Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de la Universidad

Modelación y Computación Gráfica para Ingenieros

Square Knight

Juego en 2 Dimensiones: Propuesta Personal.

Integrante: Daniel Soto G.
Profesora: Nancy Hitschfeld K.
Auxiliares: Pablo Pizarro R.
Pablo Polanco
Ayudantes: Joaquín T. Paris
Rodrigo E. Ramos T.
Sergio Leiva
Fecha de realización: 11 de mayo de 2017
Fecha de entrega: 14 de mayo de 2017
Santiago, Chile

Resumen

Este informe trata sobre el desarrollo de un juego arcade de accion y plataformas. Se usaron funciones de colisión incluidas en pygame y se implementaron distintas funciones para colisionar con el nivel. Se generan además power-ups al azar y enemigos con una IA básica. Los objetivos principales fueron implementados con una serie de clases que permiten la extensión a otras estructuras de niveles y nuevos tipos de enemigos o power-ups. Se usan librerías de Python, pygame y shapely. Leer `README.md` para instrucciones de uso e instalación.

1. Introducción

El problema planteado es generar un juego de acción y plataformas en dos dimensiones. El juego debe permitir al usuario controlar a un personaje con total libertad para moverse dentro de la pantalla. Este personaje debe ser capaz de moverse hacia los lados, saltar y atacar. En la escena deben aparecer enemigos al azar, los cuales tienen que ser capaces de seguir al jugador, sin importar su posición en el nivel. Un sistema para encontrar colisiones es necesario. Al hacer y recibir daño se tienen que reproducir sonidos, al igual que al saltar. Deben haber representaciones gráficas de los puntos de vida del jugador y su puntaje, visibles durante el loop principal del juego.

2. Solución

La modelación del juego fue realizada a través de las siguientes clases:

1. `Platform`: Esta clase modela una plataforma rectangular en el nivel. Tiene métodos para encontrar su posición relativa con respecto a cierta `Entidad`.

2. `Level`: Esta clase modela un nivel con un conjunto de `Plataformas`, dándoles un mismo color a todas. Incluye un método `draw()` para dibujar en la vista todas las plataformas.

3. `Entity`: Ambos el jugador y los enemigos heredan de esta clase. Contiene métodos para moverse hacia la izquierda y la derecha y saltar. También actualiza la coordenada vertical con respecto al conjunto de platformas del nivel, con tal de evitar que la entidad entre a una plataforma. Para las coordenadas horizontales, esto se maneja en las métodos de movimiento lateral. Además se incluyen métodos auxiliares booleanos y de estado, para simplificar los condicionales en los métodos de esta clase. Finalmente, incluye métodos para dibujar a la entidad en la vista.

    i) `Player`: Esta clase contiene métodos para atacar, booleanos auxiliares y contadores para manejar momentos de invincibilidad luego de ser golpeado. Además se almacena una instancia de la clase `Sword` para manejar los ataques. También hay métodos para actualizar el estado de la entidad.

    ii) `Enemy`: Esta clase contiene métodos que le entregan una IA básica para ser capaz de perseguir al jugador y alcanzarlo en cualquier posición del nivel (sólo para niveles simples). También hay métodos auxiliares booleanos y para actualizar el estado de la entidad.

4. `Sword`: Aquí se almacenan 4 figuras para la espada, una en cada dirección ortogonal. Se manejan aqui la duración de los ataques y en que dirección deben ser dibujados. Esto se hace con la ayuda de una clase auxiliar `Attack`

    i) `Attack`: Esta clase se encarga de contar las frames que dura un ataque, y las frames donde no es posible atacar luego de terminar un ataque.

5. `PowerUp`: Esta clase encuentra posiciones válidas dentro del nivel para generar un power-up, y tiene un atributo que caracteriza su efecto.

6. `Sound`: Es una clase auxiliar, que carga los sonidos del juego al abrir la aplicación, y tiene métodos para reproducir sonidos específicos.

Además se usa la clase `CenteredFigure`, provista por Pablo Pizarro R., en muchas de estas clases. Finalmente en el archivo `constants.py` se almacenan todos los valores constantes usados en el juego, para poder alterar sus valores globalmente de manera directa.

Todas estas clases son usadas en conjunto en el archivo `square_knight.py` para permitir la ejecución del juego. La aplicación tiene 3 estados posibles:

- `main_menu`: En este estado se dibuja una Pantalla introductoria, con una instrucción para poder comenzar el juego al presionar la barra espaciadora.
- `playing`: Este estado es el loop principal del juego, donde se ejecuta la mayor parte de la lógica y se simulan los comportamientos del modelo junto al controlador, para mostrarlo en la vista.
- `game_over`: El juego cambia a este estado cuando la vida del jugador llega a 0, y muestra el high-score del jugador junto con una instrucción para jugar de nuevo.

En cualquier momento el jugador puede decidir cerrar la aplicación al presionar `escape` o el botón del sistema para cerrar la ventana. El comportamiento de la aplicación puede ser representado mediante el siguiente diagrama de estado:

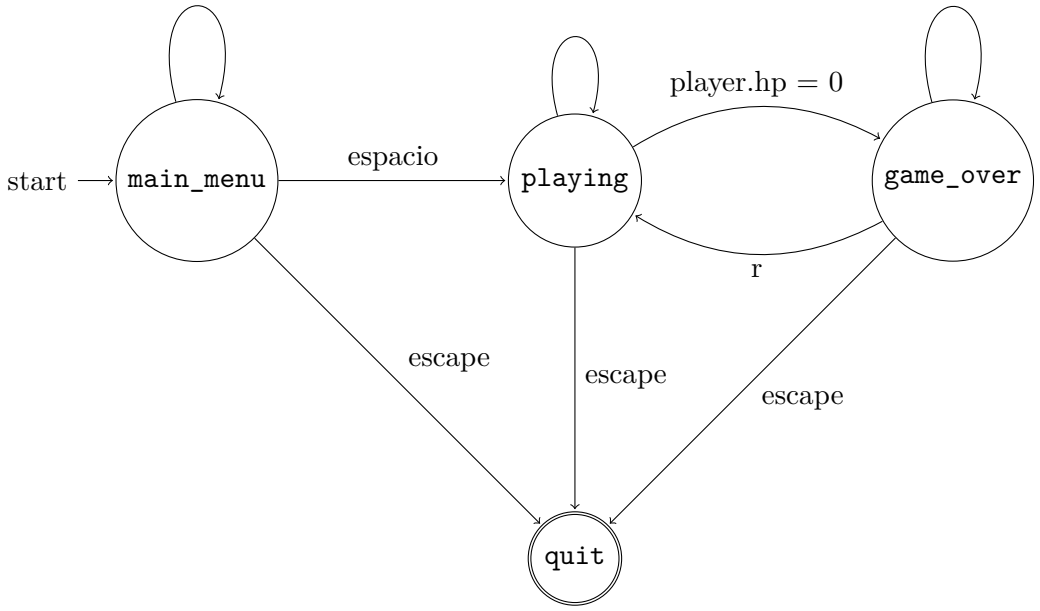


Figura 2.1: Diagrama de Estado de la Aplicación

3. Dificultades

Las mayores dificultades en la implementación de las clases fueron los métodos encargados de las colisiones, pues requerían lógica bastante precisa, la cual debía ser ordenada de tal manera que no se cumplieran dos condiciones simultáneamente. Pese a esto, la implementación de las colisiones con el nivel fueron bastante simplificadas gracias a la implementación de la clase `Plataforma`. Además esto

lo hace más flexible, pues al no ser programada a la fuerza para ciertas posiciones de las plataformas, la colisión también funciona para plataformas en distintas posiciones[1].

La colisión entre entidades y power-ups no fue mayor dificultad, gracias al método `intersects()` incluido en CenteredFigure, el cual se basa en el método provisto por pygame para intersectar polígonos.

Otra de las dificultades notables fue la implementación de la IA de los enemigos. Esto es gracias a que hay muchos casos donde el enemigo tiene que llegar al jugador, pero establecer un camino previamente es difícil. Este problema se solucionó con la ayuda de los método auxiliares incluidos en la clase `Enemy`, los que facilitaron diferenciar los distintos casos para los cuales eran necesarios distintos comportamientos del enemigo.

Ambos este caso y el de la colisión con el nivel fueron complejos pues existían muchos casos de borde, para los cuales había que refinar los condicionales para que funcionaran debidamente.

Por último, el conteo de la duración de los ataques fue un poco compleja al comienzo, por lo que se utilizó la clase auxiliar `Attack`, cuyo único trabajo era contar estos fotogramas y retornar valores booleanos para saber en que parte del ataque se encontraba el jugador.

En general, no se encontraron dificultades peores a estas durante la implementación de la tarea. Esto es principalmente gracias a la utilización de la programación orientada a objetos, pues permitió modularizar el problema en segmentos simples y fácilmente modificables. Esto además permite extender este proyecto en el futuro con mayor facilidad.

4. Conclusiones

La programación orientada a objetos ayudó inmensamente a la implementación de los modelos, lo que permitió hacer un juego fácilmente expandible en el futuro. El resultado final es un juego con todas las funciones esperadas, aparece la vida y el puntaje en pantalla, el jugador se puede mover libremente y atacar en la dirección que se desee. Los enemigos siguen al jugador por cualquier lugar del nivel y el jugador recibe daño al ser tocado por un enemigo. Los enemigos reciben daño[2] al ser golpeados por la espada del jugador. Cuando una entidad es golpeada, esta recibe frames de invincibilidad, donde no puede recibir daño. Existen 3 tipos de power-ups:

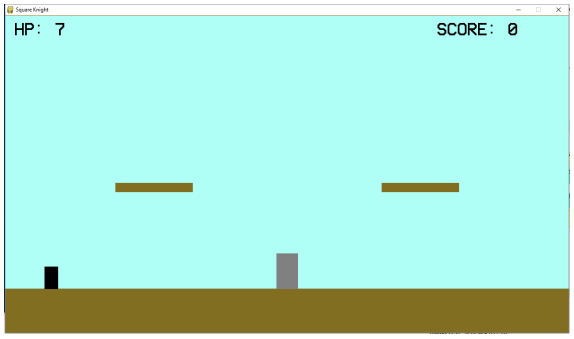
1. Heal (Verde): Aumenta la vida del jugador en 1. No tiene máximo. Probabilidad 50 %.
2. Attack Up (Rojo): Aumenta el ataque del jugador en 0.5 puntos. Tiene máximo de 4. Probabilidad 30 %.
3. Range Up (Azul): Aumenta el tamaño de la espada del jugador con un factor de escala igual a 1.1. Se aplica multiplicativamente. No tiene máximo, pero al aumentar el tamaño, comienza a dejar un punto ciego cerca del jugador. Probabilidad 20 %.

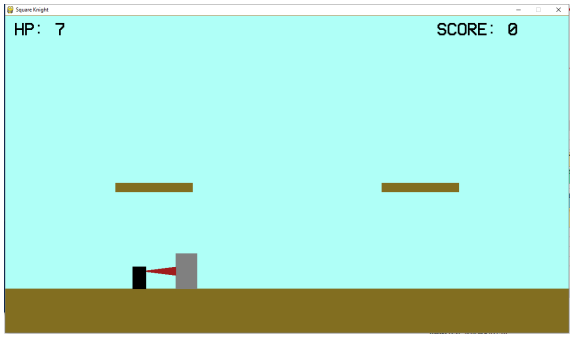
[1] No sirve para todas las distintas configuraciones, pues depende de el orden en el que se analizan las plataformas. Esto puede ser arreglado implementando un método en la clase Level que retorne la plataforma más cercana a la entidad.

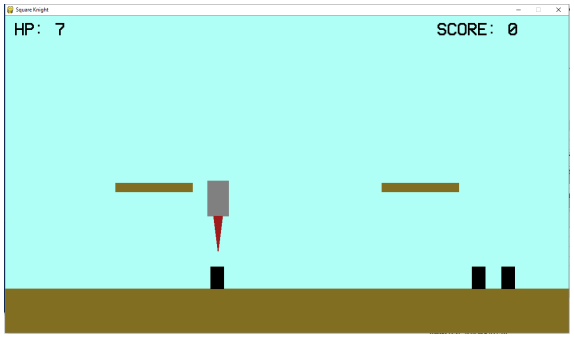
[2] Su movimiento también es alterado.

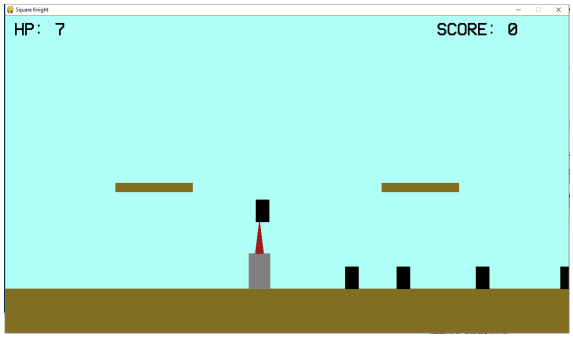
El modelo MVC utilizado para ordenar la estructura del juego también fue bastante útil, pues proporcionaba un esquema claro de cómo se generan los fotogramas y cómo se debe interpretar el input del usuario.

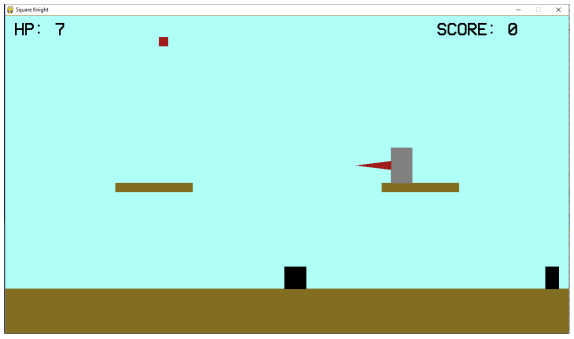
A continuación se muestran capturas de pantalla del juego en ejecución.











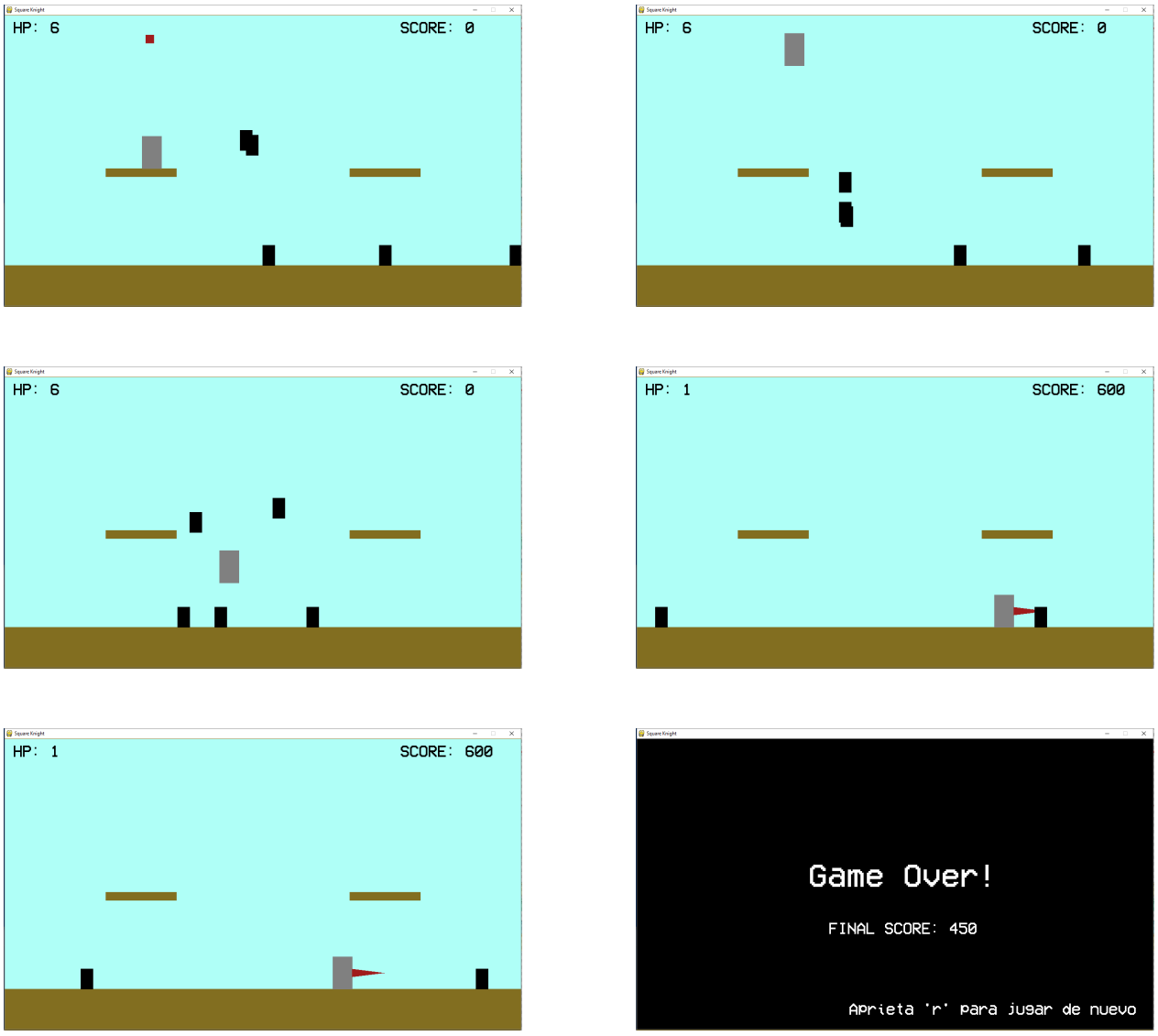


Figura 4.1: Capturas del juego en acción.

Para probar el juego, por favor leer `README.md`.