas código, el editor intentará anticiparse a lo que escribas para mostrar sugerencias:



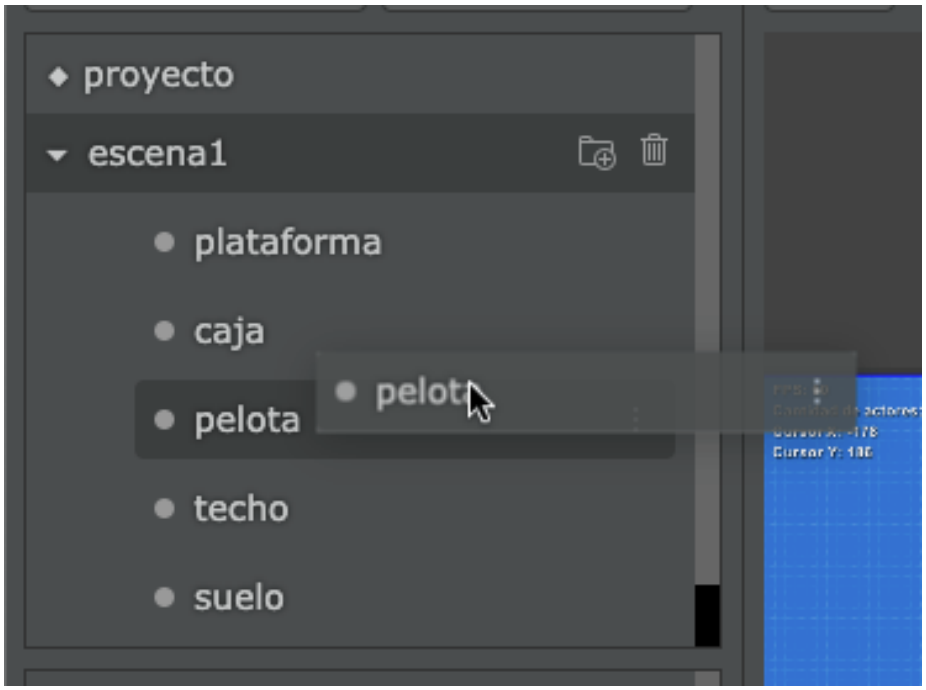
Además, si estás escribiendo la llamada a un método el editor te mostrará los argumentos que podrías incluir:



En este caso, el método decir admite dos argumentos: un mensaje y una duración. Notá que la duración tiene un símbolo de pregunta, lo que indica que ese argumento es opcional.

Obtener referencias a actores

Para obtener referencias rápidas a los actores puedes simplemente "arrastrar y soltar" los actores sobre el código del editor:



Cuando el editor detecte que arrojaste un actor sobre el editor va a escribir una linea de código como esta:

```
let pelota = this.pilas.obtener_actor_por_nombre("pelota");
```

Lo que es bastante útil, porque nos evita escribir ese código.

Y si el actor que arrastramos sobre el editor es un actor desactivado, el editor va

a colocar este código en su lugar:

```
let pelota = this.pilas.clonar("pelota");
```

Fragmentos rápidos de código

Además del autocompletado el editor nos permite escribir fragmentos rápidos, por ejemplo si escribis "observar" y pulsas enter, el editor va a autocompletar este texto:

```
this.pilas.observar("x", this.x);
```

Además va a situar el cursor en la posición para que podamos personalizar esta linea de código rápidamente.

Otros de los fragmentos rápidos que incluye el editor son los siguientes:

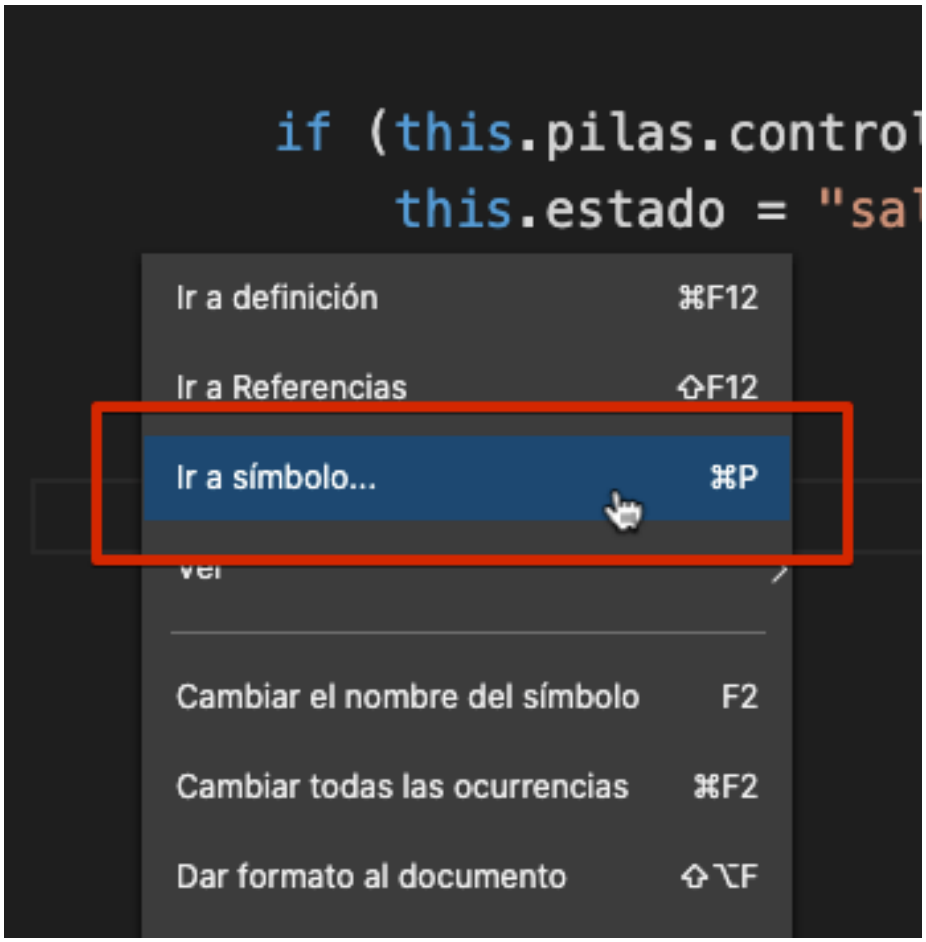
- `pilas`
- `clonar`
- `clonar_en`
- `control`
- `animar`

Te recomendamos probar escribir alguno de estos atajos para familiarizarte, sobre todo el fragmento `control` que es bastante completo.

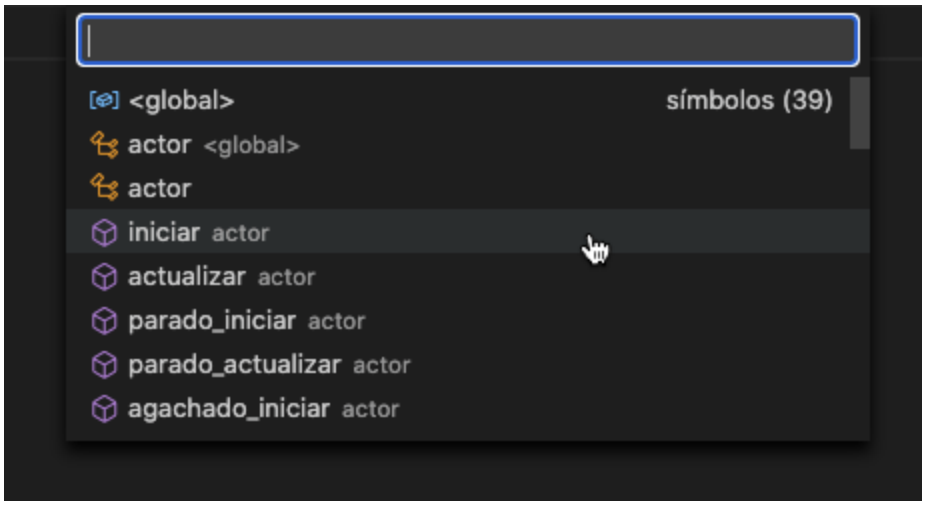
Navegar métodos rápidamente

Cuando tenemos un actor con muchos métodos es aconsejable hacer búsquedas en lugar de recorrer el código hacia arriba y abajo con la barra de scroll.

Para buscar métodos por nombres se puede pulsar el botón derecho del mouse sobre el editor y seleccionar la siguiente opción:



Luego vas a notar que aparecerá una barra de búsqueda con todos los métodos del actor, simplemente escribí o selecciona el método al que quieras ingresar.



Ten en cuenta que también puedes activar esta funcionalidad pulsando `ctrl+P` como atajo.

Uso del teclado

El teclado se puede utilizar de dos formas básicas: por consulta para mover personajes o mediante eventos para responder a pulsaciones de teclas específicas:

Primer forma: ¿Cómo mover un personaje con el teclado?

En la mayoría de los juegos para computadoras se puede controlar al protagonista del juego usando el teclado.

Pilas hace bastante sencillo esto mediante un objeto llamado "control" que nos va a permitir acceder a las teclas básicas para controlar personajes.

Por ejemplo, para mover un actor hacia izquierda y derecha usando las flechitas del teclado podemos editar el método "actualizar" del actor y colocar estas lineas de código:

```
actualizar() {
  if (this.control.izquierda) {
    this.x -= 5;
  }

  if (this.control.derecha) {
    this.x += 5;
  }
}
```

El objeto "control" tiene varios atributos que podemos consultar (mediante la palabra "if") para saber si en ese instante la tecla está pulsada o no.

Estas son otros atributos que tiene el objeto "control":

Tecla | Código para consultar la tecla | nombre |
- | | – |
Flecha izquierda | this.control.izquierda | "izquierda" |

Flecha derecha | this.control.derecha | “derecha” |
Flecha arriba | this.control.arriba | “arriba” |
Flecha abajo | this.control.abajo | “abajo” |
Tecla Space | this.control.espacio | “espacio” |
Alt | this.control.alt | “alt” |
Control | this.control.control | “control” |
Shift | this.control.shift | “shift” |
Tab | this.control.tab | “tab” |
Backspace | this.control.backspace | “backspace” |
Meta | this.control.meta | “meta” |
Escape | this.control.escape | “escape” |
Enter | this.control.enter | “enter” |
Tecla a | this.control.tecla_a | “a” |
Tecla b | this.control.tecla_b | “b” |
Tecla c | this.control.tecla_c | “c” |
Tecla d | this.control.tecla_d | “d” |
Tecla e | this.control.tecla_e | “e” |
Tecla f | this.control.tecla_f | “f” |
Tecla g | this.control.tecla_g | “g” |
Tecla h | this.control.tecla_h | “h” |
Tecla i | this.control.tecla_i | “i” |
Tecla j | this.control.tecla_j | “j” |
Tecla k | this.control.tecla_k | “k” |
Tecla l | this.control.tecla_l | “l” |
Tecla m | this.control.tecla_m | “m” |
Tecla n | this.control.tecla_n | “n” |
Tecla ñ | this.control.tecla_ñ | “ñ” |
Tecla o | this.control.tecla_o | “o” |
Tecla p | this.control.tecla_p | “p” |
Tecla q | this.control.tecla_q | “q” |
Tecla r | this.control.tecla_r | “r” |
Tecla s | this.control.tecla_s | “s” |
Tecla t | this.control.tecla_t | “t” |
Tecla u | this.control.tecla_u | “u” |
Tecla v | this.control.tecla_v | “v” |
Tecla w | this.control.tecla_w | “w” |
Tecla x | this.control.tecla_x | “x” |
Tecla y | this.control.tecla_y | “y” |
Tecla z | this.control.tecla_z | “z” |
Tecla 1 | this.control.tecla_1 | “1” |
Tecla 2 | this.control.tecla_2 | “2” |
Tecla 3 | this.control.tecla_3 | “3” |
Tecla 4 | this.control.tecla_4 | “4” |
Tecla 5 | this.control.tecla_5 | “5” |
Tecla 6 | this.control.tecla_6 | “6” |
Tecla 7 | this.control.tecla_7 | “7” |

Tecla 8 | this.control.tecla_8 | “8” |
Tecla 9 | this.control.tecla_9 | “9” |

Segunda forma: ¿Cómo reaccionar a eventos del teclado?

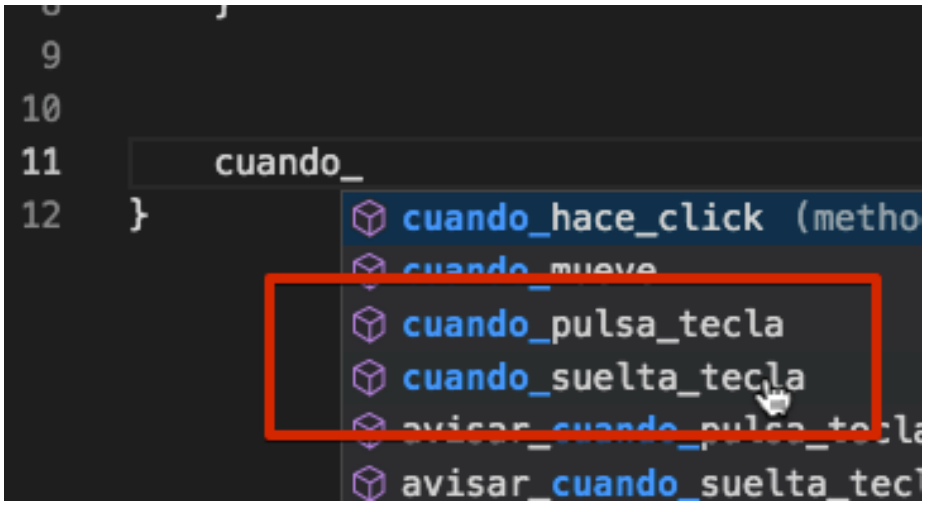
Otra forma de acceder al teclado es mediante eventos, para ejecutar una función justo cuando se pulsa una tecla o se suelta.

Esto es útil por ejemplo en los juegos por turnos o cuando queremos que el teclado funcione sin repetición, imaginá que si hacemos un juego donde el jugador pueda disparar, queremos que pulse y suelte la tecla de disparo muchas veces, no que la deje pulsada y eso dispare automáticamente.

Para capturar eventos de teclado de esta forma tendrías que definir los métodos `cuando_pulsa_tecla` y `cuando_suelta_tecla` así:

```
cuando_pulsa_tecla(tecla) {
  if (tecla == "espacio") {
    this.disparar();
  }
}

cuando_suelta_tecla(tecla) {
  if (tecla == "1") {
    this.decir("Has soltado la tecla 1")
  }
}
```

Estos métodos están disponibles tanto en las escenas como en los actores. Puedes comprobarlo escribiendo “cuando” en el editor y viendo cómo se sugieren las opciones:


9
10
11
12
cuando_
}
cuando_hace_click (metho
cuando_pulsa_tecla
cuando_suelta_tecla
avisar_cuando_suelta_tecl

Controles Gamepad

Pilas permite acceder a controles externos tipo gamepad para que puedas hacer juegos mucho más interactivos y fáciles de jugar.

Cada vez que conectes un gamepad pilas lo reconocerá y creará un objeto interno para que puedas acceder a cada uno de los botones y controles del gamepad.

Tipos de gamepad soportados

Pilas interactúa con algunas funciones públicas de los navegadores para acceder a los gamepad, así que no soporta todo tipo de controles.

Los gamepads que probamos y pudimos ver funcionando son los siguientes:

- gamepad xbox360 sobre windows 10 (tanto cableado usb y como mediante un receptor inalámbrico usb).
- pro controller de nintendo switch (bluetooth).
- joy-con de nintendo switch (bluetooth).

Cómo verás la lista no es muy larga, muchos gamepads genéricos no funcionan con navegadores modernos, y otros gamepads modernos como los de PS4 aún no llegamos a probarlos (pero creemos que podrían funcionar).

Así que te recomendamos realizar una prueba antes de comenzar a programar: ejecuta alguno de los ejemplos diseñados para probar gamepads y corrobora si tu gamepad funciona correctamente dentro de pilas.

Los ejemplos que te recomendados visitar son “gamepad-simple” y “gamepad-analógicos”:

Acceder a los controles

Para controlar un actor mediante un gamepad no hay que hacer nada especial, si pilas detecta un gamepad conectado va a permitirte acceder a él mediante los atributos:

```
this.control.izquierda
```

Figure 60: ejemplos

```
this.control.derecha
this.control.arriba
this.control.abajo
etc...
```

Estos atributos son muy convenientes, porque el jugador va a poder controlar al personaje usando un gamepad o el teclado.

Estos atributos se pueden consultar desde el código de un actor así (ten en cuenta escribir este código dentro de la función `actualizar`):

```
if (this.control.izquierda) {
  this.x -= 5;
}

if (this.control.arriba) {
  this.y += 5;
}

// etc.
```

Sin embargo, puede suceder que quieras acceder a todas las funcionalidad de un gamepad, como el hecho de que tienen dos controles analógicos o varios botones. En ese caso, deberías acceder a los controles mediante la variable `this.pilas.control.pad_1`.

Por ejemplo, para mover un personaje de forma gradual usando el analógico izquierdo deberías usar un código como este:

```
this.x += this.control.gamepad_1.analogico_izquierdo_x * 10;
this.y += this.control.gamepad_1.analogico_izquierdo_y * 10;
```

Aquí no hace falta usar una condición de tipo `if`, porque el atributo `analogico_izquierdo_x` y `analogico_izquierdo_y` nos devolverá un número

entre `-1` y `1` si el control se está moviendo en alguna dirección o simplemente `0` si está en reposo.

Para el analógico derecho también existe un atributo llamado `analogico_derecho_x` y `analogico_derecho_y`.

Luego el acceso a los botones se realiza mediante los atributos comenzados con el prefijo `boton_`, por ejemplo estos:

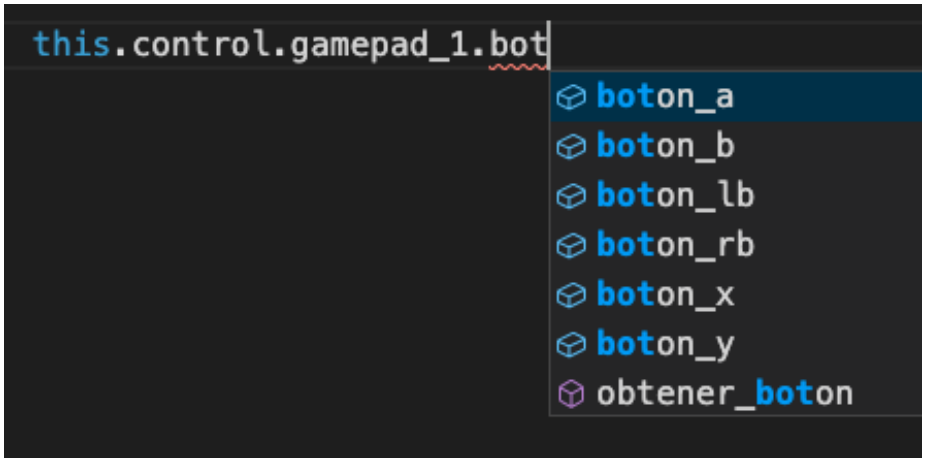


Figure 61: botones

Esta imagen resume varios los controles más importantes:

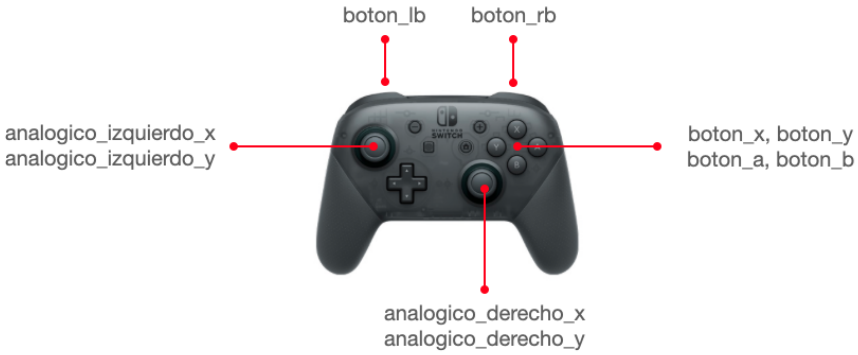


Figure 62: diagrama

Sin embargo te recomendados ver el autocompletado código directamente en el editor porque hay muchos más atributos para explorar ahí. # Dibujado simple en pantalla

En esta sección veremos como dibujar libremente, ya sean lineas, círculos, etc..

Comenzaremos con una forma de dibujado muy sencilla, existe un actor llamado Pizarra que podemos colocar en una escena para dibujar figuras geométricas sencillas.

Cuando se agrega el actor pizarra, vas a observar que soporta algunos métodos para dibujar. Esta es una lista de los más utilizados:

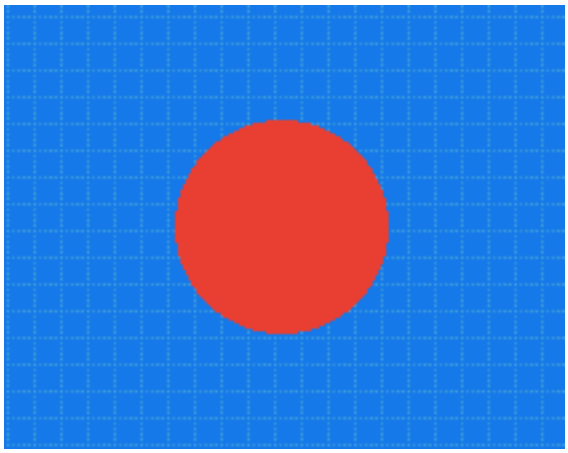
Dibujar círculos

Para dibujar círculos tenemos una función que pinta el contenido de un círculo en base a 4 parámetros:

- x: la posición horizontal del centro del círculo.
- y: la posición vertical del centro del círculo.
- radio: el radio del círculo.
- color: el color de relleno que tendrá el círculo.

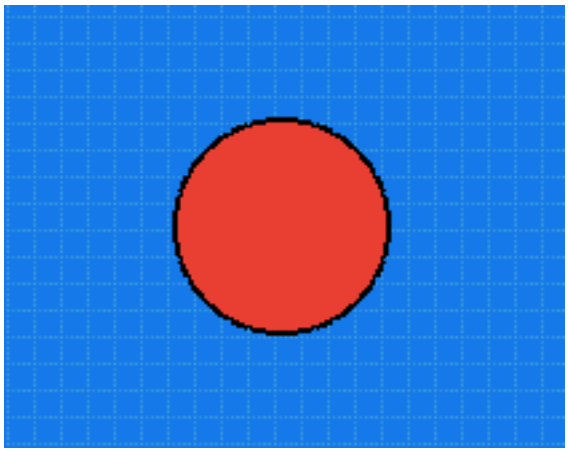
Por ejemplo, esta llamada a la función generará un círculo en la posición `x=100 y=0` con un radio de 40 píxeles y de color rojo:

```
this.dibujar_circulo(100, 0, 40, "rojo");
```



Opcionalmente, si quieres dibujar el contorno del círculo de otro color o con un grosor en particular podrías llamar a esta función:

```
this.dibujar_borde_de_circulo(100, 0, 40, "negro", 2);
```



Los primeros 3 parámetros son idénticos al de la función anterior, corresponden a los parámetros `x`, `y` y `radio`, mientras que el argumento que le sigue será el color del borde y el último el grosor en píxeles.

Colores

En los ejemplos anteriores utilicé colores como "rojo", "negro" y "verde"; sin embargo esta no es la única forma de especificar colores. Los nombres de colores son útiles y fáciles de recordar, pero no ofrecen mucha variedad.

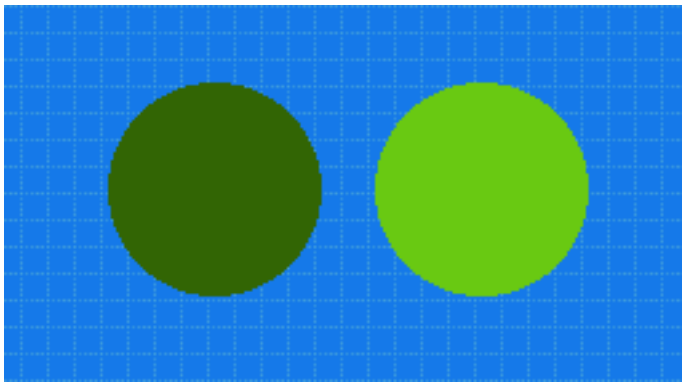
Otra forma de definir colores con mucha flexibilidad es mediante la función `pilas.colores.generar`. Esta función genera un color "mezclando" tres componentes de color (rojo, verde y azúl).

Cada componente de color tienen que ser un número entre 0 y 255, por ejemplo:

```
let color_verde_oscuro = this.pilas.colores.generar(0, 100, 0);
let color_verde_claro = this.pilas.colores.generar(0, 200, 0);
```

A su vez, una vez que tenemos el color generado, podemos usarlo en las funciones para dibujar. Por ejemplo, si queremos dibujar dos círculos con estos colores podemos escribir lo siguiente:

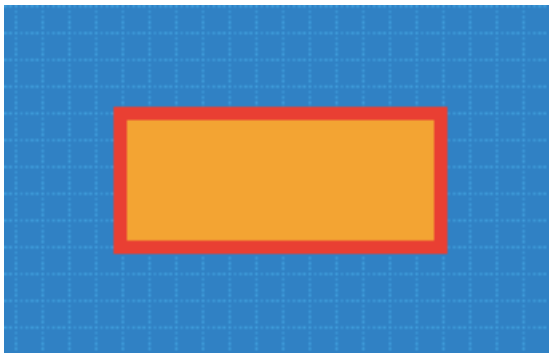
```
this.dibujar_circulo(-50, 50, 40, color_verde_claro);
this.dibujar_circulo(+50, 50, 40, color_verde_oscuro);
```



Rectángulos

De forma similar a los círculos, también existen funciones para dibujar rectángulos y bordes de rectángulos.

```
this.dibujar_rectangulo(0, 0, 120, 50, "naranja");
this.dibujar_borde_de_rectangulo(0, 0, 120, 50, "rojo", 5);
```



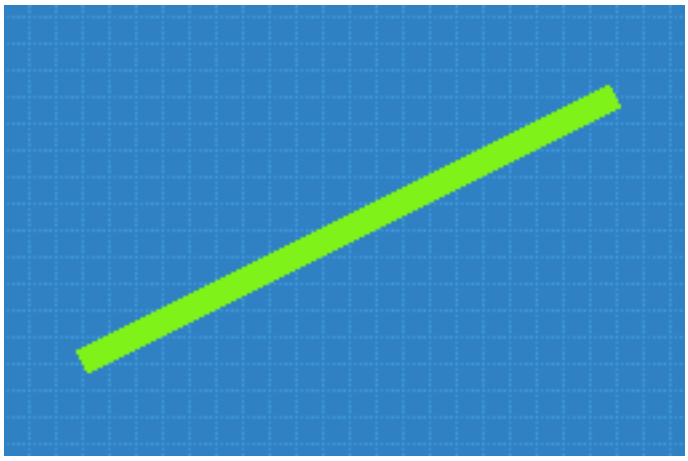
Los argumentos de estas funciones son: el punto de originen, en este caso `x=0` `y=0`, luego el ancho y alto del rectángulo y por último el color.

Lineas

Para dibujar lineas, tenemos que especificar dos coordenadas, color y grosor de la linea:

Por ejemplo, para dibujar una linea de color “verde” desde el punto (0, 0) al punto (200, 100) podemos escribir:

```
this.dibujar_linea(0, 0, 200, 100, "verde", 10);
```



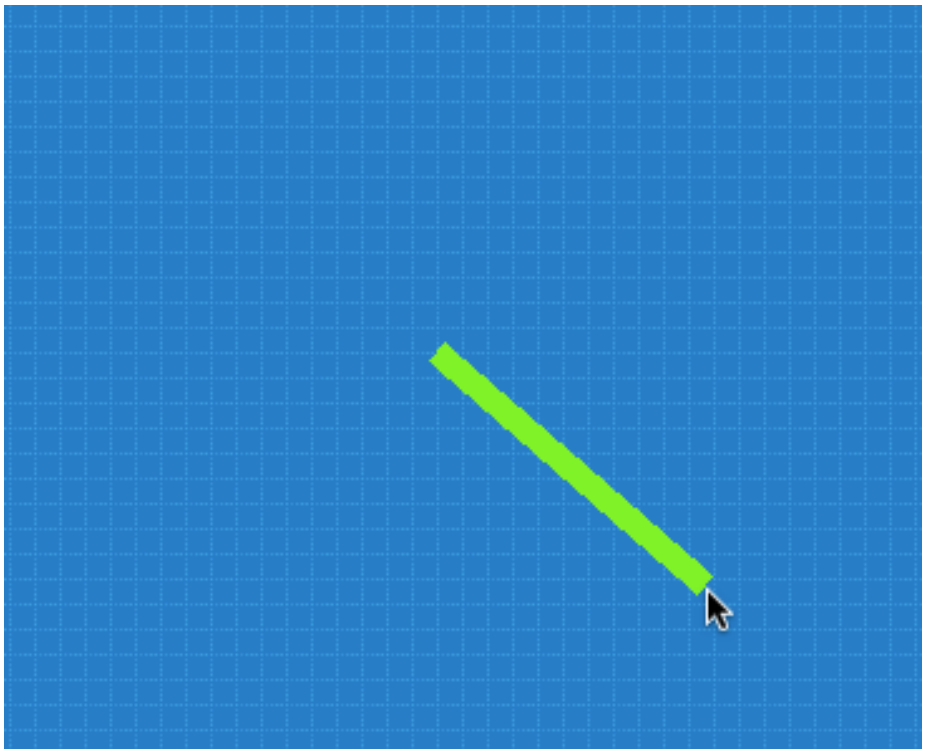
Animaciones

Si bien el actor pizarra pude moverse en pantalla, tener una figura física e incluso ser utilizado como cualquier otro actor. También es posible usarlo para crear animaciones cuadro a cuadro.

El actor Pizarra incluye una función llamada `limpiar` que si se combina con funciones de dibujado se pueden hacer algunas animaciones simples.

Por ejemplo, si queremos dibujar una linea que señale la posición del mouse constántemente podemos hacerlo así, usando la función actualizar:

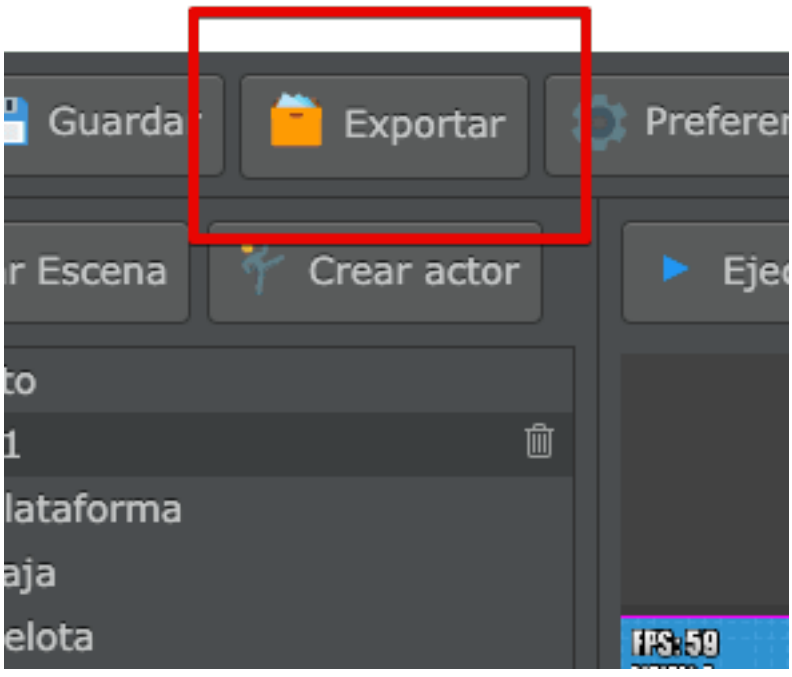
```
actualizar() {
    this.limpiar();
    this.dibujar_linea(0, 0, this.pilas.cursor_x, this.pilas.cursor_y, "verde", 10);
}
```



Cómo exportar juegos

El editor de pilas incluye la posibilidad de exportar proyectos completos para ser utilizados sin el editor.

Para exportar un juego simplemente hay que pulsar el botón Exportar del editor:



A continuación se nos va a mostrar una pantalla de exportación, que demora unos minutos ya que tiene que recopilar todos los recursos necesarios, y luego obtendremos un archivo comprimido con el proyecto:

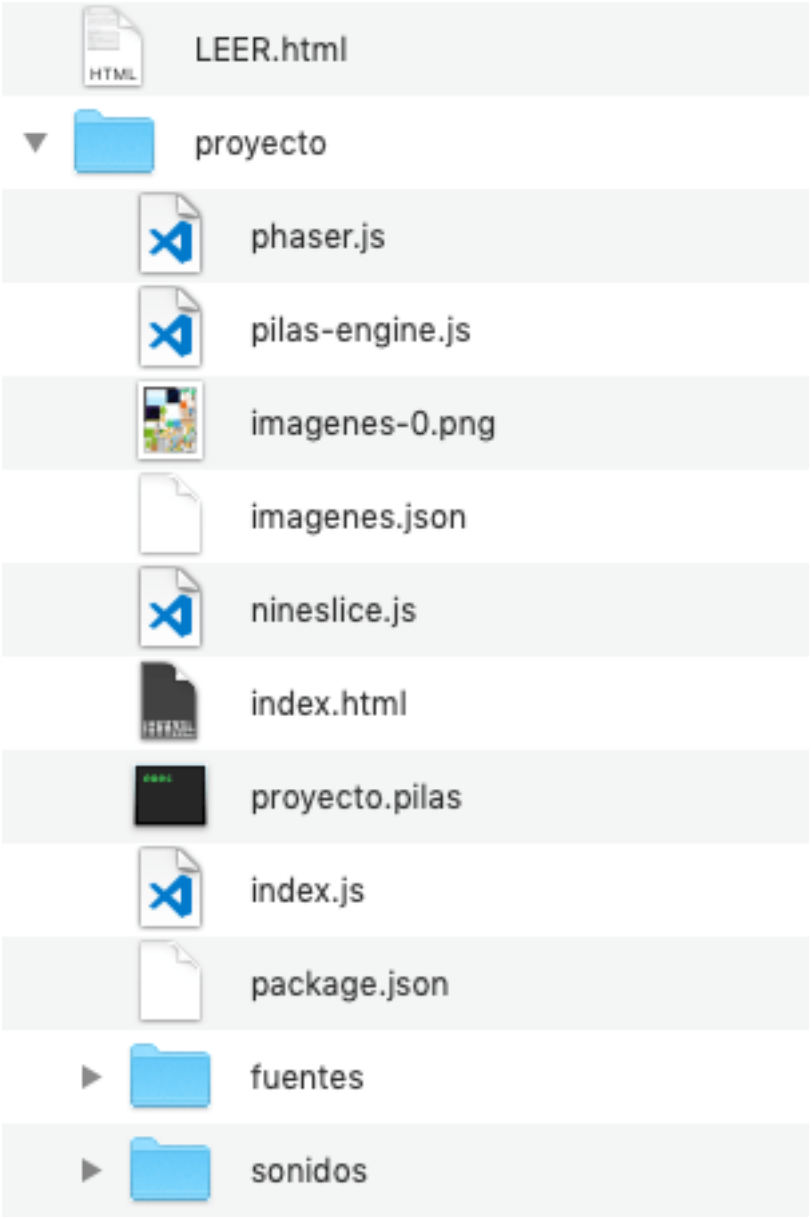
Este archivo .zip aloja el proyecto completo, así que podemos utilizarlo para punto de partida para llevar nuestro juego a otro lugar.

A continuación se muestran algunas alternativas de publicación:

¿Qué contiene el archivo exportado?

El archivo .zip se puede descomprimir fácilmente con las herramientas que vienen instaladas en cualquier sistema operativo.

Una vez que se descomprime, vamos a observar un archivo de ayuda y una carpeta llamada “proyecto” con todos los recursos de nuestro juego:



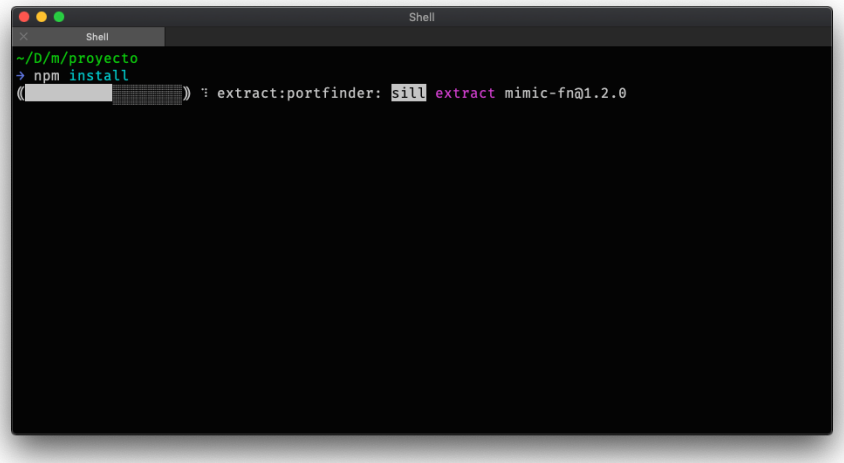
Cómo ejecutar mi juego sin el editor

El primer paso es tener instalado Node.JS y acceso a un terminal para ejecutar comandos.

Luego, se tiene que ingresar en el directorio “proyecto” y ejecutar el comando:

```
npm install
```

Si todo va bien, deberíamos ver que el instalador de paquetes de node trabaja unos segundos para descargar todo lo necesario para seguir trabajando:



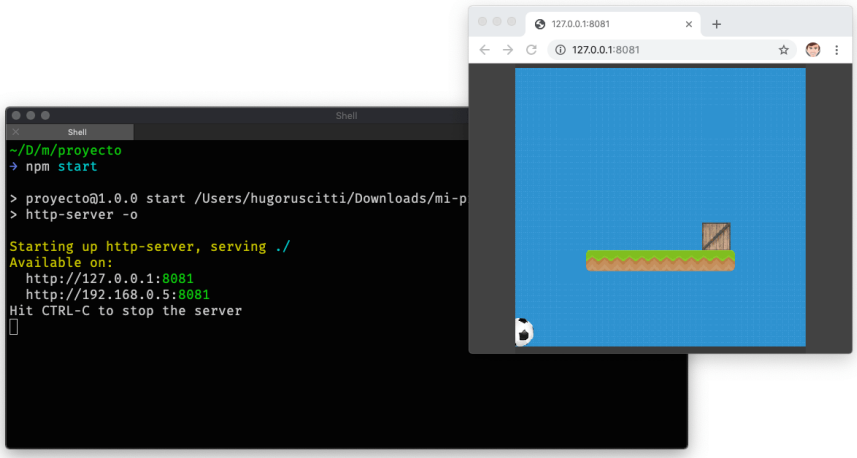
A partir de este punto, vas a tener la posiblidad de hacer alguna de estas cosas:

Probar la aplicación en un servidor local

Para iniciar el juego directamente en un navegador podrías escribir este comando en el mismo directorio “proyecto”:

```
npm start
```

A continuación se va a abrir el navegador y tu juego.



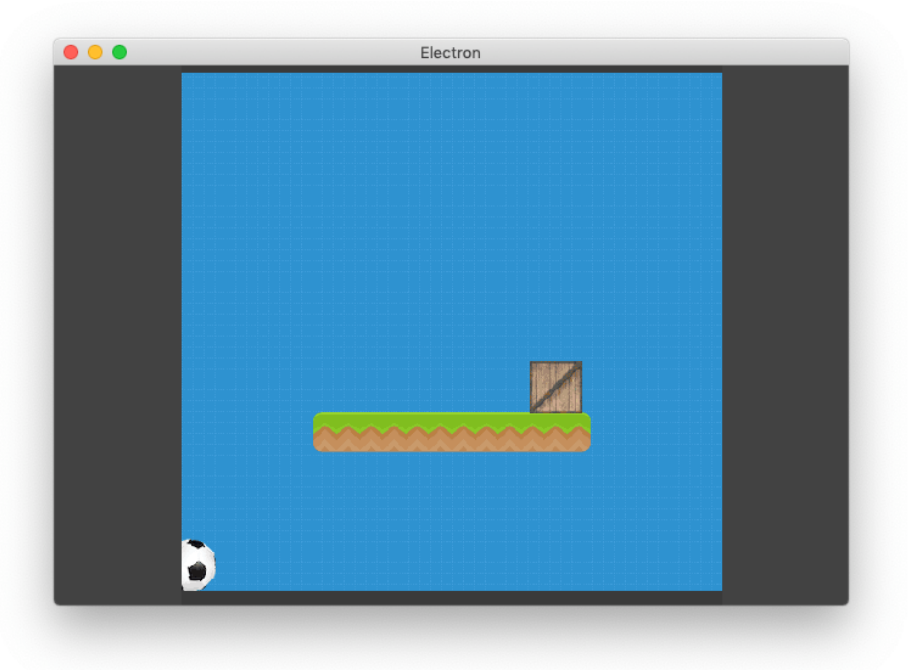
Ten en cuenta que pilas no funciona directamente abriendo el archivo index.html en un navegador, la razón por la que no puede andar directamente es porque necesita cargar archivos mientras se inicializa usando ajax (algo que los navegadores no pueden hacer si se inicializan abriendo un archivo .html de forma directa).

Ejecutar tu juego en electron de forma offline

Otra opción es lanzar el juego como si se tratara de una aplicación dedicada, sin necesidad de un navegador.

Para eso, tendrías que ejecutar este comando:

```
npm run electron
```



Cómo crear versiones empaquetas para distribuir (.exe, dmg etc)

Una opción muy solicitada a la hora de exportar juegos es poder hacerlos completamente portables, y que la persona que recibe el juego no tenga que instalar o configurar nada para jugar al juego, solamente hacer "doble click" en un archivo y jugar.

Pilas permite hacer esto mediante la herramienta que mencionamos antes llamada electron y una serie de scripts.

Tomemos el ejemplo más solicitado, hagamos que nuestro juego se pueda compilar para windows y distribuirse como un archivo .exe. Lo que tenemos que hacer es hacer doble click en los archivos:

- instalar_dependencias.bat
- generar_version_exe.bat

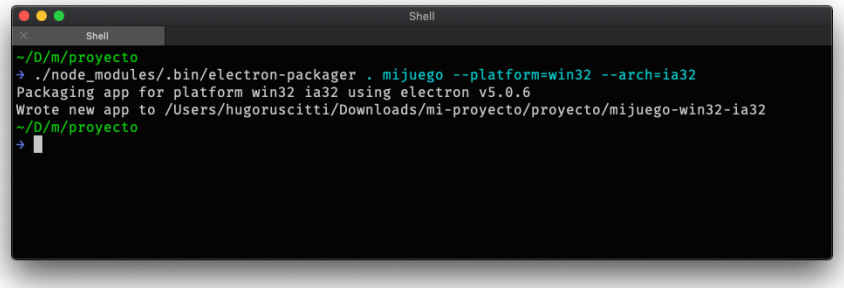
o bien, abrir el comando "cmd", ingresar en el directorio del juego exportado y ejecutar estos comandos:

```
npm install electron-packager --save-dev
```

```
node_modules/.bin/electron-packager . mijuego --platform=win32 --arch=ia32
```

El comando demorará unos segundos, cuando termine aparecerá un mensaje

indicando que los archivos están listos:



En mi caso, los archivos se generaron en un directorio llamado "mijuego-win32-ia32". Ese directorio completo es el que se puede entregar a los usuarios de windows para que puedan ejecutar el juego haciendo doble click en el archivo .exe:

Si además de windows estás buscando llevar tu juego a otras plataformas como gnu/linux, mac/osx o raspberry también hay un parámetro para lanzar la compilación en todas las plataformas soportadas así:

```
node_modules/.bin/electron-packager . mijuego --all
```

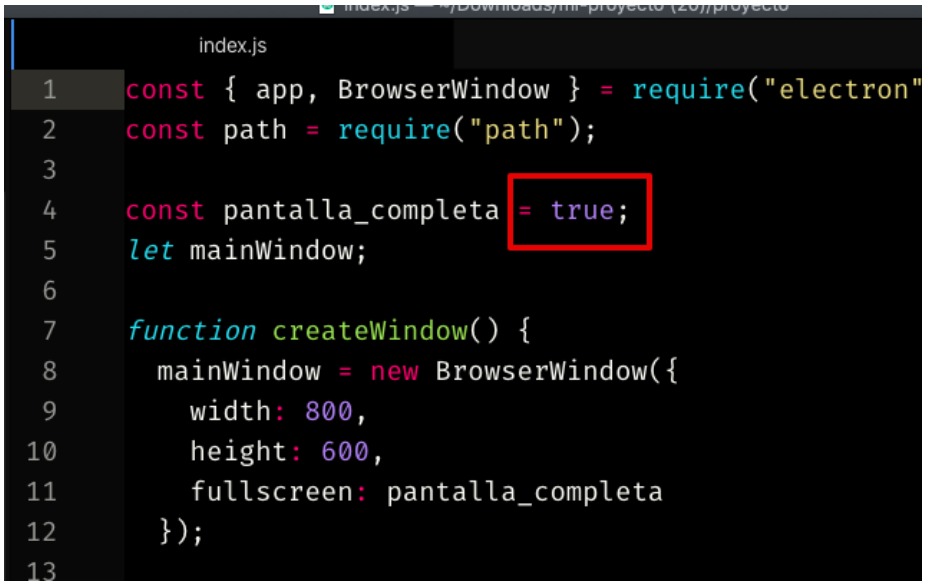
Hay varios parámetros más, pero para no extendernos mucho te dejamos la

documentación detallada de electron-packager

Por cierto, si quieres investigar exactamente qué hacen estos scripts podrías profundizar tu investigación mirando la documentación de electron sobre distribución de binarios ya que se describe cada uno de los pasos de compilación de forma manual.

Cómo ejecutar el juego en pantalla completa

Si empaquetas el juego con electron, como muestra la sección anterior, también tienes la posibilidad de configurar la compilación para que funcione en modo pantalla completa. Para eso tienes que editar el archivo `index.js` y colocar `true` en la variable `pantalla_completa`:



Cómo llevar mi juego a celulares, tablets y tiendas oficiales de apple o google

Pilas incluye una forma sencilla de probar juegos en celulares y tablets, si solo quieres probar tus juegos te recomendamos mirar ahí primero.

Ahora bien, si en realidad lo que buscas es llevar tu juego de forma permanente a un celular o tablet, ya sea para tí, amigos o para publicar en un tienda oficial, lo que necesitas es instalar cordova, los SDK oficiales para tu tipo de dispositivo (android o ios) y compilar tu juego por completo.

```
npm install cordova -g
```

```
cordova create cordova
```

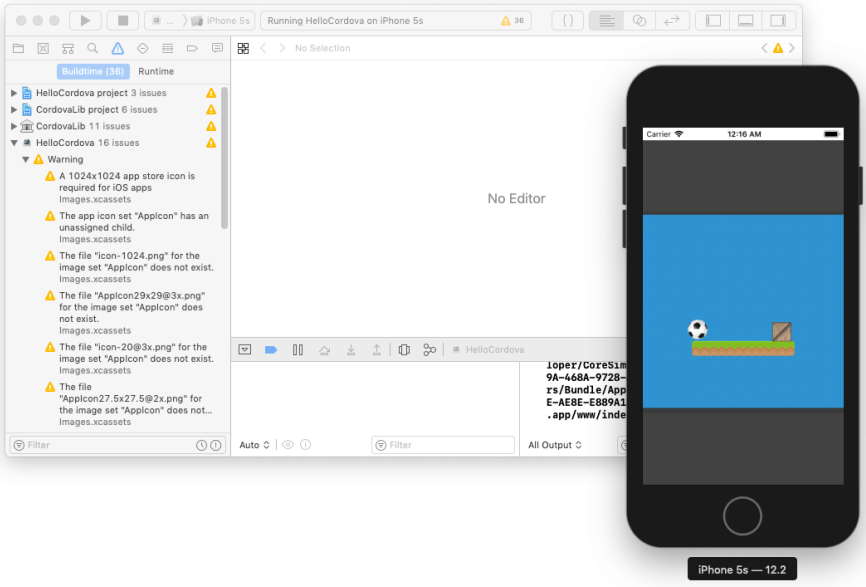
Luego, tendrías que mover todos los archivos del directorio “proyecto” (index.html, pilas-engine.js etc...) dentro del directorio “cordova/www”.

Por último, tendrías que agregar la plataforma principal (android o ios) y luego iniciar la compilación:

Por ejemplo, para ios:

```
cd cordova
cordova platform add ios
cordova build ios
```

y luego abrir el proyecto desde el directorio “platforms/ios” en xcode y ejecutar:



Y para android:

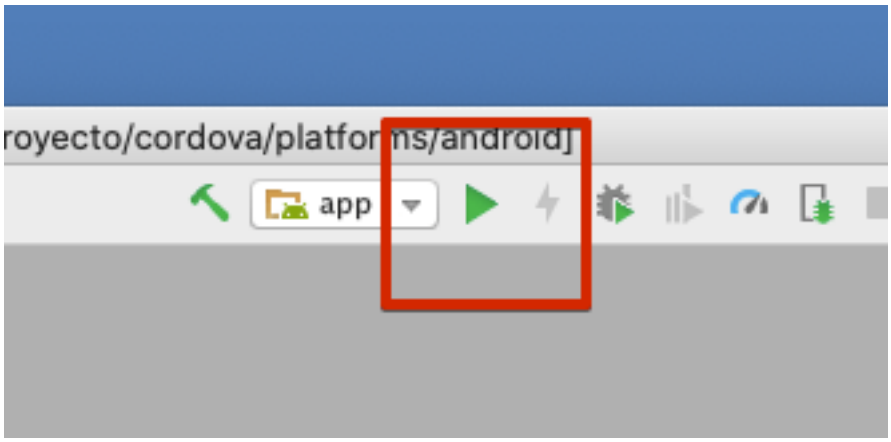
```
cd cordova
cordova platform add android
```

En este punto, asegúrate de haber movido todos los archivos del directorio “proyecto” (index.html, pilas-engine.js etc...) dentro del directorio “cordova/www”.

Luego tienes que generar el proyecto para android con este comando:

```
cordova build android
```

y luego abrir el proyecto del directorio “platforms/android” con Android Studio y pulsar el botón ejecutar para lanzar el emulador:



O bien, abrí el menú “Build” y luego “Build APK(s)” o “Generate Signed APK” para crear el un archivo `.apk` e instalarlo directamente en tu equipo o subirlo a la tienda Play Store de Google.

Colaborar con Pilas Engine

Pilas Engine es un proyecto de Software Libre, que se desarrolla de forma colaborativa por un equipo de varios colaboradores, personas del equipo de desarrollo y la comunidad de usuarios.

Todas las personas que quieran participar del desarrollo son bienvenidas, Pilas nos es solo una herramienta de software, es un proyecto colaborativo que sostiene un equipo de personas amigables y respetuosas que buscan mejorar la calidad de la educación brindándole a los Jóvenes la posibilidad de volverse protagonistas en el uso de la tecnología.

Por ese motivo, queremos invitarte a participar. ¡Vos también puedes formar parte del equipo de Pilas!.

Te damos algunas ideas de participación que podrían interesante:

Participar en el foro de Pilas Engine

La tarea mas recomendable para quienes recién comienzan es dar de alta una cuenta en nuestro foro (http://foro.pilas-engine.com.ar), presentarse y ayudarnos a resolver preguntas de otros usuarios.

No hace falta experiencia previa para colaborar en el foro, puedes ingresar, saludar y contarnos qué te parece Pilas, cómo lo conociste; vas a notar que el foro es el lugar ideal para aportar ideas, conversar, y publicar juegos.

Ayudar a difundir Pilas

Muchas personas no conocen Pilas, y creemos que podría interesarles. Así que otra forma de colaborar es difundir la existencia de Pilas entre las personas que conoces.

Además, si conoces escuelas o proyectos donde se utilice Pilas dales ánimo para que nos cuenten sus proyectos, queremos ayudar y mejorar Pilas con cada posibilidad que se nos presente.

También es super útil si tienes la posibilidad de mencionar Pilas las redes sociales: twitter, facebook, youtube o ¡donde quieras!

Creá juegos, cursos o tutoriales

Necesitamos tutoriales, juegos y cursos sobre Pilas Engine.

Si estás haciendo un juego con Pilas, contanos de qué se trata; puedes escribirnos directamente desde el foro de mensajes de la web (http://foro.pilas-engine.com.ar).

Si te gusta enseñar y contar tus experiencias con la herramienta, te invitamos a escribir y proponer cursos dentro del foro; además, en la web tenemos una sección en donde podemos publicar artículos, así que seguramente también podamos ayudarte a subirlos ahí.

Pilas como biblioteca externa

Pilas se puede usar como una biblioteca externa para permitir un uso avanzado y muy personalizado de pilas.

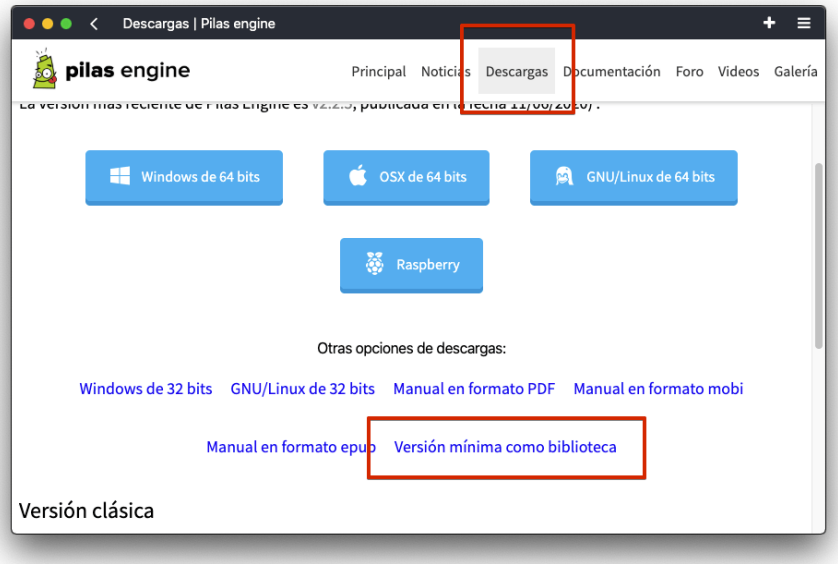
De todas formas, la opción más recomendada para llevar tu juego a blogs, mobile u otros medios es siguiendo las instrucciones que están en esta otra sección: Cómo exportar juegos

Esta sección solo es aconsejable para programadores avanzados que quieran integrar pilas a otras bibliotecas, como react, vue o similares, partiendo de un código minimalista, sin imágenes, sonidos ni actores predeterminados de pilas.

Archivos iniciales

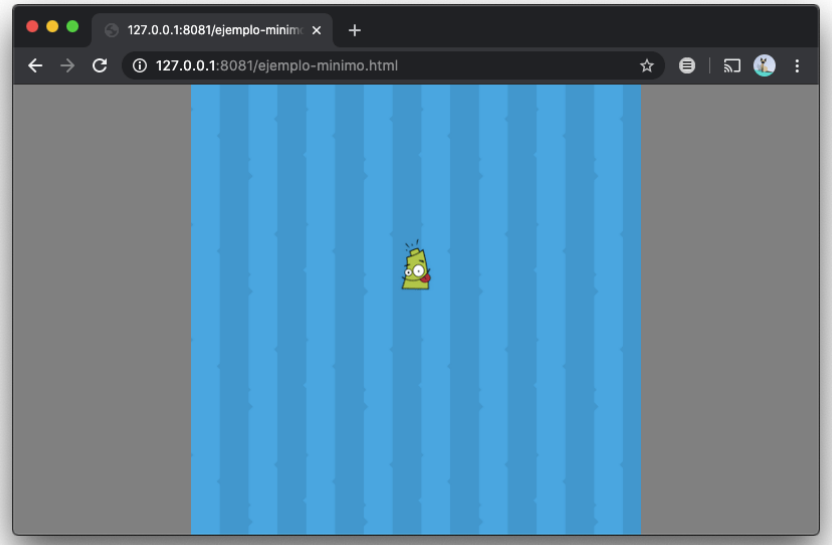
Para utilizar pilas como biblioteca se suele crear un archivo `.html` e incorporar las dependencias del proyecto, como `phaser` y `nine-slice`. Así que para hacer más sencilla esta tarea inicial pilas se distribuye junto a una plantilla que tiene todos estos archivos listos y configurados para utilizar.

Ingresá en el sitio de pilas, y en la sección descargas vas a encontrar un link para descargar una plantilla llamada “versión-minima”:



Este archivo `.zip` tiene lo mínimo y necesario para cargar pilas y poder usar el motor directamente:

Ten en cuenta que para utilizar esta versión tienes que descomprimir el archivo `.zip`, luego iniciar un servidor web y por último abrir el archivo `.html` desde un navegador:



Es importante usar un servidor web y no abrir el archivo directamente ya que pilas (y phaser) usan una tecnología llamada Ajax para cargar imágenes y otros recursos.

A partir de ahí, para editar tu juego deberías inspeccionar el archivo `.html`, en donde aparecerá el código mínimo para iniciar pilas, crear un actor y moverlo por la pantalla:

```
var imagenes = [
  {
    nombre: "logo",
    ruta: "logo.png"
  },
  {
    nombre: "fondo",
    ruta: "fondo.png"
  }
];

var pilas = pilasengine.iniciar(500, 500, recursos, opciones, imagenes, true);

pilas.onready = function() {
  let logo = pilas.actores.actor_basico("logo");
  logo.transparencia = 100;

  logo
```

```
    .animar()
    .mostrar()
    .mover_hasta(0, 50);
};
```

Por supuesto puedes cambiar ese código y adaptarlo a tus necesidades, pero no olvides que esta versión mínima no incluye las tipografías, sonidos e imágenes que incluye pilas, deberías cargar archivos `.png` propios para tener más posibilidades.

Entorno para colaboradores de Pilas

Si sos desarrollador y queres replicar el entorno completo para editar el código de Pilas te recomendamos una serie de pasos para tener un entorno completo de desarrollo.

Ten en cuenta que si bien esta es una sección muy avanzada del manual de Pilas, cualquier duda que te surja al respecto puedes realizarla en el foro de Pilas (http://foro.pilas-engine.com.ar/) para que podemos mejorar el manual.

¿Qué software se utiliza internamente en Pilas?

Para realizar Pilas usamos principalmente Javascript junto a varios Frameworks y utilidades. Las más importantes son:

- **Phaser**: Es la biblioteca multimedia que utilizamos para dibujar el espacio de juego de pilas, Phaser nos permite cargar imágenes, dibujarlas en pantalla, reproducir sonidos, detectar colisiones, realizar simulaciones físicas etc...
- **ember-js**: Es el framework web que utilizamos para crear la estructura visual y de interacción de la aplicación; incluye componentes, rutas, controladores y complementos. Ember también nos permite empaquetar toda la aplicación y publicarla en la web.
- **Typescript**: Lo utilizamos para el código fuente del motor Pilas. El motor de Pilas es el encargado de declarar todos los actores existentes, es donde vive la estructura de escenas, y donde se encapsulan todas las llamadas a Phaser.
- **pandoc**: Es la herramienta que usamos para componer el manual de pilas. Pandoc convierte archivos markdown en archivos .html para abrir desde la web.
- **electron**: Sirve para empaquetar la aplicación web completa y distribuirla en formato binario, como un ejecutable que se puede usar de forma offline.

Estructura de directorios

Estos son los archivos principales del repositorio y qué función cumplen:

Directorio | ¿Qué función cumple? |
- | |
api_docs | Almacena el resultado de la documentación automática. Si ejecutás "make api" vas a ver cómo se recorre el código de Pilas y se genera esta documentación. |
app | Es un directorio importante, aquí está todo el código fuente de la aplicación realizado con Ember. Dentro están las plantillas, los controladores, componentes y rutas. Más adelante en este tutorial damos detalles sobre cómo editar los archivos de este directorio. Pero a grandes razgos, con "make ejecutar" o "make compilar" se puede invocar a Ember para que convierta todo ese directorio en una aplicación web. |
manual | Contiene todo el manual de Pilas en formato markdown. Podes editar el contenido con cualquier editor de textos, y para convertirlo en html y que se vea dentro de la aplicación deberías ejecutar el comando "make pilas_manual". |
pilas-engine | Contiene el código fuente del motor de pilas. El código está diseñado usando typescript, por ese motivo para compilarlo hay que ejecutar el comando "make compilar_pilas" o "make compilar_pilas_live" para hacerlo de forma continua. |
public | Almacena todos los archivos que se deben servir desde la aplicación emberjs sin procesar. Por ejemplo imágenes, fuentes y sonidos. Pero además, este directorio también se actualiza desde otros comandos. Por ejemplo, cuando compilamos el código del directorio *pilas-engine* con el comando "make compilar_pilas", el archivo generado se copia a este directorio "public". |
tests | Almacena todos los tests de unidad, integración y aceptación. Estos tests se ejecutan cuando se invoca el comando "make test" o cuando se ejecuta "make ejecutar" y luego se ingresa a http://localhost:4200/tests. Además, se ejecutan en remoto cada vez que se hace un push al repositorio. Más adelante en este documento se describe cómo funcionan los tests en la estructura del proyecto. |

Repositorio y modelo de trabajo

Antes de iniciar, necesitas tener instalado NodeJS 8 y el gestor de paquetes "yarn".

Si quieres asegurarte de tener todo correctamente instalado escribí los siguientes comandos:

```
npm install -g yarn
node -v
```

El código completo de Pilas está almacenado en Github dentro del siguiente repositorio:

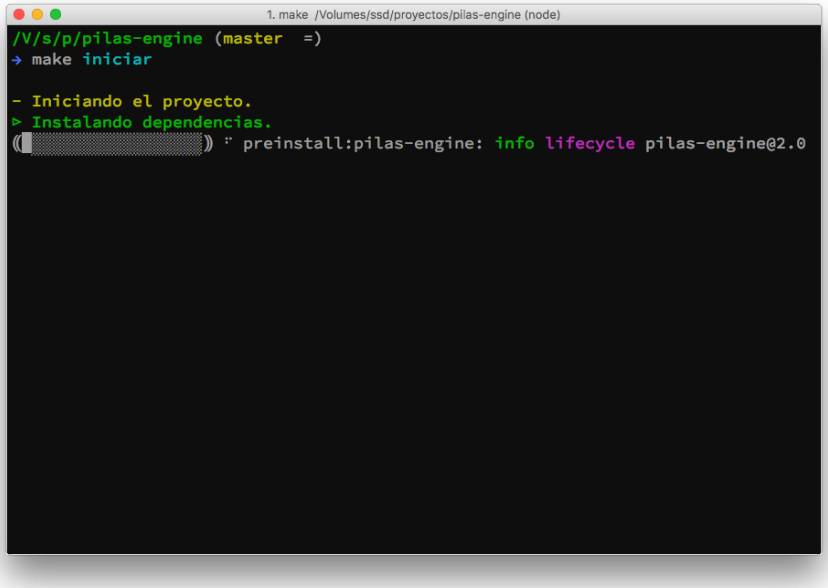
- https://github.com/pilas-engine/pilas-engine

Te recomendamos iniciar un fork del repositorio antes de continuar, hay instrucciones sobre cómo realizar un fork del repositorio aquí: https://help.github.com/articles/fork-a-repo/

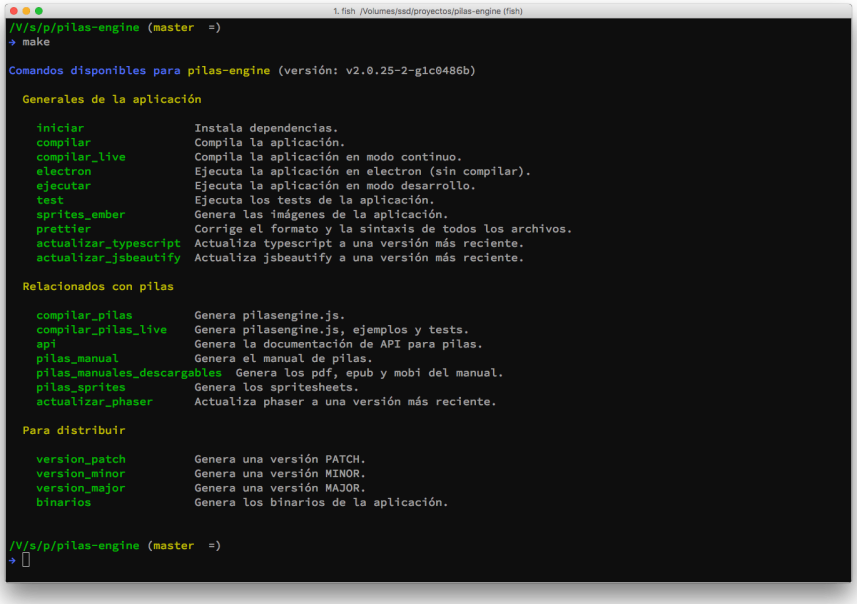
Una vez que tengas realizado el fork, cloná el repositorio con estos comandos e instalá las dependencias del proyecto:

```
make iniciar
```

Deberías ver cómo se van instalando todas las dependencias dentro del directorio "node_modules". Esto puede demorar varios minutos:



Una vez finalizada la instalación de dependencias puedes ejecutar el comando "make". Cuando lo escribas, van a aparecer en pantalla todos los atajos que usamos dentro del proyecto:

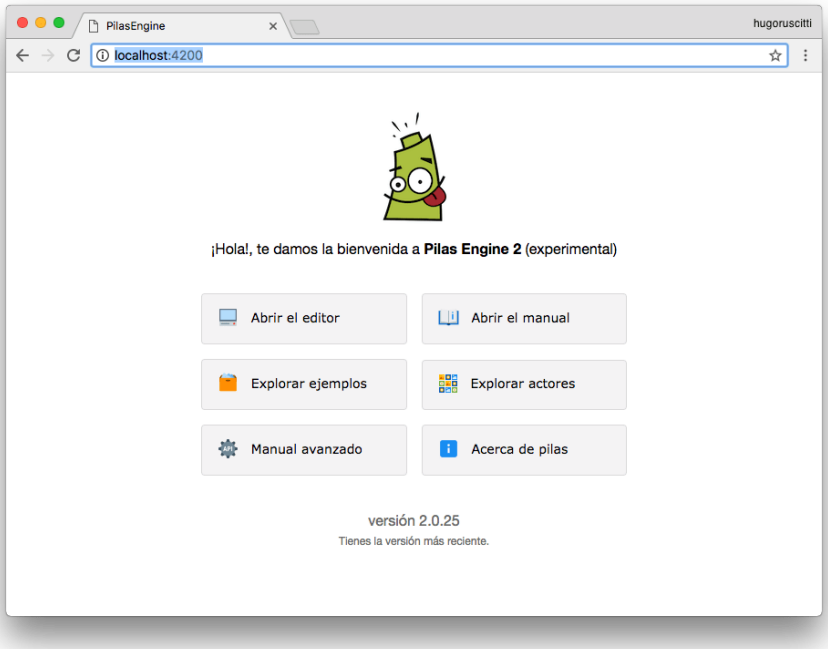


Obviamente en la mayoría de los casos usamos solo algunos comandos, pero siempre vamos a mantener actualizado este listado para simplificar el desarrollo.

Para poner en funcionamiento el servidor de Ember puedes ejecutar este comando:

```
make ejecutar
```

y luego abrí esta dirección en el navegador: http://localhost:4200



Ten en cuenta que el servidor de Ember va a estar en ejecución; cada vez que modifiques un archivo del código de la aplicación Ember se va a encargar de actualizar el navegador automáticamente.

Estilo de programación

Te recomendamos escribir el código en español siempre que sea posible, y aquellas variables o métodos que tienen varias palabras intentaremos escribirlas como "nombre_del_metodo" en lugar de "nombreDelMetodo".

Además, es muy importante que tengas configurado tu editor para usar automáticamente prettier (https://prettier.io/).

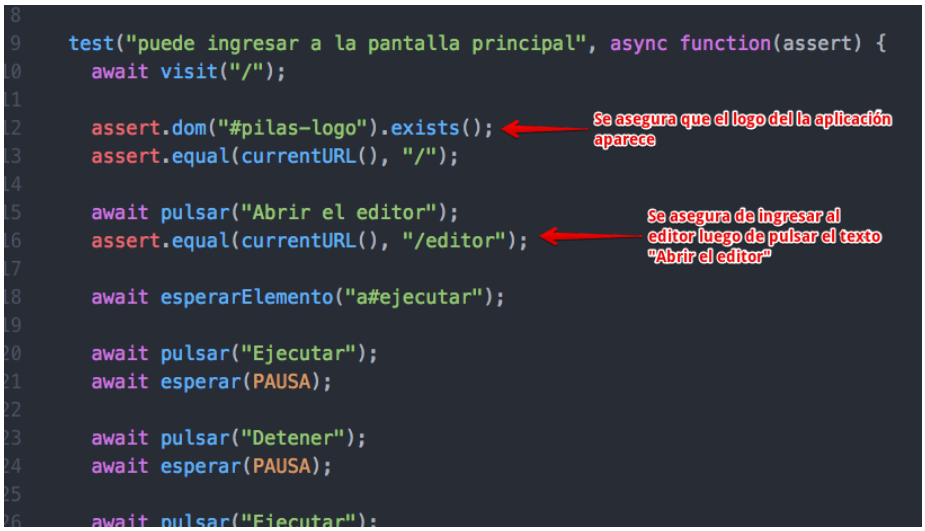
Tests

El proyecto incluye tests que intentamos llevar de la mano con cada característica que agregamos a la herramienta. Los tests se almacenan en el directorio "tests" y cada uno de ellos está diseñado para encargarse del funcionamiento de una característica de la herramienta.

Por ejemplo, el archivo `tests/acceptance/puede-ingresar-al-editor-test.js` se encarga de simular el comportamiento de un usuario que ingresa en la

aplicación y comienza a interactuar con la interfaz.

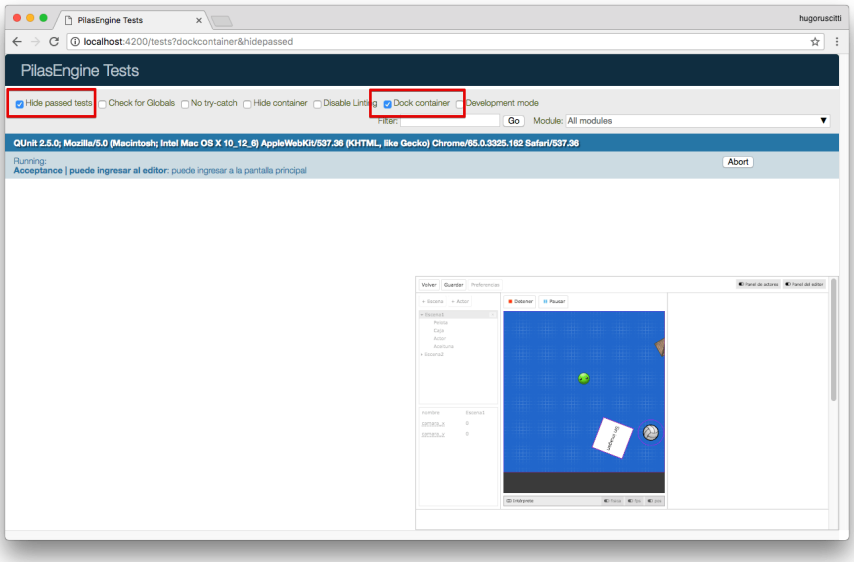
Cada sentencia del archivo intenta simular una acción del usuario y viene acompañada de alguna validación:



De esta forma, tenemos la seguridad de que podemos validar el funcionamiento de toda la herramienta de forma automatizada. Nos permite detectar errores y darnos confianza para re-estructurar o mejorar partes internas de la aplicación sabiendo qué funcionalidad podría comprometer.

Si ejecutas el comando “make ejecutar”, y abrís la aplicación con un navegador desde “http://localhost:4200”, también vas a poder ejecutar los tests directamente en el navegador.

Tendrías que ingresar a la dirección “http://localhost:4200/tests”:



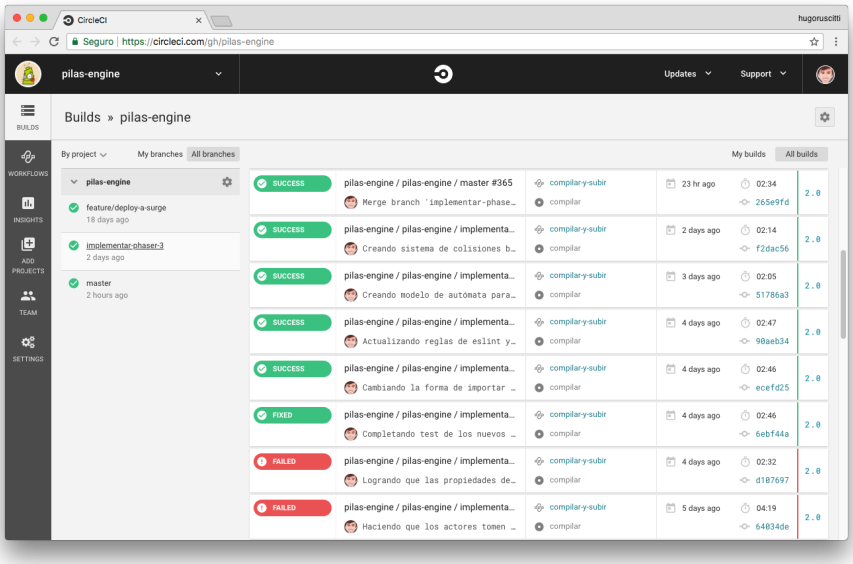
En la imagen marqué dos opciones que me parecen útiles para todas las personas, la primera evita que la pantalla se llene con detalles de tests que corren bien y la segunda es simplemente estética: sirve para ver cómo se ven los tests visualmente al costado de la pantalla.

Automatización e integración contínua

Hay varias tareas que se realizan de forma automática cada vez que realizamos un push al repositorio.

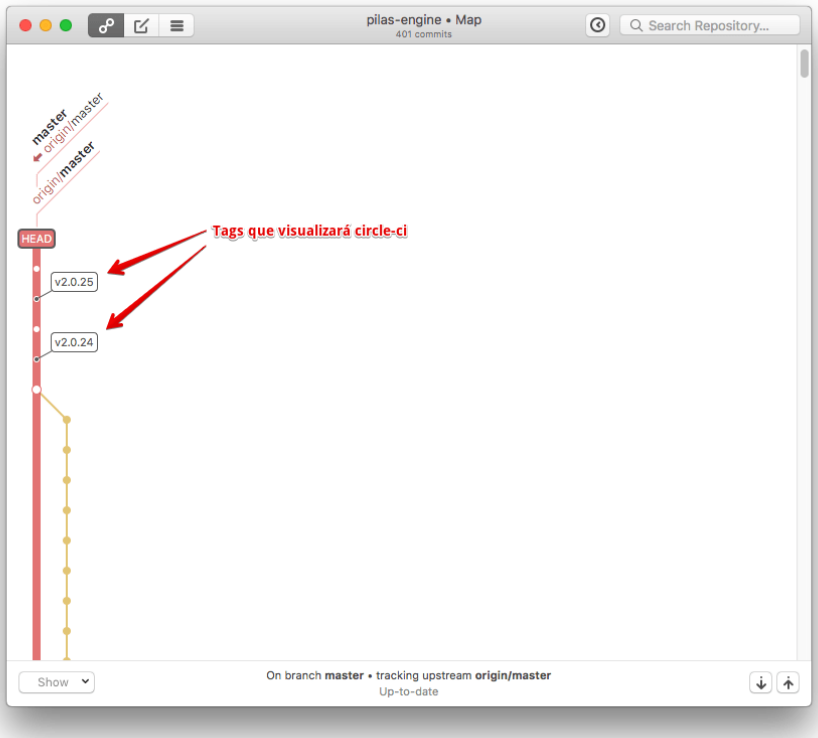
Por ejemplo, cada vez que se ejecuta un push sobre github, un servicio llamado "circle-ci" se encarga de correr todos los tests del proyecto desde cero.

Este servicio no solo ejecuta los tests, sino que nos avisa por email si alguna de las pruebas falló o si alguna fase de la instalación tuvo problemas. Además nos genera un listado histórico para corroborar si un cambio introduce algún tipo de error:

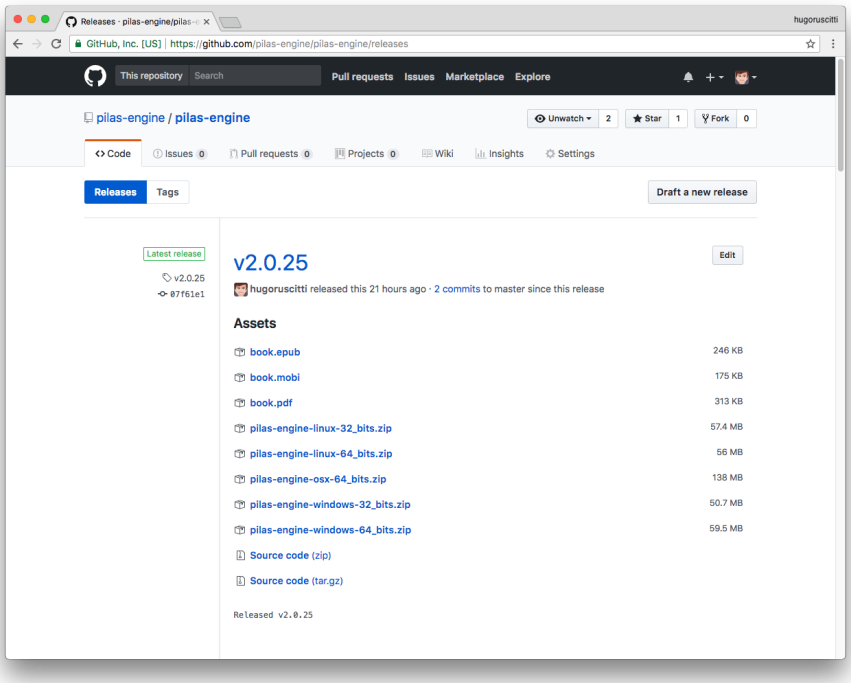


También usamos “circle-ci” cada vez que hacemos un lanzamiento estable de la herramienta para que se generen los binarios y las versiones web de la aplicación.

La forma que utilizamos para marcar que Pilas se puede publicar es realizando un tag. Los tags los va a detectar circle CI para generar binarios y subir una versión actualizada de la aplicación a la web.



Por ejemplo, en la imagen se ve el tag de la versión v2.0.25. Cuando eso sucedió, "circle-ci" generó estos archivos para que esa versión de pilas esté compilada y disponible para descargar desde aquí:



Si bien estos pasos son automáticos, todo el detalle de lo que se debe ejecutar está en el archivo `.circleci/config.yml`.