**Proyecto: un juego de plataformas**

Toda la realidad es un juego.

Iain Banks, *el jugador de juegos*



Gran parte de mi fascinación inicial por los ordenadores, como la de muchos niños nerds, tenía que ver con los videojuegos.

Me sentí atraído por los diminutos mundos simulados que podía manipular y en los que las historias se desarrollaban (más o menos), supongo, más por la forma en que proyectaba mi imaginación en ellos que por las posibilidades que realmente ofrecían.

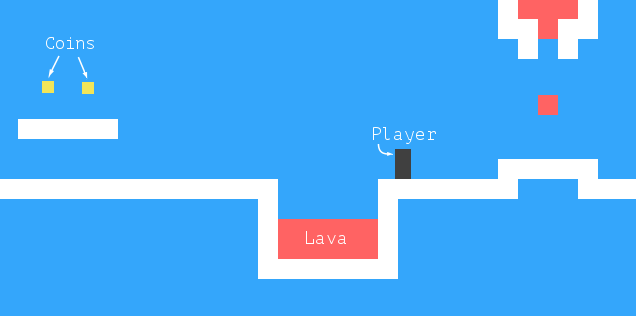
No le deseo a nadie una carrera en programación de juegos. Al igual que en la industria de la música, la discrepancia entre el número de jóvenes deseosos de trabajar en ella y la demanda real de esas personas crea un entorno bastante poco saludable. Pero escribir juegos por diversión es divertido.

Los juegos de plataformas (o juegos de “saltar y correr”) son juegos que esperan que el jugador mueva una figura a través de un mundo, que generalmente es bidimensional y se ve de lado, mientras salta sobre cosas.

# El juego

Nuestro juego se basará aproximadamente en [Dark Blue](http://www.lessmilk.com/games/10) de Thomas Palef.

Elegí ese juego porque es entretenido y minimalista y porque se puede construir sin demasiado código. Se parece a esto:



La caja oscura representa al jugador, cuya tarea es recolectar las cajas amarillas (monedas) mientras evita la materia roja (lava). Se completa un nivel cuando se han recolectado todas las monedas.

El jugador puede caminar con las teclas de flecha izquierda y derecha y puede saltar con la flecha hacia arriba. Saltar es una especialidad de este personaje del juego. Puede alcanzar varias veces su propia altura y cambiar de dirección en el aire. Puede que esto no sea del todo realista, pero ayuda a que el jugador tenga la sensación de tener el control directo del avatar en pantalla.

El juego consiste en un fondo estático, dispuesto como una cuadrícula, con los elementos móviles superpuestos sobre ese fondo. Cada campo de la cuadrícula está vacío, sólido o de lava. Los elementos móviles son el jugador, las monedas y ciertas piezas de lava. Las posiciones de estos elementos no están restringidas a la cuadrícula; sus coordenadas pueden ser fraccionarias, lo que permite un movimiento suave.

## La tecnología

Usaremos el DOM del navegador para mostrar el juego, y leeremos la entrada del usuario manejando eventos clave.

El código relacionado con la pantalla y el teclado es solo una pequeña parte del trabajo que necesitamos hacer para construir este juego. Como todo parece cuadros de colores, dibujar es sencillo: creamos elementos DOM y usamos estilos para darles un color de fondo, tamaño y posición.

Podemos representar el fondo como una tabla, ya que es una cuadrícula de cuadrados que no cambia. Los elementos que se mueven libremente se pueden superponer utilizando elementos absolutamente posicionados.

En los juegos y otros programas que deben animar gráficos y responder a la entrada del usuario sin demoras apreciables, la eficiencia es importante. Aunque el DOM no fue diseñado originalmente para gráficos de alto rendimiento, en realidad es mejor de lo que cabría esperar. Viste algunas animaciones en capítulos anteriores . En una máquina moderna, un juego simple como este funciona bien, incluso si no nos preocupamos mucho por la optimización.

## Niveles

Queremos una forma legible y editable por humanos para especificar niveles. Dado que está bien que todo comience en una cuadrícula, podríamos usar cadenas grandes en las que cada carácter representa un elemento, ya sea una parte de la cuadrícula de fondo o un elemento en movimiento.

El plan para un nivel pequeño podría verse así:

let simpleLevelPlan = `

......................

.. # ................ # ..

.. # .............. =. # ..

.. # ......... oo ... # ..

.. #. @ ...... ##### ... # ..

.. ##### ............ # ..

...... # ++++++++++ ++ # ..

...... ############## ..

...................... ` ;

Los puntos son espacios vacíos, los caracteres hash ( #) son paredes y los signos más son lava. La posición inicial del jugador es el signo arroba ( @). Cada carácter O es una moneda, y el signo igual ( =) en la parte superior es un bloque de lava que se mueve de un lado a otro horizontalmente.

Admitiremos dos tipos adicionales de lava en movimiento: el carácter de tubería ( |) crea manchas que se mueven verticalmente e vindica lava que *gotea;* lava que se mueve verticalmente que no rebota hacia adelante y hacia atrás, sino que solo se mueve hacia abajo, saltando a su posición inicial golpea el suelo.

Todo un juego consta de varios niveles que el jugador debe completar. Se completa un nivel cuando se han recolectado todas las monedas. Si el jugador toca lava, el nivel actual se restaura a su posición inicial y el jugador puede intentarlo de nuevo.

## Leer un nivel

La siguiente clase almacena un objeto de nivel. Su argumento debe ser la cadena que define el nivel.

class Level {

constructor ( plan ) {

dejar filas = planificar . recortar (). dividir ( "\ n" ). mapa ( l => [ ... l ]);

esto . altura = filas . longitud ;

esto . ancho = filas [ 0 ]. longitud ;

esto . startActors = [];

esto . filas = filas . map (( fila , y ) => {

return row . map (( ch , x ) => {

let type = levelChars [ ch ];

if ( typeof type == "string" ) return type ;

this . startActors . push (

tipo . crear( nuevo Vec ( x , y ), ch ));

volver "vacío" ;

});

});

}

}

El trim se utiliza para eliminar los espacios en blanco al principio y al final de la cadena del plan. Esto permite que nuestro plan de ejemplo comience con una nueva línea para que todas las líneas estén directamente una debajo de la otra. La cadena restante se divide en caracteres de nueva línea y cada línea se distribuye en una matriz, produciendo matrices de caracteres.

Por lo que rows contiene una matriz de matrices de caracteres, las filas del plan. Podemos derivar el ancho y la altura del nivel a partir de estos. Pero aún debemos separar los elementos en movimiento de la cuadrícula de fondo. Llamaremos *actores de* elementos móviles. Se almacenarán en una serie de objetos. El fondo será una serie de matrices de cadenas, la celebración de los tipos de campo como "empty", "wall"o "lava".

Para crear estas matrices, asignamos las filas y luego su contenido. Recuerde que mappasa el índice de la matriz como segundo argumento a la función de mapeo, que nos dice las coordenadas x e y de un carácter dado. Las posiciones en el juego se almacenarán como pares de coordenadas, siendo la parte superior izquierda 0,0 y cada cuadrado de fondo de 1 unidad de alto y ancho.

Para interpretar los personajes en el plan, el Level usa el levelChars, que asigna elementos de fondo a cadenas y personajes de actor a clases. Cuando typees una clase de actor, su create estático se usa para crear un objeto, que se agrega startActors y la función de mapeo regresa "empty"para este cuadrado de fondo.

La posición del actor se almacena como un Vec. Este es un vector bidimensional, un objeto con propiedades xy.

A medida que avanza el juego, los actores terminarán en diferentes lugares o incluso desaparecerán por completo (como hacen las monedas cuando se recolectan). Usaremos una State para rastrear el estado de un juego terrestre.

clase Estado {

constructor ( nivel , actores , estado ) {

esto . nivel = nivel ;

esto . actores = actores ;

esto . status = status ;

}

static start ( level ) {

return new State ( level , level . startActors , "playing" );

}

get player () {

devuelve esto . actores . buscar ( a => a . tipo == "jugador" );

}

}

La statuspropiedad cambiará a "lost"o "won"cuando el juego ha terminado.

Esta es nuevamente una estructura de datos persistente: la actualización del estado del juego crea un nuevo estado y deja el anterior intacto.

## Actores

Los objetos Actor representan la posición actual y el estado de un elemento en movimiento dado en nuestro juego. Todos los objetos de actor se ajustan a la misma interfaz. Su pos contiene las coordenadas de la esquina superior izquierda del elemento y su size mantiene su tamaño.

Luego tienen un update, que se utiliza para calcular su nuevo estado y posición después de un período de tiempo determinado. Simula lo que hace el actor, moviéndose en respuesta a las teclas de flecha del jugador y rebotando hacia adelante y hacia atrás en busca de la lava, y devuelve un objeto actor nuevo y actualizado.

Una type contiene una cadena que identifica el tipo del actor- "player", "coin"o "lava". Esto es útil cuando se dibuja el juego: el aspecto del rectángulo dibujado para un actor se basa en su tipo.

Las clases de actor tienen un create estático que utiliza el Level para crear un actor a partir de un personaje en el plano de nivel. Se le dan las coordenadas del personaje y el personaje en sí, lo cual es necesario porque la Lava maneja varios caracteres diferentes.

Esta es la Vec que usaremos para nuestros valores bidimensionales, como la posición y el tamaño de los actores.

class Vec {

constructor ( x , y ) {

esto . x = x ; esto . y = y ;

}

más ( otro ) {

return new Vec ( este . x + otro . x , este . y + otro . y );

}

times ( factor ) {

return new Vec ( este . x \* factor , este . y \* factor );

}

}

El times escala un vector por un número dado. Será útil cuando necesitemos multiplicar un vector de velocidad por un intervalo de tiempo para obtener la distancia recorrida durante ese tiempo.

Los diferentes tipos de actores obtienen sus propias clases ya que su comportamiento es muy diferente. Definamos estas clases. Llegaremos a sus update más tarde.

La clase de jugador tiene una propiedad speed almacena su velocidad actual para simular el impulso y la gravedad.

class Player {

constructor ( pos , speed ) {

this . pos = pos ;

esto . velocidad = velocidad ;

}

get type () { return "jugador" ; }

static create ( pos ) {

return new Player ( pos . plus ( new Vec ( 0 , - 0.5 )),

new Vec ( 0 , 0 ));

}

}

Jugador . prototipo . tamaño = nuevo Vec ( 0.8 , 1.5 );

Debido a que un jugador tiene un cuadrado y medio de altura, su posición inicial se establece en medio cuadrado por encima de la posición donde @apareció el personaje. De esta manera, su parte inferior se alinea con la parte inferior del cuadrado en el que apareció.

La size es la misma para todas las instancias de Player, por lo que la almacenamos en el prototipo en lugar de en las propias instancias. Podríamos haber usado un getter como type, pero eso crearía y devolvería un nuevo Vec cada vez que se lea la propiedad, lo que sería un desperdicio. (Las cadenas, al ser inmutables, no tienen que volver a crearse cada vez que se evalúan).

Al construir un Lavaactor, necesitamos inicializar el objeto de manera diferente dependiendo del personaje en el que se basa. La lava dinámica se mueve a su velocidad actual hasta que choca contra un obstáculo. En ese punto, si tiene una reset, volverá a su posición inicial (goteando). Si no lo hace, invertirá su velocidad y continuará en la otra dirección (rebotando).

El create mira el carácter que Level el constructor y crea el actor de lava apropiado.

class Lava {

constructor ( pos , speed , reset ) {

this . pos = pos ;

esto . velocidad = velocidad ;

esto . reiniciar = reiniciar ;

}

get type () { return "lava" ; }

static create ( pos , ch ) {

if ( ch == "=" ) {

return new Lava ( pos , new Vec ( 2 , 0 ));

} else if ( ch == "|" ) {

return new Lava ( pos , new Vec ( 0 , 2 ));

} else if ( ch == "v" ) {

return new Lava ( pos , new Vec ( 0 , 3 ), pos );

}

}

}

Lava . prototipo . tamaño = nuevo Vec ( 1 , 1 );

Coin los actores son relativamente simples. En su mayoría, simplemente se sientan en su lugar. Pero para animar un poco el juego, se les da un "bamboleo", un ligero movimiento vertical hacia adelante y hacia atrás. Para rastrear esto, un objeto de moneda almacena una posición base, así como una wobble que rastrea la fase del movimiento de rebote. Juntos, estos determinan la posición real de la moneda (almacenada en la pos).

class Coin {

constructor ( pos , basePos , wobble ) {

this . pos = pos ;

esto . basePos = basePos ;

esto . bamboleo = bamboleo ;

}

get type () { return "moneda" ; }

static create ( pos ) {

let basePos = pos . más ( nuevo Vec ( 0.2 , 0.1 ));

devolver nueva moneda ( basePos , basePos ,

Math . random () \* Math . PI \* 2 );

}

}

Coin . prototipo . tamaño = nuevo Vec ( 0.6 , 0.6 );

En el [Capítulo 14](https://eloquentjavascript.net/14_dom.html#sin_cos) , vimos que Math.sin nos da la coordenada y de un punto en un círculo. Esa coordenada va y viene en una forma de onda suave a medida que nos movemos a lo largo del círculo, lo que hace que la función seno sea útil para modelar un movimiento ondulado.

Para evitar una situación en la que todas las monedas se muevan hacia arriba y hacia abajo sincrónicamente, la fase inicial de cada moneda es aleatoria. La *fase* de Math.sin la onda de, el ancho de una onda que produce, es 2π. Multiplicamos el valor devuelto por Math.random ese número para darle a la moneda una posición inicial aleatoria en la ola.

Ahora podemos definir el levelChars que asigna los personajes del plan a tipos de cuadrícula de fondo o clases de actores.

const levelChars = {

"." : "vacío" , "#" : "muro" , "+" : "lava" ,

"@" : Jugador , "o" : Moneda ,

"=" : Lava , "|" : Lava , "v" : Lava

};

Eso nos da todas las partes necesarias para crear una Level.

let simpleLevel = new Level ( simpleLevelPlan );

consola . log ( `$ { simpleLevel . width } por $ { simpleLevel . height } ` );

// → 22 por 9

La tarea que tenemos por delante es mostrar dichos niveles en la pantalla y modelar el tiempo y el movimiento dentro de ellos.

## Encapsulación como carga

La mayor parte del código de este capítulo no se preocupa mucho por la encapsulación por dos razones. Primero, la encapsulación requiere un esfuerzo adicional. Hace que los programas sean más grandes y requiere la introducción de conceptos e interfaces adicionales. Dado que hay una cantidad limitada de código que puede arrojar a un lector antes de que sus ojos se pongan vidriosos, he hecho un esfuerzo por mantener el programa pequeño.

En segundo lugar, los diversos elementos de este juego están tan estrechamente vinculados entre sí que si el comportamiento de uno de ellos cambia, es poco probable que cualquiera de los demás pueda permanecer igual. Las interfaces entre los elementos terminarían codificando muchas suposiciones sobre el funcionamiento del juego. Esto los hace mucho menos efectivos: siempre que cambie una parte del sistema, aún debe preocuparse por la forma en que afecta a las otras partes porque sus interfaces no cubrirían la nueva situación.

Algunos *puntos de corte* en un sistema se prestan bien a la separación mediante interfaces rigurosas, pero otros no. Tratar de encapsular algo que no es un límite adecuado es una forma segura de desperdiciar mucha energía. Cuando comete este error, generalmente notará que sus interfaces se vuelven torpemente grandes y detalladas y que deben cambiarse con frecuencia, a medida que el programa evoluciona.

Hay una cosa que nos *vamos a* encapsular, y que es el subsistema de dibujo. La razón de esto es que mostraremos el mismo juego de una manera diferente en el [próximo capítulo](https://eloquentjavascript.net/17_canvas.html#canvasdisplay) . Al colocar el dibujo detrás de una interfaz, podemos cargar el mismo programa de juego allí y conectar un nuevo módulo de visualización.

## Dibujo

La encapsulación del código de dibujo se realiza mediante la definición de un objeto de *visualización* , que muestra un nivel y un estado determinados. El tipo de visualización que definimos se llama DOMDisplay porque usa elementos DOM para mostrar el nivel.

Usaremos una hoja de estilo para establecer los colores reales y otras propiedades fijas de los elementos que componen el juego. También sería posible asignar directamente a la style de los elementos cuando los creamos, pero eso produciría programas más detallados.

La siguiente función auxiliar proporciona una forma sucinta de crear un elemento y darle algunos atributos y nodos secundarios:

función elt ( nombre , atributos , ... hijos ) {

dejar dom = documento . createElement ( nombre );

for ( let attr of Object . keys ( attrs )) {

dom . setAttribute ( atributo , atributos [ atributo ]);

}

para ( dejar hijo de niños ) {

dom . appendChild ( niño );

}

return dom ;

}

Una pantalla se crea dándole un elemento principal al que debe agregarse y un objeto de nivel.

class DOMDisplay {

constructor ( padre , nivel ) {

esto . dom = elt ( "div" , { clase : "juego" }, drawGrid ( nivel ));

esto . actorLayer = null ;

los padres . appendChild ( este . dom );

}

clear () { esto . dom . eliminar (); }

}

La cuadrícula de fondo del nivel, que nunca cambia, se dibuja una vez. Los actores se vuelven a dibujar cada vez que la pantalla se actualiza con un estado determinado. La actorLayer se utilizará para rastrear el elemento que contiene a los actores para que puedan ser removidos y reemplazados fácilmente.

Nuestras coordenadas y tamaños se rastrean en unidades de cuadrícula, donde un tamaño o distancia de 1 significa un bloque de cuadrícula. Al establecer el tamaño de los píxeles, tendremos que aumentar la escala de estas coordenadas: todo en el juego sería ridículamente pequeño, con un solo píxel por cuadrado. La scale constante da el número de píxeles que ocupa una sola unidad en la pantalla.

escala constante = 20 ;

function drawGrid ( nivel ) {

return elt ( "tabla" , {

clase : "fondo" ,

estilo : `ancho: $ { nivel . ancho \* escala } px`

}, ... nivel . filas . mapa ( fila =>

elt ( "tr" , { estilo : `altura: $ { escala } px` },

... fila . mapa (tipo => elt ( "td" , { clase : tipo })))

));

}

Como se mencionó, el fondo se dibuja como un <table>elemento. Esto corresponde muy bien a la estructura de la rows del nivel: cada fila de la cuadrícula se convierte en una fila de tabla ( <tr>elemento). Las cadenas de la cuadrícula se utilizan como nombres de clase para los <td>elementos cell ( ) de la tabla . El operador de extensión (punto triple) se utiliza para pasar matrices de nodos secundarios elt como argumentos separados.

El siguiente CSS hace que la tabla se vea como el fondo que queremos:

.background { background : rgb ( 52 , 166 , 251 );

diseño de mesa : fijo ;

espaciado de bordes : 0 ; }

.background td { relleno : 0 ; }

.lava { fondo : rgb ( 255 , 100 , 100 ); }

.wall { fondo : blanco; }

Algunos de estos ( table-layout, border-spacingy padding) se utilizan para suprimir el comportamiento predeterminado no deseado. No queremos que el diseño de la tabla dependa del contenido de sus celdas, y no queremos espacio entre las celdas de la tabla o relleno dentro de ellas.

La background establece el color de fondo. CSS permite que los colores se especifiquen como palabras ( white) o con un formato como rgb(R, G, B), donde los componentes rojo, verde y azul del color se separan en tres números del 0 al 255. Entonces, en rgb(52, 166, 251), el componente rojo es 52, el verde es 166 y el azul es 251. Dado que el componente azul es el más grande, el color resultante será azulado. Puedes ver que en la .lavaregla, el primer número (rojo) es el más grande.

Dibujamos a cada actor creando un elemento DOM para él y estableciendo la posición y el tamaño de ese elemento en función de las propiedades del actor. Los valores deben multiplicarse por scalepara pasar de unidades de juego a píxeles.

function drawActors ( actores ) {

return elt ( "div" , {}, ... actores . map ( actor => {

let rect = elt ( "div" , { class : `actor $ { actor . type } ` }) ;

rect . style . width = `$ { actor . size . x \* scale } px` ;

rect . estilo . altura = `$ { actor . tamaño . y \* escala } px` ;

rect . estilo . izquierda = `$ { actor . pos . x \* escala } px` ;

rect . estilo . top = `$ { actor . pos . y \* escala } px` ;

return rect ;

}));

}

Para darle a un elemento más de una clase, separamos los nombres de las clases por espacios. En el código CSS que se muestra a continuación, la actorclase da a los actores su posición absoluta. Su nombre de tipo se usa como una clase adicional para darles un color. No tenemos que definir la lavaclase nuevamente porque estamos reutilizando la clase para los cuadrados de la cuadrícula de lava que definimos anteriormente.

.actor { posición : absoluta ; }

.coin { fondo : rgb ( 241 , 229 , 89 ); }

.player { fondo : rgb ( 64 , 64 , 64 ); }

El syncState se utiliza para hacer que la pantalla muestre un estado determinado. Primero elimina los viejos gráficos de los actores, si los hay, y luego vuelve a dibujar a los actores en sus nuevas posiciones. Puede ser tentador intentar reutilizar los elementos DOM para los actores, pero para que eso funcione, necesitaríamos mucha contabilidad adicional para asociar actores con elementos DOM y asegurarnos de eliminar elementos cuando sus actores desaparecen. Dado que normalmente solo habrá un puñado de actores en el juego, redibujarlos a todos no es costoso.

DOMDisplay . prototipo . syncState = function ( state ) {

if ( this . actorLayer ) this . actorLayer . eliminar ();

esto . actorLayer = drawActors ( estado . actores );

esto . dom . appendChild ( este . actorLayer );

esto . dom . className = `juego $ {estado . status } ` ;

esto . scrollPlayerIntoView ( estado );

};

Al agregar el estado actual del nivel como un nombre de clase a la envoltura, podemos diseñar el actor del jugador de manera ligeramente diferente cuando se gana o se pierde el juego agregando una regla de CSS que tiene efecto solo cuando el jugador tiene un elemento ancestro con una clase determinada.

.lost .player {

fondo : rgb ( 160 , 64 , 64 );

}

.won .player {

box-shadow : -4px -7px 8px blanco , 4px -7px 8px blanco ;

}

Después de tocar la lava, el color del jugador se vuelve rojo oscuro, lo que sugiere quemaduras. Cuando se ha recolectado la última moneda, agregamos dos sombras blancas borrosas, una en la parte superior izquierda y otra en la parte superior derecha, para crear un efecto de halo blanco.

No podemos asumir que el nivel siempre encaja en la *ventana* gráfica, el elemento en el que dibujamos el juego. Por eso scrollPlayerIntoView se necesita la llamada. Asegura que si el nivel sobresale fuera de la ventana gráfica, desplazamos esa ventana para asegurarnos de que el jugador está cerca de su centro. El siguiente CSS le da al elemento DOM envolvente del juego un tamaño máximo y asegura que cualquier cosa que sobresalga de la caja del elemento no sea visible. También le damos una posición relativa para que los actores dentro de él estén posicionados en relación con la esquina superior izquierda del nivel.

.game {

desbordamiento : oculto ;

ancho máximo : 600px ;

altura máxima : 450px ;

posición : relativa ;

}

En el scrollPlayerIntoView, encontramos la posición del jugador y actualizamos la posición de desplazamiento del elemento de envoltura. Cambiamos la posición de desplazamiento manipulando las propiedades scrollLefty de ese elemento scrollTop cuando el jugador está demasiado cerca del borde.

DOMDisplay . prototipo . scrollPlayerIntoView = function ( state ) {

let width = this . dom . clientWidth ;

deje que la altura = esto . dom . clientHeight ;

dejar margen = ancho / 3 ;

// La ventana

deja izquierda = esto . dom . scrollLeft , derecha = izquierda + ancho ;

let top = this . dom . scrollTop , bottom = top + height ;

dejar que el jugador = estado . reproductor ;

deja centro = jugador . pos . plus ( jugador . tamaño . veces ( 0,5 ))

. tiempos ( escala );

si ( centro . x < margen + izquierdo ) {

esto . dom . scrollLeft = centro . x - margen ;

} Demás si ( el centro . X > derecho - margen ) {

esto . dom . scrollLeft = centro . x + margen - ancho ;

}

if ( centro . y < margen superior + ) {

esto . dom . scrollTop = centro . y - margen ;

} Demás si ( el centro . Y > inferior - margen ) {

esto . dom . scrollTop = centro . y + margen - altura ;

}

};

La forma en que se encuentra el centro del jugador muestra cómo los métodos de nuestro Vec permiten que los cálculos con objetos se escriban de una manera relativamente legible. Para encontrar el centro del actor, agregamos su posición (su esquina superior izquierda) y la mitad de su tamaño. Ese es el centro en coordenadas de nivel, pero lo necesitamos en coordenadas de píxeles, por lo que luego multiplicamos el vector resultante por nuestra escala de visualización.

A continuación, una serie de comprobaciones verifica que la posición del jugador no esté fuera del rango permitido. Tenga en cuenta que a veces esto establecerá coordenadas de desplazamiento sin sentido que están por debajo de cero o más allá del área de desplazamiento del elemento. Esto está bien, el DOM los restringirá a valores aceptables. Si se establece scroll Leften -10, se convertirá en 0.

Habría sido un poco más sencillo intentar siempre desplazar el reproductor al centro de la ventana gráfica. Pero esto crea un efecto bastante discordante. A medida que saltas, la vista cambiará constantemente hacia arriba y hacia abajo. Es más agradable tener un área “neutral” en el medio de la pantalla donde puede moverse sin causar ningún desplazamiento.

Ahora podemos mostrar nuestro pequeño nivel.

< link rel = "stylesheet" href = "css / game.css" >

< script >

let simpleLevel = new Level ( simpleLevelPlan );

let display = new DOMDisplay ( document . body , simpleLevel );

pantalla . syncState ( State . start ( simpleLevel ));

</ script >

La <link>etiqueta, cuando se usa con rel="stylesheet", es una forma de cargar un archivo CSS en una página. El archivo game.csscontiene los estilos necesarios para nuestro juego.

## Movimiento y colisión

Ahora estamos en el punto en el que podemos comenzar a agregar movimiento, el aspecto más interesante del juego. El enfoque básico, adoptado por la mayoría de juegos como este, es dividir el tiempo en pequeños pasos y, para cada paso, mover a los actores una distancia correspondiente a su velocidad multiplicada por el tamaño del paso de tiempo. Mediremos el tiempo en segundos, por lo que las velocidades se expresan en unidades por segundo.

Mover cosas es fácil. La parte difícil es lidiar con las interacciones entre los elementos. Cuando el jugador golpea una pared o el suelo, no debe simplemente atravesarlo. El juego debe notar cuando un movimiento dado hace que un objeto golpee a otro objeto y responder en consecuencia. Para las paredes, el movimiento debe detenerse. Al golpear una moneda, debe recogerse. Al tocar la lava, el juego debería perderse.

Resolver esto para el caso general es una gran tarea. Puede encontrar bibliotecas, generalmente llamadas *motores de física* , que simulan la interacción entre objetos físicos en dos o tres dimensiones. Adoptaremos un enfoque más modesto en este capítulo, manejando solo colisiones entre objetos rectangulares y manejándolos de una manera bastante simplista.

Antes de mover el reproductor o un bloque de lava, probamos si el movimiento lo llevaría dentro de una pared. Si es así, simplemente cancelamos el movimiento por completo. La respuesta a tal colisión depende del tipo de actor: el jugador se detendrá, mientras que un bloque de lava se recuperará.

Este enfoque requiere que nuestros pasos de tiempo sean bastante pequeños, ya que hará que el movimiento se detenga antes de que los objetos se toquen. Si los pasos de tiempo (y por lo tanto los pasos de movimiento) son demasiado grandes, el jugador terminaría flotando a una distancia notable sobre el suelo. Otro enfoque, posiblemente mejor pero más complicado, sería encontrar el lugar exacto de colisión y moverse allí. Adoptaremos el enfoque simple y ocultaremos sus problemas asegurándonos de que la animación se desarrolle en pequeños pasos.

Este método nos dice si un rectángulo (especificado por una posición y un tamaño) toca un elemento de cuadrícula del tipo dado.

Nivel . prototipo . toca = función ( pos , tamaño , tipo ) {

var xStart = Math . piso ( pos . x );

var xEnd = Math . techo ( pos . x + tamaño . x );

var yStart = Math . piso ( pos . y );

var yEnd = Matemáticas . techo ( pos . y + tamaño . y );

for ( var y = yStart ; y < yEnd ; y ++ ) {

for ( var x = xStart ; x < xEnd ; x ++ ) {

let isOutside = x < 0 | | x > = esto . ancho | |

y < 0 | | y > = esto . altura ;

dejar aquí = ¿está fuera ? "muro" : esto . filas [ y ] [ x ];

si ( aquí == tipo ) devuelve verdadero ;

}

}

devolver falso ;

};

El método calcula el conjunto de cuadrados de la cuadrícula con los que se superpone el cuerpo usando Math.floory Math.ceilen sus coordenadas. Recuerde que los cuadrados de la cuadrícula tienen un tamaño de 1 por 1 unidades. Al redondear los lados de un cuadro hacia arriba y hacia abajo, obtenemos el rango de cuadrados de fondo que toca el cuadro.

Recorrimos el bloque de cuadrados de la cuadrícula que se encuentran redondeando las coordenadas y regresamos true cuando se encuentra un cuadrado coincidente. Los cuadrados fuera del nivel siempre se tratan "wall" para garantizar que el jugador no pueda abandonar el mundo y que no intentemos leer accidentalmente fuera de los límites de nuestra rows.

El update de estado se utiliza touches para averiguar si el jugador está tocando lava.

Estado . prototipo . actualizar = función ( tiempo , claves ) {

dejar actores = esto . actores

. mapa ( actor => actor . actualizar ( hora , esto , claves ));

let newState = new State ( este . nivel , actores , este . estado );

if ( newState . status ! = "playing" ) return newState ;

let player = newState . reproductor ;

si ( este . nivel . toca ( jugador . pos , jugador . tamaño , "lava" )) {

devolver nuevo estado ( este . nivel , actores , "perdido" );

}

para ( deje que el actor de los actores ) {

si ( actor ! = jugador & & se superponen ( actor , jugador )) {

newState = actor . colisionar ( newState );

}

}

return newState ;

};

Al método se le pasa un paso de tiempo y una estructura de datos que le dice qué teclas se mantienen presionadas. Lo primero que hace es llamar al update en todos los actores, produciendo una variedad de actores actualizados. Los actores también obtienen el paso de tiempo, las claves y el estado, para que puedan basar su actualización en ellos. Solo el jugador realmente leerá las teclas, ya que ese es el único actor que está controlado por el teclado.

Si el juego ya terminó, no es necesario realizar ningún procesamiento adicional (el juego no se puede ganar después de perderlo o viceversa). De lo contrario, el método prueba si el jugador está tocando lava de fondo. Si es así, el juego está perdido y terminamos. Finalmente, si el juego realmente continúa, ve si otros actores se superponen al jugador.

La superposición entre actores se detecta con la overlap. Toma dos objetos actor y devuelve verdadero cuando se tocan, que es el caso cuando se superponen tanto a lo largo del eje x como a lo largo del eje y.

superposición de funciones ( actor1 , actor2 ) {

return actor1 . pos . x + actor1 . tamaño . x > actor2 . pos . x & &

actor1 . pos . x < actor2 . pos . x + actor2 . tamaño . x & &

actor1 . pos . y + actor1 .tamaño . y > actor2 . pos . y & &

actor1 . pos . y < actor2 . pos . y + actor2 . tamaño . y ;

}

Si algún actor se superpone, su collide tiene la oportunidad de actualizar el estado. Tocar a un actor de lava establece el estado del juego en "lost". Las monedas se desvanecen cuando las tocas y configura el estado "won"cuando son la última moneda del nivel.

Lava . prototipo . colisionar = función ( estado ) {

devolver nuevo estado ( estado . nivel , estado . actores , "perdido" );

};

Coin . prototipo . colisionar = función ( estado ) {

dejar filtrado = estado . actores . filtro ( a => a ! = esto );

dejar estado = estado . estado ;

si ( ! filtrado . algunos ( a => a . tipo == "moneda" )) estado = "ganado" ;

regreso nuevo estado ( estado . nivel , filtrado , estado );

};

## Actualizaciones de actores

Los update de los objetos actor toman como argumentos el paso de tiempo, el objeto de estado y un keys. El del Lava de actor ignora el keys.

Lava . prototipo . actualizar = función ( tiempo , estado ) {

dejar newPos = esto . pos . más ( esto . velocidad . veces ( tiempo ));

si ( ! Estado . nivel . toques ( NewPOS , este . tamaño , "pared" )) {

vuelven nueva lava ( NewPOS ,esto . velocidad , esto . reiniciar );

} else if ( this . reset ) {

return new Lava ( this . reset , this . speed , this . reset );

} else {

return new Lava ( this . pos , this . speed . times ( - 1 ));

}

};

Este update calcula una nueva posición sumando el producto del paso de tiempo y la velocidad actual a su posición anterior. Si ningún obstáculo bloquea esa nueva posición, se mueve allí. Si hay un obstáculo, el comportamiento depende del tipo de bloque de lava: la lava que gotea tiene una reset a la que salta hacia atrás cuando golpea algo. La lava que rebota invierte su velocidad multiplicándola por -1 para que comience a moverse en la dirección opuesta.

Las monedas usan su update para tambalearse. Ignoran las colisiones con la cuadrícula, ya que simplemente se tambalean dentro de su propio cuadrado.

const wobbleSpeed = 8 , wobbleDist = 0.07 ;

Coin . prototipo . update = function ( time ) {

let wobble = this . wobble + time \* wobbleSpeed ;

deje wobblePos = Math . pecado ( bamboleo ) \* wobbleDist ;

devolver nueva moneda ( this . basePos . plus ( new Vec ( 0 , wobblePos )),

esto . basePos , bamboleo );

};

La wobble se incrementa para rastrear el tiempo y luego se usa como argumento Math.sin para encontrar la nueva posición en la ola. La posición actual de la moneda se calcula a partir de su posición base y un desplazamiento basado en esta onda.

Eso deja al propio jugador. El movimiento del jugador se maneja por separado por eje porque golpear el piso no debe evitar el movimiento horizontal, y golpear una pared no debe detener el movimiento de caída o salto.

const playerXSpeed = 7 ;

gravedad constante = 30 ;

const jumpSpeed = 17 ;

Jugador . prototipo . actualizar = función ( tiempo , estado , claves ) {

let xSpeed = 0 ;

if ( teclas . ArrowLeft ) xSpeed - = playerXSpeed ;

if ( teclas . ArrowRight ) xSpeed + = playerXSpeed ;

deje pos = esto . pos ;

let moveX = pos .más ( new Vec ( xSpeed \* time , 0 ));

if ( ! estado . nivel . toca ( movidoX , este . tamaño , "muro" )) {

pos = movidoX ;

}

deje ySpeed = this . velocidad . y + tiempo \* gravedad ;

let moveY = pos . plus ( new Vec ( 0 , ySpeed \* time ));

if ( ! estado . nivel . toca ( movidoY , este . tamaño , "pared" )) {

pos = movidoY ;

} else if ( teclas . ArrowUp & & ySpeed > 0 ) {

ySpeed = - jumpSpeed ;

} más {

ySpeed = 0 ;

}

return new Player ( pos , new Vec ( xSpeed , ySpeed ));

};

El movimiento horizontal se calcula basándose en el estado de las teclas de flecha izquierda y derecha. Cuando no hay una pared que bloquee la nueva posición creada por este movimiento, se utiliza. De lo contrario, se mantiene la posición anterior.

El movimiento vertical funciona de manera similar, pero debe simular el salto y la gravedad. La velocidad vertical del jugador ( ySpeed) se acelera primero para tener en cuenta la gravedad.

Comprobamos las paredes de nuevo. Si no acertamos a ninguno, se usa la nueva posición. Si no *es* una pared, hay dos resultados posibles. Cuando se presiona la flecha hacia arriba *y* nos estamos moviendo hacia abajo (lo que significa que lo que golpeamos está debajo de nosotros), la velocidad se establece en un valor negativo relativamente grande. Esto hace que el jugador salte. Si ese no es el caso, el jugador simplemente chocó contra algo y la velocidad se establece en cero.

La fuerza de la gravedad, la velocidad de salto y casi todas las demás constantes de este juego se han establecido mediante prueba y error. Probé valores hasta que encontré una combinación que me gustó.

## Seguimiento de claves

Para un juego como este, no queremos que las teclas surtan efecto una vez por pulsación de tecla. Más bien, queremos que su efecto (mover la figura del jugador) permanezca activo mientras se mantenga.

Necesitamos configurar un controlador de claves que almacene el estado actual de las teclas de flecha izquierda, derecha y arriba. También querremos pedir preventDefault esas teclas para que no terminen desplazándose por la página.

La siguiente función, cuando se le da una matriz de nombres de claves, devolverá un objeto que rastrea la posición actual de esas claves. Registra controladores de eventos "keydown"y "keyup" y, cuando el código clave del evento está presente en el conjunto de códigos que está rastreando, actualiza el objeto.

función trackKeys ( teclas ) {

permiten abajo = Objeto . crear ( nulo );

función pista ( evento ) {

si ( teclas . incluye ( evento . tecla )) {

abajo [ evento . clave ] = evento . type == "keydown" ;

evento . preventDefault ();

}

}

ventana . addEventListener ( "keydown" , pista );

ventana . addEventListener ( "keyup" , pista );

volver abajo ;

}

const arrowKeys =

trackKeys ([ "ArrowLeft" , "ArrowRight" , "ArrowUp" ]);

Se utiliza la misma función de controlador para ambos tipos de eventos. Examina la typepropiedad del objeto de evento para determinar si el estado de la clave debe actualizarse a verdadero ( "keydown") o falso ( "keyup").

**Ejecutando el juego**

La requestAnimationFrame, , proporciona una buena forma de animar un juego. Pero su interfaz es bastante primitiva: usarla requiere que rastreemos el momento en el que se llamó a nuestra función la última vez y que volvamos requestAnimationFrame a llamar después de cada fotograma.

Definamos una función auxiliar que envuelva esas partes aburridas en una interfaz conveniente y nos permita simplemente llamar runAnimation, dándole una función que espera una diferencia de tiempo como argumento y dibuja un solo marco. Cuando la función de marco devuelve el valor false, la animación se detiene.

función runAnimation ( frameFunc ) {

let lastTime = null ;

marco de función ( tiempo ) {

if ( lastTime ! = null ) {

let timeStep = Math . min ( tiempo - lastTime , 100 ) / 1000 ;

if ( frameFunc ( timeStep ) === falso ) return ;

}

lastTime = tiempo ;

requestAnimationFrame ( marco );

}

requestAnimationFrame ( marco );

}

He establecido un paso de fotograma máximo de 100 milisegundos (una décima de segundo). Cuando la pestaña o ventana del navegador con nuestra página está oculta, las requestAnimationFrame llamadas se suspenderán hasta que la pestaña o ventana se muestre nuevamente. En este caso, la diferencia entre lastTimey time será todo el tiempo que estuvo oculta la página. Avanzar en el juego tanto en un solo paso parecería tonto y podría causar efectos secundarios extraños, como que el jugador se caiga al suelo.

La función también convierte los pasos de tiempo en segundos, que son una cantidad más fácil de pensar que milisegundos.

La runLevel toma un Level y un constructor de visualización y devuelve una promesa. Muestra el nivel (en document.body) y permite al usuario jugar a través de él. Cuando el nivel finaliza (perdido o ganado), runLevel espera un segundo más (para que el usuario vea lo que sucede) y luego borra la pantalla, detiene la animación y resuelve la promesa del estado final del juego.

función runLevel ( nivel , visualización ) {

dejar visualización = nueva visualización ( documento . cuerpo , nivel );

let state = State . inicio ( nivel );

deje que el final = 1 ;

return new Promise ( resolve => {

runAnimation ( time => {

state = state . update( hora , teclas de flecha );

pantalla . syncState ( estado );

if ( estado . estado == "jugando" ) {

devolver verdadero ;

} más si ( final > 0 ) {

final - = tiempo ;

devuelve verdadero ;

} else {

display . claro ();

resolver ( estado . estado );

devolver falso ;

}

});

});

}

Un juego es una secuencia de niveles. Siempre que el jugador muere, se reinicia el nivel actual. Cuando se completa un nivel, pasamos al siguiente nivel. Esto se puede expresar mediante la siguiente función, que toma una matriz de planos de nivel (cadenas) y un constructor de visualización:

async función runGame ( planes , Display ) {

para ( dejar nivel = 0 ; nivel < planes . longitud ;) {

dejó estado = Aguarde nivel de ejecución ( nuevo Nivel ( planes [ nivel ]),

Display );

if ( estado == "ganó" ) nivel ++ ;

}

consola . log ( "¡Has ganado!" );

}

Debido a que hicimos runLevelreturn una promesa, runGamese puede escribir usando una async. Devuelve otra promesa, que se resuelve cuando el jugador termina el juego.

Hay un conjunto de planes de nivel disponibles en el GAME\_LEVELSenlace de [la caja de arena de este capítulo](https://eloquentjavascript.net/code#16) . Esta página los alimenta para runGameiniciar un juego real.

< link rel = "stylesheet" href = "css / game.css" >

< cuerpo >

< secuencia de comandos >

runGame ( GAME\_LEVELS , DOMDisplay );

</ script >

</ body >

Vea si puede vencerlos. Me divertí mucho construyéndolos.

**Ejercicios**

**Juego terminado**

Es tradicional que los juegos de plataformas hagan que el jugador comience con un número limitado de *vidas* y reste una vida cada vez que muere. Cuando el jugador se queda sin vidas, el juego se reinicia desde el principio.

Ajústese runGamepara implementar vidas. Haga que el jugador comience con tres. Muestra el número actual de vidas (usando console.log) cada vez que comienza un nivel.

< link rel = "stylesheet" href = "css / game.css" >

< cuerpo >

< script >

// La antigua función runGame. Modificarlo ...

asíncrono función runGame ( planes , Display ) {

a ( dejar de nivel = 0 ; nivel < planes . De longitud ;) {

deje de estado = Aguarde nivel de ejecución ( nueva Nivel ( planes [ nivel ]),

Display );

si ( estado == "ganó" ) nivel ++ ;

}

consola . log ( "¡Has ganado!" );

}

runGame ( GAME\_LEVELS , DOMDisplay );

</ script >

</ body >

**Pausando el juego**

Haz posible pausar (suspender) y reanudar el juego presionando la tecla Esc.

Esto se puede hacer cambiando la runLevel para usar otro controlador de eventos de teclado e interrumpiendo o reanudando la animación cada vez que se presiona la tecla Esc.

Es runAnimation posible que la interfaz no parezca adecuada para esto a primera vista, pero lo es si reorganiza la forma en que la runLevelllama.

Cuando tenga eso funcionando, hay algo más que podría intentar. La forma en que hemos registrado los controladores de eventos del teclado es algo problemática. El arrowKeysobjeto es actualmente un enlace global y sus controladores de eventos se mantienen incluso cuando no se está ejecutando ningún juego. Se podría decir que se *filtran* fuera de nuestro sistema. Amplíe trackKeyspara proporcionar una manera de anular el registro de sus controladores y luego cambie runLevelpara registrar sus controladores cuando se inicie y anular el registro de nuevo cuando finalice.

< link rel = "stylesheet" href = "css / game.css" >

< cuerpo >

< script >

// La antigua función runLevel. Modifique esto ...

función runLevel ( nivel , Visualización ) {

dejar visualización = nueva Visualización ( documento . Cuerpo , nivel );

let state = State . inicio ( nivel );

deje que el final = 1 ;

devolver nueva promesa ( resolver => {

runAnimation ( time => {

state = state . update ( time , arrowKeys );

display . syncState ( state );

if ( state . status == "playing" ) {

return true ;

} más si ( final > 0 ) {

final - = tiempo ;

devuelve verdadero ;

} else {

display . claro ();

resolver ( estado . estado );

devolver falso ;

}

});

});

}

runGame ( GAME\_LEVELS , DOMDisplay );

</ script >

</ body >

**Un monstruo**

Es tradicional que los juegos de plataformas tengan enemigos a los que puedes saltar para derrotarlos. Este ejercicio le pide que agregue un tipo de actor de este tipo al juego.

Lo llamaremos monstruo. Los monstruos se mueven solo horizontalmente. Puede hacer que se muevan en la dirección del jugador, rebotar hacia adelante y hacia atrás como lava horizontal o tener cualquier patrón de movimiento que desee. La clase no tiene que manejar la caída, pero debe asegurarse de que el monstruo no atraviese las paredes.

Cuando un monstruo toca al jugador, el efecto depende de si el jugador está saltando sobre él o no. Puede aproximarse a esto comprobando si la parte inferior del jugador está cerca de la parte superior del monstruo. Si este es el caso, el monstruo desaparece. Si no, el juego está perdido.

< link rel = "stylesheet" href = "css / game.css" >

< style > .monster { background : purple } </ style >

< cuerpo >

< script >

// Completa los métodos constructor, actualización y colisión

class Monster {

constructor ( pos , / \* ... \* / ) {}

get type () { return "monstruo" ; }

static create ( pos ) {

return new Monster ( pos . plus ( new Vec ( 0 , - 1 )));

}

actualizar ( hora , estado ) {}

colisionar ( estado ) {}

}

Monster . prototipo . tamaño = nuevo Vec ( 1.2 , 2 );

levelChars [ "M" ] = Monstruo ;

runLevel ( nuevo nivel ( `

...................................

######### #######################

.. # ........................ ...... #..

# .............................. #

.. # ...... ........................ #

.. # ...................... ..... o .. #.. #

.. @ ........................... #

.. ##### ##### .............. ########.

.......... # .. o..o..o .. o .. # ........

.......... # ........... M .. # ........

.... ...... ################ ........

.................... ..............

` ), DOMDisplay );

</ script >

</ body >