Oddicey

por RGV



Enciclopedia

de diseño

Índice

[Control de versiones 4](#_Toc478827446)

[Iteraciones del documento 4](#_Toc478827447)

[Diseño 5](#_Toc478827448)

[Actores 5](#_Toc478827449)

[Dado *(Die)* 5](#_Toc478827450)

[Objetos / Obstáculos 5](#_Toc478827451)

[Huellas *(Marks)* 5](#_Toc478827452)

[Huella básica *(Mark)* 5](#_Toc478827453)

[Huella durmiente *(Sleepy Mark)* 5](#_Toc478827454)

[Huella perezosa *(Lazy Mark)* 5](#_Toc478827455)

[Teletransportadores *(Teleporters)* 6](#_Toc478827456)

[Cabina de teletransporte *(Teleporter Cabin)* 6](#_Toc478827457)

[Salida de teletransporte *(Teleporter Output)* 6](#_Toc478827458)

[Sumidero de teletransporte *(Teleporter Sink)* 6](#_Toc478827459)

[Teletransportador básico *(Teleporter)* 6](#_Toc478827460)

[Teletransportador dicotómico *(Teleporter)* 6](#_Toc478827461)

[Teletransportador complejo*(Teleporter)* 7](#_Toc478827462)

[Veletas *(Vanes)* 7](#_Toc478827463)

[Veleta básica *(Vane)* 7](#_Toc478827464)

[Veleta intermitente *(Flipping Vane)* 7](#_Toc478827465)

[Poste *(Post)* 7](#_Toc478827466)

[Apartado técnico 8](#_Toc478827467)

[Unidades 8](#_Toc478827468)

[Desarrollo 9](#_Toc478827469)

[Componentes 9](#_Toc478827470)

[Jugador (Player) 9](#_Toc478827471)

[Cubo rodante *(RollingCube)* 9](#_Toc478827472)

[Prefabricados 9](#_Toc478827473)

[Dado *(Die)* 9](#_Toc478827474)

Control de versiones

|  |  |  |
| --- | --- | --- |
| Iteraciones del documento | | |
| Versión | Fecha | Descripción |
| 1.0 | 20/03/17 | Inicio del proyecto (esqueleto del documento) |
| 1.1 | 23/03/17 | Iteraciones #0 y #1 |
| 1.2 | 25/03/17 | Iteraciones #2 y #3 |
| 1.3 | 29/03/17 | Iteraciones #4 y #5 |
| 1.4 | 01/04/17 | Repaso de iteración #5 e iteración #6 |
| 1.5 | 04/05/17 | Hasta el momento previo a la iteración #8 (prototipo) |
| 1.6 | 08/01/18 | Iteración #8 |

Apartado técnico

# Unidades

## Dimensiones por defecto

Se asume 1080p como la resolución por defecto de los dispositivos objetivo. Las modificaciones de relación de aspecto y resolución se harán con base en cambios desde 1080p y no al contrario.

## Casillas

La unidad atómica del juego es de 4 unidades de longitud de Unity. Cada cuatro unidades componen una casilla. Estas casillas se representan por la posición de su centro en el eje horizontal.

La casilla origen, que sería la ubicada en el centro de coordenadas, tendría su centro en x=0, y=0 (y z=0, aunque sea indistinto). Por lo tanto abarcaría desde (-2,0) hasta (2,0), y no desde (0,0) hasta (4,0) como podría pensarse.

Esto puede entenderse como que las propias casillas tienen el punto de pivote en su centro geométrico y no en su extremo inferior izquierdo, lo cual suena mucho más intuitivo.

Disponer de tal forma las casillas permite aliviar mucho trabajo a la hora de colocar elementos en el diseño de niveles, ya que basta con posicionarlos en los múltiplos de las unidades de longitud de una casilla. Si el pivote de las casillas no estuviese en el centro, habría que realizar el ajuste, y la edición de niveles sería mucho menos natural.

## Recursos gráficos de actores

Los actores son todos los elementos que toman partida en el juego. No son recursos de actores los fondos, por ejemplo.

La casilla se representará con una cuadrícula de *400∙400* píxeles. Esto no quiere decir que los recursos tengan que ceñirse a ese tamaño, sino que tomarán tales casillas como referencia.

Todos los archivos de recursos gráficos deben tener un tamaño múltiplo del tamaño base. Es decir, cada recurso gráfico tendrá un archivo de *n∙n* casillas, que es equivalente a *400n∙400n* píxeles.

Además, todos estos recursos tomarán una casilla base, que representa el lugar por el que el dado del jugador pasará rodando.

De nuevo esto no significa que el recurso tenga que ceñirse a esos límites. Sólo son referencias.

Además de ello, los recursos gráficos tienen que tener en cuenta dónde serán posicionados sus pivotes. No todos los recursos pueden mantenerlos en el centro. Dónde se posiciona el pivote viene relacionado con dónde está la casilla base por la que el personaje rodará.

Los tamaños de los recursos gráficos se notificarán según el archivo, no según el recurso en sí.

Esto es, si un recurso ocupa sólo un poco más que *1∙1* casillas (por ejemplo, una plataforma de *1∙1* cuya hierba sobresale sin embargo por los cuatro lados), su tamaño será de *3∙3* casillas, aunque gran parte de la superficie esté vacía.

## Recursos gráficos de no actores

Los recursos gráficos de no actores, como los involucrados en el fondo, no tienen por qué seguir las restricciones anteriores. Es más lógico ajustar sus tamaños al tamaño por defecto de pantalla, que se asume como 1080p, que hacerlos múltiplos o submúltiplos del personaje principal, como si fueran un actor más.

En este punto hay mucha más libertad de tamaños que en los estrictos casos de actores, donde todo tiene que cuadrar perfectamente con todo para el correcto funcionamiento de las físicas, los disparadores, etc.

Por ejemplo, el diseño de una nube podría partir de 1080p y después ser recortado a gusto.

# Posicionamiento

## Personaje principal

El protagonista partirá desde la casilla origen. Esto será inmutable, independientemente de que el nivel vertical en que el protagonista comienza sea muy diferente al que después ocupa durante el escenario. Comenzar en el origen siempre ayuda a tener en cualquier momento noción del espacio que ocupa el protagonista, tanto en la altitud (nivel vertical, ya mencionado), como en el avance horizontal —así, para saber cuánto ha avanzado el personaje sólo hace falta ver su casilla actual—.

## Cámara

La cámara situará por defecto al personaje principal desplazado hacia abajo y hacia la izquierda. Dado que la cámara es ortográfica, lo más sencillo es hacerla encajar con un cierto número de casillas en el eje vertical (ya que el tamaño de una cámara ortográfica en Unity indica su distancia desde el centro hasta la arista superior). Actualmente la cámara ortográfica tiene un tamaño de 16 unidades; esto es, cuatro casillas desde el centro hasta el extremo superior, y por tanto otras cuatro hasta el inferior.

Es un problema fijar el tamaño en el eje vertical por lo siguiente: para no dar ventaja o desventaja a un jugador en función de su relación de aspecto, las distancias en el eje horizontal deben ser independientes a la pantalla en la que se juega; no obstante, si el eje vertical debe tener ocho casillas por decreto, esto significa que ante diferentes relaciones de aspecto no se tendrá la misma cantidad de espacio visible hacia izquierda y derecha.

Actualmente no se ha resuelto esto. Posibles soluciones serían:

* Corregir la relación de aspecto (poco elegante): añadir bandas negras en los laterales es una solución muy poco popular, por motivos evidentes. Distorsionar el aspecto del juego es intolerable.
* Mover el personaje principal respecto a la cámara en función de la relación de aspecto: solución más intuitiva. El dado, en este caso, se posicionará siempre tal que desde él al final de la pantalla haya un número de casillas invariable. Es una solución parcial, ya que pese a que la distancia sea la misma, el jugador tendrá más visión de conjunto del escenario si su relación de aspecto le juega a favor.
* Queda fuera de concurso la solución habitual de este problema: piénsese en ejemplos como los juegos de la saga *Plantas contra Zombis*, que resuelven muy elegantemente la diferencia de relación de aspectos. Dado que tanto escenario como actores aliados son estáticos, los desarrolladores sólo necesitan ajustar la cantidad de espacio que el jugador ve desde la posición de sus personajes —plantas que coloca— hasta el punto desde el que llegan los enemigos —rumbo de los zombis—. Ante más espacio (más resolución, o relación de aspecto favorable), los desarrolladores sólo muestran más escenario por los lugares por los que no se da juego efectivo; esto es: los laterales, que al estar fuera de las casillas no sirven para nada, y el extremo contrario al rumbo de los enemigos, que al estar detrás del objetivo de los enemigos, tampoco entra en juego. Seguir esta solución en un escenario dinámico como el que ocupa deviene en la solución anterior.

No hay que olvidar que el personaje principal parte, forzosamente, del origen de coordenadas. Esto significa que es la cámara la que desfasa su ubicación inicial para ajustar el dado en su perspectiva.

La cámara actualmente se ubica en *16∙6* unidades respecto al dado. Dicho de otro modo, el dado está situado cuatro casillas a la izquierda del centro de la cámara, y en la tercera casilla empezando por la parte inferior de la pantalla.

Esta debe ser la ubicación estándar del dado, pero no es fija. Habrá situaciones, como el final del nivel, en que entre la cámara y el dado haya un desfase distinto. En el final del nivel, porque la cámara, al detectar el disparador *EOL* *(End of Level)*, dejará de moverse para que el dado llegue al final del nivel coincidiendo con el extremo derecho de la cámara. Un efecto resultón y que ayuda a evitar recursos ornamentales tras el final del nivel.

También se podrán dar estos desfases en el eje vertical en función de la situación (por determinar).

Diseño

# Personajes

## Dado *(Die)*

El dado es el personaje controlable por el jugador. Por sí solo avanza rodando al ritmo de la música, y permite durante ese avance que el jugador le cambie su cara activa.

En cada rodado, el jugador puede hacer una o ninguna rotaciones de cara. El cambio de cara se puede hacer cuando el dado está terminando su rodado, pero no cuando lo está empezando.

# Objetos / Obstáculos

## Huellas *(Marks)*

Obstáculo base. Muestra una marca con el dibujo de una cara determinada del dado.

Su mecánica consiste en que el jugador active la cara requerida. Según el tipo de huella, el momento de activar esta cara requerida será distinto, así como su nivel de restricción.

Cada huella puede estar en dos estados: estado de acierto y estado de error. Estos estados son reforzados gráficamente (y quizá auditivamente también), para que el jugador sepa qué estado tiene una huella en todo momento. Además de esos estados, una vez la huella es “activada” comprueba si el jugador consiguió éxito o fracaso. Dependiendo del contexto de la huella, como se verá más adelante, el jugador puede buscar una u otra condición, pero por defecto la condición de éxito proporciona al jugador un efecto positivo, y un efecto negativo es lanzado si se fracasa. También estas dos condiciones son reforzadas. No obstante, el hecho del éxito o fracaso en principio está directamente relacionado con el estado de acierto o error. Si el jugador activa la huella cuando esta está en estado de acierto, el resultado de la huella será de éxito; si la huella es por el contrario activada cuando está en estado de error, el resultado será de fracaso.

Se ha propuesto que las huellas estén encubiertas estéticamente para imbuirlas con mayor naturalidad en la temática. Dado que existen dados animalizados como vacas, son un buen recurso para representar el obstáculo de las huellas. Podrían, por ejemplo, dar coces en caso de fracaso o dormirse en estado de acierto. Las pilas de huellas pasarían a ser vacas apiladas, “rebaños” por ejemplo, y bien animadas ofrecerían una vista muy resultona.

Dado que en el alcance original incluye más de una temática y no sólo los valles —véase la fábrica de dados, por ejemplo—, donde estarían ubicadas las vacas, esta propuesta se queda por el momento en eso mismo.

Además, el objetivo principal de las vacas sería reducir la inevitable disonancia del juego. Si bien la disminuiría, otros actores como los portales o los transportadores seguirían haciéndola palpable. Evitarla por completo supondría modificar estéticamente todos estos actores (los transportadores podrían ser abejas que polinizan diferentes flores), y ello añadiría mucho coste al apartado gráfico.

## Huella básica *(Mark)*

La huella básica es la huella por antonomasia. Cuando el jugador rota la cara del dado, la huella reacciona. Si la nueva cara activa es correcta, la huella activa o mantiene su estado de acierto; si la nueva cara activa es incorrecta, la huella activa o mantiene su estado de error.

## Huella durmiente *(Sleepy Mark)*

Tipo de huella más restrictivo que la huella básica. Cuando el jugador activa la cara requerida, la huella se activa durante un único pulso. Esto hace que el jugador se vea obligado a activar la cara requerida justo en el momento en que va a pisar la huella, ya que si lo hace antes, en el siguiente pulso la huella volverá a apagarse (“dormirse”).

## Huella perezosa *(Lazy Mark)*

Tipo de huella más restrictivo que la huella básica, pero menos que la huella durmiente. Cuando el jugador activa la cara requerida, la huella comienza a activarse (“se despereza”), pero no será hasta el siguiente pulso musical cuando de verdad se active. Esto hace que el jugador se vea obligado a activar la cara requerida como mínimo con una casilla entre su posición y la de la huella, ya que si lo hace después la huella se estará desperezando y el jugador fracasará.

## Teletransportadores *(Teleporters)*

Un teletransportador es un obstáculo compuesto.

Se compone por una cabina de teletransporte, una o más salidas de teletransporte y un o ningún sumidero de teletransporte, que es una salida de teletransporte especial.

En función de los elementos que lo compongan, el teletransportador será de un tipo u otro, aunque en realidad todos mantienen la misma mecánica. Esta mecánica consiste en que el jugador, al llegar a la cabina del teletransportador, sea o no trasladado a alguna salida de teletransporte. Se desgranarán por tanto los diferentes elementes del teletransportador para después combinarlos en distintos tipos.

## Cabina de teletransporte *(Teleporter Cabin)*

La cabina de teletransporte siempre compone un teletransportador. Cuando el jugador alcanza una cabina de teletransporte, las salidas de ese teletransportador son activadas (no confundir que una salida sea activada con que estuviese ya en funcionamiento. Ver siguiente apartado). Por sí sola, la cabina de teletransporte no hace nada.

## Salida de teletransporte *(Teleporter Output)*

Una salida de teletransporte contiene una huella. La huella de una salida de teletransporte nunca coincidirá con la huella de otra salida de ese mismo teletransportador, si la tiene. Mejor dicho, el estado de las salidas de un mismo teletransportador (el estado de sus huellas) no coincidirán en el instante en que el jugador vaya a llegar a la cabina que las activará.

Esto es debido a que el jugador elegirá a qué salida de teletransporte viajar mediante sus huellas. Es decir, cuando el personaje llega a la cabina, viajará a la única salida de teletransporte cuya huella esté en estado de acierto. En definitiva, en todo aquel momento en que el jugador vaya a entrar en la cabina de teletransporte, las huellas tienen que estar en un estado unívoco; sólo una de las huellas puede devolver éxito.

Las huellas de las salidas de teletransporte tienen su funcionamiento habitual. Mantienen un estado de acierto o de error y, una vez son activadas, avisan de éxito o fracaso.

## Sumidero de teletransporte *(Teleporter Sink)*

Un sumidero de teletransporte es un tipo particular de salida de teletransporte. Esta salida no contiene huella. El jugador será transportado al sumidero si ninguna de las salidas de teletransporte se encuentra en estado de acierto.

Un teletransportador no tiene por qué tener sumidero, y en caso de tenerlo evidentemente sólo puede tener uno.

Cuando un teletransportador no tiene sumidero y ninguna de sus huellas es activada en éxito, el dado sigue su camino como si no hubiese pasado por un teletransportador.

## Teletransportador simple *(Teleporter)*

El teletransportador simple o básico es el tipo unario de teletransportador. Lo componen una cabina y una salida. Puede o no tener sumidero. De tenerlo, su mecánica no es tan básica como en el caso de que no lo tenga, por lo dicho en el apartado anterior.

La función más inmediata del teletransportador unario es cambiar de posición al jugador, sin más. No es sino una forma de mover convenientemente al jugador encubierta en una huella. En cualquier caso, las posibilidades mecánicas son bastante amplias jugando con el no tener que transportarse o incluso tener como objetivo la salida sumidero.

## Teletransportador dicotómico *(Teleporter)*

El teletransportador dicotómico es el tipo binario de teletransportador. Lo componen una cabina y dos salidas, con o sin sumidero.

Su mecánica más básica es la de confrontar al jugador con una bifurcación de su camino. Por supuesto si entra en juego el sumidero también se puede jugar con él como salida posible.

En este punto, aunque también en el teletransportador unario podría darse, surge la posibilidad de que la cabina del teletransportador esté situada a continuación de alguna salida, por lo que incurrir en esa salida devolvería al jugador a una posición por la que ya ha pasado y podría generarse un bucle interesante.

## Teletransportador complejo*(Teleporter)*

Es teletransportador complejo cualquier teletransportador *n-*ario. Simplemente se trata de una generalización del dicotómico. Si las huellas de sus salidas no son básicas, quizá sea demasiado difícil mantenerlas unívocas.

## Veletas *(Vanes)*

La veleta, o girador, es un obstáculo muy sencillo. Cuando el personaje lo alcanza, el sentido de su dirección puede o no ser invertido en función de la veleta, que “señala” hacia un lugar. El sentido hacia el que mira el girador puede ser el mismo hacia el que el jugador avanza o su opuesto. Para llevarlo a la particularidad de este diseño, esta veleta puede estar apuntando hacia la derecha o hacia la izquierda. Si el dado del personaje llega desde el sentido opuesto al que la veleta del girador apunta, el sentido del dado es opuesto. Sin embargo, si el sentido del dado coincide con el de la veleta, el dado continuará su marcha.

## Veleta básica *(Vane)*

El girador más básico simplemente apunta hacia el sentido contrario que el dado lleva al llegar a él. Cuando el dado llega, cambia el sentido de su marcha. Como caso más general, el girador estará mirando hacia la izquierda y el dado llegará desde ese lado avanzando hacia la derecha, y al toparse con el girador invertirá el sentido de su marcha y volverá por donde ha venido, hacia la izquierda.

## Veleta intermitente *(Flipping Vane)*

No obstante, el girador básico no justifica la definición que se ha dado de los giradores, ya que ante un girador básico nunca sucederá que su sentido y el del dado coincidan y por tanto el dado siga su marcha.

Tipos no básicos de veletas, como el girador intermitente, evidencian la definición genérica de giradores. El girador intermitente cambiará el sentido hacia el que apunta a cada pulso musical. Que cambie o no el sentido del dado dependerá de cuándo este lo alcance.

## Veleta anclada *(StaticVane)*

A fin de facilitar visualmente la inferencia al jugador, esta veleta jamás cambia su sentido, independientemente de que lo haga el dado. Como caso general tendrá el mismo valor que una veleta básica, pero aliviará el movimiento de actores en pantalla. Además, ciertas combinaciones dinámicas podrían requerirla.

## Poste *(Post)*

Los objetos postes tienen la función de punto de guardado. Sirven como *checkpoint*. Cuando el jugador alcanza la casilla en la que se ubica el poste, este pasa a ser su nuevo punto de guardado, al que volverá cuando muera (si debe volver y no terminar la partida, a falta de determinar la consecuencia ante un fallo del jugador).

## Portales *(Portals)*

Un portal es un objeto compuesto, pero de mucha menos complejidad que los teletransportadores. Parte de la misma idea que un teletransportador simple sin sumidero: existe un lugar desde el que se transporta al jugador a otro. En el caso del teletransportador este lugar era la cabina, y en el del portal es el portal en sí mismo. Las salidas del teletransportador eran las que tenían peso sobre la decisión mecánica del jugador, pero en el caso del portal son meras posiciones que marcan dónde será llevado el jugador.

Los portales se justifican respecto a los teletransportadores simples sin sumidero por dos cuestiones:

* Puede suceder que se quiera transportar al jugador a un lugar que no aparece en pantalla. Por tanto, la mecánica de los teletransportadores, que tienen las huellas objetivo en las salidas, no sirve.
* Los portales, bien escondidos o complejos de abrir, pueden llevar a zonas alternativas de los niveles.

Generalmente los portales requerirán superar una pila de huellas para ser abiertos (aunque no tiene por qué ser necesario ni por qué ser ese el obstáculo. Actualmente, tal como se encuentra el diseño, el portal tendrá junto a sí una pila de huellas y se abrirá al completarse esta o no tendrá nada y estará directamente abierto).

## Pila de huellas *(MarksPile)*

Una pila de huellas es, como su nombre indica, un conjunto de huellas apiladas, de manera que para ser superado tienen que serlo todas ellas una a una.

Una huella desaparecerá de una pila de huellas al entrar en estado de acierto.

Una pila de huellas considerará éxito si no mantiene huellas no superadas al llegar a ella el jugador.

Esta mecánica implica que no todas las huellas sean susceptibles de pertenecer a una pila. La huella básica puede ser la única huella miembro de una pila. Podría serlo también la huella perezosa, pero quizá el espacio en pantalla haga de esto una dificultad, ya que cada huella perezosa implica un mínimo de tres casillas de distancia. Por otro lado, la huella durmiente ni siquiera podría pertenecer a una pila a no ser que fuese la última y la condición de éxito se comprobase en una casilla posterior a la que la pila ocupa.

Desarrollo

# Ordenación

## Capas de orden *(Sorting Layers)*

* **Background**: no actores; decorado. Segundo plano.
* **Floor:** plataformas.
* **Mark:** marcas de obstáculos.
* **Player:** el dado y sus animaciones.
* **Editor:** elementos que no participan en el juego, como la rejilla.

Mayor prioridad a elementos situados más abajo en la enumeración.

## Capas de renderizado *(Layer)*

Sin uso.

## Etiquetas *(Tags)*

* **Player:** dado. Es etiqueta por defecto, pero se aprovecha para este uso.
* **Mark:** huellas, para colisiones con el dado.
* **NotFloor:** para los disparadores de final de plataforma *EOP (End of Platform).* Autocontenidos en los prefabricados de las plataformas, como elementos hijo de los extremos. Indican al dado cuándo ha perdido suelo y debe caer.
* **Floor:** casillas unitarias de las plataformas. Indica al dado cuándo ha ganado suelo o sigue en él, y por tanto no debe seguir cayendo ni empezar a hacerlo.
* **Boundary:** mánager y otros elementos de gestión, como algunos disparadores (por ejemplo *End of Level)*. Anteriormente usada la etiqueta por defecto “EditorOnly”.

# Componentes

## Jugador *(Player)*

Puede ser **teletransportado**. La teletransportación será llevada a cabo cuando su cubo rodante termine la rotación actual.

También gestiona si el personaje **pierde o gana suelo**, y le manda al notificador propagar tal cambio de estado.

## Cubo rodante *(RollingCube)*

Al despertar **busca un componente colisionador** para establecer sus dimensiones.

Cuando su función pública de rodado es llamada, llama a una de sus dos funciones privadas de rodado según su dirección, y estas preparan el pivote y llaman a la corrutina.

Si cuando se va a rodar ya se está rodando se lanza a la consola un error, para saber que un pulso ha sido perdido y que esto no debería ocurrir jamás.

La corrutina dura un poco menos de lo que debería para asegurar que no se perderá el siguiente pulso bajo ningún concepto. Un error hacía que en ocasiones la corrutina no terminase a tiempo, por lo que esto se evita reduciendo su tiempo con base en una constante de error (*error = 0.05f* significa que la corrutina durará un 95% de lo que duran los pulsos).

## Fondo estático *(StaticBackground)*

**Requiere** que el objeto sea hijo de la cámara principal. Esto evita tener que gestionar el avance, ya que la cámara sigue al jugador por defecto. También, como la cámara se frena al llegar al final del nivel y en otras circunstancias determinadas, el fondo lo hará por ser hijo de ella.

Dado el diseño de su algoritmo, **requiere** que la imagen se posicione con respecto a la cámara tan a la derecha como se quiera que comience. Es decir, al comenzar la ejecución el fondo se verá tal como estaba ubicado antes, y a partir de ahí se desplazará en función del movimiento del dado.

En definitiva: en condiciones habituales la imagen del fondo tendrá el mínimo margen izquierdo posible (aunque habrá que tener en cuenta las relaciones de aspecto), pero si el desplazamiento se inicia hacia la derecha o hay un giro muy pronto, quizá convendrá dejar algo de margen izquierdo para evitar que el fondo se corte en ese extremo.

El desarrollo de este algoritmo ha dado varios problemas (ver reporte de la *Iteración #7)*, por lo que se desgranará aquí:

* Descripción: en abstracto, el componente simplemente guarda referencias de dónde está el fondo situado antes de la ejecución para después, durante esta, ir actualizando su posición en función del movimiento del dado. El objetivo es que cuando el dado llega al final del nivel el fondo estático también muestre justo su final (esto es, su extremo derecho por defecto), por lo que el fondo estático puede ser tan ancho como se quiera, que era la pretensión. No obstante, ante varias capas podría darse una incoherencia si una más lejana es notablemente más ancha que una más cercana, ya que se desplazará a mucha más velocidad y romperá la sensación de perspectiva.
* Como puede deducirse, no es posible configurar el fondo estático a una cierta velocidad, ya que él mismo la inferirá en función de su anchura y la anchura del nivel (distancia hasta el final, disparador *End of Level)*.
* Puede modificarse la escala local de la imagen, ya que el algoritmo la tiene en cuenta para medir cuál es el ancho del fondo.
* Funcionamiento: (los atributos están documentados en el código y puede adivinarse fácilmente el uso de las variables locales)
  + *Start()*: se guardan datos sobre la cámara y el fondo. Sobre la cámara, es necesario saber dónde está su extremo derecho —aunque podría ser el izquierdo, se ha tomado como referencia el avance hacia la derecha del dado—, su posición y la distancia de su borde al centro, que podría calcularse también con el tamaño ortográfico pero se ha hecho mediante resta de unidades. Sobre el fondo, se guarda la posición local original —es decir, dónde está ubicado relativamente, respecto a la cámara) y la diferencia entre el extremo de la cámara y el extremo del fondo, para tener constancia de cuántas unidades deberá haberse movido al final y por tanto cuál es el movimiento infinitesimal.
  + *Update():* se mide la posición actual de la cámara con pivote en el extremo derecho. También se normaliza este infinitesimal, que representa, de 0 a 1, en qué porcentaje del nivel se encuentra la cámara. Usará este porcentaje para colocar el fondo más alineado a la derecha —si tiende a cero— o a la izquierda —si tiende a uno—.

## Fondo dinámico *(Parallax Background)*

Componente reutilizado de un proyecto anterior. Simplemente mueve una imagen hacia la dirección —izquierda por defecto— y con la velocidad que le sean dadas. Cuando esta imagen sobrepasa la pantalla, se sitúa de nuevo detrás de ella, creando un efecto de carrusel. Para evitar tiempos muertos, se duplica la imagen tantas veces como se le diga, a fin de llenar la pantalla. A no ser que la imagen del fondo sea más pequeña que la pantalla, con el valor por defecto de un clon será suficiente, porque el primero se reposicionará antes de que el segundo avance más de la cuenta.

**Requiere** una escala natural, ya que si se cambia podría fallar el posicionamiento.

Es posible que, al ser hijo de la cámara y estar desplazándose con ella, las velocidades no se estén correspondiendo con lo que el jugador ve realmente, ya que la resultante será un vector consecuencia del movimiento de la cámara, que sigue al jugador, y la propia velocidad del fondo dinámico. Podrá revisarse si genera un efecto extraño.

Al igual que en el caso estático, hay que tener en cuenta siempre el efecto de perspectiva deseado. Una capa lejana irá más lenta que una cercana. En este momento es responsabilidad del desarrollador, aunque, a costa de disminuir la libertad de los parámetros, podría crearse un comportamiento que manejara todos los fondos dinámicos y les atribuyese una velocidad determinada (por ejemplo, en función de la lejanía, aunque al estar trabajando con cámara ortográfica y capas de ordenación, igualmente se requeriría un parámetro para esta lejanía. Al final sólo ahorraría un cálculo al desarrollador, que escribiría lejanía en lugar de velocidad directamente).

## Cámara que sigue al jugador *(Tracker)*

En caso de que sea necesario, se podrá generalizar el comportamiento de perseguidor como tal sin demasiado esfuerzo, ya que la estructura de este comportamiento, si bien incluye peculiaridades de la cámara que otros perseguidores no tendrían, está bastante bien compartimentada y aísla lo general. Por ejemplo, incluye una referencia pública al perseguido para ser añadida desde el inspector o por código, en lugar de directamente perseguir al jugador.

**Requiere** un colisionador, que debe estar situado justo en el borde derecho de la cámara. Servirá para colisionar con los disparadores que paran la cámara. Dicho más técnicamente, cuando se colisiona con un disparador de este tipo en realidad se deberá cambiar el estado de las variables *x* e *y,* que gobernarán en qué eje sigue el perseguidor al perseguido. En este caso, la cámara al jugador. Debido a que **actualmente sólo hay un disparador de persecución**, el objeto especial *fin de nivel* —que es el disparador de persecución por antonomasia—, simplemente se coteja la etiqueta con *Boundary* (antes *EditorOnly)* para cesar las persecuciones. Se deberá controlar a más granularidad esta colisión.

## Transformador de casilla *(Square Transform)*

Transformador alternativo. Permite desplazar directamente entre casillas en lugar de unidades, facilitando el posicionamiento durante el diseño de niveles.

Sin embargo, cualquier objeto que lo incluya estará restringido a moverse discretamente por casillas, ya que si el transformador de casilla detecta una posición no alineada con casillas, la ajusta —*snap—.*

# Prefabricados

## Dado *(Die)*

* Colisionador **cuadrado,** **obligatoriamente de** 4x4 unidades. Guía las rotaciones.
* Imagen **basada en** un lado igual a 2 unidades.
* *Player*. **Único** prefabricado y elemento en escena que tendrá este componente.
* *RollingCube*. Receptor del pulso que manda el sistema motor y gestor de la rotación.
* *Die*.Gestor del cambio de cara.

## Plataforma *(Platform)*

El prefabricado de la plataforma ya incluye tres unidades de plataforma, aunque puede tener más. Estas tres se corresponden con:

* Extremo izquierdo: incluye un disparador *End of Platform* como hijo, una casilla a su izquierda.
* Centro: es la unidad que se duplica en caso de aumentar el tamaño de la plataforma.
* Extremo derecho: incluye un disparador *End of Platform* como hijo, una casilla a su derecha.

Actualmente no se pueden generar plataformas con un tamaño menor a tres unidades, y esto puede requerir modificaciones en el futuro. Aunque tediosas, no serán problemáticas.

Para tener cierta coherencia estructural, el disparador de final de plataforma ha sido guardado como un prefabricado aparte. Contradictoriamente, esto rompe la coherencia de la propia estructura de prefabricados de Unity, debido a la limitación de los prefabricados hijos (un prefabricado se guarda como nuevo si se hace hijo de otro). Se ha optado por no cargar el proyecto con un complemento que aumente el manejo sobre los prefabricados hijos.

El tamaño y la posición de los prefabricados de plataforma se modifican con el editor de plataformas —comportamiento—. En caso de necesitarse versatilidad temática, el editor cogerá los recursos convenientes. Por ejemplo: si la plataforma es herbácea, usará correspondientemente unidades de plataforma de hierba; si es lignaria, unidades de madera.

## Fondos

Los estáticos se han desarrollado mediante la adicción de un objeto especial llamado *final de nivel —EOL—*.

Ha sido considerado que sólo se desea su desplazamiento en el eje horizontal. De ahí que su componente use números en punto flotante en lugar de vectores bidimensionales.

Donde de verdad radica el interés de los fondos estáticos es en el comportamiento, ya explicado antes, y no en el prefabricado, que es más una ordenación en la jerarquía. Mismo caso es el de los dinámicos.

# Editores

Existen a disposición del diseño varios editores diferentes, que se han desarrollado *ad hoc* según las características del juego se iban incorporando.

## Constructor *(Builder)*

Clase abstracta.

Los componentes de editor se han desarrollado como hijos de este, que hereda de *MonoBehaviour* y que se ejecuta también fuera del modo de juego (mediante el decorador *[ExecuteInEditMode]*). Aunque lo más natural sería usar la clase base *ScriptableObject*, algunos procedimientos propios de estos editores, como eliminar objetos de la escena, pueden solucionarse de forma mucho más sencilla con la clase por defecto de los comportamientos de Unity.

No obstante, mantener estos componentes durante la ejecución del juego podría llevar a error, sobre todo tras el despliegue. Para evitar esto, la plantilla Constructor se encarga de autoeliminarse si detecta que comienza la ejecución. Al contar ese instante ya como tiempo de ejecución, al detenerla el componente estará de nuevo en el objeto.

Además de todo lo anterior, incluye información estática sobre el tamaño de casilla, funciones para comprobar si una coordenada es casilla y traducción de datos entre casillas y unidades.

## Constructor de plataforma *(Platform Builder)*

Mediante *OnValidate()*, consigue adaptar la plataforma que lo incluye según el usuario desplaza el deslizador con el que se presenta su variable de tamaño. Lo que hace en realidad es dejar la variable por defecto —tamaño de tres unidades— y duplicar el centro tantas veces como sea necesario.

Puede ampliarse mediante gestión aleatoria de las teselas que se añaden, para dar variedad a la plataforma.

## Constructor de prefabricados *(Prefabs Builder)*

Por mejorar.

Posee una lista de prefabricados y otra que referencia a los objetos que ha creado. Es capaz de generar y destruir objetos. Su único objetivo es el de agilizar el tiempo que se tarda en añadir actores y plataformas a la escena, y evitar los molestos arrastrados desde la carpeta de recursos.

Podría duplicarse para diferenciar entre añadir plataformas y añadir actores, para que la jerarquía de escena quede tan limpia como sea necesario. Las plataformas serán añadidas al objeto vacío “Floor”.

Recursos gráficos

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Medidas y pivotes de actores | | | | | |
| Recurso | Casillas | | Pivote | | Observaciones |
| Die\_*x* | 1 | 1 | 0.5 | 0.5 | Caras desde *x=1* hasta *x=6* |
| Mark\_*x* | 1 | 1 | 0.5 | 1 | No se basa en casilla por donde pasará el dado |
| Post | 3 | 2 | 0.5 | 0.25 |  |
| Vane | 3 | 2 | 0.5 | 0.25 |  |
| Teleporter *x* | 3 | 3 | 0.5 | 0.5 | *Cabin, Output* y *Sink* son iguales cambiando color |
| Portal | 3 | 3 | 0.5 | 0.5 | Por decidir dónde se situará la pila de huellas |
| Grass platform (inner) | 2 | 3 | 0.5 | 0.75 | Con *Flip.X* activado, sirve de extremo derecho |
| Grass platform (outer) | 2 | 3 | 0.5 | 0.75 |  |

Utilidades

# Extensiones

## Extensiones de vector *(Utils.Extensions.VectorExtensions)*

* **Familia *Snap:*** redondea coordenadas del vector a un valor.
  + Snap(Vector3): redondea cada coordenada a cada valor respectivo del vector.
  + Snap(float, float, float): redondea cada coordenada a cada parámetro.
  + Snap(float): redondea cada coordenada al parámetro.
* **Familia *Coords:*** modifica cierta coordenada de un vector y la devuelve. NO CONFUNDIR con una modificación interna del objeto; simplemente lo devuelve modificado. Si se quiere modificar el propio objeto, hay que hacer la asignación. Esto concuerda con el uso que Unity hace en C# de ciertos atributos compuestos, como la posición.
  + X(float): devuelve el mismo vector con la coordenada *x* cambiada al parámetro.
  + Y(float): devuelve el mismo vector con la coordenada *y* cambiada al parámetro.
  + Z(float): devuelve el mismo vector con la coordenada z cambiada al parámetro.
  + XY(float, float): devuelve el mismo vector con las coordenadas *x, y* cambiadas a sus parámetros despectivos. *Recurrente en proyectos 2D como este.*

## Extensiones de objeto de juego *(Utils.Extensions.GameObjectExtensions)*

* **Familia *Clone:*** duplica el objeto usando *GameObject.Instantiate.*
  + Clone(Vector3, string = null): clona el objeto en la posición del primer parámetro y con el nombre del segundo. Si el segundo parámetro es nulo, se clona con el mismo nombre que el original.
  + Clone(string = null): clona el objeto en la misma posición que el original y con el nombre del parámetro. Si este es nulo, se clona con el mismo nombre que el original.

## Extensiones de transformador *((Utils.Extensions.TransformExtensions)*

* **Familia *Destroy:*** añade funcionalidades de eliminación de objetos para evitar pasos.
  + DestroyChildren(string): elimina todos los hijos con el nombre del parámetro. En su desarrollo se ha hecho un moldeo y se ha revertido la lista de hijos, para evitar problemas al eliminar objetos de una lista mientras se itera en ella (error habitual del bucle *foreach*). Usa el método *DestroyImmediate()* de *GameObject* para las eliminaciones. De hecho, incluye la especificación *“GameObject.”* pese a que el compilador la marca como innecesaria (¡No lo es! Parece un error del analizador semántico, dado que si se elimina, la eliminación no funciona).

Errores

# Problemas conocidos

* La pila de huellas genera un error cuando todas las huellas han sido superadas.
* Añadir a la huella que se desuscriba si está escuchando cuando es desactivada genera problemas, ya que cuando no debería estar escuchando ya, parece que sigue haciéndolo (al menos, su variable *listening* aún lo indica).

# Errores potenciales

# y zonas proclives

* Si se cambia el tamaño del colisionador del dado, o se desliga del resto de tamaños que lo involucran, puede haber un problema tanto con las colisiones como con el aspecto del dado durante su rodado.
* El sistema de los teletransportadores aún no está preparado para hacer frente a varias huellas en estado de acierto al momento de ser activadas por su cabina. Esto, directamente, nunca debería ocurrir, desde el diseño.
* Los colisionadores de las huellas son pequeñas líneas a ras de su superficie. En cualquier momento en que el suelo del dado no esté exactamente bien configurado, fallarán.
* Los siguientes componentes son considerados únicos, pero no implementan el patrón *Singleton:*
  + Jugador.
  + Dado (puede dejar de serlo en el futuro y habrá que cambiar algunas referencias para que ahora busquen el componente Jugador).
  + Cubo rodante (puede dejar de serlo).
  + Notificador.
* La pila de huellas (y puede que muchos otros prefabricados) deberá abordar su reinicio si es necesario. Tanto poner el suelo a cero como activar sus hijos será su tarea, ya que de otra forma no habrá manera de reiniciarla. (¿O sí? Probar cambiando su condición de activa y ver si se reinicia sola).
* Se ha considerado que la pila de huellas **sólo** se asocia un portal. Si se amplía su uso en el diseño tendrá que modificarse la implementación.
* La taxonomía de componentes **puede requerir hijos** específicos para los *sprites*, por el hecho de que la animación sea absoluta respecto al padre.
* El fondo estático puede fallar si el nivel tiende a hacer que el dado se mueva hacia la izquierda, o si lo hace por defecto. Pero puede que no sea necesaria ninguna modificación, si el algoritmo toma bien la referencia del “punto 0” y del “punto 1”.

# Consideraciones

* ~~Cuando el sentido del dado es inverso, hacia la izquierda, los colisionadores siguen estando en la derecha, por lo que no se comportan como deben. Podrían espejarse al ser notificados de que el dado ha cambiado su sentido, o simplemente reestructurarse siendo un pequeño botón en el centro, para que no intervenga el sentido de giro.~~ Solventado con la segunda opción.
* El notificador itera con bucle “para cada”, por lo que si se elimina algún elemento durante la iteración, fallará. Ocurrió en la notificación de pulsos y se solucionó elegantemente. Si es necesario, cualquier otra notificación puede copiar esta solución.
* El cálculo de los fondos estáticos lleva indirectamente asociado un colisionador en la cámara. Este colisionador está en su extremo derecho y hará que la cámara se desvincule del jugador cuando se alcanza el final de nivel. No obstante, no está aún contemplado que el colisionador se reposicione en función del tamaño de la cámara, ni que tenga que parar por su parte izquierda en caso de llevar el rumbo contrario. Los fondos sí que funcionarán igualmente, ya que no se basan en la posición del colisionador sino en las dimensiones de la cámara. Tampoco está contemplado que se reactive el seguimiento cuando el dado cambia su sentido.

# Optimizaciones potenciales

* El notificador podría hacerse *Singleton* o estático para evitar todas las búsquedas lentas cuando se le manda notificar algo (el método *FindObjectOfType()* no tiene muy buen rendimiento).
* Podría implementarse un patrón Delegación real, pero el notificador actual lo simula en un rendimiento aceptable, que habrá que monitorizar a menudo según se van añadiendo suscriptores. Sería interesante si se puede hacer que un cierto actor se suscriba sólo a una serie de notificaciones, ya que disminuiría mucho la carga de notificaciones. No será tomado en cuenta si el notificador no da problemas.
* El resto de posibles optimizaciones está especificado en código con comentarios no de documentación.

Ideas

# Mecánicas

* Veletas que se mantienen sólo durante un número de turnos (mecánicamente similar a paradores).
* Paradores (que actúan durante un número de turnos o bajo condición de éxito).
* Caídas a pisos inferiores (probablemente activando las físicas, aunque hay que pensarlo bien para evitarlo si es posible). Por propio peso, esto desemboca también en caídas al vacío y consecuente pérdida de una vida, lo cual estaba pensado ya.
* El **portal** puede conducir a una salida que sea también un portal. Piénsese en él como en el teletransportador de doble sentido. Esto agilizaría mucho el diseño cuando el portal busca llevar momentáneamente a zonas secretas, ya que un mero girador haría al dado volver al portal y por tanto salir por donde había entrado, continuando su marcha original. También puede ser este un tipo de portal, de manera que no se obligue a que se dé esta bidirección en todos los portales.

# Dinámicas

* Dos veletas simples contrapuestas mantienen al dado entre una zona. Puede servir para crear una parte parada donde el peso viene de lo vertical (un jefe que está arriba y lanza cosas abajo, etc.). También dos portales pueden cumplir esta funcionalidad, y otros objetos.
* Subida constante en cierto tramo, estilo plataformas de desplazamiento vertical (*Towerfall Ascension, Rainbow Island)*. Implementable mediante teletransportadores enlazados y giradores.
* Lugares donde se propone una “solución de puzle” más que un reto de plataformas o rítmico. Por ejemplo, que haya un teletransportador con una única salida y que esta sea sumidero que está situado de camino a la cabina; el jugador puede verse encerrado, pero mediante alguna solución podrá continuar su camino.

# Desarrollo

* El prefabricado para final de nivel puede generalizarse y que sirva como disparador para cualquier situación en la que la cámara deba pararse sin que lo haga el jugador, si lo hubiere.

