5. RESULTADOS

En este capítulo se muestran los resultados obtenidos a lo largo del desarrollo del proyecto, describiendo cada una de sus iteraciones. Por otro lado, se explicará el algoritmo diseñado, los experimentos realizados y la aplicación gráfica.

5.1 Funcionalidades del sistema

En este punto vamos a mostrar una visión general del software final relacionando los conceptos más importantes. El software final es una aplicación web donde el usuario puede visualizar el aprendizaje por refuerzo de un agente en un mundo introducido por el propio usuario. El esquema funcional del sistema se muestra en Fig. 5.1.

En los siguientes apartados se muestran detalladamente el trabajo, desarrollo y análisis de cada una de las iteraciones que se han tenido que realizar para el desarrollo de este software final.

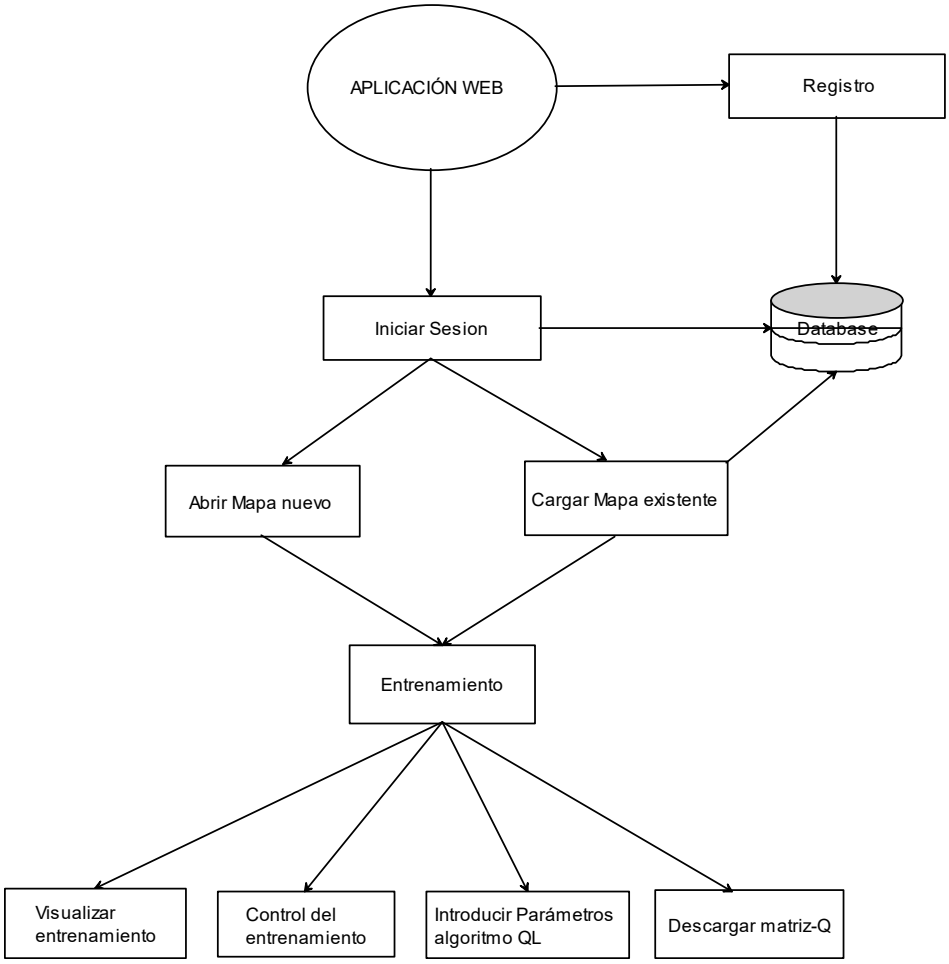

Figura 5.1: Esquema funcional del sistema.

5.2 Iteración 1: Lectura del mapa de entrenamiento

La primera iteración del desarrollo del sistema se basa en la primera toma de contacto con los entornos para editar el mapa y su correspondiente lectura en JavaScript. Se va a dividir este apartado en cada una de las historias de usuario que se han llevado a cabo en esta iteración.

5.2.1 Estudio y uso del entorno Tiled

En la reunión de planificación para esta iteración se discutió el Framework que se utilizaría para que el usuario de la aplicación pudiera crear sus mapas o mundos. Al final se llegó a la conclusión de que el mejor Framework sería Tiled, ya que es una de las herramientas más famosas para editar mapas basados en celdas. Las características más importantes por las que se ha elegido Tiled se enumeran a continuación:

- Es posiblemente el Framework más flexible para la edición de mapas.
- No hay restricciones en el tamaño del mapa ni de la celda.
- No hay límite en el número de capas o conjuntos de patrones.
- Permite establecer propiedades a los elementos del mapa, cosa más que imprescindible para poder reconocer en cada momento donde se sitúa el agente.
- Permite exportar el formato en formato Translation Memory Exchange (TMX), JSON o CSV. Esta variedad de formatos ayuda mucho a que luego se pueda leer ese mapa en nuestro sistema.

Una vez hecho esto, se dispuso a la descarga e instalación de Tiled. Tiled puede ser instalado tanto en Windows como en distribuciones de Linux. Hay que añadir que el presente TFG se ha desarrollado en torno a la versión 0.17.2 de Tiled ya que es una de las versiones más apropiadas que nos permite crear mapas leíbles en el futuro. Con el objetivo de probar un poco la herramienta, se hicieron algunas pruebas sobre su funcionamiento. A continuación, se describirá como se ha creado un mapa básico en Tiled a partir de un conjunto de patrones:

- Primero se ha seleccionado la opción de menú *Archivo* > *Nuevo* donde se ha especificado el tamaño del mapa en celdas y el tamaño de cada celda. Finalmente se ha pulsado el botón aceptar.
- Después se ha añadido el conjunto de patrones para formar el mapa a partir de los patrones que se ha querido que formen el mapa. Se ha seleccionado la opción

Nuevo conjunto de patrones, que se encuentra en el conjunto de opciones de abajo a la derecha, y se le ha dado al botón explorar para añadir la imagen correspondiente. El tamaño de cada celda se ha modificarlo al gusto para que luego se añada al mapa. Hay que señalar que el conjunto de patrones ha tenido que ser una imagen en formato PNG o JPG, cuyos patrones han tenido que estar divididos en celdas iguales para que Tiled haya podido reconocerlo.

- Posteriormente se muestra toda la interfaz con todas las opciones disponibles para la edición del mapa. En Fig.5.3 se puede observar esta interfaz gráfica. En la parte de arriba está la barra de herramientas con las opciones para gestionar imágenes y objetos. En el recuadro de arriba a la izquierda se muestran las diferentes capas de nuestro mapa, mientras que en el recuadro de abajo se puede observar el conjunto de patrones añadido previamente. En la columna de la izquierda se pueden observar cada uno de los atributos del mapa, como los de cada cerda u objeto. Y finalmente, en el centro está el mapa que se está editando. Para añadir más capas, ya sean de patrones u objetos, se ha pulsado la opción Añadir Capa, que se encuentra debajo del listado de capas.
- Una vez conocidas estas opciones, se han arrastrado patrones o celdas al mapa, formando así el mapa deseado. Después de crear la capa o las capas de patrones, se ha añadido una capa de objetos para especificar las partes de colisiones y para especificar el tipo de objeto que se trata. Esto es una característica de esencial importancia para la futura lectura del mapa, como se detallará próximamente.
- Por último, en Fig. 5.4 se puede ver el ejemplo de un mapa creado. Este es un mapa adecuado para su futura lectura por Phaser, ya que entre otras cosas se le ha añadido objetos. En un punto futuro se detalla la forma de crear un mapa exactamente para que Phaser lo lea. Cuando tenemos este mapa, finalmente hay que exportarlo en formato JSON.

5.2.2 Estudio del entorno Phaser

Phaser es el entorno o librería que se ha utilizado en JavaScript para leer el mapa que se ha editado. Phaser es probablemente el motor de juegos de HTML5 más popular. Las razones por las que se ha tomado la decisión de elegir a Phaser para la lectura de mapas son las siguientes:

- Al ser el motor de juegos más popular, tiene una excelente documentación, así como un número enorme de ejemplos para poder aprender del entorno.
- Tiene un excelente rendimiento ya que usa Pixi.js como motor de renderizado. Pixi.js usa WebGL o Canvas dependiendo del dispositivo utilizado.
- Carga los recursos por adelantado, reduciendo mucho la complejidad del código.
- Tiene tres motores físicos integrados y gratis para usar: Arcade Physics, AABB y Ninja Physics.
- Permite crear Sprites para añadirlos al mapa.
- Puede crear grupos de Sprites para facilitar la edición grupal permitiendo así una reducción de tiempo de desarrollo y complejidad de código.
- Tiene una cámara genial para permitir el enfoque que se desee.
- Permite la lectura de mapas Tiled. Esta es una de las características más importantes por las que se ha elegido Phaser. Tiene un montón de funciones para manejar y gestionar mapas formados por celdas, pero estos deben tener el formato apropiado que se mostrará a continuación.
- Es un entorno muy probado y utilizado, por lo tanto, asegura que es un sistema robusto con la documentación y solución de dudas suficiente.

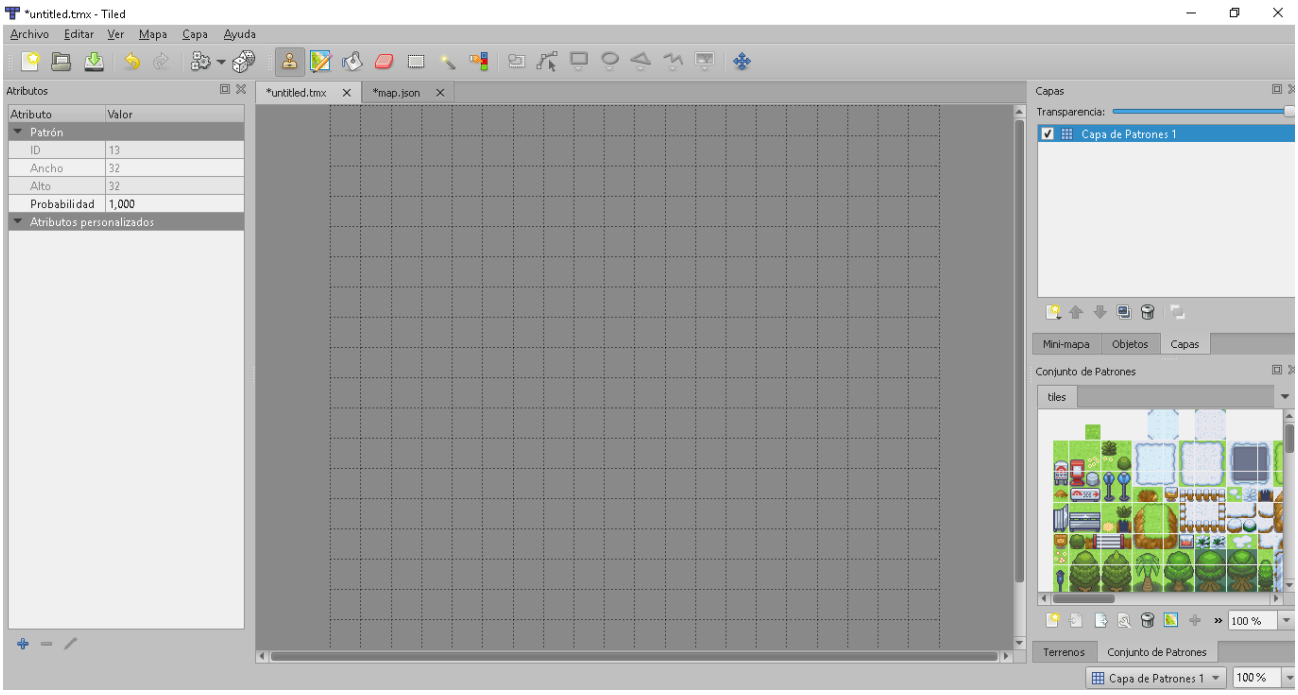

Figura 5.3: Interfaz gráfica de Tiled.

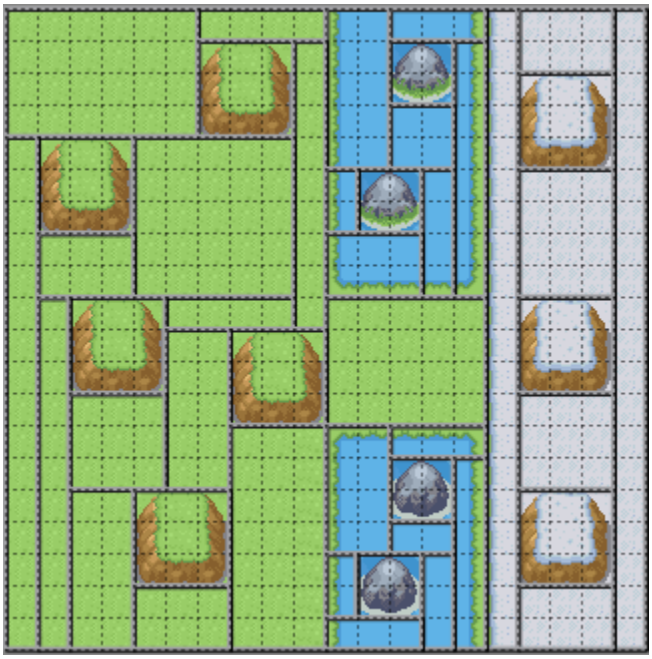

Figura 5.4: Mapa Tiled.

5.2.3 Edición adecuada del mapa en Tiled

En este apartado se explica como se ha editado el mapa en Tiled para que Phaser pueda leerlos. Hay que remarcar que los pasos seguidos han tenido que ser precisos, ya que de otra forma Phaser lanzaba errores de lectura.

Como se menciona anteriormente, la versión usada de Tiled ha sido la 0.17.2. Esta es una de las versiones que es capaz de exportar archivos JSON adecuados para Phaser. Se ha empezado usando la ultima versión, que es la 1.0.3, pero se ha visto que los archivos JSON exportados no eran soportados por Phaser. Por lo tanto, se ha tenido que reducir drásticamente la versión para lograr la aceptación de Phaser.

Aparte de la versión, se han realizado unas ligeras modificaciones o añadidos a la lista de pasos para haber creado el mapa del apartado 5.2.1. Los pasos añadidos son los siguientes:

- Al darle a *Archivo* > *Nuevo* y salir la ventana emergente de *Crear un nuevo mapa*, hay que dejar seleccionado el formato CSV para el conjunto de patrones.

- Al abrir un nuevo conjunto de patrones, en la versión de Tiled de Ubuntu, cuando sale la ventana emergente para elegir el archivo, se ha tenido que marcar la opción del CheckBox de *Adjuntar al mapa*. Esta opción es importante ya que en el JSON creado se muestra el conjunto de patrones adjunto con el que se ha creado ese mapa, y Phaser tiene que saber donde se encuentra el archivo del conjunto de patrones que forma el mapa.
- Tiled permite añadir más de un conjunto de patrones para crear un mapa, sin embargo, el sistema que se ha creado solo permite la lectura de un conjunto de patrones adjunto al mapa. Si se quiere crear un mapa con más de un conjunto de patrones, el usuario es el encargado de crear un conjunto de patrones que contenga todos los patrones necesarios; y Tiled es una herramienta que permite facilmente esa edición.
- Los mapas creados han tenido que tener una única capa de objetos y una o varias capas de patrones. Phaser es capaz de leer más de una capa de objetos, pero para el presente TFG solo es necesario una capa de objetos para especificar el tipo de entorno en cada celda del mapa. Un ejemplo de estructura de capas se puede observar en Fig. 5.5 donde se puede observar una capa de objetos y otra capa de patrones.
- Para que el agente de nuestro sistema pueda saber donde se encuentra en cada momento, se ha tenido que crear una capa de objetos, donde cada celda del mapa está sobre un objeto. Por lo tanto, para añadir objetos se ha tenido que seleccionar la capa objetos, pulsar la opción *Insertar rectángulo* y arrastrar con el cursor para cubrir la parte del mapa con el rectángulo. En Fig. 5.6 se puede observar la opción marcada y como el cuadrado cubre una montaña. Todos los mapas han tenido que estar cubiertos por objetos, como se puede observar en Fig. 5.4, donde el mapa está cubierto por rectángulos.
- A continuación, se ha tenido que añadir una propiedad *Type* a cada objeto creado. Esta propiedad ha sido asignada de tipo *String* y hay que ponerle el tipo de elemento que es, para que luego nuestra aplicación de JavaScript sepa el tipo de elemento de cada patrón. Para terminar, se ha tenido que exportar el mapa en formato JSON y dejado el archivo de conjunto de patrones en el directorio asignado en el archivo JSON. Este último paso no es estrictamente necesario ya

que Phaser leerá correctamente el archivo JSON, pero para tener todo ordenado se ha decidido dejar los conjuntos de patrones en la carpeta correspondiente.

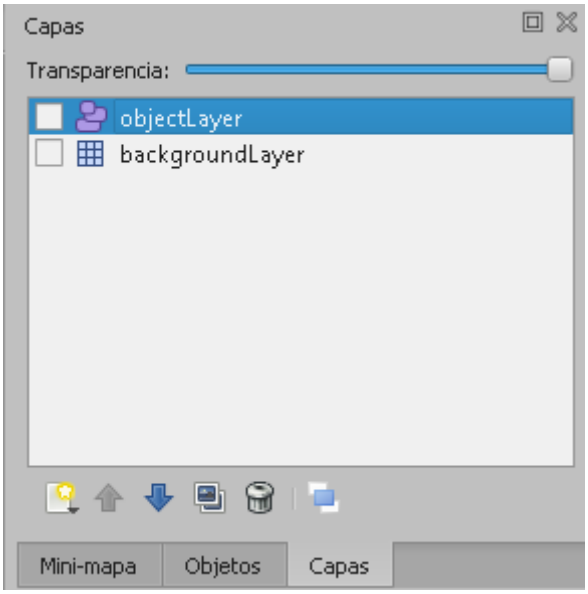

Figura 5.5: Capas del mapa.

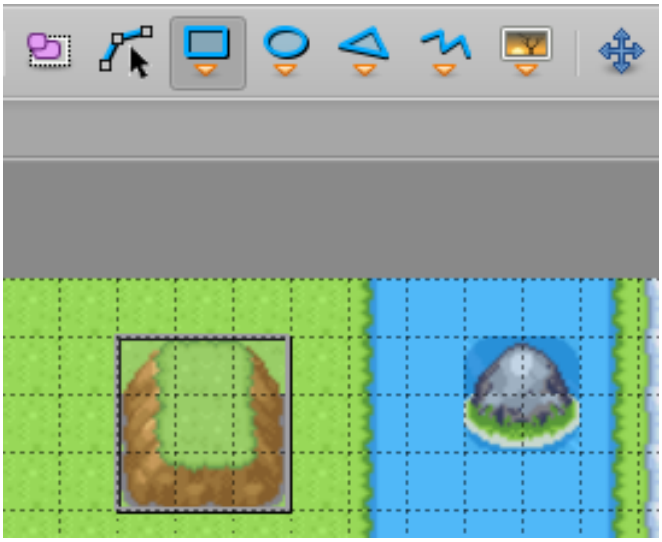

Figura 5.6: Creación de objetos en el mapa.

Para terminar este apartado, se ha tenido que analizar la estructura del archivo JSON creado. Este mapa JSON contiene toda la información de cada capa de patrones y de objetos, así como su conjunto de patrones asociado. No es necesario explicar detalladamente cada uno de los atributos del archivo, ya que Phaser hace todo el trabajo

de lectura. Por lo tanto, se van a explicar los atributos del archivo JSON que son necesarios de analizar:

- Height: Se trata del número de celdas a lo alto que tiene el mapa.
- Width: Es el número de celdas que tiene el mapa a lo ancho.
- Tileheight: Son los pixeles que forman la celda a lo alto.
- Tilewidth: Son los pixeles que forman la celda a lo ancho
- Tilesets: Se trata de una lista con los archivos de conjuntos de patrones que forman el mapa. Como en esta aplicación solo se permite uno, la lista tendrá un único elemento. Este conjunto de patrones muestra sus características, como las columnas que tiene, su directorio, el nombre del archivo, los pixeles a lo ancho y lo alto, el tamaño de la celda, etc.
- Layers: Es una lista con todas las capas que forman el mapa. En el ejemplo creado hay una capa de patrones y una capa de objetos. La capa de patrones tiene el atributo Data, que es una lista con el identificador de cada uno de los patrones que forman la capa, así como el tamaño del mapa y el nombre de la capa. Mientras que la capa de objetos tiene como atributos su nombre y una lista de objetos que lo forman. Cada objeto de la lista muestra su tamaño en pixeles, su posición en el mapa y sus propiedades. En estas propiedades es donde se puede observar el tipo de elemento que es ese objeto.

5.2.4 Implementación de la lectura del mapa con Phaser

Para leer los mapas creados con Tiled se ha tenido que crear un prototipo en JavaScript para probar el funcionamiento de Phaser con Tiled, así como para ver como compaginan entre sí.

Como ejemplo de mapa se ha utilizado el creado por los puntos anteriores. La estructura de carpetas empleada ha sido por un lado el archivo HTML y por otro los directorios *Assets* y *Js*. En *Assets* se encuentran las imágenes empleadas y el mapa Tiled, mientras que en *Js* se encuentran los archivos y librerías de JavaScript. Mantener estructurado el directorio es un aspecto muy importante en el desarrollo de videojuegos web.

En cuanto al archivo HTML, se han importado cada una de las clases del juego Phaser. En la etiqueta *Body* se ha llamado a la clase *Main*, mientras que en la etiqueta

Head se han importado la librería Phaser y las clases *Preload* y *Game*. Los estados en Phaser son partes separadas de la lógica del juego, por lo que las clases *Preload* y *Game* son los dos estados del juego. En *Preload* se cargan los gráficos utilizados mientras que en *Game* comienza el juego. La transición de un estado a otro es inapreciable cuando se ha implementado el juego ya que no hay ningún menú de juego ni nada por el estilo, únicamente se cargan las imágenes y el mapa en *Preload* y en *Game* se lanza el Juego.

Para analizar mejor la relación y contenido de estos estados del juego, se puede observar la Fig. 5.7, donde se muestra el Diagrama de clases asociado.

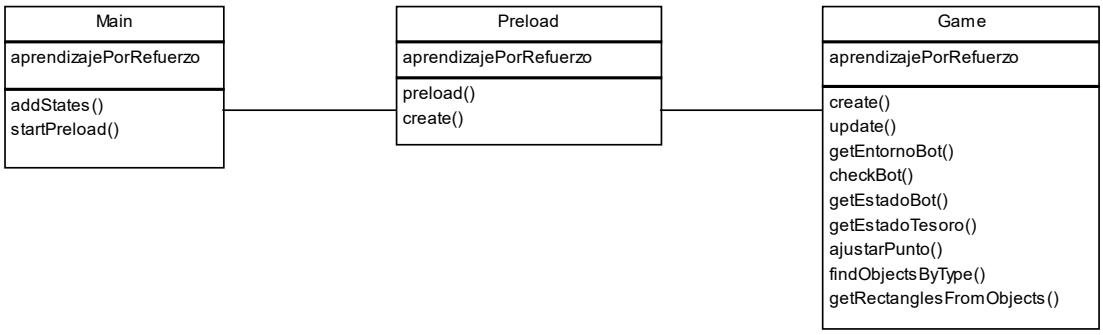

Figura 5.7: Diagrama de clases.

Como se puede observar, las tres clases tienen un único atributo, que es el objeto del juego. Este objeto es utilizado en cada estado ya que contiene los diferentes estados del juego. En el Listado 5.1 muestra la forma de inicializar este objeto en todas las clases.

```
1   var AprendizajePorRefuerzo = AprendizajePorRefuerzo || {};
```

Listado 5.1: Inicialización del objeto que contiene el juego.

En la clase *Main* tiene dos métodos. Los métodos se describen a continuación:

- **AddStates():** En este método se inicializa el juego con la librería Phaser. Se crea un objeto Game cuyos parámetros son la altura y anchura del juego en píxeles y el tipo de renderizado. Cuando se ha inicializado el juego en Phaser se han añadido los estados *Preload* y *Game* gracias a la función *state.add()*, cuyos parámetros son el nombre en clave del estado y la funcion donde está. En el Listado 5.2 se puede ver el código del método.

- **StartPreload():** Con este método únicamente se pasa al estado inicial *Preload*. Para ello se utiliza la función *state.start()*, cuyo parámetro es el nombre en clave del estado a donde se quiere realizar la transición. En el Listado 5.3 se puede ver la línea de código de este método.

```
3   AprendizajePorRefuerzo.game = new Phaser.Game(320, 320, Phaser.AUTO, '');

5   AprendizajePorRefuerzo.game.state.add('Preload', AprendizajePorRefuerzo.Preload);
6   AprendizajePorRefuerzo.game.state.add('Game', AprendizajePorRefuerzo.Game);
```

Listado 5.2: Código del método *addStates()*.

```
8   AprendizajePorRefuerzo.game.state.start('Preload');
```

Listado 5.3: Código del método *startPreload()*.

La clase o estado *Preload*, como se ha mencionado antes, se encarga de la carga de los gráficos del juego y la transición al estado *Game* para que empiece el juego. En este estado se crea su función y se le añade la propiedad *prototype* para que esa función pueda ser llamada como un objeto. En esa función hay dos métodos cuyo código se puede ver en el Listado 5.4. Estos métodos son los siguientes:

- **Preload():** En este método cargamos el mapa Tiled con la función *load.tilemap()*, cuyos parámetros son el nombre en clave del mapa, la URL del mapa y el formato de este. También se cargan las imágenes con la función *load.image()*, donde sus parámetros son el nombre en clave de la imagen y la URL de la imagen.
- **Create():** Este método ha desempeñado la función de indicar el motor físico del juego, indicando el tipo Arcade. Y finalmente se ha pasado al estado *Game*.

El estado *Game* es el estado principal del juego, ya que contiene prácticamente toda la lógica de este. Como en el estado *Preload*, se ha creado su función y se ha añadido la propiedad *prototype*. Dentro de la función hay nueve métodos, de los cuales se describen los aspectos más importantes.

```
14   AprendizajePorRefuerzo.Preload.prototype = {
15       preload: function() {
16           this.load.tilemap('map', urlMapa, null, Phaser.Tilemap.TILED_JSON);
17           this.load.image('gameTiles', urlTileset);
18           this.load.image('tesoro', urlTesoro);
19           this.load.image('bot', urlBot);
20           this.load.image('circuloAmarillo', urlCirculo);
21       },
22       create: function() {
23           this.game.physics.startSystem(Phaser.Physics.ARCADE);
24           this.game.state.start('Game');
25       }
26   };
```

Listado 5.4: función *prototype* del estado *Preload*.

Create()

En esta función se han inicializado la mayoría de los atributos del juego y se han añadido los elementos respectivos al juego. En el Listado 5.5 se puede ver la forma de añadir el mapa Tiled y el conjunto de patrones al juego. También se ha tenido que especificar la capa de patrones del mapa Tiled, por lo tanto se ha añadido con la función *map.createLayer()*, donde el argumento es el nombre de la capa que se le ha asignado en Tiled.

En el Listado 5.6 se muestra la forma de añadir la imagen o Sprite del agente al juego. Por lo tanto, esto sirve para añadir el Sprite del agente y del tesoro. Para ello se usa la función *add.tileSprite()*, donde los parámetros son la posición X e Y en el mapa en píxeles, donde se ha usado la función *ajustarPunto()* que se explica más adelante, el tamaño del Sprite en píxeles y el nombre en clave de la imagen que usará como Sprite. Este es el nombre en clave cargado en el estado *Preload*. Una vez creados los Sprites hay que ajustarlos en el mapa, por lo que se ha escalado la imagen en un 80% con la función *scale.setTo()* y se ha ajustado el centro del eje de coordenadas del propio Sprite en el centro con la función *anchor.setTo()*, donde se le ha pasado el parámetro 0.5.

```
7    this.map = this.game.add.tilemap('map');
8    this.map.addTilesetImage('tiles2', 'gameTiles');
9    this.backgroundlayer = this.map.createLayer('backgroundLayer');
10   this.backgroundlayer.resizeWorld();
```

Listado 5.5: Código para añadir el mapa Tiled y el conjunto de patrones

También, se ha tenido que habilitar el motor físico a los Sprites creados con la función *physics.enable()*, cuyos parámetros son los Sprites a los que se quiere añadir el motor y el tipo de motor. También hay que especificar que los Sprites no pueden salir del límite del juego, eso se hace asignando *True* a la propiedad del Sprite *body.collideWorldBounds.*

Por último, hay que crear una lista para cada tipo de elemento del mapa. Para ello se han utilizado las funciones *findObjectsByType()* y *getRectanglesFromObjects()*, las cuales se explican más adelante. Los elementos del entorno de este mapa son las distintas propiedades añadidas anteriormente en Tiled. En este ejemplo los elementos son césped, agua, hielo, islas, montaña helada y montaña normal.

```
11    this.inicioBotX = this.ajustarPunto("x");
12    this.inicioBotY = this.ajustarPunto("y");
13    this.bot = this.game.add.tileSprite(this.inicioBotX, this.inicioBotY, 16, 16, 'bot');
14    this.bot.anchor.setTo(0.5);
15    this.bot.scale.setTo(0.8);
```

Listado 5.6: Código para añadir el Sprite del agente.

Update()

En este método se crea la lógia del propio juego ejecutandose constantemente. Por ejemplo, comprueba si se ha pulsado una tecla o si se ha chocado contra un objeto. En este ejemplo se creó un cursor para manejar el agente con el teclado y una detección constante por si el agente había encontrado el tesoro con la función *checkBot()*, que se verá más adelante. No se necesario mostrar el código completo de esta función ya que no tiene relación con la aplicación futura, solo se hizo para probar el motor físico y la colisión con el tesoro.

GetEntornoBot()

Este método tiene como finalidad comunicar el elemento del entorno donde está el agente. Para ello se utiliza la lista de objetos creados en el método *create()* y la función de phaser *Rectangle.intersects()* para ver si el sprite del agente se encuentra sobre tal objeto. En el Listado 5.7 se puede ver el código para ver la intersección entre los objetos de césped y el agente.

```
65   for (var i = 0; i < this.cespedRectangles.length; i++) {
66       if (Phaser.Rectangle.intersects(this.bot.getBounds(), this.cespedRectangles[i])) {
67               console.log("cesped");
68       }
69   }
```

Listado 5.7: Código para saber si el agente se encuentra en el césped.

CheckBot()

Esta función lo único que hace es saber si el estado del agente se encuentra en el estado del tesoro. Si fuera así notificaría de que el agente ha encontrado el tesoro. Para que esto funcione se necesita el estado actual del tesoro y del agente, para ello se utilizan las funciónes *getEstadoBot()* y *getEstadoTesoro(),* que se explican ahora.

GetEstadoBot y GetEstadoTesoro()

Estas funciones devuelven la celda del mapa donde se cuentra el agente y el estado, respectivamente. Para ello se usa la función de Phaser *getTileWorldXY(),* cuyos argumentos son las posiciones en pixeles *X* e *Y* del Sprite y su tamaño. En el Listado 5.8 se puede ver el código de la función *getEstadoBot().*

```
101   getEstadoBot: function() {
102       return this.map.getTileWorldXY(this.bot.x, this.bot.y, this.map.tileWidt
h, this.map.tileHeight, this.backgroundlayer);
103   },
```

Listado 5.8: Código de la función *getEstadoBot().*

AjustarPunto(eje)

Esta función se ha implementado para que devuelva una posición aleatoria del mapa. Pero esta posición no es una cualquiera, sino que tiene que ser el centro exacto de una celda del mapa, ya que el agente debe empezar centrado a entrenarse. Por lo tanto, devuelve aleatoriamente uno de los centros de las celdas del mapa.

El argumento eje puede ser *X* o *Y* por lo que el punto devuelto es en el eje correspondiente. Hay que añadir que el eje de coordenadas del mapa empieza arriba a la

izquierda, por lo tanto, las *X* son crecientes hacia la derecha y las *Y* son crecientes hacia abajo.

Esta función simplemente calcula un número aleatorio entre el rango del mapa con la función de Phaser *world.randomX* o *world.randomY*. Con el cálculo del Listado 5.9 se puede observar el cálculo para ajustar el punto al centro de la celda. El cálculo observado está enfocado al eje de las *X*. Los pasos del cálculo se explican a continuación:

1. Si el número aleatorio es el último pixel a lo ancho, únicamente se le resta la mitad del ancho de lacelda.
2. Si el número aleatorio es un número distinto al último píxel a lo ancho, el punto hay que ajustarlo al borde izquierdo de la celda, por lo que al número aleatorio hay que restarle la diferencia de su posición al borde izquierdo.
3. Para ajustar al punto del borde izquierdo al centro de la celda hay que sumarle la mitad de la celda.

```
48   var rx = this.game.world.randomX;
49   if (rx !== this.map.widthInPixels) {
50       var x = (rx - (rx % this.map.tileWidth) + (this.map.tileWidth / 2));
51   }
52   else {
53       var x = this.map.widthInPixels - (this.map.tileWidth / 2);
54   }
55   return x;
```

Listado 5.9: Código para ajustar punto en el eje *X*.

FindObjectsByType(type)

Esta función devuelve la lista de los objetos de un tipo. El argumento *type* tiene que ser exactamente igual al nombre de la propiedad *type* introducida en el mapa Tiled. Como se puede observar en el Listado 5.10, se implementa un bucle con todos los objetos de la capa *objectLayer* para añadirlos a la lista.

```
130   findObjectsByType: function(type, map) {
131       var result = [];
132       map.objects['objectLayer'].forEach(function(element) {
133           if (element.properties.type === type) {
134               result.push(element);
135           }
136       });
137       return result;
138   },
```

Listado 5.10: Código de la función *findObjectsByType()*

GetRectanglesFromObjects(objects)

Este método tiene como objetivo devolver una lista de objetos *Phaser.Rectangle*. Para ello, se le ha pasado la lista de objetos creados en el método *findObjetcByType()*. Este objeto *Phaser.Rectangle* tiene como parámetros la posición exacta del objeto y su tamaño en píxeles, para así formar el rectángulo en el mapa.

Una vez explicadas todas las clases junto con sus métodos, se muestra el resultado de la implementación en Fig. 5.8. Se puede observar el agente representado como el Sprite del robot y el tesoro como el Sprite de la bolita roja.

5.3 Iteracion 2: Implementar en Python el algoritmo QL

En esta iteración se ha desarrollado un prototipo de aprendizaje QL de un agente en un mapa similar al creado en Tiled. Hay que remarcar que esta iteración ha sido la más larga, ya que se ha tardado bastante en conseguir que el agente aprenda de la forma correcta, por ello se han llevado a cabo varias correcciones en el algoritmo QL.

Se ha implementado primero el algoritmo QL en un prototipo en Python porque el desarrollo y las pruebas serían más sencillas que implementarlo directamente en el sistema final.

Figura 5.8: Mapa Tiled

5.3.1 Implementación del mundo en Python

Se ha tenido que crear un mapa similar a los creados en Tiled, por eso se ha usado la librería de Python TkInter. Este entorno permite crear un mapa basado en celdas, por lo que a continuación se explican los aspectos más importantes de la implementación. Esta implementación ha seguido unos pasos muy parecida a la implementación en Phaser, por lo que no se va a detallar demasiado. El mundo se ha creado en una clase llamada *mundo.py* que se ha importado desde la clase donde se ha implementado el algoritmoQL.

Lo primero de todo ha sido crear un objeto TkInter con la función *Tk()*. Este objeto ha servido como argumento para crear el objeto *mundo.* En el Listado 5.11 se puede ver la inicialización de ese objeto. Este objeto es de tipo *Canvas* y tiene como parámetros el objeto Tkinter anteriormente mencionado y el tamaño del mapa en píxeles.

```
8   mundo = Canvas(master, width = x * width, height = y * width)
```

Listado 5.11: Código para la creación del objeto mundo.

Se ha tenido que crear un punto para el agente y uno para el tesoro, por lo que aleatoriamente se toma ese punto en los rangos del mapa. También se han tenido que crear unas variables con la recompensa de cada elemento del mundo y la recompensa del tesoro. Además, ha sido necesario crear una lista de estados, donde cada estado es una celda del mapa. Después de eso se ha creado una lista de estados para cada elemento del mundo.

En el Listado 5.12 se puede ver como se han añadido los elementos a cada lista dependiendo de la clave que tengan.

RenderizarMundo()

Una vez creadas estas listas se ha implementado esta función para renderizar continuamente el mapa. Con las listas anteriores, creamos cada celda del mundo a partir de la función de TkInter *create_rectangle()*. Los argumentos de esta función son las coordenadas de la esquina izquierda superior y la esquina derecha inferior. También

```
68   for estado in estados:
69     if keyEstados1[c] == "c":
70         estadosCesped.append(estado)
71     elif keyEstados1[c] == "a":
72         estadosAgua.append(estado)
73     elif keyEstados1[c] == "m":
74         estadosMontana.append(estado)
75     elif keyEstados1[c] == "h":
76         estadosHielo.append(estado)
77     elif keyEstados1[c] == "i":
78         estadosIsla.append(estado)
79     elif keyEstados1[c] == "mh":
80         estadosMontanaHielo.append(estado)
```

Listado 5.12: Código para inicializar las listas de los tipos de elementos.

tendría como argumentos el color de relleno de la celda, entre otros. Tendría la forma *create_rectangle(x1, y1, x2, y2, **kwargs)*. Entonces, recorremos con bucles *for* todas las listas creando Sprites de cada tipo. También se crea el Sprite del agente en esta función, para ello se utilizará la coordenada del agente.

Moverse(dx,dy)

Esta función básicamente mueve un sprite a una nueva coordenada. Sirve para mover el Sprite del agente dependiendo de la acción que realice, y devuelve la recompensa dependiendo del nuevo estado donde acabe el agente a partir de ese movimiento. Por lo que dependiendo de si ese estado es un tipo de entorno u otro, se devuelve la recompensa asignada para cada tipo de entorno o la recompensa del tesoro, si lo encuentra.

Inicialmente se establece la nueva coordenada del agente sumando los argumentos *dx* y *dy* a la anterior coordenada del agente. Estos argumentos dependerán del movimiento de la acción que se quiera tomar en el algoritmo QL, que ya se verán cuales son.

A continuación, se utiliza la función de TkInter *coords()* para mover el Sprite del agente, donde sus argumentos son, el objeto del Sprite del agente, y las coordenadas de la esquina izquierda superior y la esquina derecha inferior. Mejor visto, la función sería *coords(spriteBot, x1, y1, x2, y2).* Hay que añadir que el origen del eje de coordenadas es la esquina superior izquierda del mapa, como en Phaser.

Una vez hecho esto, a partir de condiciones *if-else* se devuelve la recompensa correspondiente de la nueva coordenada del agente.

ReiniciarJuego()

En cada episodio del algoritmo QL se tiene que reiniciar el entrenamiento. Por lo que esta función se encarga de poner el agente en el estado inicial y poner el sumatorio de resultados al valor inicial.

HaReiniciado()

Este método simplemente devuelve una variable de tipo *Boolean* para saber si el agente ha encontrado el tesoro o no.

Con esta clase se ha conseguido que el mapa Tiled de Fig. 5.8 sea creado con la librería TkInter. En Fig. 5.9 se puede ver el mapa, donde el Sprite amarillo pequeño es el agente y el Sprite amarillo grande es el tesoro.

5.3.2 Implementación del algoritmo QL en Python

Esta ha sido la parte del desarrollo del proyecto que más esfuerzo ha conllevado, ya que, como se explica anteriormente, se ha tenido que corregir el algoritmo implementado hasta que el agente tuviera un aprendizaje adecuado.

Lo primero que se ha tenido que hacer para realizar la implementación del algoritmo QL, ha sido pensar como llevar la teoría del algoritmo a la práctica. A continuación se explica paso por paso como se ha interpretado la teoría.

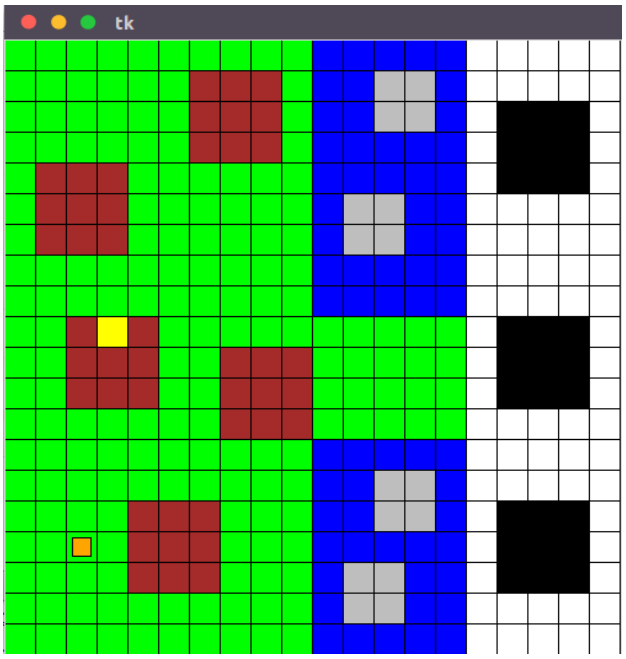

Figura 5.9: Mapa creado con la librería TkInter.

Estados

Los estados posibles donde puede ir el agente son cada una de las celdas que forman el mapa.

Acciones

Las acciones posibles que puede realizar el agente son cuatro: arriba, abajo, izquierda y derecha.

Función acción-valor

La función *acción-valor* que presenta el valor de realizar una acción a en el estado s es representada por una matriz-Q. Esta matriz se basa en una tabla, cuyas filas son los estados del mapa y las columnas son las acciones que puede realizar el agente.

En Fig. 5.10 se puede observar un ejemplo de matriz-Q de este algoritmo.

Velocidad de aprendizaje

La velocidad de aprendizaje α viene definida por el valor dinámico $\alpha = 60/(59 + N(s,a))$, donde $N(s,a)$ es el número de veces que se ha hecho una determinada acción en un estado. En el programa es una matriz donde las filas son los estados y las acciones son las columnas, por lo que cada elemento de la matriz tendrá el número de veces que se realizó la acción en ese estado. Esto también ayuda a la exploración, ya que al principio el agente prácticamente solo se acuerda de la nueva experiencia, y cada vez va aprendiendo menos con el tiempo.

Factor descuento

El factor de descuento es un valor que será introducido por el usuario por lo que no es necesario especificarlo.

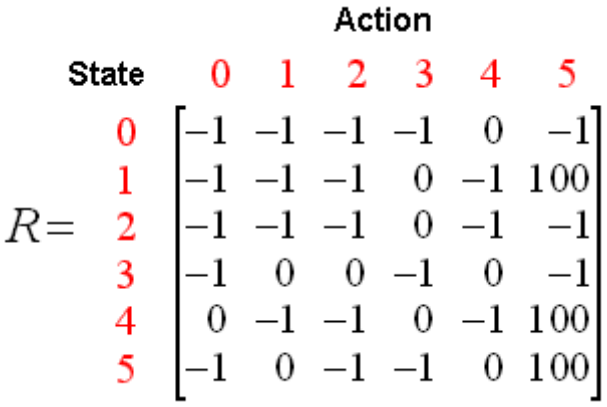

Figura 5.10: Ejemplo de matriz-Q

Elección de la acción

La elección de la acción futura ha sido el mayor problema, ya que posiblemente ha sido la causa del defectuoso aprendizaje que se tenía al principio. Al prinicipio se intentó seguir el proceso de Decisión de Markov para la toma de decisiones. La idea era que cada acción en un estado tuviera una probabilidad directamente proporcional a su valor en la matriz-Q, y la suma de esas probabilidades sumara 1. Por lo que las acciones con mayor valor, tienen más probabilidades de llevarse a cabo.

Este modelo es genial para incentivar la exploración al principio. Pero por desgracia, durante su implementación en la práctica no salian los resultados esperados ya que el cambio de los valores de la matriz-Q no era el apropiado y por lo tanto, las acciones tampoco eran correctas y el aprendizaje era defectuoso.

Entoces, se cambió la forma de tomar las acciones a simplemente tomar siempre la acción que tuviera el mayor valor en ese estado. De ese modo, siempre se tomaba la mejor acción, es decir, siempre se llevaba una estrategia voraz. Sin embargo, esta voracidad se compensó con la exploración incorporando recompensas negativas, menos la del tesoro que era positiva. Esto hacía que al principio siempre se exploraran los sitios por donde el agente no había pasado, ya que debido a las recompensas negativas los sitios más transitados deberían tener menos valor que los sitios más transitados.

El uso de recompensas negativas, hace que se puedan llamar penalizaciones en vez de recompensas. Se vió desde una perspectiva en la que cada acción llevaba una penalización, a no ser que encontrase el tesoro. Entonces esa penalización puede ser

interpretada como la energia que pierde el agente con cada acción, por lo tanto, el agente al final buscará la política óptima que le permita gastar menos energia para llegar al tesoro.

El agente con esta estrategia hace un aprendizaje que le debería permitir conocer el camino más óptimo desde cualquier punto del mapa hacia el tesoro. Por eso en cada episodio del aprendizaje se muestra la suma de refuerzos que se han recogido.

Algoritmo QL

El algoritmo QL mostrado en el Listado 3.1 es el mismo al usado en el algoritmo de este programa. Se utiliza la ecuación (1.1) para actualizar los valores de la matriz-Q.

Más adelante se hará la explicación detallada del algoritmo QL con el código en Python.

La implementación del algoritmo QL se ha realizado en una clase llamada *RL.py*. Ahora se va a explicar detalladamente lo implementado en esta clase y su relación con los puntos anteriores.

Inicialización de variables y tablas

Como se muestra en el Listado 5.13, hay que inicializar las tablas, las aciones y los estados del agente y el tesoro. Las acciones, y los estados son referenciados de la clase *mundo.py* ya que ahí es donde se crea el mapa y se implementa el movimiento del agente. También se inicializan la matriz-Q y la tabla N como diccionarios para facilitar el acceso a sus valores.

```
6   discount = 0.75
7   acciones = Mundo.acciones
8   estados = Mundo.estados
9   final = Mundo.final
10  Q = {}
11  N = {}
```

Listado 5.13: Código de inicialización de variables para el algoritmo QL.

PosiblesAcciones(s)

Este método tiene como argumento un estado *s* del mapa. Por lo tanto, tiene como objetivo devolver las posibles acciones del agente en ese estado. Por ejemplo, si el estado es la última celda por la derecha, las acciones posibles serán todas menos la derecha.

InicializarTablas()

En este método se inicializan la matriz-Q y la tabla N. Como se puede observar en el Listado 5.14, se recorren todos los estados del mapa y se usa la función *posiblesAcciones()* para saber las acciones posibles en ese estado. Entonces, si la acción es posible se inicializa a 0, de otra manera , para especificar que es una acción imposible, se inicializa a -999. En cuanto a la Tabla N, todos los valores se inicializan a 0.

```
40    def inicializarTablas():
41        for estado in estados:
42            temp = {}
43            tempA = {}
44            posAcc = posiblesAcciones(estado)
45            for acc in acciones:
46                tempA[acc] = 0
47                if acc in posAcc:
48                    temp[acc] = 0.1
49                else:
50                    temp[acc] = -999
51
52            Q[estado] = temp
53            N[estado] = tempA
```

Listado 5.14: Código de la función *inicializarTablas()*.

```
55     def moverse(accion):
56         if accion == acciones[0]:
57             r = Mundo.moverse(0, -1)
58         elif accion == acciones[1]:
59             r = Mundo.moverse(0, 1)
60         elif accion == acciones[2]:
61             r = Mundo.moverse(-1, 0)
62         elif accion == acciones[3]:
63             r = Mundo.moverse(1, 0)
64         return r , s2
```

Listado 5.15: Código de la función *moverse()*.

Moverse(accion)

En este método, básicamente se implementa un paso o experiencia del agente. Dependiendo del argumento *accion*, el agente se moverá de un estado a otro, por lo tanto, este movimiento conlleva la obtención de la recompensa y el estado donde se ha movido. El agente solo podrá moverse a sus estados adyacentes en un paso o experiencia. En el

listado 5.15 se puede observar como se obtiene la recompensa simplemente comparando con condiciones *if-else*.

MaxQ(s)

Esta es la fucción utilizada para tomar la acción con mayor valor en cada estado. En el listado 5.16 se puede observar como funciona. Simplemente se toma el valor mayor de la fila de la matriz-Q correspondiente al estado *s* y se devuelve conjuntamente con la acción *a* que produce ese valor.

```
66    def maxQ(s):
67        val = None
68        acc = None
69        for a, q in Q[s].items():
70            if val is None or (q > val):
71                val = q
72                acc = a
74        return acc,val
```

Listado 5.16 : Código de la función *maxQ()*.

IncQ(s, a, alpha, inc)

Esta función implementa exactamente la ecuación (1.1) para actualizar los valores de la matriz-Q. Los argumentos introducidos son el estado actual *s*, la acción elegida *a*, la velocidad de aprendizaje *alpha* y el incremento.

El incremento es la suma de la recompensa *r* con el valor máximo que toma el agente en el estado próximo s' con la acción próxima a' que consigue ese valor, reducido por el factor descuento γ. Esta reducción con γ es debida a que, como se explica anteriormente en el MDP, los valores de las acciones futuras tienen menos importancia que los valores de las acciones actuales. En la ecuación (5.1) se puede ver el incremento como parte de la ecuación (1.1). En listado 5.17 se puede observar la ecuación implementada.

$$R(s) + \gamma max Q_{a'}\,(a', s') \tag{5.1}$$

```
75    def incQ(s, a, alpha, inc):
76        Q[s][a] = Q[s][a] + alpha*(inc - Q[s][a])
```

Listado 5.17: Código de la función *incQ()*.

Run()

Este es el método principal, donde se implementa el algoritmo QL utilizando los métodos y variables anteriores. En este prototipo de python el agente empieza siempre en el mismo punto al empezar cada episodio. Cada episodio empieza cuando en el anterior se ha encontrado el tesoro.

El entrenamiento dura hasta que la suma de las recompensas o los resultados de cada episodio son iguales durante 10 episodios, esto significa que el agente ya ha aprendido lo suficiente. En el Listado 5.18 se puede ver el código del algoritmo QL.

Cuando se llegan a las 10 repeticiones seguidas de resultados, el tiempo de ejecución entre paso y paso es un poco mayor para poder ver bien como el agente ha aprendido, ya que durante el entrenamiento los movimientos son dificiles de apreciar debido al mínimo tiempo de ejecución entre cada paso. En el entrenamiento el tiempo es necesariamente mínimo, ya que si fuera mayor el agente tardaría más en aprender. También se podría haber puesto como límite de entrenamiento el número de episodios, ya que cuanto más episodios se hayan llevado a cabo por el agente, mayor habrá sido su aprendizaje.

En relación con el algoritmo QL del Listado 3.1, se va a explicar detalladamente la implementación de este algoritmo QL donde cada iteración del bucle *while* es una experiencia nueva para el agente. Por lo tanto, los pasos de cada iteración son los siguientes:

1. Se crea una variable *s* que especifica el estado actual del agente.
2. Se selecciona la acción que tiene que tomar el agente para conseguir el mayor valor en el estado actual. Eso se ha hecho con la función *maxQ().*
3. Se utiliza esa acción para mover al agente al nuevo estado con la función *moverse().* Con ese movimiento se recoge la recompensa *r* y el estado siguiente *s2.*
4. A continuación se incrementa el valor correspondiente en la tabla N.
5. Con la tabla N actualizada, se usa para inicializar la variable *alpha* para la velocidad aprendizaje con la ecuación explicada anteriormente.
6. Con la función *maxQ() s*e recoge el mayor valor posible a tomar en el estado siguiente *s2.*

7. Se aplica la ecuación (1.1) con la función *incQ()*. Para ello el parámetro incremento es la ecuación (5.1) con la recompensa *r*, el factor descuento γ y el mayor valor tomado en el paso 6.
8. Se controla si el agente ha encontrado el tesoro, de ser así se reestablecerían los valores del episodio con la función *Mundo.reiniciarJuego()*.
9. Si se han repetido los resultados de los últimos 10 episodios, se descarga la matriz Q y se disminuye el tiempo de ejecución entre cada experiencia con la función *time.sleep()*.

Una vez creado correctamente el algoritmo QL en Python, se muestran varios experimentos de aprendizajes. En la figura 5.11, se pueden observar dos gráficas de un aprendizaje cada una donde se muestra el resultado por episodio. Se puede observar que en los dos experimentos los puntos de la gráfica cada vez son más estables formando una linea recta de puntos paralela al eje *X* en un único resultado, es decir, convergen en una política óptima.

```
101   def run():
102       global discount
103       time.sleep(1)
104       inicializarTablas()
105       t = 1
106       pruebas = 0
107       sleep = 0.005
108       scores = []
109       while True:
110           s = Mundo.bot
111           accion, _ = maxQ(s)
112           r, s2 = moverse(accion)
113           N[s][accion] += 1
114           alpha = float(60)/float(59 + N[s][accion])
115           _, maxVal = maxQ(s2)
115           incQ(s, accion, alpha, r + discount * maxVal)
117           t += 1.0
118           if Mundo.haReiniciado():
119               scores.append(Mundo.score)
120               Mundo.reiniciarJuego()
121               time.sleep(0.01)
122               t = 1.0
123               pruebas += 1
124           if Mundo.repeticiones == 10 and t == 1.0:
125               imprimirQmatrix()
126               sleep=0.1
127           time.sleep(sleep)
```

Listado 5.18: Código de la función *run()*, donde está el algoritmo QL

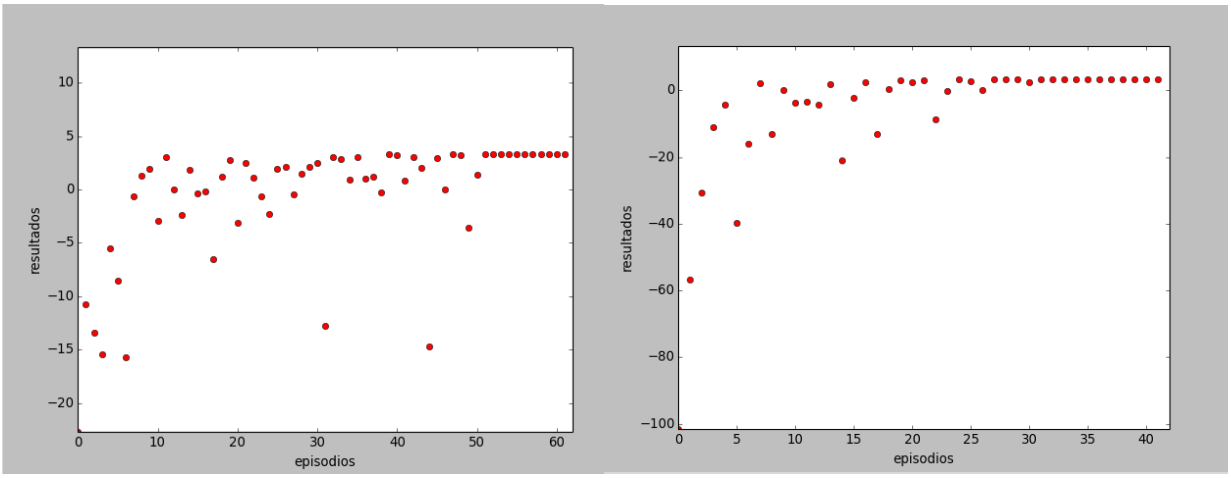

Figura 5.11: Gráficas de dos aprendizajes.

5.4 Iteración 3: Implementación del algoritmo QL junto a la librería Phaser

Esta ha sido la iteración más corta debido a que se ha adaptado el código de la iteración 2 al código de la iteración 1. Esta iteración ha durado una semana, desde que se hizo la reunión de planificación de esta iteración hasta la reunión de planificación de la iteración 4. Esta duración de una semana fue porque se iba un poco justo de tiempo con la entrega, ya que se había perdido mucho tiempo en la iteración 3.

5.4.1 Adaptación del juego Phaser al algoritmo QL

Respecto al código que se habia implementado con Phaser, solo se ha modificado y añadido código en la clase *Game.js*. A continuación se explican los cambios que se han llevado a cabo.

Create()

En la función create se han inicializado las variables necesarias para el algoritmo QL. Estas son las inicializaciones realizadas:

- Se ha creado una lista de acciones con los Strings de *arriba, abajo, izquierda* y *derecha.*
- Se ha creado un diccionario u objeto de JavaScript para asociar las acciones a los respectivos incrementos en la posición del agente. En el Listado 5.19 se puede ver el código.

- Se ha creado una lista de estados con la función *inicializarEstados()*.
- Se ha inicializado la matriz-Q y la tabla N exactamente como si hizo con el código en Python. Son dos diccionarios y se inicializan con la función *inicializarTablas()*.
- Se han creado variables contadoras del número de episodios y de los resultados de cada episodio. También se ha creado una variable para contar las iteraciones o pasos de cada episodio. Por último, se han creado dos variables para especificar el número máximo de episodios que tendrá el entrenamiento y el número máximo de iteraciones por episodio.
- Se han creado nuevas variables para calcular ΔQ. Dependiendo de ΔQse sabe si se debe parar el entrenamiento. Ya se explicará más adelante como funciona y por qué ya no se usan las repeticiones de resultados por episodio para parar el entrenamiento. Estas nuevas variables creadas son una lista con las mejores valoraciones de cada estado, un lista igual pero del paso o experiencia siguiente y el valor del ΔQ mínimo.

IncQ(s,a,alpha,inc)

Esta función es exactamente igual a la implementada en Python.

```
138   this.inc = {
139       "arriba": [0, -1],
140       "abajo": [0, 1],
141       "izquierda": [-1, 0],
142       "derecha": [1, 0]
143   };
```

Listado 5.19: incrementos para las posiciones del agente.

MaxQ(s)

Esta función tiene la misma funcionalidad que la implementada en Python pero es un poco diferente ya que JavaScript no tiene las mismas funciones para recorrer diccionarios que Python.

GetPosiblesAcciones(s)

Es la misma función a la implementada en Python. Es usada en la función *inicializarTablas()* para la matriz-Q y la tabla N.

Moverse(accion)

Este método no es exactamente igual al implementado en Python ya que no devuelve la recompensa de la acción ni el siguiente estado. Únicamente se desplaza el Sprite del agente a la nueva posición. Esto se hace con la suma de la posición actual del agente al incremento asociado a la acción multiplicado por el ancho o largo de la celda, dependiendo si es movimiento en el eje de las *X* o de las *Y*. En el Listado 5.20 se puede ver el código de este método.

```
384   moverse: function(accion) {
385       this.spriteBot.body.x += this.inc[accion][0] * this.map.tileWidth;
386       this.spriteBot.body.y += this.inc[accion][1] * this.map.tileHeight;
387   },
```

Listado 5.20: Código de la función *moverse()*.

InicializarEstados()

En este método simplemente se devuelve una lista con los estados del mapa. Para ello, simplemente se recorre con dos bucles *for* la matriz de celdas del mapa y se añaden las posiciones de cada celda a la lista *estados*, que se devuelve posteriormente.

InicializarTablas()

Este método es exactamente igual al creado en el código de Python. Se inicializa la matriz-Q y la tabla N.

MejorValoración()

Como se menciona anteriormente, el entrenamiento del agente en esta implementación termina dependiendo de ΔQ. Este valor se calcula en cada paso o experiencia del agente. Entonces para calcular este valor, hay que realizar unos cálculos en diferentes métodos. Este método es uno de ellos y se calcula la mejor valoración de cada estado, por lo tanto, se utiliza la función *maxQ()*. En el listado 5.21 se muestra el código de este método.

```
318   mejorValoracion: function() {
319       var mejorValEstados = {};
320       for (var i = 0; i < this.estados.length; i++) {
321           var valAcc = this.maxQ(this.estados[i]);
322           mejorValEstados[this.estados[i]] = valAcc[1];
323       }
324       return mejorValEstados;
325   },
```

Listado 5.21: Código de la función *mejorValoración()*

DifValoraciones()

En este método se termina el cálculo empezado por la función *mejorValoración()* para obtener el valor de ΔQ. La lista de pasos para calcular este valor es la siguiente:

1. Crear una lista *diferencial* que tendrá las diferencias entre las listas de las mejores valoraciones para cada estado. Estas listas son las creadas en el método *create()*.
2. Se recorre la lista de estados del mapa.
3. Para cada estado se crea la diferencia de sus mejores valoraciones en el paso actual y en el próximo. Para ello se utilizan las listas mencionadas tomando el valor absoluto de sus diferencias en ese estado.
4. Se añade esa diferencia a la lista *diferencias*.
5. Se toma el valor máximo de la lista *diferencias* y se devuelve como valor de ΔQ. El valor es el máximo porque es la diferencia más característica, que servirá para compararla con ΔQ mínimo.

En el Listado 5.22 se muestra el código de esta función. Para entender mejor este proceso, se puede observar Fig. 5.12, donde se muestra gráficamente el proceso para el cálculo de ΔQ.

```
311   difValoraciones: function() {
312       var diferencial = [];
313       for (var i = 0; i < this.estados.length; i++) {
314           diferencial.push(Math.abs(this.mejorValEstados[this.estados[i]] - this.mejor
ValEstadosProx[this.estados[i]]));
315       }
316       return Math.max.apply(null, diferencial);
317   },
```

Listado 5.22: Código de la función *difValoraciones()*.

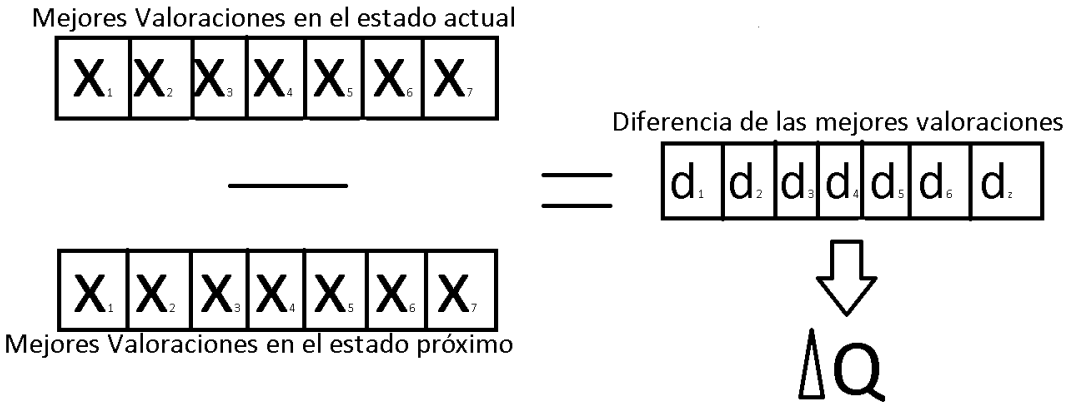

Figura 5.12: Esquema del cálculo de ΔQ.

Update()

Este es el metodo principal del algoritmo QL donde de aplican los métodos anteriores. El método *update()* es ejecutado por Phaser continuamente, es decir, cada fotograma que es renderizado equivale a una llamada al método. Por ejemplo, si la velocidad de renderizado es de 60 Fotogramas por segundo (FPS), pues el método se llamará 60 veces en un segundo.

Cada vez que se llama al método, se realiza un paso o experiencia del agente. Por lo tanto, si comparamos la función *update()* con la función *run()* de Fig. 5.17, cada iteración del bucle *while* equivale a una llamada al método *update()*.

Con esto sabido, se presupone que la velocidad de cálculo o aprendizaje depende de los FPS del juego, por lo que no se ha podido sobrecargar mucho la complejidad del mundo. Se ha podido eliminar el movimiento del agente, pero para visualizar el ApR se ha visto necesario mostrarlo.

En el Listado 5.23 se puede ver el código de la función. Las diferencias respecto a la función *run()* del código en Python son las siguientes:

- Se tiene una variable booleana *parar* para saber cuando parar de entrenar al agente.
- Cada vez que se empieza un nuevo episodio, el agente empieza en un estado aleatorio, por lo tanto, no se empieza siempre desde el mismo estado. Esto se hace para que se pueda aprender la política óptima desde todos los estados del mapa,

entonces al final del entrenamiento el agente debería conocer el camino óptimo al tesoro desde cualquier estado. También se hace para mejorar la exploración del mapa, ya que de la otra forma hay partes del mapa que son poco explorados. El único problema de esto es que el aprendizaje tardará más en llevarse a cabo, pero el resultado será mejor.

- Debido a que el agente empieza en estados distintos en cada episodio, la condición para acabar el entrenamiento es distinta ya que no se pueden dar repeticiones en los resultados por episodio. Esto es debido a que el camino óptimo del agente en cada episodio es siempre distinto. Por ello se utiliza el valor ΔQ. Entonces para que el entrenamiento acabe, el valor de ΔQ debe de ser menor que el ΔQ mínimo un número de iteraciones consecutivas. En este ejemplo, el número de veces consecutivas es 10. Si nunca se llega a este número de veces consecutivas, el entrenamiento para en un número máximo de episodios. Además, para que se contabilice el valor de ΔQ, se tienen que haber superado un mínimo de pasos, ya que al principio del entrenamiento el valor de ΔQ suele ser 0. ΔQ al principio suele ser 0 porque como las recompensas son negativas, las acciones sin intentar serán las que mejor valoración tengan. Por lo tanto, el valor de ΔQ se estabiliza cuando se han intentado la mayoría de las acciones un número considerable de veces.
- Si no se encuentra el tesoso antes del número máximo de iteraciones por episodio, se empieza un nuevo episodio.

Una vez explicada la implementación del algoritmo QL en el sistema junto a sus respectivas diferencias a la implementación en Python, se muestran dos experimentos. En Fig. 5.13 se muestran estos dos experimentos.

Se puede observar que ahora también convergen en una política óptima. Sin embargo, la convergencia ya no es una línea recta paralela al eje *X* en un único resultado como en Fig. 5.11 debido a que el agente empieza en cada episodio en un estado distinto. Esto hace que la convergencia cueste más y se den mayores picos en la gráfica. Por ejemplo, el agente podría empezar en una zona del mapa con refuerzos muy malos, produciendo resultados bastante malos.

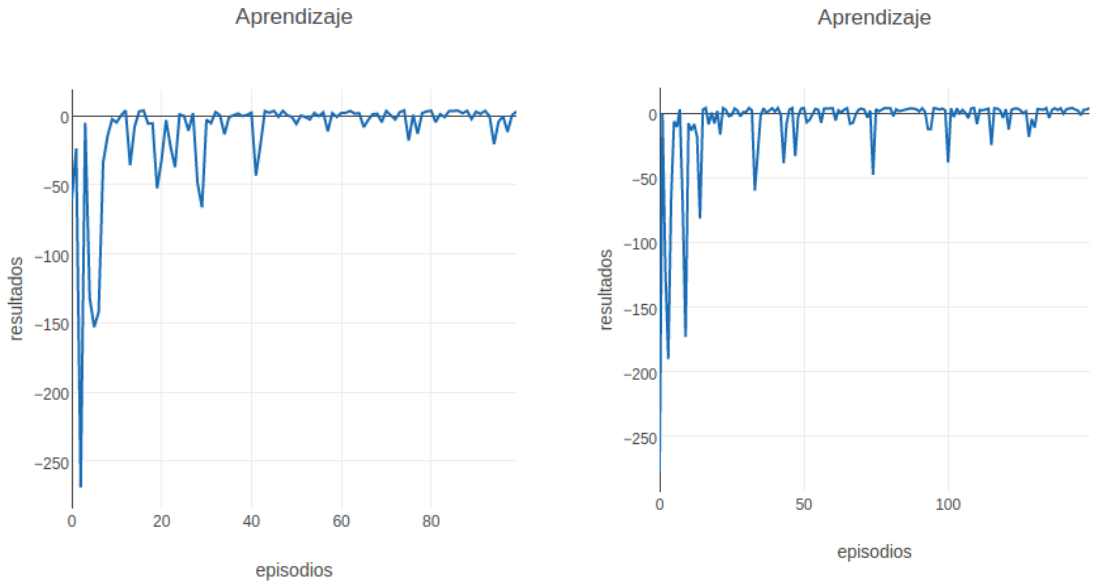

Figura 5.11: Gráficas de dos aprendizajes

5.5 Iteración 4: Implementación del servidor REST y las interfaces gráficas

En esta iteración se comienza la implementación del servidor REST, así como algunas de las interfaces gráficas de la aplicación. El desarrollo de la API REST no se realiza en su totalidad porque al no haberse implementado la base de datos todavía, no tenía sentido implementar todo a su totalidad. Además, la base de datos es con la librería Flask-SQLAlchemy, por lo tanto, la base de datos y el servidor REST son muy dependientes entre sí.

En cuanto a las interfaces gráficas, se utiliza el mismo archivo CSS para todos los archivos HTML. También se utiliza la librería de Bootstrap para ayudar en la interfaz gráfica y la librería de JQuery para facilitar la implementación de la web dinámicamente.

5.5 Iteración 4: Implementación del servidor REST y las interfaces gráficas

En esta iteración se comienza la implementación del servidor REST, así como algunas de las interfaces gráficas de la aplicación. El desarrollo de la API REST no se realiza en su totalidad porque al no haberse implementado la base de datos todavía, no tenía sentido implementar todo a su totalidad. Además, la base de datos es con la librería Flask-SQLAlchemy, por lo tanto, la base de datos y el servidor REST son muy dependientes entre sí.

```
218   if (!parar) {
219       if (!this.done && this.iteraciones < iteracionesMaximas) {
220           var estado = this.getEstadoBot();
221           var accVal = this.maxQ(estado);
222           var accion = accVal[0];
223           this.moverse(accion);
224           this.checkBot();
225           var refuerzo = this.getRefuerzoBot();
226           this.N[estado][accion] += 1;
227           var alpha = 60 / (59 + this.N[estado][accion]);
228           var proximoEstado = this.getEstadoBot();
229           var accValProx = this.maxQ(proximoEstado);
230           var inc = refuerzo + discountFactor * accValProx[1];
231           this.mejorValEstados = this.mejorValoracion();
232           this.incQ(estado, accion, alpha, inc);
233           this.mejorValEstadosProx = this.mejorValoracion();
234           this.difQ = this.difValoraciones();
235           this.iteraciones += 1;
236       } else {
237           this.score = 1;
238           this.iteraciones = 0;
239           this.pasos++;
240           this.spriteBot.x = this.ajustarPunto("x");
241           this.spriteBot.y = this.ajustarPunto("y");
242           this.done = false;
243       }
244       if (this.difQ < this.difQmin && this.pasos > 30) {
245           this.repeticiones++;
246       } else {
247           this.repeticiones = 0;
248       }
249       if (this.repeticiones === repDifQ || this.pasos === maxPasos) {
250
251           parar = true;
252       }
253   }
```

Listado 5.23: Código de la función *update()*

En cuanto a las interfaces gráficas, se utiliza el mismo archivo CSS para todos los archivos HTML. También se utiliza la librería de Bootstrap para ayudar en la interfaz gráfica y la librería de JQuery para facilitar la implementación de la web dinámicamente.

5.5.1 API REST

Al haber acabado la implementación del algoritmo QL en nuestra aplicación con Phaser, se implementa un servidor de tipo Transferencia de Estado Representacional (REST). Con este servidor se aceptan peticiones HTTP, con las que el usuario solicita la aplicación del algoritmo QL sobre el mapa que éste haya introducido.

Esta implementación se lleva a cabo con Flask, por lo tanto, la estructura de directorios de la aplicación es algo esencial para el correcto funcionamiento. La estructura es la siguiente:

- /static
    - /js
    - /css
    - /assets
- /templates
- App.py

El directorio *static* está formado a su vez por tres directorios. El directorio *js* contiene los archivos de JavaScript, el directorio *css* contiene los archivos CSS y el directorio *assets* contiene las imágenes del juego Phaser.

Los directorio *static* y *templates* deben llamarse así para que Flask puede acceder a ellos mediante la función *url_for()*, que se verá más adelante. En este caso solo se van a incluir en *templates* las archivos HTML de la ventana para Iniciar Sesión y de la ventanta para el entrenamiento y visualización del agente.

Una vez explicada la estructura de la API, se explica a continuación la implementación del archivo *app.py*, ya que es donde se implementa el servidor REST.

Lo primero ha sido importar la librería Flask y crear una instancia de Flask, como se muestra en el Listado 5.24. El argumento *__name__* del objeto Flask es el nombre del módulo de la aplicación.

```
9    from flask import Flask
10   app = Flask(__name__)
```

Listado 5.24: Crear la instancia de Flask.

Después, hay que saber arrancar el servidor. Para ello se utiliza el código del Listado 5.25.

```
178    if __name__=="__main__":
179         app.secret_key = os.urandom(12)
180         app.run(debug=True)
```

Listado 5.25: Código para arrancar el servidor.

Como se puede apreciar en la línea 179, se ha creado una clave secreta aleatoria. Esto se hace para que no se puedan editar las cookies de los usuarios.

Ahora se procede a implementar la parte más importante de la aplicación ya que representa la forma en la que se trabaja en esta librería. Se usa el decorador *route()* para decirle a Flask qué URL va a llamar a esta función. Como primer argumento, se indica la URL que va a llamar a esa función.

En el Listado 5.26 se puede ver la implementación para la ventana de Iniciar sesión. En el decorador *route()* la URL es */login* y el segundo argumento es la lista de métodos HTTP que acepta la función. En este caso, *login* acepta *GET* y *POST*. *GET* es para mostrar la ventana del *login* al usuario y *POST* es para recibir el formulario que el usuario rellene cuando inicie sesión.

Para saber cual de los dos métodos se aplica a la función, se utiliza la función de Flask *request.method*. En este caso, si se trata de *GET*, se devuelve el HTML del *login* con la función de Flask *render_template(),* cuyo argumento es el nombre del archivo HTML que se encuentra en el directorio *templates*.

Si es el caso de *POST*, se toman los datos del formulario con la función de Flask *request.form.* Así, se puede ver de la linea 15 a la 18 como se recogen los datos del usuario y se comparan para ver si son correctos. Si se introducen unos datos correctos, se accederá a la cuenta del usuario con la función de Flask *redirect().* Esta función redirecciona a la URL indicada por la función de Flask *url_for(),* donde sus argumentos son el nombre de la función asociada a un decorador *route()* y los argumentos que se pasan como parámetros a ese decorador. Si los datos son incorrectos se manda un mensaje de error con la función *flash()* y se devuelve de nuevo el HTML del *login.*

Después se ha implementado la función para el decorador *route()* de la cuenta de los usuarios. Sin embargo, como la implementación de la API REST hasta entonces ha sido simple, únicamente se muestra la ventana del entrenamiento cuando el usuario entre en su cuenta. En el Listado 5.27 se muestra la implementación para la ventana del entrenamiento.

La URL creada tiene asociado el nombre del usuario como se puede ver en la línea 27. Se usa */home/<email>* indicando que *email* es el argumento recibido por la llamada

a la función asociada al decorador *route()*. De esta manera se crea una URL única para cada usuario. Posteriormente se devuelve el HTML del entrenamiento.

Una vez hecho esto, no se implementa nada más en la API REST hasta la creación de la base de datos en la siguiente iteración.

```
12   @app.route('/login', methods=['GET','POST'])
13   def login():
14       if request.method =='POST':
15           POST_USUARIO = str(request.form['email'])
16           nombre = POST_USUARIO.split('@')[0]
17           POST_PWD = str(request.form['password'])
18           if POST_USUARIO == 'admin@gmail.com' and POST_PWD == 'admin':
19               return redirect(url_for('home', email = nombre))
20           else:
21               flash('CAMPOS INCORRECTOS')
22               return render_template('login.html')
23       else:
24           return render_template('login.html')
```

Listado 5.26: Código para la ventana *login*.

```
27   @app.route("/home/<email>")
28   def home(email):
29         return render_template('index.html', name = email)
```

Listado 5.27: Código para la ventana del entrenamiento.

5.5.2 Implementación de la interfaz gráfica para iniciar sesión

La implementación de esta interfaz gráfica se realiza en el archivo *login.html*. Se usa la librería de Bootstrap. Esta librería es importada como se puede ver en el Listado 5.28. Se puede observar que se utiliza la función *url_for()* de Flask entre doble corchete. Para que HTML reconozca funciones de Flask se utiliza el Framework Jinja2, que viene incluido en la librería de Flask. Por lo tanto, se usa el doble corchete para indicar que es una función de Flask. Todas las importaciones se tienen que hacer con esta función.

```
8 < script src = "{{ url_for('static', filename='js/lib/bootstrap.min.js') }}" > < /script>
```

Listado 5.28: Código para importar Bootstrap con Jinja2.

En cuanto a la implementación del *body*, se ha usado Bootstrap para dividir la interfaz en 2 filas. Estas filas contienen lo siguiente:

- **Fila 1:** Contiene el título de la aplicación. Esto se expone con la etiqueta *h1*.

- **Fila 2:** Contiene un formulario para iniciar sesión y un controlador de mensajes de error de Flask. Primero se implementa un básico formulario. Posteriormente se manda el formulario a la URL */login*. Hay que añadir que el botón del formulario es de una clase de Bootstrap. Debajo del formulario, añade un código de Jinja2 para controlar dinámicamente los mensajes enviados por la función *flash()* de Flask. Para ello se usa la función *get_flashed_messages()*.

El resultado de la anterior implementación da lugar a la interfaz gráfica que se puede observar en Fig. 5.12.

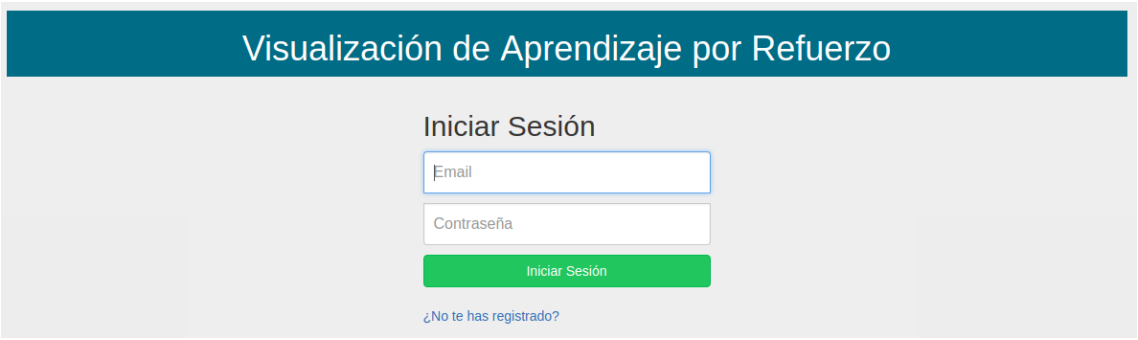

Figura 5.12: Interfaz gráfica del *login*.

5.5.3 Implementación de la interfaz gráfica del entrenamiento.

En esta interfaz se usa la librería Bootstrap y la librería JQuery para la funcionalidad de los controles del entrenamiento. La implementación de esta interfaz gráfica se realiza en el archivo *index.html*.

Se ha reutilizado el código del archivo HTML creado en la iteración 3. Todos los archivos de JavaScript han sido importados, así como el archivo CSS. La interfaz implementada se puede observar en Fig. 5.13.

La implementación del *body* se compone de dos filas. La primera es un simple menú de usuario donde se puede cerrar sesión. La segunda se divide en dos columnas. Cada columna de la segunda fila se explica en los siguientes puntos:

- La primera columna se divide en tres filas. La primera fila contiene una caja de cuatro botones para el control y visualización del entrenamiento. Los botones son *entrenar*, *parar*, *reiniciar* y *mostrar/quitar valoraciones de los estados*. La segunda fila contiene un indicador con el estado del agente y el mapa Phaser. Los

estados son *parado*, *entrenando* y *entrenado.* La última fila contiene los valores de un estado del mapa cuando se deja el cursor sobre ese estado en el mapa.

- La segunda columna se divide en tres filas. La primera fila se divide a su vez en dos columnas. La primera de esas columnas es un formulario donde el usuario tiene que introducir los parámetros para el aprendizaje; y en la segunda se muestran los datos de ese aprendizaje. La segunda fila tiene la lista de tipos de refuerzos del mapa junto a los valores que introduce el usuario. La tercera fila tiene dos botones, uno para poner valores por defecto a los parámetros del aprendizaje, y otro para descargar la matriz-Q en un archivo CSV.

Hay que señalar que los botones del panel de control del entrenamiento son de una clase de Bootstrap. En cuanto al evento de tipo *click* de cada botón del panel de control del entrenamiento, se han creado las siguientes funciones en la clase *Game.js*.

Botón entrenar

Se ha implementado la función *empezarJuego()*. Tiene como objetivo recoger los parámetros para empezar el entrenamiento y avisar al juego Phaser para comenzar a entrenar. Para ello se usa una variable global de tipo *Boolean* llamada *parar*, con el fin de que la función *update()* permita al agente generar experiencias o no. También se cambia el estado del agente a *entrenando.*

Botón parar

Se ha implementado la función *pararJuego()*. En esta función únicamente se cambia el estado del agente a *parado* y la variable *parar* se asigna a *true.* Por lo tanto, el agente no entrena en estos momentos.

Botón reiniciar

Aquí se cambia el estado del agente a *parado* y se empieza el juego Phaser de nuevo, por lo tanto, se llama a la función de Phaser *game.state.start('Game').*

Botón mostrar o quitar valoraciones

Se implementa en la función *círculos().* Es una nueva funcionalidad que se le añade al usuario para que pueda visualizar mejor las valoraciones de cada estado. Entonces cuando se pulsa el botón se muestra, o se deja de mostrar si se estaba mostrando,

un círculo encima de cada estado, cuyo tamaño es directamente proporcional a la máxima valoración de ese estado.

Se crea otra variable global de tipo *Boolean* llamada *mostrar* para controlar la muestra de los círculos en el método *update()*. Para mostrar los círculos, eliminarlos y ajustarlos se utilizan las funciones *crearCírculos()*, *ajustarCírculos()* y *destruirCírculos()*. Estas funciones se explican en los siguientes puntos:

- **CrearCírculos():** Se crea un diccionario *círculos* en el que se asigna a cada estado el Sprite de un círculo.
- **AjustarCírculos():** Posteriormente se escala el tamaño de estos círculos dependiendo del máximo valor de cada estado. Para hacer de forma correcta el escalado, se toma este valor y se normaliza entre 0 y 1. Sin embargo, los valores que sean menores que 0.1, se dejan en 0.1 porque si no el círculo es inapreciable. Entonces, cada Sprite se escala con la función *scale.setTo()* de Phaser.
- **DestruirCírculos():** Se eliminan los Sprites de cada círculo. Para eliminar Sprites en Phaser se usa la función *destroy()*.

Un ejemplo del resultado de pulsar el botón *Mostrar/Quitar valoraciones* se muestra en Fig. 5.14. En este ejemplo se puede apreciar la distribución de las valoraciones en el tamaño de los círculos, siendo mayores los círculos que se encuentran cerca del tesoro.

Figura 5.13: Interfaz gráfica del entrenamiento.

5.6 Iteración 5: Implementación de la base de datos y finalización del servidor REST.

Esta es la última iteración del desarrollo de la aplicación, donde se añaden las interfaces gráficas que faltan, se implementa la base de datos y se acaba la implementación del servidor REST con Flask.

5.6.1 Implementación de la interfaz gráfica del registro

Para la interfaz gráfica del registro se ha creado otro archivo HTML llamado *registro.html*. Se utiliza el archivo CSS y Bootstrap, como en las otras interfaces. Esta interfaz es prácticamente igual a la del *login*, solo se diferencia brevemente en el formulario. El formulario al ser completado es enviado a la URL */registro*. Además, este formulario pide al usuario confirmar su contraseña y que no añada un correo electrónico ya existente.

En Fig. 5.15 se puede observar el resultado de la implementación de la interfaz. Hay que añadir que también se han usado botones predeterminados de Bootstrap.

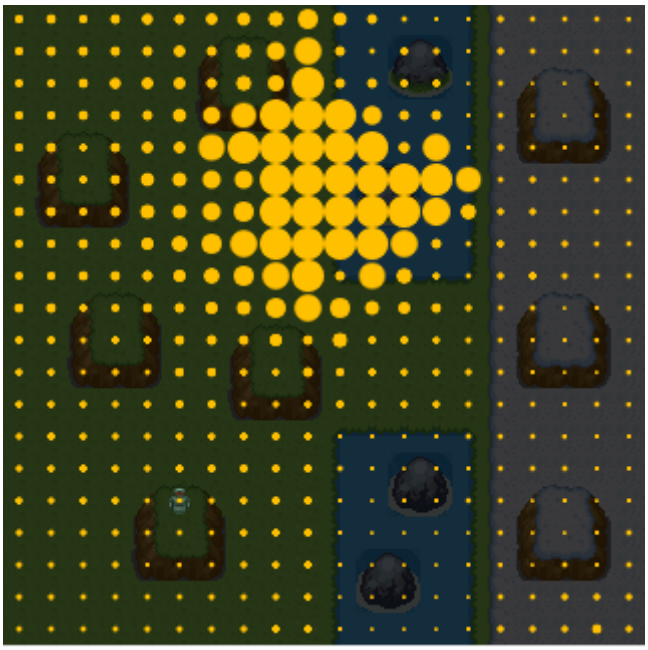

Figura 5.14: Ejemplo de muestra de círculos.

Figura 5.15: Interfaz gráfica del registro.

5.6.2 Implementación de la interfaz gráfica de la carga de mapas

Esta interfaz se crea en la clase *carga.html*. Se utilizan dos filas de Bootstrap. La primera es la misma barra de menú de usuario que la interfaz del entrenamiento. La segunda fila es un complejo formulario explicado en los siguientes puntos:

- El formulario tiene dos formas, dependiendo de si selecciona la opción *abrir mapa nuevo* o *cargar mapa*. Estas dos formas quieren decir que algunos elementos del mapa aparecerán o se esconderán dependiendo de la opción. Para ocultar o enseñar los elementos se ha utilizado el atributo de CSS *style.display*.
- Si la forma es *abrir mapa nuevo*, se le pide al usuario introducir el mapa en formato JSON y un conjunto de patrones. Esta opción es seleccionada cuando el usuario quiere introducir en su cuenta un mapa por primera vez.
- Cuando el usuario quiera seleccionar un mapa que ha introducido con anterioridad, seleccionará la opción *cargar mapa*. Entonces únicamente le aparecerá un listado con los mapas que ya ha cargado en su cuenta, de donde deberá seleccionar uno.

Para ver el resultado de esta implementación, en Fig. 5.16 se muestran las dos formas de esta interfaz. La interfaz de arriba es la opción *abrir nuevo mapa* y la de abajo es la opción *cargar mapa*.

5.6.3 Implementación de la base de datos

Como se explica con anterioridad, la base de datos se crea con la librería SQLAlchemy, que viene incluida en la librería Flask-SQLAlchemy para una mejor compenetración entre ellos.

La finalidad de la base de datos es almacenar la lista de usuarios registrados con sus mapas asociados, permitiendo así que los usuarios puedan registrarse, iniciar sesión, así como cargar y añadir mapas.

Para crear la base de datos correctamente, se han implementado los archivos *models.py* y *database.py*. Estos archivos luego son importados en *app.py* para añadirlo a Flask. En el archivo *models.py* se han creado las dos tablas y la relación entre ellas. Cada tabla es una clase de Python. Las dos tablas se forman de la forma siguiente:

- La tabla *Users* contiene los usuarios con sus respectivas contraseñas. La clave primaria es el ID de cada usuario. También se establece una relación con la tabla *Mapas* para saber los mapas que tiene cada usuario. En el Listado 5.29 se puede ver el código para implementar esta tabla. Hay que especificar el tipo que es cada elemento y en la línea 11 se puede observar como se hace la relación con la función *relationship.*
- La tabla *Mapas* contiene cada mapa que han añadido los usuarios con sus propiedades. Se añade una clave foránea, que es el ID del usuario que añadió ese mapa, estableciendo así la relación. El resto de propiedades o atributos de los mapas son algunos como el archivo JSON, el nombre del archivo, el conjunto de patrones, la lista de refuerzo, etc.

En cuanto al archivo *database.py*, se muestra su código en el listado 5.30. En la línea 5 se especifica que la base de datos es de tipo SQLite y el directorio donde está la base de datos. Posteriormente en la línea 6 se crea una sesión de tipo *scoped* en la base de datos con las funciónes *scoped_session()* y *sessionmaker().* Esta sesión podrá instanciarse en la clase *app.py* para acceder a la base de datos. Por último se crea el método *init_db()*, donde en la línea 11 se importan las tablas creadas en *models.py*. Este método es el llamado por *app.py* para crear una sesión, creando una nueva base de datos si no estaba creada antes.

Figura 5.16: Interfaces gráficas para cargar el mapa.

```
5    class User(Base):
6        __tablename__ = "users"
7
8        id = Column(Integer, primary_key=True)
9        username = Column(String)
10       password = Column(String)
11       maps = relationship("Map")
12
13       def __init__(self, username, password):
14           self.username = username
15           self.password = password
```

Listado 5.29: Código para crear la tabla *Users*.

```
5    engine = create_engine('sqlite:///data/usuarios.db', convert_unicode=True)
6    db_session
=scoped_session(sessionmaker(autocommit=False,autoflush=False,bind=engine))
7    Base = declarative_base()
8    Base.query = db_session.query_property()
9
10   def init_db():
11       import models
12       Base.metadata.create_all(bind=engine)
```

Listado 5.30: Código del archivo *database.py*.

5.6.4 Finalización de la implementación del servidor REST

Una vez implementada la base de datos, hay que saber gestionarla con Flask en el archivo *app.py*, es decir, en el archivo donde se despliega el servidor REST.

Lo primero de todo es llamar al método *init_db()* de la clase *database.py* para crear una sesión de la base de datos. Posteriormente hay que hacer que Flask elimine las sesiones de la base de datos automáticamente, por ejemplo, cuando la aplicación se cierra. Para ello se puede ver el Listado 5.31, donde se crea un decorador *teardown_appcontext*. La función asociada a este decorador eliminará la sesión con la función *db_session.remove()*.

```
172   @app.teardown_appcontext
173   def shutdown_session(exception=None):
174       db_session.remove()
```

Listado 5.31: Código para cerrar sesiones de la base de datos.

Una vez hecho esto se van a explicar los cambios y añadidos que se han hecho a cada uno de los decoradores *route()* para terminar con la implementación del servidor REST.

Login()

En esta función se reutiliza el código del Listado 5.25, añadiendo una consulta a la base de datos para el usuario y contraseña. Para realizar la consulta con SQLAlchemy, se hace de forma muy sencilla con la función *query*. Como lo que se quiere es realizar una instrucción SELECT para el usuario y contraseña, se usa la función *query.filter()*, donde sus argumentos son los atributos para esa consulta.

Posteriormente, se quiere saber si en la base de datos está ese usuario con la contraseña correspondiente, por lo que a la consulta anterior hay que aplicarle la función *first()*. En el Listado 5.32 se puede ver el código de la consulta. También se muestra como acceder a la cuenta del usuario con la función *redirect()* y *url_for()*, poniendo como parámetros el nombre del usuario y su ID.

```
20  query = User.query.filter(User.username==POST_USUARIO, User.password==POST_PWD)
21          result = query.first()
22          if result:
23              return redirect(url_for('home', email = nombre, idUser = result.id))
```

Listado 5.32: Código para hacer la consulta y acceder a la cuenta del usuario.

Registro()

Esta función es nueva y sirve para gestionar el registro de los usuarios. Es muy parecida a la función *login()*. Acepta *GET* y *POST* como métodos HTTP. Si el método es *GET*, simplemente se devuelve el archivo HTML del registro. De la otra forma, cuando es POST, se recibe el formulario del usuario para registrarse y se vuelve a hacer otra consulta para ver si el usuario existe. Si el usuario no existe y se ha verificado la contraseña correctamente, se añade el usuario y contraseña a la base de datos. En el Listado 5.33 se muestra el código para añadir un usuario en SQLAlchemy. En la línea 45 se instancia un objeto de tipo Usuario. Posteriormente, en la línea 46 se añade a la base de datos con la función *db_session.add()*. Y por último, en la línea 47 se confirma el cambio a la base de datos con la función *db_session.commit()*.

```
45          user = User(POST_USUARIO,POST_PWD)
46          db_session.add(user)
47          db_session.commit()
48          return redirect(url_for('login'))
```

Listado 5.33: Código para añadir un usuario a la base de datos.

Home()

Esta es la función principal del servidor REST ya que gestiona la cuenta de los usuarios. Este método controla la carga del mapa Tiled y posteriormente se encarga de mostrar correctamente la interfaz del entrenamiento.

También acepta los métodos HTTP *GET* y *POST*. Cuando es una petición *GET*, se cargan todos los mapas del usuario y se muestra la interfaz de carga. En el Listado 5.34 se muestra ese código. En la línea 60 se hace una consulta para todos los nombres de los

mapas de ese usuario con la función *all()*. Posteriormente se añaden esos mapas a una lista y en la línea 63 se devuelve la interfaz de carga con el nombre del usuario, el ID del usuario y la lista de mapas como argumentos.

```
58  if request.method == 'GET':
59          mapasUsuario = []
60          mapas = Map.query.filter(Map.idUser==id).all()
61          for mapa in mapas:
62              mapasUsuario.append(mapa.nombreMapa)
63          return render_template('carga.html',name=email,idUser =
id,mapasUsuario = mapasUsuario)
```

Listado 5.34: Código para mostrar la interfaz de carga de mapas.

Si es una petición POST, es para recibir el formulario de carga del usuario. Esta parte contiene una gran cantidad de control de errores por si el usuario carga algún mapa Tiled erróneamente.

Como se dijo anteriormente, este formulario tiene las formas de *abrir mapa nuevo* y *cargar mapa.*

En cuanto a *abrir mapa nuevo*, se realiza una gran cantidad de control de excepciones, como si el formato del archivo no es JSON, si ya existe ese mapa en la base de datos, etc.

Si todo es correcto, se añade ese mapa a la base de datos junto con sus propiedades. Las propiedades dependen del mapa que se haya introducido, ya que se leen y guardan automáticamente en este método. Los detalles sobre las propiedades son las siguientes:

- **Archivo JSON:** Este archivo se añade a la base de datos como un String, ya que SQLAlchemy no acepta columnas de archivos JSON en SQLite. Por lo tanto, se almacena como un String.
- **Nombre del Archivo JSON:** Sirve para mostrar al usuario la lista de mapas que tiene cargados.
- **Conjunto de patrones:** Es necesario para la correcta carga del mapa Tiled en Phaser.
- **Nombre de la capa de objetos:** Es necesario saber el nombre para que Phaser pueda cargar la lista de objetos de esta capa.

- **Lista de nombres de las capas de patrones:** Sirven para añadir las distintas capas al Mapa Tiled en Phaser.
- **Lista de refuerzos:** Es necesaria para mostrar en la interfaz de entrenamiento el listado de los refuerzos a los que el usuario asigna los valores.
- **Tamaño del mapa en píxeles:** Esta información es necesaria para renderizar el juego Phaser correctamente respecto a su tamaño, y también para controlar si es demasiado grande.
- **ID del usuario:** También es imprescindible guardar el ID del usuario que carga el mapa para establecer la relación entre los mapas y los usuarios.

Después de añadir el mapa junto a sus propiedades a la base de datos, se toman las URL de las imágenes de los círculos, del tesoro y del agente con la función *url_for()*. Una vez hecho esto, se devuelve la interfaz de entrenamiento, donde los argumentos enviados son el nombre del usuario, el mapa y sus propiedades.

Cuando se trata de *cargar mapa*, se toma del formulario el nombre del mapa que ha seleccionado el usuario y se toma el mapa y sus propiedades de la base de datos.

En estos dos tipos de formularios, cuando se quiere devolver la interfaz gráfica de entrenamiento no se puede enviar como argumentos el archivo JSON y la imagen del conjunto de patrones directamente, ya que Phaser solo acepta una URL para cargarlos. Por ello se crean dos decoradores *route()* más con el fin de tener una URL específica para cada cosa.

En el Listado 5.35 se puede ver el código de estos dos decoradores. En la línea 160 se realiza una consulta para tomar el mapa correspondiente. En la línea 161, como el mapa almacenado en la base de datos es un String, se tiene que convertir a formato JSON con la función *json.loads()*. Y finalmente, para devolver el archivo JSON correctamente se usa la función *jsonify()*. En cuanto a la función *getTileset()*, se consulta el conjunto de patrones correspondiente en la línea 167. Y por último, se devuelve la imagen con la función *send_file()*.

```
157  @app.route("/getMap/<int:id>")
158  def getMap(id):
159      nombreMapa = request.args.get('nombreMapa')
160      mapa =
Map.query.filter(Map.idUser==id,Map.nombreMapa==nombreMapa).first()
161      mapaJson = json.loads(mapa.mapa)
162      return jsonify(mapaJson)
163
164  @app.route("/getTileset/<int:id>")
165  def getTileset(id):
166      nombreMapa = request.args.get('nombreMapa')
167      mapa =
Map.query.filter(Map.idUser==id,Map.nombreMapa==nombreMapa).first()
168      nameFileTileset = mapa.nombreTileSet + ".png"
169      return
send_file(io.BytesIO(mapa.tileSet),attachment_filename=nameFileTileset,mime
type='image/png')
```

Listado 5.35: Código de los decoradores *getMap()* y *getTileset()*.

Argumentos con Jinja2

Una vez desplegado el servidor REST en el archivo *app.py*, hay que tomar los argumentos enviados a cada archivo HTML. Los argumentos se pueden acceder con Jinja2. Por ejemplo, en la barra de menú de usuario hay que mostrar el nombre del usuario, el cual es pasado como argumento. En el Listado 5.36 se muestra como acceder a ese argumento con Jinja2. Se puede ver que el acceso se realiza con el doble corchete al argumento *name*..

```
31  data-toggle="dropdown">Bienvenido, {{name}} <b class="caret"></b>
```

Listado 5.36: Ejemplo de uso de Jinja2 para tomar un argumento.

Sin embargo, Jinja2 no se puede usar desde un archivo de JavaScript externo al HTML, por lo tanto, hay que pasar esos valores como parámetros a través de una función de esos archivos externos. En el Listado 5.37 se muestra cómo se pasan esos valores en el archivo *index.html*.

```
14    <script type="text/javascript">
15        initURL("{{ urlCirculo }}","{{ urlTesoro }}","{{ urlBot
}}","{{urlMapa}}","{{urlTileset}}");
16        initGame("{{ nombreTileSet }}","{{ capaObjetos
}}","{{capasBackground}}","{{x}}","{{y}}");
17    </script>
```

Listado 5.37: Código para pasar los argumentos con Jinja2 a archivos JavaScript.

Por último, hay que explicar algunos cambios importantes que se han hecho en la clase *Game.js*. Estos cambios son debidos a la obligada generalización del código para el entrenamiento de cualquier mapa introducido por el usuario. El cambio principal ha sido la obligada automatización para recoger los tipos de refuerzos y sus valores, para ello se han implementado las siguientes funciones:

- **GetControlFormulario():** Esta función carga los parámetros del entrenamiento que el usuario ha introducido almacenándolos en una variable global llamada *listaImputs*. Si algún valor no se ha introducido devuelve un error.
- **GetRectángulos():** Se crea un diccionario para asociar una lista de objetos del entorno a cada tipo de refuerzo. Esta lista de objetos se obtiene a partir de las funciones *getRectanglesFromObjects()* y *findObjectsByType()*.
- **GetRefuerzoBot():** Este método tiene la misma funcionalidad que *getEntornoBot()* y se ha simplificado más para poder automatizar la lectura del diccionario de la función *getRectángulos()*. Hay que recordar que esta función devuelve la recompensa que recibe el agente por alcanzar ese estado. En ella simplemente se recorre cada una de las listas de objetos de ese diccionario y se compara si el rectángulo que forma el Sprite del agente colisiona con el rectángulo de los objetos de cada una de las listas.