



CIIE
DR1N

Barbeito Vázquez, Ismael

`i.barbeito@udc.es`

Echevarrieta Catalán, Nicolás

`nicolas.echevarrieta.catalan@udc.es`

Martín Prieto, Rodrigo

`r.martin1@udc.es`

Ruiz Perez, Daniel

`d.ruiz.perez@udc.es`

12 de abril de 2015

Índice general

1. Desarrollo Artístico	1
1.1. Antecedentes	1
1.1.1. Ambientación	1
1.1.2. Historia	1
1.2. Personajes	3
1.2.1. Personajes jugables y aliados	3
1.2.2. Personajes enemigos	3
1.2.3. Jefes finales	4
1.3. Otras características de la ambientación	5
1.3.1. Objetos destacados	5
1.3.2. Lugares seguros	5
1.3.3. Zonas jugables	5
1.4. Guion	7
1.4.1. Modo historia	7
1.4.2. Modo supervivencia	7
1.5. Arte conceptual	8

2. Desarrollo Técnico 2D	9
2.1. Descripción	9
2.2. Diseño del motor	9
2.2.1. Stats	10
2.2.2. State	11
2.2.3. Physics	14
2.2.4. Graphics	15
2.3. Personajes	16
2.4. Enemigos	16
2.4.1. Enemigo a distancia (cactus y muñeco de nieve)	17
2.4.2. Enemigo cuerpo a cuerpo (escorpión y pingüino)	18
2.4.3. Jefes finales (Weedel y El Temible Jetty)	19
2.4.4. Nivel de dificultad	21
2.5. Objetos	22
2.6. Escenas	22
2.6.1. Fase	23
2.7. Apartado artístico	26
2.7.1. Extracción de sprites y animación	27
2.7.2. Detalles de diseño artístico	28
2.8. Aspectos destacables	32
2.9. Manual de usuario	33
2.10. Trabajo futuro	36

2.10.1. Motor	36
2.10.2. Enemigos	37
2.10.3. Escenas	37
2.10.4. Apartado artístico	38

Capítulo 1

Desarrollo Artístico

1.1. Antecedentes

1.1.1. Ambientación

El juego está ambientado en diferentes zonas de nuestro planeta en la era actual. Ya que la Tierra ha sido marcada como objetivo de una misión alienígena (ver 1.2 Historia), también se pueden encontrar ciertos artilugios y autómatas con aspecto tecnológico-futurístico, de uso desconocido.

1.1.2. Historia

Los Socif, considerados por muchos la raza más avanzada de la galaxia, son reconocidos en todos los confines del universo por su afán explorador. Sus mapas interestelares, así como sus enciclopedias cósmicas, se encuentran entre las más completas del espacio conocido. Sin embargo, en la región 1369-32T existe un planeta del cual ninguna de sus sondas D41N ha vuelto jamás.

Lareez, el afamado explorador e inventor Socif, presentó DRIN, el nuevo modelo D41N, como solución para explorar el misterioso cuerpo celeste. Su estrategia, basada en pequeñas incursiones de carácter conservador, le ha llevado a equipar al dron con armas y sistemas de evacuación de emergencia. La misión de DRIN, pues,

consiste en reunir toda la información posible del planeta. . . sin ser destruido en el intento.

1.2. Personajes

1.2.1. Personajes jugables y aliados

DRIN: El jugador encarna el papel de esta sonda Socif (Figura 1.5) . Su propósito es investigar el planeta desconocido, así como averiguar por qué ninguna sonda D41N ha logrado regresar con sus creadores. Es el modelo más avanzado hasta la fecha, y cuenta con autonomía propia, sistema armamentístico actualizable en base a módulos y sistemas automáticos de evacuación de emergencia.

DRIN además puede modificar su aspecto y habilidades en la nave nodriza de Lareez para adaptarse a diferentes estilos de combate y exploración.

Lareez: El inventor de DRIN, y uno de los pocos Socif que persevera en la investigación del sector 1369-32T. Espera en su nave la llegada de su creación, desarrollando mejoras que ayudarán al jugador a progresar en el extraño planeta.

Su nave nodriza es uno de los puntos seguros en el desarrollo de la historia. Allí el jugador puede adquirir mejoras permanentes para progresar en su misión, que Lareez irá desarrollando a medida que obtiene información de DRIN.

Posh: Después de sus primeras incursiones, DRIN logró localizar zonas deshabitadas del planeta, y es en ellas donde Posh ha decidido hacer “negocio”. Este contrabandista intergaláctico ha conseguido algunos de los fragmentos de las viejas D41N, y estará dispuesto a vendérselas a DRIN... por un precio “justo”.

Los artículos que oferta Posh son de dudosa calidad, pero durante una estancia temporal en el planeta pueden llegar a ser muy útiles. Sin embargo, difícilmente sobrevivirían a un viaje espacial, así que DRIN deberá dejarlas en la superficie del planeta cuando vuelva a la nave nodriza.

1.2.2. Personajes enemigos

Criaturas autóctonas: Los causantes de todos los fracasos de las sondas D41N. Hasta la fecha solo se conocen algunas razas sueltas, compuestos principalmente por seres bípedos y cuadrúpedos. Son agresivas contra todos los contactos

exteriores, y deben ser tratadas con la máxima cautela. Disparar primero, preguntar después.

Constituyen los enemigos básicos del jugador. Su comportamiento siempre es el mismo, pero su aspecto varía en función de la temática de la zona.

1.2.3. Jefes finales

Jetty: Bípedo monstruoso de la zona helada que no consiente que nadie se adentre en su territorio. Es la primera gran amenaza de DRIN en el planeta desconocido. Su tremenda fuerza es un serio peligro, aunque es realmente lento. . .

Weedel: Este cuadrúpedo ha consumido algún tipo de sustancia alucinógena y confunde a la sonda con un individuo de su raza. Sus ataques se basan en embestidas que. . . ¿parecen buscar mucha proximidad?

Shampoo: Parece ser un guerrero de la raza bípeda de esta región. Sus ataques son veloces como el rayo, y aunque pueden no parecer muy poderosos, una rápida sucesión de los mismos podría ser fatal para DRIN.

M4L4N-D41N: Los habitantes de la población tecnológica parecen haber comprendido la tecnología Socif, y han logrado desarrollar un clon de DRIN con las piezas de las sondas perdidas. . .

1.3. Otras características de la ambientación

1.3.1. Objetos destacados

Partes perdidas de D41N: Los enemigos pueden dejar caer al morir estas piezas de las sondas destruidas antes de la llegada del jugador. Funcionan como potenciadores temporales o moneda de cambio para la tienda.

Modificaciones de DRIN: Lareez puede modificar algunas de las partes del exoesqueleto de DRIN para otorgarle diferentes habilidades, potenciar algunos de sus rasgos (a cambio de debilitar otros) o incluso modificar patrones de comportamiento, lo cual influirá en el estilo de juego que se pueda adoptar.

1.3.2. Lugares seguros

Nave: La nave de Lareez, el creador de DRIN, es el sitio más seguro del videojuego, y el único lugar en el que se pueden adquirir mejoras permanentes. Consta de un sistema de ocultación cuyo propósito es evitar ser detectado por los habitantes del planeta.

Tienda: Posh aprovecha los avances del jugador para ir colocando pequeños puestos de su negocio por todo el planeta. DRIN podrá adquirir aquí mejoras no permanentes.

1.3.3. Zonas jugables

Paraje helado: Poblada por pingüinos, esquimales, y el terrible yeti Jetty, es la primera región a la que llega DRIN.

Región abrasadora: Un desierto en el cual escorpiones y nómadas intentarán dar caza a la sonda. Un enorme camello, adicto a una variante alucinógena de cactus, junto a arenas movedizas y otros peligros constituyen el resto de amenazas de esta zona.

1.3. OTRAS CARACTERÍSTICAS DE LA FUNCIÓN DE DESARROLLO ARTÍSTICO

Guarida espiritual: Un pueblo recóndito que guarda las costumbres del Japón feudal. Mapaches, ninjas y un avezado samurái son los enemigos que habrá que enfrentar en esta parte del planeta.

Población tecnológica: Una ciudad humana actual, altamente poblada. Vehículos y soldados atacarán a DRIN como antesala de M4L4N-D41N, el último enemigo del juego.

1.4. Guion

Se presentan dos modos de juego:

1.4.1. Modo historia

El jugador deberá progresar por las 4 regiones, vistando para ello escenarios cerrados en los cuales será asediado por “oleadas” de enemigos. Cada escenario se genera de forma procedural, con la temática del lugar (enemigos, geografía, características del terreno, trampas...). Cuando haya superado un número concreto oleadas, podrá acudir a la tienda a comprar mejoras, y tras otra cierta cantidad aparecerá el jefe final de la zona, que una vez derrotado garantizará el acceso a la siguiente. El juego se considera superado solo si el jugador logra derrotar al jefe de la cuarta zona.

Si los puntos de salud de DRIN llegan a cero a lo largo de la partida será devuelto a la nave nodriza, y recibirá una serie de créditos de exploración (otorgados en función de la vida restante, los enemigos abatidos, el tiempo que se haya sobrevivido u otros logros). Esta moneda es la que le permitirá adquirir las mejoras de carácter permanente, desbloquear nuevas modificaciones (equivalente a estilos de juego, o “personajes”) e incluso adquirir contenido extra.

1.4.2. Modo supervivencia

Es similar al modo historia, pero cuando se vence al último jefe, se vuelve a empezar desde Paraje Helado, solo que los enemigos serán mucho más fuertes, y habrá nuevas mejoras para comprar. El objetivo es sobrevivir el mayor número de oleadas posible.

1.5. Arte conceptual

Concepto inicial, sujeto a posibles cambios, del protagonista. Se contemplan estructuras para la realización de ataques a distintos rangos, expresión de emociones a modo de feedback...

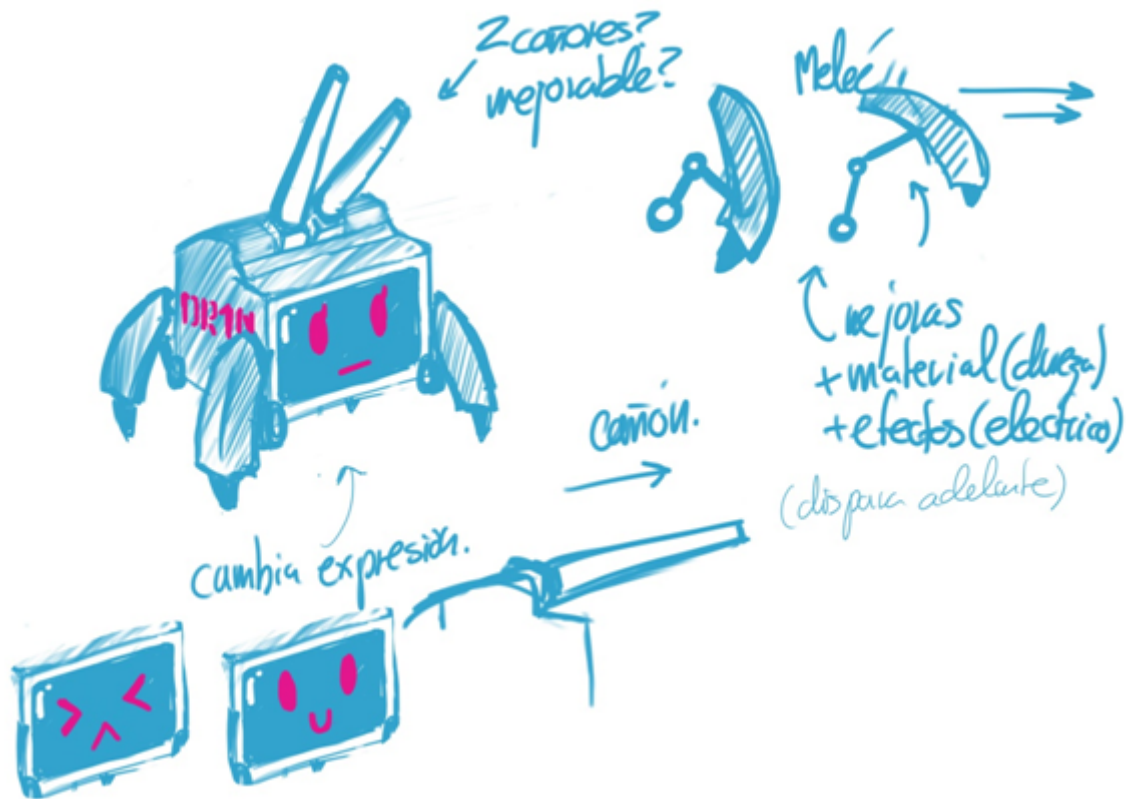


Figura 1.1: Concepto artístico de DRIN

Capítulo 2

Desarrollo Técnico 2D

2.1. Descripción

La presente memoria recoge los aspectos más destacables del desarrollo técnico 2D de la demo del videojuego DRIN. Este documento se centra solo en aquellos aspectos que se han podido implementar en relación al apartado “Memoria artística”, por lo cual es posible que haya ciertos aspectos que se hayan perdido en relación a lo citado anteriormente. Para posible trabajo futuro, consultar la sección correspondiente.

Se ha decidido estructurar la memoria entorno a los cuatro aspectos en los que se ha dividido durante el desarrollo: el motor de juego, la generación de fases y escenarios, la inteligencia artificial y el apartado gráfico.

Todos los diagramas UML se sitúan en la carpeta correspondiente, adjunta a la memoria.

2.2. Diseño del motor

Lo primero que se diseñó, de cara a facilitar la creación de contenido, fue un pequeño conjunto de clases que se ha denominado motor básico del videojuego. La idea subyacente es crear una serie de abstracciones que permitan separar los

diferentes comportamientos de los objetos que intervienen en cada una de las escenas. Con esto se logra dividir el trabajo en competencias (artístico, inteligencia artificial, control de jugador. . .) que son paralelizables.

A nivel conceptual, el motor considera que una pantalla concreta de juego se puede resumir en la interacción de varios grupos de objetos que implementan la clase Actor. Básicamente, todos los enemigos, plataformas, trampas, disparos, personajes controlables o cualquier otro tipo de agente que no sea un fondo estático es una extensión de esta clase. Esto convierte a Actor en el núcleo de todo el desarrollo, por lo cual es importante entender bien su funcionamiento.

Lo primero a tener en cuenta es que Actor es una clase que extiende a `pygame.sprite.Sprite`. Esto provoca que, entre otras cosas, todos los agentes del juego tengan dos atributos básicos por herencia: `rect` (el rectángulo de colisión) e `image` (la imagen asociada al sprite).

Además de los atributos heredados, todo actor contiene otros cinco objetos con los que trabajar. El primero de ellos es una simple referencia a la fase en la que se encuentra (para, por ejemplo, permitir a la IA conocer el mundo en el que se mueve). Los otros cuatro separan los diferentes aspectos de cada Actor: aspecto gráfico (`graphics`), físicas (`physics`), comportamiento (`state`) y características de juego (`stats`).

2.2.1. Stats

La clase Stats se comporta como un almacén que lleva la cuenta de las estadísticas de juego. Estas estadísticas son los parámetros que modifican la lógica de juego, como el daño y la vida. Es utilizada por muchas clases y métodos para llevar

actualHP: vida que le queda actualmente al actor

damage: cantidad de vida que restan los ataques a otros actores

ghostMode: indica si el “modo fantasma” esta activo, en el cual el actor no recibe daño, y su sprite parpadea.

ghostModeTimer: indica el tiempo restante hasta salir del “modo fantasma”

maxGhostModeTimer: indica el tiempo máximo que el actor puede permanecer en modo fantasma (esto es, cuando se activa, ghostModeTimer se inicializa a este valor).

maxHP: indica la vida máxima que puede tener el actor.

maxJumpSpeed: indica la máxima velocidad de salto. Se utiliza para asignar a la componente y de la velocidad cuando el actor salta.

maxSpeed: la velocidad máxima de movimiento horizontal. Se utilizar para asignar la componente x de la velocidad cuando el actor se mueve.

meleeAttackDuration: el tiempo de vida que tiene el ataque melee (esto es, la cantidad de frames que dicho actor permanencia vivo).

meleeRecoil: recarga del golpe cuerpo a cuerpo. El tiempo que debe pasar hasta poder volver a golpear.

meleeSpeed: velocidad a la que se realiza el ataque físico.

shootRecoil: similar a meleeRecoil, pero para el ataque a distancia.

shootSpeed: similar a meleeSpeed, pero para el ataque a distancia.

shotSpeed: velocidad a la que se mueve el disparo que el actor realiza.

Siendo un simple contenedor, la clase contiene solo getters, setters y adders (setters incrementales).

2.2.2. State

State se encuentra entre las clases más complejas del proyecto. Su función es otorgar a los actores de un comportamiento. También debe proporcionar un manejador (el cual se puede ver como un “decisor”, esto es, un agente capaz de decidir cómo transitar entre estados).

En un principio se optó por simular la implementación de un autómata mediante el uso del patrón estado. Idealmente, cada actor tendría un estado concreto que

sabría, en función de las entradas que recibiera, a cuales debería transitar. Sin embargo, en muchos casos esto implicaba replicar código (por ejemplo, para el caso de acciones que se desea que se realicen siempre ante ciertas entradas), por lo cual se optó por intentar crear un autómata genérico con las acciones básicas que cualquier Actor podría asumir. Se decidió que dicho conjunto de acciones fuera saltar, atacar cuerpo a cuerpo, atacar a distancia, moverse a la izquierda, moverse a la derecha y no hacer nada (acción por defecto).

Como se puede ver en (UML1), la idea es definir comportamientos abstractos comunes para las transiciones (desde cualquier estado al estado saltar se debe modificar, al menos, la velocidad en el eje Y y la animación), y luego, utilizando overriding de métodos, extender este comportamiento con nuevas clases (por ejemplo, devolver el mismo objeto en lugar de crear uno nuevo en caso de estar repitiendo la misma acción).

Todas estas ideas se materializan en actorState. Esta jerarquía de clases está pensada para definir los comportamientos de cada acción. Posee un método abstracto, `decidirSiguienteEstado`, para, mediante extensión de clases, permitir que la Inteligencia Artificial y el propio jugador utilicen las mismas llamadas a métodos. Esto es, crea una interfaz común a ambos controladores.

Esta aproximación fue finalmente descartada por un caso concreto que se produce en situaciones como la que se ilustra en (UML2). Si se quiere generalizar ciertos aspectos del control del jugador en una clase “estado genérico de jugador”, sin dejar de reutilizar aspectos de las clases de cada estado concreto de actor, nos encontramos ante un aspecto de herencia múltiple. Ejemplificando, el estado Jugador ataca debería tener el `decidirSiguienteEstado` de Estado genérico para jugador, pero también aquellas acciones necesarias para realizar un ataque, definidas en Actor ataca. Es decir, necesitamos extender esas dos clases.

El lenguaje utilizado (Python) ha presentado problemas en cuanto a la prioridad de los métodos en caso de colisión de firmas. Además, el incurrir a herencia múltiple suele ser indicativo de errores de diseño, por lo cual se plantea una nueva forma de implementar State.

Se parte del mismo supuesto inicial: el estado de un personaje se puede ver como la configuración concreta de una máquina de estados. Desde ese estado, se

tiene que poder transitar a los demás en función de las acciones que se puedan ejecutar. Además, se quiere que, siendo un gran número de acciones comunes a todos los actores, sea posible implementar jerarquías genéricas (como, por ejemplo, el autómatas que implemente las acciones por defecto).

Para lograrlo, se ha seguido una aproximación que mezcla los patrones estado y estrategia en lo que se ha llamado patrón state-decisor. Su aproximación inicial es esta: todo estado contiene un decisor, que es el objeto al cual le pregunta cómo avanzar. El estado es un nodo concreto, y el decisor conoce todas las aristas del grafo. Así pues, cada vez que el estado reciba una acción (“Estado, ejecutar acción mover”), este preguntara a su decisor qué debe hacer en respuesta, que puede ser generar un nuevo estado (que contendrá el mismo decisor) o quedarse como está. Al pasar el decisor como atributo en esta cadena, este objeto se vuelve inalterable al Actor concreto.

Una forma de comprender las consecuencias de esto es imaginar el diagrama de estados de un enemigo capaz de moverse y lanzar un rayo. En dicho grafo existirían dos nodos (moverse, disparar). Si se quisiera generar un enemigo nuevo que al disparar lanzara una bola de fuego, se podría utilizar herencia. El decisor sería el mismo (es decir, la arista de moverse a disparar y la arista de disparar a moverse), pero el nodo “disparar” cambiaría. Esto es, autómatas de estados intercambiables, en los cuales las aristas son fijas.

Con la idea del state – decisor, el siguiente paso lógico es verificar esta idea contra el requisito “generar jerarquías/autómatas genéricos”. Es decir, adaptar actorState a esta nueva filosofía.

Partiendo de las interfaces genéricas (State, Decisor), se implementan las clases ActorState y ActorDecisor. ActorDecisor contiene el conjunto de posibles acciones en forma de métodos manejador (onJump, onIdle, onShoot. . .). El método newState, el cual constituye el manejador, queda sin implementar, de cara a que este esquema sea reutilizable.

Para garantizar la permanencia de las aristas, los manejadores no crean directamente el nuevo estado. Esto es, onJump no devuelve directamente una clase JumpState, sino que se la solicita a un método getJumpingClass. De este modo, si se quiere modificar algún comportamiento, no sería necesario sobrescribir un método

onX, sino que llegará con modificar getXClass (extender la clase y hacer overriding en dicho método).

En cuanto a la jerarquía de estados, se proporciona un ActorStateInit como punto de entrada. En este estado simplemente se asigna el decisor, sin modificar al actor, y supone el inicio del autómata. Los demás estados, simplemente, realizan las modificaciones necesarias en el constructor.

En (UML3) se puede observar toda esta jerarquía. Heredar de ella permite implementar los comportamientos por defecto, extenderla con nuevas subclases ampliar su contenido, y realizar scripting en newState definir formas más complejas de transición. Como ejemplo de su utilización directa está la clase playerState, que es el estado de DRIN. En su decisor (UML4), además del método newState, sobrescribe el comportamiento de algunos métodos para que los disparos se registren de forma adecuada en sus grupos correspondientes.

Para la tercera variante en detalle, scripting, consultar la sección Inteligencia Artificial.

2.2.3. Physics

Physics se encarga de gestionar todos aquellos cálculos asociados al movimiento del Actor, implementando para ello una cinemática básica. Esta clase gira en torno a tres atributos: position (dónde está de forma absoluta el actor), scroll (posición relativa a la sección de pantalla que se muestra) y speed (velocidad actual).

El comportamiento de este objeto es simple: en cada frame, lo único que hace es recalcular la nueva posición (relativa y absoluta) en función de los cambios que State haya decidido para la velocidad. Una vez decididos, asigna la nueva posición al rectángulo de su Actor correspondiente. Esto, realizado con el suficiente refresco, es lo que provoca la sensación de movimiento.

Si bien en esencia el comportamiento es bastante similar al de los ejemplos, el hecho de que este modularizado en una clase permite realizar aproximaciones interesantes. Un ejemplo sería el asignar movimientos más complejos sin modificar el código del Actor concreto.

Cabe destacar que las colisiones no están implementadas en esta clase, ya que se ha optado por definir las de forma genérica a nivel de grupos, como se explica en el apartado “Escenas y Fases”.

2.2.4. Graphics

Se encarga de manipular el atributo `image` del Actor para dotarle de un aspecto gráfico. Graphics puede construir superficies transparentes, imágenes estáticas o sprites (a partir de una hoja de sprites).

A continuación se detallan los atributos de la clase (en la sección 2 se detallan los algoritmos utilizados).

animCD: tiempo que debe transcurrir entre dos frames de una animación.

animCDCounter: tiempo que ha transcurrido desde la anterior actualización.

animationOnGoing: indica si una animación se está reproduciendo o no. Esto sirve para crear animaciones bloqueantes, esto es, animaciones que no pueden ser detenidas a la mitad (por ejemplo, en caso de atacar en movimiento, la animación de ataque tiene prioridad).

blink: esta variable indica si el modo “parpadeo” está activo. En parpadeo, el sprite aparece y desaparece de forma consecutiva. Usado en conjunto con el “modo fantasma” (ver 2.2.1.4 Stats) y como indicador visual de este último.

conf: guarda el nombre del fichero donde se almacenan las coordenadas (para sprites).

currentAnimation: indica la animación concreta que se está reproduciendo.

currentFrame: indica el fotograma concreto de la animación actual a reproducir.

direction: la dirección a la que el sprite está mirando. Entre las constantes del proyecto se define LEFT como -1 y RIGHT como 1, coincidiendo con la orientación de los sentidos positivo y negativos de las coordenadas en pygame.

imgName: nombre del fichero que contiene los recursos gráficos que se usan en el Actor.

playingAnimation: animación concreta que se está reproduciendo ahora mismo.

sheetCoord: almacena un array de rectángulos que indica las posiciones concretas (en píxeles) de los frames que se deben leer para cada animación del fichero de imágenes.

sourceImg: almacena el fichero “imgName” en memoria.

Todos estos atributos del motor pueden ser observados en el UML5 a modo de resumen general

2.3. Personajes

2.4. Enemigos

Con el motor de juego desarrollado y todas las funcionalidades que ofrece, el desarrollo de la Inteligencia Artificial de los enemigos se limitó al algoritmo, abstrayéndose de la implementación de las funcionalidades. A pesar de haberla codificado como si fuera un script, tiene una lógica de estados subyacente que se explicará más adelante.

Se distinguen tres arquetipos de enemigos, claramente diferenciables entre sí pero con ciertas características comunes. Estas características podrían verse como una implementación de una arquitectura subsumida, propia de los paradigmas reactivos en la robótica. En este tipo de arquitectura, componentes de alto nivel inhiben las salidas de los actuadores para tomar ellos el control. De este modo, los enemigos (menos los jefes finales) implementan de forma “innata” las siguientes funcionalidades:

Esquivan disparos: Como los personajes tienen un completo control de todo lo que pasa en la fase (exactamente igual que el jugador), saben dónde están y con qué velocidad se desplazan los disparos enemigos. Con esto en mente y teniendo en cuenta que también saben su posición y velocidad de salto, esquivarlos les resulta trivial.

Saltan por encima de huecos y trampas: El comportamiento es el mismo para ambos casos: como saben la velocidad a la que se están moviendo, y sabiendo su posición en el instante t , calcular la posición en $t+1$ resulta de un simple cálculo cinemático. Si en esa futurible posición no hay plataforma porque hay un hueco o trampa, saltan y los esquivan.

Búsqueda del scroll: Siempre intentan estar en la pantalla para que DRIN no tenga que ir a buscarlos.

Intentan estar a la misma altura que DRIN, de forma que si está más arriba salta, y si está más abajo baja atravesando la plataforma (siempre y cuando estén seguros de que debajo hay o una plataforma o suelo en el que apoyarse.)

Además de esto, hemos diferenciado tres tipos de enemigos: a distancia, cuerpo a cuerpo y jefes. En las siguientes líneas se explicará brevemente las características y comportamientos de cada uno.

2.4.1. Enemigo a distancia (cactus y muñeco de nieve)

Es un enemigo cobarde; prefiere las largas distancias y odia que superen su perímetro de seguridad, por lo que hará todo lo posible por alejarse de DRIN si éste se aproxima.

Tiene un amplio rango de visión desde el que dispara, y no se moverá del sitio a no ser que sea estrictamente necesario, dando la sensación de estar enraizados al suelo (lo cual coincide con su comportamiento real como cactus y muñecos de nieve).

La principal diferencia entre ambos tiradores es que el cactus dispara en horizontal siempre, mientras que el muñeco de nieve dispara hacia la posición del jugador. Su diagrama de estados puede sintetizarse en el esquema de la figura 2.4.1

Spawn: Los enemigos aparecen tanto dentro como fuera de la pantalla

Disparar: Disparan a DRIN (según su naturaleza)

Ir a por DRIN: Si están fuera de la pantalla, van a ella

Escapar: Cuando DRIN supera su perímetro de seguridad, escapan

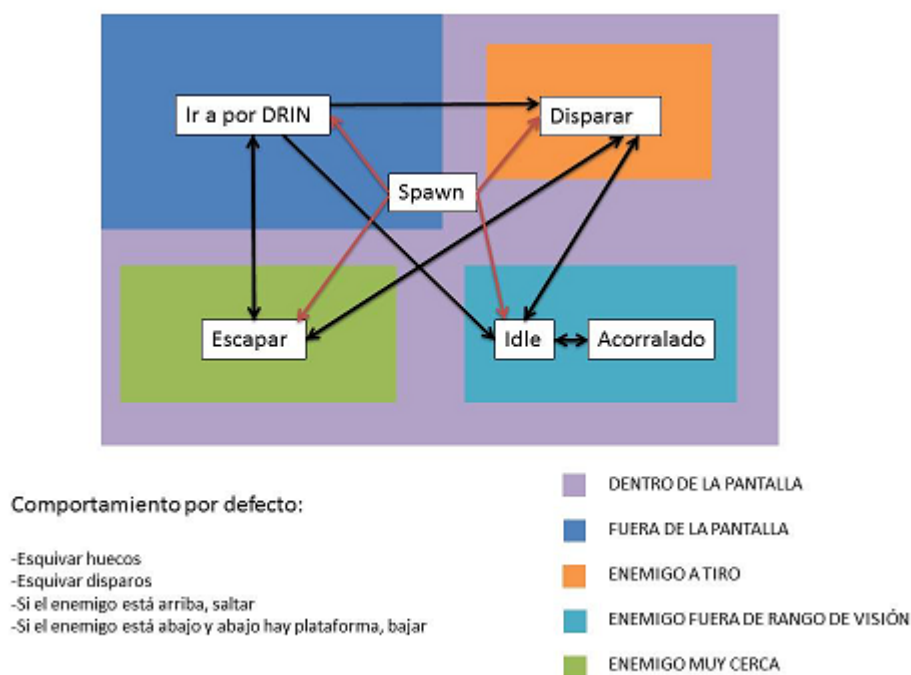


Figura 2.1: Diagrama de comportamiento de enemigos a distancia

Idle: Cuando no ven a DRIN, se quedan quietos

Acorralado: Si DRIN está demasiado cerca pero la fase se ha acabado (límite horizontal), disparan a bocajarro.

2.4.2. Enemigo cuerpo a cuerpo (escorpión y pingüino)

Su comportamiento es opuesto al de los enemigos a distancia. A pesar de tener menos rango de visión, intentará a toda costa acercarse lo máximo posible al jugador para poder atacarle.

Cuando DRIN no está en su rango de visión, se moverá por el mapa en su búsqueda.

Sus ataques son diferentes. Mientras que el escorpión ataca sin más, el pingüino recorre siempre una distancia determinada atacando (resbalando sobre hielo).

Su diagrama de estados puede sintetizarse en el esquema de la figura 2.4.2

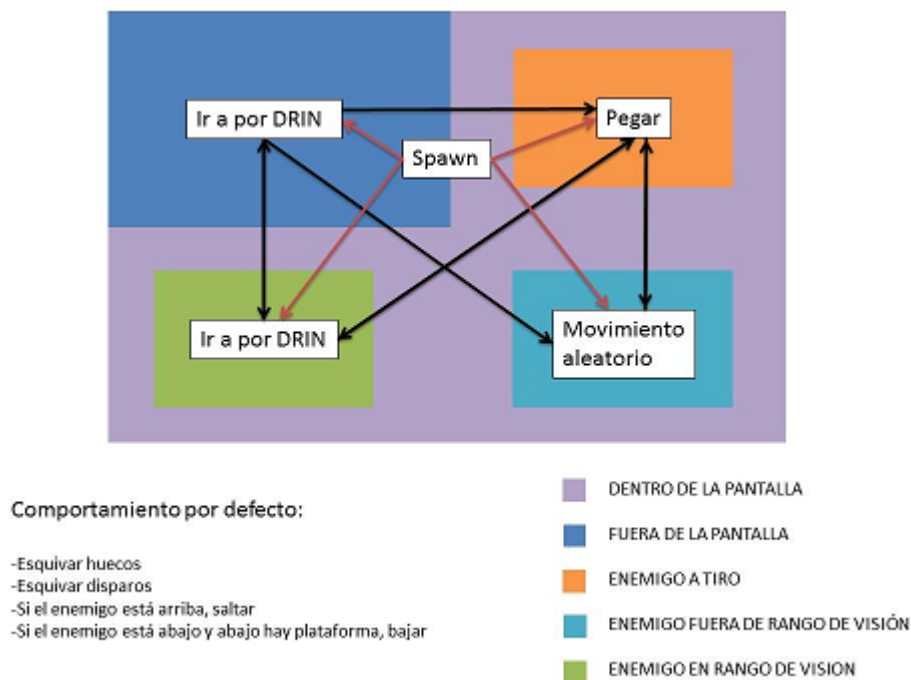


Figura 2.2: Diagrama de comportamiento de enemigos cuerpo a cuerpo

Spawn: Los enemigos aparecen tanto dentro como fuera de la pantalla

Pegar: Atacan a DRIN (según su naturaleza)

Ir a por DRIN: Si están fuera de la pantalla, van a ella

Movimiento aleatorio: Cuando no ven a DRIN, lo buscan por el mapa

2.4.3. Jefes finales (Weedel y El Temible Jetty)

Sus esquemas de estados son considerablemente más sencillos que los de los enemigos comunes. En sus fases no hay ni trampas ni huecos. Su comportamiento no se ve influenciado por la presencia de amenazas, por lo que ni perseguirán ni se alejarán del jugador.

- Weedel:

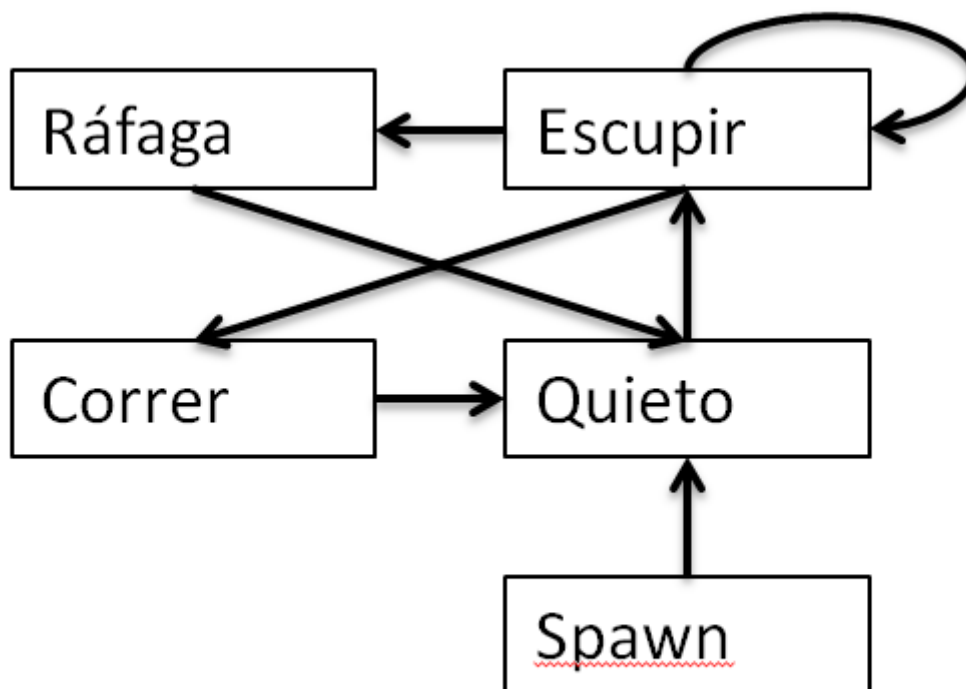


Figura 2.3: Diagrama de comportamiento de Weedel

Un camello que se dice que trafica con ciertas setas alucinógenas que al parecer consume habitualmente. Odia que rondan por su territorio por lo que escupe y atropella a todo el que ose acercarse.

Su diagrama de estados puede sintetizarse en el esquema de la figura 2.4.3

Escupir: Dispara babas hacia donde esté el objetivo.

Ráfaga: Dispara una ráfaga de babas en el sentido del objetivo.

Correr: Corre hasta chocar contra el límite de la fase (inicialmente hacia DRIN).

Quieto: Estado entre transiciones.

■ El Temible Jetty:

Con razón no encontraban al temible hombre de las nieves... ¡Lo teníamos nosotros!

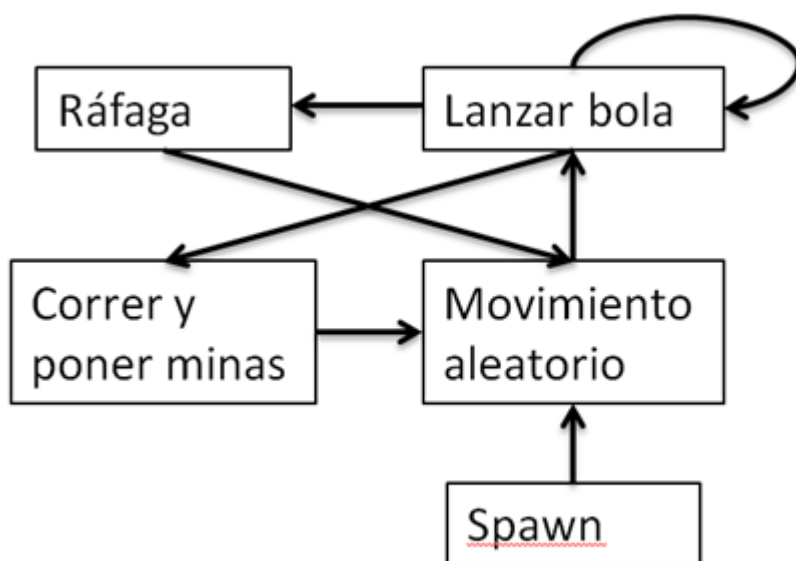


Figura 2.4: Diagrama de comportamiento del Jetty

Todavía más territorial que Weedel, lanza bolas de nieve a todo el que se le acerque y asegura su territorio poniendo minas de nieve por todos los niveles de su guarida.

Su diagrama de estados puede sintetizarse en el esquema de la figura 2.4.3

Lanzar bola: Dispara una bola de nieve a la posición de DRIN.

Ráfaga: Dispara un montón de bolas de nieve a la posición de DRIN

Correr y poner minas: Se desplaza horizontalmente por toda una plataforma mientras llena el suelo de minas de nieve compactada (como solo la tradición milenaria de los yetis cuenta).

Movimiento aleatorio: Mientras transita por los demás estados, se mueve aleatoriamente en las 4 direcciones.

2.4.4. Nivel de dificultad

Algo reseñable dentro de este apartado es la posibilidad de cambiar el nivel de dificultad del juego. Esto afecta a todos los comportamientos de los enemigos, ya que

todos sus atributos aumentan de forma beneficiosa para el enemigo en función del nivel. Este aumento se produce según la relación $\text{antiguo} * \text{nivel} / 5$ o $\text{antiguo} * \text{nivel} / 10$, dependiendo del atributo.

En función del nivel, los enemigos pueden cometer errores. Esto significa que la probabilidad de esquivar satisfactoriamente disparos, trampas y agujeros aumenta de forma directa al nivel. En todos los niveles hay una cierta probabilidad de error, pero en los niveles bajos es muy frecuente. Además las diferentes habilidades escalan de forma diferente por nivel.

Los jefes finales se ven afectados de forma similar, aumentando todos sus atributos. Mención especial a Weedel, ya que las sustancias alucinógenas que consume le otorgan el poder de creer mucho en algo, lo que hace que tenga comportamientos adicionales en función del nivel. Esto es, escupe más babas en cada ráfaga por cada dos niveles adicionales por encima del 6.

2.5. Objetos

2.6. Escenas

Las escenas del juego se han organizado en un menú como escena inicial que permite iniciar el juego, ir a la tienda o salir.

Al iniciar el juego, se entra en una secuencia de escenas compuesta por fases e interfases. Las fases vienen definidas en un documento xml (descrito más adelante), con lo que se consigue un sistema de fases escalable y configurable. Las interfases son la escena de transición entre fases, realizado de esta forma para tener una clara diferenciación de propósito, siendo el de la interfase hacer la transición cuando el usuario quiera y si se da el caso, mostrar información extra sobre la fase recién terminada o la siguiente.

En caso de que una fase termine por la muerte de DRIN, se volverá a la escena menú.

2.6.1. Fase

Como aclaración previa, para aumentar la escalabilidad y facilidad de mantenimiento, en todo el código referente a este tema no hay cableada ninguna ruta de ficheros, sino que todas se consiguen del gestor de recursos mediante un identificador. El gestor de recursos extrae esa información de un fichero XML. Así mismo, casi todos estos identificadores se sacan de la definición de la fase, por lo que tampoco están cableados.

La clase fase hereda de la clase EscenaPygame y en su constructor se le ha de pasar el director (por herencia de EscenaPygame) y un identificador de fase. Fase ha sido diseñada pensando en la reutilización e independencia. Se puede subdividir en varias secciones: definición de la fase, creación y registro de elementos, sistema de actualización (colisiones y dibujado) y dinámica de la fase.

La definición de fase se realiza en un fichero externo de formato xml predefinido. Justo después de llamar al constructor de EscenaPygame, se pide al gestor de recursos el árbol correspondiente a la fase cuyo identificador se ha pasado en el constructor, del cual se sacará toda información de la fase: nivel de dificultad, fase siguiente, largo de la fase, identificador del fondo, colocación de suelo, plataformas y trampas y la oleada de enemigos.

Dicho formato se especifica en (Figura 2.6.1)

En toda fase, si se cae por debajo del suelo, se muere. En el fichero se define en que zonas hay suelo (que esta a una altura fija), dejando como huecos el resto.

De manera analoga, se definen las plataformas, indicando la posición de la esquina superior izquierda, su longitud y el identificador del sprite.

El código interno esta preparado para que tanto plataformas como suelo se formen como 6 plataformas con diferentes sprites. Dichas plataformas se dividen en dos grupos: la plataforma per se y los sprites del cuerpo sobre el que esta la plataforma (tejado y cuerpo de la casa respectivamente). Solo las primeras participan en las colisiones, quedando las segundas como fondo. Dentro de cada subgrupo, se dividen en esquinas y centro para poder hacer texturas que ayuden a diferenciar una plataforma de otra en caso de superponerse. No obstante, por problemas con la

```
1  <?xml version="1.0"?>
2  <stage>
3      <nextPhase>nombre</nextPhase>
4      <background>[dessertBack | snowBack]</background>
5      <sky>[dessertSky | snowSky]</sky>
6      <large>[MINLENGTH..maxint]</large>
7      <difficulty>[1..10]</difficulty>
8      <floor>
9          {<surface position="[MINLENGTH..large]" length="[MINLENGTH..large]">
10             idSprite
11          </surface>}
12      </floor>
13      <platfotms>
14          {<platform>
15              <position x="[MINLENGTH..large]" y="[0..500]" />
16              <size>s</size>
17              <image>[dessertPlatform | snowPlatform]</image>
18          </platform>}
19      </platfotms>
20      <traps>
21          {<trap x="[MINLENGTH..large]" y="[0..500]">spikesTrap</trap>}
22      </traps>
23      <enemies>
24          {<enemy>
25              <type>
26                  [cactus | scorpion | camel | penguin | snowman | jetty]
27              </type>
28              <quantity>[1..n]</quantity>
29              <maxActive>[1..n]</maxActive>
30              <minWait>[0..n]</minWait>
31          </enemy>}
32      </enemies>
33  </stage>
```

Figura 2.5: Definición de una fase

repetición de tiles (localizado en la linea 16 de Platform), se ha optado por el uso de dos texturas planas, una para la plataforma y otra para el cuerpo. Cabe destacar que esta modificación se ha realizado a nivel de imagenes, con lo que de solucionarse el error, los cambios en el código serían mínimos.

Las trampas se definen indicando el tipo de trampa por su identificador y su posición.

En cuanto a la definción de enemigos, para cada tipo que se quiera que participe en la escena, se indica su identificador, el numero total que han de salir, cuantos pueden estar vivos simultaneamente y el tiempo que ha de pasar (el valor se multiplicará internamente por 60 y serán frames) desde que nació el último hasta que pueda nacer otro.

A la hora de crear y registrar los elementos del escenario, se tiene un método para cada elemento, el cual crea el elemento y llama a otro método que le registre en los grupos correspondientes.

La subdivisión en grupos se ha realizado tanto teniendo en cuenta el comportamiento (plataformas de disparos), como el orden de dibujado (los cuerpos de las plataformas se dibujan antes que las plataformas para que no las tapen). Por ello han quedado los siguientes grupos (a parte de uno general con todos los sprites) de forma que no se solapen: enemigos, jugadores, cuerpo de plataformas, plataformas, trampas, cuerpo del suelo, suelo, vacio (lo que detecta cuando te has caido de la fase), disparos del jugador, ataques cuerpo a cuerpo del jugador, disparos de los enemigos y ataques cuerpo a cuerpo de los enemigos.

El orden de dibujado es el siguiente: cuerpo de plataformas, plataformas, trampas, cuerpo del suelo, suelo, enemigos, jugadores, disparos del jugador y disparos de los enemigos. Las animaciones de los ataques cuerpo a cuerpo ya estan incluidas en el propio personaje.

En lo referente a las colisiones, se ha creado un método collision el cual recibe como parametros lo mismo que el groupcollide de pygame, y el nombre de un método el cual será llamado si la colisión es positiva. Este método implementa el comportamiento que habrá en caso de colisión. En principio existen dos comportamientos

programados: `behHurts` y `behGameOver`. En caso de que este parámetro sea `None`, no habrá comportamiento añadido.

La dinámica de la fase se implementa mediante la aparición de enemigos hasta que se ha matado a una cierta cantidad de cada tipo o se ha llegado a un game over. Para ello se ha creado una clase `Respawn`, a la cual se le añade lo referente a cada tipo de enemigo al crear la fase (se especifica en el XML), es avisada por los enemigos al morir y se encarga de indicar a la fase cuando ha de crear un nuevo enemigo y donde (de forma aleatoria entre 6 posiciones, 3 arriba y 3 abajo, equiespaciadas). También incluye un método que indica el número de enemigos que quedan por matar para terminar la fase y pasar a la interfase.

2.7. Apartado artístico

Todo el material gráfico del juego, excepto la fuente utilizada en los menús, ha sido creado desde cero, pasando siempre por las mismas etapas: concepto, boceto, digitalización, vectorización, animación y creación de hojas de sprites.

Concepto: Desarrollado a mano alzada sobre papel o en software de dibujo. Consiste en trazar la idea del personaje (aunque posiblemente sufrirá modificaciones en los pasos siguientes). Sirve como muestra al equipo de desarrollo para trabajar de manera ágil y poder contextualizar todo el contenido sobre código, ya que cada concepto suponía unas ciertas posibilidades o restricciones para cada actor.

Boceto: Una vez decidido qué concepto es válido se pasa a crear los diseños finales sobre papel, ya más limpios y listos para ser digitalizados.

Digitalización: Realizado bien mediante recreación digital en software de dibujo o vía escáner, permitió tener los diseños originales alteración, listos para ser vectorizados

Vectorización: Se sigue un proceso de vectorización en Adobe Illustrator porque el estilo visual del juego, en colores planos y diseños sencillos permite un trabajo muy ágil. Además al separar las partes articuladas de los personajes

del juego se permite un proceso de animación basado en instantáneas sencillo, y llamativo.

Animación: El proceso de animación consiste en coger el modelo vectorial base de cada personaje y moverlo ligeramente para cada frame, siguiendo el proceso que se usa con las figuras de animación en stop-motion.

Hojas de Sprites: Con el resultado del proceso anterior, el último paso es solo importar a Adobe Photoshop, teniendo cuidado de que el programa no aplicase anti-aliasing que estropease los bordes y los colores perfectamente planos. Esto además dotó a los personajes de un estilo pixel-art muy característico, consiguiendo la estética deseada. Tras añadir el colorKey en el fondo se acopla la imagen a modo de color indexado para poder manipular la tabla de colores programáticamente a posteriori.

2.7.1. Extracción de sprites y animación

Para el proceso de animación de los personajes del juego se rediseñó el sistema de extracción de los sprites de las hojas de sprites de los ejemplos. Debido al comportamiento genérico de la clase actor y de su enorme extensibilidad, si bien usar hojas de coordenadas junto a las imágenes de los sprites era viable, había que hacerlas extensibles a cualquier número de estados en los que el personaje pudiera encontrarse, así como a cualquier número de frames que tuviera la animación de ese estado.

Así pues, cada actor cuenta con una hoja de sprites y una hoja de coordenadas para él, así como otra para sus proyectiles (en caso de tenerlo). Para esto se siguió un formato estricto para que las hojas de coordenadas pudieran ser fácilmente interpretables mediante dos expresiones regulares simples.

La primera línea de la hoja incluye, separados por espacios y ordenados según las constantes de los estados definidas en el fichero de constantes, el número de frames de la animación de cada estado. Separados por dos líneas en blanco están las cuatro-tuplas que especifican la región de la hoja de sprites que representa el frame correcto de la animación. Estas tuplas están agrupadas según la animación a la que se corresponden, y estos grupos a su vez están ordenados según el estado.

Cada tupla se representa como 'x_init, y_init, ancho, alto' donde x_init e y_init son el pixel más arriba a la izquierda de la imagen.

Cuando se carga la hoja de sprites y se "trocea" siguiendo este método, el actor almacena en su atributo graphics una lista de frames de animacion, ordenados según las constantes de los estados del fichero de constantes. Al llamar a update, se itera sobre dichos frames para representar las animaciones.

2.7.2. Detalles de diseño artístico

DR1N: El concepto para DR1N está basado en los miles de robots cuadrúpedos, popularizados gracias al impulso de las impresoras 3D. Esta idea surgió gracias a la facilidad con la que se pueden hacer robots de este estilo. Se ideó un robot desechable que los exploradores del juego pudieran usar y tirar sin problema.

DR1N pasó por varias etapas, aunque las principales referencias usadas en su diseño fueron Kantai Collection, Unknown Synchron y Gundam. Se incluyen diferentes diseños usados antes del concepto final.

Enemigos: Los enemigos pasaron muy rápido a la fase de abocetado. El principal problema fue la animación, que siguió un riguroso estudio de cinemática y mecánica que incluyó ver unos doce vídeos de carreras de camellos. Por ejemplo: <https://www.youtube.com/watch?v=845X9JEGzsU>

Plataformas y fondo: El diseño de los fondos para el desierto y la fase nevada se hicieron directamente en vector debido a su sencillez. Las plataformas se realizaron pixel a pixel directamente en programas de dibujo digital, pese a que tuvieron que ser desechadas por limitaciones del motor (aunque aún figuran en las carpetas de recursos del proyecto).

Menús: Los gráficos de los menús se realizaron íntegramente de forma digital. La fuente usada es Russian Regular. Pasaron solo por los pasos de vectorización y animación (en el título). La animación de introducción se realizó en pyglet y el menú y las pantallas interfase en pygame, usando como fondo la imagen combinada que queda al final de la intro tras aparecer los diferentes personajes.

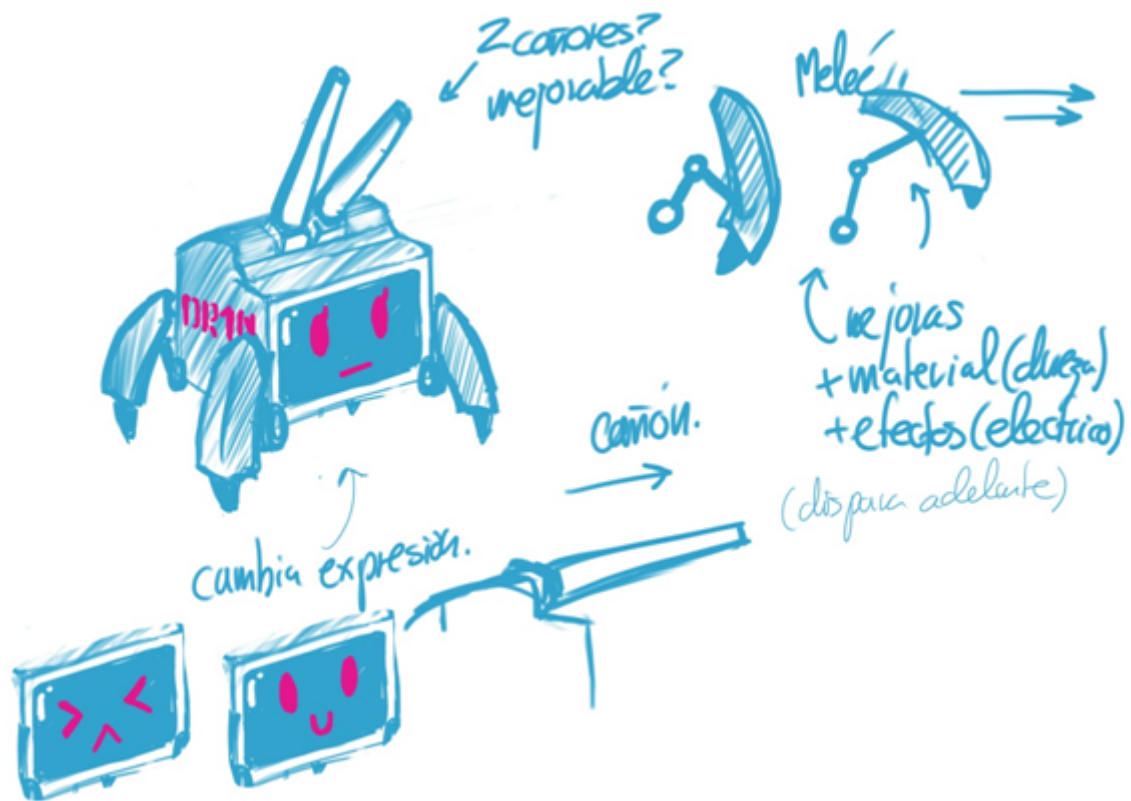


Figura 2.6: Concepto temprano de DR1N, el protagonista



Figura 2.7: Variaci3n del concepto inicial de DR1N

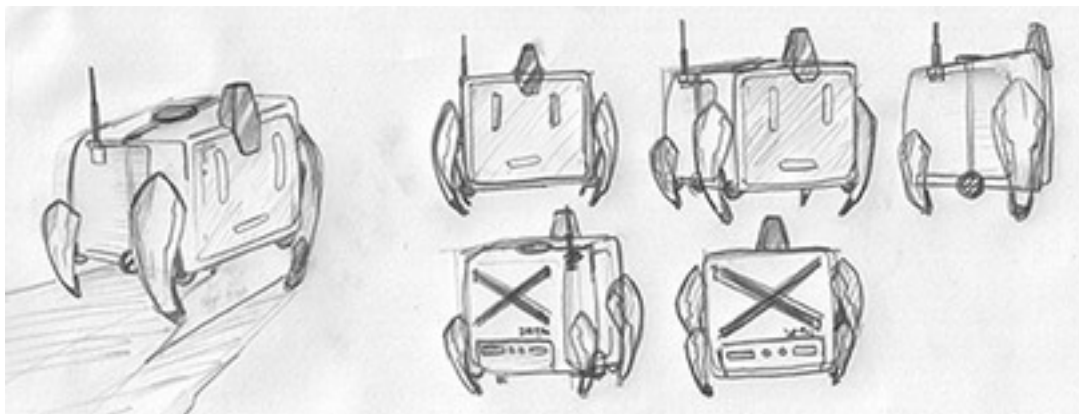


Figura 2.8: Boceto final de DR1N



Figura 2.9: Referencias principales usadas en el diseño de DR1N



Figura 2.10: Bocetos en papel de diferentes enemigos usados en el juego

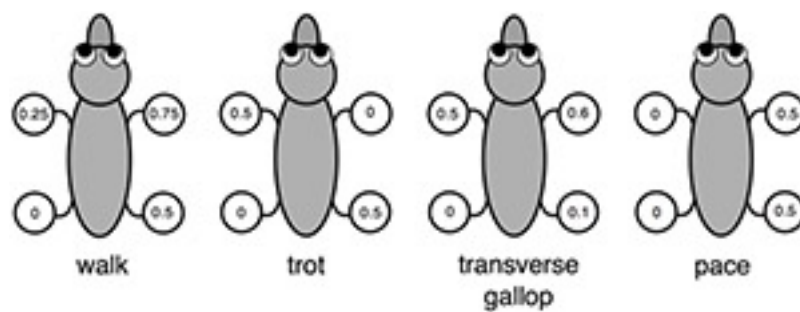


Figura 2.11: Referencia de tiempos de trote usados en la animación



Figura 2.12: Sprites en alta resolución, viables gracias al proceso de diseño vectorial

La imagen de DR1N es una modificación del diseño base y el personaje de la derecha es un boss no implementado (M4L4ND41N, que es DR1N con un cambio de paletas).

2.8. Aspectos destacables

Los aspectos más destacables son:

- Todos los sprites y animaciones son originales. Nada ha sido tomado de internet. Se ha llegado a realizar análisis de carreras de camellos para hacerlos concordantes con la realidad.
- Todo el motor interno del juego ha sido remodelado a partir del código original proporcionado, dando lugar a un motor muchísimo más modular, y por lo tanto, reutilizable.
- De la misma manera que el motor, se ha tratado de hacer que el juego sea fácilmente ampliable, así las fases del juego se definen en ficheros externos, lo que permite ampliar el número de fases sin tocar nada de código.
- En la interfase se ha preparado espacio para mostrar información de la fase recién terminada, la fase que está por empezar y las estadísticas del gameplay.
- La dificultad del juego viene determinada por la configuración de la fase, y por un parámetro de dificultad variable, con el cual se modifica el comportamiento de todos los enemigos, así como sus estadísticas.
- El sistema interno del juego está preparado para, en cualquier momento, definir un conjunto de objetos los cuales serán dropeados a veces por los enemigos al morir, y tendrán efectos en DR1N, como pueden ser curarle, o aumentar su ataque.
- Así mismo, también está preparada la base para un sistema de mejoras que modifiquen los atributos de DR1N de forma permanente.



Figura 2.13: Menú inicial

2.9. Manual de usuario

Al ejecutar el juego, después de una animación, aparece el menú de inicio (Figura 2.9). En dicho menú, se podrá elegir el nivel de dificultad, iniciar el juego, visitar la tienda (aun no implementado) y salir del juego.

Los controles dentro del juego son sencillos:

Movimiento: $\triangleleft \nabla \triangleright$.

Saltar: \triangle .

Disparar: X (Figura 2.9)

Ataque cuerpo a cuerpo: Z (Figura 2.9)

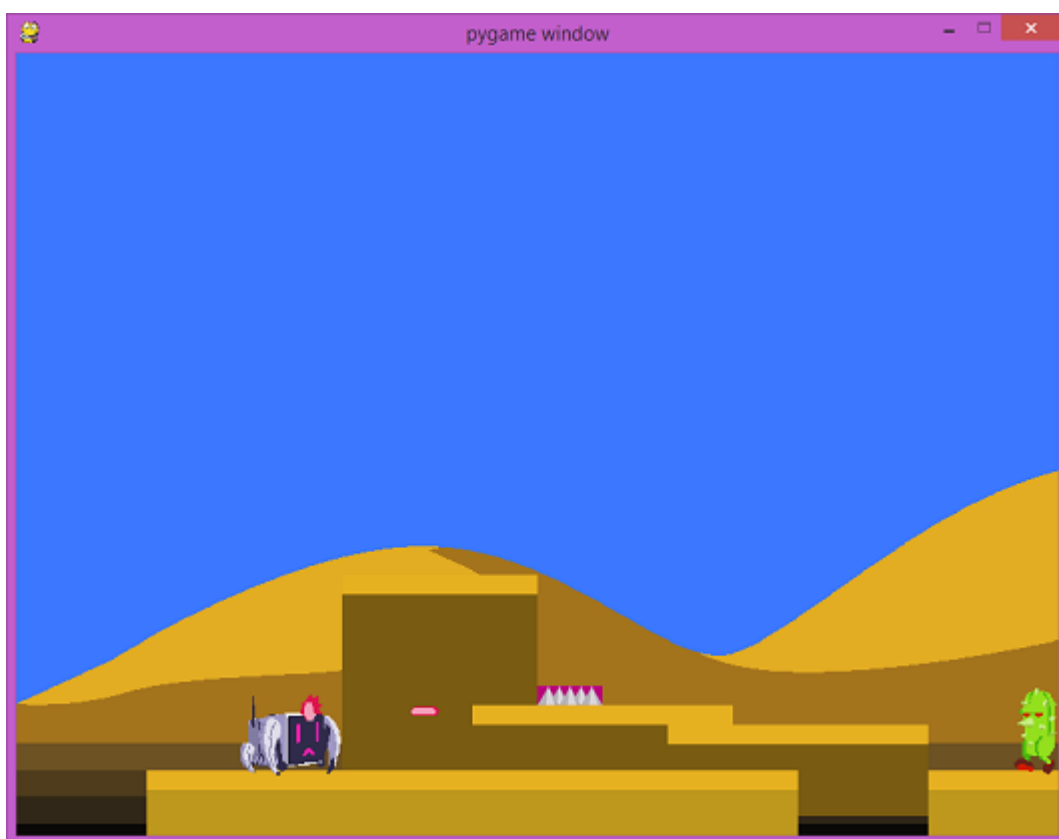


Figura 2.14: Ataque a distancia

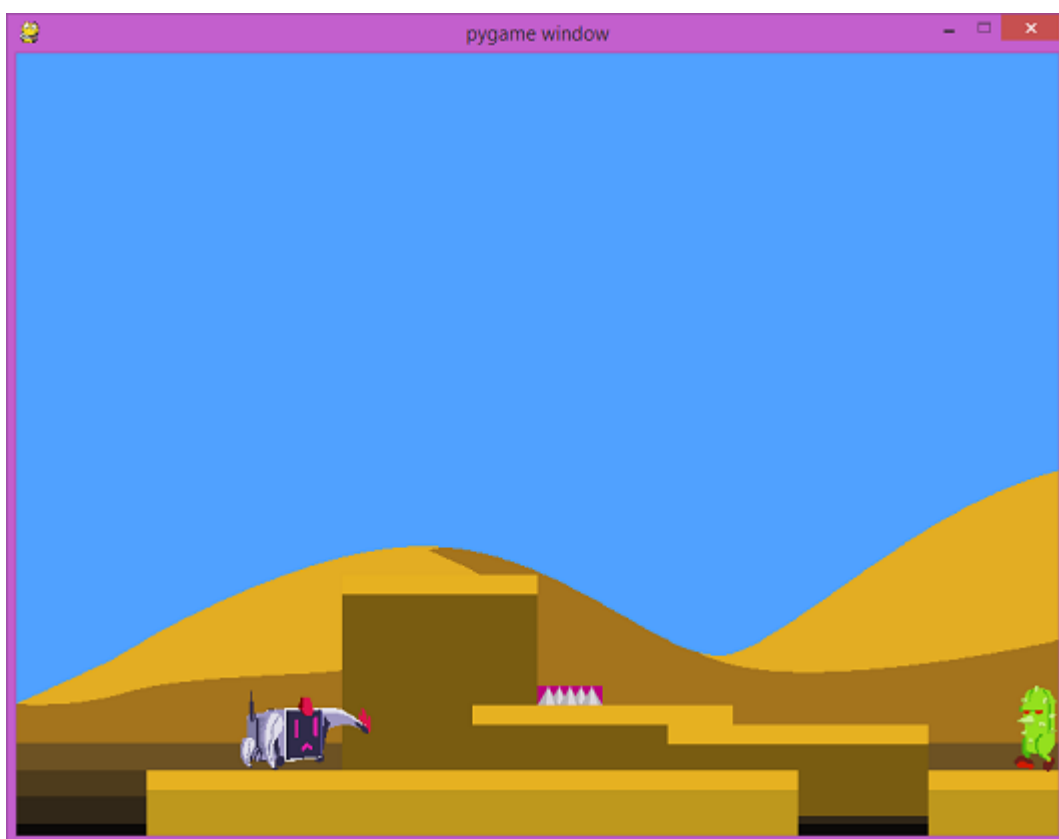


Figura 2.15: Ataque cuerpo a cuerpo

Dentro de la fase, el objetivo es acabar con todos los enemigos sin morir, bien sea por caerse al vacío o porque la vida de DR1N llegue a 0 (porque los enemigos nos han dado o DR1N a tocado una trampa).

Para ejecutar el juego, basta con abrir una consola de comandos dentro de la carpeta DRIN y ejecutar `"python main.py"`.^{Es} necesario tener instaladas las librerías `pygame` y `pyglet`, así como Python 2.7.9

2.10. Trabajo futuro

2.10.1. Motor

Si bien este motor implementa las funcionalidades necesarias para poder hacer un desarrollo más abstracto que simplemente con las librerías de `pygame`, tiene ciertas carencias. En futuras revisiones habría que considerar la revisión del diseño de la clase `State` para intentar ofrecer una interfaz más sencilla (quizá tipo lenguaje de scripting). Así mismo, sería interesante crear un motor de colisiones propiamente dicho (implementadas ahora mismo en fase, como se menciona en el apartado X Escenas y fases).

En cuanto a trabajo descartado, todos los actores tienen implementado, por defecto, un método `drop` sin comportamiento definido, que se llama cuando el actor muere. Con la generación de un nuevo tipo de actor, sería posible implementar un sistema de mejoras, aprovechando para ello la clase `Upgrade`.

La clase `Upgrade` (UML6) implementa un patrón decorador construido alrededor de la clase `Stats`. Básicamente, `Upgrade` es una subclase de `Stats` con todos sus atributos y métodos, y a mayores un atributo de clase `Stats`. `Upgrade` por defecto delega todas sus llamadas a dicho atributo, y mantiene a 0 los atributos heredados. Para crear mejoras, basta con extender dicha clase, darle un valor a cada uno de los atributos y sobrescribir los getters. Esto es, de querer una "HPUpgrade", bastaría con darle un valor a `maxHP`, y hacer que el getter devuelva la suma de este valor y el que proporciona el atributo `stats`. (UML 1)

2.10.2. Enemigos

En este apartado, las futuras actualizaciones podrían ser muy extensas. Con más tiempo se podrían haber implementado habilidades ocultas en los enemigos que realicen cada cierto tiempo (por ejemplo los cactus lanzar ráfagas de pinchos, o que Jetty pusiera muñecos de nieve por el mapa cuando esquivas sus disparos, etc). Incluso podría pensarse en que, a partir de cierto nivel los enemigos entraran en modo furia antes de morir atacando más fuerte, etc.

El patrón de estados subyacente es simple pero eficaz. Aun así, con más tiempo se podría hacer que los enemigos fueran más hábiles. Ejemplos de esto podría ser lo siguiente:

- Que los enemigos sólo salten hacia otra plataforma cuando tienen la certeza de llegar a ella. Esto es, que tengan en cuenta su distancia de salto para poder saltar a plataformas muy alejadas pero muy abajo o que no salten cuando hay una cerca pero arriba y la gravedad se lo impide. Además, en caso de implementar plataformas móviles, se podría complicar bastante.
- En el caso de haber muros en la fase que colisionen con los disparos, tenerlo en cuenta a la hora de disparar y esquivar disparos
- Con respecto a los enemigos cuerpo a cuerpo, podrían introducirse comportamientos aleatorios para que no se comporten todos siguiéndote del mismo modo. Uno podría escapar porque “tiene miedo”, otro saltar para que le acierte un disparo que le iba a dar a otro en vez... Con esto, en el caso de que en las fases de los jefes finales hubiera enemigos normales, éstos podrían “sacrificarse” por el jefe, evitando que DRIN pueda acercarse a él.

2.10.3. Escenas

Arreglar aspectos graficos en: plataformas, el sistema de expulsión de DR1N cuando muere, los proyectiles del camello.

Añadir las fases y enemigos que se planearon en un inicio pero no hubo tiempo a implementar ni dibujar.

Con la tienda funcional, se planea un sistema de mejoras que deberán ser compradas mediante una moneda que se consiga matando en las fases. Para ello se tendría que tener una representación persistente de DR1N (pues actualmente se crea en cada fase). La cantidad monetaria que se consiga con cada muerte dependerá del tipo de enemigo y nivel de la fase, ganándose más según avanzas en el juego y cuanto más rápido acabes la fase.

Incluir a la interfase estadísticas de la fase terminada e información de la siguiente fase como pueda ser el número de enemigos, nivel, mejor puntuación, etc..

Modificar las fases de forma que se generen de forma procedural, manteniendo una concordancia con la zona en la que se esté (desierto, nieve, etc.), y que se diseñe la oleada de enemigos en función de una dificultad seleccionable.

2.10.4. Apartado artístico

Auto-Extracción: Lo ideal sería no tener que escribir a mano estos ficheros, como trabajo futuro se plantea diseñar o bien un script que los genere o un algoritmo que lea los sprites desde la hoja automáticamente sin pre-cargarlos.

Plataformas: Los gráficos de las plataformas son imágenes de colores planos, aunque se incluye una carpeta con gráficos más complejos que habría que emplear a modo de "tiles" (como se explica en la sección de Fases y Escenas). Dado que el proceso original simplemente estiraba estas imágenes para rellenar toda la superficie, se ha optado por esta opción más sencilla. Sin embargo, debido a que es poco vistosa, en el futuro se planea diseñar un algoritmo de tileado que rellene las plataformas para que estas tengan una cierta textura.

Paletas de colores programáticas: Al trabajar en un espacio de colores indexado para las hojas de sprites, en el futuro se planea crear variaciones del personaje principal y de los enemigos simplemente cambiando su esquema de colores y tamaño programáticamente para reflejar sus estadísticas.

Overlapping de Sprites: Para lograr mejores resultados se plantea como trabajo futuro la técnica de superposición de mini-sprites para lograr un resultado más dinámico y más expresividad en los personajes. Esto se basa en que cada actor es un conjunto de sprites animados, cada uno a su propio ritmo y solapados,

creando la imagen final del personaje. Esta técnica permite cosas tales como que el personaje parpadee sin tener que crear sprites específicos del personaje parpadeando, ya que la expresión facial estaría superpuesta sobre la cara y esta podría cambiar independientemente del estado o la animación en la que se encontrase el personaje.