Memoria del videojuego Drylands:Kingdom Across

Marino Tejedor, Álvaro de las Heras, Adrián Sánchez, David Valdevira

Contenido

1.Introducción

El videojuego que hemos desarrollado para la asignatura es Drylands: Kingdom Across. Sin extender mucho el documento de diseño, que se puede ver en este enlace, se trata de un videojuego de tipo dungeon crawler de perspectiva aérea. En él vamos explorando cada sala en un mapa generado de forma procedimental hasta encontrar nuestro objetivo. La historia jugable se ha quedado bastante corta en comparación con lo previsto, como vamos a poder ver más adelante. La mecánica de combate es simple con dos habilidades, y el movimiento también es simple, fácil de acostumbrarse a él.

Como todos los grupos hemos utilizado el motor Slick2D para desarrollarlo, con ayuda del software TiledMapEditor y de su librería Java. Como IDEs hemos utilizado NetBeans e IntelliJ, y en el apartado artístico GIMP y Audacity.

Visualmente hemos querido imitar un juego retro con gráficos PixelArt, dado que no teníamos tiempo o recursos suficientes para desarrollar sprites mejores, además de que la mayoría de los hemos tomado de recursos libres en Internet. En el apartado sonoro hemos acompañado con obras de 8-bits para ayudar a toda la integración del videojuego.

El resultado es un videojuego completo, pero que hemos tenido que recortar en bastantes aspectos para poder entregarlo a tiempo. En las conclusiones incidiremos en las causas.

2. Expectativas y progreso

A lo largo del desarrollo hemos tenido bastantes ideas diferentes de lo que iba a ser nuestro videojuego. El primer enfoque que nos gustó fue hacer un vista aérea pixel Art del estilo dungeon crawler. A partir de aquí se nos abrió un mundo de posibilidades y tuvimos bastantes ideas, que debatimos por los Issues de Github y en persona en las reuniones.

Algunas de estas ideas fueron incluir una jerarquía de clases o profesiones para el jugador amplia, incluir combate cuerpo a cuerpo y a distancia, salas con tesoros, posibilidad de subir de nivel, diferentes bosses, ...

Al final la mecánica se quedó en recorrer mazmorras y un combate cuerpo a cuerpo con “estocadas” y bloqueos, no ha dado tiempo de implementar más cosas.

Además tuvimos ideas más avanzadas que no se encuentran en juegos similares a lo que queremos que ya están en el mercado (o no en la mayoría), como el falso 3D con efecto paralelaje , el modo online o la generación de mapas.

Sobre el modo online la idea era simple, un servidor que reenviase los mensajes entre medias, y los clientes mandando y recibiendo datos para sincronizar el juego en un socket abierto en un hilo independiente. El problema era de concurrencia, aunque la idea parecía simple. También había librerías compatibles con lwjgl que podíamos usar. No llegamos a hacer esto por falta de tiempo

El paralelaje era una idea para dar impresión de 3D que se nos ocurrió para la fase de pirámide. La cuestión es que en un plataformas es mucho más fácil implementar un fondo que avance a diferente velocidad, logrando un resultado muy bueno. En un perspectiva aérea no es tan directo, pero pensamos que podía ser una buena idea en casos como nuestro mapa de pirámide.

Básicamente, al ascender por la pirámide, veríamos los mapas que hemos generado de forma procedimentaria para los niveles inferiores a una distancia mayor, al asomarnos por los bordes. No hemos podido hacerlo porque no hemos implementado la historia jugable a ese nivel.

Respecto a la generación de mapas, fue lo primero que hicimos, pues si no la exploración no tenía sentido al largo plazo si los mapas eran estáticos, lo detallo en un apartado posterior.

3. Desarrollo del videojuego

La parte jugable comprende la primera mazmorra que tenemos que explorar. El núcleo del juego se basa en una serie de clases desarrolladas.

3.1 Clase Player

La primera de todas es la clase Player. En un primer momento era una clase independiente, aunque luego se portó casi todo su contenido a una clase Personaje , de la cual heredarían sus contenidos Player y Enemigo.

La clase Player describe al jugador en una posición absoluta del mapa y del juego. Como esto es muy amplio, describo por partes de todo lo que se ocupa.

3.1.1 Posición

La posición se describe en cuatro floats, a partir de 0. Como se describe la posición en términos absolutos desde el principio del mapa, necesitaremos representarlo en función del offset del mapa, como veremos más tarde.

Dos de estos floats representan la posición real, uno en el eje X y otro en el eje Y, y serán los que representen al jugador y se devuelvan con los getters.

Los otros dos son pasos temporales, que permiten calcular una nueva posición y verificarla. Gracias a esto podemos verificar colisiones y actualizar la posición en función del contexto. El funcionamiento básico entonces es actualizar la posición futura en los pasos intermedios, comprobar si en esa posición se encontraría un muro o un enemigo, y actualizar la posición o resetearla en función de ello.

Además, al actualizar la posición llevamos la cuenta del ángulo en el que se está moviendo para luego aplicarlo en el retroceso.

Por último actualizamos un boolean para saber en qué dirección está mirando y mostrar una animación u otra.

3.1.2 Vida

Tenemos en un atributo la vida del personaje, y métodos getVida y damage para quitarle los puntos de daño que reciba.

3.1.3 Animación, sprites y render

Pasamos por argumento los sprites utilizados para tomar las animaciones de idle, run y otras secundarias como la de retroceso.

Ahora con estas animaciones y sabiendo el estado del jugador, tenemos un método getAnim() que devuelve la animación que hay que renderizar en cada momento.

He añadido un método estático para dar realismo al videojuego que renderiza los personajes ordenados en el espacio. Es decir, muestra por encima a los personajes que están delante. Para ello recibe una lista de personajes a renderizar, los reordena por su posición en el eje Y, y luego los renderiza en ese orden, haciendo ese simple truco conseguimos bastante profundidad.

3.1.4 Movimiento

Como hemos visto antes, la posición queda determinada por los cuatro floats posx, newx, posy, newy. La forma de movernos será siempre la misma, obtener la dirección del movimiento en función del estado (sin movimiento en el block, hacia un target cuando atacas, hacia el ratón cuando corres), la distancia máxima que puedes recorrer para los milisegundos que han pasado (que es delta * velocidad) , y llamar a un método que establece la nueva posición temporal teniendo en cuenta que va a moverse en la dirección que indica el sitio hacia donde queremos ir pero como máximo la distancia indicada.

Estos cálculos los facilita mucho el uso de la clase Vector2f de Slick2D pero hay parte hecha matemáticamente por desconocimiento de esta utilidad.

3.2 Clases Personaje y Enemigo

A raíz de ver que el resto de NPCs se podían regir por la misma lógica que el jugador, traspasamos las funcionalidades de Player a Personaje de forma que sirviese de clase padre para el jugador y los enemigos.

La diferencia entre ambos estará en la actualización del movimiento, que en los enemigos se hará por IA, y en las habilidades y estadísticas que tengan cada uno.

3.3 Clase Mapa

La clase mapa implementa el mapa de tiles, la cámara y la detección de eventos. Principalmente el constructor obtiene dos mapas, el principal y el de información, la resolución en la que se ejecuta el juego.

3.3.1 Cámara

La cámara se define por el offsetX y el offsetY, que son dos floats que indican (en negativo) desde donde se empieza a renderizar en pantalla.

Para que la cámara se actualice, tiene dos modos, centrada en una sala, y libre. Por defecto la cámara sigue al jugador con unos milisegundos de desfase para que el movimiento sea natural y no coloque al personaje constantemente en el medio. Lo que hacemos es intentar que el jugador esté centrado en la pantalla, y modificar los offset una distancia máxima para conseguirlo. Lleva unos contadores internos que van aumentando y disminuyendo con las deltas en función del tiempo de desfase para que el movimiento sea suave y gradual.

Por otro lado, la cámara se centra en las salas. Para saber si estamos en una sala o no, utilizamos una capa de tiles del mapa, la tercera (capa 2 contando el 0) que define la zona que es sala, y su extensión. Al entrar el jugador en una zona de cámara centrada (lo sabrá porque las variables centroX y centroY están a 0) busca en centro de la sala contando tiles en horizontal y vertical. Con este centro la cámara ahora quedará fija para que lo que esté en el centro de la pantalla es la sala y no el jugador. El propósito de esto es que la jugabilidad en combate sea más cómoda.

3.3.2 Lista de centros y salas

Además del centro de la sala actual, guardaremos los centros que vayamos encontrando. Así sabremos si la sala que visitamos es nueva o no, para invocar enemigos cuando la sala es nueva.

También tiene información de cuanto miden las salas y por ello tiene un método en el que devuelve puntos aleatorios de esa sala como potenciales posiciones de enemigos.

3.3.3 Minimapa

Como juego dungeon crawler, tener información de por dónde has pasado ya es crucial. Para ello tenemos el mapa info que hemos indicado antes. Es un mapa vacío en el cual vamos a ir guardando la información del suelo por donde vamos pasando. Es tan simple como localizar los tiles que estamos viendo sabiendo el offset del mapa y la resolución de la pantalla, y copiar esos tiles al TileMap auxiliar.

Lo único que falta es poder ver el mapa entero. Reducimos la escala con una regla de 3 sabiendo las dimensiones del mapa y las dimensiones de la pantalla, en las dos dimensiones, quedándonos con la escala más pequeña, que nos permitirá ver el mapa completo. Con todo esto ahora podemos pulsar la tecla que le hemos asociado, que es el tabulador, y el render del juego en vez de mostrar toda la partida, llamará a un método del mapa que renderiza el minimapa que llevamos explorado a pantalla completa con la escala calculada. Para completar, en el juego principal y sabiendo la escala en la que se muestra el mapa, podemos mostrar la posición del jugador con la posición real multiplicada por la escala, con una animación simple de un punto.

NOTA: Este mapa no se guarda, así que en vez de recurrir a la librería que genera el mapa y lo guarda (más lento) hemos añadido un mapa de bits con un boolean por cada tile que indica si lo hemos visto o no. De esta forma al cerrar el juego y volver a abrir podremos regenerar el minimapa. Lo explico en profundidad en la clase guardado.

3.4 Clases Habilidad y derivadas

Las habilidades se implementan en clases independientes, que pueden ser usadas tanto para enemigos como para el propio jugador. La idea inicial era hacer una gama de habilidades amplia, pero al final hemos implementado un ataque y un bloqueo. Para dotar a un personaje de una determinada habilidad basta con crear un nuevo objeto de la clase que queramos dentro de la clase de personaje que la vaya a utilizar

3.4.1 Base

La base de la habilidad incluye el enfriamiento (establecer el máximo y contar el CD) así como métodos abstractos para terminar y llamar a la habilidad en cuestión.

Tiene también un método para sondear si está activa, y llamar al método de update correspondiente.

3.4.2 Dash

El dash es la habilidad de ataque, en la que nos lanzaremos hacia la posición que marque el ratón (si está dentro de su rango) en una estocada. En el constructor podemos personalizar el rango de acción, la velocidad a la que atacamos, … Tiene un método update en el que la posición se calcula de forma muy similar a como andamos, solo que se dirige hacia un objetivo targetX, targetY que queda definido fijo al canalizar la habilidad. Una vez llegue a ese objetivo empezará a bajar el enfriamiento, y deceleraremos con un contador hasta que estemos parados, donde termina la habilidad. De la misma forma que baja el contador con la delta del update, crece al acelerar cuando el personaje se lanza, hasta que alcanza la velocidad máxima.

3.4.3 Block

El bloqueo es la habilidad defensiva. El personaje que la realiza se endurece mientras la esté usando hasta un tiempo máximo definido en el constructor. Después de ese tiempo o si el jugador levanta la tecla correspondiente (o si un enemigo dejase por IA dejar de usarla) empezará a contar el cooldown de la habilidad. Hemos diseñado una animación específica para esta habilidad, de aspecto metálico en la que el personaje se endurece mientras bloquea.

3.4.2 Combo

No es una clase derivada de habilidad, pero la complementa. Para dar más juego y premiar la habilidad hemos implementado una mecánica en la que llevamos la cuenta de los golpes seguidos que hemos acertado (o ataques bloqueados) y reducimos el enfriamiento de las habilidades en la misma medida. Como el combo es de un jugador y no de una habilidad, creamos un objeto de tipo combo en el jugador y lo pasamos a las habilidades en el constructor para que todas las habilidades contribuyan a sumar o resetear esta racha.

3.5 Clase Guardado

Para salvaguardar el estado de la partida en las mazmorras usamos la utilidad SavedState incluida en Slick2D. Para usarla, en el constructor de nuestra clase guardado tenemos un objeto SavedState que inicializamos con el nombre de la partida que queramos cargar. De momento cargamos siempre la misma partida, pues al iniciar la partida llamamos siempre a “partida”, pero podríamos seleccionar varias simplemente pidiendo una cadena y pasándola al constructor. A partir de aquí tenemos:

3.5.1 Estado de la partida

Con dos métodos set y get sabemos si hay una partida guardada o no. El algoritmo es simple, recurrimos a un campo del SavedState que es “estado”. Para crear una nueva y dejarla establecida vale con poner este campo a “cargada”. Para comprobarla lo leemos, con el campo por defecto por valor “nueva”. Si leemos “nueva” es que no hay partida. Esto puede parecer trivial, raro o poco eficiente, pero permite forzar una partida a nueva solo con cambiar un campo, y en estos casos poder recuperar forzadamente una partida borrada sin querer.

3.5.2 Guardar

Para guardar tenemos dos métodos, uno que toma por parámetro un jugador y guarda todos sus estados, y otro que hace lo mismo con el mapa. Particularmente éste tiene bastante más complicación, pues hay que guardar dos informaciones problemáticas.

Una de ellas es la lista de centros. A priori no sabemos qué tamaño tiene, y no podemos guardar en un SavedState un array, de forma que la primera forma que surgió fue guardar centroXn y centroYn

con en un bucle for, de forma que al leerlos si usamos un valor por defecto que no se vaya a dar nunca, como x=0,y=0, sabemos si hemos acabado.

Lo más complicado era el minimapa. La solución que utilizamos fue usar un mapa binario donde cada bit represente a un tile, indicando si se ha visto o no. Este mapa de bits podemos pasarlo a un bytearray, que codificamos después en Base64 y guardamos en una cadena:

```
byte[] encoded = Base64.getEncoder().encode(bitmap.toByteArray());
return new String(encoded);
```

Esta cadena si podemos guardarla, de forma que para guardar información implementamos una función miembro de la clase mapa que nos de esta string que codifica el minimapa y la guardamos en el SavedState.

Otro punto problemático es que queríamos implementar un autoguardado. Cuando guardamos la partida a petición del usuario y notamos una caída de frames no debería pasar nada, pero si queremos guardar automáticamente y el juego va a ir mal, entonces se convierte en algo no deseable. Lo que hicimos fue dividir la solución en dos partes. Por una, guardamos solo cuando es estrictamente necesario, es decir, cuando conseguimos algo, que es limpiar una sala de monstruos(en el update principal del juego). Por otra, este método de guardar mapa que es el más pesado en entrada/salida, se lleva a cabo en un hilo aparte de forma que no bloquee el bucle principal del juego.

3.5.3 Cargar

Igual que guardamos los datos de jugador y mapa, cargamos la información de la misma manera, con getFloat o getString.

La mejora que hacemos en este aspecto es que vamos a cargar datos solo si existen. Lo que hacemos es llamar a los get de SavedState con su propia información por defecto. De esta manera si no encuentra un determinado valor por algún error inesperado retorna lo que tenía ya el objeto. Por todo esto lo que haremos será crear un jugador con sus parámetros por defecto para la ocasión y después intentar cargarlo si la partida no es nueva.

3.6 Clases para la Narrativa

Para las escenas en las que hay diálogo, y para añadir profundidad en la parte jugable, hemos desarrollado dos clases de utilidades. Ambas se basan en la clase Frase, que guarda una frase y la referencia del personaje que la dice. De esta forma para contar la historia tenemos que instanciar a los personajes correspondientes y asignarles las frases.

3.6.1 Diálogo

Guarda un ArrayList de frases, que va a reproducir por orden. Para que sea estético, vamos mostrando carácter por carácter la frase, centrada siempre debajo del personaje con la información del objeto frase y el tamaño que nos devuelve la fuente. En el update actualizamos los caracteres que vamos mostrando, y reproducimos un sonido de tecla cada vez que mostramos uno nuevo. Otro método se ejecutará cuando pulsemos la tecla intro desde el juego principal, y pasará la frase si ya ha terminado todos los caracteres, o aumentará la velocidad al disminuir a la mitad los milisegundos por carácter. Si el diálogo ha terminado, eleva una EOFException, que indica al bucle principal del juego que ha terminado, para actualizar el estado y poder mover al personaje, atacar, etc.

3.6.2 Pensamiento

Siguiendo el mismo mecanismo que antes, tenemos una clase que sobre un personaje, en este caso el jugador, renderiza una frase con los pensamientos que va teniendo. La diferencia es que no está pensada para ser bloqueante, sino para que aparezcan aleatoriamente. Para que funcione se añaden frases a un arraylist de frases "normales" que suelta aleatoriamente, y otras frases que se añaden a arrays especiales que dan información de su vida, que no se solapan, sino que se escoge una del array que se corresponda con el nivel de salud actual del personaje.

Esta es una forma de empatizar con el personaje y también de contar detalles de la historia no transcendentes.

3.7 Generación de mapas

Ha sido una de las partes que más tiempo ha llevado, por la investigación y por la programación, pero al final funciona bien y da mucha vida al juego. Implementaré en el paquete MapasProcedurales las funciones que llevan a cabo este propósito.

3.7.1 Mapa lógico

Lo primero que hacemos es generar un mapa lógico en una matriz del tamaño que le indiquemos con un número de salas, que no se toquen entre sí. Hacemos un array de objetos Célula (clase que describe cada celda y que indica qué contiene, si está conectada, etc.) y con generadores aleatorios de números colocamos las salas comprobando que no estén adyacentes. Esta parte era fácil, nos enfrentamos ahora al primer problema que nos ofrece esto, que es unir las salas.

Tras pensar en muchos algoritmos, elaboro (no sé si ya existe, que es probable, pero no lo he encontrado en ningún sitio) uno que me permite conectar todo con las rutas más cortas posibles. Para ello, lo que hago es llevar una cuenta de los grupos aislados que hay, que en un principio serán tantos como salas haya. Ahora iteramos por los grupos uniéndolos (buscamos la distancia más corta entre dos grupos cualesquiera, si es una línea recta se traza, y si es diagonal se elige aleatoriamente un vértice para trazar dos líneas) hasta que solo quede exactamente un grupo, es decir, que el mapa se pueda recorrer por completo. Después de esto realizo una pasada de seguridad por toda la matriz para comprobar las conexiones ya que a veces se dan bugs de que el objeto célula indica que está conectado otro que no existe.

3.7.1 Mapa gráfico

Ahora que tenemos el mapa lógico necesitamos pasarlo a un TMX. La primera aproximación pasó por hacer ingeniería inversa sobre el archivo tmx genérico, que es en esencia un archivo en formato XML que contiene la información como sigue:

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.0" tiledversion="1.1.4" orientation="orthogonal" renderorder="right-down"
width="10" height="10" tilewidth="16" tileheight="16" infinite="0" nextobjectid="1">
 <tileset firstgid="1" name="0x72_16x16DungeonTileset.v4" tilewidth="16" tileheight="16"
tilecount="256" columns="16">
  <image source="0x72_16x16DungeonTileset.v4.png" width="256" height="256"/>
 </tileset>
```

```
 <layer name="Capa de Patrones 1" width="10" height="10">
  <data encoding="csv">
0,0,0,0,51,51,0,0,0,0,
0,0,0,0,51,51,0,0,0,0,
0,0,33,34,51,51,34,35,0,0,
0,0,51,51,51,51,51,51,0,0,
34,34,51,51,51,51,51,51,35,35,
51,51,51,51,51,51,51,51,51,51,
0,0,51,51,51,51,51,51,0,0,
0,0,51,51,51,51,51,51,0,0,
0,0,0,0,51,51,0,0,0,0,
0,0,0,0,51,51,0,0,0,0
</data>
 </layer>
 <layer name="Capa de patrones 2" width="10" height="10">
  <data encoding="csv">
0,0,2,3,0,0,1,2,0,0,
0,0,18,19,0,0,17,18,0,0,
0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,
25,25,0,0,0,0,0,0,25,25,
0,0,0,0,0,0,0,0,0,0,
0,0,25,25,0,0,25,25,0,0,
0,0,0,0,0,0,0,0,0,0
</data>
 </layer>
</map>
```

Las primeras líneas son información del tamaño del mapa, de los tiles, y la fuente que se ha usado para los tiles, pero lo importante es que los mapas están descritos por matrices de números, donde el número no es más que el tile usado, en el orden que los importa de la fuente (si es 0 es que está vacío, por eso el campo firstgid del archivo fuente de tiles). Desafortunadamente los juegos de slick2d necesitan un formato base64-gzip y no esta notación en csv, que por cierto ocuparía muchísimo sitio para un mapa completo de nuestras características. Este archivo se vería así:

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.0" tiledversion="1.1.4" orientation="orthogonal" renderorder="right-down"
width="10" height="10" tilewidth="16" tileheight="16" infinite="0" nextobjectid="1">
 <tileset firstgid="1" name="0x72_16x16DungeonTileset.v4" tilewidth="16" tileheight="16"
tilecount="256" columns="16">
  <image source="0x72_16x16DungeonTileset.v4.png" width="256" height="256"/>
 </tileset>
 <layer name="Capa de Patrones 1" width="10" height="10">
  <data encoding="base64" compression="gzip">
   H4sIAAAAAAAACmNgQAXGUEwIEFKnCMRKSOpAbGU85qBjGFBCMwcdK0MxLnlc5hKSJ1UdunpCAJc6AJ2YLPiQAQAA
  </data>
 </layer>
```

```
 <layer name="Capa de patrones 2" width="10" height="10">
  <data encoding="base64" compression="gzip">
   H4sIAAAAAAAACmNggAAmIGZmQABGqBg6EAJiYSS+IFRsqABJKCZXnlh95JoDAHGPpZaQAQAA
  </data>
 </layer>
</map>
```

Tras muchos experimentos de ingeniería inversa nos dimos cuenta de que lo que había que hacer para codificar la información era pasar la matriz a un array de integers (necesariamente de 4 bytes) en little endian, y después comprimir ese bytearray, y mostrar la salida en una cadena Base64.
Con esto ya teníamos todo listo, pero al mismo tiempo un compañero del grupo descubrió que en el repositorio de GitHub de los creadores de Tiled, hay una librería para lenguaje java que permite manipular mapas igual que el programa gráfico. Esta librería resuelve el problema de parseo de la información y la codificación pues tiene librerías para abrir y escribir TMX en el formato que queramos de forma transparente a nosotros, y además nos ofrece escribir toda la información de forma cómoda, ya que lo que alteramos nosotros son objetos, no cadenas.
La complicación que tuvimos fue obtener esta librería y la documentación, ya que usaba Maven y al ser un software que no conocíamos tardamos bastante.
Solo quedaba ahora hacer presets de las salas y los pasillos, y colocarlos con esta librería según indicase el mapa lógico para después escribir el mapa en el formato base64-gzip que pide Slick2D.

4. Implementación conjunta de todo el juego

La programación del juego en conjunto se basa en un StateBasedGame y en diferentes estados que programamos de forma hasta cierto punto independiente. De esta forma tenemos un estado inicial, que es el menú de inicio, y podemos acceder a una nueva partida, o cargar la existente. En sí estos no son estados, sino que comprobarán el estado de la partida y dirigirán a nuevos estados en consecuencia.

Los estados que implementamos además del menú de inicio son las diferentes escenas que nos introducen en la historia, hasta el estado CoreGame.java que es el acto 1 de la historia jugable. Todas las escenas implementan en mayor o menor medida las clases que hemos indicado anteriormente. A grandes rasgos hay escenas con fondo negro y letras bajando que no son más que imágenes, para contar por ejemplo el contexto de la historia. Después están las escenas interactivas, que se dan sobre mapas y con jugadores. Haciendo uso de la utilidad diálogo contamos la escena a partir de los personajes que generamos. Por último tenemos estados jugables (sólo hemos llegado a hacer uno) en los que implementamos todas las clases (menos el díalogo). Para los mapas comprobamos si la partida existe o si existe un mapa, para generar uno o para seguir con el que ya existía.

Además de todo esto, hacemos todos los updates necesarios, comprobamos colisiones, determinamos el resultado del combate cuando dos personajes colisionan, comprobamos eventos, etc. En el render mostramos todos los personajes y el mapa según hemos indicado antes. También es en este método que renderizamos la interfaz de usuario, que se resume a la barra de vida, y los iconos de habilidad con el cooldown restante. Solo queda mostrar efectos de la estela en los ataques, que se lleva a cabo con líneas degradadas a transparentes.

5. Sucesión de eventos

El juego arranca en un menú que nos permite iniciar una nueva partida o cargar una existente. El botón de cargar lógicamente carga la partida en el estado que lo dejamos, y la opción de nueva partida borra la existente y comienza la aventura.

Lo primero que vemos es un párrafo sobre fondo negro con la explicación del contexto en el que se sitúa el videojuego. Éste se ha dejado sin modificar, de forma que sigue haciendo una civilización ancestral que tras una guerra dejó un desierto y ruinas con los restos de sus descubrimientos. Dos reinos quedan separados a cada lado, de los cuales nuestro protagonista comienza en Auria. Después podemos ver como nuestro personaje es expulsado del reino. Todavía el jugador no debería entender nada, salvo que es una expulsión injusta. Tras esta escena tenemos un tutorial muy básico, y después nos remontamos unas horas atrás.

En ese tiempo el jugador se ha adentrado en la mazmorra como explorador oficial del reino que es. De ahora en adelante es todo jugable, hasta que encuentras el final de la mazmorra. Entonces otro explorador nos traiciona justo cuando descubrimos el portal que comunica los dos reinos y nos lleva a la superficie, para el juicio, y ahora el jugador entiende lo que ha pasado.

Expulsan a nuestro protagonista, que deambula por el desierto hasta que encuentra la pirámide. A partir de aquí hemos considerado las opciones de dejar el final del juego abierto, o de contar la historia que conllevaría el descubrimiento de la pirámide.

Dado que teníamos pensada toda la historia y que estaba bastante entretenida a nuestro parecer, la hemos contado en un fondo negro con letras a modo de epílogo, para dar luego paso al final.

6. Expectativas, resultado y conclusiones

Las expectativas que teníamos desde nuestro punto de vista desinformado e inexperto eran bastante grandes. En este sentido, pensamos en muchas habilidades, efectos, progresión del personaje, clases o trabajos, una historia más larga, etc.

El resultado es bastante diferente de nuestra idea inicial. Para nada está mal, y de hecho ha tenido muchas horas de trabajo, programación, investigación, así como de planificación y de “brainstorming”. Pero si es cierto que planificamos bastantes cosas y no hemos podido implementar todo lo que nos gustaría.

Las causas son varias. Para empezar, no teníamos ningún contacto nadie del grupo con el desarrollo de un videojuego y todo lo que ello supone. Por esto creíamos que nos iba dar tiempo cuando realmente íbamos a tardar bastante. Además, algunas de las ideas que propusimos en reuniones o en github no teníamos muy claras como llevarlas a cabo de forma técnica. Al igual que comento esto, también es cierto que nos han faltado horas de trabajo en el grupo y que realmente podríamos haber conseguido sacar un poco más de lo que hemos conseguido.

En otro orden de cosas también parte de esta falta de horas puede deberse a que yo (Marino Tejedor) como jefe de proyecto podría haber repartido mejor el trabajo y motivado más al grupo para que trabajase. Al margen de esto debo decir también que en cierto momento del desarrollo se perdió el propósito inicial del videojuego, que es entretener, y nos quedamos solo en lo superficial.

Por la parte técnica ha habido dos cosas que nos han retrasado notablemente. La primera es que al no conocer el motor, varios del grupo hemos programado cosas que ya estaban en las utilidades de Slick2D. Es por ejemplo el caso del movimiento, que al ser de lo primero que hicimos, no sabíamos del vector2f, y lo hicimos manualmente. También es el caso del menú, en el cual se capturan los clicks sobre los botones con la posición del ratón, procedimiento muy malo comparado con utilizar los componentes que nos ofrece Slick2D así como sus utilidades de MouseOver.

La segunda ha sido la generación de mapas procedimental. Esto a diferencia de lo anterior no ha sido para nada un error, pero ha consumido mucho tiempo. No había hecho nunca nada parecido, y por ello he invertido mucho tiempo en pensar todas y cada una de las fases. Además como he dicho antes, también ha llevado mucho tiempo la ingeniería inversa de los TMX, encontrar la librería de Tiled y conseguir añadirla a nuestro proyecto por el tema de Maven, etc. El resultado de este motor de generación de mapas, eso sí, ha sido un éxito, pues genera para cualquier tipo de tiles, cualquier tamaño y número de salas un mapa jugable que cumple con todos los requisitos. No solo es un detalle, aporta profundidad a nuestro juego ya que en nuestra historia los templos que dejaron los ancestros se protegen cambiando de forma cada vez que sales y vuelves a entrar.

En conclusión, hemos tenido un contacto bastante instructivo sobre todo el proyecto y la orquestación que supone hacer un videojuego. Hemos aprendido sobre todo la complicación de las técnicas que vemos comúnmente pero sin darnos cuenta de cómo se hacen y del trabajo que llevan. También hemos visto todas las carencias que tenemos al intentar abarcar algo tan amplio que necesita de artistas de diferentes tipos, diseñadores, programadores, etc.

Abstrayéndonos del videojuego, si bien yo por lo menos no estaba contento con el motor o con utilizar Java, al final de la asignatura tengo que decir que he aprendido a usar Java cómodamente y me ha acabado gustando la forma de trabajo del lenguaje.

Por último y aunque no es algo nuevo pero si escaso en las asignaturas de la universidad, buscar soluciones y salidas de forma independiente a problemas que cada grupo hemos tenido según ha salido surgiendo, buscando algoritmos, librerías, ayuda en otros programadores, … también ha sido muy enriquecedor.

Esto es todo, solo queda desear que el juego sea divertido, que es el fin último, y que guste tanto por la jugabilidad como por la historia.