Parte I

Anexos

Apéndice A

Manual Pogamut 3

A.1. Instalación y Servidor

A.1.1. Instalación

En esta sección se describe el proceso de instalación, así como el software necesario para la misma.

Prerrequisitos

- Una copia del videojuego Unreal Tournament 2004 (UT2004)
- Unreal Engine 2 Runtime (UE2) and Unreal Development Kit (UDK)
  - UE2 es gratuito para uso no comercial. Es posible descargar la versión Demo de http://apacudn.epicgames.com/Two/UnrealEngine2Runtime22262002.html
  - UDK is una versión libre de Unreal Engine 3. Está incluido en el paquete de instalación de Pogamut 3.
- JDK y Netbeans
  - Descarga conjunta en http://www.oracle.com/technetwork/java/javase/downloads/jdk netbeans- jsp-142931.html
- Pogamut 3
  - Descargar "Pogamut 3 Java instaler full" en la sección Downloads de la página oficial http://diana.ms.mff.cuni.cz/main/tiki-index.php?page=Download

Proceso de Instalación

Para no tener problemas en el proceso de instalación de todos los elementos que necesitamos, seguir el siguiente orden, ya que al instalar Pogamut se nos pedirá la ubicación de UT2004, UE2 y Netbeans.

1. Instalar el videojuego Unreal Tournament 2004.

2. Instalar JDK y NetBeans.

    - En Windows 7 instalar Netbeans directamente en C:/ en lugar de en la ubicación por defecto, debido a problemas con los permisos.

3. Instalar Unreal Engine 2 Runtime.

4. Por último, instalar Pogamut.

    - Realizar la instalación como aparece por defecto, es decir, con todos los elementos seleccionados, ya que todos son necesarios para que todo funcione correctamente.
    - Si hemos instalado los programas en la ruta por defecto, el instalador de Pogamut los detectará automáticamente. En caso contrario tendremos que buscar la ruta en la que hayamos instalado el programa requerido por el proceso de instalación.

A.1.2. Ejecución del bot en UT2004

En esta sección se describe cómo configurar correctamente el servidor de UT2004, de forma que luego podamos conectar nuestro bot al mismo sin problemas. También se describe el proceso de conexión de dicho bot al servidor utilizando NetBeans.

Configuración del servidor

Para crear un servidor en UT2004, la forma más sencilla es hacerlo desde el propio juego. Para ello, abrimos UT2004 y seleccionamos la opción Alojar Partida. Una vez hecho esto, debemos seleccionar el tipo de juego. Para que éste sea compatible con nuestro bot, debemos elegir uno de los tipos de juego personalizados, situados al final de la lista (Figura A.1). Lo más común es seleccionar el tipo "GameBots DeathMatch".

Una vez hecho esto, configuramos el juego a nuestro gusto. En la sección "Reglas de servidor" debemos tener siempre en cuenta las siguiente consideraciones (Figura A.2):

- Activar la opción "Servidor LAN" para mejorar el rendimiento si estamos trabajando con ordenadores conectados en LAN.
- Deseleccionar la opción "Anunciar servidor" para evitar problemas con copias ilegales de UT2004.
- Activar "Ignore UTAN Bans" para evitar problemas con copias ilegales de UT2004.

Por último, podemos crear dos tipos de servidores:

- Mixto: nada más crear el servidor, pasaríamos directamente a jugar en él y, posteriormente, podríamos conectar otros bots contra los que jugaríamos. Esta opción no permite visualizar los acontecimientos que ocurren en el servidor, como es el caso del "servidor dedicado"
- Dedicado: este tipo de servidor permite visualizar en modo texto la configuración del servidor y los acontecimientos que ocurren, tales como la conexión de un jugador o bot, la muerte y resurrección de los mismos, etc. Si queremos, desde el mismo ordenador, conectarnos para jugar en el servidor, sólo tenemos que ejecutar otra vez UT2004 y conectarnos a él como jugadores.

Figura A.1: Modos de juego GameBots

Figura A.2: Configuración del servidor

Conexión del bot al servidor

Lo primero que debemos hacer es abrir nuestro proyecto en NetBeans. Una opción inteligente sería construir nuestro bot sobre "EmptyBot", un proyecto de ejemplo incluido por Pogamut,

que sería el equivalente al "Hello world" de los bots. Para ello seleccionamos "nuevo proyecto" en NetBeans, seleccionamos 00-Emptybot situado en la carpeta Samples/Pogamut UT2004 y le damos un nombre. Nuestro proyecto quedará guardado por defecto en Documentos/NetBeansProyects. La figura A.3 muestra como acceder al código fuente de dicho proyecto.

Figura A.3: Código en NetBeans

A continuación, en la pestaña "Services" hacemos click derecho sobre "UT2004 Servers" y seleccionamos "Add server" (Figura A.4). En el diálogo que aparece (Figura A.5) elegimos un nombre para el servidor y, en cuanto a la URI, tenemos dos opciones:

- Si el servidor está en el mismo ordenador desde el cual vamos a conectar el bot, escribimos localhost (si localhost no funciona probar con 127.0.0.1:3001 en su lugar).
- Escribir la IP del ordenador donde se encuentra el servidor.

Una vez hecho esto, hacemos click en Close y todo estará preparado para ejecutar el bot sobre el servidor. En caso de aparecer un triángulo amarillo con una exclamación sobre el servidor que hemos creado, revisar la configuración del mismo, ya que posiblemente no tengamos problemas para acceder a él desde el ordenador donde está alojado el servidor pero sí desde uno externo.

Por defecto, siempre aparecerá el puerto 3001 al poner la IP. Ésto se debe a que los puertos por defecto son:

- 3000 para BotConnection
- 3001 para ControlServer
- 3002 para SpectatorConnection

Ahora, el servidor esta en ejecución y la IDE sabe como conectarse a él. Para ejecutar el bot simplemente corremos el bot mediante el botón con el triángulo verde de "play". Si todo funciona, el bot se conectará al servidor y Netbeans nos mostrará la información referente a su ejecución. Si queremos inspeccionar el bot desde el juego, tenemos dos opciones.

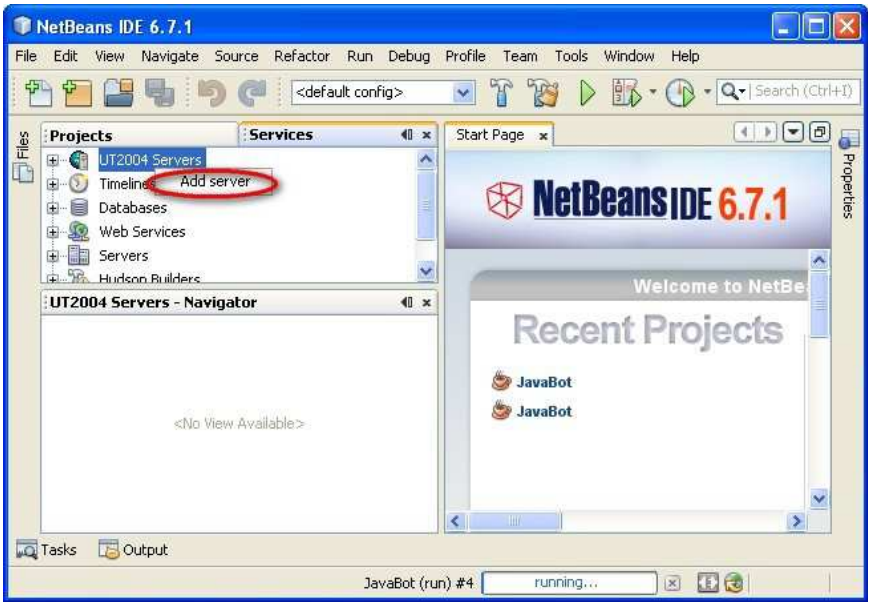

Figura A.4: Añadir servidor UT2004 a NetBeans

Figura A.5: Host del servidor UT2004 en NetBeans

- Abrir el juego y conectarnos al servidor como jugadore o espectador.
- Desde NetBeans, hacer click derecho sobre el servidor y seleccionar Spectate (Figura A.6).

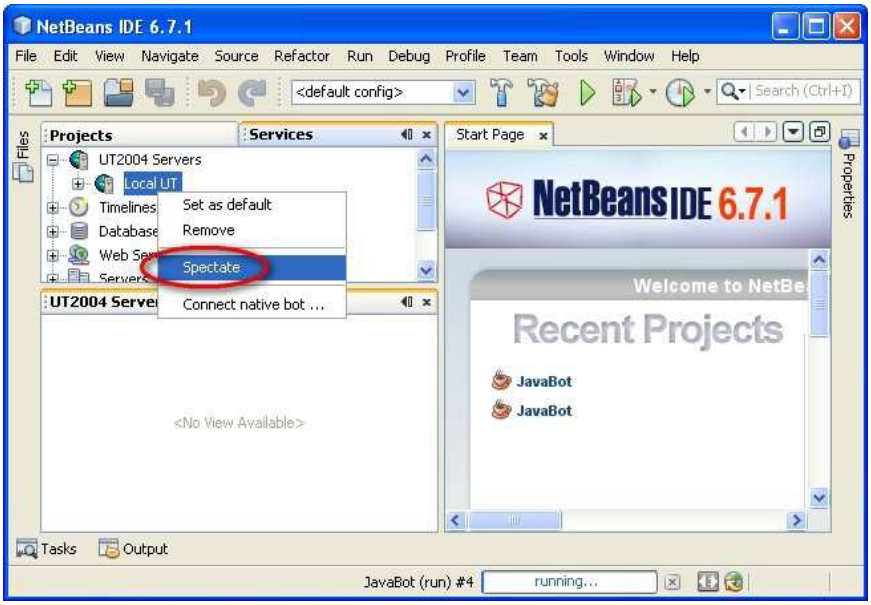

Figura A.6: Modo espectador en UT2004 desde NetBeans

Figura A.7: Consola del comandos con opciones propias de Pogamut 3 para UT2004

Por último, GAMEBOTS2004 incluye funcionalidades especiales, tales como la visualización de los navegadores y las líneas de movimiento, los rayos para detectar obstáculos, etc., los cuales pode-

mos activar y desactivar a nuestro antojo mediante un menú al que podemos acceder presionando ALT + H durante la ejecución del juego (Figura A.7).

A.2. Modos de movimiento del bot

Existen dos maneras prefijadas para tratar el movimiento de los bots.

1. **Navegación:** El bot se mueve eligiendo un punto de navegación en el mapa y calculando la ruta a seguir para alcanzar dicho punto de navegación (navpoint).
2. **Raycasting:** El bot se mueve basándose en la geometría del mundo, analizándola a través de rayos en busca de intersecciones.

A continuación se explica en detalle ambas implementaciones, las cuales podrían ser usadas en paralelo, creando un bot mas completo.

A.2.1. Bot de Navegación

El mapa está cubierto por nodos llamados puntos de navegación (navpoints). En teoría cada navpoint está situado en un sitio seguro y alcanzable por el bot. Los puntos conectados estan unidos por una línea. Para que el bot sea capaz de ir de un sitio A a un sitio B, vamos a necesitar una planificador de ruta: Path planner.

En Pogamut existen distintos interfaces de Path planner, siendo UT2004A StarPathPlanner el planificador por defecto. Este planificador usa el algoritmo A* para el cálculo de rutas, siendo exactamente el mismo que el de los bots nativos de UT. También existe otra implementación declarada en FloydWarshallPathPlanner, la cual precalcula todas las rutas posibles entre todos los nodos al principio, lo cual tiene un coste inicial considerable. Cabe decir que podemos implementar nuestro propio algoritmo planificador.

Una vez tenemos la ruta calculada ya solo nos queda ejecutarla, de ello se encarga el Path executor. Este módulo contiene el Path navigator que es el que se encargar de recorrer la ruta calculada anteriormente, evitar obstáculos, abrir puertas, esperar ascensores,etc. El navegador por defecto es UT2004PathNavigator. En este caso, también podemos crear nuestro propio navegador.

Veamos un ejemplo de implementación:

Algoritmo A.1 Ejemplo de bot de navegación

```
protected void goToRandomNavPoint() {
targetNavPoint = pickRandomNavPoint();
// find path to the random navpoint, path is computed asynchronously
// so the handle will hold the result onlt after some time
IPathFuture<ILocated> pathHandle = pathPlanner.computePath(info.getLocation(), targetNav-
Point);
// make the path executor follow the path, executor listens for the
// asynchronous result of path planning
pathExecutor.followPath(pathHandle);
}
```

En el ejemplo podemos ver como crea un punto de navegación al azar, y encontramos la ruta hasta dicho punto. Una vez la tenemos con followPath hacemos que el bot siga la ruta calculada.

Podemos declarar un listener dentro del Path Executor, para ser notificados acerca del estado del bot mientras ejecuta la ruta. Esto será útil para saber si el bot se ha atascado, si el bot ha llegado a su destino o incluso si la ejecución de la ruta ha fallado.

Para hacer un uso más simple y potente de la navegación a través de navpoints del bot, podemos usar la libreria Jungigation. Las principales ventajas de son:

- Exclusión de líneas prohibidas: Así evitaremos el cálculo de una mala ruta.
- Planificación avanzada basada en los objetos del mapa
- Saber exactamente el tiempo en segundos de la ruta.

El principal añadido de esta librería es que el planificador de ruta solo usa caminos que han sido probados y que se sabe con seguridad que el bot puede ir por ellos. Para ello hemos tenido que previamente usar una utilidad para calcular dichas rutas, por lo que esta funcionalidad no será interesante a no ser que implementemos un bot con aprendizaje.

A la hora de testear nuestro bot con navegación, podemos seguir a través de UT a nuestro bot, haciendo visibles en el mapa (ALT+G) los puntos y líneas de navegación. Otra forma es visualizar el mapa de navegación desde Netbeans.

A.2.2. Bot con raycasting

Como hemos dicho anteriormente el raycasting se base en rayos, a través de los cuales veremos sus intersecciones con el mundo/mapa. Para usar raycasting en nuestro bot, debemos seguir tres pasos:

1. **Activar raycasting y visualización de rayos:** podemos hacerlo de dos formas, o bien computar continuamente las intersecciones de los rayos con el mundo (Autotrace=true) o bien definir nuestros rayos y calcular solo las intersecciones que nos interesen (DrawTraceLine=true).
2. **Inicializar rayos:** definimos los rayos que deseemos a través de AddRay, pasándole como parámetros principales el nombre, el vector de dirección y la longitud del rayo. Podemos crear tantos rayos como queramos.
3. **Manejar resultados:** una vez que el rayo ha sido lanzado, UT nos devuelve el resultado en un mensaje. Podemos tratarlos de dos maneras: Creando un listener y que nos notifique cuando un rayo ha recibido algo del entorno o bien comprobando la presencia de algún objeto periódicamente, por ejemplo en el método logic().

Cuando ya hayamos sido notificados, definiremos las funciones de actuación que nos interesen y el objeto de AutoTrace ray será actualizado automáticamente.

A.3. Implementación del bot

Crear un bot Una vez que ya tenemos el servidor conectado a Netbeans es el momento de empezar a programar el bot. Para eso lo primero es entender que nos da Netbeans con su plantilla básica, y a partir de ahí desarrollar nuestro propio código.

A.3.1. Clases principales

En este apartado vamos a explicar las clases esenciales para crear un bot, desde la que se encarga de dar vida al bot a la que le permite moverse.

cz.cuni.amis.pogamut.ut2004.utils.SingleUT2004BotRunner

Esta clase es la encargada de lanzar un thread de ejecución para hacer correr al bot. Se sitúa dentro del main principal. Primero se establecen los parámetros con el servidor (host y port). Pueden ser de manera manual o que sea la propia clase que los encuentre. Después de establecer los parámetros se llama a startAgent() que es la encargada de lanzar el thread de ejecución. Se acaba cuando se elimina al bot de la partida.

cz.cuni.amis.utils.exception.PogamutException

En todas las funciones donde se ejecutan instrucciones de pogamut se añade esta clase para capturar excepciones. Únicamente se encarga de tratar cualquier excepción de pogamut que se pueda producir dentro de la función. En las funciones iniciales solo está definido en main() y en logic()

cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004BotModuleController

Esta clase contiene el controlador más avanzado del sistema. Tiene todos los módulos útiles para manejar el bot. Es la clase principal. Antes de pasar a explicar los módulos vamos a comentar las funciones iniciales del bot que se encargan de establecer la secuencia inicial de ejecución. Esta secuencia inicial de ejecución se corresponde a este protocolo de GameBots2004:

1. Llama a *this.prepareBot()*
2. → Recibe un mensaje HELLO_BOT, siendo el primer mensaje dentro de UT2004, el cual se encarga de pedir la conexión de un bot con el servidor. Si el servidor está lleno se termina la conexión.
3. ← Envía un mensaje READY. Como respuesta el servidor te envía al juego un mensaje NFO con información sobre la partida.
4. Ahora es cuando se establece la comunicación con GameBots2004.
5. → Captura el evento InitCommandRequested confirmando que la inicialización esta preparada.
6. Ejecuta this.getInitializeCommand().
7. ← Envía el comando INIT que se ha creado en la función anterior, con los parámetros iniciales del bot.
8. → Recibe el mensaje ConfigChange.
9. →Recibe el mensaje InitedMessage.
10. Llama a *this.botInitialized()*.
11. ... se recibe el primer mensaje SLF mostrando la información que el bot ya puede conocer de sí mismo, al haber sido introducido ya en el sistema. También recibe el mensaje END para terminar la comunicación síncrona con el servidor.

12. Ejecuta *this.botSpawned()*.

13. El bot pasa a correr dentro del entorno por medio de la función *logic()*.

Todos estos pasos se explican más en detalle ahora al conocer las funciones iniciales que se ejecutan en este protocolo de comunicación.

prepareBot()

Incluida en la clase *cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004BotController*.

Se encarga de establecer la configuración del mundo antes de saber nada sobre el UT2004. Simplemente prepara el sistema virtual. Se ejecuta antes de conectar al bot con el entorno pero después de construir el UT2004Bot.

getInitializeCommand()

Incluida en la clase *cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004BotController*.

Aquí es donde caracterizamos a nuestro bot con las opciones que deseemos. Envía un objeto *Initialize* que contiene todos los parámetros generales del bot.

Para poder tratar con este objeto hay que importar la clase *Initialize* que está dentro de la libería *cz.cuni.amis.pogamut.ut2004.communication.messages.gbcommands* al igual que el resto de comandos de pogamut que veremos más adelante.

La clase *Initialize* se corresponde al comando de GameBots2004 INIT.

botInitialized(GameInfo info, ConfigChange config, InitedMessage init)

Incluida en la clase *cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004BotController*.

Primera vez que el bot tiene información sobre el mundo, aunque todavía no ha sido creado.

Se llama a esta función una vez que el servidor ha enviado el mensaje INITED (clase de java: *InitedMessage*). Esto significa que el comando INIT (*Initialize*) ha sido ejecutado correctamente y la comunicación entre el bot y el servidor es correcta. En este momento ya es posible establecer nuevas comunicaciones entre el bot y el servidor por medio de comandos, aunque todavía el bot no ha sido creado en el entorno, así que los comandos de movimiento no se pueden ejecutar.

Contiene los primeros parámetros de configuración del bot dentro del entorno, información del juego en ese momento (*GameInfo*), configuración del sistema actual (*ConfigChange*) y mensajes de inicialización (*InitedMessage*).

Hay que importar las clases *GameInfo*, *ConfigChange* e *InitedMessage* dentro de la librería *cz.cuni.amis.pogamut.ut2004.communication.messages.gbinfomessages*, al igual que el resto de mensajes que genera pogamut.

El parámetro *GameInfo info* se corresponde con el mensaje NFO de GameBots2004, el cual se envía como respuesta al comando READY. Contiene información sobre el tipo de juego, el mapa, número de equipos...

El parámetro *ConfigChange config* se corresponde con el mensaje CONFCH de GameBots2004, el cual se envía cada vez que la configuración del bot cambia.

El parámetro *InitedMessage init* se corresponde con el mensaje INITED de GameBots2004, el cual se envía una vez que el comando INIT ha sido recibido correctamente. Contiene información de los atributos del bot, como su velocidad, máxima salud...

botSpawned(GameInfo info, ConfigChange config, InitedMessage init, Self self)

Incluida en la clase *cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004BotController*.

Se ejecuta cuando se crea el bot por primera vez, justo al entrar en el entorno del juego. Cuando el bot renace después de haber sido matado no ejecuta esta función.

Se puede llamar de manera automática activando la opción *ManualSpawn* en el comando INIT o de manera manual dentro de *botInitialized* con la función *respawn()*.

El parámetro *Self self* se corresponde con el mensaje SLF de GameBots2004, el cual envía información sobre el estado del bot, en este caso, nada más haber sido creado. La clase se importa desde *cz.cuni.amis.pogamut.ut2004.communication.messages.gbinfomessages*.

Logic()

Incluida en la clase cz.cuni.amis.pogamut.ut2004.bot.impl.

UT2004BotLogicController. Función principal del sistema. Entra justo después de crear el bot con *botSpawned.*. Solo se sale de ella cuando tiene lugar algún evento externo (se muere el bot) o cuando se elimina al bot de la partida. Todo lo que se ejecute dentro debe ser procesado rápidamente ya que se ejecuta iterativamente cada 0.25 segundos, por lo que no es recomendable hacer bucles en su interior.

botKilled(BotKilled event)

Incluida en la clase cz.cuni.amis.pogamut.ut2004.bot.impl.

UT2004BotController. Se activa cuando se produce un evento avisándote de que tu bot ha muerto. Se genera un mensaje asíncrono dentro del sistema que hace que salgas de la *logic()*. Este mensaje asíncrono se corresponde con el mensaje DIE de GameBots2004.

Entre otras cosas, puede avisar con un log de usuario que nuestro bot ha muerto, cambiar la apariencia del bot o cualquier otro cambio que queramos antes de que renazca y vuelva a la partida.

Para poder ejecutar la función es necesario importar la clase *BotKilled* desde *cz.cuni.amis.pogamut.ut2004.communication.messages.gbinfomessages*, para poder cargar la clase que contiene el mensaje.

A.3.2. Clase ModuleController

En esta sección vamos a centrarnos en todos los componentes contenidos en la clase *UT2004ModuleController*.

Doce de sus componentes son módulos completos y complejos que permiten recoger y enviar información sobre el bot y sobre su entorno. Estos son dichos módulos:

- Game
- Info
- Players
- Descriptors

- Items
- Senses
- Weaponry
- Config
- RayCasting
- Body
    - Shoot
    - Move

A parte de estos módulos existen otros componentes muy importantes para ayudar con el manejo del bot pero que no se les puede considerar módulos propiamente dichos.

- Random
- PathPlanner
- PathExecutor
- ListenerRegistrator
- Act
- World

A continuación vamos a explicar todos (los módulos y el resto de componentes), incluyendo la mayoría de acciones que se pueden realizar con ellos.

Game

Se encuentra dentro de esta librería: *cz.cuni.amis.pogamut.ut2004.agent.module.sensor*.

Módulo de memoria especializado en obtener información general sobre el juego.

Se puede usar a partir de *botInitialized()*, ya que es la primera vez que se tiene información sobre el mundo debido a los mensajes que ya han ido llegando por parte del servidor como *GameInfo* e *InitedMessage*. El módulo se abastece entre otras de la información proporcionada por estos mensajes.

Requiere la clase *bot* como parámetro.

Con este módulo podemos saber:

- Tipo de juego (DeathMatch, Capturar bandera, etc).
- Nombre del mapa en el que se jugará.
- Tiempo total y el tiempo que habrá para jugar antes de cambiar de escenario.
- Número máximo de equipos y puntuaciones de los equipos (o jugadores).
- Salud, armadura y adrenalina que dispondrá el bot al inicio y la máxima que puede llegar a conseguir.

- Estado de inicio del bot (parado o en movimiento)

A parte de tener información sobre el juego, dispone de una serie de listeners que permiten actualizar el módulo cada vez que uno de los mensajes que afectan al módulo llega. Estos son los eventos/mensajes que controla el módulo por medio de los listeners:

- GameInfo.
- InitedMessage.
- BeginMessage.
- MutatorListObtained.
- FlagInfo.

Info

Se encuentra dentro de esta librería: *cz.cuni.amis.pogamut.ut2004.agent.module.sensor*.

Es un módulo de memoria especializado en información general sobre el paradero del agente.

Se puede usar a partir de *botSpawned()* ya que es la primera vez que el bot existe y recibe información sobre su estado por medio del mensaje *Self*.

Aparte de la clase *bot*, también requiere como parámetro el módulo *Game*.

Con este módulo podemos saber todo esto sobre el bot:

- u nombre y su equipo.
- Que jugadores y equipos son enemigos y quienes son amigos.
- Localización:
  - Donde está.
  - Distancia a la que está de un lugar.
  - Comprobación de si está en una localización exacta.
  - Posición a la que se encuentra respecto al suelo para saber si está en un tejado
  - en algún lugar elevado.
- Rotación sobre el sistema de coordenadas en el que se encuentra, permitiendo la direccionalidad del bot.
- Velocidad actual.
- Posición en la que se encuentra:
  - En movimiento
  - Agachado
  - Andando
  - Frente un objeto ...
- Comprobar si posee habilidades especiales o bonos:

    - Multiplicador de daño.
    - Invisibilidad.
    - Otros mutators definidos al iniciar el servidor de la partida.
- Salud, armadura y adrenalina actual.
- Cuáles son sus armas en uso (primaria y secundaria).
- Cantidad de munición en cada arma en uso (primaria y secundaria).
- Comprobación de si está disparando o no y que arma está usando.
- Puntuaciones propias (puntos, muertes...).
- Características físicas del mapa y el entorno que le afectan:
    - Fricción del suelo.
    - Fricción de fluidos.
- Saber si esta dentro del agua o no.
- Saber si el lugar donde esta le produce daño (lava...) y la cantidad de daño producido por dicho elemento.
- Saber si está en un lugar que le produce la muerte (casi) instantánea.
- Tipo de daño que le está produciendo el lugar.
- Comprueba si el lugar donde está le prohíbe usar su inventario (armas u objetos)
- Comprueba si el lugar donde está afecta al disparo de proyectiles (viento...)
- Máxima velocidad alcanzable en el suelo, en el aire, en un escalón, en el agua, cayéndose o esquivando objetos.
- Ratio de aceleración. • Informa si se puede mover mientras esta en el aire.
- Informa sobre el punto de navegación más cercano.
- Informa sobre el punto de navegación visible más cercano.
- Informa sobre el objeto más cercano.
- Informa sobre el objeto visible más cercano.
- Informa sobre el jugador más cercano.
- Informa sobre el jugador visible más cercano.

A parte de todo esto, también dispone de una serie de listeners que se encargan de actualizar los datos del módulo.

Estos son los eventos/mensajes que controla el módulo por medio de los listeners:

- BotKilled (número de suicidios).
- PlayerKilled.
- InitedMessage.

- PlayerScore.
- TeamScore.
- VolumeChanged.
- Self.
- ConfigChange.

Players

Se encuentra dentro de esta librería: *cz.cuni.amis.pogamut.ut2004.agent.module.sensor*.

Es un módulo de memoria especializado en informar sobre otros jugadores. Todos los objetos de este tipo se auto-actualizan a lo largo del tiempo hasta que son destruidos.

Se puede usar a partir de *botInitialized()*, ya que es la primera vez que el bot tiene información sobre el mundo.

Requiere la clase bot como parámetro.

Este módulo permite al bot conocer todo esto sobre el resto de jugadores:

- Comprobar si puede alcanzar a otros enemigos, amigos o jugadores.
- Mapa con todos los enemigos, amigos o jugadores que son alcanzables.
- Comprobar si puede ver a otros enemigos, amigos o jugadores.
- Mapa con todos los enemigos, amigos o jugadores que son visibles.
- Mapa (id+player) de todos los enemigos o amigos.
- Informar sobre el enemigo, amigo o jugador más cercano.
- Informar sobre el enemigo, amigo o jugador visible más cercano.
- Últimos datos sobre un jugador.
- Datos sobre un jugador, enemigo o amigo elegido de manera aleatoria.
- Comprobar si un equipo o jugador es amigo o enemigo.

También dispone de una serie de listeners que se encargan de actualizar los datos del módulo. Estos son los eventos/mensajes que controla el módulo por medio de los listeners:

- Player.
- PlayerLeft.
- Self.

Para tener información sobre cómo han muerto los jugadores usar el módulo “senses”.

Descriptors

Se encuentra dentro de esta librería: *cz.cuni.amis.pogamut.ut2004.agent.module.sensor*.

Es un módulo sensorial que informa de las características generales de cualquier Item, en este caso son la categoría a la que pertenece (Weapon, Ammo, Health, Armor, Shield, Adrenaline, Other) y el grupo al que pertenece (Assault_Rifle, Minigun..., Health, Mini_Health..., Small_Armor, Super_Armor, Adrenaline, Udamage, Key...).

El conjunto de las categorías que puede tener un ítem esta dentro de un mapa hash formado por el nombre de la categoría y el tipo de ítem que es. Ocurre lo mismo con los grupos.

Suele combinarse con el módulo ítems por medio de mapas hash para tener una información completa sobre los ítems del escenario y su descriptor asociado.

Este módulo se puede usar a partir de *botInitialized()*, ya que es la primera vez que el bot tiene información sobre el mundo.

Requiere la clase bot como parámetro.

Items

Se encuentra dentro de esta librería: *cz.cuni.amis.pogamut.ut2004.agent.module.sensor*.

Es un módulo de memoria especializado en objetos que se encuentran en el mapa.

Se puede usar a partir de *botInitialized()*, ya que es la primera vez que el bot tiene información sobre el mundo.

Aparte de la clase bot, también requiere el módulo Info como parámetro.

Con este módulo se pueden obtener:

- Mapas con todos los items del escenario. También se puede restringir la búsqueda a los objetos visibles o a los alcanzables. Incluso se puede reducir más pidiendo solo los ítems que tengan un tipo, una categoría un grupo o un id determinado.
- Obtener un único ítem.
- Mapas con la lista de puntos donde salen los items. Se puede restringir pidiendo solo los ítems que tengan un tipo, una categoría un grupo o un id determinado.
- Mapas con la lista de puntos donde ahora mismo hay ítems disponibles. Se puede restringir pidiendo solo los ítems que tengan un tipo, una categoría un grupo o un id determinado.

También dispone de una serie de listeners que se encargan de actualizar los datos del módulo. Estos son los eventos/mensajes que controla el módulo por medio de los listeners:

- Item.
- MapPointsListObtained.
- NavPoint.
- EndMessage.
- ItemPickedUp.

Senses

Se encuentra dentro de esta librería: *cz.cuni.amis.pogamut.ut2004.agent.module.sensor*.

Es un módulo de memoria especializado en la parte sensorial del agente.

Se puede usar a partir de *botInitialized()*, ya que es la primera vez que el bot tiene información sobre el mundo.

Aparte de la clase bot requiere como parámetros a los módulos *Info* y *Players*.

Con este modulo podemos controlar todo esto sobre el bot:

- Saber cómo murió.
- Informar sobre el jugador que le ha golpeado.
- Informar sobre la localización donde fue golpeado o donde se chocó con un objeto del mapa.
- Último elemento recogido.
- Último daño que ha causado.
- Último daño que le han hecho.
- Último tiro que le impactó.
- Coordenadas donde se ha escuchado un ruido y el tipo de ruido que lo ha producido.
- Informar sobre la muerte de un jugador
- Comprobar si alguna de estas acciones ha sucedido (ahora o recientemente):
    - Ha muerto recientemente.
    - Ha obtenido adrenalina recientemente.
    - Le están hiriendo.
    - Está golpeando a otro jugador.
    - Está causando daño a alguien.
    - Está chocando con un elemento del mapa.
    - Está cayendo por un borde.
    - Está escuchando algún ruido.
    - Ha recogido algún elemento recientemente.
    - Algún jugador acaba de morir.
    - Está siendo dañado por algún jugador.
    - Está viendo algún proyectil.

Este módulo conecta un montón de listeners y proporciona muchos métodos para saber el actual estado del bot.

Estos son los eventos/mensajes que controla el módulo por medio de los listeners:

- Bumped.
- WallCollision.
- FallEdge.

- HearNoise.
- HearPickup.
- BotDamaged.
- IncomingProjectile.
- PlayerDamaged.
- PlayerKilled.
- AdrenalineGained.
- BotKilled.
- ItemPickedUp.
- BeginMessage.

Weaponry

Se encuentra dentro de esta librería: *cz.cuni.amis.pogamut.ut2004.agent.module.sensomotoric*.

Es un módulo de memoria especializado en el inventario del bot (armas y munición).

Se puede usar a partir de *botInitialized()*, ya que es la primera vez que el bot tiene información sobre el mundo.

Aparte de la clase bot, requiere como parámetro el módulo *Descriptors* con la información referente al tipo de arma.

El módulo le permite al bot saber:

- La cantidad de munición del arma actual (disparo primario y secundario) o de todas las armas del inventario.
- Características del arma que está usando.
- Armas cuerpo a cuerpo que dispone.
- Permite cambiar el arma por otra disponible en el inventario.
- Si alguna de estas condiciones se cumple:
  - Tiene munición en el arma actual
  - Tiene algún arma de cuerpo a cuerpo en el inventario.
  - Tiene algún arma de disparar en el inventario.
  - Tiene algún arma.
  - Tiene munición en el arma primaria o secundaria.
  - Tiene el arma en el inventario y está cargada.

Dispone de una serie de listeners que se encargan de actualizar los datos del módulo.

Estos son los eventos/mensajes que controla el módulo por medio de los listeners:

- AddInventoryMsg.

- ItemPickedUp.
- WeaponUpdate.
- SelfUpdate.
- Thrown.
- BotKilled.

Config

Se encuentra dentro de esta librería: *cz.cuni.amis.pogamut.ut2004.agent.module.sensomotoric*.

Es un módulo de memoria especializado en la configuración del agente dentro de UT2004.

Este módulo se puede usar a partir de *botInitialized()*, ya que es la primera vez que el bot tiene información sobre el mundo.

Se necesita la clase *bot*.

Este módulo permite realizar todo esto:

- Obtener y modificar la configuración general del bot dentro del juego (todos los campos cargados con *Initialize*).
- Consultar o modificar la velocidad de rotación del bot. Girará más o menos rápido según el valor que se elija.
- Cambiar el retraso entre envíos síncronos (de 0.1 a 2 segundos).
- Comprobar y modificar el uso de raytracing.
- Comprobar y modificar el uso invulnerabilidad.
- Comprobar y modificar la regeneración automática del bot cada vez que lo maten. Si se activa, cada vez que el bot muera renacerá instantáneamente, en el otro caso habrá que llamar a la función "respawn" en algún momento para que renazca el bot.
- Activar o no la focalización del bot en un objetivo siempre que lo haya detectado. Si se activa permite al bot desplazarse a una localización pero mirando hacia otro lado.
- Activar o no todos los mensajes síncronos. Si no se activa hay alguno de los mensajes síncronos del sistema que no se habilitan para una mayor fluidez del juego.
- Activar o no la recogida automática de ítems. Si no se activa, hay que ejecutar el comando *Pick* cada vez que se quiera recoger un ítem del escenario.

En este módulo solo hay un listener que controla un único evento/mensaje:

- ConfigChange.

RayCasting

Soporte para la creación de rayos que permiten detectar objetos en la escena. El rayo se pone verde cuando el camino está libre y rojo cuando hay obstáculos en la dirección y longitud del rayo. Lo que consiguen los rayos es que el bot circule solo por las zonas donde los rayos son verdes.

Requiere la clase *bot* como parámetro.

A la hora de crear un rayo hay que introducir estos campos que determinan su forma:

- Dirección del rayo. Indica en qué dirección apunta, tomando como punto (0,0,0) el bot.
- Longitud del rayo, a mayor longitud más lejos detecta objetos pero también es más fácil que se bloquee el bot al tener todos los rayos en rojo.
- Elegir entre los 2 tipos de rayos posibles:
  - Rápido pero escaso de información.
  - Lento pero más completo.
- Condición booleana para activar o no el vector normal con respecto al suelo. Si se activa hay que tener en cuenta para la dirección del rayo la normal con respecto al suelo.
- Condición booleana para tener en cuenta como obstáculos a los agentes dentro del juego o solo tener en cuenta la geometría.
- Después de inicializar todos los rayos hay que preparar los rayos para que escuchen las respuestas.

raycasting.getAllRaysInitialized().addListener(new FlagListener<Boolean>())

Con esta línea inicializa los rayos y activa el listener flag que se encarga de avisar cuando alguno de los rayos toca algo (se vuelve rojo).

Los rayos creados son de tipo "AutoTraceRay". Cada uno de estos rayos te permite saber:

- La localización exacta del objeto detectado.
- El tiempo global en el cual el objeto fue visto o actualizado por última vez.
- Si ha sido golpeado el rayo por algún objeto.
- El vector normal del plano donde el rayo ha sido golpeado.

Este módulo se puede usar a partir de *botInitialized()*, ya que es la primera vez que el bot tiene información sobre el mundo.

Body

Esta es la clase que contiene a este módulo: *cz.cuni.amis.pogamut.ut2004.bot.command. CompleteBotCommandsWrapper.*

Maneja los comandos que pueden ser usados como entrada de datos en el bot, todos ellos pertenecientes a *cz.cuni.amis.pogamut.ut2004.communication.messages.gbcommand.*

Requiere la clase bot como parámetro. Dentro de este módulo existen otros 6 módulos:

Action Proporciona los comandos de acción del bot. o Arrogar armas o Iteración con elementos del entorno (recoger ítems) o Otros comandos que no han podido ser clasificados en otras categorías (renacer).

Communication Proporciona los comandos de comunicación. o Enviar mensajes (privados, globales o de equipo). o Recibir mensajes (privados, globales o de equipo).

ConfigureCommands Permite cambiar los atributos del bot. Nombre, skin, velocidad de movimiento y de rotación, invulnerabilidad, etc.

AdvancedLocomotion Módulo "move", explicado a continuación.

AdvancedShooting Módulo "shoot", explicado a continuación.

SimpleRayCasting Control básico de rayos. Similar al módulo comentado anteriormente de "rayCasting". Su funcionalidad es la misma, el uso de rayos para detectar por donde puede moverse el bot y por donde no puede hacerlo.

Este módulo se puede usar a partir de *prepareBot()*, permitiendo usar alguna de sus funciones antes siquiera de tener el escenario cargado.

Shoot

Esta es la clase que contiene a este módulo: *cz.cuni.amis.pogamut.ut2004.bot.command. AdvancedShooting*.

Son acciones de disparo avanzadas que puede realizar el bot (no te dan información, son comandos que tú mandas al sistema para que el bot dispare). Estos son las acciones de disparo:

- Disparar arma actual con el modo seleccionado a un jugador o a un objeto. Si se gira entre 15^{o} o 30^{o} como mínimo dejara de disparar al objetivo. Seguirá al objeto hasta que llegue otra instrucción que le haga cambiar su foco de atención.
- Disparar arma actual con el modo seleccionado a una localización concreta. Si se gira entre 15^{o} o 30^{o} como mínimo dejara de disparar al objetivo.
- Disparar arma primaria a un jugador o a un objeto. Si se gira entre 15^{o} o 30^{o} como mínimo dejara de disparar al objetivo. Seguirá al objeto hasta que llegue otra instrucción que le haga cambiar su foco de atención.
- Disparar arma primaria a una localización concreta. Si se gira entre 15^{o} o 30^{o} como mínimo dejara de disparar al objetivo.
- Recargar arma primaria
- Disparar arma secundaria a un jugador o a un objeto. Si se gira entre 15^{o} o 30^{o} como mínimo dejara de disparar al objetivo. Seguirá al objeto hasta que llegue otra instrucción que le haga cambiar su foco de atención.
- Disparar arma secundaria a una localización concreta. Si se gira entre 15^{o} o 30^{o} como mínimo dejara de disparar al objetivo.
- Recargar arma secundaria
- Parar de disparar el arma actual.
- Parar de disparar completamente sin tener en cuenta el modo de disparo.

En este módulo solo hay un listener que controla un único evento/mensaje:

- BeginMessage.

Move

Esta es la clase que contiene a este módulo: *cz.cuni.amis.pogamut.ut2004.bot.command. AdvancedLocomotion*.

Son movimientos avanzados que puede realizar el bot (no te dan información, son comandos que tú mandas al sistema para que el bot se mueva). Estos son los movimientos:

- Esquivar disparos u objetivos, realizando un salto en la dirección indicada.
- Salto simple o Salto doble en la dirección que se encuentra.
- Ir de una localización (con vector de dirección incluido) a otra localización. Primero llega a la localización inicial. Evita tener que realizar dos *moveTo* ya que entre uno y otro tiene una parada.
- Correr de manera continua hace delante, hasta que le entre otro comando que le haga cambiar la velocidad.
- Moverse a una localización desde la que está ahora mismo .
- Cambiar la velocidad de giro, la velocidad de carrera y la velocidad de andar.
- Parar el bot en seco. Elimina cualquier movimiento o giro que estuviera haciendo.
- Girar la cara hacia un objeto o jugador. Poner el foco de atención en dicho objeto o jugador.
- Rotar horizontal o verticalmente el bot una cantidad de grados dados, para cambiar la vista del bot .
- Disparar a derecha/izquierda una determinada distancia.

Random

Módulo generador de números aleatorios especialmente útiles a la hora de llevar a cabo una toma de decisiones.

PathPlanner

Esta es la librería que contiene a este módulo: *cz.cuni.amis.pogamut.base.agent.navigation*.

Se encarga de calcular la ruta de acceso a un punto usando como apoyo los puntos de navegación del mapa.

A partir de dos puntos del mapa es responsable de encontrar la ruta genérica, incluso si no encuentra ninguna ruta te avisa de ello por medio de la excepción *cz.cuni.amis.pogamut.Client. navigation.PathNotConstructable*.

Hay varias formas de implementar esta ruta, como pueden ser el algoritmo de Dijkstra, A*...

El método A* ya esta implementado dentro de pogamut. Se encuentra en la misma librería que el *PathPlanner*. A* se dedica a buscar el camino más corto y a enviarlo a través de mensajes de GameBots. Debido a las restricciones que tiene el *PathPlanner* solo puede devolver paths con un tamaño máximo de 16 elementos.

Aun así no es necesario seguir este método genérico incluido dentro de pogamut. El programador puede incluir su propio planificador de rutas y que el bot siga su planificador en vez de seguir el planificador por defecto.

Otro aspecto interesante es que permite gestionar de manera asíncrona el path. De esta manera nada más empezar te da la ruta más rápida que ha podido encontrar pero conforme va avanzando por el path si encuentra otra ruta mejor, actualiza el path y sigue por esa nueva ruta. De esta manera es más eficiente ya que reacciona rápido a la hora de buscar un path pero sobre la marcha sigue intentado mejorar ese path.

Lo peor de seguir el *PathPlanner* de pogamut es que siempre sigue los puntos de navegación por lo que siempre va de punto a punto y no resulta muy realista, así que aunque es una función bastante útil, usarla tal como viene no ayuda a crear un bot humano.

Se inicializa con la función *initializePathFinding()* y si el path usado no es el genérico se debe sobrescribir este método para poder introducir el planificador propio.

Este módulo se puede usar a partir de *botSpawned()*, ya que necesita que el bot este cargado en el mapa para poder empezar a calcular los puntos de navegación.

Todo esto sobre el *pathPlanner* está más desarrollado en la parte del documento dedicada al movimiento.

PathExecutor

Esta es la librería que contiene a este módulo: *cz.cuni.amis.pogamut.base.agent.navigation.*

Se encarga de ejecutar el *pathPlanner* que haya sido calculado previamente.

Carga los listeners necesarios para detectar cuando ha llegado a uno de los puntos.

Cuando ha llegado a uno de los puntos del path vuelve a pedir el *pathPlanner* por si ha sido actualizado mientras estaba en ejecución. Con ese nuevo path vuelve a moverse hasta el siguiente punto y así hasta completar todo el *pathPlanner* calculado.

El *pathExecutor* es de tipo *UTPathExecutor*, encargado de enviar los mensajes de GameBots para que el bot actúe.

Es capaz de saber el Id del próximo objetivo, ya sea un objeto del mapa, una persona o un punto concreto del mapa. Normalmente sigue los puntos de navegación pero el punto final no tiene porque ser un punto de navegación, puedes apuntar a un player que se está moviendo y el *pathPlanner* se va actualizando siguiendo los puntos de navegación y una vez se pare el player, con el último mensaje del *pathExecutor* llegar al player aunque no esté situado en un punto de navegación.

Este módulo se puede usar a partir de *botSpawned()*, ya que necesita que el bot este cargado en el mapa para poder empezar a calcular los puntos de navegación.

Todo esto sobre el *pathExecutor* está más desarrollado en la parte del documento dedicada al movimiento.

ListenerRegistrator

Es el método encargado de autoinicializar los listeners por medio de *AnnotationListenerRegistrator*. Se encarga de proporcionar una manera sencilla y práctica de registrar los listeners, incluso encargándose de iterar entre todos los listeners registrados.

Hay 5 tipos de listeners diferentes:

- EventListener
- ObjectClassEventListener

- ObjectClassListener
- ObjectEventListener
- ObjectListener

Además de registrar los listeners puede:

- Eliminar los listeners que han sido creados.
- Saber si un listener ha sido creado.
- Saber la cantidad de listeners creados.

Todo este mundo de los eventos está mejor explicado en otra parte del documento.

Act

Forma parte de IAct, que es el entorno encargado de gestionar todas las acciones y comandos que se pueden realizar en este mundo. Ligado a world (IVisionWorldView) ya que los dos juntos forman todo el entorno del juego, Uno se encarga de las acciones (IAct) y el otro del entorno y los eventos que suceden en él (IVisionWorldView).

En resumen Act es el encargado de controlar las acciones que se ejecutan como consecuencia de unos listeners.

No se usa mucho ya que todo lo que desde Act se puede hacer, es lo mismo que te permiten hacer todos los módulos antes comentados.

Por ejemplo:

body.getCommunication().sendGlobalTextMessage("Hello world!");

es lo mismo que usar,

getAct().act(new SendMessage().setGlobal(true).setText("Hello world!");

En este caso dentro de *body.getCommunication()* ya están definidos todos los tipos de mensajes que se pueden enviar con funciones como *sendGlobalTextMessage* creadas, así que es mucho más intuitivo a la hora de programar, por este motivos usaremos siempre la primera opción y dejaremos el Act como recurso genérico.

World

Forma parte de IVisionWorldView, que es el entorno encargado de gestionar el entorno y todos los eventos que se producen en él. Ligado a act.

Este componente se relaciona con los eventos y los listeners que son en otra parte del documento.

A.3.3. Otros comandos interesantes

Anotación de java: @JProp

Anotación que se coloca delante de las variables java que quieras monitorizar en tiempo de ejecución. En tiempo de ejecución podemos acceder a estas variables, para modificarlas y ver su comportamiento en el juego, desde "Servidores → UT2004 servers → Name Server → Bot pogamut → Introspection → Properties.

GBCOMMANDS

Esta librería *cz.cuni.amis.pogamut.ut2004.communication.messages.gbCommands* contiene todos comandos que se pueden introducir en GameBots. Al usar java con Netbeans no es necesario introducir los comandos de manera literal. Java nos permite tener una clase para cada uno de los comandos que se pueden manejar. Con esto cada vez que se use en una clase un comando

acciones que se pueden realizar en GameBots pero java nos da una interfaz más compleja con la que poder utilizarlos sin tener que preocuparnos de tener que llamar a cada uno de los comandos de manera literal.

Ahora vamos a poner la lista de todas las clases de comandos, la mayoría se entienden por si mismas así que no es necesario explicarlas, aun así dentro del javadoc de pogamut hay un extensa explicación de los campos y las funciones que componen cada una de las clases (una por comando). Además en el javadoc se puede ver con que comando de GameBots está relacionada cada clase.

- Act
- AddBot
- AddInventory
- AddRay
- Combo
- CommandPlayer
- Configuration
- ConfigurationObserver
- Console
- ContinuousMove
- DialogBegin
- DialogCancel
- DialogEnd
- DialogItem
- DisconnectObserver
- Dodge
- DriveTo

- EndPlayers
- EnterVehicle
- FactoryUse
- FastTrace
- GetAllInvetories
- GetAllNavPoints
- GetAllStatus
- GetGameInfo
- GetItemCategory
- GetMaps
- GetPath
- GetPlayers
- GetSelf
- GetSpecialObjects
- GetVisibleObjects
- GiveInventory
- ChangeAttribute
- ChangeMap
- ChangeTeam ChangeWeapon
- CheckReachability
- Initialize
- InitializeObserver
- Jump
- Kick
- LeaveVehicle
- Move
- PasswordReply Pause
- Pick Ping
- PlaySound
- Quit
- Ready
- Record

- RemoveRay
- Respawn
- Rotate
- SendMessage
- SetCrouch
- SetDialog
- SetGameSpeed SetLock SetPassword
- SetPlayerControl
- SetRoute
- SetSendKeys
- SetSkin
- SetWalk
- Shoot
- ShowText
- SpawnActor StartAnimation
- StartPlayers
- Stop
- StopRecord
- StopShooting
- Throw
- Trace
- TurnTo

A.4. Eventos

A.4.1. Interacción con el mundo

Las interfaces *IworldView* junto con *Iact* representan la API básica para acceder al mundo.

Información que recibimos (Listeners)

La interfaz *IWorldView* (cz.cuni.amis.pogamut.base.communication.worldview) nos ofrece tanto los sentidos del bot y como memoria simple. El mundo es representado por:

- Objetos (IworldObject): cz.cuni.amis.pogamut.base.communication.worldview.object AliveMessage AutoTraceRay BombInfo ConfigChange DominationPoint FlagInfo GameInfo IncomingProjectile InitedMessage Item ItemCategory Mover MyInventory NavPoint Player Self TeamScore Vehicle.
- Eventos (IWorldEvent): cz.cuni.amis.pogamut.base. communication.worldview.event
  - Dos Categorías:
    - Eventos de Objetos (IWorldObjectEvent): cz.cuni.amis.pogamut.base. communication.worldview.object
      - WorldObjectAppearedEvent
      - WorldObjectDestroyedEvent
      - WorldObjectDisappearedEvent
      - WorldObjectFirstEncounteredEvent
      - WorldObjectUpdatedEvent
      - WorldObjectEvent (cualquiera de los eventos por parte del objeto)
    - Eventos no asociados a ningún objeto (utilizados directamente de IWorldEvent). La lista de eventos es la siguiente. Observar que esta lista cuenta también con los eventos pertenecientes a objetos, ya que podríamos querer, por ejemplo, ser conscientes de todos los objetos que apareciesen en pantalla, para lo que usaríamos WorldObjectAppearedEvent.
      - AddInventoryMsg
      - AdrenalineGained
      - AnimationBotID
      - AnimationEnd
      - AnimationPort
      - AnimationStop
      - BeginMessage
      - BotDamaged
      - BotFirstSpawned
      - BotKilled
      - Bumped
      - ComboStarted
      - DialogCommand
      - DialogFailed
      - DialogOk
      - EndMessage
      - EnteredVehicle
      - FactoryUsed
      - FallEdge
      - FastTraceResponse
      - GamePaused
      - GameResumed

- GBEvent
- GlobalChat
- HandShakeEnd
- HandShakeStart
- HearNoise
- HearPickup
- HelloBotHandshake
- HelloControlServerHandshake
- HelloObserverHandshake
- ChangedWeapon
- InitCommandRequest
- ItemCategoryEnd
- ItemCategoryStart
- ItemDescriptorObtained
- ItemListEnd
- ItemListStart
- ItemPickedUp
- JumpPerformed
- KeyEvent
- Landed
- ListObtained
- LockedVehicle
- LostInventory
- MapFinished
- MapChange
- MapList
- MapListEnd
- MapListObtained
- MapListStart
- MapPointListObtained
- MoverListEnd
- MoverListObtained
- MoverListStart
- Mutator
- MutatorListEnd
- MutatorListObtained
- MutatorListStart
- MyInventoryEnd
- MyInventoryStart
- NavPointListEnd
- NavPointListStart
- NavPointNeighbourLink
- NavPointNeighbourLinkEnd
- NavPointNeighbourLinkStart
- ObjectSelected

- ◇ PasswdOk
- ◇ PasswdWrong
- ◇ Password
- ◇ Path
- ◇ PathList
- ◇ PathListEnd
- ◇ PathListStart
- ◇ PlayerDamaged
- ◇ PlayerInput
- ◇ PlayerJoinsGame
- ◇ PlayerKilled
- ◇ PlayerLeft
- ◇ PlayerListEnd
- ◇ PlayerListObtained
- ◇ PlayerListStart
- ◇ PlayerScore
- ◇ Pong
- ◇ ReadyCommandRequest
- ◇ Reachable
- ◇ RecordingEnded
- ◇ RecordingStarted
- ◇ ShootingStarted
- ◇ ShootingStopped
- ◇ Spawn
- ◇ TeamChanged
- ◇ TeamChat
- ◇ Thrown
- ◇ TraceResponse
- ◇ Trigger
- ◇ VolumeChanged
- ◇ WallCollision
- ◇ WeaponUpdate
- ◇ WorldEventIdentityWrapper
- ◇ WorldObjectAppearedEvent
- ◇ WorldObjectDestroyedEvent
- ◇ WorldObjectDisappearedEvent
- ◇ WorldObjectEvent
- ◇ WorldObjectFirstEncounteredEvent
- ◇ WorldObjectUpdatedEvent
- ◇ ZoneChangedBot

En el siguiente apartado se muestra una información más detallada de los eventos pertenecientes a IWorldEvent.

Cuando queramos utilizar uno de estos eventos u objetos debemos incluir su correspondiente librería. Para ello consultar en la API la librería concreta de cada uno.

Una vez explicado qué información podemos obtener del mundo y de qué manera, vamos a ver cómo manipularla, es decir, cuál es la forma más sencilla de recibir dicha información.

La clase UT2004BotModuleController autoinicializa AnnotationListenerRegistrator. Ésto quiere decir que no será necesario manipular los listeners manualmente, es decir, tener que añadirlos, elminarlos, etc. además de reprogramarlos, saber cómo iterar para seleccionar el adecuado en cada momento, etc. En vez de eso, AnnotationListenerRegistrator permite registrar los listeners de forma muy sencilla y práctica, y él mismo se encargará de iterar entre todos los listeners registrados. Una vez registrado el listener, definimos una función que realice las acciones deseadas en cada caso.

Hay cinco diferentes tipos de listeners que podemos registrar. Los dos últimos son difíciles de usar, puesto que es difícil obtener el strig correspondiente al id específico del objeto, pero es necesario saber de su existencia, aunque todavía desconocemos cómo utilizarlos y si serán necesarios:

- EventListener: reacciona al evento que nosotros definamos por medio del campo eventClass. Por ejemplo:
  - Reg: *@EventListener(eventClass = Bumped.class)*, reacciona a eventos de la clase Bumped.
  - Proc: *protected void bumped(Bumped event)* {
- ObjectClassListener: reacciona a todos los eventos ocurridos a una clase de objeto concreta, definida por medio del campo objectClass. Por ejemplo:
  - Reg: *@ObjectClassListener (objectClass = Player.class)*, reacciona a eventos ocurridos a la clase Player.
  - Proc: *protected void player(Player object)* {
- ObjectClassEventListener: reacciona a eventos de una clase concreta (de entre los eventos asociados a objetos) que ocurren a objetos de una clase concreta.
  - Ej: *@ObjectClassEventListener(eventClass = WorldObjectAppearedEvent.class, objectClass = Player.class)*, reacciona cuando un jugador cualquiera "aparece" en nuestro campo de visión.
  - Proc: *protected void playerAppeared(WorldObjectAppearedEvent<Player> event)* {
- ObjectListener: similar a *ObjectClassListener* para un objeto concreto, es decir, un objeto con un id determinado dentro una clase concreta.
- ObjectEventListener: similar a *ObjectClassEventListener* para un objeto concreto, es decir, un evento de una clase concreta (de entre los eventos asociados a objetos) que ocurre a un objeto con un id determinado dentro una clase concreta.

Información que enviamos (Actions)

Como hemos dicho, nuestro bot será de la clase *UT2004BotModuleControler*, contenida en *cz.cuni.amis.pogamut.ut2004.bot.impl*. Para dar órdenes al mismo no usaremos directamente la interfaz *IAct*, es decir, pese a que somos libres de utilizar el método act (perteneciente como hemos dicho a la clase *UT2004BotModuleControler*), las acciones las enviaremos, de manera más sencilla e intuitiva, mediante el método body.

El método body está dividido en los siguiente métodos:

- Action getAction()

- Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.Action* command module.

- Communication getCommunication()

  - Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.Communication* command module.

- ConfigureCommands getConfigureCommands()

  - Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.ConfigureCommands* command module.

- AdvancedLocomotion getLocomotion()

  - Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.AdvancedLocomotion* command module.

- AdvancedShooting getShooting()

  - Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.AdvancedShooting* command module.

- SimpleRayCasting getSimpleRayCasting()

  - Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.SimpleRayCasting* command module.

La clase UT2004BotModuleControler ofrece también los métodos shoot para acceder directamente a body.getShooting() y move para body.getLocomotion().

Para acceder a los mismos, debemos incluir las librerías correspondientes a cada clase contenida en Action, Comunication, ConfigureCommands, AdvancedLocomotion, AdvancedShooting y SimpleRayCasting. Para ello consultar la API.

A.4.2. Descripción de los eventos

En este apartado se trata de explicar de manera resumida los eventos más importantes que tendremos que utilizar durante la implementación de nuestro bot. Debemos tener en cuenta las siguientes consideraciones:

- Muchos de los eventos son utilizados para la conexión con el servidor y cosas por el estilo, lo cual ya está implementado y no necesitamos para implementar nuestro bot.

- Algunos de los eventos indican el principio y el final de la llegada de un lote síncrono de datos. (no sé si es necesario o está ya implementado en la clase que controle cada lote)

- Nuestra implementación está orientada al tipo de partida DeathMatch sin chat, por lo que no tendremos en cuenta eventos referentes a equipos, banderas, chat, etc.

Los más importantes

AddInventoryMessage: Enviado cuando conseguimos una nueva arma o munición para un arma que todavía no tenemos (notifica nuevo objeto en nuestro inventario, no cuando lo recogemos).

AdrenalineGained: obtenemos adrenalina, recogiéndola o matando a alguien.

BotDamaged: nuestro bot ha sido dañado.

BotKilled: nuestro bot ha muerto.

Bumped: nuestro bot es tocado por otro actor.

ComboStarted: el jugador observado inicia un combo (el combo empieza cuando la adrenalina llega a 100 y se va gastando al usarlo).

FallEdge: enviado cuando estamos al borde de un precipicio. Si el bot estaba corriendo, el mensaje llega cuando el bot está ya cayendo. Si el bot estaba andando, cuando el mensaje llega estamos al borde del precipicio (no podemos caer cuando andamos).

HearNoise: se dispara cuando oímos un sonido, lo cual puede ser un jugador caminando o disparando, una bala impactando contra el suelo, un ascensor subiendo y bajando, etc. Podemos conocer dónde se produce el ruido y por qué es producido.

HearPickup: oyes a otro jugador recogiendo un objeto del suelo.

ChangedWeapon: enviado cuando el bot cambia el arma (el cambio de arma será muy probablemente debido a un comando introducido por nosotros mismos). Podemos saber qué arma usábamos antes del cambio y qué arma usamos actualmente.

ItemPickedUp: cuando recogemos un item (no tiene por qué añadirse a nuestro inventario, puede ser adrenalina o vida).

JumpPerformed: Enviado cuando el jugador observado salta.

KeyEvent: enviado cuando presionamos una tecla. Imagino que será útil si queremos modificar nuestro bot en tiempo de ejecución.

Landed: el bot toma tierra después de una caída.

PlayerDamaged: nuestro bot hiere a otro jugador.

PlayerJoinsGame: un jugador se une al juego.

PlayerKilled: otro jugador ha muerto, y podemos obtener información acerca de su muerte.

PlayerLeft: un jugador abandona la partida.

ShootingStarted: el jugador observado comienza a disparar.

ShootingStopped: el jugador observado deja de disparar.

Spawn: recibido cada vez que el bot resucita.

VolumeChanged: alguna parte del cuerpo del bot ha cambiado de zona (al agua, lava, etc.).

WallColission: el bot ha colisionado contra un muro.

ZoneChangebot: el bot cambia de zona, es decir, el cuerpo completo del bot está en una sola zona después de que no era así.

Respuesta a comandos (pueden no ser necesarios)

Posiblemente el comando que los utilice ya incluye su tratamiento.

EnteredVehicle / LockedVehicle: enviado como respuesta al comando ENTER, si hemos conseguido entrar al vehículo (y por tanto comenzamos a conducirlo) o no (está ocupado o no podemos cogerlo), respectivamente

FactoryUsed: respuesta al comando USE(FactoryUse).

FastTraceResponse: respuesta al comando FASTTRACE.

LostInventory: hemos perdido un item del inventario (arrojamos un arma, etc.).

Reachable: booleano como contestación a un comando checkreach, es decir, nos dice si se puede acceder directamente a alcanzar un objeto (simplemente caminando).

Thrown: respuesta al comando THROWN, es decir, cuando tiramos un arma.

TraceResponse: respuesta al comando TRACE.

Trigger: respuesta cuando activamos algún disparador. Podemos saber qué y si realmente lo hemos hecho nosotros.

WeaponUpdated: el bot cambia de arma. Aquí exportamos el estado del arma anterior, para tener información correcta de las armas de nuestro inventario, lo cual sería un problema por el retraso del paquete síncrono. (posiblemente ya tratado en la memoria de inventario)

Especiales (curiosos al menos)

ObjectSelected: referentes a un comportamiento alternativo del bot, el cual no es útil para un bot propiamente dicho pero podría llegar a ser útil según en qué consista nuestro trabajo de investigación. Esta función se activa pulsando ALT + LEFT SHIFT. Con esto, podemos apuntar un objeto de los visibles en pantalla con el cursor del ratón. Una vez hecho esto podemos identificar objetos, para lo cual sirve este evento.

PlayerInput: permite al jugador cuando apunta a nuestro bot con el cursor, darle órdenes de forma numérica (0-9).

Descartados

Los referentes a animation, dialog, handshake, inventory, map, mutators, navPoint, password, path, etc.

A.4.3. Eventos clasificados por grupos

Lotes síncronos

Comienzo y finalización de la transmisión de lotes síncronos

- HandShakeStart / HandShakeEnd
- ItemCategoryStart / ItemCategoryEnd
- ItemListStart / ItemListEnd
- MutatorListStart / MutatorListEnd

- MyInventoryStart / MyInventoryEnd
- PathListStart / PathListEnd
- PlayerListStart / PlayerListEnd

Eventos de objetos

Eventos especiales referentes a objetos. (ya explicados)

- WorldObjectEvent (superclase de los siguientes, es decir, cualquier evento de objeto)
- WorldObjectAppearedEvent
- WorldObjectDestroyedEvent
- WorldObjectDisappearedEvent
- WorldObjectFirstEncounteredEvent
- WorldObjectUpdatedEvent

Items / Inventary

- AdrenalineGained
- AddInventoryMsg
- ItemDescriptorObtained
- ItemPickedUp
- LostInventory
- WeaponUpdate
- Reachable

Puntos de Navegación

- NavPointNeighbourLink
- NavPointListStart / NavPointListEnd
- NavPointNeighbourLinkStart / NavPointNeighbourLinkEnd

Mapa

- MapFinished
- MapChange
- MapList
- MapListEnd
- MapListObtained
- MapListStart
- MapPointListObtained

Player

- PlayerDamaged
- PlayerInput
- PlayerJoinsGame
- PlayerKilled
- PlayerLeft
- PlayerListObtained
- PlayerScore
- ComboStarted

Mutator

- Mutator
- MutatorListObtained

Sonidos

- HearNoise
- HearPickup
- VolumeChanged

Mover

- MoverListEnd
- MoverListObtained
- MoverListStart

Dialog

- DialogCommand
- DialogFailed
- DialogOk

Vehículos

- EnteredVehicle
- LockedVehicle

Disparar

- ShootingStarted
- ShootingStopped

Bot

- BotDamaged
- BotFirstSpawned
- BotKilled
- Bumped
- ChangedWeapon
- FallEdge
- JumpPerformed
- Landed
- Spawn
- Thrown
- WallCollision
- ZoneChangedBot

Grabar

Respuesta a los comandos REC y STOPREC respectivamente.

- RecordingStarted
- RecordingEnded

Estado del juego

Se pone y se quita la pausa del juego.

- GameResumed
- GamePaused

Path

En la clase Path se encuentra la función getPath(). Una vez enviada esta petición, se nos devuelve un lote PathList cuyo comienzo y final está delimitado por PathListStart y PathListEnd respectivamente.

- Path
- PathList
- PathListStart / PathListEnd

Otros

- ObjectSelected
- BeginMessage
- EndMessage
- FactoryUsed
- FastTraceResponse
- GBEvent
- InitCommandRequest
- KeyEvent
- ListObtained
- ReadyCommandRequest
- TraceResponse
- Trigger
- WorldEventIdentityWrapper

Apéndice B

Botprize

A continuación se muestran las reglas de la competición y los resultados finales obtenidos en el concurso de 2010, que ganaron los españoles Raúl Arrabales y Jorge Muñoz, profesores en la Universidad Carlos III de Madrid, ganaron el concurso.

B.1. Reglas de competición

B.1.1. The task

The competition task is to create a computer game bot which is indistinguishable from a human player. Those entries that pass this test will share the major prize of A$7,000 cash, and will also be offered a trip to 2K Australia's studio in Canberra. If the major prize is not won, a minor prize of A$2,000 plus a trip to the studio will be awarded. A member of the winning team (either major or minor) will be invited to visit 2K Australia's studio, at 2K Australia's expense, up to an amount of A$5,000, in addition to the cash prize.

The competition will be run in Seoul, South Korea at the 2010 IEEE Conference on Computational Intelligence and Games.

To participate in the finals, one team member must register for and attend the conference.

The game used for the competition will be based on a modified version of the DeathMatch game type for the First-Person Shooter, Unreal Tournament 2004. This modified version provides a socket-based interface (called Gamebots) that allows control of bots from an external program. In addition, several extra modifications will be made especially for the competition:

- Chatting will be disabled (this is not a chatbot competition!)
- Some aspects of the game play will be modified to facilitate the competition.

B.1.2. To enter

Competitors must advise their intention to enter the competition on or before the end of July 2011, by email to the competition organizers. Some of the final rounds will take place in August, leading up to the conference, and the remainder at the conference itself. If there are a large number of entries, the organizers reserve the right to carry out qualification trials to select the entries to take part in the finals. The programs for the trials need not be final versions.

Other conditions of entry are:

- Individuals or teams may enter.
- No-one can enter more than one bot (either as an individual or as part of a team).
- No-one associated with 2K or with the organization of the competition may enter.
- Entrants must affirm that they have intellectual rights to their entry and that it and its components comply with all artistic licenses.
- Entrants younger than 18 years of age must provide a written statement of permission by at least one parent or guardian.
- Entrants must be willing to allow videos/mpegs of their entries in action at the competition to be published and become public domain.

B.1.3. Testing protocol

The precise details of judging for 2010 have yet to be finalized. Note that the in-game judging system is different from that used in 2010, and at the recent human-like bot competition.

Judging will be done using an in-game judging system. We have set up a server so that teams can test their bots in the same environment and using the same system that will be used for the final judging. Point your UT2004 client or your bots at the BotPrize server (195.242.237.18).

The system is based on a modification to the Link Gun. It is now used by players to "tag" other players as either human or bot. The primary mode applies the BOT tag, which the secondary mode applies the HUMAN tag. Once an opponent is tagged, the player will see the word BOT or HUMAN as a label on the opponent's avatar. Players can change their judgment at any time by using the Link Gun again. Judgments have no effect on player health, or on game scores.

B.2. Resultados botprize 2010

Resultados de humanidad La Tabla B.1 muestra los resultados obtenidos por los bot, en cuanto a humanidad se refiere, o sea, el porcentaje de jueces que han determinado que el bot era humano.

Bot name	**Team**	**Affiliation**	**Humanness**
Conscious-Robots	Raúl Arrabales y Jorge Muñoz	Carlos III University, Madrid	31.8182 %
UTˆ2	Igor Karpov, Jacob Schrum y Risto Miikulainen	University of Texas, Austin	27.2727 %
ICE-2010	Akihiro Kojima, Daichi Hirono, Takumi Sato, Seiji Murakami y Ruck Thawonmas	Intelligent Computer Entertainment Lab. Ritsumeikan University, Japan	23.3333 %
Discordia	Casey Rosenthal y Clare Bates Congdon	University of Southern Maine	17.7778 %
w00t	Daniel Büscher, Matthias Gorzellik, Jannis Seyfried y Björn Witt	Institut für Informatik Albert-Ludwigs Universität Freiburg, Deutschland	9.3023 %

Cuadro B.1: Humanidad de los bots.

Como se puede ver en los datos, el equipo ganador fue el que se presentó por la Universidad Carlos III de Madrid, pero no pudieron superar el Test de Turing, ni consiguieron superar el nivel de humanidad de los bot UT2004 que está en torno al 35 %.

La Tabla B.2 muestra los mismos resultados anteriores pero en este caso para los jugadores humanos.

Player	**Affiliation**	**Humanness**
Mads Frost	IT University Copenhagen	80 %
Simon and Will Lucas	University of Essex	59.0909 %
Ben Weber	UC Santa Cruz	48.2759 %
Nicola Beume	TU Dortmund University	47.0588 %
Minh Tran	Edith Cowan University	42.3077 %
Gordon Calleja	IT University Copenhagen	38.0952 %
Mike Preuss	TU Dortmund University	35.4839 %

Cuadro B.2: Humanidad de los jugadores humanos.

Según estos resultados incluso el peor de los humanos supera al mejor de los bot, aunque por un estrecho margen que muestra los importantes avances de los bots en cuanto a humanidad se refiere.

B.3. Resultados de los jueces

En la Tabla B.3 se muestran los mejores bots a la hora de juzgar a sus compañeros.

Bot name	**Accuracy**
Discordia	54.8387 %
w00t	53.8462 %
UTˆ2	45.7447 %

Cuadro B.3: Precisión de los bot al juzgar.

En la Tabla B.4 se muestran los mejores jueces.

Player	**Accuracy**
Gordon Calleja	78.5714 %
Nicola Beume	67.2131 %
Minh Tran	64.2857 %
Ben Weber	64.0845 %
Mike Preuss	59.7015 %
Mads Frost	57.6923 %
Simon and Will Lucas	54.7945 %

Cuadro B.4: Precisión de los jueces al juzgar.

Apéndice C

Resolución analítica del modelo propuesto

Una vez definido el problema, procedemos a calcular los valores que nos proporcionan un sistema que ofrezca un máximo valor de $\overline{p}(T)$. Si tenemos,

$$\begin{cases} \dot{a}(t) = \frac{1}{\tau}(1 - a(t)) - \gamma(t) \cdot (\frac{1}{\tau} + a(t) \cdot (\frac{1}{\varepsilon} - \frac{1}{\tau}) \\ \dot{p}(t) = \gamma(t) \cdot a(t) \end{cases}$$

donde $\gamma(t) = \{0, 1\}$ y queremos encontrar el conjunto $\{\gamma_k(t_k)\}$ que maximice $p(t)$. Discretizando,

$$\begin{cases} a_{k+1} - a_k = -h(\frac{1}{\tau}(1 - a_k) - \gamma_k \cdot (\frac{1}{\tau} + a_k \cdot (\frac{1}{\varepsilon} - \frac{1}{\tau})) \\ p_{k+1} - p_k = h(\gamma_k a_k) \end{cases}$$

donde h es un paso temporal, $k = 0, 1, 2, ..., N$, de forma que $a(0) = a_0$, $p(T) = p_N$, puesto que $T = \{t_1, t_2, ..., t_N\}$. En la versión discretizada el problema se puede reformular así (sabiendo que h es constante):

"Encontrar el conjunto de decisiones $\{\gamma_k(t_k)\}$ tal que maximice $\sum_{k=0}^{N} \gamma_k a_k$

Esto es, debemos calcular las $\{\gamma_k(t_k)\}$ tal que

$$p_{_N} = \underset{\gamma_{_0}, \gamma_{_1}, ... \gamma_{_N}}{max} \textstyle\sum_{k=0}^{N} \gamma_k a_k,$$

que como se inicia en a_0, denotamos $p_{_N}^{MAX}(a_0)$.

Para resolverlo aplicamos la Regla de Bellman (*An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.* [Oviedo, 2005]). El criterio de Bellman calcula la secuencia completa $(\gamma_{_0}, \gamma_{_1}, ... \gamma_{_N})$ de forma recursiva y hacia atrás, por tanto,

$$p_{N}^{MAX}(a_0) = \max_{\gamma_0}[\gamma_0 a_0 + \max_{\gamma_1,\cdots\gamma_N} \sum_{k=1}^{N} \gamma_k a_k], \text{ donde}$$

$$p_{N-1}^{MAX}(a_1) = \max_{\gamma_1,\cdots\gamma_N} \sum_{k=1}^{N} \gamma_k a_k$$

iterando de nuevo se obtiene la secuencia:

$$p_{N}^{MAX}(a_0) = \max_{\gamma_0}[\gamma_0 a_0 + \max_{\gamma_1}[\gamma_1 a_1 + \max_{\gamma_2}[\gamma_2 a_2 + ... + ... + \max_{\gamma_N}[\gamma_N a_N]]....]$$

Para resolver el sistema, empezamos a resolverlo desde la última decisión a la primera. Como la úlitma no afecta al futuro, la maximizción es local. En nuestro caso,consiste en:

$$\gamma_N(t_N) = \begin{cases} 1, & \text{si} \quad a_N \in (a_M,\, 0), \;\; \dot{a}_N < 0 \\ 0, & \text{si} \quad a_N \in (0, a_M), \;\; \dot{a}_N > 0 \end{cases}$$

por tanto,

$$p_{0}^{MAX}(a_N) = \begin{cases} a_N, & \text{si} \quad a_N \in (a_M,\, 0), \;\; \dot{a}_N < 0 \\ 0, & \text{si} \quad a_N \in (0, a_M), \;\; \dot{a}_N > 0 \end{cases}$$

esto es, según el sistema se encuentre en fase de generación de soluciones o de ejecución. Una vez que sabemos cuál es la decisión óptima en $\gamma_N(t_N)$ calculamos para el instante anterior $\gamma_{N-1}(t_{N-1})$ aplicando la ecuación:

$$p_{1}^{MAX}(a_{N-1}) = \max_{\gamma_{N-1}}[\gamma_{N-1} a_{N-1} + p_{0}^{MAX}(a_N)]$$

sabemos que

$$a_N = a_{N-1} - \tfrac{h}{\tau}((1 - a_{N-1}) - \gamma_{N-1} \cdot (1 + a_{N-1} \cdot (\tfrac{\tau}{\varepsilon} - 1)), \text{ por tanto}$$

$$p_{1}^{MAX}(a_{N-1}) = \max_{\gamma_{N-1}}[\gamma_{N-1} a_{N-1} + p_{0}^{MAX}(a_{N-1} - \tfrac{h}{\tau}((1 - a_{N-1}) - \gamma_{N-1} \cdot (1 + a_{N-1} \cdot (\tfrac{\tau}{\varepsilon} - 1))))]$$

Dado que $\gamma_{N-1} = \{0, 1\}$, sólo tendremos que calcular cuál de los dos casos es mayor:

$$a_{_{N-1}} + p_{0}^{MAX}[a_{N-1} - \tfrac{h}{\tau}((1 - a_{N-1}) - (1 + a_{N-1} \cdot (\tfrac{\tau}{\varepsilon} - 1)))] \gtrless p_{0}^{MAX}[a_{N-1} - \tfrac{h}{\tau}((1 - a_{N-1})]$$

que, simplificando, queda como:

$$a_{_{N-1}} + p_{0}^{MAX}[(1 - \tfrac{h}{\varepsilon}) \cdot a_{N-1}] \gtrless p_{0}^{MAX}[(1 - \tfrac{h}{\tau}) \cdot a_{N-1} + \tfrac{h}{\tau}]$$

La condición de equilibrio se cumple para un valor de a_{N-1}crítico, que denotamos, a^*_{N-1},con el que se puede reescribir la ecuación del siguiente modo:

$$p_{1}^{MAX}(a_{N-1}) = \begin{cases} a_{_{N-1}} + p_{0}^{MAX}[(1 - \tfrac{h}{\varepsilon}) \cdot a_{N-1}], & \text{si } a_{N-1} > a^*_{N-1} \\ p_{0}^{MAX}[(1 - \tfrac{h}{\tau}) \cdot a_{N-1} + \tfrac{h}{\tau}], & \text{si } a_{N-1} \leqslant a^*_{N-1} \end{cases}$$

El procedimiento se puede repetir para $(k = 2, ..., N)$ obteniéndose los valores $\{a^*_0, a^*_1, ..., a^*_{N-1}, a^*_N\}$, que se obtienen resolviendo iterativamente la ecuación:

$$p_{N-k}^{MAX}(a_{N-k}) = \max_{\gamma_{N-k}}[\gamma_{_{N-k}} a_{_{N-k}} + p_{k-1}^{MAX}(a_{N-k} - \tfrac{h}{\tau}((1 - a_{N-k}) - \gamma_{N-k} \cdot (1 + a_{N-k} \cdot (\tfrac{\tau}{\varepsilon} - 1)))),$$

donde $k = 1, ..., N$.

Apéndice D

Modelo Infotaxis

Infotaxis is fully described in Vergassola et al. (2007). For completion, we detail its core modules in terms of probabilistic robotics (Thrun et al., 2005) as employed in our robot implementation. The model combines a belief function – the robot internal knowledge about his environment, updated as cues are encountered – along with decision-making – execution of an action that maximizes a reward. In Infotaxis, the robot is provided with a statistical description of the odor plume that he uses to infer the probability that the source be located at any point of his internal grid-based probability map of the environment. The statistical description of the odor plume is derived from the resolution of the following advection-diffusion equation

$$D\nabla^2 U(\mathbf{r}) + \vec{V} \cdot \nabla U(\mathbf{r}) - \frac{1}{\tau} U(\mathbf{r}) - R\delta(\mathbf{r} - \mathbf{r}_0) = 0$$

for an odor source located at r_0 and emitting 'particles' or patches' at a rate R. The particles propagate with diffusivity D, have a mean lifetime described by τ and are advected by a mean current or wind V. $U(r)$ is the local concentration at location r and δ is the Dirac delta function. In such an environment, the mean frequency of odor encounters with a spherical sensor of radius 'a' follows the Smoluchowski's (1917) expression

$$R(\mathbf{r}) = 4\pi D a \cdot U(\mathbf{r})$$

This model provides a framework by which to take into consideration the geometry of the environment when navigating. It can be easily solved through numerical methods and makes it possible for autonomous robots to iteratively infer knowledge about their surrounding. In the continuous case, the solution to Eq. 1 writes:

$$U(\mathbf{r}|\mathbf{r}_0) = \frac{1}{4\pi D\,|\mathbf{r} - \mathbf{r}_0|} e^{-\frac{V}{2D}(y - y_0)} e^{-\frac{|\mathbf{r} - \mathbf{r}_0|}{\lambda}} \quad \text{where} \quad \lambda = \sqrt{\frac{D\tau}{\left(1 + \frac{V^2\tau}{4D}\right)}}$$

For the non-continuous case, i.e., under the influence of a pulsed odor source $R(t)$ at location r_0, we derived such function by solving the non-homogeneous diffusion-advection equation which, for the two-dimensional problem with mean vertical wind V_y, takes the form:

$$U(\mathbf{r}, t|\mathbf{r}_0) = e^{-\frac{V}{2D}(y - y_0) - \frac{1}{\tau}\left(1 + \frac{V^2\tau}{4D}\right)t} \left[\frac{e^{-\frac{|\mathbf{r} - \mathbf{r}_0|}{4Dt}}}{4\pi D t} * R(t) e^{\frac{1}{\tau}\left(1 + \frac{V^2\tau}{4D}\right)t} \right]$$

Belief function

Let us consider the trace $\Gamma_t = \{(r_1, t_1), (r_2, t_2), \dots (r_n, t_n)\}$ of the hits (odor encounters) experienced by the searcher at locations $x_1 \dots x_n$and times $t_1 <, \dots, < t_n < t$ during the path from its start to the current time t. The belief function is given by the posterior probability for the source to be located in r_0 given the trace Γ_t

$$P_t(\mathbf{r}_0|\Gamma_t) = \frac{\mathcal{L}(\Gamma_t|\mathbf{r}_0)}{\int \mathcal{L}(\Gamma_t|\mathbf{r}_x) d\mathbf{r}_x}$$

where $\mathcal{L}(\Gamma_t|r_0)$ is the likelihood of experimenting the trace Γ_t for a source in r_0. In the robot implementation of infotaxis, consecutive detections are not considered as they belong to a same patch and are correlated (see Section "Materials and Methods"). We therefore employ the following likelihood function (from Eq. 13 in supplementary materials of Vergassola et al., 2007) instead of the one considered in the original algorithm

$$\mathcal{L}(\Gamma_t|r_0) = e^{-\sum_i \int_{V_i} R(r(t')|r_0) dt'} \prod_{i=1}^{T} R(r(t_i), t_i|r_0)$$

in which T represents the transitions from no-detection to detection, i.e., new patches, and the $V_{i's}$ are the time intervals of absence of detection. Note from Eq. 6 that the absence of correlations permits to update the probability map without storing the whole history. Indeed, $P_{t+\triangle t}(r_0) = P_t(r_0)$ update $\triangle t$ (nparticles), where 'nparticles' is the number of detections that the searcher experienced during the short time interval $\triangle t$. Memory requirements are therefore kept to a minimum.

Decision-making

For the decision-making, the robot moves in the direction that minimizes its local uncertainty about the location of the source. The expected reduction of entropy – reward function – for the robot moving from r_i to r_j, consists of two terms:

$$\Delta H(\mathbf{r}_i \rightarrow \mathbf{r}_j) = P_t(\mathbf{r}_j)(0 - H)$$

$$+ \left[1 - P_t(\mathbf{r}_j)\right] \Delta S$$

The first term (Eq. 7) evaluates the reduction of entropy if the source is found at the next step. Reaching the source in r_j occurs with estimated probability P_t and the entropy goes from H to 0. The second term (Eq. 8) corresponds to the reduction of entropy if the source is not found. It occurs with probability $1 - P_t$ and $\triangle S$ represents the information gain in r_j coming from expected odor encounters. The first term is seen as exploitative as it drives the searcher toward locations where the probability of finding the source is high. The second term is explorative as it compels the searcher toward regions with lower probabilities of source discovery but high information gains. The expected reduction of entropy in the case where the source is not found derives from the probability sum of experiencing i new detections during the movement, where encounters are modeled by means of a Poisson-distributed random variable ρ_i

$$\Delta S = \rho_0 \Delta S_0 + \rho_1 \Delta S_1 + \rho_2 \Delta S_2 \dots$$

therefore accounting for all possible cases that new information is detected along the way (either 1, 2 or n encounters).

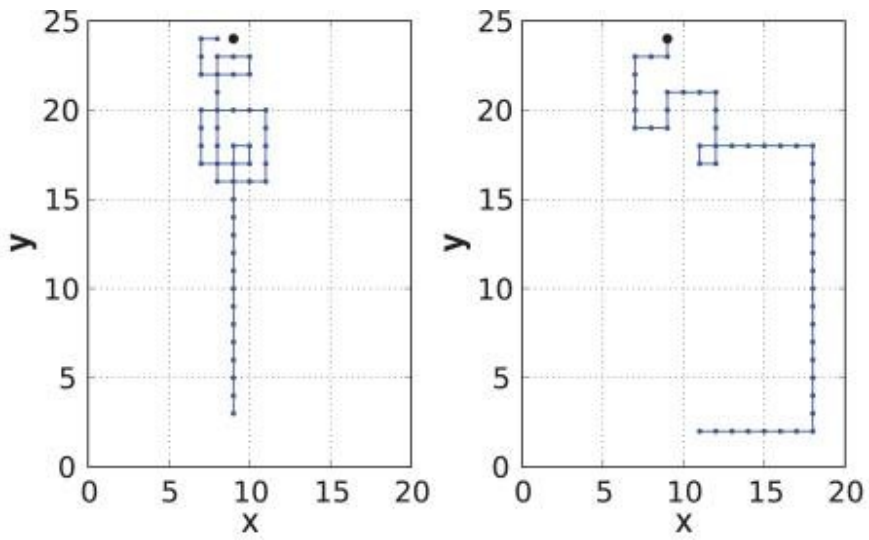

Figura D.1: Robot Navigation Patterns. Left: Examples of robot trajectories. The source is continuous and located at (9,24). The robot starting point is (10,2).

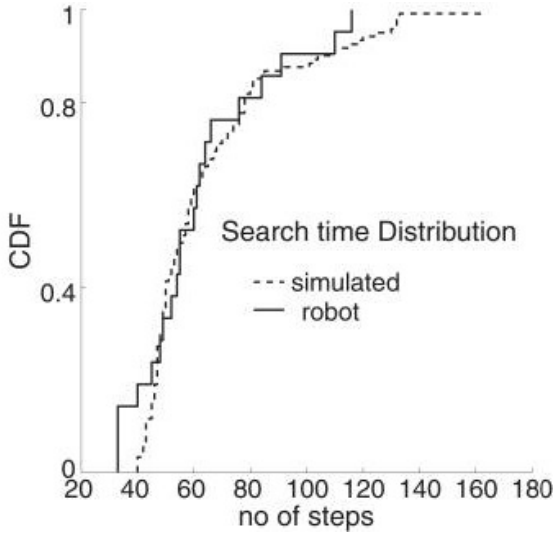

Figura D.2: Robot vs simulated infotaxis. Cumulative distributions of the number of steps (search time) for robot infotaxis (plain curve) and for simulated infotaxis (dashed curve).