UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Jugador autónomo para el videojuego Slay the Spire

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: López Tadeo, Javier
Tutor: Abad Cerdá, Francisco José

Curso 2018-2019

Resum

Slay the Spire és un videojoc roguelike, RPG de cartes i masmorres desenvolupat per MegaCrit. El funcionament bàsic del joc consisteix en combats entre 2 personatges, un d'ells controlat pel jugador i l'altre pel mateix joc. El guanyador serà el que redueixi els punts de vida de l'enemic a 0. Per a això farà ús de cartes amb diferents efectes (mal, defensa, aplicar vulnerabilitats ...).

L'objectiu del projecte és construir un jugador autònom que reconegui els diferents elements del joc (cartes en mà, punts de vida, intenció de l'enemic ...), u obtingui una estratègia a partir d'aquestes dades i jugui entorn a aquesta estratègia.

El desenvolupament del treball s'ha dut a terme mitjançant Python. Per a aquest projecte s'ha desenvolupat un reconeixement d'imatge per als diferents components del videojoc a partir de captures directes d'aquest, una estratègia en base als elements reconeguts, i, finalment l'automatització del joc. L'automatització ha consistit en la injecció d'esdeveniments de ratolí, que és el dispositiu d'entrada amb el qual es controla el joc.

Paraules clau: Slay the Spire, videojoc, automatització, reconeiximent, autónom, cartes, Python

Resumen

Slay the Spire es un videojuego roguelike, RPG de cartas y mazmorras desarrollado por MegaCrit. El funcionamiento básico del juego consiste en combates entre dos personajes, uno de ellos controlado por el jugador y el otro por el propio juego. El ganador será el que reduzca los puntos de vida del enemigo a cero. Para ello hará uso de cartas con diferentes efectos (daño, defensa, aplicar vulnerabilidades...).

El objetivo del proyecto es construir un jugador autónomo que reconozca los diferentes elementos del juego (cartas en mano, puntos de vida, intención del enemigo...), obtenga una estrategia a partir de dichos datos y juegue entorno a esa estrategia.

El desarrollo del trabajo se ha llevado a cabo en Python. Para este proyecto se ha desarrollado un reconocimiento de imagen para los diferentes componentes del videojuego a partir de capturas directas de este, una estrategia en base a los elementos reconocidos, y, por último la automatización de juego. La automatización ha consistido en la inyección de eventos de ratón, que es el dispositivo de entrada con el que se controla el juego.

Palabras clave: Slay the Spire, videojuego, automatización, reconocimiento, autónomo, cartas, Python

Abstract

Slay the Spire is a roguelike videogame, RPG of cards and dungeons developed by MegaCrit. The basic mechanics of the game consists of combats between 2 characters, one of them controlled by the player and the other by the game itself. The winner will be the one that reduces the enemy's life points to 0. For this, he will use cards with different effects (damage, defense, apply vulnerabilities ...).

The objective of the project is to build an autonomous player that recognizes the different elements of the game (cards in hand, points of life, enemy intention ...), obtain a strategy from said data and play around that strategy.

The development of the work has been carried out with Python. For this project, an image recognition has been developed for the different videogame components from screenshots of it, a strategy based on the recognized elements, and, finally, game automation. Automation has consisted of the injection of mouse events, which is the input device with which the game is controlled.

Key words: Slay the Spire, videogame, automation, recognition, autonomous, cards, Python

Índice general

Índice general VII
Índice de figuras IX

1 Introducción 1
1.1 Motivación . . . 3
1.2 Objetivos . . . 3
1.3 Estructura de la memoria . . . 3
2 Problema 5
2.1 Análisis del problema . . . 5
2.2 Estado del arte . . . 5
2.3 Posibles soluciones . . . 7
2.3.1 *Template matching* . . . 7
2.3.2 Extracción de puntos . . . 10
2.3.3 Filtrado por color . . . 10
2.4 Solución propuesta . . . 11
3 Diseño de la solución 13
3.1 Arquitectura . . . 13
3.2 Diseño detallado . . . 14
4 Tecnologías 17
4.1 Python . . . 17
4.2 Processing . . . 17
4.3 GitHub . . . 18
4.4 PyCharm . . . 18
4.5 Tesseract . . . 18
4.6 OpenCV . . . 19
4.7 *Beautiful Soup* . . . 19
4.8 *Pickle* . . . 19
5 Desarrollo 21
5.1 *Web scraping* . . . 21
5.2 Obtención cartas . . . 22
5.3 Extracción de puntos . . . 22
5.4 Reconocimiento de cartas . . . 24
5.5 Reconocimiento de enemigos . . . 26
5.6 Jugador autónomo . . . 28
6 Conclusiones 31
7 Trabajos futuros 33
Bibliografía 35

Índice de figuras

1.1 HUD . . . 2

2.1 *Template* . . . 8
2.2 Imagen original . . . 8
2.3 Deslizamiento . . . 8
2.4 Ejemplo *template matching* con suma de diferencias al cuadrado . . . 8
2.5 Ejemplo *template matching* con correlación cruzada . . . 8
2.6 Ejemplo de falsos positivos . . . 9
2.7 Ejemplo plantilla . . . 9
2.8 Ejemplo plantilla . . . 9
2.9 Imagen original . . . 9

3.1 Arquitectura . . . 13
3.2 Información de una carta . . . 14
3.3 Energia o maná . . . 14
3.4 Jugador . . . 14
3.5 Enemigo . . . 14
3.6 Esquema clases . . . 15

5.1 Ejemplo mano del jugador . . . 22
5.2 Siluetas de la carta mediante Processing . . . 23
5.3 Puntos característicos de la carta . . . 23
5.4 Ejemplo normalización . . . 25
5.5 Región de interés . . . 26
5.6 Ejemplo barras de vida . . . 27
5.7 Ejemplo barras de vida filtradas . . . 27
5.8 Texto limpio . . . 27
5.9 Texto filtrado . . . 27
5.10 Plantilla . . . 27
5.11 Máscara . . . 27
5.12 Resultado *match template* . . . 28

CAPÍTULO 1
Introducción

En las últimos años se ha podido observar un crecimiento progresivo en el ámbito del reconocimiento de imágenes. Uno de los sectores en los que se puede poner en práctica las técnicas de reconocimiento de imagen, es el de los videojuegos. Concretamente en los juegos de cartas como Magic the Gathering o Hearthstone. Esto se puede observar en plataformas de streaming como Twitch, que nos ofrecen un reconocimiento de los elementos del juego en tiempo real.

Otro de los campos que ha experimentado un aumento es el de la automatización de tareas. Este sector tiene un único propósito: permitir que las tareas monótonas y repetitivas sean llevadas a cabo por la máquina, dejando más tiempo al humano para que se centre en tareas más complejas.

En este trabajo se presenta una combinación de ambos sectores. Se ha implementado un jugador autónomo para el videojuego Slay the Spire. Para ello se ha desarrollado un sistema que reconoce las diferentes cartas del videojuego en tiempo real y utiliza la información obtenida para poner en marcha un jugador autónomo que sigue un tipo de estrategia para avanzar entre los diferentes niveles de forma automática.

A continuación se explica cada elemento de la interfaz del juego y una breve explicación de su jugabilidad. En la figura 1.1 podemos ver una enumeración de los distintos componentes del juego.

Slay the Spire es un juego *roguelike* y *deckbuilding*. El género *roguelike* se basa en mazmorras creadas aleatoriamente manteniendo un patrón base, que suele incluir muerte permanente y una premisa de juego sencilla y con poca narrativa. El único objetivo del personaje es avanzar por la mazmorra hasta llegar al final o morir. Aunque la muerte es permanente, durante la partida se pueden desbloquear objetos o habilidades que se mantendrán para el resto de partidas. Por otro lado, en los juegos de género *deckbuilding* se empieza con una baraja base que se va mejorando nivel tras nivel mediante las recompensas obtenidas.

En Slay the Spire se puede escoger al inicio de la mazmorra entre tres personajes, cada uno con unas mecánicas diferentes. El objetivo principal del juego es derrotar a los enemigos que encontramos en las distintos pisos de la mazmorra e intentar que no reduzcan los puntos de vida de nuestro personaje a cero. Para conseguir este objetivo haremos uso de las cartas que, o bien nos darán puntos de bloqueo para defendernos, o infligirán puntos de daño a los enemigos. Cada personaje empieza con un mazo inicial que se compone de cartas básicas y conforme se vayan superando los distintos niveles se podrán añadir nuevas cartas al mazo. Las cartas pueden ser de 3 tipos: habilidad, poder o ataque. Las cartas de habilidad en general suelen ser cartas de supervivencia, ya que son las que nos dan puntos de bloqueo o mejoras de defensa. Las cartas de ataque, como su nombre indica, son cartas que infligen daño a los enemigos o nos proporcionan potenciadores de

ataque. Y por último, las cartas de poder son las que nos ofrecen una variedad de ventajas, por ejemplo robar más cartas. Además, tambien podemos conseguir reliquias que nos conceden bonificaciones o maleficios durante toda la partida.

Slay the Spire es un juego por turnos, donde cada turno robaremos una cierta cantidad de cartas y obtendremos una cierta cantidad de puntos de maná. Los puntos de maná serán necesarios para jugar las cartas, ya que cada una tiene un coste dependiendo de su funcionalidad. Una vez consumido el maná o quedarnos sin cartas en la mano finaliza el turno, en este momento el enemigo realiza su intención que pueden ser un ataque, potenciador o debilidad. Después de que el enemigo termine de ejecutar su turno el jugador retoma la iniciativa y roba cierta cantidad de cartas para continuar el combate.

Figura 1.1: HUD

1. Inventario de reliquias: En este área se agrupan las reliquias (objetos que proporcionan distintos efectos durante la partida) que se van obteniendo durante el transcurso de la partida.
2. Puntos de vida: Marca la vida actual y máxima del jugador.
3. Oro: Aquí se muestra la cantidad de oro que tiene el jugador que podrá gastar en las tiendas posteriormente.
4. Pociones: Objetos de un único uso que sólo tienen efecto durante un combate.
5. Piso: Señala el nivel de la mazmorra en la que se ubica el jugador.
6. Mapa: Contiene la información de cada piso.
7. Baraja: En esta sección podremos encontrar las cartas con las que el jugador se enfrentará a los enemigos.
8. Ajustes: Permite modificar la configuración del juego.
9. Avatar: Personaje que representa al jugador.
10. Enemigo: Personaje al que se enfrenta el jugador durante la partida.

11. Intención del enemigo: Efecto que causará el enemigo al final del turno.
12. Armadura: Muestra los puntos de bloqueo extra que tiene el enemigo.
13. Estados: Contiene los estados del enemigo como podrían ser los potenciadores o debilitaciones.
14. Baraja activa: En esta sección se encuentran las cartas que quedan en la baraja a lo largo del combate.
15. Indicador de maná: Indica la cantidad de puntos de maná para gastar en las cartas.
16. Mano del jugador: Aquí residen las cartas que ha robado el jugador y que puede usar en el turno actual.
17. Pasar turno: Botón para pasar el turno.
18. Pila de descarte: Conjunto de cartas usadas por el jugador.

1.1 Motivación

La motivación personal de este trabajo es principalmente la pasión que tengo por los videojuegos y los juegos de mesa. Particularmente, con los juegos de estrategia y de cartas. Por otro lado, siempre he creído que el tiempo es oro y el hecho de poder ahorrar tiempo quitándose tareas sencillas y repetitivas son lo que me ha llevado a desarrollar la parte de automatización en este proyecto. Además, cierto es que siempre me ha fascinado los avances que han hecho grandes empresas como DeepMind respecto a IA jugando a videojuegos. La intención de este proyecto es aproximarme a este ámbito y aprender más.

1.2 Objetivos

El objetivo global de este proyecto es el desarrollo de un *software* que implemente un jugador autónomo del videojuego Slay the Spire. Dicho software tiene como entrada la propia ventana del juego, sobre la que reconoce tanto las cartas de la mano del jugador como de los elementos asociados a la partida. Finalmente ofrece una salida de datos que se manifiesta en movimientos del ratón.

Los objetivos específicos de este trabajo son:

1. Definir y construir la base de datos para el reconocimiento de las cartas.
2. Desarrollar un sistema para reconocer los elementos del juego.
3. Implementar un jugador autónomo que realice las acciones que podría hacer el jugador.
4. Diseñar una estrategia en base a la información obtenido por el reconocedor.

1.3 Estructura de la memoria

Esta memoria tiene una estructura tradicional dividida en siete capítulos, además de contar con una bibliografía al final del trabajo, donde se encuentran las referencias que han ayudado a la elaboración de este trabajo.

En el capítulo 2 se ha realizado un análisis del problema, contextualizándolo en la época actual y finalmente explicar algunas de las posibles soluciones.

A continuación, en el capítulo 3, se ha planteado el diseño de la solución, mostrando la arquitectura que sigue, indicando los grandes bloques en los que se divide la solución. Se identifican los diferentes componentes y cómo se relacionan entre ellos mediante diagramas. Además se ha detallado el diseño de la solución, presentando las clases principales que implementan la solución.

El capítulo 4 trata sobre las herramientas y tecnologías que se han usado en este proyecto, explicando su funcionamiento y origen.

Como cuerpo del trabajo, el capítulo 5 detalla la solución que se ha propuesto en los capítulos anteriores, exponiendo cómo hemos abordado los diferentes problemas y expresando de forma mas técnica la solución llevada a cabo.

Y por último en los capítulos 6 y 7 se presentan las conclusiones del resultado obtenido y los trabajos futuros que podrían ampliar este proyecto.

CAPÍTULO 2
Problema

2.1 Análisis del problema

El problema a la hora de diseñar un jugador autónomo es decidir que tareas va a ejecutar y en base a qué parámetros hará dicha tarea. Para ello, la primera cuestión que se nos plantea es identificar los diferentes elementos del entorno. El siguiente problema es procesar los datos obtenidos. Y por último, diseñar una salida que es la tarea que ejecuta el jugador.

En el caso de este proyecto la mayor dificultad que se nos presenta es reconocer los componentes del videojuego, como pueden ser las cartas de la mano del jugador o el estado de los enemigos. Además este trabajo debe ofrecer la misma funcionalidad en las diferentes resoluciones en las que se puede ejecutar el videojuego.

2.2 Estado del arte

Los avances logrados en el campo del *machine learning* y el uso de servicios de datos de alto ancho de banda están consiguiendo un gran crecimiento en la tecnología basada en reconocimiento de imágenes. Empresas de diferentes sectores como el comercio electrónico, la automoción y el de los videojuegos están adoptando muy rápidamente la tecnología del reconococimiento de imágenes a sus proyectos. Actualmente, este tipo de tecnologías está dividido en hardware, software y servicios. El campo del hardware esta dominado por los móviles de última generación, los cuales tienen un importante papel en el crecimiento del reconocimiento de imágenes. Esto se debe a que existe una incremento en la necesidad de aplicaciones y servicios de seguridad con nuevas tecnologías basadas en vigilancia y reconocimiento facial.

El reconocimiento de imágenes se refiere a las tecnologías que identifican lugares, personas, logos, edificios o cualquier tipo de objeto en una imagen. Los usuarios a través de aplicaciones, redes sociales y páginas web, comparten una enorme cantidad de datos. Las empresas usan esta gran cantidad de datos para ofrecer mejores servicios a las personas que los usan.

El reconocimiento de imágenes es una parte de la visión por computador. Exactamente es el proceso de identificar y detectar un objeto en un vídeo o una imagen. Pero el término de visión por computador es un término más amplio que incluye métodos para recolectar información, procesamiento y análisis de datos. Sin embargo, ¿cómo funciona realmente el reconocimiento de imágenes?

Por ejemplo, Facebook actualmente es capaz de realizar un reconocimiento de cara con una precisión del 98 % [20]. Dicha precisión es comparable con la habilidad de reconocer caras de los humanos. Esta gran precisión se debe a la eficacia que tiene su reconocedor a la hora de clasificar imágenes. La clasificación de imágenes se basa en la búsqueda y comparación de patrones en los datos de entrada. De hecho, el reconocimiento de imágenes es, básicamente, elegir una categoría entre muchas. El paso más difícil a la hora de reconocer imágenes es recolectar y organizar los datos, construir un modelo de clasificación y usarlo para reconocer las imágenes.

A día de hoy se sigue intentando mejorar las técnicas de clasificación y reconocimiento de imágenes. Un ejemplo de ello es ImageNet y su *challenge* [21]. ImageNet es una gran base de datos diseñada para software de reconocimientos de objetos. Este *dataset* cuenta con más de 14 millones de imágenes y 20000 categorías. Su desafío empezó a partir de 2010 cuando la principal investigadora de ImageNet propuso que los equipos de investigación evaluaran sus algoritmos sobre un conjunto de datos y compitiesen por conseguir la mayor precisión. Los primeros años del desafío apenas se conseguían precisiones alrededor del 15 % llegando, en la actualidad, la mayoría de los equipos a estar por encima del 95 %.

Por otro parte, el sector de los videojuegos ha a empezado a verse afectado por los avances en inteligencia artificial. Primero, fueron juegos de mesa como el ajedrez [13] y el go [14], donde una máquina ganó a campeones de ambos sectores. Pero poco a poco se fue buscando apoyo en los videojuegos, empezando por videojuegos simples y terminando por ganar en juegos complejos como Dota 2 o StarCraft II [12].

Los videojuegos han sido, desde hace décadas, campos de prueba para testear los avances en los sistemas de inteligencia artificial. Del mismo modo que avanzaban las capacidades de estos sistemas, se han tenido que buscar juegos más complejos que pudiesen suponer un reto a los desarrolladores de IA. En los últimos años, StarCraft ha sido considerado como uno de los juegos de estrategia en tiempo real más desafiantes y complicados. Por ello se ha ganado el puesto principal para las investigaciones de IA.

Sin embargo, a principios de 2019, DeepMind creó AlphaStar, la primera inteligencia artificial capaz de derrotar a jugadores profesionales de StarCraft II. Aunque ya habían conseguido éxitos en otros videojuegos como Mario, Quake III y Dota 2, ninguna de estas técnicas se ajustaba para el problema que planteaba StarCraft. Incluso simplificando los mapas o con ciertas restricciones no eran capaces de sobrepasar las habilidades de los jugadores profesionales. No obstante, AlphaStar lo consigue sin ningún tipo de restricción o norma especial, usando redes neuronales profundas y entrenando con datos del juego mediante aprendizaje supervisado y por refuerzo.

StarCraft es un videojuego creado por Blizzard Entertainment en 2010 [15], se basa en un universo de ciencia ficción con una jugabilidad multicapa diseñada para desafiar al jugador. StarCraft lleva siendo desde hace 20 años uno de los juegos en el top de los deportes electrónicos.

A pesar de haber diferentes modalidades de juego, la más común es el 1 vs 1. Lo primero que debe hacer el jugador es elegir una raza entre tres posibilidades, cada una de ellas con su propio estilo de juego. Los jugadores empiezan la partida con unas unidades recolectoras con las cuales tendrán que recoger recursos para crear nuevos edificios y nuevas unidades. De manera progresiva se van desarrollando nuevas tecnologías que permitirá al jugador crear mejores estructuras y unidades.

En StarCraft se debe balancear los propósitos a largo y corto plazo y a su vez adaptarse a situaciones inesperadas. Dominar este problema requiere adentrarse en varios retos de la IA como por ejemplo:

- Información parcial: A diferencia de otros juegos como podría ser el ajedrez, en StarCraft el jugador no posee toda la información de la partida, ya que el jugador debe ir explorando el mapa y despejando lo que se conoce como, "niebla de guerra".

- Tiempo real: En este juego el jugador debe hacer acciones constantemente, a diferencia de juego de mesa como el ajedrez que están basados en turnos.

- Teoría de juegos: StarCraft es como el piedra-papel-tijera. No hay una estrategia dominante que gane al resto de tácticas.

- Planificación a largo plazo: Como algunos problemas de la vida real, nuestras acciones no tienen una consecuencia instantánea. Si no que verán su efecto al cabo de un tiempo.

En este proyecto se va a desarrollar un jugador autónomo para el juego Slay the Spire que, a diferencia de StarCraft II, es un juego por turnos. La elección de este tipo de juego es debido a que el desarrollo de un jugador autónomo es más sencillo que un videojuego con acciones en tiempo real.

El comportamiento de AlphaStar está generado por una red neuronal profunda que recibe los datos de entrada a través de la interfaz del juego, y proporciona una salida de datos mediante instrucciones que realizan acciones en el juego. Al igual que este proyecto, el trabajo desarrollado en esta memoria, obtiene la información de la interfaz del juego. Sin embargo, la salida de AlphaStar se manifiesta con instrucciones que realizan acciones directas del juego mientras que nuestro jugador autónomo lo hace con acciones del ratón.

2.3 Posibles soluciones

Para el caso de un videojuego de cartas donde hay un camara estática que ofrece una imagen sin ruido y en la que la jugabilidad se basa por turnos, no es necesario utilizar técnicas como las redes neuronales o el *Deep Learning* para el reconocimiento. Además de que el juego posee un conjunto de imágenes limitado al número de cartas que hay en este. Para ello utilizaremos otro tipo de técnicas más sencillas y que se adaptan mejor al problema.

Como se ha comentado previamente, a la hora de hacer el reconocimiento, tenemos varios elementos que identificar. Entre estos se encuentran las cartas, las barras de vida y las intenciones del enemigo. A continuación plantearemos algunas de las técnicas que se pueden usar para los diferentes elementos.

2.3.1. *Template matching*

La primera solución que se evaluó durante el desarrollo de este trabajo es el *template matching*. Esta técnica de procesado de imágenes consiste en buscar una imagen pequeña (figura 2.1) que es la plantilla, en una imagen más grande (figura 2.2) . Esto se consigue deslizando el *template* por la imagen original para cada uno de los píxeles cómo se puede observar en la figura 2.9. Mientras se desplaza la plantilla se compara la plantilla con la porción de la imagen original en la que se encuentra.

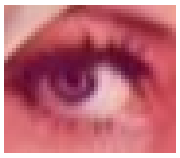


Figura 2.1: *Template*

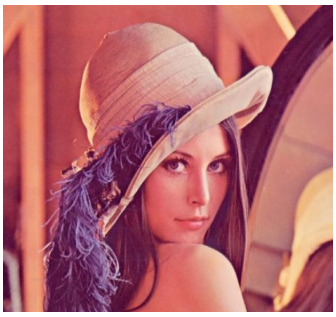


Figura 2.2: Imagen original

Figura 2.3: Deslizamiento

El *match* se hace calculando un número que denota cuánto se diferencian ambas imágenes. Existen varios métodos para calcular este número. Por ejemplo, en la figura 2.4 se ha usado la suma de diferencias al cuadrado que realiza una distancia euclídea al cuadrado por cada uno de los píxeles de la plantilla. Por esta razón, se observa en la región del ojo de la imagen resultado un tono mas oscuro que indica que hay menos diferencia entre ambas imágenes.

Figura 2.4: Ejemplo *template matching* con suma de diferencias al cuadrado

Otro método que se puede usar es la correlación cruzada. En este caso cada píxel de la plantilla se multiplica por los de la imagen original y se hace la suma de los resultados. Por este motivo, en la figura 2.5 se observa que en la zona del ojo hay un tono más brillante denotando que la suma es mayor y por ello que las imágenes se parecen más en ese píxel.

Figura 2.5: Ejemplo *template matching* con correlación cruzada

Cartas

En el caso de las cartas esta técnica es muy útil ya que cada carta es única, además de que la propia imagen de la carta nos sirve de *template*. Sin embargo, existen dos problemas con el uso de esta técnica que residen en el propio juego. Como se muestra en la figura 5.1 cuando las cartas se van acumulando en la mano del jugador, estas se solapan unas con otras, y se va curvando su distribución. El problema que tiene el solapamiento es que el *template matching* no encontraría la plantilla, ya que esta técnica busca el cien por cien de la imagen. Por otro lado debido a la curvatura en la distribución de las cartas, este método no haría *matches* ya que no es invariante a la rotación.

Barra de vida

Para el caso de las barras de vida el uso de esta técnica es inviable. Esto se debe a la simplicidad de las barras de vida que generarían una gran cantidad de falsos positivos. Podemos ver un ejemplo de este caso en la figura 2.6

Figura 2.6: Ejemplo de falsos positivos

Intenciones del enemigo

Al igual que en el caso de la cartas, las intenciones del enemigo vienen dadas por una imagen característica y única. A diferencia de las cartas, las intenciones no sufren ningún tipo de rotación en el transcurso de la partida. De modo que este método sería ideal para la detección de este elemento

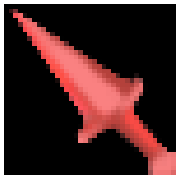


Figura 2.7: Ejemplo plantilla

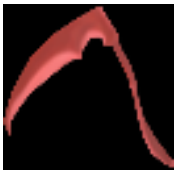


Figura 2.8: Ejemplo plantilla

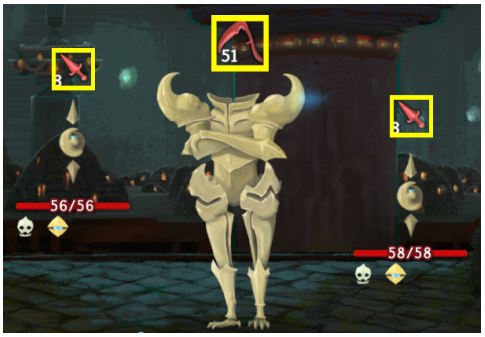


Figura 2.9: Imagen original

2.3.2. Extracción de puntos

Otra de las posibles soluciones que podríamos usar para el reconocimiento es la extracción de puntos. Este método consiste en obtener puntos de la imagen que sean característicos. La ventaja de esta técnica es que es invariante a la rotación.

Cartas

Con este procedimiento queda solventado el problema de la rotación de las cartas. Además, del arte de la carta se puede obtener mucha información para los puntos que ayudarán posteriormente a identificar las cartas. Con esto queda solucionado también el problema del solapamiento ya que podemos obtener bastantes puntos sin la totalidad de la imagen.

Barra de vida

Al igual que el *template matching*, la extracción de puntos no es útil para el reconocimiento de las barras de vida. Debido otra vez a la simplicidad de estas, donde no se pueden encontrar ningún punto característico dentro de ellas.

Intenciones del enemigo

En el caso de las intenciones del enemigo la extracción de punto es viable, ya que cada una de la intenciones es bastante característica y se podría obtener información suficiente para diferenciarlos.

2.3.3. Filtrado por color

Otra forma de realizar el reconocimiento es utilizar el filtrado por color. Con esta técnica lo que se quiere conseguir es aislar el elemento deseado de otros elementos como podría ser el fondo.

Cartas

En el caso de las cartas con este método no habría ningún problema con la rotación o el solapamiento de las cartas. Sin embargo, no resulta útil dado que las imágenes de las cartas presentan una gran variedad de colores, por lo que es difícil filtrarlos por estos.

Barra de vida

Para el caso de las barras de vida esta técnica es bastante apropiada, ya que al final las barras de vida son simplemente una línea de un único color que se mantiene estática durante el transcurso de un combate. Por tanto es muy fácil aislarlas del resto de elementos.

Intenciones del enemigo

A pesar de que las intenciones del enemigo no posean una complejidad de colores como las cartas, es verdad que tampoco poseen una simplicidad de colores como las barras de vida haciendo su filtrado también un poco complejo.

2.4 Solución propuesta

Ya que cada uno de los elementos a reconocer se reconoce mejor con una técnica u otra, finalmente se ha optado por una combinación de las posibles soluciones,

Cartas

Para el caso de las cartas, se ha optado la extracción de puntos, ya que este método nos soluciona el problema de la rotación y del solapamiento de las cartas.

Barra de vida

En el caso de las barras de vida se ha elegido el filtrado de color. Esto se debe a la simplicidad que tienen, haciendo su filtrado muy sencillo.

Intenciones del enemigo

Por último, en el caso de las intenciones del enemigo, se ha seleccionado el *template matching*, puesto que las intenciones del enemigo encajan perfectamente en el prototipo de plantilla para hacer los *matches*.

CAPÍTULO 3
Diseño de la solución

3.1 Arquitectura

A continuación, se plantea la arquitectura que ha seguido este proyecto, explicando como hemos estructurado el sistema, sus partes y sus funciones y como interactuan estas partes. Podemos ver el esquema de la arquitectura en la figura 3.1.

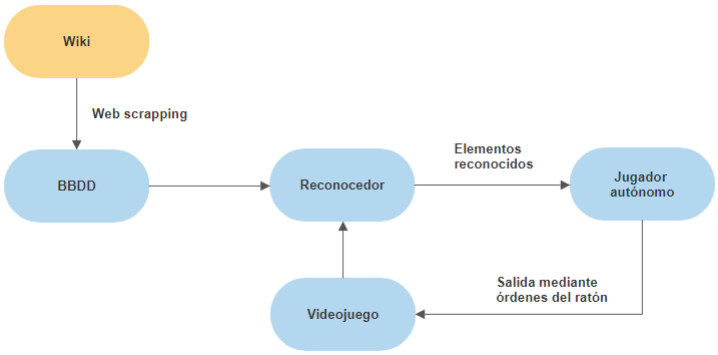


Figura 3.1: Arquitectura

Uno de los elementos más importantes del jugador autónomo implementado en este trabajo es el reconocedor de imágenes. Dicho reconocedor se basa en una base de datos en la que se almacena las cartas del juego. Esta base de datos se ha completado con la información obtenida de la wiki del juego [18] mediante *web scraping* usando la librería *beautiful soup* de Python. En esta wiki se encuentra tanto las imágenes de las cartas como la información asociada a estas (coste de maná, nombre...) El único problema de esta wiki ha sido la desactualización en las imágenes de la carta, por ello hemos desarrollado un bot que toma capturas directas de todas las cartas del videojuego.

Una vez obtenidas las imágenes de la cartas, el siguiente paso es reconocerlas. Para ello se ha utilizado el lenguaje y entorno de programación Processing. Con esta herramienta hemos realizado la extracción de los puntos característicos de las cartas, con los cuales se ha diseñado un reconocedor en Python basado en diferencias de color por región alrededor de los puntos. Además de las cartas también hemos de reconocer las intenciones y la vida del enemigo. Para el primer caso hemos hecho uso de la librería OpenCV y sus métodos de template matching. En el segundo caso, se ha localizado la posición de las barras de vida mediante una función de filtrado de color y a continuación se ha reconocido los números de estas con ayuda de la librería pytesseract.

Por último se ha diseñado un jugador autónomo también en Python que recibe la información obtenida del reconocedor. Con esta información el jugador da una salida

en base a la estrategia proporcionada, la cual se manifiesta con el movimiento del ratón gracias a la librería de pyautogui.

3.2 Diseño detallado

Para poblar las base de datos se ha utilizado la información de la wiki del juego. En la figura 3.2 podemos ver un ejemplo de una carta del juego. De esta información descartamos la rareza de la carta, ya que en nuestro proyecto no tiene ningún uso. A parte, como se ha comentado anteriormente la imagen de las cartas están desactualizadas por lo que también descartamos este dato. Las imágenes de las cartas las obtendremos directamente del juego mediante capturas.

Name	Picture	Rarity	Type	Energy	Description
Bash		Starter	Attack	2	Deal 8(10) damage. Apply 2(3) Vulnerable.

Figura 3.2: Información de una carta

A continuación el siguiente elemento a reconocer es la vida de los personajes. Este caso se compone tanto de la vida de los enemigos como del propio jugador. Como se puede ver en la figura 3.4 y en la 3.5 los elementos a reconocer son los mismos. Estos son la vida actual de cada uno, la vida máxima y la defensa/armadura. La única diferencia entre ambos, es que el jugador tiene un contador de maná como se puede ver en la figura 3.3, por lo tanto tiene un maná actual y un maná máximo. Para poder reconocer estos elementos necesitamos determinar su posición. En el caso del jugador es fácil ya que esta es fija durante toda la partida. Por otro lado en el caso del enemigo esta es variable, por ello como se ha comentado anteriormente se hace uso de una técnica por filtrado de color para determinarla.

Figura 3.3: Energia o maná

Figura 3.4: Jugador

Figura 3.5: Enemigo

Una vez definidos los elementos básicos **Carta**, **Jugador** y **Enemigo**, el siguiente paso es definir el resto de clases del proyecto. La primera clase que encontramos es la **Mano**, es decir, la mano del jugador. Esta clase es la que contiene en todo momento las cartas que posee el jugador, además de una lista de las cartas también contiene la posición de estas.

Seguidamente nos encontramos con la clase **Visión**, esta clase se trata del reconocedor. Esta clase contiene una variable con la imagen del juego que se va actualizando cada turno. Además es la que contiene todos los métodos necesarios para reconcocer los diferentes elementos del juego.

Por último, tenemos la clase **Entorno** que se trata del jugador y es la que contiene toda la información de la partida. A parte es la encargada de pedir la información de los elementos a la clase **Visión** y con esta prepara una salida y decide que cartas de la **Mano** jugar mediante los movimientos de ratón.

Podemos ver el esquema del diseño de las clases en la figura. 3.6

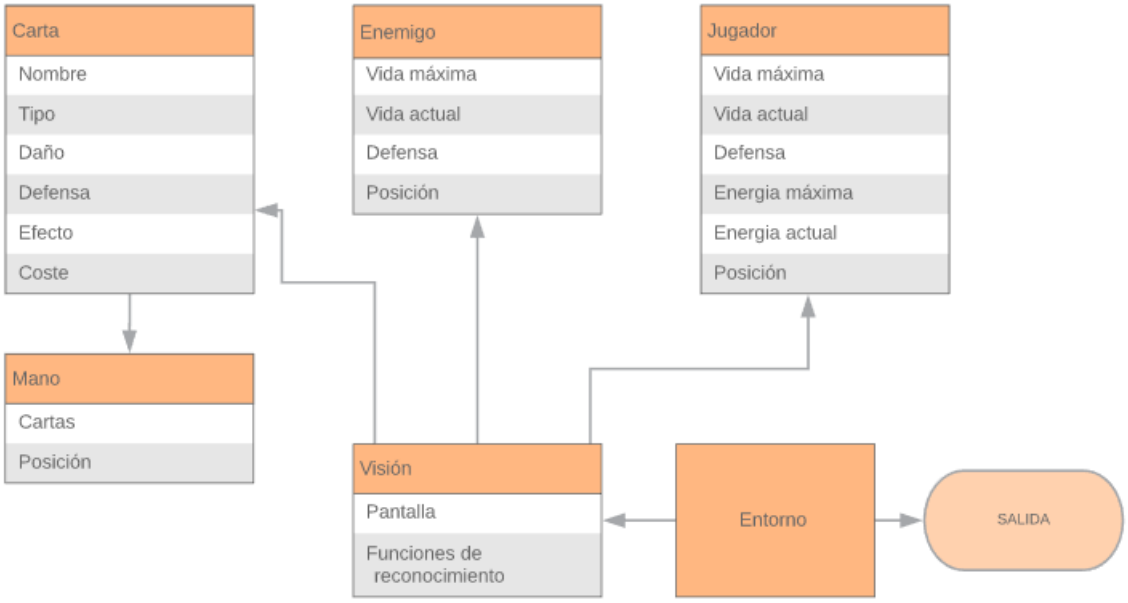


Figura 3.6: Esquema clases

CAPÍTULO 4
Tecnologías

Para el desarrollo de este proyecto se han utilizado Python y Processing, que serán detallados en pronfundidad seguidamente. También se ha usado GitHub para tener un control de versiones y almacenar el trabajo en la nube. Además, se han utilizado una gran variedad de librerías de Python en la que destacan pytesseract y cv2. El primero de ellos es un contenedor para el motor Tesseract-OCR de Google. El caso de cv2 es el mismo, nos ofrece una API para acceder a OpenCV.

Como entorno de desarrollo se ha usado Pycharm en el caso de Python. Por otro lado, Processing ofrece su propio entorno de trabajo.

4.1 Python

Python es un lenguaje de programación interpretado cuya filosofía se basa en una sintaxis que favorezca un código sencillo y legible para el programador.

Es un lenguaje interpretado, de tipado fuerte y dinámico y multiplataforma. Tratándose de un lenguaje de programación multiparadigma, dado que soporta programación imperativa, orientado a objetos y, en menor medida, programación funcional.

Es administrado por la *Python Software Foundation*. Posee una licencia de código abierto, denominada *Python Software Foundation License* que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1.

Además, Python ofrece una enorme variedad de paquetes y librerías. En este proyecto se han utilizado las librerías **cv2**, **pyautogui**, **win32gui**, **PIL**, **pickle** y ***Beautiful Soup*** entre otras.

4.2 Processing

Processing es un entorno de desarrollo y lenguaje de programación basado en Java. Es un sistema sencillo de usar y es utilizado para la enseñanza y el desarrollo de programas multimedia interactivos.

El principal objetivo de Processing es obrar como herramienta para comunidades ajenas a la programación, como artistas o diseñadores visuales. Esto es debido en parte a que es una plataforma *plug and play*, es decir, simplemente se instala y se usa sin tener que configurar nada y sin ningún tipo de preparativo. A pesar de estar basado en Java, hace un uso simplificado de la sintaxis y de un modelo de programación de gráficos.

4.3 GitHub

GitHub es una plataforma de desarrollo colaborativo. Github es un plataforma web, que proporciona a los desarrolladores un servicio para almacenar y gestionar su código. De la misma forma permite llevar un registro y control de los cambios que se producen en el código utilizando un sistema de control de versiones Git.

Git es un sistema de control de versiones *open source* creada en 2005 por Linus Torvalds. Concretamente, Git es un sistema de control de versiones distribuido, es decir, no hay un servidor con una copia del repositorio, sino que la copia se mantiene entre los diferentes clientes que están haciendo uso del repositorio. Esto supone una serie de ventajas frente a un sistema centralizado sobre todo en proyectos de gran envergadura donde haya una gran cantidad de desarrolladores trabajando al mismo tiempo.

4.4 PyCharm

PyCharm es un entorno de desarrollo integrado específico para el lenguaje de programación Python. Está desarrollado por la compañía JetBrains presentando un comprobador de unidad integrado, un depurador gráfico, análisis de código, integración con sistemas de control de versiones y es compatible con el desarrollo web.

4.5 Tesseract

Tesseract es un motor OCR (Optical Character Recognition) es decir, una herramienta para reconocer texto en imágenes.

Tesseract fue originalmente dessarrollado por Hewllet Packard en el año 1980, que finalmente fue liberado como código abierto en 2005. Actualmente el proyecto Tesseract está siendo desarrollado por Google y distribuido bajo licencia Apache. Hoy por hoy Tesseract está considerado como una de las herramientas OCR con mayor precisión.

El tratamiento de la imagen está basado en una arquitectura paso por paso por filtros, pero algunos de los pasos eran un poco inusuales en su época y probablemente se mantienen así incluso ahora. El primer paso es un análisis de componentes conectados en el cual se almacenan los contornos de los componentes. Para la época este primer filtro fue una decisión de diseño computacionalmente cara, pero tenía una clara ventaja. Es simple detectar texto invertido (texto blanco sobre fondo negro) y reconocerlo tan fácilmente como el texto negro sobre fondo blanco gracias a la inspección de contornos anidados, y el número de contornos hijos y nietos. Tesseract fue probablemente el primer motor OCR capaz de manejar texto blanco sobre fondo negro de manera trivial. Llegados a este punto, los contornos estaban agrupados todos juntos, simplemente anidando, en regiones.

Las regiones están organizadas por líneas. Y las líneas se pueden analizar para caracteres de anchura fija o para caracteres de anchura proporcional. Las líneas de texto se separan en palabras de acuerdo al espacio de separación entre los caracteres. Los textos con caracteres de anchura fija se separan en celdas de forma inmediata y los textos con anchura proporcional se separan usando espacios definidos y difusos.

A continuación el reconocimiento procede con un proceso en dos pasos. En el primero de ellos se hace un intento de reconocer cada palabra. Cada una de ellas que satisface los requisitos se pasan a un clasificador adaptativo como datos de entrenamiento. Este clasificador tendrá entonces mayor precisión con el resto de las palabras a reconocer.

Una vez acabado de reconocer el texto por primera vez, se hace una segunda pasada por si ahora, que está el clasificador mejor entrenado, es capaz de hacer un reconocimiento en aquellas palabras que no fueron bien reconocidas al principio del texto.

4.6 OpenCV

OpenCV (Open source computer vision) es una libreria que contiene funciones principalmente enfocadas a la visisión por computadora en tiempo real. En sus inicios fue desarrollado por Intel. Desde su aparición en 1999 ha sido utilizado en infinidad de aplicaciones, esto se debe a que su publicación se da bajo licencia Berkeley Software Distribution (BSD).

Existen otras funciones como redimensionamiento o recorte de imágenes que se encuentran también en esta librería. Estas características junto la detección de objetos han decantado la integración de este software en el proyecto.

4.7 *Beautiful Soup*

Beautiful soup es una paquete de Python para hacer un análisis de sintaxis de documentos XML y HTML. Esta librería crea un arbol de análisis que se utiliza para extraer datos del documento, utilizado comúnmente para *web scraping*.

4.8 *Pickle*

El módulo de *pickle* implementa protocolos binarios para serializar y deserializar un objeto estructurado de Python. Esta librería permite convertir objetos en una secuencia de bytes y viceversa. El objetivo de esta librería es almacenar los *pickles* en un archivo o base de datos, o transferirlos a través de una red.

CAPÍTULO 5
Desarrollo

5.1 *Web scraping*

Para poder empezar con este proyecto necesitamos una base de datos con las cartas que se van a reconocer usando la información encontrada en la wiki del videojuego. Para obtener esta información hacemos uso de la librería ***Beautiful Soup*** de Python. Está biblioteca esta diseñada para analizar y extraer información de documentos HTML. Con esta librería se ahorra mucho tiempo a la hora de obtener información de páginas web.

La figura 3.2 muestra el ejemplo de una carta en la wiki. Las cartas estarían agrupadas por filas dentro de una tabla, siendo las columnas los datos de las cartas. Para hacer el *sraping* se ha usado el código mostrado en el *listing* 5.1. En las primeras líneas del código encontramos la llamada para obtener el HTML de la web. A continuación se busca la tabla que contiene las cartas y se recorren sus filas. Por último por cada fila se recorren sus columnas eliminando las que esten vacías.

```
1  import requests
2  from bs4 import BeautifulSoup
3
4  page = requests.get("https://slay-the-spire.fandom.com/wiki/Ironclad_Cards")
5  soup = BeautifulSoup(page.content, 'html.parser')
6
7  data = []
8  table = soup.table
9
10 table_body = table.find('tbody')
11 rows = table.find_all('tr')
12 for row in rows:
13     cols = row.find_all('td')
14     cols = [element.text.strip() for element in cols]
15     data.append([element for element in cols if element])
```

Listing 5.1: Código web scraping

5.2 Obtención cartas

Como se ha comentado anteriormente, el problema con la wiki es que las imágenes de las cartas estan desactualizadas. Para solucionar esto hemos hecho uso de un *mod* del juego que permite abrir una consola que nos permite, entre otras cosas, añadirnos y quitarnos cartas de la mano. Además del mod hemos hecho uso de la librería **pyautogui** que nos permite controlar por codigo el ratón y el teclado.

Mediante esta librería se ha hecho un bot que nos toma capturas de cada una de las cartas del juego. Para ello se recorre la lista de las cartas obtenidas en el apartado anterior. Con la consola habilitada por el mod, vamos añadiendo cada una de las cartas. A continuación toma una captura de esta y luego descarta la carta, repitiendo este proceso para cada una de las cartas.

5.3 Extracción de puntos

Para el reconocedor que se ha implementado en este proyecto, necesitamos extraer de cada carta de la mano una región alrededor de un conjunto de puntos en la imagen de la carta. Por esta razón necesitamos saber de cada carta la posición de estos puntos. Como podemos ver en la figura 5.1 las cartas se van solapando y colocando en la mano del jugador con una curvatura. Por tanto, no podemos simplemente escoger unos puntos y trasladarlos de forma trivial mediante el ancho de la carta. Por ello se ha implementado un *script* en Processing que ayuda a obtener la posición de estos.

Figura 5.1: Ejemplo mano del jugador

La primera parte del programa que se ha desarrollado en Processing nos muestra una imagen en pantalla y de forma interactiva mediante el ratón podemos marcar puntos sobre la pantalla. Además en todo momento se nos muestra una cruceta encima del ratón que podemos rotar con el uso del teclado. Con cada click del ratón marcamos en pantalla el punto. Del mismo modo añadimos a un *array* el punto y hacemos un *print* del mismo.

Una vez implementada la interacción ahora tenemos que marcar puntos sobre la imagen. Los puntos que marcamos son las esquinas izquierdas superiores de cada carta con la cruceta del ratón con un ángulo igual a la rotación de la carta. Esto lo hacemos para posteriormente dibujar un cuadrilátero alrededor de cada carta que ayudará a seleccionar puntos en la misma posición en todas las cartas para evitar el solapamiento.

Para dibujar el cuadrado que delimita cada una de las cartas hemos hecho uso del *listing* 5.2. Con este código lo que conseguimos es obtener las coordenadas de la pantalla de un punto normalizado. El punto esta normalizado respecto a una carta, por ello a esta función le pasamos como parámetro el array de las esquinas que habíamos recolectado anteriormente, la posición de la carta y el ancho de estas. Entonces si para cada una de las cartas hacemos uso de la función y unimos el resultado de los puntos normalizados [0,0], [0,1], [1,0] y [1,1] obtenemos el resultado de la figura 5.2.

Figura 5.2: Siluetas de la carta mediante Processing

También se ha implementado una función que pinta una cruceta en el punto [0,0] de cada carta y que podemos desplazar con el teclado. Con esta cruceta podremos seleccionar puntos característicos que esten contenidos dentro del arte de la carta que eviten el solapamiento y que sean comunes para todas ellas. Aparte esta cruceta también nos sirve para comprobar el ajuste de los puntos que hemos marcado anteriormente y de esta manera intentar conseguir una desviación mínima.

```
V2 normalizedToPixel(ArrayList<CornerAngle> ddbb, int card, float cardWidth,
    float x, float y) {

  CornerAngle org = ddbb.get(card);

  float ux = cos(radians(org.angle))*cardWidth * x;
  float uy = sin(radians(org.angle))*cardWidth * x;

  float vx = -sin(radians(org.angle))*cardWidth * y;
  float vy = cos(radians(org.angle))*cardWidth * y;

  V2 r = new V2();
  r.x = org.x + ux + vx;
  r.y = org.y + uy + vy;
  return r;

}
```

Listing 5.2: Transformación de un punto normalizado a coordenadas

Probando con diferentes cantidades de puntos, se ha optado por hacer uso de 20 puntos distribuidos por el arte de la carta, siendo esta cantidad suficiente para reconocer todas las cartas del juego. En la figura 5.3 se puede ver un ejemplo con los puntos de las carta que se extraen. Como se puede observar están todos situados sobre el lado izquierdo, ya que es el lado que no queda solapado cuando se acumulan muchas cartas en mano. Con estos puntos y traduciendo la función del listing 5.2 a Python procedemos desarrollar el reconocimiento de las cartas.

Figura 5.3: Puntos característicos de la carta

5.4 Reconocimiento de cartas

Para el reconocimiento se ha desarrollado un reconocedor basado en el color de las cartas. Lo primero que se ha desarrollado es una base de datos con la información de color que necesitamos de cada carta.

Se han utilizado las imágenes de las cartas que ya teniamos y los puntos obtenidos en el apartado anterior para llenar la base de datos con información de los colores. Para ello se ha creado un área circular alrededor de los puntos característicos. El uso de esta área es debido a la distribución de las cartas en la mano, siendo una región circular, invariante a la rotación. Por cada una de las áreas, se ha hecho la media de color RGB de los puntos contenidos en ellas. El uso de la media y no guardarse los colores píxel por píxel se debe a que el ajuste de estos puntos no es perfecto, entonces hacer una media suaviza el error que se podría obtener. Por lo tanto, nuestra base de datos queda conformada por un conjunto de medias para cada una de las cartas del juego.

Una vez tenemos ya la base de datos de los colores, simplemente tenemos que comparar una imagen de entrada, que sería el dibujo de la carta, con la información de la base de datos. Comparamos la imagen de entrada con cada una de las cartas en la base de datos, haciendo una diferencia de color de las regiones de los canales RGB de los píxeles contenidos en esa región. La carta reconocida sera la carta de la base de datos que menos diferencia nos de. Con esto ya hemos desarrollado un reconocedor simple, que nos identifica cualquier carta del juego dado el arte de la carta.

El siguiente paso es llevar este reconocedor al juego y para cualquier cantidad de cartas en mano. Para ello lo primero que tenemos que hacer es capturar la imagen del juego. Esto lo conseguimos utilizando la librería de Python **win32ui** que nos permite interactuar con la UI del sistema operativo Windows. El *listing* 5.3 es el código utilizado para tomar la captura del la ventana del juego. De la línea 3 a la 9 se busca la ventana del juego y se crea un contexto del dispositivo. En el siguiente conjunto de líneas se crea un bitmap a partir *output* del juego y por último en la linea 17 forma la imagen RGB del bitmap creado anteriormente.

```
1  import win32ui
2
3  self.hwnd = win32gui.FindWindow(None, 'Slay the Spire')
4  left, top, right, bot = win32gui.GetWindowRect(hwnd)
5  w = right - left
6  h = bot - top
7  hwnddc = win32gui.GetWindowDC(self.hwnd)
8  mfcdc = win32ui.CreateDCFromHandle(hwnddc)
9  savedc = mfcdc.CreateCompatibleDC()
10
11 save_bit_map = win32ui.CreateBitmap()
12 save_bit_map.CreateCompatibleBitmap(mfcdc, w, h)
13 savedc.SelectObject(save_bit_map)
14 bmpinfo = save_bit_map.GetInfo()
15 bmpstr = save_bit_map.GetBitmapBits(True)
16
17 hand_image = Image.frombuffer(
18     'RGB',
19     (bmpinfo['bmWidth'], bmpinfo['bmHeight']),
20     bmpstr, 'raw', 'BGRX', 0, 1)
```

Listing 5.3: Screenshot

El siguiente paso es conocer el número de cartas que tenemos en la mano para que el reconocedor extraiga de la base de datos la posición de las esquinas de las cartas, ya que

esta posición depende del número de cartas en mano. El número de cartas en mano esta limitado entre 0 y 10 cartas.

En este problema se ha optado de nuevo por una técnica de diferencia de color. En este caso se ha hecho uso del borde de las cartas, ya que es de un único color y por tanto facil de diferenciar. Para reconocer el número de las cartas, se ha guardado en la base de datos los puntos donde deberían estar las esquinas superior e inferior izquierdas de la última carta para cada uno de los diez posibles casos. Empezando por el caso mas alto comprobamos si la línea de píxeles trazada entre las dos esquinas tiene una media de color aproximada al color del borde de la carta. Si coincide podriamos determinar que nos encontramos en el caso de X cartas, si el color no coincidiese pasariamos al siguiente caso y así hasta llegar al caso de 0 cartas.

Una vez obtenida la cantidad de cartas en mano, se puede extraer de la base de datos las coordenadas de los puntos característicos correspondientes a dicha cantidad. Con estas coordenadas y con la cantidad de cartas en mano, se puede hacer uso de la función para transformar el punto normalizado a pixel que se ha traducido de Processing a Python. Por ejemplo, el punto A de la figura 5.4 tiene las coordenadas normalizadas (0.25, 0.25), si a la función se le pasa como parámetros el punto A, la posición de la carta y el número de cartas total en mano, se obtiene como resultado (910, 897) para una pantalla 1080p. Este resultado quiere decir que el punto normalizado A corresponde al píxel (910, 897) de la pantalla. Entonces con esta función se obtiene de la imagen total del juego los píxeles que corresponden a los puntos clave normalizados de cada una de las cartas. Con este conjunto de píxeles se haría uso del de la estrategia de reconocimiento que se ha explicado anteriormente, creando una región circular de píxeles y haciendo la diferencia de color de los canales RGB respecto a la base de datos.

Figura 5.4: Ejemplo normalización

El uso de puntos caracteríticos normalizados se debe a querer obtener de todas las cartas el mismo punto, ya que una carta puede estar en cualquier posición de la mano. Además, haciendolo de esta forma se ahorra trabajo a la hora de extraer estos puntos porque no se tienen que sacar puntos posición por posición, simplemente escogemos unos puntos normalizados e indicamos los límites de cada posición y ya obtendriamos puntos para cada una de ellas.

En resumen, para reconcocer las cartas de la ventana del juego se necesitan las imágenes de cada una de las cartas, la captura de la imagen del juego, la cantidad de cartas en mano y los puntos característicos normalizados. Las imágenes de las cartas se han obtenido con capturas del juego mediante un bot. Para la captura de la ventana del juego se ha hecho uso de la librería **win32gui** que nos permite resolver este problema de forma sencilla. El número de cartas en mano se ha obtenido buscando en la captura del juego en unos puntos preseleccionados si existe el borde de la carta para cada uno de los diez casos posibles. Y por último para obtener los puntos característicos, primero se deben tomar de la base de datos los puntos normalizados correspondientes al número de car-

tas y luego transformar estos puntos con la función mencionada anteriormente. Con esto queda implementado el reconocedor de cartas completo, que distingue todas las cartas que se encuentran en la mano del jugador. El siguiente paso es reconocer la posición de los enemigos y sus intenciones que queda desarrollado en el siguiente apartado.

5.5 Reconocimiento de enemigos

En este apartado se explica cómo se ha reconocido a los enemigos, tanto su posición, como su puntos de vida y las intenciones.

Para calcular la posición del enemigo, se ha optado por localizar las barras de vida en la pantalla del juego, por ser una línea de color que destaca entre el resto de elementos. Tras un estudio previo de la posición de los enemigos nos hemos dado cuenta que suelen estar siempre en la misma posición como se puede observar en la figura 5.5. Por ello se ha definido una *region of interest* (ROI) que nos permita acotar la búsqueda para reducir el coste computacional y nos facilite encontrar las barras de vida.

Figura 5.5: Región de interés

A continuación, una vez ya hemos obtenido nuestra región de interés el siguiente paso es localizar las barras de vida. Se ha creado un filtro, que elimina de la imagen los colores que no correspondan con un color aproximado al de las barras de vida. Obtendríamos a partir de un ejemplo como el de la figura 5.6 un resultado como el de la figura 5.7. Este filtro, además de indicarnos dónde se encuentra la posición de los enemigos, nos ayuda a localizar dónde se encuentra el texto que nos indica la vida del enemigo. Siendo la posición del texto el hueco entre dos secciones de vida, omitiendo los huecos pares donde no hay nada. Como la posición de los enemigos siempre se hace al principio del combate, las barras de vida de los enemigos siempre están al 100 % por tanto se asegura que siempre hayan dos trozos de barra de vida por enemigo. El problema con esto, como podemos ver en la segunda figura, es que el filtro deja pequeñas porciones de color rojo entre los espacios del texto. Para ello una vez hemos filtrado aplicamos un segundo filtro que nos elimine las islas de pixeles con menos de X pixeles.

Figura 5.6: Ejemplo barras de vida

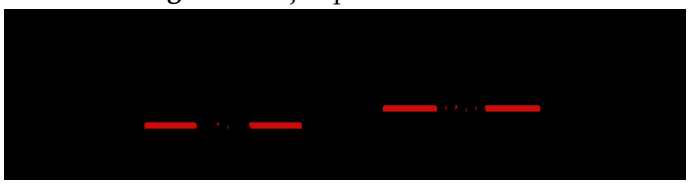


Figura 5.7: Ejemplo barras de vida filtradas

Ahora que ya conocemos la posición del texto que contiene los puntos de vida de los enemigos, el siguiente paso es reconocer el texto. Para este problema usamos la librería de python *pytesseract*, que es un contenedor para el motor de Google Tesseract-OCR que ya hemos explicado anteriormente. A fin de facilitar al motor el reconocimiento y aumentar la precisión, transformamos la región que contiene el texto a un escala de grises. Después aplicamos un método de valor umbral para eliminar los píxeles que no sean blancos. Podemos ver la transformación realizada en las figuras 5.8 y 5.9.

Figura 5.8: Texto limpio

Figura 5.9: Texto filtrado

Por último nos quedaría reconocer las intenciones del enemigo. En este caso se ha utilizado la librería de Python de OpenCV. Concretamente, hemos hecho uso de su método de *template matching* para encontrar las apariciones de una plantilla en una imagen. Para este caso no nos sirve el *template matching* si no que usaremos el *match* con máscara aplicada. Esto es debido a que la plantilla de las intenciones tienen un fondo como podemos ver en la figura 5.10. Este fondo va variando entre los diferentes niveles del juego, por tanto nunca encontraríamos la intención del enemigo. Además del fondo, también está el texto que va variando según el tipo de de enemigos que también causaria que fallase más veces la comparación de plantillas. Entonces, como hemos dicho antes, haremos uso de una máscara para evitar el problema.

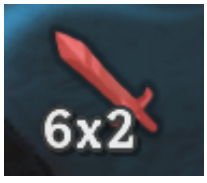
[h] 

Figura 5.10: Plantilla

[h] 

Figura 5.11: Máscara

Para crear la máscara hemos hecho uso de la herramienta libre y gratuita GIMP para edición de imágenes, dejando la parte de la imagen que caracteriza a cada intención en blanco y el fondo en negro. Podemos ver un ejemplo en la figura 5.11. Para hacer el match

hacemos uso del código en el listing 5.4.

```
1  import numpy as np
2  import cv2
3
4  img = cv2.imread('./image.jpg')
5  tmpl = cv2.imread('./tmpl.png')
6  mask = cv2.imread('./mask.png')
7  w, h = tmpl.shape[:-1]
8
9  res = cv2.matchTemplate(img, tmpl, cv2.CV_TM_SQDIFF, mask)
10 min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)
11
12 top_left = min_loc
13 bottom_right = (top_left[0] + w, top_left[1] + h)
14 cv2.rectangle(img, top_left, bottom_right, (0, 0, 255), 2)
15
16 cv2.imshow("images", np.hstack([img]))
17 cv2.waitKey(0)
```

Listing 5.4: *Match template* de las intenciones con máscara

En las 2 primeras líneas importamos el módulo de OpenCV y numpy. Este último lo utilizamos para trabajar con matrices. Entre las líneas 4 y 7 cargamos la imagen donde se hará la comparación, la plantilla y la máscara, además de extraer la anchura y altura de la plantilla. La línea 9 hace la función de OpenCV que realizan la comparación. La siguiente línea de código extrae el valor mínimo y el valor máximo de la comparación y sus respectivas posiciones en la imagen. En este caso al usar el método de CV_TM_SQDIFF, es decir, suma de diferencias al cuadrado (o SSD) buscamos el valor mínimo. Podemos ver un resultado del match en la figura 5.12.

Figura 5.12: Resultado *match template*

5.6 Jugador autónomo

El primer paso para implementar el jugador autónomo es desarrollar una forma de que pueda controlar automáticamente el ratón. Para ello existe, en Python una librería llamada *pyautogui* que nos provee, de la forma más sencilla posible, lo que buscamos. A continuación se muestra una conjunto reducido de los métodos que nos ofrece *pyautogui*.

En el listing 5.5 tenemos un conjunto de funciones que nos aportan información sobre el estado general del dispositivo.

```
pyautogui.position()  # Posicion actual del raton
(968, 56)
pyautogui.size()  # Resolucion de la pantalla
(1920, 1080)
pyautogui.onScreen(x, y)  # True si x e y estan dentro de la pantalla
True
```

Listing 5.5: Información general

En siguiente listing 5.6 se nos muestra dos constantes. La primera de ellas sirve para que se haga una pausa de X segundos entre llamada y llamada a funciones de pyautogui. La segunda de ellas sirve para habilitar una parada de emergencia moviendo el cursor en la esquina izquierda lanzando una excepción al programa. Al dejar el control del ratón al script es posible que alguna vez se quede en bucle o no se pueda manejar correctamente el ratón. Con esta función podemos cancelar la ejecución del programa.

```
pyautogui.PAUSE = 2.5
pyautogui.FAILSAFE = True
```

Listing 5.6: Constantes

A continuación el listing 5.7 nos muestra el conjunto de funciones correspondientes a las acciones del ratón. Las dos primeras funciones son simplemente para mover el ratón, siendo la diferencia que la primera mueve el ratón a las coordenadas X e Y de la pantalla y la segunda de ellas desplaza el ratón una posición relativa respecto a su posición actual. Ambas con una duración que se le pasa por parámetro.

Las dos siguiente funciones arrastran el ratón a una posición mientras mantienen el botón del ratón indicado en el parametro *button*. Al igual que en las anteriores, la diferencia entre una y otra es un desplazamiento relativo.

El siguiente conjunto de funciones sirven para accionar los botones del ratón. Aunque se puede hacer cualquier tipo de click solo con la función de la línea 11, existen el resto de funciones para aportar claridad al código. La función de la línea 16 permite hacer el *scroll*, siendo el parámetro *amount_to_scroll* positivo un *scroll* hacia arriba y en negativo un *scroll* hacia abajo.

```
# Mueve el raton a las coordenadas X e Y
pyautogui.moveTo(x, y, duration=num_seconds)
# Mueve el raton una distancia relativa
pyautogui.moveRel(xOffset, yOffset, duration=num_seconds, button='left')

# Arrastra el raton a la coordenada X e Y
pyautogui.dragTo(x, y, duration=num_seconds)
# Arrastra el raton una distancia relativa
pyautogui.dragRel(xOffset, yOffset, duration=num_seconds, button='left')

# Clicks del raton
pyautogui.click(button='right', clicks=3, interval=0.25)
pyautogui.rightClick(x=moveToX, y=moveToY)
pyautogui.middleClick(x=moveToX, y=moveToY)
pyautogui.doubleClick(x=moveToX, y=moveToY)
pyautogui.tripleClick(x=moveToX, y=moveToY)
pyautogui.scroll(amount_to_scroll, x=moveToX, y=moveToY)
```

Listing 5.7: Acciones del ratón

Además de las funciones con el ratón, *pyautogui*, incluye funciones para el manejo del teclado, como por ejemplo pulsar una tecla, realizar una secuencia de teclas para usar los atajos, escribir un texto... Pero para jugar al videojuego Slay the Spire no es necesario el uso del teclado. Por tanto, solo usaremos las funciones del ratón mencionadas anteriormente.

Nuestro jugador autónomo ahora mismo solo consta de 2 acciones, pasar turno y jugar carta. La primera de ellas es trivial, ya que únicamente tendremos que hacer que el ratón se dirija a la posición del botón que pasa el turno del jugador. Esta posición es fija en todos los niveles del juego, por ello es un problema trivial. La segunda acción tiene dos casos diferentes. Jugar una carta de ataque o jugar una carta de habilidad. La diferencia entre cada uno de ellos es la posición donde se debe lanzar la carta, siendo el primero de ellos la posición del enemigo y la segunda una posición distinta donde está el enemigo. Por tanto para ambos casos necesitaremos conocer la posición de las cartas y la posición del enemigo. Ambos valores ya los hemos obtenido en los apartados anteriores.

Para seleccionar las cartas, tenemos todas las posiciones de todas las esquinas superiores izquierdas de cada carta que tenemos en la mano. Por tanto, simplemente tenemos que simular que el jugador hace clic en esa posición con un pequeño desplazamiento hacia abajo y derecha, para asegurarnos que se seleccione correctamente la carta.

La posición del enemigo se conoce a través de la posición de la barra de vida, concretamente la zona donde se encuentra el texto de la barra de vida. Entonces en el caso de las cartas de ataque simplemente tendremos que desplazar el cursor hacia una posición un poco más arriba que la del texto. Y para el caso de las cartas de habilidad, a una posición lejana a la del texto.

En resumen, la ejecución de pasar turno consistirá en dirigir el ratón hacia el botón que realiza esta acción. Y, para jugar una carta, tendremos que mover el ratón primero hacia la posición de la carta y luego dependiendo del tipo de carta hacia el enemigo o hacia una zona diferente del enemigo, por ejemplo, la posición de nuestro personaje.

Por último, nos queda la estrategia, que en esta primera versión será una muy sencilla. El jugador autónomo conoce las cartas que tiene en la mano, sabe si son ataques o si son defensa, entonces la estrategia a seguir es jugar cartas de ataque, teniendo prioridad las que mas puntos de daño tenga. El jugador pasa turno cuando se queda sin puntos de maná o sin cartas de ataque. En el segundo caso, como aún queda maná, el jugador lanza cartas de defensa hasta quedarse sin puntos, teniendo otra vez preferencia las cartas con mayor puntos de defensa. Una vez se ha quedado sin puntos de maná, el jugador finaliza el turno.

CAPÍTULO 6
Conclusiones

En conclusión, el desarrollo de este proyecto ha consistido en 3 fases principales: recolección de datos, reconocimiento y automatización. Para la primera de ellas, se han usado técnicas de *web scraping* para obtener información de los elementos del videojuego en su wiki. También se ha implementado un bot sencillo que tomaba capturas para adquirir las imágenes que posteriormente se usarían en el reconocimiento. A continuación, la fase del reconocimiento ha consistido en el análisis de la interfaz del juego y sus distintos elementos para elegir el método de reconocerlos que mejor se ajustase a cada uno, y en la implementación de estos. Por último, la fase de automatización ha consistido en la inyección de eventos de ratón, que es el dispositivo de entrada con el que se controla el juego. Estos movimientos dependerán de las cartas que se tengan que jugar en base a la estrategia que se ha diseñado.

Respecto los objetivos propuestos al principio del proyecto, podemos ver que se han cumplido cada uno de ellos. Aunque el último de ellos, que trata sobre la estrategia que sigue el jugador autónomo se podría mejorar, como se comentará en el siguiente capítulo.

El jugador autónomo consigue superar un promedio de 5 combates sin morir, dependiendo bastante del tipo de enemigo con el que se enfrente, ya que cada uno requiere un tipo de estrategia. Este resultado es bastante malo si se tiene en cuenta que cada partida cuenta con un total de 30 combates. Esto se debe a que la estrategia es muy trivial, pudiéndose mejorar notablemente. Además, la automatización ahora mismo se centra en el combate y no utiliza todas las mecánicas disponibles. Por otro lado, los resultados respecto al reconococimiento son muy satisfactorios, consiguiendo una precisión del 100 %. Para obtener este resultado se ha utilizado el mismo bot que se empleó en la toma de capturas de las cartas, pero en este caso se hizo capturas de cada una de las cartas en todas las posiciones posibles. Una vez obtenidas estas imágenes se lanzó el reconocedor para cada una de ellos obteniendo un resultado positivo.

En este proyecto se han aplicado muchos de los conocimientos aprendidos durante la carrera. Está bastante relacionado con la asignatura de SGI (Sistemas Gráficos interactivos), pero muchas otras disciplinas han enseñado a cómo enfrentarse a un problema y gestionar su desarrollo.

Por otro lado he aprendido a hacer uso de nuevas tecnologías que no conocía. Entre ellas Tesseract, un motor de reconocimiento de caracteres, y Processing, un entorno de desarrollo de aplicaciones interactivas.

En resumen, el desarollo de este proyecto ha sido muy satisfactorio. Por una parte, he aprendido a usar nuevas tecnologías y he reforzado los conocimientos que ya tenía. Además, he trabajado con uno de mis videojuegos favoritos.

CAPÍTULO 7
Trabajos futuros

Tras un vistazo rápido a una partida del juego Slay the Spire podemos observar que nuestro proyecto dista bastante de poder conseguir la victoria. Por ello a continuación, se hablará del trabajo a realizar a partir de esta primera versión para poder llegar a conseguir la victoria.

- Reconocimiento de todos los elementos: Como se puede ver en una partida de Slay the Spire, hay muchos más elementos de los que actualmente se están reconociendo. Por ejemplo, al acabar un combate, podemos obtener recompensas como las pociones. Estas pociones se nos irán guardando en un inventario y tienen poderosas habilidades que nos ayudarían a ganar el combate. Otro ejemplo son las reliquias, también obtenidas al acabar un combate, y que nos ofrecen habilidades pasivas que a lo largo de un combate pueden ser muy rentables.
- Automatización total: Ahora mismo sólo se automatiza un combate, que además no hace uso de todas las herramientas. Por ejemplo, las pociones que se han comentado antes. Pero, a parte de los combates, hay más funciones que se podrían automatizar como la selección de recompensas o la selección de niveles.
- Mejora de la estrategia: La estrategia que sigue ahora mismo nuestro jugador autónomo es muy simple. Usa cartas de ataque hasta que no le quedan en la mano, luego si aún tiene maná, juega cartas de defensa. Esta técnica, además de ser sencilla, no es suficiente para conseguir la victoria. Debería tener más en cuenta las intenciones del enemigo que, a pesar de ser reconocida, no se tienen en cuenta. También habría que tomar acciones según el enemigo que tenemos enfrente ya que cada uno de ellos posee unas características y habilidades distintas.

Bibliografía

[1] Mark Lutz. *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media, Inc. 2013.

[2] Ray Smith. *An Overview of the Tesseract OCR Engine.*

[3] OpenCV Documentation. https://docs.opencv.org/master/

[4] Bradski, Gary, and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc. 2008.

[5] Al Sweigart. *Automate the Boring Stuff with Python.* No Starch Press. 2015

[6] John C. Russ. *The Image Processing Handbook, Seventh Edition*. CRC Press. 2015.

[7] Mallick, Satya. (2016). *A Brief History of Image Recognition and Object Detection*. Consultado en https://www.learnopencv.com/image-recognition-and-object-detection-part1/. 2016.

[8] *Color Filtering OpenCV Python Tutorial*. Consultado en https://pythonprogramming.net/color-filter-python-opencv-tutorial/.

[9] *Reading game frames in Python with OpenCV - Python Plays GTA V*. Consultado en https://pythonprogramming.net/game-frames-open-cv-python-plays-gta-v/.

[10] *Multi-scale Template Matching using Python and OpenCV*. Consultado en https://www.pyimagesearch.com/2015/01/26/multi-scale-template-matching-using-python-opencv/.

[11] Ojala, T., Pietikainen, M. y Maenpaa, T. (2001). *Multiresolution gray-scale and rotation invariant texture classification with local binary patterns*. 10.1109/TPAMI.2002.1017623

[12] AlphaStar: Mastering the Real-Time StrategyGame StarCraft II. https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii.

[13] Israel Viena. [2010]. Deep Blue, la máquina que desafió la inteligencia humana https://www.abc.es/historia/abci-deep-blue-maquina-desafio-inteligencia-humana-201002110300-1133709000633_noticia.html

[14] AlphaGo is the first computer program to defeat a professional human Go player, the first to defeat a Go world champion, and is arguably the strongest Go player in history. https://deepmind.com/research/case-studies/alphago-the-story-so-far

[15] StarCraft II: Wings of Liberty. (2019, 27 de agosto). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 15:33, agosto 29, 2019 desde https://es.wikipedia.org/w/index.php?title=StarCraft_II:_Wings_of_Liberty&oldid=118591408.

[16] Slay the Spire website. https://www.megacrit.com/

[17] *Start using the console to test things in Slay the Spire*. Consultado en https://github.com/daviscook477/BaseMod/wiki/Console.

[18] Slay the Spire wiki. https://slay-the-spire.fandom.com/wiki/Slay_the_Spire_Wiki

[19] Casey Reas. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press. 2015.

[20] DeepFace: Closing the Gap to Human-Level Performance in Face Verification. Facebook Research. Retrieved 2019-07-25. https://research.fb.com/publications/deepface-closing-the-gap-to-human-level-performance-in-face-verification/

[21] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei. *ImageNet Large Scale Visual Recognition Challenge*. International Journal of Computer Vision, 2015

[22] ImageNet Classifation table. https://paperswithcode.com/sota/image-classification-on-imagenet