1 Introducción

En esta sección vamos a describir en qué consistirá el proyecto, justificando las razones de por qué se ha elegido.

La idea del proyecto consiste en implementar un videojuego basado en un juego de rol. Para ello veremos primeramente en qué consisten los juegos de rol, además de indicar qué características de los mismos incluiremos en nuestro proyecto.

1.1 ¿Qué es un juego de rol?

Un juego de rol representa un mundo imaginario con una serie de propiedades y restricciones donde un conjunto de personajes pueden actuar en el mismo de acuerdo a sus características, objetivos y necesidades.

Ser un jugador de rol consiste en asumir el papel de un determinado personaje ficticio y actuar de acorde a sus características en un mundo previamente definido.

Todos los juegos de rol son controlados por un Master, que es la persona responsable de dirigir el juego, es decir, de definir el mundo y controlar las acciones de los distintos jugadores.

De este modo, existen unas reglas de juego, acordadas entre los jugadores y el Master, que permiten definir las distintas acciones a realizar durante la partida.

En realidad, un juego de rol consiste en una serie de interacciones entre los jugadores (que dirigen a sus personajes en las aventuras) y un Master (el cual desarrolla el mundo en el que suceden dichas aventuras). La mayor parte del juego se lleva a cabo mediante un intercambio verbal: los jugadores le dicen al Master lo que sus personajes intentarán hacer, y el Master les confirma si pueden o no hacerlo, además de indicarles lo que sucede en este último caso. Las reglas determinan lo que puede hacerse frente a una situación concreta: si hay varias posibilidades, la situación se resuelve con una tirada de dados determinada. De este modo, aquellas acciones cuya resolución sea indeterminada necesitarán de dichas tiradas, dando como resultado una respuesta aleatoria de entre las disponibles, basada en probabilidades.

Un ejemplo de las acciones asociadas a las tiradas de dados podría ser la siguiente:

Un personaje desea trepar una pared. Las posibilidades serían dos: conseguir trepar la pared (éxito), o no conseguir treparla (pifia). Para ello, se lanzaría una tirada de dados, y en función de su valor, si supera o no un determinado umbral, se tendría un éxito o un fracaso.

También se utilizan los dados para seleccionar las características de los personajes, ya que éstos se crean al azar.

Las tiradas pueden incluir a varios dados, y a dados de distinto número de caras. Normalmente, se suelen especificar las tiradas de esta forma: *xDy* (donde *x* indica el número de dados empleados en la tirada, e *y* indica el número de caras de los dados). Por ejemplo, 2D6 hace referencia a una tirada de dos dados de seis caras. Es decir, se lanzarían los dos dados y se sumarían los valores obtenidos.

A veces, a los valores obtenidos por los dados se le suman unas cifras específicas. Por ejemplo, 2D6 + 2 significa que al valor obtenido por la suma de los dos dados se le añade un 2.

Para que un jugador pueda asumir el papel de un personaje necesita una serie de características que definan a dicho personaje. Estas características, generalmente, pueden ser las siguientes:

- Nombre (nombre del personaje)
- Género (género al que pertenece el personaje: masculino o femenino)
- Especie (raza a la que pertenece el personaje)
- Fuerza (fuerza muscular del personaje)
- Constitución (constitución física del personaje)
- Tamaño (masa corporal del personaje)
- Inteligencia (capacidad del personaje para pensar, memorizar, y coordinar algunas acciones)
- Destreza (medida de coordinación y velocidad del personaje)

La especie de los personajes, principalmente, puede pertenecer a una de las siguientes:

- Humano (raza predeterminada)
- Elfo (criaturas de los bosques semejantes a los humanos, aunque más delgados y levemente más bajos con orejas puntiagudas)
- Enano (pequeños humanoides robustos que habitan bajo la superficie de la tierra)
- Hobbit (pequeños humanoides civilizados que viven en aldeas)
- Orco (especie humanoide nocturna que habita en montañas y tierras sin cultivar)

Puede haber más variedad en función del juego de rol.
Las características anteriores difieren según la especie.

Otro aspecto de los personajes son los puntos de golpe, es decir, la vida del personaje. Dichos puntos indican cuanto daño es capaz de soportar un personaje antes de morir. Cuando se realiza un ataque, dichos puntos pueden verse afectados. Al llegar a 0 el personaje muere.

Los personajes también tienen fatiga durante la partida. Es decir, sufren un determinado cansancio, ya sea en forma de hambre, de sed, o de fatiga en general. Para ello, es necesario que descansen, coman o beban, ya que en otro caso se verían afectados los puntos de golpe.

Cada personaje puede tener, también, determinadas habilidades o acciones especiales, tales como nadar, trepar, saltar, robar, percibir, etc... Dichas habilidades se obtienen a partir de unos determinados modificadores de habilidad obtenidos en base a los valores de las características del

personaje. El poder llevar a cabo una habilidad con éxito depende de una tirada de dados determinada. Además, las habilidades se pueden enfrentar entre sí, de modo que un personaje, para tener éxito en su habilidad, tendrá que sobrepasar el éxito de la habilidad de su contrincante.

Los personajes pueden realizar una gran variedad de acciones en el mundo, tales como caminar, entrar en edificios, comer, beber, descansar, coger objetos, comprar objetos, atacar, realizar magias, o acciones asociadas con las habilidades como trepar, robar, esconderse, buscar trampas, etc...
Algunas de ellas requerirán tiradas de dados y otras no, por tratarse de acciones triviales. Por ejemplo, entrar en un edificio o caminar no requiere ninguna probabilidad, sin embargo, realizar un hechizo o atacar, necesita de una tirada de dados que obtendrá su resultado.

Los ataques vienen definidos por varios factores como son los puntos de daño del arma utilizada y el modificador de daño que tenga el personaje. Dichos valores se contrarrestan con los puntos de armadura que tenga el receptor del ataque.

Otro aspecto, son las magias. Los personajes de rol pueden aprender hechizos y realizar una serie de magias, ya sea con fines de ataque, de defensa o para crear algún efecto determinado.

Cada personaje tiene asociado un inventario, donde se recogen todos los elementos que posea, tales como monedas de oro, armas, armaduras, escudos, pergaminos mágicos, etc...
Al comprar objetos, el personaje los inserta en el inventario. Normalmente, hay un límite de objetos para llevar.

Un atributo relacionado con el inventario es la carga del personaje. Este atributo hace referencia al peso que puede transportar un personaje. Las armaduras, escudos y armas tienen un peso determinado que en función de su valor hacen que el personaje se fatigue antes.

El sistema de juego en los juegos de rol, normalmente, se basa en turnos, de modo que cada jugador, por turno, solicita una o varias acciones al Master, que posteriormente informa de su resultado.

Cada personaje tiene un momento de reacción, que indica en qué momento puede actuar. Este atributo se puede utilizar para saber el número de acciones a realizar por turno, ya que si un personaje tiene un buen momento de reacción, le da tiempo a realizar más de una acción. Su valor se puede ver afectado por las armas y los escudos que entorpecen la rapidez de actuación.

Por último, comentaremos que el mundo donde se desarrollan los juegos de rol son mundos llenos de incertidumbre, donde surgen varias situaciones de conflicto (diseñadas por el Master) que requerirán que los jugadores tomen decisiones críticas en función sus roles (personajes).

Una vez que hemos visto, en rasgos generales, en qué consiste un juego de rol pasemos a justificar que nos puede aportar como proyecto.

1.2 ¿Por qué un juego de rol?

Como vemos, un juego de rol puede ser una simulación de un entorno donde unos personajes presentan un cierto comportamiento. De este modo, podría verse como un sistema multiagente, donde cada personaje sería un agente en sí. Esta es la idea principal del proyecto.

Para implementar un juego de rol bastaría con que la función controladora del Master la llevara el ordenador, mientras que los jugadores serían los usuarios que jugarían asociándose a personajes dentro de un entorno ficticio creado por el mismo ordenador.

Como proyecto, un videojuego de rol nos puede aportar varias ideas interesantes, como la creación de un mundo simulado con ciertas restricciones y reglas, donde un conjunto de agentes tienen que tomar una serie de decisiones de comportamiento.

Además, existen dos tipos de agentes:

- Jugadores manejados por el usuario, también denominados *PJs (Personajes Jugadores)*
- Jugadores manejados por el ordenador, también denominados *PNJs (Personajes No Jugadores)*

La diferencia entre uno y otro tipo radicará en la toma de decisiones, es decir, los PJs actuarán del modo que ordene el usuario y los PNJs actuarán en base a un comportamiento inteligente programado. De este modo, podremos ver como interactuarán los distintos agentes, ya sean personajes humanos o computacionales, comparando sus decisiones.

En conclusión, el proyecto consistirá en un sistema multiagente, donde cada agente tendrá disponible una serie de acciones y tendrá que elegir aquella que le parezca más correcta frente a una situación del mundo determinada. Una vez tomada dicha acción su éxito dependerá de determinadas circunstancias, bien representadas en tiradas de dados, o bien asociadas a las características del personaje y del mundo.

1.3 Características del juego

Para que el juego sea multiagente necesitará tener una estructura multijugador, es decir, que puedan participar diversos jugadores, ya sean PJs o PNJs.

Ambos tipos de jugadores serán tratados de forma similar por el juego, ya que la única diferencia, como ya hemos mencionado, se basará en la forma de pensar.

Como ya hemos dicho, el ordenador, que hará de Master, creará el mundo del juego, donde cada personaje percibirá una parte del mismo (radio de percepción) que le servirá para juzgar que acción debe tomar en esa situación. El ordenador, será el encargado, como buen Master, de aceptar o de rechazar la acción solicitada, así como de lanzar los dados para ver su resultado.

Los jugadores solicitarán sus acciones por turnos. De este modo, los jugadores serán ordenados crecientemente según su momento de reacción. Aquel personaje que tenga el menor valor de reacción, será el más rápido en actuar, y por consiguiente, tendrá más preferencia. Cuando el personaje realice su acción (o acciones en caso de que pueda hacer varias), se pasará el turno al siguiente en el orden establecido.

Veamos, a continuación, los elementos de rol que implementaremos en nuestro juego.

Trataremos de implementar un resumen de reglas a partir de un juego de rol real, RuneQuest. El basarnos en un juego real nos permite obtener una serie de reglas equilibradas que sabemos que obtienen resultados coherentes.

Además, utilizaremos un resumen, ya que incluir todos los aspectos de rol es un trabajo muy extenso, por tratarse de un juego muy abierto, y no se corresponde con el propósito de nuestro proyecto. Así que implementar aún más restricciones o posibilidades podemos dejarlo como ampliación futura.

El mundo que vamos a crear estará formado por cuatro elementos:

- Personajes (asociados a los jugadores)
- Objetos (elementos estáticos del mundo que pueden ser utilizados por los personajes)
- Edificios (elementos estáticos del mundo que sirven de entrada a otros escenarios o submundos)
- Casillas (tipos de terrenos del mundo)

Veamos las características de cada uno.

PERSONAJES:

Especie (esp)

Raza del personaje. Consideraremos las siguientes: humano, hobbit, elfo, gigante, mono.

Fuerza (fue)

Fuerza del personaje. Su valor, obtenido a partir de los dados, diferirá según la especie:

Humano	3D6
Hobbit	2D6
Elfo	2D6 + 2
Gigante	3D6 + 6
Mono	2D6

Podrá aumentarse como máximo hasta la suma de la constitución y el tamaño del personaje.

Constitución (con)
Constitución del personaje. Su valor, según la especie, será el siguiente:

Humano	3D6
Hobbit	2D6 + 12
Elfo	3D6
Gigante	2D6 + 6
Mono	2D6

Podrá aumentarse como máximo hasta la suma de la fuerza y tamaño del personaje.

Tamaño (tam)
Tamaño del personaje. Su valor, según la especie, será el siguiente:

Humano	2D6 + 6
Hobbit	2D3
Elfo	2D6 + 8
Gigante	3D6 + 6
Mono	2D3

Su valor no podrá modificarse.

Inteligencia (inte)
Inteligencia del personaje. Su valor, según la especie, será el siguiente:

Humano	2D6 + 6
Hobbit	2D6 + 6
Elfo	3D6 + 6
Gigante	3D6
Mono	0

Su valor no podrá modificarse.

Destreza (des)
Destreza del personaje. Su valor, según la especie, será el siguiente:

Humano	3D6
Hobbit	2D6 + 10
Elfo	3D6 + 3
Gigante	3D6
Mono	3D6 + 10

Podrá aumentarse como máximo hasta el doble del valor actual.

Puntos de golpe (pg)
Puntos de golpe del personaje. Su valor se obtendrá a partir de la media alta de la constitución y el tamaño del personaje. Representarán los puntos de vida del personaje, de modo que cuando llegue a 0 el personaje morirá.

Energía (ene)
Energía del personaje. Su valor máximo se obtendrá a partir de la suma de la fuerza y la constitución del personaje. Se irá decrementando en 1 cada 10 turnos. En función de la carga del personaje los decrementos serán mayores. Cuando su valor llegue a 0 irá decrementando en 1 los puntos de golpe del personaje en cada turno.

Hambre (ham)
Resistencia del personaje al hambre. Su valor máximo se obtendrá a partir de la suma de la constitución y el tamaño del personaje. Al igual que la energía, se irá decrementando en 1 cada 10 turnos. La carga también influirá en decrementos mayores. Cuando su valor llegue a 0 decrementará los puntos de golpe en cada turno.

Sed (sed)
Resistencia del personaje a la sed. Presentará las mismas características que el atributo anterior.

Modificador de habilidad de sigilo (mh1)
Modificador de sigilo del personaje. Necesario para todas aquellas acciones que impliquen habilidades de sigilo. Su valor se obtendrá a partir de la destreza y el tamaño del personaje. La destreza será una influencia primaria y el tamaño será una influencia negativa. Las influencias positivas incrementarán el modificador por cada punto que superen de 10, o decrementarán el modificador por cada punto que bajen de 10. Las influencias negativas actuarán a la inversa.

Modificador de habilidad de percepción (mh2)
Modificador de percepción del personaje. Necesario para todas aquellas acciones que impliquen habilidades de percepción. Su valor se obtendrá a partir de la inteligencia y la constitución del personaje. La inteligencia será una influencia primaria y la constitución será una influencia secundaria. La influencia primaria actuará como ya comentamos. La influencia secundaria incrementará el modificador por cada dos puntos que supere de 10, o decrementará el modificador por cada dos puntos que baje de 10.

Momento de reacción (mmr)
Momento de reacción del personaje. Indicará cuando el personaje podrá actuar. Cuanto más bajo sea su valor más rápido actuará el personaje. Además, en función de este atributo, un personaje tendrá más o menos acciones disponibles por turno.
Su valor se obtendrá a partir del tamaño y la destreza del personaje de la siguiente forma:

TAM/DES	MMR TAM	MMR DES
1 a 9	3	4
10 a 15	2	3
16 a 19	1	2
20 o más	0	1

Para obtener el mmr total se sumarán los obtenidos en tamaño y destreza.

El número de acciones por turno serán:

MMR	ACCIONES
1 a 3	4
4 a 6	3
7 a 9	2
10 o más	1

Los personajes participarán en orden creciente de momento de reacción. Las armas y los escudos incrementarán este valor, provocando que los personaje sean más lentos a la hora de actuar.

Equipo (equ)
Inventario del personaje. Indicará la lista de elementos que tendrá como pertenencias el personaje. El número de objetos máximo será 6 de forma que se podrá llevar lo siguiente:

- Un escudo
- Un arma
- Una armadura
- Monedas de oro
- Una llave
- Una poción

Todos los elementos serán excluyentes (excepto las monedas de oro). De modo que si un personaje, por ejemplo, compra un escudo diferente del que tiene, se sustituirá por el actual.

Habrá sólo un tipo de llave, de forma que cuando un personaje ya la tenga no podrá obtener otra.

Habrá distintos tipos de pociones (numeradas del 1 al 10). El personaje podrá obtener una nueva poción siempre y cuando no la tenga en su inventario, o tenga otra distinta, en cuyo caso se sustituiría.

Las monedas de oro, presentes en cantidades de 50 unidades, se podrán incrementar.

OBJETOS
Tipo (tipo)
Tipo de objeto. Consideraremos los siguientes:

- árbol
- setas
- tarro de miel
- cofre
- bruja
- candelabro
- huesos
- portal mágico
- estatua
- esclavos prisioneros
- esclavos libres
- trampa
- bibliotecaria
- mesa
- biblioteca
- posadera
- cama

- vela
- escoba
- armero
- estatua con armadura
- espada corta
- espada ancha
- hacha
- bola y cadena
- escudo tipo1
- escudo tipo 2
- escudo tipo 3
- maniquí
- cota de cuero
- cota de anillos
- armadura
- coraza

Visibilidad (visi)
Visibilidad del objeto. Indicará si el objeto está visible o invisible en el mundo. Sólo serán invisibles los objetos trampas.

Equipo (equ)
Inventario del objeto, en caso de los objetos cofres. Indicará el elemento que contiene. Su valor podrá ser alguno de los siguientes:

- llave
- poción (i) (i: 1->10)
- monedas (50 unidades)

Veamos ahora, ya que hemos indicado todos los objetos del juego, los incrementos de carga y mmr que originan los objetos escudos, armas y armaduras.

OBJ	INCR (MMR)	INCR (CAR)
Espada corta	2	1
Espada ancha	2	2
Hacha	2	3
Bola y cadena	2	4
Escudo tipo 1	3	1
Escudo tipo 2	3	2
Escudo tipo 3	3	4
Cota de cuero	-	1
Cota de anillos	-	2
Armadura	-	3
Coraza	-	4

La carga total o el incremento de mmr total será la suma de los valores asociados a los objetos que tenga el personaje.

EDIFICIOS

Tipo (tipo)

Tipo de edificio. Se considerarán los siguientes:

- Castillo (que requerirán una llave para entrar)
- Posada (donde el personaje podrá descansar y curarse las heridas)
- Biblioteca (donde el personaje podrá estudiar)
- Tienda de armas (donde el personaje podrá comprar los objetos armas indicados)
- Tienda de escudos (donde el personaje podrá comprar los objetos escudos indicados)
- Tienda de armaduras (donde el personaje podrá comprar los objetos armaduras indicados)

Llave (llav)

Llave del edificio. Indicará si el edificio necesita llave o no para entrar.

CASILLAS

Tipo

Tipo de casillas. Consideraremos las siguientes:

- Hierba
- Tierra
- Piedra
- Poza de agua
- Interior de castillos
- Interior de tiendas y centros

Las acciones que podrá realizar cada personaje podrán ser las siguientes:

(1) Moverse por el mundo

Cada personaje podrá caminar por toda la extensión del mundo teniendo en cuenta que habrá elementos bloqueantes, que no podrá sobrepasar. Todos los personajes serán bloqueantes entre sí. Todos los objetos serán bloqueantes excepto los árboles, setas, cofres, tarros de miel, trampas, camas y los elementos de las tiendas (armas, escudos y armaduras).

Cuando un personaje pase por encima de una trampa visible, sus puntos de golpe se decrementarán en 2 puntos. Si la trampa es invisible se activará haciéndose visible, y se realizará una tirada de dados de habilidad para saber si el personaje la detecta o no.

La tirada de habilidad consistirá en lanzar 1D100 comprobando su resultado con el modificador de habilidad correspondiente. En este caso, se utilizará el mh2 (percepción). Si el valor obtenido en el dado es menor o igual al modificador se obtendrá un éxito, y el personaje detectará la trampa a tiempo antes de pasar por ella. En caso contrario, el personaje pasará por la trampa perdiendo 2 puntos de golpe.

(2) Entrar/Salir de edificios

Los personajes podrán acceder al interior de los edificios a partir de la puerta de la fachada, siendo bloqueantes las paredes del edificio. Para salir del mismo, los personajes tendrán que moverse hacia la salida del recinto, normalmente, situada en el centro del muro inferior. Es necesario indicar que aquellos personajes que pertenezcan a la especie mono no podrán acceder a ningún edificio.

Para acceder a los castillos se necesitará tener una llave en el inventario.

(3) Atacar

Para atacar, los personajes utilizarán el arma que tengan en el inventario. En caso de no tener ninguna, usarán un arma natural (puños, piernas, etc...).

Cada arma tendrá asociados unos puntos de daño (pd). Consideraremos los siguientes pd para los tipos de armas:

Espada corta	1D6 + 1
Espada ancha	1D8 + 1
Hacha	1D8 + 2
Bola y cadena	1D10 + 1

En caso de que el personaje no tenga ningún arma en su inventario se considerarán unos puntos de armas naturales. Todas las especies tendrán como pd natural 1D6, exceptuando a los gigantes que tendrán 2D6.

Además de los pd asociados al arma, cada personaje atacante utilizará un modificador de daño (md) que se sumará a los pd. El modificador de daño se obtendrá a partir de la suma de la fuerza y el tamaño según la siguiente tabla:
Nota: Se considerará que tanto fue como tam son mayores que 0. En otro caso md sería 0.

FUE + TAM	MD
1 a 12	-1D4
13 a 24	0
25 a 32	+1D4
33 a 40	+1D6
41 a 56	+2D6
57 o más	+1D6 por cada incremento de 16 unidades

El ataque realizado por un personaje consistirá en sumar su pd más su md. Ese será el daño infligido total por el atacante. Pero en un combate no solo se tendrá en cuenta el ataque realizado, si no también la resistencia a dicho ataque. Esto lo constituirán los puntos de armadura (pa) del personaje atacado. Para que el atacado sufra daño tendrá que recibir un golpe que sobrepase sus puntos de armadura. El resultado de daño de un combate vendrá dado por la siguiente expresión:

$$daño = pa - (pd + md)$$

El daño obtenido, en caso de ser un valor negativo (el ataque sobrepasa la armadura), se restará a los puntos de golpe del atacado.

Los puntos de armadura de un personaje vendrán determinados de la armadura y/o escudo que posea en su inventario, o, al igual que antes, de unos puntos de armadura natural. Consideraremos los siguientes valores:

Todas las especies tendrá como pa un valor igual a 3, excepto los gigantes que tendrán 6.

Veamos ahora los pa asociados a los escudos y armaduras:

Escudo tipo 1	1D2
Escudo tipo 2	1D4
Escudo tipo 3	1D6
Cota de cuero	2
Cota de anillos	5
Armadura	7
Coraza	9

Los pa totales serán la suma de los obtenidos en los elementos del inventario del personaje (escudo y armadura).

(4) Robar

Para robar será necesario un enfrentamiento de habilidades entre los dos personajes. El personaje que robe utilizará su habilidad de sigilo (mh1), mientras el personaje víctima utilizará su habilidad de percepción (mh2).

En primer lugar, se lanzará una tirada de 1D100 para el personaje víctima del robo. Se comprobará si el valor obtenido es menor o igual que el modificador de percepción. En tal caso, el personaje tendrá un éxito. A continuación, se lanzará otra tirada de habilidad para el personaje ladrón, teniendo como umbral la diferencia entre los dos modificadores de ambos personajes. Si esta tirada sale exitosa, el personaje ladrón cometerá el robo, en otro caso no se llevaría a cabo, ya que la víctima detectaría el intento de robo. En caso de que errara el personaje víctima en la primera tirada, el éxito del robo se basaría en una tirada simple de sigilo del personaje ladrón.

Si el robo sale exitoso, se incrementará el inventario del personaje ladrón con los elementos que no posea según las siguientes reglas:

- El número de monedas de oro de la víctima pasarán a incrementar el número de monedas del ladrón, quedándose la víctima con 0 monedas.

- Las armas, escudos y armaduras no se verán afectadas por el robo.
- Las pociones se intercambiarán en caso de que ambos personajes tengan distintos tipos. En caso de que el ladrón no tuviera poción y la víctima sí, se robaría la de la víctima, quedándose ésta sin ninguna. En otro caso no se robaría la poción.
- La llave se robará sólo en caso de que la víctima tenga una y el ladrón no.

(5) Comer

Los personajes podrán comer, y por tanto, restablecer sus valores de ham al máximo a través de los objetos setas y tarros de miel. Los primeros incrementarán 8 puntos, mientras que los segundos restablecerán hasta el máximo.

Los tarros de miel sólo servirán 1 vez, de modo que el personaje que los utilice para comer los gastará, a diferencia de las setas que no se acabarán en el juego.

(6) Beber

Los personajes podrán beber a través de terrenos que tengan pozas de agua. De este modo podrán restablecer el valor de sed al máximo.

(7) Descansar/Gastar turno

Los personajes podrán descansar en cualquier momento gastando todo su turno de forma que puedan restablecer el valor de ene al máximo. No podrán realizar más acciones en ese turno.

(8) Abrir cofres

Todos los cofres que haya en el mundo podrán ser abiertos por los personajes de modo que incrementarán su inventario con los elementos que encuentren dentro (cumpliendo las reglas que se vieron en la acción robar). Cuando hayan cogido el contenido de un cofre, éste desaparecerá. En caso de que el personaje no necesite el elemento del interior del cofre, éste se quedará inalterado.

El mundo tendrá un sistema de realimentación de modo que cuando un personaje muera, todos los elementos de su inventario (exceptuando armas, armaduras y escudos) se quedarán en el mundo en forma de cofres. Para las monedas de oro, se crearán cofres cada 50 unidades.

Los monos tendrán vetada esta acción.

(9) Liberar prisioneros

Los personajes podrán rescatar aquellos esclavos prisioneros que aparezcan en los castillos. A cambio recibirán una recompensa de 50 monedas de oro.

Los monos tendrán vetada esta acción.

(10) Entrenar/Aceptar misiones

Los personajes podrán entrenarse realizando misiones. Para ello necesitarán una bruja que les entrene, que les encomendará una misión determinada. Las misiones consistirán en encontrar una determinada poción y llevarla a un portal mágico de algún castillo. Los personajes tendrán que situarse en medio del portal, y gastar su turno.

Habrá 10 misiones asociadas a los 10 tipos de pociones que haya. Las misiones serán crecientes, de forma que cuando un personaje haya realizado la misión 1, la próxima será la 2, después la 3, etc...
En caso de que se llegase a la 10 se volvería a repetir el ciclo.
Además, el entrenamiento supondrá un coste incremental. Cada personaje tendrá que pagar 50 monedas de oro por nivel de misión. Por ejemplo, para la primera misión pagará 50, para la segunda 100, para la tercera 150, y así sucesivamente.

Los monos tendrán vetada esta acción.

(11) Finalizar misiones

Como ya hemos dicho, los personajes finalizarán la misión encomendada cuando lleven la poción objetivo a un portal mágico. Finalizar una misión supondrá, además, una recompensa. Se incrementarán en 1 punto las características fue, con, des, restableciendo al máximo los valores de pg, ene, ham, sed, y ajustando, nuevamente, los valores de mh1, mh2, y mmr.

(12) Curarse en Posadas

Los personajes podrán curarse, es decir, restablecer al máximo sus pg (además de ham, sed y ene), entrando en posadas. Para ello, tendrán que pagar 50 monedas a la posadera.

(13) Estudiar en Bibliotecas

Los personajes podrán incrementar sus puntos de mh1, y mh2 estudiando en bibliotecas. Tendrán que pagar a la bibliotecaria 50 monedas cada vez que estudien, y obtendrán un incremento de 1 punto. Además, cada estudio supondrá un decremento de 2 puntos por nivel de estudios en ham, sed y ene. Los modificadores de habilidad podrán incrementarse hasta un máximo de 100 puntos.

(14) Comprar armas/escudos/armaduras

Para comprar estos elementos los personajes tendrán que entrar en las tiendas correspondientes y comprárselos al vendedor. Cada elemento tendrá un precio determinado. Como ya sabemos, un personaje no podrá obtener un elemento que ya tenga, pero sí podrá hacer un cierto "intercambio" de venta/compra para obtener otro elemento distinto del mismo tipo. Por ejemplo, supongamos que un personaje tiene una espada y quiere comprarse un hacha. Para ello, tendrá que venderle la espada al vendedor y comprar el hacha. Lo mismo podría hacer al revés, vender el hacha y comprar una espada, con el consiguiente ahorro de dinero (el personaje recuperaría dinero).

Los precios de los elementos serán los siguientes:

Espada corta	100 monedas de oro
Espada ancha	150 monedas de oro
Hacha	200 monedas de oro
Bola y cadena	250 monedas de oro
Escudo tipo 1	100 monedas de oro
Escudo tipo 2	150 monedas de oro
Escudo tipo 3	200 monedas de oro
Cota de cuero	100 monedas de oro
Cota de anillos	150 monedas de oro
Armadura	200 monedas de oro
Coraza	250 monedas de oro

Con esto ya hemos visto todas las características de rol que implementaremos en nuestro juego.
A continuación vamos a ver los objetivos del proyecto para luego centrarnos en el diseño del mismo.

1.4 Objetivos

Como ya dijimos en la primera parte de la introducción, la idea principal del proyecto se basa en una simulación de comportamientos de un conjunto de agentes en un mundo ficticio.

De este modo, podemos establecer como objetivos del proyecto los siguientes:

- El primer objetivo del proyecto, sería el obtener distintos comportamientos de los agentes PNJs frente a los estados del mundo o situaciones. Como cada agente PNJ tendrá asociado un determinado comportamiento inteligente, y por tanto, actuará de modo distinto, su toma de decisiones será diferente a otra que tome otro agente que tenga otra "inteligencia" distinta.

- Un segundo objetivo podrá ser que cuando haya varios agentes actuando en el mundo, tanto PJs como PNJs, su comportamiento sea similar, es decir, que intenten actuar de forma parecida a como lo haría un PJ en esa situación, simulando, en cierta forma, un comportamiento "inteligente".

- El tercer objetivo consistiría en obtener agentes más o menos racionales, es decir, agentes cuyas decisiones sean o se acerquen a las más correctas para cumplir sus objetivos.

Los tres objetivos del proyecto tienen una base común y es conseguir que los personajes piensen como si fuesen reales dentro de un mundo determinado, y ésto coincide con la idea que se tiene en un juego de rol.

2 Estructura de diseño

En esta sección vamos a explicar las partes fundamentales en las que se dividirá el proyecto. Para ver cuál es la estructura del proyecto basta con analizar lo que vamos a hacer; la idea es un sistema multiagente donde los agentes, que pueden ser de dos tipos: PJs y PNJs, actúen en un mundo imaginario.

Veamos cada parte independientemente.

En primer lugar, como el sistema es multiagente o multijugador, necesitaremos que el proyecto presente una arquitectura en red, es decir, que permita a los jugadores conectarse al juego para poder participar, y que las acciones de cada uno de ellos modifiquen el mundo, de modo que cada jugador vea el mundo con los cambios del resto de los jugadores.

Gráficamente:

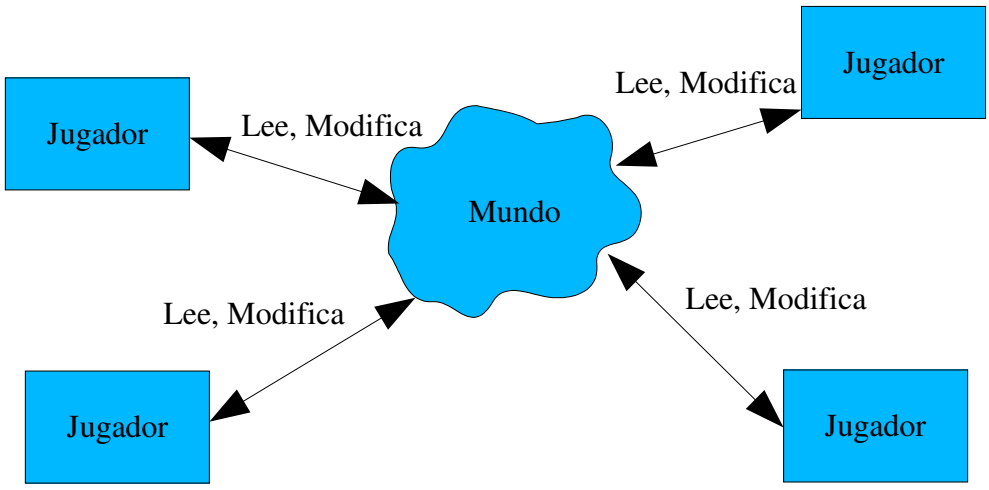


Fig. 1. Modelo multijugador

En segundo lugar, para que los agentes actúen, necesitarán percibir el entorno de alguna forma. Ambos tipos de jugadores tienen un radio de percepción, que les permite analizar la situación. Dicho radio de percepción puede ser un conjunto de datos que represente una parte del mundo. Sobre dichos datos los PNJs pueden realizar su toma de decisiones, pero el problema reside en los PJs. ¿Cómo se puede hacer visible el conjunto de datos para los usuarios? La solución es bastante sencilla, necesitaremos una interfaz gráfica para los PJs que represente el estado del mundo en cada momento.

Los usuarios, una vez conectados al juego, leerán la parte del mundo que se corresponda con su radio de percepción, la visualizarán, y realizarán una determinada acción que modifique el mundo en dicho radio.

Gráficamente:

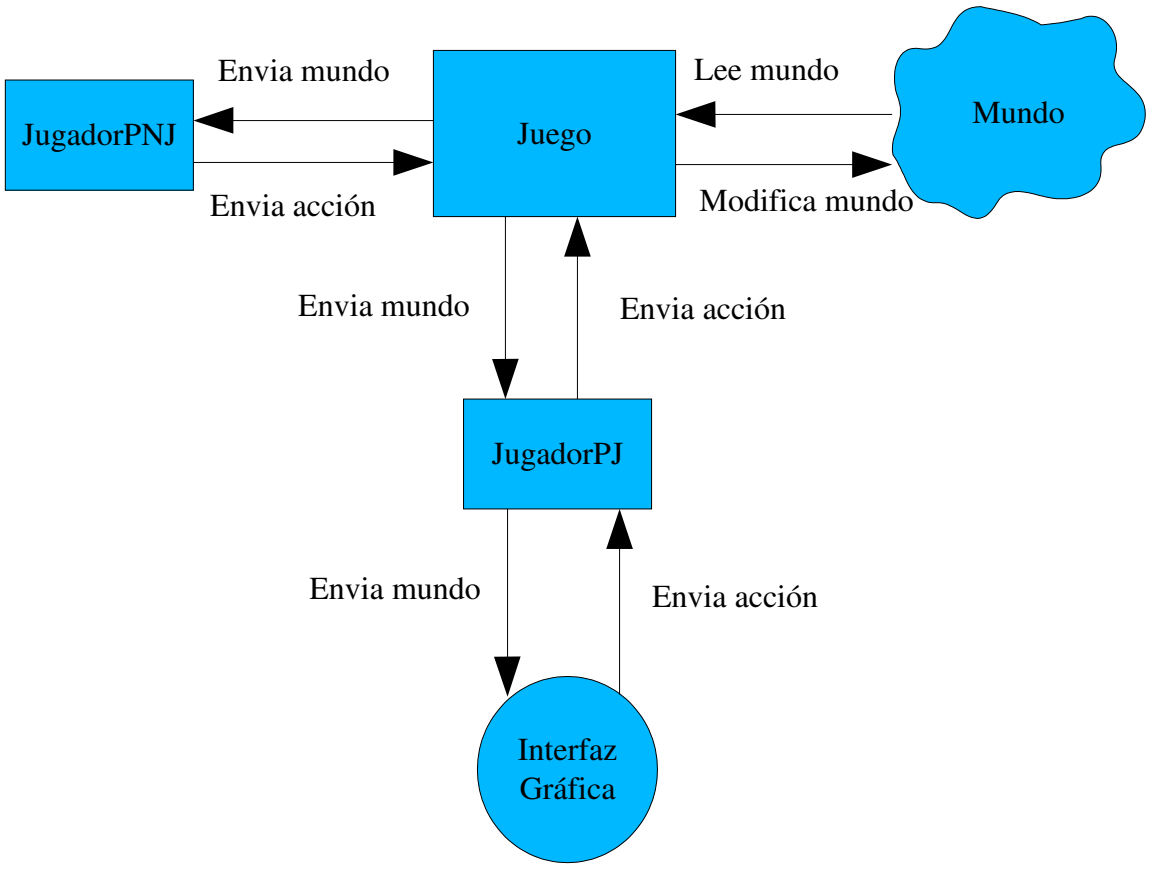


Fig. 2. Modelo parcial del proyecto

En tercer lugar, los agentes computacionales tendrán que tomar decisiones para realizar las acciones correspondientes. Por tanto, necesitaremos un comportamiento inteligente que analice la situación en cada caso y obtenga como resultado una decisión que se convertirá en la acción que realice el agente.

El esquema sería el siguiente, el cliente PNJ se conecta al juego, percibe la parte del mundo asociada a su radio de percepción, y sobre dicha información lanza un programa de análisis (algoritmo inteligente) que de como resultado la acción correspondiente, que a su vez se envía como respuesta al juego para modificar el mundo.

Gráficamente, el funcionamiento es el siguiente:

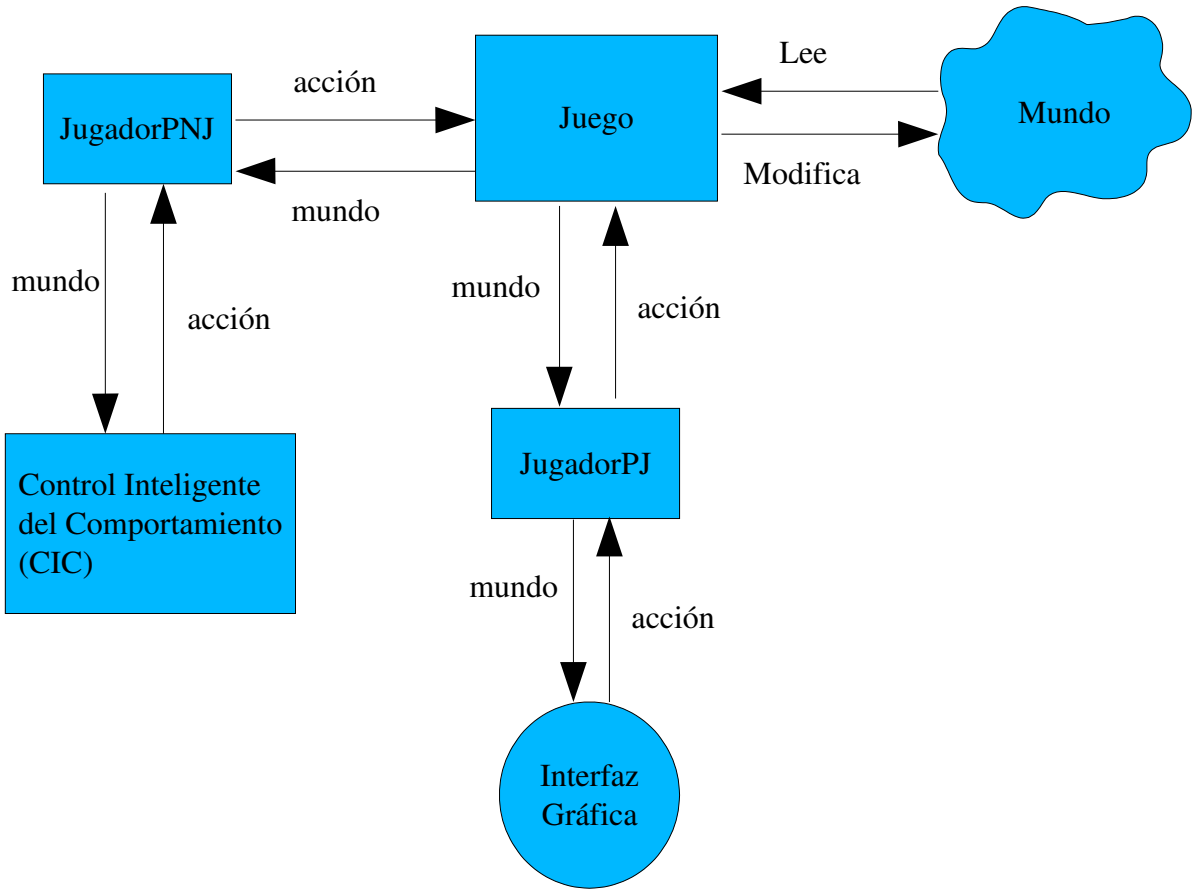


Fig. 3. Modelo final del proyecto

De esta forma, podemos comprobar que las partes fundamentales en las que se basa el proyecto son: Arquitectura de red, Interfaz Gráfica y Control Inteligente del Comportamiento.

La arquitectura en red se implementará en base a un esquema cliente/servidor. El servidor será el encargado de recibir las peticiones de los clientes (PJs y PNJs) y de realizar en el mundo los cambios pertinentes. El servidor será, por tanto, el programa principal del proyecto. Ocupará el puesto del "Master" del juego.
Para los clientes tendremos dos programas; uno para el cliente PJ, y otro para el cliente PNJ.
Ambos se comunicarán con el servidor a través de un socket.

Para implementar la arquitectura en red se utilizará el lenguaje C/C++ y las funciones asociadas al manejo de sockets.

La interfaz gráfica se implementará también en C/C++ con ayuda de las librerías OpenGL, SDL y SDL_image.
La primera de ellas, se utilizará para el dibujo de los objetos, la segunda para el manejo de los eventos de la interfaz, y la tercera para el manejo de las texturas.

El comportamiento inteligente se implementará en el lenguaje Python. El hecho de utilizar este lenguaje radica en su característica de ser un lenguaje interpretado, lo cual proporciona que la inteligencia sea dinámica, es decir, que se pueda modificar de forma independiente al resto del proyecto, ya que un lenguaje interpretado proporciona la ventaja de ser muy rápido de modificar por no necesitar compilarse, y por tanto, no tiene que alterar el resto del programa en C/C++.

El sistema operativo en el que se ejecutará el proyecto será Linux.

El esquema del proyecto, por tanto, es el siguiente:

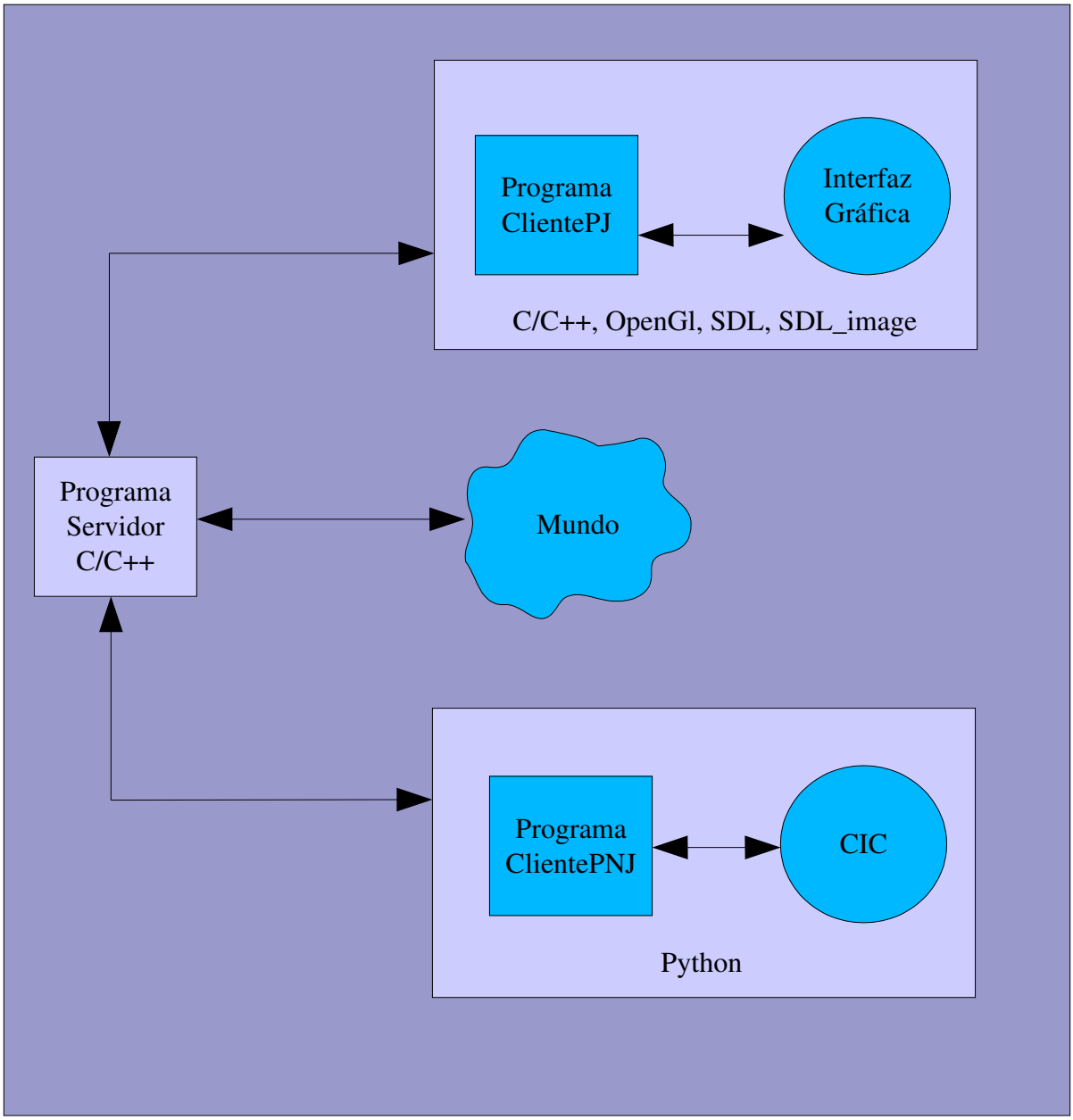


Fig. 4. Esquema del proyecto

3 Arquitectura en Red: modelo Cliente/Servidor

En esta sección vamos a explicar el modelo cliente/servidor que permite que el juego sea multijugador. En primer lugar describiremos el funcionamiento del servidor, y posteriormente, detallaremos el protocolo de comunicación empleado, así como el cliente y las estructuras de datos.

3.1 Servidor

El servidor es la parte principal del proyecto. En realidad, es el propio juego en sí mismo, ya que se encarga de controlar todo el funcionamiento del proyecto.

Cada cliente o jugador constituirá un personaje del juego y tendrá una serie de atributos específicos que lo definirán.

Como ya dijimos anteriormente, el servidor gestionará las peticiones de los clientes que se conecten a él según un orden establecido. Dicho orden establecerá el turno entre los jugadores. De este modo, podemos ver que el sistema de juego será iterativo, porque hasta que un cliente no realice la acción solicitada no se cederá el turno al siguiente. Sin embargo, la estructura del servidor no podrá ser iterativa, ya que si actuara de este modo, los clientes se quedarían bloqueados. El servidor tendrá que estar en continua interacción con cada cliente conectado. Por ello, su funcionamiento será concurrente.

Básicamente, el proceso será el siguiente:

1. El cliente se conecta al servidor.

2. El servidor crea un personaje asociado al cliente y lo inserta en la lista de clientes conectados.

3. El cliente solicita al servidor una determinada acción a realizar en el mundo.

4. El servidor comprueba si el turno le corresponde al cliente solicitante.

5. En caso de afirmativo, el servidor comprueba la acción solicitada.

6. Si la acción es válida, es decir, puede realizarse en el estado del mundo actual, el servidor modifica el mundo con la acción del cliente e incrementa el turno.

7. Si la acción no es válida, el servidor espera hasta que el cliente solicite una acción correcta.

Los puntos 1 y 2 se realizarán cuando el cliente se conecte al servidor. Los puntos 3-6 constituirán un bucle de interacción con los clientes ya conectados.

Veamos ahora cada punto con detenimiento.
En primer lugar, el punto 1 se refiere a la conexión de los clientes con el servidor. Para ello necesitaremos un mecanismo de transmisión que permita la comunicación entre ambas partes. Utilizaremos la interfaz socket.

El servidor dispondrá de dos sockets. Uno de ellos se utilizará para aceptar las conexiones con los clientes (socket pasivo), que estará permanentemente en modo de escucha; y el otro será para la interacción con los clientes (socket dinámico).

El funcionamiento de los sockets en el servidor se puede ver en el diagrama siguiente:

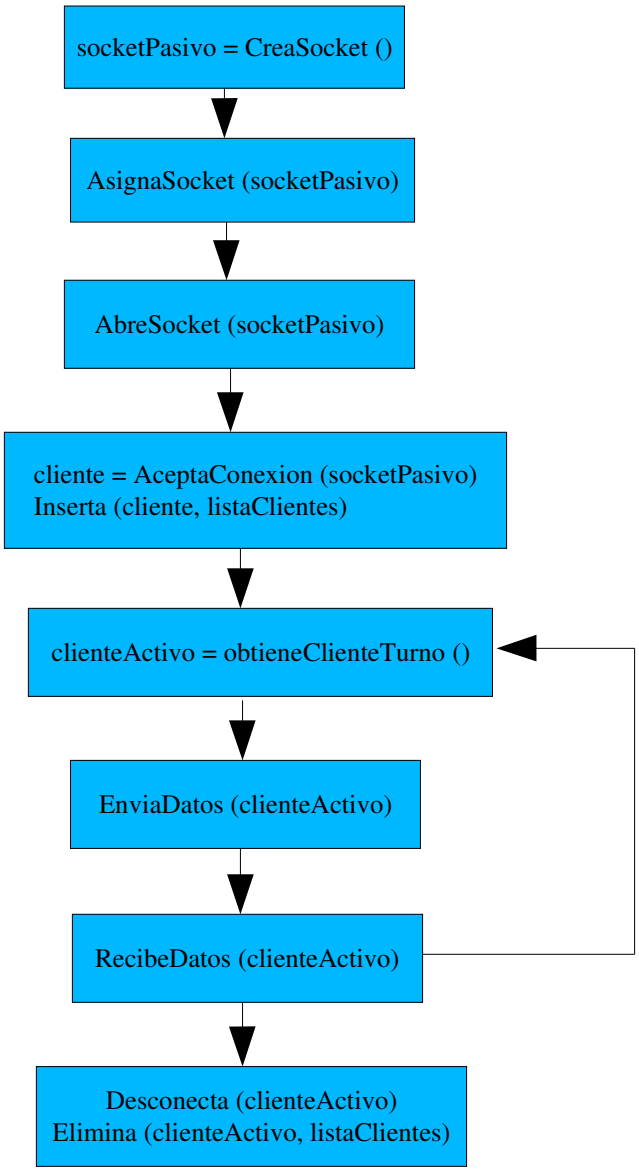


Fig. 5. Diagrama de funcionamiento de los sockets en el servidor

Si observamos el diagrama podemos ver lo siguiente:

Inicialmente, se crea el socket pasivo, que como ya hemos dicho, es el que hace que el servidor esté continuamente escuchando. Posteriormente, se le asigna a dicho socket los parámetros necesarios para la conexión, como por ejemplo, la dirección IP, el puerto y la familia de protocolos correspondientes. Seguidamente, se abre dicho socket en modo escucha, y se acepta en un nuevo socket la conexión con el cliente. Dicho cliente se inserta en la lista de clientes conectados. A continuación, se obtiene el cliente al cual le toca el turno, y para dicho cliente se realiza la interacción. Finalmente, el cliente se desconecta y se elimina de la lista de clientes conectados.

Este sería el esquema del funcionamiento de los sockets en el servidor. Para implementar estas rutinas de forma sencilla se utilizará la clase ConexionServidor que incluirá en cada método los pasos anteriores.

Pasemos a continuación al punto 2.

Una vez que se ha aceptado un cliente, el servidor le asociará a dicho cliente una serie de atributos para definirle un personaje. Dichos atributos serán las características de rol ya comentadas.

Seguidamente, se insertará el nuevo personaje en una lista de clientes conectados. Los clientes de dicha lista serán ordenados de menor a mayor en función del valor del momento de reacción (mmr) que tengan asignados. Para obtener el cliente activo, es decir, el cliente al que le toque el turno, se obtendrá el módulo entre la variable turno y el tamaño de la lista de conectados. Dicho valor nos devolverá la posición del cliente en la lista. La variable turno se irá incrementando en cada iteración.

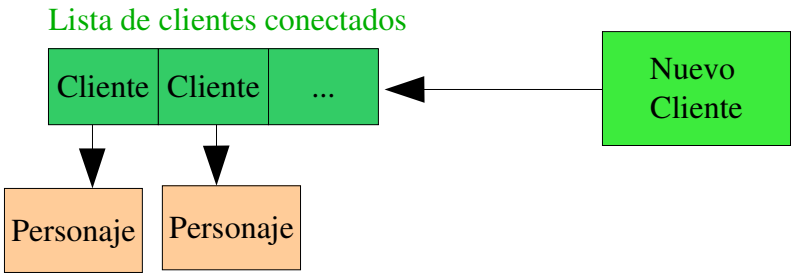


Fig. 6. Diagrama de la lista de clientes conectados

Ya que se ha explicado cómo se conecta el cliente con el servidor, pasamos a explicar el bucle de interacción con los clientes ya conectados.

Como ya hemos mencionado, los clientes no se deben de quedar bloqueados a pesar de que no les toque su turno. Constantemente deben de estar interactuando con el servidor, ya que si no los jugadores se quedarían inactivos. La idea consiste en que el servidor sólo acepte las peticiones del jugador activo obviando las del resto. De esta forma, cada cliente estaría en un bucle de transmisión

enviando y recibiendo información hasta que solicitase desconectarse. Para implementar dicho bucle se utilizarán hebras. Para cada jugador que se conecte al servidor se creará una hebra, y se ejecutará indefinidamente hasta que dicho jugador se desconecte del juego. Así se tendrán varias hebras ejecutándose simultáneamente (tantas como clientes) dándole al juego cierta concurrencia desde el punto de vista de estar siempre en comunicación con el servidor, no en el modo de realizar las acciones. Por ello, es necesario distinguir entre el sistema de juego (que será iterativo porque atenderá al cliente que le toque el turno) y el funcionamiento del servidor (que será concurrente, ya que atenderá a todos los clientes). Si el servidor siguiera el esquema del juego (iterativo), sólo se atendería a un cliente cada vez, y el resto, al no estar en continua interacción, se bloquearían.

Veamos ahora el funcionamiento de las hebras.

El primer paso en la hebra será enviar al cliente la información asociada a su radio de percepción para que éste pueda tomar las decisiones.

Según los cambios producidos por acciones anteriores, ya sean del propio jugador o del resto, el jugador necesitará refrescar una parte específica de su radio. Es decir, no siempre va a necesitar recibir toda la información, porque a lo mejor no se ha modificado todo su radio. Cada cliente tendrá almacenada la estructura del mundo que perciba, y según los cambios producidos la irá refrescando. Esta forma hace que los envíos de información sean más eficientes, ya que se disminuye la cantidad de datos transmitidos; sólo se envía lo necesario.

Para controlar qué parte enviar de información y que parte no, se utilizará una clase llamada Envio, formada por el socket del cliente y un flag que indique qué datos enviar. Se tendrá una lista de envíos de igual tamaño que la lista de clientes. El flag anteriormente indicado se irá modificando según los cambios que el cliente necesite percibir.

Una vez enviada la información del mundo al cliente, el servidor esperará recibir la solicitud del cliente, es decir, la acción que haya decidido realizar el jugador (punto 3). Para ello, el servidor comprobará si el cliente solicitante coincide con el cliente activo (punto 4). En tal caso (punto5), el servidor analizará la petición y modificará el mundo (punto 6).

Para cada cliente activo, el servidor utilizará un temporizador de 5 segundos. Este será el máximo tiempo de espera para recibir una decisión del jugador. En caso de que expirara dicho contador, se cedería el turno. El hecho de utilizar un temporizador es para que el juego no se quede eternamente esperando una respuesta de un jugador, ya que aunque el juego es por turnos, los tiempos de respuesta son "interactivos" en cierto modo, si no el resto de los jugadores se quedarían bloqueados en espera de la respuesta de un cliente.

Otro aspecto es que el turno de cada cliente puede englobar a varias acciones.
Hasta ahora hemos dicho que cada turno constituía una sola acción, y que una vez aceptada por el servidor, se cedía el turno a otro cliente. Esto no es del todo cierto, ya que en función del momento de reacción del personaje, un jugador podría ser capaz de realizar varias acciones en su mismo

turno. El temporizador contaría 5 segundos para cada una de estas acciones a realizar y sólo cuando todas las acciones se hubiesen realizado (o finalizase el tiempo total) se incrementaría el turno para el cliente siguiente.

Como podemos comprobar, puede haber acciones que no se realicen porque estén fuera del tiempo de respuesta, y por tanto sean obviadas por el servidor.

Cuando la acción solicitada por el cliente no sea válida (punto 7), el servidor no la tendrá en cuenta, y esperará una respuesta correcta hasta que la solicite el cliente o expire el temporizador.

Cuando la acción solicitada sea de desconexión, el cliente será eliminado de la lista de clientes conectados y de la lista de envíos. Además se reordenarán la lista de clientes conectados y se recalculará el próximo cliente activo. Ésta acción será la única que no se requerirá realizar dentro del turno, es decir, un cliente podrá desconectarse siempre que desee aunque no sea su turno.

Como resumen veamos el algoritmo básico que seguirá el servidor:

```
sockPasivo = ConexionServidor ()
asignaConexion (sockPasivo)
abreConexion (sockPasivo)

turno = 0
listaClientes = []
listaEnvios = []

Repetir:

    sockCliente = aceptaCliente (sockPasivo)
    nuevoPersonaje = creaPersonaje (sockCliente)
    inserta (nuevoPersonaje, listaClientes)
    ordena (listaClientes)
    inserta (sockCliente, listaEnvios)
    actualiza (listaEnvios)

    sockTurno = obtieneClienteTurno (listaClientes, turno)

    creaHebraCliente ()
```

El algoritmo de la hebra será el siguiente:

```
finHebra = 0
Mientras (finHebra == 0) Repetir:

    mundo = obtieneRadioPercepcion (sockCliente)
    envia (mundo, sockCliente)

    accionSolicitada = recibe (sockCliente)

    Si (sockCliente == sockTurno):

        Si (temporizador == 5):
            turno++
            sockTurno = obtieneClienteTurno (listaClientes, turno)

        Si (valida (accionSolicitada)):

            Si (accionSolicitada == desconexion):

                desconectaCliente (sockCliente)

                elimina (sockCliente, listaClientes)
                ordena (listaClientes)
                sockTurno = obtieneClienteTurno ()

                elimina (sockCliente, listaEnvios)
                actualiza (listaEnvios)

                finHebra = 1

            Si no:
                modificaMundo (accionSolicitada)
                actualiza (listaEnvios)
                turno++
                sockTurno = obtieneClienteTurno (listaClientes, turno)
```

3.2 Protocolo de comunicación

Ya que hemos visto el funcionamiento del servidor vamos a describir el protocolo de comunicación entre el servidor y el cliente. Es decir, las reglas que siguen ambos programas para la transmisión e interpretación adecuada de los datos.

Para que la transmisión sea lo más sencilla y eficiente posible utilizaremos como dato de transmisión el tipo entero. Cliente y servidor se comunicarán entre sí en base a números enteros.

Como las estructuras de datos del mundo consistirán en listas de enteros, dichas listas se enviarán elemento a elemento. Para saber cuál es el tamaño de cada lista se enviará un entero indicando dicho tamaño. De esta forma, para cada lista se enviará un entero para que el destino sepa el número de elementos que va a recibir. Posteriormente, se enviará cada uno de dichos elementos. El receptor tendrá que reconstruir cada lista con los datos recibidos.

Las acciones también vendrán indicadas con números enteros, y según su valor el servidor interpretará que el cliente solicita realizar una acción u otra.

Los resultados de las acciones se convertirán en cambios en las estructuras de datos del mundo, y serán "visibles" al cliente cuando éste reciba dichas estructuras asociadas a su radio de percepción.

Otro aspecto importante para ponerse de acuerdo el cliente y el servidor es la información que se necesita enviar. Como ya comentamos en la subsección anterior, tendremos una lista de envíos donde se especificará en cada caso qué cliente necesita qué información. De este modo, antes de enviar ninguna lista del mundo, el servidor tendrá que enviarle al cliente el flag de envío, para que el cliente sepa qué datos va a recibir.

También será necesario enviar al cliente el tamaño del mundo, el socket asociado al propio cliente, y el turno, es decir, el cliente activo, para que sepa si le toca jugar o no en ese momento.

Como vemos, el servidor enviará al cliente el tamaño del mundo, el flag de envío, el socket del cliente y del cliente activo en ese momento, y las listas de datos del mundo asociadas al radio de percepción. A su vez, recibirá del cliente la acción que haya decidido éste realizar. Dependiendo del tipo de acción, el cliente enviará información adicional asociada, como por ejemplo, una posición en el mundo, ya que habrá acciones que necesiten saber unas coordenadas.

El protocolo se irá repitiendo para cada cliente cada vez.

En el diagrama siguiente podemos ver el protocolo de comunicación:

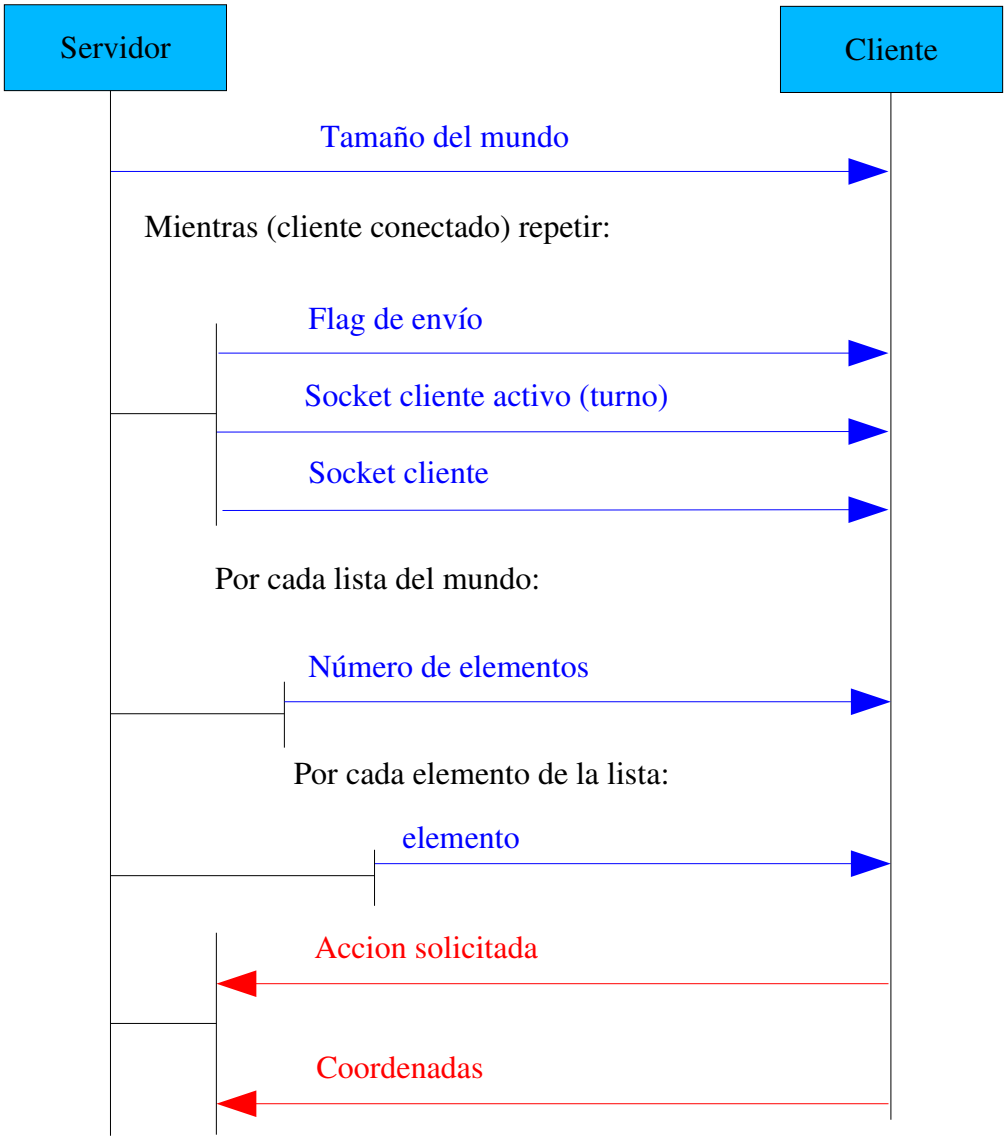


Fig. 7. Diagrama del protocolo de comunicación cliente/servidor

Veamos detenidamente el protocolo anterior.

Servidor->Cliente: Tamaño del mundo
El tamaño del mundo vendrá determinado por seis valores:
- Límite inferior del mundo exterior
- Límite superior del mundo exterior
- Límite inferior del interior de los castillos
- Límite superior del interior de los castillos
- Límite inferior del interior de las casas
- Límite superior del interior de las casas

Las casas harán referencia a los edificios asociados a las tiendas, bibliotecas y posadas. Todos ellos tendrán las mismas dimensiones.
Los límites inferiores y superiores se aplicarán tanto a la dimensión horizontal como a la vertical.

Servidor->Cliente: Flag de envío
Según el valor del flag se indicará qué listas del mundo se van a mandar.

Servidor->Cliente: Turno
En realidad, se enviará el socket del cliente al que le toque el turno. Este valor servirá para que el cliente lo compare con su propio socket y sepa si le corresponde jugar.

Servidor->Cliente: Socket del cliente
Se enviará el socket del cliente para que éste conozca su valor y pueda comparar lo anterior.

Servidor->Cliente: Número de elementos
Por cada lista del mundo enviada (asociadas a los personajes, edificios, objetos y casillas) se mandará, inicialmente, su tamaño.

Servidor->Cliente: elementos
Una vez enviado el tamaño de la lista correspondiente, el servidor mandará uno a uno todos los elementos de la lista. Por ejemplo, si la lista es de personajes, se enviarán al cliente uno a uno todos los personajes. Y por cada personaje, uno a uno todos sus atributos. El cliente tendrá que reconstruirlos.

Cliente->Servidor: Acción solicitada
El cliente, una vez tomada una decisión, enviará un entero asociado a la acción que desee realizar.
La correspondencia será la siguiente:

Movimiento del personaje	*acción: 1
Movimiento del cursor (ya se verá)	*acción: 2
Descansar/Gastar turno	acción: 3
Atacar	*acción: 4
Robar	*acción: 5
Comer setas	acción: 6
Comer miel	*acción: 7
Abrir cofre	*acción: 8
Entrenar/Aceptar misión	acción: 9
Liberar prisionero	*acción: 10
Estudiar	acción: 11
Descansar en Posadas	acción: 12
Comprar arma	acción: 13-16 (asociado a cada tipo)
Comprar escudo	acción: 17-19 (asociado a cada tipo)
Comprar armadura	acción: 20-23 (asociado a cada tipo)
Beber agua	acción: 24
Finalizar misión	acción: 25
Actuar con menús (ya se verá)	acción: 26

Cliente->Servidor: Coordenadas
Para las acciones marcadas con un (*) se necesitará enviar, además, las coordenadas del elemento con el que solicita actuar el cliente, o bien, las nuevas coordenadas del personaje del cliente, en caso de que se solicite movimiento.

3.3 Cliente

Veamos ahora el funcionamiento del cliente.
Como ya sabemos, en el proyecto hay dos tipos de clientes: los manejados por el usuario humano, y los manejados por el usuario ordenador. El servidor no distinguirá entre ambos a la hora de interpretar las acciones o de enviar/recibir información. Ambos tipos de jugadores serán tratados de la misma manera. La diferencia que existirá entre ambos radicará en la forma de tomar las decisiones y en el lenguaje de programación.

Los clientes PJs serán implementados, al igual que el servidor, en C/C++.
Los clientes PNJs serán implementados en Python.

Las decisiones de los PJs serán las que tomen los propios usuarios, y para ello, cada cliente humano necesitará una interfaz gráfica que sirva para visualizar el estado del mundo actual en ese momento. La interfaz no será más que una transformación visual de las listas de datos del mundo.
Las acciones que soliciten los clientes serán los eventos que se produzcan en dicha interfaz y se transformarán en enteros que se enviarán al servidor.

Las decisiones de los PNJs vendrán determinadas por un análisis de la situación, que se realizará en base a un comportamiento inteligente, y las acciones solicitadas serán los resultados de dicho análisis, que también se convertirán en números enteros para enviar al servidor.

Ambos tipos de clientes utilizarán un socket para conectarse al servidor, y les servirá dicho socket además para el intercambio de información.

Una ventaja de utilizar sockets como interfaz de comunicación es su independencia frente al lenguaje de programación. El que el servidor esté implementado en C/C++ y que el PNJ lo esté en Python no presenta ningún problema a la hora de transmitir datos. Si en vez de tratar a los PNJs como clientes se trataran como módulos Python, las transmisiones serían en realidad llamadas a funciones y habría que realizar una serie de conversiones de datos de C a Python y viceversa. Lo cual entraría en contradicción con lo dicho anteriormente; los jugadores no serían tratados igualmente. Habría "verdaderos" clientes, que serían los PJs, y otros jugadores que no serían clientes, si no módulos externos.

También es importante saber que los clientes, ya sean PJs o PNJs, no podrán conocer más que lo que les envíe el servidor. Por ejemplo, un jugador1 no podrá saber qué está haciendo un jugador2 si no está en el radio de percepción del jugador1. Sólo se conocerá lo que entre en el radio. Los jugadores no podrán comunicarse entre sí por sí solos, sólo podrán a través del servidor que hará de mediador. Por ejemplo, un jugador1 podrá atacar a un jugador2 sólo si el servidor acepta dicha solicitud.

Otro dato importante es que, como ya hemos comentado en secciones anteriores, cada personaje tendrá una serie de características propias. Dichas características serán ocultas para el resto de los personajes. Por ejemplo, un personaje no podrá saber cuanta fuerza tendrá otro personaje (aunque se podría suponer por el resultado de los ataques).
Esta información será confidencial para cada personaje, ya que en caso contrario, sobre todo con los PNJs, se tendría toda la información disponible para la toma de decisiones, y éstas serían siempre las óptimas dejando al juego sin incertidumbre, y por tanto no hablaríamos de agentes.

De esta forma, cada personaje constituirá una caja negra para el resto, y sólo será transparente para el servidor.

En realidad, el servidor, como Master del juego, es el que tendrá toda la información y la filtrará para cada jugador (cliente). De otra forma, el jugador sabría lo mismo que el Master y se rompería el sistema del juego de rol.

Al igual que explicábamos en el servidor, veamos el proceso que se realizará en el cliente:

1. El cliente se conecta al servidor.

2. El cliente recibe las estructuras de datos.

3. El cliente toma una decisión.

4. El cliente envía como petición al servidor la decisión tomada.

Veamos cada punto independientemente.

La conexión con el servidor (punto 1) se realizará en base a los sockets. El cliente creará un socket y a través de él, especificando la dirección IP correspondiente, se conectará al servidor.

A diferencia del servidor, el cliente sólo utilizará un único socket, que le servirá para la transmisión de los datos, y que tendrá que cerrar cuando el servidor indique su desconexión. De este modo, mientras el cliente terminará su ejecución en algún momento, el servidor será un programa permanentemente en funcionamiento (sólo se terminaría la hebra asociada al cliente) que o bien estará interactuando con los clientes conectados, o bien estará en modo de espera, en caso de que no haya ninguna conexión.

El funcionamiento de los sockets en el cliente podemos verlo en el diagrama siguiente:

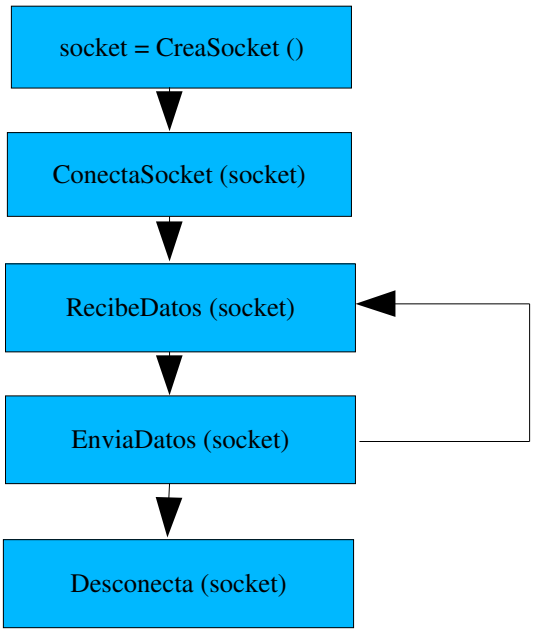


Fig. 8. Diagrama de funcionamiento de los sockets en el cliente

Para implementar el socket se utilizará una clase similar a la que se utilizaba en el servidor, pero en este caso orientada al cliente. La clase se denominará conexionCliente.

Los puntos 2-4 del proceso estarán dentro del bucle de interacción con el servidor.

La recepción de las estructuras de datos (punto 2) se realizará como ya se comentó en el protocolo de comunicación. El cliente recibirá una serie de elementos que tendrá que reconstruir en las listas correspondientes.

La toma de decisiones (punto 3) se basará, como ya sabemos, en una interfaz gráfica o en una función de comportamiento inteligente, según el cliente en cuestión. Ambos apartados los analizaremos detalladamente en secciones posteriores.

Una vez tomada la decisión, el cliente enviará un número entero al servidor indicándole la acción deseada (punto 4).

Veamos ahora como resumen el algoritmo que utilizará el cliente.

Nótese que ahora incluimos los nuevos aspectos explicados.

```
socket = ConexionCliente ()
conectaAServidor (socket)

recibe (tamanoMundo, socket)

fin = 0
Mientras (fin == 0) Repetir:

        recibe (flagEnvio, socket)
        recibe (turno, socket)
        recibe (sockCliente, socket)

        Si (sockCliente == -1):
                fin = 1

        según (flagEnvio):
                recibe (mundo, socket)

        accion = tomaDecision (tamanoMundo, mundo, turno, sockCliente)

        envia (accion, socket)
```

3.4 Estructuras de datos

Una vez que hemos visto el funcionamiento de la arquitectura cliente/servidor pasaremos a describir las estructuras de datos.

El mundo imaginario del juego estará formado por personajes, objetos, edificios y terrenos.
Cada uno de estos elementos tendrá a su vez una serie de campos indicando sus características o propiedades.

La forma más práctica de tener almacenados dichos elementos es utilizando listas. De este modo, tendríamos una lista para los personajes, otra para los objetos, otra para los edificios, y finalmente, otra para las casillas o terrenos. Todos los elementos estarían indexados en la lista de datos correspondiente.

Las listas serán la estructura más utilizada en el proyecto. En secciones anteriores ya hemos hablado de una lista de clientes conectados y de una lista de envíos. Como vemos, tendremos listas de elementos de distintos tipos, por lo que necesitaremos un tipo lista paramétrico, cuyos elementos puedan ser diferentes.

A continuación, describiremos cada una de las listas de elementos mencionadas.

En primer lugar, tendremos una lista para almacenar los personajes. Dicha lista coincidirá con la de los clientes conectados, ya que como sabemos, un cliente es en realidad un personaje.

Cada personaje será una estructura con los siguientes campos:

- **sock:** Socket del personaje (creado en el servidor cuando se conecta el cliente)
- **esp:** Especie a la que pertenece el personaje
- **fue:** Fuerza del personaje
- **con:** Constitución del personaje
- **tam:** Tamaño del personaje
- **inte:** Inteligencia del personaje
- **des:** Destreza del personaje
- **pg:** Puntos de golpe del personaje
- **ene:** Energía del personaje
- **ham:** Hambre del personaje
- **sed:** Sed del personaje
- **maxpg:** Valor máximo de los puntos de golpe del personaje
- **maxene:** Valor máximo de la energía del personaje
- **maxham:** Valor máximo del hambre del personaje

- **maxsed:** Valor máximo de la sed del personaje
- **mh1:** Modificador de la habilidad 1 del personaje (sigilo)
- **mh2:** Modificador de la habilidad 2 del personaje (percepción)
- **mmr:** Momento de reacción del personaje
- **posx:** Posición x del personaje en el mundo
- **posy:** Posición y del personaje en el mundo
- **edif:** Identificador del edificio donde se encuentra el personaje
- **estu:** Número de veces que haya estudiado el personaje
- **entr:** Número de veces que se haya entrenado el personaje
- **obj:** Objetivo o misión que tenga encomendada el personaje
- **nturnos:** Número de turnos del personaje (número de acciones que pueda realizar por turno)
- **equ**: Equipo o inventario del personaje

El campo sock será utilizado como identificador del personaje.

El campo mmr será el valor empleado para la ordenación de la lista de clientes conectados. Aquel personaje que tenga menor valor de momento de reacción tendrá preferencia en el turno.

El campo nturnos será el que indicará las acciones por turno que tendrá disponible el personaje. Su valor se calculará en función del campo mmr. A menor mmr, mayor número de acciones.

Todos los campos, como ya dijimos en la sección del cliente, se representarán mediante valores enteros y serán confidenciales para cada personaje, siendo controlados por el servidor.

La estructura personaje, además de las estructuras objeto, edificio y casilla vendrán definidas en forma de clases.

La lista de objetos vendrá definida por objetos con los siguientes campos:

- **id:** Identificador del objeto
- **tipo:** Tipo de objeto
- **visi**: Visibilidad del objeto (si el objeto es visible o no)
- **posx:** Posición x del objeto en el mundo
- **posy:** Posición y del objeto en el mundo
- **longx:** Longitud horizontal del objeto
- **longy:** Longitud vertical del objeto
- **casilx:** Número de casillas que ocupa el objeto en el eje x
- **casily:** Número de casillas que ocupa el objeto en el eje y
- **edif:** Identificador del edificio donde se encuentra el objeto
- **equ:** Tipo de subobjeto que contenga el objeto en caso de tratarse de un objeto contenedor

Los objetos serán todos los elementos estáticos del mundo, que o bien serán modificables por los jugadores, o bien serán inalterables frente a las acciones de los personajes.

La siguiente lista que vamos a ver es la constituida por edificios cuyos atributos serán:

- **id:** Identificador del edificio
- **tipo**: Tipo de edificio
- **llav**: Llave del edificio (si el edificio necesita una llave para entrar o no)
- **posx:** Posición x del edificio en el mundo
- **posy:** Posición y del edificio en el mundo
- **longx:** Longitud horizontal del edificio
- **longy:** Longitud vertical del edificio
- **casilx:** Número de casillas que ocupa el edificio en el eje x
- **casily:** Número de casillas que ocupa el edificio en el eje y

Los edificios serán también elementos estáticos, pero a diferencia de los objetos, permitirán la entrada a nuevos escenarios (submundos interiores).

La última lista que vamos a ver es la asociada a las casillas o terrenos.
Cada casilla tendrá los atributos siguientes:

- **tipo**: Tipo de la casilla
- **posx:** Posición x de la casilla en el mundo
- **posy:** Posición y de la casilla en el mundo

Los campos posx, posy serán los identificadores para cada casilla, ya que en realidad una casilla es una coordenada del mundo que tendrá asociada un tipo de terreno.

Como podemos observar, las listas guardan relación en algunos campos, es decir, hay atributos que coinciden con los identificadores de los elementos de otras listas. Es la forma de conectar las listas entre sí para definir el mundo. Veamos cada caso:

Cada personaje tendrá una determinada posición en el mundo, por tanto necesitará estar enlazado con alguna casilla. Además, su posición podrá pertenecer tanto al mundo exterior como al interior de algún edificio, por lo que también estará asociado a un elemento de la lista de edificios.

Cada objeto también estará localizado en una determinada posición en el mundo, ya sea en el exterior o en el interior de algún edificio, por tanto, también estará enlazado con algún item de la lista de edificios y casillas.

Por su parte los edificios, también necesitarán estar localizados, así que estarán asociados a una determinada casilla que indicará la posición del vértice inferior izquierdo del edificio.

En caso de que algún personaje u objeto estuviese en el exterior, el campo asociado a la lista de edificios sería nulo.

El mundo puede verse, de esta forma, como una macroestructura formada por distintas listas conectadas entre sí y asociadas a cada tipo de elemento que pueda haber.

Gráficamente, las listas de datos se enlazarán de la siguiente forma:

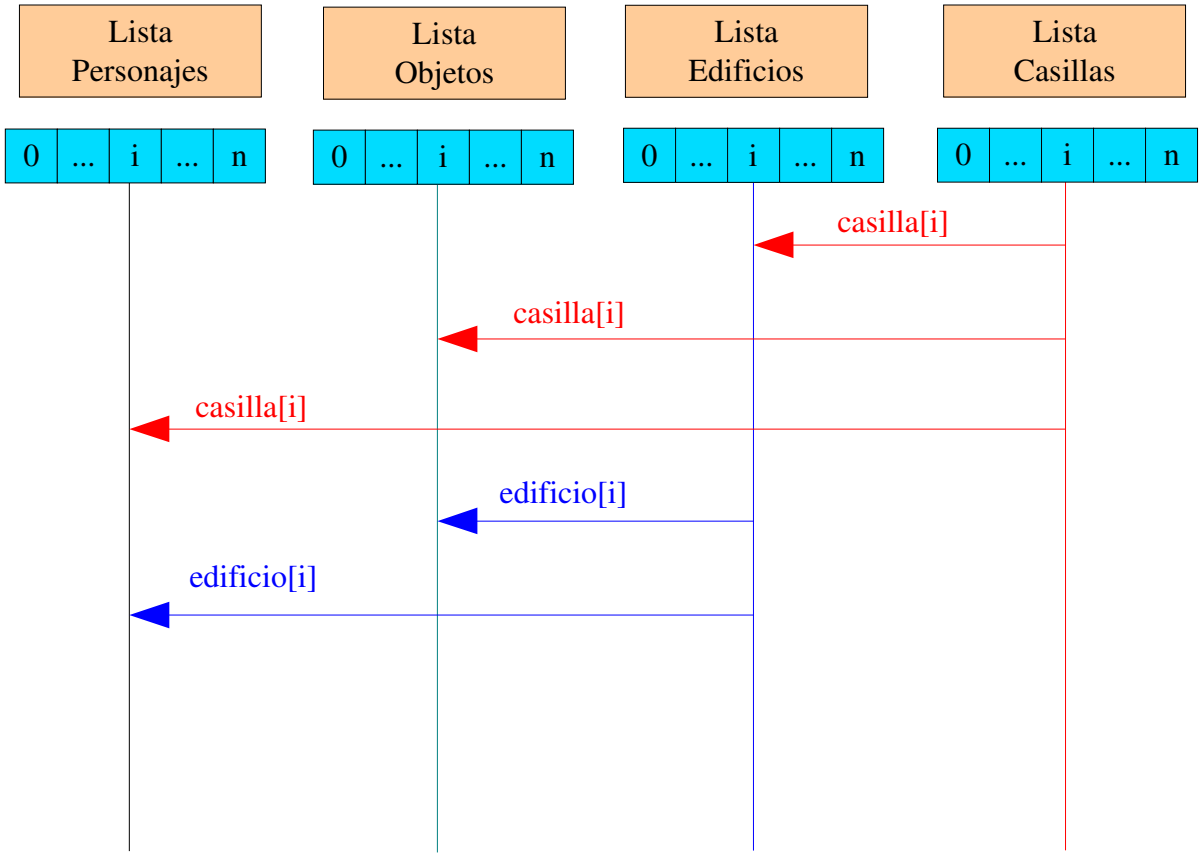


Fig. 9. Relación entre las listas del mundo

En la sección del protocolo de comunicación se habló a cerca de una lista de envíos, que determinaba para cada cliente qué listas de datos eran necesarias o no para la transmisión. Ahora que conocemos las listas de datos empleadas, podemos comentar los valores que podrá tomar el flag de envío.

Si el flag está a 0, no se enviará ninguna lista de datos.
Si el flag está a 1, se enviará la lista de personajes y la lista de objetos.
Si el flag está a 2, se enviarán todas las listas: personajes, objetos, edificios y casillas.

Esta división obedece a las modificaciones asociadas a las acciones de los jugadores. Habrá, fundamentalmente, dos tipos de acciones: aquellas que supongan un desplazamiento del radio de percepción, lo cual implicaría enviar todas las listas (flag de envío a 2); o bien acciones que modifiquen elementos dentro del mismo radio de percepción, lo cual implicaría enviar las listas correspondientes a los elementos modificados (flag de envío a 1).

Los valores asociados a los objetos, edificios y casillas se obtendrán a partir de unos ficheros de datos. El servidor se encargará de cargar en las listas asociadas dichos elementos en función del radio de percepción del cliente. Sólo los valores de los personajes serán creados sin utilizar ningún fichero de texto. La explicación es bien sencilla, todos las casillas, edificios y objetos constituyen un mundo estático, por lo cuál ya estarán creados en los ficheros de datos mencionados, mientras que los personajes necesitarán crearse sobre la marcha, ya que serán elementos dinámicos que surgirán cuando se conecte cada cliente. De este modo, tendremos un fichero de texto asociado a los objetos, otro asociado a los edificios, y otro asociado a las casillas, lo cuál viene a decirnos que el mundo se tendrá almacenado a priori en ficheros de texto y no en memoria. Sólo se cargará lo necesario leyendo de dichos ficheros. Esto hace que se utilicen los recursos de memoria más eficientemente.

Los ficheros de edificios y casillas no se modificarán nunca en el transcurso del juego, ya que no es normal que se alteren dichos elementos. Sin embargo, sí se modificarán los objetos, por lo que cuando una acción implique como resultado el variar algún objeto, el servidor tendrá que modificar dicho objeto en cuestión en el fichero de datos, para que la próxima vez que se lea ya aparezca modificado.

Las dimensiones del mundo también se leerán de los ficheros, en concreto, del fichero de casillas. Será el servidor el encargado de leer dichos límites para enviarlos a los clientes.
Las dimensiones serán las asociadas al mundo exterior, y a los mundos interiores.

Las modificaciones de los objetos se realizarán buscando en el fichero el objeto en cuestión e insertando sus nuevos valores.

Los ficheros de datos se crearán utilizando un programa Python. De este modo, se podrá modificar la definición del mundo de forma cómoda, sin tener que modificar o recompilar el resto del proyecto. Al igual que con el comportamiento inteligente de los PNJ, la definición del mundo será independiente.

Veamos cada fichero de datos independientemente.

Fichero edificios.dat
Se creará a partir de un programa Python que irá insertando en el fichero uno a uno todos los edificios con sus correspondientes atributos.
El fichero presentará un conjunto de edificios, estructurados por líneas. De modo que cada línea de texto se corresponderá con los atributos de un edificio. El formato será el siguiente:

<id> <tipo> <llav> <posx> <posy> <longx> <longy> <casilx> <casily>
...

Los atributos *posx, posy* serán aleatorios.

Fichero objetos.dat
Se creará en el mismo programa Python y seguirá una estructura similar: cada línea de texto se asociará a cada objeto. El formato será el siguiente:

<id> <tipo> <visi> <posx> <posy> <longx> <longy> <casilx> <casily> <edif> <equ>
...

Al igual que antes, los atributos *posx, posy* serán aleatorios.

Fichero casillas.dat
También se creará en el mismo programa Python. Primeramente, se insertará en el fichero las dimensiones del mundo e interiores. Seguidamente, se insertará línea a línea la información asociada a cada casilla. El formato será el siguiente:

<mundoInf> <mundoSup> <castilInf> <castilSup> <casaInf> <casaSup>
<tipo> <i> <j>
...

El atributo *tipo* será de tipo aleatorio entre (hierba, tierra, piedra, poza de agua).
Los atributos *i, j* se corresponderán con las coordenadas de la casilla y sus valores irán iterados de forma ordenada dentro de los límites del mundo externo. Es necesario indicar, que este fichero sólo se referirá a las casillas del mundo exterior. Las casillas de los mundos interiores no hará falta crearlas como tales. Ya veremos esto con más detalle en las secciones posteriores.

Por último, los personajes al ser dinámicos no necesitarán estar almacenados en ningún fichero. Cada vez que se conecte un cliente al juego, el servidor le asignará un personaje calculando sus atributos como ya comentamos en la sección de características de rol.

Para implementar las tiradas de dados utilizaremos números aleatorios generados en base a una semilla basada en el temporizador del sistema. De este modo, su valor siempre será distinto, y permitirá obtener valores diferentes cada vez.

Por último, vamos a comentar cómo se obtiene el radio de percepción.

Para cada cliente, el servidor generará una posición aleatoria en el mundo exterior. A partir de dicha posición como punto central se definirá un intervalo de casillas, que coincidirá con el radio de percepción del cliente. Todos los elementos que estén dentro de ese radio serán los que enviará el servidor.

4 Gráficos

En esta sección vamos a explicar la interfaz gráfica del proyecto, detallando tanto la forma de dibujar el mundo, como el modo de interpretar las acciones del usuario PJ.
Primeramente veremos cómo visualizar las estructuras de datos del mundo de forma genérica, para luego centrarnos en el dibujo de las casillas, objetos, personajes, edificios, e interiores.

4.1 Visualización del mundo

Como ya sabemos, la interfaz gráfica es la forma de visualizar las estructuras de datos del mundo asociadas a un personaje PJ.

Cada elemento del mundo tendrá unas determinadas coordenadas *x* e *y* que lo localizarán en un punto específico. Sin embargo, toda la representación gráfica se basará en un mundo tridimensional. De este modo, nuestras coordenadas 3D serán realmente:

x = x”
y = z”
z = y”= 0

donde *(x, y, z)* son las coordenadas de nuestro mundo, y *(x”, y”, z”)* son los ejes de referencia.

Como vemos, nuestra coordenada *x* coincidirá con el eje *x”* (representará la longitud horizontal); nuestra coordenada *y* coincidirá con el eje de profundidad *z”* (representará la distancia o profundidad); y nuestra coordenada *z* coincidirá con el eje *y”* (representará la longitud vertical).

Como nuestra *z* (asociado al eje de referencia *y"*) será igual a 0, nuestro mundo se dibujará sobre dicho plano que representará el nivel del suelo. De este modo, a partir de ahora hablaremos de que los elementos se posicionaran en unas coordenadas (x,y), ya que la coordenada z sería 0.

El mundo se podría ver como un mundo dividido en un determinado número de casillas, donde cada una de ellas sería una coordenada (x,y) que se asociaría a la posición de cada elemento. Por ello, como ya vimos en la sección anterior de estructuras de datos, las listas de elementos estaban enlazadas con la lista de casillas.

Además, el mundo estará limitado, luego tendrá un determinado número de casillas tanto para *x* como para *y*. Los personajes sólo podrán desplazarse dentro de estos límites.

La interfaz gráfica se dividirá fundamentalmente en dos procesos:

1. Dibujado del mundo
2. Interacción con el usuario (solicitud de acciones)

La primera parte se desarrollará utilizando las funciones gráficas de la librería OpenGL y SDL_image. La segunda parte, asociada al manejo de eventos, se desarrollará utilizando la librería SDL.

En primer lugar, necesitaremos crear una ventana que será nuestra aplicación. En dicha ventana se dibujará la parte del mundo asociada al radio de percepción del cliente PJ (1). Por cada usuario que se conecte se creará una de estas ventanas.

En ella se gestionarán una serie de eventos, que serán las acciones que solicite el usuario a través del teclado (2). Dichos eventos asociados a las acciones se enviarán al servidor, y éste, en caso de aceptarlos, modificará las estructuras pertinentes, de modo que el usuario verá la modificación en pantalla.

Los eventos que se tendrán en cuenta, principalmente, para la ventana, serán los siguientes:

- **Evento expose**
  Controlará el refresco de la ventana, es decir, cada vez que la ventana sea expuesta se redibujará su contenido.

- **Evento de redimensionado**
  Controlará el redimensionamiento de la ventana, de forma, que cada vez que se modifique su tamaño, se adaptará la visualización a las nuevas dimensiones de la ventana.

- **Evento de pulsar tecla**
  Controlará la tecla que haya pulsado el usuario. Éste será el evento asociado a las acciones. En función de su valor, el cliente solicitará una petición u otra, que el servidor analizará.

- **Evento de cierre**
  Controlará el cierre de la ventana, y por consiguiente, solicitará una petición de desconexión al servidor. (Aunque también se podrá utilizar una determinada tecla para el cierre)

Para la visualización del mundo, se realizará un recorrido por cada lista dibujando cada elemento. Cada vez que el flag de envío sea mayor que 0 (se reciben datos modificados) se redibujará el mundo en la ventana.

En caso de que el usuario no realice ninguna acción, se enviará al servidor una acción por defecto que no implicará modificación alguna, por lo que no habría redibujado. De este modo, la interacción cliente/servidor será continua.

Siempre se redibujarán las listas de datos que tenga almacenadas el cliente, ya sean modificadas por los nuevos valores recibidos o no, por tanto, siempre habrá un "mundo" dibujable aunque no se reciba información nueva.

La parte principal de la visualización vendrá definida por el bucle de eventos denominado manejador de eventos de la ventana. Dicho bucle estará siempre ejecutándose hasta que el usuario solicite el cierre de la aplicación.

En el bucle constantemente se leerán los eventos y se enviará/recibirá información con el servidor dibujando los cambios en el contenido de la ventana.

Veamos a continuación un esquema del algoritmo del PJ, que será una especificación del que vimos en la sección del cliente:

```
socket = ConexionCliente ()
conectaAServidor (socket)

recibe (tamanoMundo, socket)

fin = 0
Mientras (fin == 0) Repetir:

        recibe (flagEnvio, socket)
        recibe (turno, socket)
        recibe (sockCliente, socket)

        Si (flagEnvio > 0):

                recibe (listasMundo, socket)
                dibujaMundo (listasMundo)
```

```
Mientras (haya evento) Repetir:
        Según (evento):
                caso "Expose":
                        inicializaVentana ()
                        dibujaMundo (listasMundo)

                caso "Resize":
                        inicializaVentana (nuevoTamaño)
                        dibujaMundo (listasMundo)

                caso "Tecla":
                        Si (tecla == "cierre"):
                                cerrarAplicacion ()
                                fin = 1
                        Si no:

                                accion = obtieneAccion (tecla)

                                Si (turno == sockCliente):
                                        envia (accion, socket)
                                Si no:
                                        envia (accionPorDefecto, socket)

                caso "Cierre":
                        cerrarAplicacion ()
                        fin = 1

Si (no hay evento):
        envia (accionPorDefecto, socket)
```

Como ya sabemos, el radio de percepción de cada personaje coincidirá con el número de casillas que ocupe la ventana, y los elementos definidos en este rango será lo que se dibujará.

La posición del personaje siempre será central en la ventana, pudiéndose ver tanto casillas a izquierda-derecha como delante-atrás.

El radio irá desplazándose a cada movimiento del personaje. De este modo, se irá descubriendo más parte del mundo en la dirección en la que se mueva dicho personaje. Será como hacer un scroll del mundo a cada paso.

Cada movimiento supondrá un desplazamiento en una casilla ya sea dirección izquierda, derecha, delante o atrás.

Utilizaremos una perspectiva isométrica para visualizar el mundo (ángulo de cámara a 45 grados), de esta manera, en vez de ver el mundo plano desde arriba, lo visualizaremos con una cierta inclinación.

Como ya sabemos, los elementos a dibujar serán los personajes, objetos, edificios y casillas. Y serán o bien rectángulos (casillas, personajes y objetos), o bien cubos (edificios).

A cada rectángulo se le asociará una textura, que en realidad, será el elemento en sí. Es decir, a cada rectángulo se le aplicará una determinada textura con la imagen del elemento en cuestión.

Para los cubos, cada cara del mismo tendrá asociado una textura.

Los rectángulos se dibujarán indicando sus vértices. Para los cubos se tratará cada cara como un rectángulo independiente a dibujar. Además, tendrá que asociarse una correspondencia entre los vértices de los rectángulos y los vértices de la textura (texels), para ajustar la textura al rectángulo (texelización).

Las imágenes de los rectángulos tendrán activo el canal alpha (imágenes RGBA), necesario para definir el fondo de la imagen como transparente. Esto será útil, porque de este modo, se dibujará el objeto de la imagen opaco, dejando transparente la parte de la imagen no asociada al objeto, es decir, el fondo. El efecto quedará como si se visualizase un objeto y no una imagen. El canal alpha es el canal que define la transparencia de los objetos [Foley].

Las texturas de los edificios serán imágenes RGB. No tendrán transparencias.

El proceso de visualización en OpenGl puede definirse con el siguiente esquema [Foley]:

Fig. 10. Esquema de OpenGL

Los datos, es decir, cada elemento a dibujar, se compondrá de un conjunto de píxels (puntos en pantalla). El proceso de pixelización consistirá en dibujar dichos píxels realizando un barrido del elemento. Seguidamente, como cada píxel tendrá una determinada profundidad (recordemos que el mundo será definido en 3D), necesitaremos un cierto orden de visualización. Para ello, utilizaremos el buffer de profundidad (Z-buffer) que analizará la posición de cada píxel. Cuando un píxel tenga

asociada una profundidad menor o igual que los píxels anteriores se incluirá en el Frame-buffer (buffer de visualización). Posteriormente, se aplicará la transparencia (en caso de tener activo el canal alpha) al píxel realizando una serie de operaciones de superposición de los colores RGB. Finalmente, el último paso será volcar los datos del buffer de visualización a pantalla.

Como hemos podido comprobar, OpenGL utilizará dos buffers; uno para el testeo de profundidad, que indicará que píxels dibujar, y otro para dibujar en pantalla los píxels.

Un problema asociado a la profundidad es el siguiente:

Supongamos que tenemos dos objetos, uno transparente y otro opaco, de forma que el opaco esté a mayor profundidad que el transparente, y que además, ambos objetos se solapan.

Si se dibuja primero el objeto transparente, debido al orden establecido por el Z-buffer, no se dibujarían los píxels del objeto opaco que quedaran tapados. Esto sería incongruente con la característica de la transparencia, ya que tendrían que verse dichos píxels.

Una solución para ello sería desactivar el Z-buffer para dibujar el objeto opaco. Otra solución sería dibujar siempre los objetos opacos primero.

Un caso similar sería en vez de tener un objeto transparente y otro opaco, que los dos fueran transparentes. El problema reside en que las operaciones que realizan la transparencia no son conmutativas. Es decir el orden de los colores RGB puede producir resultados incorrectos al alterarse. Por ello, sería necesario ordenar los elementos a priori por orden de mayor a menor profundidad para después aplicarles el Z-buffer. Dicha solución también se puede aplicar al primer caso.

En el proyecto, optaremos por dibujar primero los objetos opacos (casillas, edificios), y dibujar después los objetos transparentes (personajes, y mayoría de objetos) siguiendo un orden de mayor a menor profundidad con respecto a la cámara.

Para esto último, tendremos que ordenar las listas de elementos (personajes y objetos) en orden de mayor a menor coordenada *y*, ya que en nuestro caso coincidirá con la distancia hacia la cámara (profundidad).

Como son dos listas las que habrá que ordenar, para realizar el ordenamiento de forma práctica, podríamos ordenar ambas listas en una lista auxiliar. Cada elementos de dicha lista sería una estructura con dos campos: uno indicando el tipo de elemento (personaje u objeto) y otro indicando su posición en la lista correspondiente. Para ello utilizaremos una clase denominada Elemento.

Cuando el servidor envíe al cliente las listas del mundo, añadirá también la lista de elementos.
De este modo, el cliente sólo tendrá que escanear dicha lista en orden, buscando los elementos en las listas originales para poderlos dibujar posteriormente.
En caso de que el cliente fuese un PNJ ignoraría dicha lista, ya que no necesita ningún interfaz gráfico.

Como ya hemos comentado, utilizaremos texturas en los rectángulos que se dibujen. Para crearlas usaremos una clase Textura que gestionará el tratamiento con imágenes. Dicha clase cargará en memoria una textura mediante la librería SDL_image.

A continuación podemos ver un ejemplo de visualización:

Fig. 11. Ejemplo de visualización del mundo

Otro apartado a comentar en la interfaz es que también se mostrarán menús de opciones. De esta manera, el cliente podrá cómodamente elegir la acción que desee realizar, así como seleccionar la especie de su personaje al principio del juego. Además, se podrá mostrar a lo largo de la partida un cuadro de características que contenga los atributos de rol del personaje.

El menú de comienzo se implementará de una forma parecida a la visualización del mundo. Es decir, se mostrará en la ventana de la aplicación y se gestionará mediante el manejador de eventos de la ventana. El evento de seleccionar una de las opciones posibles (selección de especie del personaje) es el que permitirá entrar en la visualización del mundo.

El menú de acciones y el cuadro de características se implementarán dibujando rectángulos con texturas RGBA, además se utilizará una librería para escribir fuentes en pantalla (librería glf).

Gráficamente:

Fig. 12. Menú de selección de la especie del personaje

Fig. 13. Menú de acciones (centro) y cuadro de características (derecha)

Todas las funciones de visualización de la interfaz se implementarán en una clase que gestionará todo el funcionamiento visual. Dicha clase de denominará EntornoG.

Llegados a este punto podemos resumir que la interfaz gráfica se basa en 3 secciones importantes: Gestión de la ventana de la aplicación mediante el control de los eventos, dibujado de cada elemento mediante rectángulos y cubos, y manejo de texturas con y sin canal alpha.

También estarían los menús, pero en realidad pueden ser derivaciones de los puntos anteriores con la única diferencia del empleo de texto (en el caso del menú de acciones y cuadro de características).

4.2 Visualización de las casillas

Para dibujar las casillas se hará un recorrido por la lista de casillas recibida dibujando cada una de ellas.

Cada casilla será en realidad un rectángulo de dimensión 1x1, que se posicionará en las coordenadas (x,y) asociadas a la casilla en cuestión.

Además, cada casilla será de un determinado tipo, por lo que utilizaremos una clase de textura diferente en función del tipo asociado.

La textura asociada a la casilla será del tipo RGB, ya que los terrenos no utilizarán imágenes con transparencias.

El funcionamiento será el siguiente:

Tendremos un bucle con tantas iteraciones como tamaño tenga la lista de casillas.
En cada iteración, obtendremos una casilla de la lista. Seguidamente, obtendremos las coordenadas de la casilla y el tipo. En función de éste último, buscaremos la textura correspondiente en la lista de texturas, y finalmente, llamaremos al método GL_poligono, indicándole tanto las coordenadas como las dimensiones y la textura. Este procedimiento se irá repitiendo hasta obtener un terreno de casillas completo a visualizar asociado al radio de percepción del personaje.

En realidad, no haría falta tener una lista de casillas como estructura del mundo si todas fueran del mismo tipo, ya que sólo bastaría con hacer un bucle de rectángulos aplicando la misma textura. Sin embargo, cuando hablamos de casillas pertenecientes a un mundo exterior, disponemos de un conjunto variado y aleatorio de tipos de suelo. Es decir, un terreno del exterior puede tener tanto hierba, como piedra, tierra o agua. Es por ello, por lo que sólo para el mundo exterior necesitaremos una lista de casillas. En los mundos interiores, las casillas serán todas del mismo tipo y no se necesitará leer sus valores de la lista. Cuando estemos dibujando interiores se obviará dicha lista.

Las casillas exteriores se habrán creado en el fichero de datos asociado, y tendrán definidas las coordenadas como valores iterados para cada pareja (x,y) dentro de los límites del mundo exterior. El tipo se habrá creado de forma aleatoria para obtener, como ya hemos dicho, un terreno variado.

Las casillas interiores se crearán dentro del mismo bucle anterior, definiendo también las coordenadas como valores iterados (0-max) pero dentro de los límites del recinto interior. Como tipo asociado se asignará el mismo en cada caso.

Tanto las casillas exteriores como las interiores tendrán las mismas dimensiones.

Al dibujar los rectángulos secuencialmente se obtendrá un efecto de suelo continuo.

Es necesario destacar que los límites serán diferentes en función del mundo. Es decir, cada mundo (exterior, interior tipoA, interior tipoB, ...) tendrán un tamaño distinto y por tanto sus dimensiones serán diferentes.

Los mundos interiores se asociarán al interior de cada edificio. De forma que cada edificio definirá un submundo.

Fig. 14. Ejemplo de suelo exterior (hierba, piedra, tierra, pozas de agua)

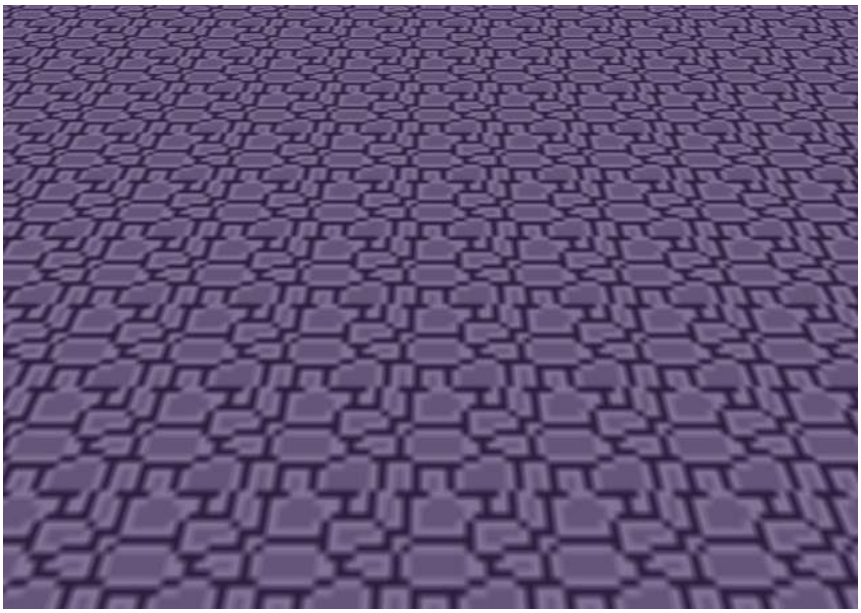


Fig. 15. Ejemplo de suelo interior (todas las casillas del mismo tipo)

4.3 Visualización de los objetos

Los objetos y los personajes se dibujarán de forma combinada en el orden que indique la lista de elementos. Se tendrá un bucle de tantas iteraciones como elementos tenga la lista. En cada iteración se obtendrá un elemento de la lista. En función de su tipo (personaje u objeto) se dibujará una cosa u otra.

A diferencia de las casillas, donde sólo se tenía en cuenta los atributos de posición y tipo, para los objetos se necesitará saber además el número de casillas (x,y) que ocupen.

Es necesario distinguir entre el tamaño de la textura y el tamaño del rectángulo. El primero viene definido como el ancho y alto de la imagen, mientras que el segundo viene definido por las casillas que ocupa. De este modo, tendremos objetos más o menos grandes en función del número de casillas.

Las texturas asociadas a los objetos serán tanto RGB como RGBA, ya que tendremos objetos con imágenes con y sin transparencias.

Además, los objetos podrán ser tanto rectángulos como cubos.

Los cubos tendrán asociadas dos tipos de texturas, normalmente una textura para la cara frontal y otra textura para las caras laterales.

Los objetos se corresponderán con todos aquellos elementos del mundo que no sean casillas, edificios y personajes dinámicos. Es decir, podrá haber personajes estáticos, como por ejemplo, vendedores, prisioneros, etc..., que no tendrán una participación directa en el juego, ya que no dispondrán de las características propias de los personajes dinámicos, pero generalmente sí podrán modificarse y cambiar algunas características de los PJs o PNJs que interactúen con ellos.

El procedimiento para dibujar los objetos será el siguiente:

En primer lugar se obtendrá un elemento de la lista. Dicho elemento nos indicará el tipo y el índice del elemento a buscar. Una vez que se haya obtenido el objeto (en caso de que el elemento sea de tipo objeto), se comprobará el valor del atributo “visi”. Sólo si el objeto es visible (visi = 1) se dibujará. Seguidamente, se obtendrán las coordenadas (x,y), la longitud (en casillas) y el tipo al que pertenezca el objeto. En función de éste último, se obtendrá la textura o texturas asociadas (a partir de la lista de texturas) y se llamará al método GL_cubo o GL_poligonoT indicando los parámetros anteriores.

Para dibujar los objetos sin volumen de forma que no se note que son rectángulos con texturas, se utilizará una técnica denominada billboarding que dará como resultado el efecto de que el objeto siempre mirará de frente a la cámara [Foley]. Para implementar dicha técnica, bastará con que al

objeto no le afecte la perspectiva isométrica inicial. Si cada objeto se rota 45 grados en sentido inverso, quedará siempre de frente hacia nosotros. En realidad, lo que hacemos es deshacer la rotación de la cámara hecha al principio.

Fig. 16. Ejemplo de objetos

En este ejemplo podemos observar cómo se han dibujado objetos planos con transparencias (bibliotecaria, mesa y vela) y objetos con cubos (bibliotecas).

Además, podemos ver también que las bibliotecas tienen una textura para la cara frontal (madera con libros), y otro tipo de textura para las caras laterales (sólo madera), que en este caso podemos apreciar en la cara superior.

Tanto la bibliotecaria como la mesa y la vela ocupan 1 casilla en el eje x, a diferencia de las bibliotecas que ocupan cada una de ellas 2 casillas tanto en el eje x como en el z. Es por esto que la biblioteca aparece más ancha y más alta que los demás objetos.

Fig. 17. Ejemplo de objetos

En este caso podemos observar, al igual que antes, objetos planos con transparencias (posadera, mesa, vela y escoba) y objetos cúbicos (cama).

Si observamos, ahora el cubo de la cama tiene una textura en la cara superior (sábana, manta) y otro tipo de textura para las caras frontal y laterales (madera).

El efecto de ver al personaje "dentro" del cubo se consigue dibujando primero la cama y luego al personaje.

Fig. 18. Ejemplo de objetos

En este otro ejemplo, podemos ver que todos los objetos son rectángulos con texturas transparentes, aunque el efecto de textura plana no se nota por la técnica de billboarding ya comentada. Todos los objetos miran de frente a la cámara.

4.4 Visualización de los personajes

La visualización de los personajes será muy similar a la de los objetos. Aunque existen algunas diferencias:

- Todos los personajes tendrán la misma longitud en casillas (x,y) que será 1x1.
  Por esta razón, no existen atributos en los personajes que indiquen su tamaño.

- Todos los personajes se dibujarán como rectángulos y con texturas RGBA.

De esta forma, el proceso será el siguiente:

Se obtendrá un elemento de la lista de elementos. Si el tipo es personaje, se buscará en la lista de personajes y se obtendrán sus coordenadas y su especie. En función de ésta, se buscará la textura correspondiente, y finalmente, se dibujará el personaje utilizando el método GL_poligonoT.

La técnica de billboarding también será aplicada para los personajes, de modo que se rotarán 45 grados en el sentido opuesto al que se había rotado la cámara (perspectiva isométrica).

Para indicar al usuario PJ cuál es su personaje, se dibujará un puntero encima de dicho personaje.
Para ello utilizaremos el método GL_cursor que dibujará un triángulo en una posición superior a la que se encuentre el personaje activo. Dicho cursor también se utilizará para señalar a los objetos/personajes con los que se quiera interactuar. Una vez señalado el elemento en cuestión se abrirá el menú de acciones ya comentado.

El cursor tendrá asociado dos estados, que se diferenciarán visualmente por el color que adapte el puntero. Los valores de los estados serán 0 y 1.
El 0 hará referencia al estado en que el cursor sólo señale al personaje.
El 1 se corresponderá al estado cuando el personaje quiera actuar con algún elemento, y permitirá desplazar el cursor hacia dicho elemento.

Cuando el cursor esté a 0, se desplazará con cada movimiento del personaje, de forma que siempre quedará encima de él. Para ello, en cada visualización habrá que dibujar el cursor actualizando sus coordenadas con respecto al personaje.

Las coordenadas del cursor serán las que se enviarán al servidor para indicarle la posición del elemento con el cual se quiera actuar. Además, dichas coordenadas tendrán que volver a su posición original (con el personaje) cuando la acción se haya realizado (con éxito o no). Para ello, el servidor actualizará su valor y se lo enviará al cliente junto a las listas de datos del mundo. Los clientes PNJs también enviarán dichas coordenadas al servidor (para indicar el elemento de la interacción), pero obviarán las actualizaciones de las mismas, ya que no necesitarán dibujar nada.

Fig. 19. Ejemplo de cursor en estado 0 (señala al personaje)

Fig. 20. Ejemplo de cursor en estado 1 (señala a un elemento)

Como el cursor se dibujará después del mundo, y además será opaco, se desactivará el Z-buffer al dibujarlo.

4.5 Visualización de los edificios

Como ya sabemos, los edificios se dibujarán utilizando cubos.
Las caras de los cubos tendrán asociadas 3 texturas de la siguiente forma: la cara delantera tendrá asociada un tipo de textura, normalmente de fachada de edificio; por otra parte, las caras laterales (izquierda y derecha) se asociarán con texturas de tipo lateral de edificio, y finalmente, a la cara superior se le asignará una textura de tejado. Como vemos, no hemos mencionado la cara inferior, que al constituir la base del cubo no se hará visible en ningún momento, y por tanto no la dibujaremos.

De este modo, los cubos se dibujarán cara a cara asociando las texturas correspondientes.
El método GL_cubo será el encargado de implementar este proceso.

Cada cubo se posicionará en las coordenadas (x,y) asociadas que se corresponderán con el vértice inferior de la cara delantera.

Para que los cubos queden perpendiculares al terreno, será necesario rotarlos 90 grados en el eje x.

El proceso de dibujado será el siguiente:

Sólo si el personaje se encuentra en el mundo exterior se dibujarán los edificios de la lista recibida.
Al igual que en los casos anteriores, tendremos un bucle que iterará sobre los elementos de la lista. Obtendremos cada uno de los edificios, y por tanto, sus atributos, como la posición, longitud y tipo. Según éste último cargaremos un tipo de texturas u otro. Finalmente, llamaremos al método GL_cubo para que dibuje el edificio con los parámetros correspondientes. Éste método, básicamente, irá dibujando cada cara (indicando los cuatro vértices) de forma independiente.

Los edificios servirán como entrada hacia otros submundos. Cuando el personaje entre en ellos, desaparecerá el mundo exterior y se dibujará otro mundo más pequeño. Cuando el personaje salga de dicho submundo, se volverá a visualizar el mundo exterior.

Para que el personaje pueda entrar en los edificios, se tendrá que cumplir una de estas dos condiciones:

- El edificio tendrá el atributo de llav a 0, es decir, no será un edificio que requiera llave para entrar.

- El edificio tendrá el atributo de llav a 1 y el personaje tendrá un elemento llav en su equipo.

Se cumpla tanto una condición como otra, el personaje entrará finalmente en el edificio cuando sus coordenadas (x,y) coincidan con las coordenadas delanteras del interior del recinto que englobe el cubo del edificio. De este modo, el personaje no podrá entrar por los laterales o por la parte posterior. El servidor no aceptará ninguna acción que implique entrar a un edificio por una zona que no sea la parte frontal.

La longitud del edificio será el número de casillas (x,y) que ocupe. Normalmente, se tratará de tamaños 3x3 o 2x2. Las casillas accesibles serán las dos o tres casillas interiores delanteras.

Fig. 21. Ejemplo de edificios

Como podemos ver en el ejemplo, se muestra al personaje entre dos edificios. Claramente, podemos apreciar el cubo diseñado con los tres tipos de texturas ya comentados. En este caso los edificios mostrados son 2x2.

Si el personaje quisiese entrar en el edificio de la izquierda, tendría que avanzar hacia la fachada. Cuando su posición coincidiera con el interior del recinto el servidor modificaría el atributo “edif” del personaje con el identificador del edificio en cuestión. El método GL_display visualizaría el submundo asociado al edificio en vez del mundo exterior.

El cómo dibujar los submundos de los edificios lo veremos en la sección siguiente.

4.6 Visualización de los interiores

Los submundos de los edificios no serán en realidad "submundos", ya que no necesitarán información adicional asociada. Los elementos que se incluirán en cada submundo interior se encontrarán definidos o bien en la lista de personajes, o bien en el fichero de objetos.

En un submundo interior sólo se necesitará información referente a personajes y objetos, ya que para las casillas no hará falta, por tratarse de terrenos homogéneos, y tampoco será necesario para los edificios, porque sólo aparecerán en el exterior.

La diferencia entre un elemento (personaje u objeto) del exterior y un elemento del interior será el campo “edif”. Cuando dicho campo tenga un valor -1, el elemento en cuestión se dibujará en el exterior. Cuando por el contrario, presente un valor diferente, dicho valor hará referencia al edificio al que pertenezca, y por tanto, será dibujado cuando el personaje activo se encuentre dentro de dicho edificio.

El método GL_display, como ya sabemos, dibujará los elementos que se encuentren en el radio de percepción del personaje. Cuando dicho personaje entre en un edificio, el servidor modificará el campo “edif” y cargará en las listas del mundo todos los elementos cuyo atributo “edif” coincida con el del personaje. De este modo, al enviarse las listas modificadas al cliente gráfico se dibujarán los elementos asociados al interior del edificio.

Para las casillas, el mecanismo será muy simple. En función del tipo del edificio se utilizará una textura determinada para todas las casillas. También se tendrán unos límites determinados del submundo en función del tipo. Bastará con aplicar el mismo bucle que se mencionó en la sección de visualización de casillas. De este modo, obtendremos un terreno uniforme más pequeño que el del mundo exterior.

Para salir del interior, el personaje tendrá que colocarse en una casilla determinada que el servidor interpretará como salida y que, por consiguiente, dará como resultado el modificar el campo “edif” a -1 (exterior).

Otro aspecto importante son las coordenadas restauradas del personaje al salir. Si el personaje al entrar, está colocado en unas determinadas coordenadas (entra por un sitio), al salir debe coincidir con dichas coordenadas (debe salir por el mismo sitio). Para ello, cuando el servidor acepte una acción de entrar, almacenará las coordenadas para restaurarlas cuando dicho personaje salga. De este modo, se consigue un orden lógico de entrada/salida.

El cambio de visualización de un mundo a otro, será posible gracias al método GL_display, que como ya comentamos, redibujará todos los elementos del radio de percepción que se tengan en ese momento. Por tanto, al cambiar las listas del mundo, la visualización se modificará. Los elementos visibles en el exterior ya no lo serán porque no formarán parte del nuevo radio de percepción del personaje, que se habrá actualizado al interior del edificio.

Podemos resumir que el entrar o salir de un submundo sólo consistirá en un cambio en el radio de percepción.

Un último detalle es que los escenarios de los interiores requerirán dibujar muros o paredes limitando el recinto. Por ello, cuando el personaje activo se encuentre en el interior de un edificio habrá que dibujar rectángulos que rodeen el entorno. Dichos rectángulos se dibujarán en unas determinadas coordenadas en función del tamaño del submundo. Además, según el tipo de edificio, tendrán asociada una textura.

Los muros se dividirán en dos grupos: los muros que rodearán los laterales (dibujados a lo largo de nuestro eje y), y los muros que rodearán la parte delantera y trasera (dibujados a lo largo del eje x).

Habrá un muro en la parte delantera que no se dibujará. Será la posición asociada a la salida del recinto. Por ello, para que dicha salida sea visible al personaje, se dejará un hueco en la pared del submundo interior.

Los muros del eje y serán rotados 90 grados en el eje y (para que se vean perpendiculares de forma lateral).

Los muros del eje x serán rotados 90 grados en el eje x (para que se vean perpendiculares de frente).

El proceso de dibujado será el siguiente:

Se tendrán dos bucles anidados. Uno asociado al eje x, y el otro para el eje y. Los límites del bucle vendrán impuestos por las dimensiones que tenga asociado el submundo interior correspondiente.

El bucle x sólo tendrá en cuenta dos posiciones, la mínima y la máxima del recinto.
El bucle y iterará en todos los valores dentro de los límites.

En cada iteración se dibujará un muro, bien en la posición dada por los iteradores (x,y) si es un muro lateral, o bien en la posición inversa (y,x) si es un muro frontal.

Para dibujar los muros utilizaremos la función GL_poligono. Las texturas asociadas serán del tipo RGB.

Fig. 22. Ejemplo de interior

En este ejemplo, podemos observar el recinto de un edificio, que es más pequeño que el radio de percepción del personaje. Como vemos, se ha asignado un color negro al fondo, visible en las partes en las que no hay dibujados elementos del mundo.

Fig. 23. Ejemplo de interior

En este segundo ejemplo, podemos ver un recinto más grande, que ocupa más del radio de percepción del personaje.

En ambos ejemplos puede comprobarse el efecto de los muros que rodean el recinto.

4.7 Visualización de los menús

Por último, vamos a comentar la parte de los menús del juego.
Como ya sabemos, habrá una pantalla de inicio en la que el cliente podrá elegir al personaje con el que quiera jugar; una pantalla de fin, que indicará el final del juego para un determinado personaje; un cuadro de características, que imprimirá los atributos del personaje; y un menú de acciones que mostrará las acciones asociadas a un elemento para que el personaje elija.

Tanto la pantalla de inicio como la de fin serán texturas. Para visualizarlas, se definirá un rectángulo lo suficientemente grande como para que englobe la ventana de la aplicación. Dicho rectángulo se asociará con la textura.

La única diferencia entre la pantalla de inicio y la de fin, es que la primera presentará un icono que se podrá mover para elegir una opción. El icono será otro rectángulo asociado a una textura RGBA. El movimiento vendrá dado por el evento de pulsar una tecla. En función de ésta, el icono se trasladará a unas coordenadas determinadas de entre las pertenecientes al intervalo de opciones.

El cuadro de características, como ya se dijo, será un rectángulo asociado a una textura RGBA sobre el que se imprimirá texto indicando los atributos del personaje. Para ello, sólo habrá que convertir el valor de cada atributo en una cadena de caracteres para que sea imprimible por las funciones de la librería glf.

Dicho cuadro se visualizará o no, dependiendo de un evento asociado a una tecla determinada.

Finalmente, el menú de acciones, será también un rectángulo con textura RGBA y con cadenas de texto imprimibles que se asociarán a cada una de las acciones disponibles. Además, también se utilizará un icono para seleccionar la acción correspondiente de una forma parecida a la pantalla de inicio.

La acción solicitada, indicada por el icono, será la petición del cliente PJ al servidor. Dicha petición será aceptada (reflejándose sus consecuencias), o rechazada (no habrá ningún cambio).

5 Control Inteligente del Comportamiento

En esta sección vamos a explicar el comportamiento inteligente utilizado en el proyecto, describiendo las arquitecturas de agentes empleadas así como los algoritmos diseñados.

Principalmente, veremos dos arquitecturas de agentes en las secciones posteriores: arquitecturas reactivas y deliberativas.

En primer lugar vamos a aclarar conceptos importantes como qué es la inteligencia artificial, qué es un agente y los tipos de agentes que puede haber, para posteriormente aplicarlo a nuestro proyecto. Será un modo de justificar por qué es necesario utilizar "inteligencia" en el juego.

La inteligencia artificial parte de dos ideas: el principio de inteligencia y el principio de conocimiento.

El principio de inteligencia según Lenat-Feigembaum describe a la inteligencia como la capacidad para **buscar** una solución adecuada en un espacio de potenciales alternativas.

El principio de conocimiento se basa en que un sistema exhibe un comportamiento inteligente debido básicamente al **conocimiento** específico que puede manejar, y no a los algoritmos que incorpora para ello.

De este modo, se puede definir la inteligencia artificial como la suma de dos componentes: búsqueda y conocimiento.

La búsqueda será una forma de analizar las posibles alternativas para obtener una solución, y el conocimiento será la información necesaria que apoyará la decisión tomada.

Un agente es un ente colocado en un entorno que es capaz de percibirlo mediante sensores y actuar en él con actuadores. Más concretamente, un agente es aquel sistema que cumple unas determinadas propiedades:

- Reactivo (que sea capaz de detectar y actuar)
- Autónomo (que sea independiente de otros sistemas)
- Orientado por metas (que tenga un propósito definido)
- Temporalmente continuo (que exista durante un tiempo constante)

La inteligencia artificial es la que permite al agente tomar las decisiones en el entorno.

En nuestro caso, tenemos un entorno bien definido, y además, tenemos personajes independientes que son capaces de percibirlo, aunque de modo imperfecto, debido a que su radio de percepción es limitado. Dichos personajes, a través del conocimiento que adquieren del mundo, realizan una serie

de acciones guiadas por determinados objetivos. Resumiendo, los personajes del juego son propiamente agentes, ya que cumplen todas las propiedades anteriores.

Como ya sabemos, tendremos dos tipos de agentes: PJ y PNJ.
La toma de decisiones y objetivos de los agentes PJ vendrá definida por las decisiones que tome el usuario humano. Sin embargo, los agentes PNJ tendrán que simular dicho comportamiento al estar controlados por el computador. Es por ello, que necesitamos un comportamiento inteligente asociado a ellos.

Los agentes que utilizaremos en el proyecto serán agentes situados, ya que cumplirán las siguientes características:

- Están situados en un ambiente determinado (mundo)
- Su conducta está guiada por un conjunto de objetivos (los que determine el usuario, o los que determine el comportamiento inteligente de los agentes computacionales)
- Poseen recursos propios (características de los personajes)
- Son capaces de percibir su entorno de modo imperfecto (radio de percepción asociado a cada personaje)
- Tienen una representación interna de su ambiente (listas de datos del mundo)
- Poseen habilidades (acciones disponibles de los personajes)
- Su conducta tiende siempre a satisfacer sus objetivos, teniendo en cuenta los recursos, percepciones y habilidades disponibles en cada momento (toma de decisiones)

Ya que se describió en la sección anterior cómo percibían y actuaban los agentes PJ, veremos a lo largo de toda esta sección cómo funcionan los agentes PNJ.

La diferencia radical está en que los usuarios necesitan una interfaz gráfica para percibir y solicitar las acciones, y por el contrario, los PNJ necesitan realizar un análisis aplicado a las listas del mundo recibidas para poder solicitar las acciones.

Los agentes PNJ podrán tomar dos tipos de arquitecturas: reactivas o deliberativas.

Las arquitecturas reactivas se pueden ver como arquitecturas horizontales divididas en diferentes niveles de abstracción donde cada nivel puede percibir y actuar.

Gráficamente:

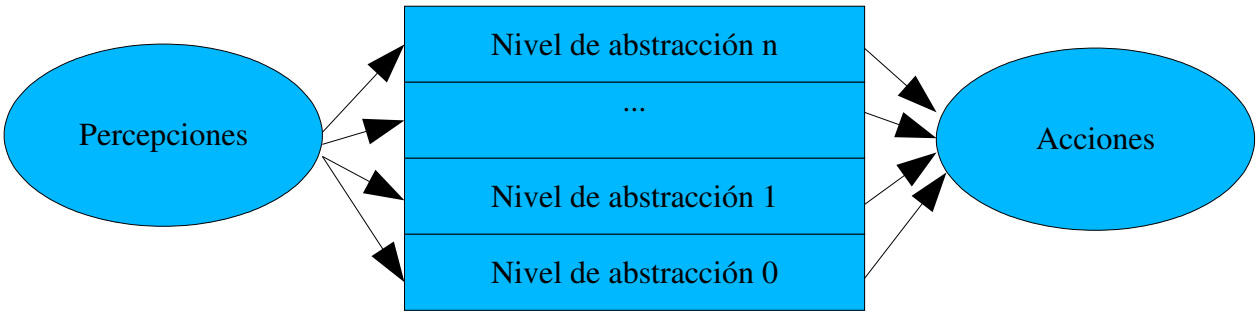


Fig. 24. Arquitectura horizontal

Las arquitecturas reactivas proponen una arquitectura que actúa según un modelo estímulo respuesta (conductista). No utilizan razonamiento simbólico complejo si no que mantienen un conjunto de patrones de comportamiento que se activan por la percepción y tienen efecto directo sobre las acciones. En realidad, su comportamiento es el de un agente reflejo que contiene reglas condición-acción.

Su funcionamiento podría ser el siguiente:

funcion tomaDecision (percepción):

estadoMundo ← interpreta (percepción)

regla ← acopla (estadoMundo, regla)

accion ← obtieneAccion (regla)

devuelve accion

Podría haber también agentes reflejo con memoria, que actualizarían el estado obtenido a partir de las percepciones con información adicional. Sería una forma de tener aún más conocimiento para elegir las acciones.

Ya que hemos visto las arquitecturas reactivas, pasemos a las deliberativas.

Las arquitecturas deliberativas, al contrario de las reactivas, se pueden ver como arquitecturas verticales, donde existe una jerarquía entre los niveles de abstracción. El nivel inferior es el que percibe, y el nivel superior es el que actúa, de modo que, para tomar una decisión a partir de las percepciones es necesario pasar por una serie de niveles intermedios.

Gráficamente:

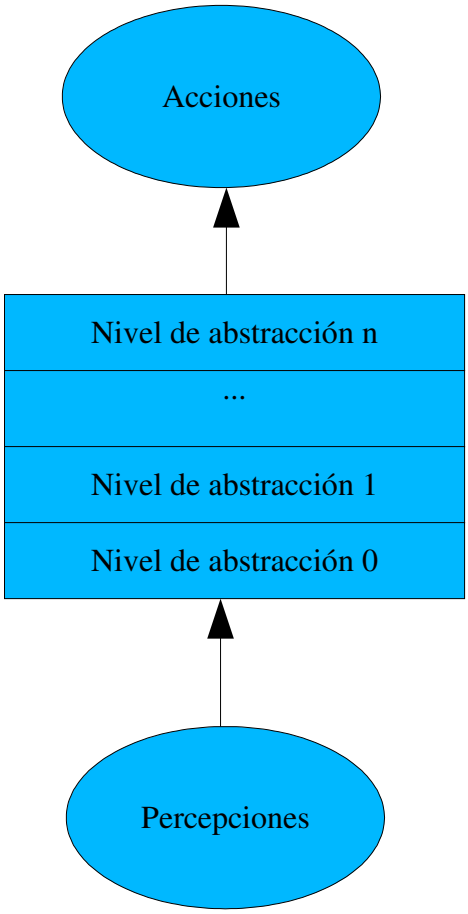


Fig. 25. Arquitectura vertical

Las arquitecturas deliberativas suelen basarse en la teoría clásica de planificación con los siguientes elementos:

- Un estado inicial (estado a analizar)
- Un conjunto de operadores/planes (acciones)
- Un estado final (el obtenido al aplicar una serie de acciones al inicial)

El agente delibera para determinar que acciones debe encadenar para lograr su objetivo siguiendo un enfoque top-down.

Las arquitecturas deliberativas siguen un funcionamiento de agentes basados en objetivos donde se busca un plan (sucesión de acciones) que alcancen un objetivo determinado.

Los objetivos en los agentes reactivos van implícitos en los patrones condición-acción.

El funcionamiento de un agente deliberativo puede verse en el siguiente esquema:

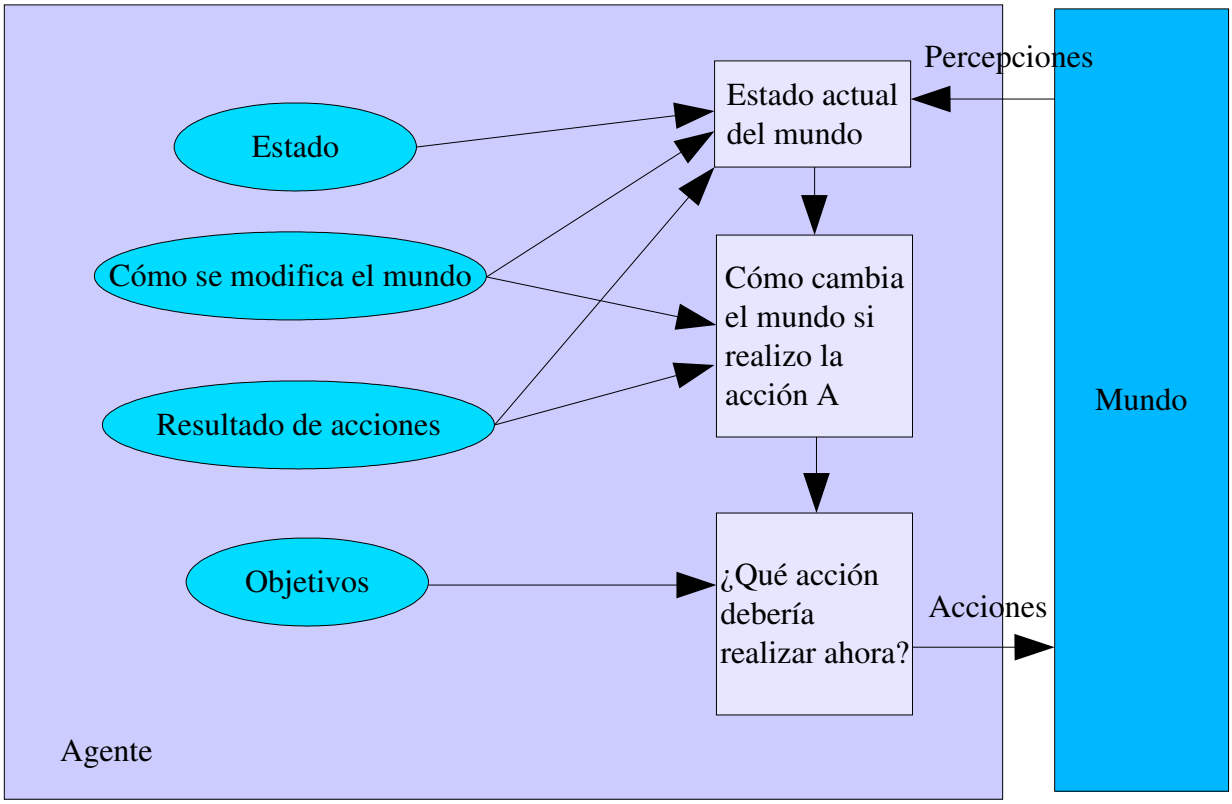


Fig. 26. Funcionamiento del agente deliberativo

Al igual que decíamos en los agentes reactivos, podría haber también agentes deliberativos con memoria que complementarían la información recibida de las percepciones para elaborar el plan de acciones.

En conclusión, en el proyecto utilizaremos cuatro tipos de arquitecturas de agentes para los PNJ:

- Agentes reactivos (basados en reglas condición-acción)
- Agentes reactivos con memoria (que incorporarán algún tipo de información adicional para solventar los problemas que surjan por la imperfección de las percepciones)
- Agentes deliberativos (que planifican para obtener un determinado objetivo)
- Agentes deliberativos con memoria (que al igual que antes, incorporarán información complementaria a las percepciones)

Los comportamientos de estos tipos de agentes nos servirán como comparativas de actuación frente a un determinado estado del mundo.

Aquellos agentes que realicen las acciones más correctas para cumplir sus propósitos serán agentes racionales. Además, si se comportan de forma similar a como lo haría un PJ, serán también agentes "inteligentes" en cierto modo.

Con estas condiciones se habrían alcanzado los tres objetivos del proyecto dando lugar a una verdadera simulación.

Como ya dijimos, los clientes PNJ se implementarán en lenguaje Python. La interfaz de conexión serán los sockets, y el protocolo y mecanismo de intercambio de información será el comentado en secciones previas.

Las listas del mundo, al enviarse elemento a elemento, tendrán que reconstruirse también en el cliente PNJ. Al estar éste implementado en otro lenguaje, se crearán las mismas clases que para el cliente PJ, pero en este caso en Python.
Tendremos, por tanto, una clase para los personajes, para los objetos, para los edificios y para las casillas. La clase elemento sólo se utilizará para almacenar en algún sitio los datos enviados por el servidor asociados a la ordenación en profundidad.

Al no tener menú de inicio para seleccionar la especie del personaje, el PNJ utilizará la línea de comandos. En caso de no especificar dicha especie, se tendría una por defecto.
El tipo de inteligencia (reactiva o deliberativa) también se obtendrá a partir de la línea de comandos.

El algoritmo del cliente PNJ se presenta en el siguiente esquema, que nuevamente es una especificación del que se incluyó en la sección del cliente:

```
sockCliente = socket ()
conectaAServidor (sockCliente)

Si (argc >= 2):
        especie = argv[1]
        inteligencia = argv[2]

Si no:
        especie = valorPorDefecto
        inteligencia = valorPorDefecto

envia (especie, sockCliente)
recibe (tamanoMundo, sock)

fin = 0
Mientras (fin == 0) Repetir:

        recibe (flagEnvio, sockCliente)
        recibe (turno, sockCliente)
        recibe (sockCliente, sockCliente)

        Si (flagEnvio > 0):

                Si (sockCliente == -1):
                        envia (desconexion, sockCliente)
                        fin = 1
                Si no:
                        según (flagEnvio):
                                recibe (mundo, sockCliente)

        Si (fin == 0):

                Si (inteligencia == 0):


                        accion = tomaDecisionReactiva (tamanoMundo, mundo, turno,
                                                       sockCliente)
```

Si no:

accion = tomaDecisionDeliberativa (tamanoMundo, mundo, turno, sockCliente)

envia (accion, sockCliente)

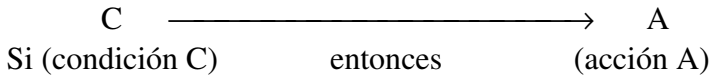
cierra (sockCliente)

En las subsecciones siguientes vamos a ver detenidamente el funcionamiento de la función tomaDecisionReactiva y tomaDecisionDeliberativa asociadas a los dos tipos de arquitecturas.

5.1 Comportamiento Reactivo

Vamos ahora a concentrarnos en la arquitectura reactiva.
Ya hemos dicho que nuestros agentes reactivos serán agentes reflejo basados en reglas condición-acción. Nos falta por saber cómo representar dicho funcionamiento. Una representación adecuada para los sistemas reactivos basados en reglas son los sistemas de producción [Nilsson].

Un sistema de producción está formado por un conjunto no vacío de reglas de producción o simplemente producciones, donde cada una de ellas es de la forma:



Normalmente, a la condición de la regla se le denomina precondición, y a la acción se le denomina postcondición.

Los sistemas de producción se caracterizan por:

- La condición se formula habitualmente como una fórmula lógica (expresión) cuya verdad dispara la acción.
- Dada una percepción se busca la regla que "acople" con ella y se dispara.
- La acción puede ser una acción primitiva, un conjunto de acciones que deben ejecutarse simultáneamente, o la llamada a otro sistema de producción.
- Entre las reglas debe haber alguna(s) que codifiquen el criterio(s) de parada, así como la acción a realizar cuando no se consigue el acoplamiento de la percepción con alguna regla.

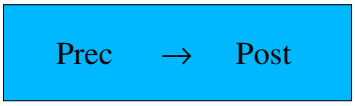
En nuestro caso, no habrá criterio de parada, ya que continuamente se requerirá una acción, a no ser que el personaje no exista ya en el juego. Lo que sí habrá será una regla de acción por defecto, que se lanzará cuando las otras reglas no se cumplan.

Como condición podremos tener una o varias precondiciones, ya que habrá reglas que necesiten que se cumplan varios requisitos a la vez (enlace Y). En caso de que las condiciones sean disyuntivas se dividirá la regla en varias reglas.

Las reglas tendrán una única postcondición que será la acción que solicite el personaje PNJ al servidor.

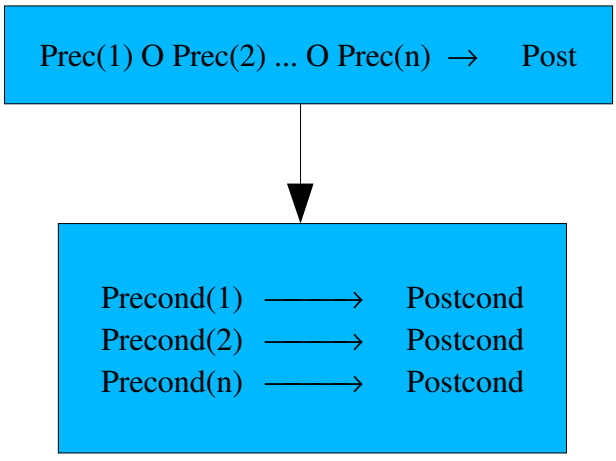
Gráficamente:

Posibilidad 1: Regla sencilla



Posibilidad 2: Regla compuesta

Posibilidad 3: Regla compuesta por disyunciones -> Varias reglas sencillas



El orden de los objetivos será el orden que sigan las reglas. El análisis se basará en un recorrido por todas las reglas, comprobando si se cumplen las precondiciones, en caso de que haya alguna que se verifique, se convertirá en la regla objetivo y su postcondición será la acción a realizar. Cuando no se cumpla ninguna regla, se llegará a una por defecto que tendrá por precondición una condición siempre verdadera, y por tanto, dará como postcondición una acción por defecto a realizar.

Las reglas tendrán que almacenarse en algún sitio, de modo que puedan leerse cómodamente.
Una forma de solucionarlo es tenerlas en un fichero de texto, donde cada línea de caracteres se corresponda con una regla. Cuando el cliente PNJ reactivo necesite tomar una decisión tendrá que abrir el fichero y buscar una regla verdadera. La primera que se cumpla será la elegida, de este modo, las reglas más preferentes serán las que ocupen las primeras líneas.

Para verificar las precondiciones, habrá que comprobar si se corresponden con el estado actual del mundo. El estado actual será el estado que tenga el mundo en el momento de tomar una decisión.

A continuación, podemos ver el contenido de los ficheros de reglas utilizados en el proyecto. Hemos incluido dos modelos para tener dos comportamientos diferentes, uno simple, basado sólo en 10 reglas, y otro más complejo basado en 55 reglas. El primer caso servirá de ejemplo y se asociará a un tipo de personaje bastante "primitivo", el segundo caso será el que defina el comportamiento del agente reactivo que estamos analizando.

Fichero: reglas1.txt

```
tieneHambre tieneComidaCerca Comer
tieneHambre tieneComidaLejos MoverHaciaComida
tieneHambre MoverAleatorio
tieneSed tieneAguaCerca Beber
tieneSed tieneAguaLejos MoverHaciaAgua
tieneSed MoverAleatorio
tieneCansancio Descansar
tienePersonajeCerca Robar
tienePersonajeLejos MoverHaciaPersonaje
porDefecto MoverAleatorio
```

Fichero reglas2.txt

```
tieneHambre tieneComidaCerca Comer
tieneHambre tieneComidaLejos MoverHaciaComida
tieneHambre estaExterior MoverAleatorio
tieneHambre estaSalida Salir
tieneHambre salidaLibre MoverHaciaSalida
tieneSed estaExterior tieneAguaCerca Beber
tieneSed estaExterior tieneAguaLejos MoverHaciaAgua
tieneSed estaExterior MoverAleatorio
tieneSed estaSalida Salir
tieneSed salidaLibre MoverHaciaSalida
tieneCansancio Descansar
tieneDinero estaHerido estaExterior tienePosadaCerca EntrarPosada
tieneDinero estaHerido estaExterior tienePosadaLejos MoverHaciaPosada
tieneDinero estaHerido tienePosaderaCerca Curarse
tieneDinero estaHerido tienePosaderaLejos MoverHaciaPosadera
estaPosada estaSalida Salir
estaPosada salidaLibre MoverHaciaSalida
```

```
estaHerido tienePersonajeCerca Defenderse
!tieneObjetivo tieneDineroBruja tieneBrujaCerca AceptarMision
!tieneObjetivo tieneDineroBruja tieneBrujaLejos MoverHaciaBruja
tieneLlave estaExterior tieneCastilloCerca EntrarCastillo
tieneLlave estaExterior tieneCastilloLejos MoverHaciaCastillo
tienePocionObjetivo estaPortal FinalizarMision
tienePocionObjetivo tienePortalLejos MoverHaciaPortal
tienePrisioneroCerca LiberarPrisionero
tienePrisioneroLejos MoverHaciaPrisionero
tieneDinero estaExterior tieneBibliotecaCerca EntrarBiblioteca
tieneDinero estaExterior tieneBibliotecaLejos MoverHaciaBiblioteca
tieneDinero tieneBibliotecariaCerca Estudiar
tieneDinero tieneBibliotecariaLejos MoverHaciaBibliotecaria
estaBiblioteca estaSalida Salir
estaBiblioteca salidaLibre MoverHaciaSalida
tieneDineroArmas estaExterior tieneArmasCerca EntrarArmas
tieneDineroArmas estaExterior tieneArmasLejos MoverHaciaArmas
estaArmas tieneDineroArmas tieneVendedorArmasCerca ComprarArma
estaArmastieneDineroArmas tieneVendedorArmasLejos MoverHaciaVendedorArmas
estaArmas estaSalida Salir
estaArmas salidaLibre MoverHaciaSalida
tieneDineroEscudos estaExterior tieneEscudosCerca EntrarEscudos
tieneDineroEscudos estaExterior tieneEscudosLejos MoverHaciaEscudos
estaEscudos tieneDineroEscudos tieneVendedorEscudosCerca ComprarEscudo
estaEscudos tieneDineroEscudos tieneVendedorEscudosLejos
MoverHaciaVendedorEscudos
estaEscudos estaSalida Salir
estaEscudos salidaLibre MoverHaciaSalida
tieneDineroArmaduras estaExterior tieneArmadurasCerca EntrarArmaduras
tieneDineroArmaduras estaExterior tieneArmadurasLejos MoverHaciaArmaduras
estaArmaduras tieneDineroArmaduras tieneVendedorArmadurasCerca
ComprarArmadura
estaArmaduras tieneDineroArmaduras tieneVendedorArmadurasLejos
MoverHaciaVendedorArmaduras
estaArmaduras estaSalida Salir
estaArmaduras salidaLibre MoverHaciaSalida
tieneCofreCerca AbrirCofre
tieneCofreLejos MoverHaciaCofre
tienePersonajeCerca !tienePocionObjetivo Robar
tienePersonajeLejos MoverHaciaPersonaje
porDefecto MoverAleatorio
```

Como vemos, los patrones (precondiciones o postcondición) van separados por espacios en blanco.

La postcondición será siempre el último patrón, y todo lo anterior se interpretará como precondiciones.

Las precondiciones precedidas por un símbolo "!", serán precondiciones negativas, de la forma Si NOT A entonces B.

La precondición "porDefecto" será una condición cuyo valor será siempre verdadero.

El cliente PNJ tendrá que realizar un "parser" de la información contenida en el fichero de texto para interpretar las reglas. Para ello, tendremos una función que leerá cada línea del fichero (cada regla) y la dividirá en dos partes, una lista de precondiciones y una postcondición. La lista de precondiciones será analizada elemento a elemento. Es decir, se comprobará si cada componente de la lista se cumple en el estado del mundo. Cada una de las precondiciones hará referencia a un atributo del personaje o a la distancia entre el personaje y algún elemento. Por ejemplo, se considerará si el personaje tiene hambre o está en algún edificio, o bien si se encuentra cerca o lejos de algún otro personaje, objeto o edificio. De este modo, se obtendrá el valor del atributo en cuestión o se medirá la distancia, para al final obtener un valor verdadero o falso.

Veamos cómo se interpretan cada una de las precondiciones:

tieneHambre:
Se comprueba si el atributo "ham" del personaje tiene un valor menor que la mitad del atributo "maxham". En ese caso sería verdadero.

tieneSed:
Se comprueba si el atributo "sed" del personaje tiene un valor menor que una tercera parte del atributo "maxsed". En ese caso sería verdadero.

tieneCansancio:
Se comprueba si el atributo "ene" del personaje tiene un valor menor que la mitad del atributo "maxene". En ese caso sería verdadero.

tieneDinero:
tieneDineroBruja:
tieneDineroArmas:
tieneDineroEscudos:
tieneDineroArmaduras:
Se comprueba si en el vector asociado al equipo del personaje, el elemento asignado al dinero tiene un valor suficiente para la condición indicada. En ese caso sería verdadero.

tieneObjetivo:
Se comprueba si el atributo "obj" del personaje tiene un valor distinto de 0. En ese caso sería verdadero.

!tieneObjetivo:
Se comprueba el caso opuesto al anterior.

tienePocionObjetivo:
Se comprueba si el atributo "obj" del personaje coincide con el valor del elemento del vector equipo asociado a la poción que tenga el personaje. En ese caso sería verdadero.

!tienePocionObjetivo:
Se comprueba el caso opuesto al anterior.

tieneLlave:
Se comprueba si el valor del elemento del vector equipo asociado a la llave del personaje tiene un valor distinto de -1. En ese caso sería verdadero.

estaHerido:
Se comprueba si el valor del atributo "pg" del personaje es menor al atributo "maxpg". En ese caso sería verdadero.

estaExterior:
Se comprueba si el valor del atributo "edif" del personaje es igual a -1. En ese caso sería verdadero.

salidaLibre:
Se comprueba si las coordenadas asociadas a la salida de un edificio están ocupadas por algún personaje. En ese caso sería verdadero.

estaSalida:
Se comprueba si las coordenadas del personaje coinciden con las de la salida de un edificio. En ese caso sería verdadero.

estaPosada:
estaBiblioteca:
estaArmas:
estaEscudos:
estaArmaduras:
Se comprueba si el atributo "edif" del personaje coincide con alguno de los edificios indicados. En ese caso sería verdadero.

estaPortal:
Se comprueba si las coordenadas del personaje coinciden con las del objeto indicado. En ese caso sería verdadero.

tiene"Algo"Cerca:
Se comprueba si las coordenadas del edificio, objeto, personaje o casilla indicados son adyacentes a las del personaje. En ese caso sería verdadero.

tiene"Algo"Lejos:
Se comprueba si las coordenadas del edificio, objeto, personaje o casilla indicados se encuentran dentro del radio de percepción del personaje. En ese caso sería verdadero.

El valor verdadero será un 1 lógico. El valor falso será un 0 lógico.

Cuando todas las precondiciones de la regla se cumplan, se devolverá la postcondición asociada. En otro caso, se pasaría a la línea siguiente (regla siguiente) para volver a hacer el proceso de verificación.

Una vez que se tenga la postcondición, tendrá también que traducirse ésta a un valor entero que se enviará al servidor. Para ello se utilizará otra función que transforme las postcondiciones en valores enteros.

Según el tipo de postcondición, se enviarán, además, las coordenadas del elemento con el que se interactúe. Esto se hará siempre que dicho elemento tenga que modificarse como resultado de la acción del personaje.

Para las acciones de movimiento del PNJ también se enviarán las coordenadas resultantes.
En caso de que el movimiento no pueda realizarse, o no le toque el turno al cliente, se enviará al servidor una acción por defecto que no supondrá ningún cambio. Al igual que decíamos con el clientePJ, esto hará que haya una interacción continua con el servidor.

A continuación, como resumen, podemos ver un esquema del funcionamiento del agente reactivo:

```
funcion tomaDecisionReactiva (tamanoMundo, mundo, turno, sockCliente):

    turnoCliente = obtieneTurnoCliente ()

    Si (turno == turnoCliente):

        especie = obtieneEspecie ()

        según (especie):

            ficheroReglas = abreFicheroReglas ()

        regla = leeLinea (ficheroReglas)
```

```
        Mientras (regla != ‘‘) repetir:
            postcond = compruebaRegla (regla, tamanoMundo, mundo)
            Si (postcond != ‘‘) :
                regla = ‘‘
            Si no:
                regla = LeeLinea (ficheroReglas)
        enviaAccion (postcond, tamanoMundo, mundo, sockCliente)
    Si no:
        envia (0, sockCliente)
```

Un aspecto muy importante que no hemos comentado es el relacionado a los movimientos del personaje.

Cuando un personaje quiera acercarse a un elemento determinado, tendrá que tomar el camino más corto. Además, en caso de que hubiera obstáculos entorpeciendo la trayectoria hacia dicho elemento, el personaje tendría que ser capaz de rodearlos o evitarlos para finalmente llegar a su objetivo.

De este modo, siempre que la postcondición implique una acción de movimiento dirigido, tendrán que obtenerse unas coordenadas de movimiento dentro del camino óptimo al destino.

Para obtener los caminos óptimos a los destinos utilizaremos el algoritmo A* [Nilsson].

Dicho algoritmo está basado en una búsqueda heurística que va generando un grafo a partir de un nodo inicial para llegar a un nodo solución. Para ello, se utilizan tres componentes; una asociada al costo de ir desde el nodo inicial al nodo a valorar (función g(n)), otra asociada a la estimación del costo desde el nodo a valorar hasta el nodo solución (función h(n)), y finalmente, la componente f(n) que es la suma de las anteriores.

La componente h(n) será una heurística para saber cómo de bueno es el nodo para alcanzar la solución.

En nuestro caso, cada nodo será un estado definido con un par de coordenadas (x,y) del mundo, asociadas a la posición inicial del personaje. La heurística será la distancia en valor absoluto hacia

el destino, es decir, la suma en x y en y del número de casillas que haya entre la posición inicial y la posición objetivo. De este modo, cada nodo será en realidad una casilla del mundo.

El algoritmo utiliza dos listas de nodos. Una lista, denominada ABIERTOS, que incluye aquellos nodos generados no expandidos; y otra lista, denominada CERRADOS, que incluye aquellos nodos generados y expandidos.

La forma de generar nuevos nodos, o mejor dicho, nuevas casillas, consistirá en aplicar un movimiento al nodo a expandir. Es decir, aplicaremos a la casilla a evaluar desplazamientos hacia arriba, abajo, izquierda y derecha, obteniendo, de esta forma, cuatro casillas hijas.

Cada nodo tendrá un enlace a su nodo padre, que nos servirá para obtener el camino solución en el grafo generado. Partiendo del nodo solución, sólo habrá que ir subiendo por el grafo (mediante los enlaces paternos) para obtener el nodo que produjo dicha solución. Lo que nos interesará saber es qué movimiento (arriba, abajo, izquierda, derecha) es el que produjo el camino óptimo en el grafo.

Los nodos que utilizaremos serán una estructura compuesta por los siguientes elementos:

- Un identificador del nodo (que será un número entero)
- El identificador del nodo padre
- La posición en x de la casilla
- La posición en y de la casilla
- La operación que se ha realizado para obtener el nodo (1 arrb, 2 abjo, 3 izda, 4 dcha)
- La componente g
- La componente h, que será el número de casillas que diste la casilla con la casilla destino
- La componente f, que será la suma de h y g

Para implementar dicha estructura utilizaremos una clase en Python denominada NodoA.

Para implementar el algoritmo A* utilizaremos una función que seguirá los siguientes pasos:
(Nota: Un nodo se considerará igual a otro si sus coordenadas coinciden)

- Se calculará el nodo inicial compuesto por las coordenadas de la casilla del personaje, con valor de g a 0, valor de h al que corresponda, valor de f a g + h, identificador a 0 (que se irá incrementando por cada nodo generado del grafo), y con identificador padre y operación a nulo.

- Se crearán como vacías las listas ABIERTOS y CERRADOS

- Se insertará el nodo inicial en ABIERTOS

- Se entrará en un bucle que se repetirá mientras ABIERTOS tenga nodos.

- En cada iteración del bucle, se eliminará aquel nodo de ABIERTOS con menor f y se insertará en CERRADOS convirtiéndose en el nodo a analizar (nodo actual).

- Se comprobará si el nodo actual es solución (su distancia a la casilla destino será mínima) y en tal caso, se ascenderá por el grafo hasta obtener el nodo hijo del inicial cuyo movimiento asociado ha obtenido la rama solución. Dicho movimiento será el valor que devolverá la función.

- En caso de que no sea solución, se calcularán los hijos del nodo actual aplicando los cuatro movimientos.

- Para cada hijo generado, se actualizarán sus valores, de modo que se incrementará el identificador, se asignará como identificador padre el identificador del nodo actual, se asignarán las nuevas coordenadas así como el movimiento generado, se incrementará g, y se calculará h y f.

  También se comprobará si es un nodo aceptable, es decir, se verá que no forme ciclos (que no aparezca en sus antecesores), y que no se corresponda a casillas inaccesibles, como la ocupada por personajes, ciertos objetos bloqueantes, paredes de edificios, y aquellas casillas que se salgan de los límites.

  Seguidamente, si el nodo es aceptable, se comprobará si aparece en la lista de ABIERTOS o CERRADOS.

  En caso de que esté en ABIERTOS, se comprobarán los valores g de ambos nodos coincidentes. Si el valor de g obtenido es menor, se actualizará el nodo de ABIERTOS con el nuevo valor de g, de f, de identificador de nodo padre y de operación.

- En caso de que esté en CERRADOS, se actuará de forma similar.

- Si el nodo hijo no se encontrara ni en ABIERTOS ni en CERRADOS se insertaría en ABIERTOS.

De esta manera, obtendremos las coordenadas óptimas en el siguiente paso que debe dar un personaje en su trayectoria hacia un destino próximo. Normalmente, en un algoritmo A* se requiere devolver toda la solución, es decir, toda la rama del grafo que va desde el nodo inicial hasta el objetivo. En nuestro caso no será así, ya que este tipo de soluciones se corresponden a problemas no interactivos como el 8-puzzle que devuelven una sucesión de estados para alcanzar la solución. En nuestro proyecto, en que los jugadores actúan por turnos, no se puede devolver todo el camino, ya que sólo se requiere una acción cada vez. No importa que el personaje tenga varias acciones por turno, ya que para cada acción se realizará siempre un análisis y, por tanto, se volverá a calcular este algoritmo. Además, otra diferencia entre nuestro caso y problemas como el 8-puzzle, es que aparte de los turnos, el estado del mundo siempre se puede estar modificando por las acciones asociadas a

otros personajes. No hay un estado concreto.

Formalmente, en el algoritmo A* cuando se actualiza un nodo ya generado en CERRADOS, se propaga dicha modificación hacia abajo recursivamente hasta encontrar un nodo sin sucesores o un nodo para el que se haya encontrado un camino igual o mejor (menor valor de g). Esta parte, normalmente, no se implementa porque puede generar una eficiencia de orden exponencial. Por ello, y más en nuestro caso, que se requieren tomas de decisiones en tiempo "interactivo", es necesario optimizar el tiempo de procesamiento, y por ello, se sustituye este procedimiento por el mecanismo cuando el nodo aparece en ABIERTOS.

Otra diferencia con el algoritmo formal, es que no se utiliza una lista de nodos hijos para cada nodo, ya que en nuestro caso cada nodo queda totalmente localizado por su padre y su identificador.

Con esto, ya hemos analizado el funcionamiento de un agente reactivo. Nos queda saber cómo implementar el agente reactivo con memoria.

Como ya dijimos, un agente utiliza la memoria para complementar las imperfecciones de las percepciones. En nuestro caso, las percepciones no presentan ningún problema, ya que dentro del radio de percepción, cada personaje es capaz de detectar si hay comida, agua, edificios, personajes, etc... Además, según lo visto anteriormente será capaz de acercarse de forma óptima a dichos elementos.

El problema puede aparecer cuando no se pueden utilizar las percepciones. Un ejemplo sería el siguiente:

Un personaje detecta un cofre en su radio de percepción. Dicho personaje decide acercarse al cofre y abrirlo. No es posible, a priori, saber qué hay dentro del cofre. El personaje no puede utilizar sus percepciones para saber cuál es el contenido, ya que no es visible. Esto puede provocar que el personaje abra cofres con elementos que ya tiene en su inventario y no se pueden incrementar, tales como una llave o una determinada poción. El personaje podría quedarse en un ciclo abriendo y rechazando el contenido del cofre, ya que la regla (si tieneCofreCerca entonces abreCofre) se cumpliría.
La única forma de completar esta regla es utilizando una memoria, de modo que el personaje "recuerde" el contenido de los últimos cofres que haya abierto, no dirigiéndose a aquellos que no sean útiles.

Para ello, se tendrá una lista de 5 items máximo, donde se almacenará los 5 últimos cofres abiertos no útiles. Cada vez que el personaje detecte un cofre y se dispare la regla asociada, comprobará si dicho cofre está en la lista de items. En tal caso, comprobará nuevamente si dicho cofre sigue siendo inútil (ya que puede haber cambiado el inventario del personaje) y en caso de que lo sea se ignorará dicho objeto. Si el cofre fuese inútil y no estuviera en la lista, se insertaría al final de dicha lista, eliminando el elemento más antiguo en caso de que ya estuviese completa.

Para implementar la lista de items utilizaremos una clase en Python denominada Item, compuesta por dos atributos: id (que indicará el identificador del objeto), y elem (que indicará el tipo de elemento que incluirá el objeto).

5.2 Comportamiento Deliberativo

En la sección anterior hemos visto una arquitectura de agentes que para tomar una decisión se basan en un sistema de reglas. Ahora vamos a ver otro tipo de arquitectura donde los agentes no tendrán una estructura tan rígida, si no que serán capaces de analizar la situación para llegar a una conclusión.

Los agentes reactivos siguen un proceso de percibir y actuar. Sin embargo, los agentes deliberativos realizan una planificación para resolver el problema, es decir, perciben, planifican, y actúan.

La planificación es un proceso que busca una secuencia de operaciones (plan) que resuelva el problema. En realidad, consiste en razonar o deliberar en base a un conjunto de acciones para conseguir un determinado fin.

En un sistema de planificación aparecen ciertas características que no se presentan en otros sistemas, por ejemplo, los basados en reglas o en búsquedas. Las características se dividen en tres tipos de problemas:

- **(1) Problema del marco**
  Hace referencia a cómo representar de forma eficiente los estados y las acciones para poder razonar sobre cómo era el mundo antes y después de realizar una acción.

- **(2) Efectos dependientes del contexto**
  Hace referencia a cómo representar los efectos de una acción.

- **(3) Problema de la cualificación**
  Se basa en determinar qué precondiciones son necesarias indicar para que la acción se cumpla.

- **(4) Problema de la ramificación**
  Se basa en mostrar una ramificación de las posibles acciones.

Como algoritmo de planificación utilizaremos el método STRIPS, que soluciona los problemas anteriores [Nilsson].

Para representar los estados del mundo se utilizarán las listas de datos del mundo, que describirán la situación inicial.

Las acciones se representarán mediante un conjunto de operadores que tendrán una serie de precondiciones necesarias para ser aplicados, y una serie de postcondiciones, que serán los efectos que se provocarán en el estado inicial al aplicarse. (Con esto se soluciona el problema 1)

Para ver si un operador se cumple, habrá que comprobar si sus precondiciones coinciden con el estado actual. De modo, se tendrá que comprobar dichas precondiciones con las listas del mundo.

Los operadores se irán encadenando de forma que cuando uno de ellos necesite una precondición que no aparezca en el estado actual, se tendrá que buscar otro que al aplicarse produzca un efecto que coincida con dicha precondición.

Gráficamente, el encadenamiento sigue el siguiente esquema:

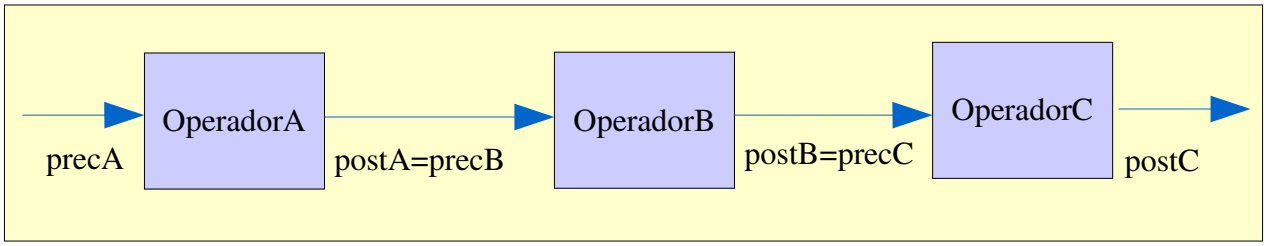


Fig. 27. Esquema del encadenamiento de operadores

Como las postcondiciones que surjan, al ir aplicando operadores, no aparecerán en el mundo, por tratarse de modificaciones hipotéticas, tendremos que completar el estado actual con dichos efectos sin modificar el mundo. Para ello, utilizaremos una lista donde se almacenarán las postcondiciones de los operadores. (Solución al problema 2)

De esta manera, un estado vendrá representado por las listas del mundo y la lista de postcondiciones.

Formalmente, en STRIPS, se va modificando el estado actual con los efectos de los operadores. En nuestro caso, no se podrá hacer de este modo porque el mundo sólo se modificará mediante el servidor, y además sólo cuando éste acepte la acción final.

En STRIPS, cada operador tiene asociada una lista de supresión (condiciones que dejan de ser ciertas), y una lista de adición (condiciones que pasan a ser ciertas). En nuestro caso, al no poderse modificar el mundo, los cambios se incluirán en la lista de postcondiciones ya comentada, que actuará como lista de adición. No utilizaremos ninguna lista de supresión, ya que las condiciones que dejen de ser ciertas se verán anuladas por las modificaciones.

Los problemas 3 y 4 se solucionan con el diseño de los operadores que mostramos a continuación:

Fichero operadores.txt

```
Comer
tieneCercaComida estaExterior
comido

Beber
tieneCercaAgua estaExterior
bebido

Descansar
vacio
descansado

Curarse
tieneCercaPosadera estaPosada tieneDinero
curado

Defenderse
tieneCercaPersonaje
defendido

FinalizarMision
estaPortal estaCastillo tienePocionObj tieneObjetivo
incrementadasCarac

AceptarMision
tieneDineroBruja tieneCercaBruja estaExterior
tieneObjetivo

Estudiar
tieneCercaBibliotecaria estaBiblioteca tieneDinero
incrementadosMH

ComprarArma
tieneCercaVendedorArmas estaArmas tieneDineroArmas
tieneArma

ComprarEscudo
tieneCercaVendedorEscudos estaEscudos tieneDineroEscudos
tieneEscudo

ComprarArmadura
tieneCercaVendedorArmaduras estaArmaduras tieneDineroArmaduras
tieneArmadura
```

```
EntrarArmas
estaExterior tieneDineroArmas tieneCercaArmas
estaArmas tieneLejosVendedorArmas

EntrarEscudos
estaExterior tieneDineroEscudos tieneCercaEscudos
estaEscudos tieneLejosVendedorEscudos

EntrarArmaduras
estaExterior tieneDineroArmaduras tieneCercaArmaduras
estaArmaduras tieneLejosVendedorArmaduras

EntrarCastillo
estaExterior tieneLlave tieneCercaCastillo
estaCastillo tieneLejosPortal tieneLejosPrisionero

EntrarPosada
estaExterior tieneDinero tieneCercaPosada
estaPosada tieneLejosPosadera

EntrarBiblioteca
estaExterior tieneDinero tieneCercaBiblioteca
estaBiblioteca tieneLejosBibliotecaria

Salir
estaInterior estaSalida
estaExterior tieneLejosComida tieneLejosAgua tieneCercaBruja

MoverHaciaComida
tieneLejosComida
tieneCercaComida

MoverHaciaAgua
tieneLejosAgua
tieneCercaAgua

MoverHaciaSalida
salidaLibre
estaSalida

MoverHaciaPosada
tieneLejosPosada
tieneCercaPosada

MoverHaciaPosadera
tieneLejosPosadera
tieneCercaPosadera

MoverHaciaBibliotecaria
tieneLejosBibliotecaria
tieneCercaBibliotecaria
```

```
MoverHaciaBruja
tieneLejosBruja
tieneCercaBruja

MoverHaciaCastillo
tieneLejosCastillo
tieneCercaCastillo

MoverHaciaPortal
tieneLejosPortal
estaPortal

MoverHaciaPrisionero
tieneLejosPrisionero
tieneCercaPrisionero

MoverHaciaBiblioteca
tieneLejosBiblioteca
tieneCercaBiblioteca

MoverHaciaArmas
tieneLejosArmas
tieneCercaArmas

MoverHaciaEscudos
tieneLejosEscudos
tieneCercaEscudos

MoverHaciaArmaduras
tieneLejosArmaduras
tieneCercaArmaduras

MoverHaciaVendedorArmas
tieneLejosVendedorArmas
tieneCercaVendedorArmas

MoverHaciaVendedorEscudos
tieneLejosVendedorEscudos
tieneCercaVendedorEscudos

MoverHaciaVendedorArmaduras
tieneLejosVendedorArmaduras
tieneCercaVendedorArmaduras

MoverHaciaCofre
tieneLejosCofre
tieneCercaCofre

MoverHaciaPersonaje
tieneLejosPersonaje
tieneCercaPersonaje
```

```
AbrirCofre
tieneCercaCofre
tieneDinero tieneDineroArmas tieneDineroEscudos tieneDineroArmaduras
tieneDineroBruja tieneLlave tienePocionObj

Robar
tieneCercaPersonaje
tieneDinero tieneDineroArmas tieneDineroEscudos tieneDineroArmaduras
tieneDineroBruja tieneLlave tienePocionObj

LiberarPrisionero
estaCastillo tieneCercaPrisionero
tieneDinero tieneDineroArmas tieneDineroEscudos tieneDineroArmaduras
tieneDineroBruja
```

Como vemos, los operadores estarán almacenados en un fichero de texto. Para cada operador se indicará su nombre (primer renglón), su lista de precondiciones (segundo renglón), y su lista de postcondiciones (tercer renglón).

La precondición "vacio" hará referencia a un operador que no tendrá ninguna lista de precondiciones asociada. Es decir, será un operador que podrá aplicarse siempre.

Para buscar los operadores de forma cómoda y eficiente habrá que leerlos del fichero de texto y cargarlos en una lista de operadores. De este modo, para obtener un determinado operador habrá que buscar su nombre entre los elementos de la lista. Para ello, utilizaremos una clase en Python denominada Operador, que estará formada por los atributos nombre (identificador del operador), prec (lista de precondiciones), y post (lista de postcondiciones).

Para cargar los operadores en una lista utilizaremos una función que irá creando objetos de la clase Operador, para posteriormente asignarles el nombre, precondiciones y postcondiciones a partir del fichero de texto.

Cada una de las tres partes del operador, se cargará en tres cadenas de caracteres conforme se vayan leyendo del fichero. La cadena asociada a las precondiciones y postcondiciones se irá descomponiendo en los distintos elementos que se insertarán de uno en uno en las listas correspondientes.

La carga de los operadores se realizará antes de llamar a la función de toma de decisiones.

Cada personaje deliberativo tendrá un determinado objetivo a cumplir. En los reactivos los objetivos venían implícitos con la acción asociada a la regla que se verificase. En los agentes deliberativos, los objetivos vendrán definidos según las percepciones.

Los objetivos tendrán asociados un valor de preferencia. De este modo, un personaje cambiará de objetivo si percibe otro con una preferencia mayor (se considerarán más preferentes aquellos

objetivos con menor valor). Veamos los objetivos que podrá tener un personaje deliberativo:

- Comer (preferencia = 1)
- Beber (preferencia = 2)
- Descansar (preferencia = 3)
- Curarse (preferencia = 4)
- Defenderse (preferencia = 5)
- FinalizarMision (preferencia = 6)
- Estudiar (preferencia = 7)
- ComprarArma (preferencia = 8)
- ComprarEscudo (preferencia = 9)
- ComprarArmadura (preferencia = 10)

Como vemos, los objetivos se corresponderán con las acciones más importantes que podrán realizar los personajes. Veamos cómo se disparará cada objetivo:

- Para "comer" será necesario que el personaje perciba tener hambre (atributo "ham").

- Para "beber" será necesario que el personaje perciba tener sed (atributo "sed") y no tenga asignado el objetivo anterior.

- Para "descansar" será necesario que el personaje perciba estar cansado (atributo "ene") y no tenga asignado ningún objetivo anterior.

- Para "curarse" será necesario que el personaje perciba estar herido (atributo "pg") y que tenga un edificio "posada" en su radio de percepción, además de que no tenga asignado ningún objetivo anterior.

- Para "defenderse" será necesario que el personaje perciba estar herido (atributo "pg") y que no se tenga asignado ningún objetivo anterior.

- Para "finalizarMision" será necesario que el personaje perciba un objeto bruja en su radio de percepción y que no tenga asignado ningún objetivo anterior.

- Para "estudiar" será necesario que el personaje perciba un edificio "biblioteca" en su radio de percepción y que no se tenga asignado ningún objetivo anterior.

- Para "comprarArma" será necesario que el personaje perciba un edificio "tienda de armas" en su radio de percepción y que no se tenga asignado ningún objetivo anterior. También se tendrán en cuenta propiedades como el momento de reacción y carga asociada a la compra del arma.

- Para “comprarEscudo” será necesario que el personaje perciba un edificio “tienda de escudos” en su radio de percepción y que no se tenga asignado ningún objetivo anterior. Como en el caso previo, se tendrá también en cuenta el momento de reacción y la carga asociada.

- Para “comprarArmadura” será necesario que el personaje perciba un edificio “tienda de armaduras” en su radio de percepción y que no se tenga asignado ningún objetivo anterior. También se tendrá en cuenta el momento de reacción.

De este modo, si un personaje, por ejemplo, tuviera actualmente el objetivo “estudiar”, y le diese hambre, como el objetivo “hambre” es más preferente que “estudiar”, el personaje cambiaría de objetivo. Si en vez de sentir hambre el personaje detectase una tienda de armaduras en su entorno, no cambiaría de objetivo por tener éste menos preferencia que “estudiar”.

En realidad, estos objetivos serán operadores objetivo. Los objetivos verdaderos serán las postcondiciones que originen dichos operadores. A partir de ahora nos referiremos a estas postcondiciones como los objetivos principales a cumplir por el personaje.

El lanzamiento de uno de estos objetivos es lo que hará que el personaje delibere (mediante STRIPS) para obtener un operador que permita alcanzar dicho objetivo. Concretamente, habrá que buscar un operador (operador objetivo) que tenga en sus postcondiciones el objetivo a cumplir. Como dicho operador tendrá a su vez una serie de precondiciones que también tendrán que cumplirse, se buscará otro operador que las proporcione. De este modo, se encadenarán los operadores solucionando subobjetivos hasta que alguno se verifique. Cuando un operador pueda aplicarse, sus postcondiciones se insertarán en la lista de cambios. Este proceso se irá repitiendo hasta que al final se encuentre la solución para el objetivo principal del personaje. También puede pasar que no se encuentre solución, y el personaje no pueda cumplir su objetivo en ese turno. Para ello, se definirán acciones por defecto, que el personaje realizará cuando no pueda hacer otra cosa, aunque el objetivo permanecerá en espera de que el personaje pueda cumplirlo en otro turno.

STRIPS utiliza, para el encadenamiento de operadores, una pila. En dicha pila se insertarán los subobjetivos a cumplir. Dichos subobjetivos podrán ser operadores, precondiciones, o el objetivo principal (que será tomado como precondición).

Nuestro algoritmo STRIPS se basará en cuatro listas principalmente:

- Lista pila (utilizada para implementar la pila de subobjetivos)

- Lista estado (utilizada como lista de cambios o postcondiciones resultantes de los operadores)

- Lista acciones (utilizada para almacenar los operadores que se vayan aplicando)

- Lista descartes (utilizada para almacenar aquellos operadores que se descarten porque no puedan llegar a la solución)

Formalmente, STRIPS devuelve el plan (conjunto de operadores a aplicar) para alcanzar el objetivo. En nuestro caso, como pasaba con el algoritmo A*, necesitaremos una sola acción a realizar, es decir, un solo operador del plan. Por ello, nuestra función STRIPS devolverá aquel operador que dispare la primera condición del operador objetivo, ya que ése será el primer paso en el plan.

Para implementar los subobjetivos a insertar en la pila, utilizaremos una clase en Python denominada ElemStrips, que tendrá los siguientes atributos: tipo (utilizado para indicar el tipo de subobjetivo: 0 objetivo principal, 1 precondición del operador objetivo, 2 precondición, 3 operador), y el atributo valor (que indicará el valor del subobjetivo).

Los pasos del algoritmo serán los siguientes:

- Se crearán como listas vacías las listas pila, estado, acciones y descartes.
- Se insertará en la pila el objetivo principal a cumplir por el personaje.
- A continuación se entrará en un bucle.

  En cada iteración del bucle, se comprobará si la pila está vacía, en cuyo caso se devolverá como resultado el operador que se tenga almacenado como accion a realizar.

  En caso de que no esté vacía, se obtendrá el elemento tope de la pila. Si el tope es un operador, se aplicará su resultado, es decir, se insertarán sus postcondiciones en la lista estado, y se eliminará de la pila.

  Si el elemento tope es una precondición, se comprobará su aparición en la lista estado. En caso de que no esté, se verá si dicha precondición se puede enlazar con la descripción del mundo actual (listas del mundo). Ya sea en un caso o en otro, si se cumple la precondición, se eliminará de la pila y además, se comprobará si es una precondición del operador objetivo, en cuyo caso se almacenará la última acción de la lista de acciones como operador a realizar. Si, además, dicha acción es una acción distinta al operador objetivo, la función terminará devolviendo dicho operador como resultado.

  Si por el contrario, la precondición no se cumple, se buscará en la lista de operadores un operador no descartado que enlace con dicha precondición. Dicho operador se insertará en la pila, y además, se insertarán una a una todas sus precondiciones de forma que la última de ellas quede como tope de la pila. El operador se insertará también en la lista de acciones.

  Si no se encuentra ningún operador que enlace con la precondición, se comprobará si el tope es el objetivo principal, en cuyo caso la función terminará sin encontrar solución. En otro caso, se eliminarán todos los elementos de la pila hasta llegar al último operador. Dicho operador se eliminará de la lista de acciones y se insertará en la lista de descartes.

De este modo, nuestra función STRIPS devolverá un operador como resultado, en caso de solución, o un valor nulo si no es posible encontrar la solución.

Los operadores que devuelva nuestra función STRIPS serán las acciones que solicitará realizar el personaje. Coincidirán con las postcondiciones que aparecían en las reglas de los personajes reactivos. Al igual que en ese caso, se utilizará la misma función para convertir dichos operadores en valores enteros a enviar al servidor, junto con las coordenadas necesarias según el caso.

Las acciones que impliquen movimiento necesitarán obtener las coordenadas asociadas al camino óptimo. Para ello, al igual que pasaba con los agentes reactivos, utilizaremos el algoritmo A*.

Nuestra función deliberativa no sigue el método formal STRIPS paso a paso, ya que hay algunas diferencias para poder adaptarlo a nuestro proyecto. Las diferencias principales son:

- En el método formal se modifica el estado del mundo a cada paso. En nuestra función, se necesita una lista adicional de cambios (lista estado) para indicar las modificaciones realizadas en el mundo, ya que el estado del mundo es inmutable por el cliente.

- En el método formal, al irse modificando el estado del mundo cada vez, cuando se ve que un operador no conduce a solución, es necesario deshacer todas las modificaciones que haya realizado en el estado del mundo. En nuestra función, cuando un operador no es útil se inserta en la lista de descartes y se elimina de la pila sin tener que deshacer ningún cambio, ya que el mundo no se ha modificado.

- En el método formal se devuelve una lista de operadores (plan) para obtener la solución. En nuestra función, se devuelve un sólo operador, aquél que verifique la primera condición del operador objetivo. De este modo, la función termina antes que el método formal, ya que no necesita seguir analizando el resto del plan.

- El método formal puede producir ciclos. STRIPS utiliza operadores con variables, de forma que durante la ejecución del algoritmo, dichas variables se sustituyen por los valores que se correspondan. El problema reside en que una rápida instanciación de las variables puede producir una decisión incorrecta y tener que deshacer los cambios dando lugar a ciclos. En nuestra función, los operadores no tienen variables asociadas, ya que definen acciones muy concretas, y de este modo evitan este problema.

Finalmente, veamos un esquema del algoritmo del cliente deliberativo:

```
funcion tomaDecisionDeliberativa (listaOperadores, objetivoActual, tamanoMundo, mundo,
                                  turno, sockCliente):

    turnoCliente = obtieneTurnoCliente ()

    Si (turno == turnoCliente):

        objetivoActual = obtieneObjetivo (objetivoActual, mundo)

        Si (objetivoActual !=  0):

            accion = Strips (objetivoActual, listaOperadores, tamanoMundo. mundo)

            Si (accion == ‘‘):

                accion = obtieneAccionPorDefecto (tamanoMundo, mundo)

        Si no:

            accion =  obtieneAccionPorDefecto (tamanoMundo, mundo)

        enviaAccion (accion, tamanoMundo, mundo, sockCliente)

    Si no:
        envia (0, sockCliente)
```

Por último, nos queda explicar el agente deliberativo con memoria.

Al igual que con los reactivos, los agentes deliberativos tendrán que utilizar una lista de items para indicar aquellos elementos (cofres) ya conocidos y que no son útiles. Esto evitará acciones cíclicas cuando no se puedan utilizar las percepciones.

Sin embargo, los agentes deliberativos presentan otro problema que también puede dar lugar a realizar acciones cíclicas. El problema reside en los objetivos. Como ya sabemos, los objetivos se disparan según las percepciones y según un orden de preferencia. Supongamos, por ejemplo, que un personaje tiene asignado el objetivo de finalizar misión. Para ello es necesario que el personaje acepte una misión dada por una bruja. Dicho objetivo se disparará cuando el personaje vea a un objeto bruja en su radio. El personaje necesitará tener dinero para que la bruja pueda ofrecerle una misión. Si el personaje no tiene dinero, según nuestra función STRIPS, tendrá que buscar un medio de conseguirlo para cumplir su objetivo.

El proceso sería: obtener dinero, pagar bruja, obtener misión, finalizar misión.
Si el personaje entra en un castillo para obtener el dinero de la bruja, ésta ya no sería visible, ya que ahora estaríamos en el interior de un edificio. Por tanto, el personaje olvidaría su objetivo al entrar al castillo. Si ahora saliese del mismo, volvería a recordar su objetivo al percibir a la bruja. De este modo, el personaje estaría en un bucle entrando y saliendo hasta que un objetivo con mayor preferencia surgiese (como por ejemplo tener hambre).

De este caso deducimos dos cosas: en primer lugar, el personaje puede olvidar sus objetivos cuando las condiciones que los dispararon por primera vez salen del radio de percepción. En segundo lugar, el personaje puede realizar acciones cíclicas. Los dos aspectos producen un agente que no cumpliría el principio de diseño, ser proactivo. Un agente que no es capaz de mantener sus objetivo da lugar a comportamientos inestables.

Por ello, necesitaremos una memoria que almacene aquellos elementos necesarios para disparar el objetivo actual.

En el ejemplo anterior, si el personaje, al entrar al castillo, hubiese recordado que ya había visto a una bruja, habría seguido con el cumplimiento del objetivo aunque ya no percibiese a dicha bruja en su radio de percepción.

Para implementar este tipo de memoria utilizaremos una clase en Python denominada Objetivo con los siguientes atributos: id (identificador del objetivo), posx, posy (coordenadas (x,y) donde se encuentra el elemento necesario para disparar el objetivo). Estas coordenadas servirán para saber donde se encuentra el elemento para buscarlo después. De este modo, aunque el elemento no aparezca en el radio de percepción, se podrá acceder a él mediante las coordenadas almacenadas.
Cuando se dispare un objetivo por primera vez, se almacenará su información utilizando dicha clase.

Anteriormente, los objetivos venían definidos por un número entero (1-10) que indicaba su nivel de preferencia. Ahora, además de dicho número entero (identificador) vendrán definidas las coordenadas del elemento disparador. En caso de que un objetivo surja sin necesidad de que se perciba ningún elemento en concreto, dichas coordenadas serán nulas.

Llegados a este punto, veremos en la siguiente sección un estudio de comportamientos frente a los cuatro tipos de arquitecturas.

5.3 Comparativa

Como sabemos, hemos visto cuatro tipos de agentes inteligentes: reactivos, reactivos con memoria, deliberativos, deliberativos con memoria.

En realidad, podrían resumirse en dos tipos: reactivos y deliberativos, ya que la memoria es un mecanismo para perfeccionar el comportamiento de las dos arquitecturas principales.

Un agente reactivo, como ya hemos visto, no realiza ningún análisis. Podríamos decir que su comportamiento es el más básico de todos. Sólo percibe, y dispara la regla que se ajuste a su percepción. No tiene objetivos concretos, ya que éstos varían según el orden de las reglas. Su comportamiento es el de la fuerza bruta, percibe todas las situaciones posibles y realiza la acción que primero se cumpla sin tener un rumbo fijo. Por ejemplo, un agente reactivo abrirá un cofre si lo percibe y no tiene otra acción más importante que hacer (no se cumple otra regla). Sin embargo, no abrirá dicho cofre como medio para obtener un objetivo. Un agente deliberativo abrirá un cofre si le sirve para un determinado fin, por ejemplo, obtener dinero.

Mientras el agente reactivo sigue un esquema de percibir-actuar, el agente deliberativo sigue un esquema de medios-fines, por ello, encadena posibles acciones que le produzcan medios para el objetivo final. Las reglas del reactivo son independientes y no mantienen ningún encadenamiento.

El agente deliberativo parte siempre de unos objetivos a cumplir. En función de ellos y del estado actual del mundo analiza la sucesión de acciones que le pueden permitir el éxito.

De este modo, podríamos decir que el agente deliberativo tendrá un comportamiento más real que el reactivo. Será incluso más “inteligente” que el agente reactivo, desde el punto de vista de que se orientará por unas metas determinadas en función de su situación. En cambio, el reactivo actuará, no por metas, si no un poco al azar según el estado del mundo.

Mientras el reactivo tendrá en cuenta toda la información de sus percepciones, el deliberativo será más selectivo con dicha información, desechando aquella que no le interese en ese momento (porque no le sirva para obtener su objetivo).

Esta diferencia es muy teórica, porque en la práctica, muchas veces ambos tipos de agentes realizarán las mismas acciones (aunque no por los mismos motivos). Incluso puede ocurrir que un agente reactivo presente un estado más favorable que un deliberativo, o bien, obtenga mejores resultados. Por ejemplo, un agente reactivo, en un momento dado, puede tener más dinero en su inventario que un deliberativo porque ha abierto todos los cofres que había a su paso. Sin embargo, un deliberativo que sólo busque el dinero necesario para un cierto objetivo, no presentará un inventario tan “completo”. En general, en función de la situación del mundo y de los personajes, una arquitectura puede presentar mejor resultados que otra. Lo cual puede ser bastante ambiguo.

Básicamente, la diferencia radical reside en lo que mueve a cada tipo de agente a actuar de la forma en que lo hace. El agente reactivo actúa así porque dicha acción puede realizarla (coinciden las precondiciones). En cambio, el deliberativo actúa así porque persigue un fin, y dicha acción le proporciona un recurso que le hace estar más cerca del éxito de su objetivo.

Las versiones de ambas arquitecturas añadiendo memoria, perfeccionan aún más el comportamiento. Al evitar las acciones cíclicas, y ayudar a las percepciones con ciertos "recuerdos" hace que los agentes parezcan más reales.

En conclusión, desde el punto de vista del comportamiento "teórico", podríamos decir que los agentes deliberativos (y más los que utilizan memoria) cumplen mejor los objetivos del proyecto. Intentan realizar las acciones más correctas para cumplir sus objetivos (racionalidad), intentan simular un comportamiento inteligente mediante la planificación (inteligencia), y por último, junto a los reactivos, y a las versiones con/sin memoria, presentan distintos tipos de toma de decisiones frente al mundo.

Gráficamente:

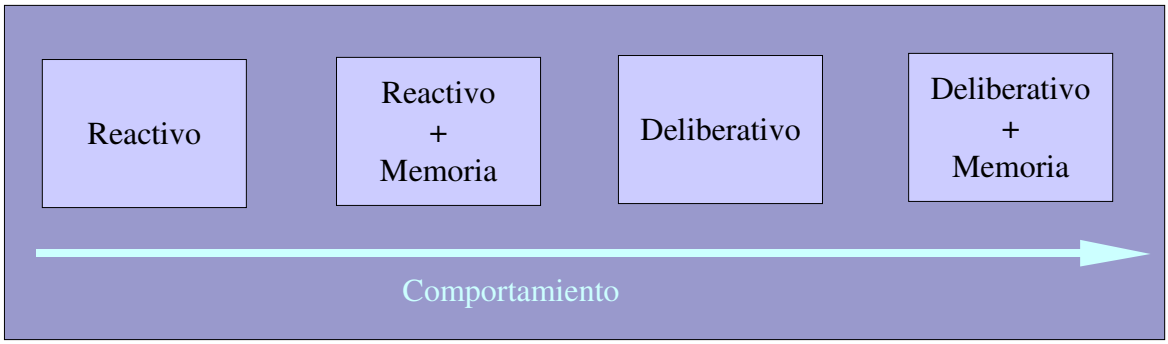


Fig. 28. Esquema de comportamiento de las arquitecturas implementadas

6 Conclusiones y optimizaciones futuras

De este modo, ya hemos explicado todo el funcionamiento del proyecto. El objetivo principal era obtener una simulación de un mundo ficticio en el que pudieran interactuar una serie de clientes, tanto usuarios humanos como computacionales, para obtener distintos comportamientos.

También hemos podido comprobar que un juego de rol es un sistema lo suficientemente complejo como para implementarlo mediante un sistema multiagente, donde cada agente realiza un análisis como toma de decisiones.

También queda justificada la arquitectura cliente/servidor necesaria para implementar la comunicación jugadores/Master.

En resumen, los puntos implementados más importantes han sido:

- Arquitectura de red cliente/servidor en C/C++ para soportar una estructura multijugador en la que el servidor gestione las peticiones de los clientes de forma ordenada (por turnos) y de modo concurrente (cada cliente es una hebra en continua ejecución).

- Estructuras de datos del mundo basadas en listas donde cada elemento se define por un identificador. Las listas se enlazan entre sí mediante los identificadores.

- Interfaz gráfica en OpenGL+SDL para interactuar con los clientes usuarios.

- Implementación del comportamiento inteligente en Python para interactuar con los clientes computacionales. Donde se distinguen arquitecturas basadas en un sistema de reglas (reactivo), arquitecturas basadas en planificación (deliberativo), y arquitecturas que incorporan algún tipo de almacenamiento adicional de datos (reactivo con memoria, deliberativo con memoria).

Como el cliente computacional está implementado en Python, será sencillo realizar futuras ampliaciones de los algoritmos de comportamiento. Modificando únicamente las funciones de toma de decisiones podrá cambiarse el tipo de inteligencia insertándose diferentes técnicas y heurísticas no implementadas en esta versión.

A continuación se muestran algunos aspectos que podrían complementar el proyecto en un futuro:

- Utilizar aprendizaje, de forma que los agentes reactivos puedan incorporar nuevas reglas a las iniciales, ya sea obteniéndolas mediante la experiencia (aquellas acciones nuevas que den mejores resultados) o deduciéndolas a partir de las iniciales.

- Utilizar agentes cooperativos, de modo que puedan compartir objetivos y colaboren para su cumplimiento. Para ello, se podría utilizar un protocolo de comunicación a través del servidor, de forma que se pueda intercambiar la información necesaria entre los agentes para llegar a un común acuerdo. También se podrían agrupar a los agentes por metas comunes en función de sus características, objetivos que tengan individualmente en ese momento, etc...
  Otra posibilidad es que en vez de que colaboren entre ellos, puedan competir influenciando negativamente en los objetivos de los adversarios.

- Implementar más características y reglas de rol, haciendo un sistema más complejo y restrictivo, además de añadir nuevas posibilidades de juego como por ejemplo, poder realizar hechizos, tener un conjunto variado de habilidades más amplio, posibilidad de venenos y antídotos, poder presentar un terreno no uniforme (precipicios, montañas), etc...

APENDICES

I. Clases implementadas

La especificación de las clases comentadas a lo largo del proyecto se puede ver a continuación:

/**/

Clase **ConexionServidor**:
Clase C++ que implementa la interfaz de conexión del servidor con los clientes

- **ConexionServidor ()**
  Constructor de la clase que crea una interfaz socket para el modo pasivo

- **~ConexionServidor ()**
  Destructor de la clase que cierra el socket creado en el constructor

- **asignaConexion ()**
  Método que asigna los parámetros especificados al socket creado

- **abreConexion ()**
  Método que abre en modo escucha el socket creado

- **aceptaCliente ()**
  Método que devuelve un nuevo socket asociado al cliente conectado

- **desconectaCliente ()**
  Método que cierra el socket del cliente especificado

- **enviaDatos ()**
  Método que envía la información especificada a través del socket del cliente

- **recibeDatos ()**
  Método que recibe información a través del socket del cliente

/**/

/**/

Clase **Envio:**

Clase C++ que implementa una estructura formada por dos campos:
sock: socket del cliente
env: flag que indica qué datos son los enviados al cliente

- **Envio ()**
  Constructor de la clase que inicializa los atributos de la misma

- **~Envio ()**
  Destructor de la clase

- **obtieneSock ()**
  Método que devuelve el socket del cliente

- **obtieneEnv ()**
  Método que devuelve el flag de envío

- **reAsignaSock ()**
  Método que actualiza el socket del cliente con un nuevo valor

- **reAsignaEnv ()**
  Método que actualiza el flag de envío con un nuevo valor

/**/

Clase **ConexionCliente**:

Clase C++ que implementa la interfaz de conexión del cliente con el servidor

- **ConexionCliente ()**
  Constructor de la clase que crea una interfaz socket para la conexión con el servidor

- **~ConexionCliente ()**
  Destructor de la clase que cierra el socket creado en el constructor

- **conectaAServidor ()**
  Método que conecta el socket creado a un servidor cuyos parámetros se especifican

- **enviaDatos ()**
  Método que envía la información especificada a través del socket creado

- **recibeDatos ()**
  Método que recibe información a través del socket creado

/**/

Clase Paramétrica **Lista:**
Clase C++ que implementa una lista de elementos de cualquier tipo

- **Lista ()**
  Constructor de la clase que inicializa una lista vacía

- **~Lista ()**
  Destructor de la clase

- **obtieneTamano ()**
  Método que devuelve el número de elementos de la lista

- **obtieneElemento ()**
  Método que devuelve el elemento cuya posición se especifica

- **insertaElemento ()**
  Método que inserta el elemento especificado al final de la lista

- **eliminaElemento ()**
  Método que elimina el elemento cuya posición se especifica

- **reAsignaElemento ()**
  Método que actualiza el elemento cuya posición se especifica con un nuevo valor

/**/

Clase **Personaje, Objeto, Edificio, Casilla:**
En este caso, hemos agrupado estas clases de C++ por presentar una especificación parecida.
Clase Personaje: Implementa una estructura con los atributos asociados a un personaje del juego
Clase Objeto: Implementa una estructura con los atributos asociados a un objeto del juego
Clase Edificio: Implementa una estructura con los atributos asociados a un edificio del juego
Clase Casilla: Implementa una estructura con los atributos asociados a una casilla del juego

- **Constructor ()**
  Constructor de la clase Personaje, Objeto, Edificio o Casilla que inicializa sus valores a nulo

- **~Destructor ()**
  Destructor de la clase Personaje, Objeto, Edificio o Casilla

- **obtieneNAtributos ()**
  Método que devuelve el número de campos que tenga el elemento

- **obtieneAtributo ()**
  Método que devuelve el valor del campo especificado del elemento

- **reAsignaAtributo ()**
  Método que actualiza el campo especificado del elemento con un nuevo valor

La clase Personaje, al tener un campo inventario (que será en realidad un vector de elementos), dispondrá además de los siguientes métodos adicionales:

- **obtieneTamEqu ()**
  Método que devuelve el número de items del equipo del personaje

- **obtieneEqu ()**
  Método que devuelve el item del equipo cuya posición se especifica

- **reAsignaEqu ()**
  Método que actualiza el item del equipo cuya posición se especifica con un nuevo valor

/**/

Clase **Elemento:**

Clase C++ que implementa una estructura con los siguientes campos:
tipo: Tipo de elemento (objeto o personaje)
posicion: Posición del elemento en su lista correspondiente

- **Elemento ()**
  Constructor de la clase Elemento que inicializa sus valores a nulo

- **~Elemento ()**
  Destructor de la clase Elemento

- **obtieneTipo ()**
  Método que devuelve el tipo del elemento (0 personaje, 1 objeto)

- **reAsignaTipo ()**
  Método que actualiza con un nuevo valor el tipo del elemento

- **obtienePosicion ()**
  Método que devuelve el índice del elemento en la lista correspondiente

- **reAsignaPosicion ()**
  Método que actualiza con un nuevo valor la posición del elemento

/**/

clase **Textura:**
Clase C++ que crea una textura imagen como vector de píxels

- **Textura ()**
  Constructor de la clase que carga en memoria la imagen cuyo nombre se especifica

- **~Textura ()**
  Destructor de la clase que libera la memoria almacenada

- **obtienePixels ()**
  Método que devuelve la imagen cargada

- **obtieneAncho ()**
  Método que devuelve el ancho de la imagen cargada

- **obtieneAlto ()**
  Método que devuelve el alto de la imagen cargada

/**/

clase **EntornoG:**
Clase C++ que implementa toda la gestión de la interfaz gráfica

- **EntornoG ()**
  Constructor de la clase que inicializa todos los parámetros de la aplicación

- **~EntornoG ()**
  Destructor de la clase

- **iniciaEntorno ()**
  Método que inicia tanto la librería SDL como la librería OpenGL

- **visualizaEntorno ()**
  Método principal encargado de gestionar el bucle de eventos de la ventana. Constituye el cliente gráfico en sí mismo. La especificación del algoritmo anteriormente vista se correspondería con este método. Incluye la visualización del menú de inicio, la conexión con el servidor, la recepción/envío de información, la visualización e interacción con el mundo visual, y el cierre de la aplicación.

- **visualizaMenuStart ()**
  Método que se encarga de gestionar los eventos del menú de inicio. Será llamado por el método visualizaEntorno.

- **visualizaOver ()**
  Método que visualiza una pantalla final al cierre de la aplicación. Será llamado por el método visualizaEntorno.

- **GL_init ()**
  Método que implementa la función de inicialización de OpenGL. Será llamado por el método iniciaEntorno.

- **GL_reshape ()**
  Método que implementa la función de redimensionado de OpenGL. Este método se utiliza cuando se produce un evento de redimensionado.

- **GL_display ()**
  Método encargado de dibujar el mundo en pantalla utilizando funciones de OpenGL. Transforma los elementos de las listas del mundo en polígonos. Por tanto, es el encargado de dibujar casillas, edificios, personajes y objetos (éstos dos últimos por orden de profundidad). Este método se utiliza cuando se produce un evento expose o de redimensionado. También se usa cuando se reciben las nuevas listas del mundo para refrescar el radio de percepción.

- **GL_displayMenuStart ()**
  Método encargado de dibujar los elementos del menú de opciones inicial. Este método se llama cuando se origina un evento que necesite un refresco de la ventana.

- **GL_displayOver ()**
  Método encargado de dibujar los elementos de la pantalla de cierre. Al igual que los métodos anteriores, se utiliza cuando se produce un evento de redibujado.

- **GL_cursor ()**
  Método encargado de dibujar un señalizador que apunte al personaje. Será llamado desde GL_display.

- **GL_caracteristicas ()**
  Método encargado de dibujar el cuadro de características que incluirá los atributos del personaje actualizados. Le servirá al usuario para la toma de decisiones. Será llamado desde GL_display.

- **GL_menuAcciones ()**
  Método encargado de dibujar el menú de acciones que le permitirán al usuario seleccionar la acción a realizar. Será llamado desde GL_display.

- **GL_imprimeTexto ()**
  Método que dibujará una cadena de texto en pantalla utilizando funciones de OpenGL y glf.
  Será llamado por el método anterior.

- **GL_poligono ()**
  Método encargado de dibujar un rectángulo en OpenGL con una textura RGB asociada.
  Se llamará a este método cada vez que haya que dibujar elementos no transparentes.

- **GL_poligonoT ()**
  Método encargado de dibujar un rectángulo en OpenGL con una textura RGBA asociada.
  Se llamará a este método cada vez que haya que dibujar elementos transparentes.

- **GL_cubo ()**
  Método encargado de dibujar un cubo en OpenGL utilizando texturas RGB en cada cara.
  Será llamado cada vez que se dibujen elementos cúbicos.

- **SDL_init ()**
  Método que implementa la función de inicialización de SDL. En realidad es el encargado de crear la ventana de la aplicación. Será llamado por el método iniciaEntorno.

- **SDL_key ()**
  Método encargado de analizar las teclas pulsadas por el usuario en el bucle de eventos del método visualizaEntorno. Asociará a cada tecla un valor entero que enviará al servidor como acción solicitada.

- **cargaTamanoMundo ()**
  Método encargado de almacenar el tamaño del mundo enviado por el servidor. Será llamado por el método visualizaEntorno.

- **cargaMundo ()**
  Método encargado de almacenar las listas de datos del mundo enviadas por el servidor. Almacenará la lista de casillas, edificios, objetos, personajes y elementos. Será llamado por el método visualizaEntorno.

- **cargaTexturas ()**
  Método encargado de almacenar una lista con todas las texturas a utilizar en el mundo. De este modo, será fácil obtener una determinada textura para aplicársela a un polígono. Será llamado por el método visualizaEntorno.

- **enviaCoordenadas ()**
  Método encargado de enviar al servidor unas coordenadas (x,y) del mundo en función del tipo de acción solicitado. Esté método también se llamará desde visualizaEntorno.

/**********/

clase N**odoA:**
Clase Python que implementa un nodo A* con los siguientes campos:
id: Identificador del nodo
idpa: Identificador del nodo padre
posx: Coordenada x
posy: Coordenada y
oper: Operador de desplazamiento utilizado
g: Componente g
h: Componente h
f: Componente f

- **obtieneAtributo ()**
  Método que devuelve el valor del atributo especificado

- **reAsignaAtributo ()**
  Método que actualiza con un nuevo valor el atributo indicado

/**********/

clase **Operador:**
Clase Python que implementa una estructura con los siguientes campos:
nombre: Nombre del operador
prec: Precondiciones del operador
post: Postcondiciones del operador

- **init ()**
  Inicializador de la clase que crea las listas vacías para prec y post

- **obtieneNombre ()**
  Método que devuelve el nombre del operador

- **obtieneNPrec ()**
  Método que devuelve el tamaño de la lista de precondiciones del operador

- **obtieneNPost ()**
  Método que devuelve el tamaño de la lista de postcondiciones del operador

- **obtienePrecond ()**
  Método que devuelve el valor de la precondición especificada del operador

- **obtienePrec ()**
  Método que devuelve la lista de precondiciones del operador

- **obtienePostcond ()**
  Método que devuelve el valor de la postcondición especificada del operador

- **obtienePost ()**
  Método que devuelve la lista de postcondiciones del operador

- **reAsignaNombre ()**
  Método que actualiza el nombre del operador con un nuevo valor

- **insertaPrecond ()**
  Método que inserta una nueva precondición en la lista de precondiciones del operador

- **insertaPostcond ()**
  Método que inserta una nueva postcondición en la lista de postcondiciones del operador

/**/

clase **ElemStrips:**
Clase Python que implementa una estructura con los siguientes campos:
tipo: Tipo de elemento para la pila de STRIPS
valor: Valor del elemento para la pila de STRIPS

- **init ()**
  Inicializador de la clase

- **obtieneAtributo ()**
  Método que devuelve el valor del atributo especificado

- **reAsignaAtributo ()**
  Método que actualiza con un nuevo valor el atributo especificado

/**/

clase **Item:**

Clase Python que implementa una estructura con los siguientes campos:
id: Identificador del objeto contenedor
elem: Tipo de elemento que contiene

- **obtieneAtributo ()**
  Método que devuelve el valor del atributo especificado

- **reAsignaAtributo ()**
  Método que actualiza con un nuevo valor el atributo especificado

/**/

clase **Objetivo:**

Clase Python que implementa una estructura con los siguientes campos:
id: Identificador del objetivo
posx: Coordenada x del elemento disparador del objetivo
posy: Coordenada y del elemento disparador del objetivo

- **obtieneAtributo ()**
  Método que devuelve el valor del atributo especificado

- **reAsignaAtributo ()**
  Método que actualiza con un nuevo valor el atributo especificado

/**/

II. Manual de usuario

En primer lugar, será necesario crear el mundo del juego. Para ello, dentro del directorio *mundo* podemos encontrar un fichero Python ejecutable llamado *creaMundo.py* que será el encargado de crear las listas *objetos.dat, edificios.dat* y *casillas.dat*. Para ejecutar dicho fichero bastará con escribir su nombre indicando la extensión (.py).

Cada vez que se ejecute dicho fichero se generarán mundos diferentes, ya que las posiciones de los elementos son aleatorias.

El fichero ejecutable podrá modificarse para definir un mundo más grande o más pequeño con más o menos elementos. Es decir, siempre se podrá ampliar o reducir las dimensiones del mundo, al igual que el número de elementos de cada tipo, pero no se podrán crear nuevos tipos de elementos, ya que en caso de hacerlo, habría que cambiar el resto del proyecto para adaptar los nuevos cambios. Lo mismo ocurriría si se insertasen o borrasen los atributos de un elemento. Será necesario respetar la estructura ya definida.

Una vez creado el mundo, se podrá ejecutar el juego. Lo primero consistirá en lanzar el programa *servidor*, que se quedará en espera de que se conecten los jugadores, ya sean usuarios PJs o PNJs. El servidor, al ejecutarse, habrá realizado una copia del fichero *objetos.dat* para poder modificarlo durante el juego.
Para interrumpir el servidor será necesario matar el proceso mediante <CTRL> - C.

Empecemos a describir el funcionamiento de los PJs.

Para que un jugador pueda conectarse al servidor tendrá que ejecutar el programa *clientePJ*. A continuación, el usuario tendrá que escribir la dirección IP donde se encuentre el servidor. Por ejemplo:

> clientePJ
> *Conexión con el servidor:* 127.0.0.1

Seguidamente, aparecerá una ventana con una pantalla de inicio en la que el usuario podrá seleccionar la raza del personaje con la que desee jugar.

Para seleccionar la especie del personaje, tendrá que desplazar el icono en forma de espada hacia arriba o hacia abajo utilizando las teclas del cursor. Una vez seleccionada la especie, tendrá que darle a la tecla <RETURN>.

Fig. 29. Pantalla de inicio

De este modo, aparecerá en la ventana de la aplicación un mundo con el personaje seleccionado en la posición central.

El personaje tendrá un puntero rojo señalándolo. Dicho puntero, además de indicar al personaje del jugador, servirá para apuntar a ciertos elementos con los cuales se quiera actuar. Ya hablaremos de esto cuando expliquemos las acciones del personaje.

Para ver las características del personaje, será necesario pulsar la tecla <CTRL> derecha. De este modo, aparecerá en pantalla un cuadro semitransparente con la siguiente información:

- Un bloque de atributos de salud en color verde indicando los puntos de golpe (P.Golpe), energía (Energia), hambre (Hambre) y sed (Sed) actuales del personaje. Se mostrarán tanto los valores reales como los valores máximos en forma de fracción separados por (/).

- Un bloque de atributos de características en color cyan indicando la fuerza (*Fue*), constitución (*Con*), tamaño (*Tam*), inteligencia (*Inte*), destreza (*Des*), habilidad de sigilo (*Sigilo*), habilidad de percepción (*Percepc*), número de estudios realizados o nivel de estudios (*N.Estu*), número de entrenamientos realizados o nivel de entrenamiento (*N.Entr*), misión actual (*Mision*), momento de reacción (*M.Reacc*), y número de acciones por turno (*N.Turnos*).

- Un bloque de elementos en color blanco indicando el inventario del personaje. Mostrará la armadura, arma, escudo, llave, número de monedas de oro, y poción que tenga el personaje en ese momento.
  Tanto las armas, escudos y armaduras como la llave se mostrarán en forma de icono. El número de monedas de oro y la poción vendrán expresados por números (las monedas en cantidades, y las pociones por su número asociado).

- Finalmente, se mostrará en rojo el turno del jugador activo en ese momento. Coincidirá con el valor del socket del cliente activo.
  Cuando haya varios jugadores en el juego, sólo podrá actuar cada uno cuando le llegue su turno, mientras tanto, tendrán que esperar a que terminen el resto de los jugadores.
  Siempre habrá un tiempo máximo por turno, y éste será de 5 segundos por jugador.

Esta información variará a lo largo del juego, según las acciones del personaje o de otros personajes.

Para no visualizar el cuadro de características se utilizará nuevamente <CTRL>.

Fig. 30. Ejemplo de cuadro de características

A continuación, vamos a ver cómo realizar las distintas acciones del personaje.

(1) Moverse por el mundo

Para ello, el jugador tendrá que utilizar las teclas del cursor (⇐ ⇑ ⇒ ⇓).
A cada movimiento, el personaje avanzará una casilla y se irá desplazando el radio de percepción del mundo modo de scroll.

(2) Entrar/Salir de edificios

Para entrar en algún edificio, el jugador tendrá que situar a su personaje frente a la fachada del mismo y avanzar hacia su interior. En el caso de tratarse de castillos, necesitará tener una llave en su inventario para poder entrar.

Fig. 31. Ejemplo de personaje frente a la fachada de un castillo

Para salir del interior de algún edificio, el jugador tendrá que avanzar su personaje hacia la salida situada en la parte inferior del recinto.

Habrá seis tipos de edificios: castillos, posadas, bibliotecas, tiendas de armas, tiendas de escudos, y tiendas de armaduras. Excepto los castillos, todos los edificios presentarán una forma similar. Para distinguir el tipo de edificio habrá que fijarse en el banderín de la fachada. De este modo:

Posada Biblioteca T. Armas T. Escudos T. Armaduras

Cuando el personaje se encuentre en el interior de los castillos, podrán aparecer trampas ocultas que o bien se activarán al pasar el personaje por encima de ellas, produciendo un decremento en los puntos de golpe, o bien, en función de su habilidad de percepción, se activarán antes de que pase por ellas.

Fig. 32. Ejemplo de personaje junto a una trampa a la derecha

(3) Descansar/Gastar turno

Cada jugador tendrá un número de acciones a realizar por turno indicadas en el atributo *N.Turnos*. De este modo, podrá realizar varias acciones en su turno.

Sin embargo, podría perder todo su turno descansando, lo que provocaría gastar todas las acciones disponibles que tenga en su turno en ese momento. El resultado sería incrementar el atributo de energía al máximo. Para ello, el jugador tendrá que utilizar la tecla <RETURN>.

(4) Interacciones con los demás personajes

Para interactuar con los demás personajes, el jugador tendrá que utilizar el puntero indicador y apuntar al personaje con el que quiera interactuar.

Para utilizar el puntero, el jugador tendrá que pulsar la tecla <ESPACIO>.
Se podrá comprobar que el puntero cambiará a un valor verde.
Para moverlo habrá que utilizar las teclas del cursor. Sólo se permitirá mover el puntero a una posición adyacente al jugador, por lo que tendrá que estar al lado del otro personaje para que el puntero lo señale correctamente.
Para cancelar la selección, el usuario tendrá que volver a pulsar la tecla <ESPACIO>.

Una vez que se haya apuntado al personaje adversario, el usuario tendrá que pulsar la tecla <RETURN>, para que aparezca un menú de acciones en un cuadro semitransparente que mostrará las acciones disponibles. El color del puntero cambiará nuevamente al color rojo.

Para seleccionar una acción del menú, será necesario mover el icono de la espada utilizando las teclas del cursor. Una vez seleccionada la acción, se tendrá que validar utilizando otra vez la tecla <RETURN>. De este modo, desaparecerá el menú de acciones y se realizará la que se haya seleccionado. Este procedimiento será necesario cada vez que se quiera interactuar con algún elemento.

Cada menú tendrá una acción de salir que se utilizará en caso de que no se quiera interactuar con el elemento.

Las acciones asociadas a la interacción con los demás personajes serán las de atacar o robar. La primera, en caso de éxito, restará puntos de golpe al adversario. La segunda, en caso de éxito, incrementará con nuevos elementos el inventario del personaje.

Fig. 33. Ejemplo de menú de acciones para actuar con el personaje de la derecha

(5) Comer

Para comer, el jugador tendrá dos opciones disponibles: apuntar a un tarro de miel o apuntar a unas setas. Tanto en un caso como en otro, se abrirá el menú de acciones comentado y el personaje podrá incrementar su atributo *Hambre*.

Recordemos que los tarros de miel restablecen todo el hambre pero sólo sirven una vez. Las setas, sin embargo, no se gastan nunca pero incrementan menos puntos de hambre.

Fig. 34. Ejemplo de menú de acciones para actuar con un tarro de miel (también podría actuar con las setas de la derecha)

(6) Beber

Para beber, el jugador tendrá que apuntar a una poza de agua. Como resultado, se mostrará el menú de acciones correspondiente. Se restablecerán todos los puntos de sed.

Fig. 35. Ejemplo de menú de acciones para actuar con una poza de agua

(7) Abrir cofres

Para abrir cofres, será necesario que el jugador los apunte y seleccione en el menú la acción de abrirlos. Como resultado, se modificará el inventario del personaje, a no ser de que ya contenga el elemento del cofre.

En un cofre podrá haber los siguientes elementos:

- Monedas de oro (en cantidades de 50 unidades)
- Llave (para abrir los castillos)
- Poción (numerada del 1 al 10)

Fig. 36. Ejemplo de menú de acciones para actuar con un cofre

Recordemos que cuando un personaje (PJ o PNJ) desaparezca del juego, su inventario se quedará en el mundo en forma de cofres. Los únicos elementos que no se tendrán en cuenta para esto serán las armas, escudos y armaduras, ya que no tendría sentido porque dichos items son ilimitados en las tiendas.

(8) Entrenar/Aceptar misiones

Para ello, el personaje no deberá tener asignada ninguna misión anterior en ese momento. Tendrá que apuntar a una bruja y aceptar la misión asociada con el consiguiente pago. Cada misión tendrá un coste de 50 monedas de oro por nivel. El jugador tendrá que pagar la cantidad de multiplicar el valor de su atributo *N.Entr* por 50. La misión aparecerá en el atributo *Misión* del cuadro de características y se corresponderá con un número de poción que el personaje deberá de encontrar.
Cada vez que el personaje entrene se incrementará el atributo *N.Entr*.

Fig. 37. Ejemplo de menú de acciones para actuar con una bruja

(9) Finalizar misión

Para finalizar la misión, el personaje deberá tener la poción correspondiente y entrar en un castillo. Una vez dentro, tendrá que situarse en el centro de un portal mágico (entre dos gárgolas) y perder turno allí. Se abrirá un menú de acciones para ello. Como resultado, se incrementarán las características del personaje en un punto, además de restablecer los atributos de salud. El atributo *Misión* se pondrá a 0.

Fig. 38. Ejemplo de menú de acciones para actuar con un portal

(10) Liberar prisioneros

Para liberar prisioneros, el jugador tendrá que estar en el interior de un castillo y apuntar a uno de los esclavos atados. Una vez seleccionada la acción de liberarlos, se incrementará el número de monedas de oro a 50 unidades.

Fig. 39. Ejemplo de menú de acciones para actuar con un prisionero

(11) Curarse en Posadas

Para restablecer todos los atributos de salud al máximo, el personaje tendrá que entrar a una posada y apuntar a la posadera para aceptar la acción de descansar. Supondrá un coste de 50 monedas de oro.

Fig. 40. Ejemplo de menú de acciones para actuar con una posadera

(12) Estudiar

Para estudiar, o mejor dicho, incrementar las habilidades de percepción y sigilo en 1 punto, será necesario que el jugador entre en una biblioteca y apunte a la bibliotecaria. Supondrá un coste de 50 monedas de oro. Además, afectará a la salud del personaje que se verá decrementada según el nivel de estudios, indicado en el atributo *N.Estu*. Cada vez que estudie, dicho atributo se incrementará.

Fig. 41. Ejemplo de menú de acciones para actuar con una bibliotecaria

(13) Comprar armas/escudos/armaduras

Para ello, bastará con entrar en la tienda correspondiente y apuntar al vendedor. Según la tienda se abrirá un menú con distintas acciones asociadas a los elementos a comprar.

Un personaje no podrá comprar un objeto que ya posea, pero sí podrá comprar otro diferente en la misma tienda. El resultado será la venta del que tiene y el pago del nuevo. El inventario se modificará.

Fig. 42. Ejemplo de menú de acciones para actuar con un vendedor de armaduras

Las armas y escudos afectarán negativamente al momento de reacción y a la carga del personaje, por tratarse de un estorbo tanto para actuar (decrementan los reflejos y el personaje realiza menos acciones por turno) como para llevar en el inventario (producen un mayor peso y el personaje se cansa más). Las armaduras sólo afectarán negativamente al momento de reacción.

A la hora de comprar, los PNJs obtendrán aquellos elementos más caros que puedan permitirse sin que supongan un elevado coste de carga y momento de reacción.

Es necesario indicar que el personaje mono sólo podrá moverse/comer/beber/descansar/atacar/robar, restringiéndose el resto de acciones.

Para conectar más jugadores se realizará como ya hemos comentado, lanzando el programa *clientePJ* e indicando la dirección IP del servidor, posteriormente.

Los jugadores podrán actuar por turnos de forma ordenada según su momento de reacción.

Veamos ahora cómo ejecutar los PNJs.

Para ello tendremos que entrar en el directorio *inteligencia*, y a su vez, según el tipo de PNJ que queramos conectar (con/sin memoria) entraremos en el subdirectorio correspondiente.

Para ejecutar un PNJ será necesario ejecutar el programa Python *clientePNJ.py* e indicar la especie del personaje y el tipo de comportamiento (reactivo/deliberativo) en la línea de comandos.

La especie será un valor comprendido entre 0 y 4 con la siguiente correspondencia:
0->Humano, 1->Hobbit, 2->Elfo, 3->Gigante, 4->Mono. Por defecto será éste último.

El tipo de comportamiento será un valor comprendido entre 0 y 1 con la siguiente correspondencia:
0->Reactivo, 1->Deliberativo. Por defecto será reactivo.

Veamos un ejemplo:

> clientePNJ.py 1 2
Ejecutará un PNJ Hobbit deliberativo

> clientePNJ.py 3
Ejecutará un PNJ Gigante reactivo

> clientePNJ.py
Ejecutará un PNJ Mono reactivo

El personaje mono siempre será reactivo. En caso de especificarlo como deliberativo, se ajustaría por defecto a reactivo.

Una vez ejecutado el programa, el PNJ entrará en el juego. Sus acciones no serán visibles a no ser que se tenga un PJ en la partida, que vería al PNJ actuando en caso de que coincidiese en su radio.

Las características, así como las reglas o acciones deliberativas realizadas del PNJ, se mostrarán en la shell.

Para desconectar un personaje PJ podrá utilizarse la tecla <ESC> o cerrar la ventana.
Para desconectar un personaje PNJ se matará el proceso con <CTRL> - C.

Ya sea desconectando al personaje PJ o muriendo éste por llegar sus pg a 0, aparecerá una pantalla de finalización. Para cerrarla se utilizará el mecanismo anterior.

Fig. 43. Pantalla final

Bibliografía:

- W. Richard Stevens, Advanced Programming in the Unix Environment., Addison-Wesley, 1992
- F. M. Márquez, Unix: Programación Avanzada, Segunda Edición. Ra-Ma, 1996
- William Stallings, Comunicaciones y Redes de Computadores, Quinta Edición. Prentice-Hall, 1997
- Mark Lutz and David Ascher, Learning Python, O'Reilly
- Python 2.1 Bible, Dave Brueck and Stephen Tanner, Hungry Minds
- Python Tutorial, Guido van Rossum, Fred L. Drake, Jr., editor, Release 2.3, 2003
- G.Brassard, P.Bratley, Fundamentos de Algoritmia, Prentice-Hall, 1997
- [Nilsson] Nils J. Nilsson, Inteligencia Artificial: una nueva síntesis, Mc Graw Hill, 2000
- [Foley] J.~Foley,Van Dam, S. Feiner,~J. Hughes, Introduction to Computer Graphics, Addison-Wesley, 1996.
- Addison-Wesley Publishing Company, OpenGL Programming Guide, Silicon Graphics, 1997
- Mark Segal, Kurt Akeley, The OpenGL Graphics System: A specification, Silicon Graphics, 1992-2003
- Una aproximación a OpenGL, A. Jaspe Villanueva, J. Dorado de la calle
- Game Development and Production, Erick Bethke, WordWare Game Developer"s Library
- RuneQuest básico, Greg Stafford, 1988