Simulador GSim

Gustavo Armagno
Facundo Benavides
Claudia Rostagnol

1 de noviembre de 2005

Facultad de Ingeniería
Universidad de la República Oriental del Uruguay

Resumen

El objetivo de este documento es realizar un análisis del simulador GSim, desarrollado por el equipo FRUTo como alternativa al simulador proporcionado por la FIRA. El resultado de este análisis brindará las pautas acerca de qué mejoras pueden efectuarse.

Índice

1. Introducción **4**

2. Descripción **4**
2.1. Arquitectura 4
2.1.1. Bajo nivel 7
2.2. Colaboración entre los componentes 12
2.2.1. Simulador GSim 12
2.2.2. Emulación del simulador de la FIRA 15
2.3. Modelo físico 16
2.3.1. Movimiento de los cuerpos 16
2.3.2. Detección y resolución de colisiones 18

3. Análisis **22**
3.1. Criterios de análisis 22
3.2. Flexibilidad de interacción 23
3.3. Velocidad de simulación 23
3.4. Estado inicial e intervalo de ejecución 24
3.5. Aproximación al modelo de la FIRA 25
3.6. Interfaz gráfica y registro del juego 29
3.7. API para automatizar aprendizaje 30
3.8. Extensibilidad 30

4. Conclusiones **31**

A. Interfaz del simulador de la FIRA **33**

1. Introducción

El objetivo de este documento es realizar un análisis del simulador GSim, desarrollado por el equipo FRUTo como alternativa al simulador proporcionado por la FIRA. El resultado de este análisis brindará las pautas acerca de qué mejoras pueden efectuarse.

La información contenida en este documento está basada principalmente en el manual de usuario del simulador GSim y en la documentación de la API. En algunos casos fue necesario consultar el código, a partir del cual se realizó una ingeniería inversa, con la finalidad de exclarecer algunos conceptos.

Descontando la introducción y la sección de referencias, este documento está organizado en tres secciones:

- **Descripción del simulador GSim**. En esta sección (Sección 2) se introducen las motivaciones que llevaron a su construcción; se realiza una descripción gradual del diseño del simulador, comenzando por la arquitectura de la solución hasta llegar a la interacción entre los distintos componentes; y finalmente, se hace referencia al modelo físico subyacente.
- **Análisis del simulador GSim**. El análisis que se realiza en esta sección (Sección 3) se divide en varias subsecciones, cada una de las cuales está asociada a un criterio de análisis determinado (ver 3.1). La primera parte de cada subsección describe aquellas características que sería deseable que tuviera cualquier simulador de fútbol de robots. En función de esta descripción se analizan las fortalezas y limitaciones del simulador GSim.
- **Conclusiones**. A partir del análisis del simulador, se resumen las fortalezas del simulador GSim y se delinean las mejoras que admite el mismo.

2. Descripción

El GSim es un simulador 2D para fútbol de robots desarrollado por el equipo FRUTo. GSim es el acrónimo de Geometric Simulator, y su nombre radica en que el simulador utiliza un enfoque geométrico para detectar y resolver colisiones entre los cuerpos, sin perjuicio de que el movimiento de los cuerpos sí se basa en un modelo físico.

Según [Cas03] la motivación original era contar, en el marco del fútbol de robots, con un entorno que permitiera trabajar fácilmente con técnicas de aprendizaje automático (programación genética, redes neuronales, etc.) con el objetivo de poder realizar experimentos sin la intervención humana, y aplicar esta experiencia en el simulador oficial de la *Middle League SimuroSot* de la FIRA. Más concretamente, la necesidad fundamental que motivó a la construcción de este simulador era la de contar con una API sobre el simulador de la FIRA que permitiera de manera programática (sin la necesidad de que intervenga un operador) crear una instancia del simulador, establecer el estado inicial (posición de jugadores y pelota) y a partir de esto ejecutar una simulación durante un cierto intervalo de tiempo. Lamentablemente, el simulador proporcionado por la FIRA no cuenta con dicha API.

2.1. Arquitectura

Como muestra la figura 1, el simulador GSim está dividido en cinco paquetes:

gsim: Núcleo central del simulador. Controla la interacción entre los distintos componentes (equipo izquierdo, equipo derecho e interfaz gráfica).

engine: Encapsula el modelo del mundo que rige la simulación.

fira: Contiene los elementos que utilizan los distintos componentes para interactuar entre sí. La interfaz es la misma que la proporcionada por el *Middle League SimuroSot* de la FIRA.

gui: Encapsula la interfaz gráfica del usuario. Permite monitorear visualmente lo que está ocurriendo en la simulación.

team: Contiene algunas estrategias simples.

util: Helper classes.

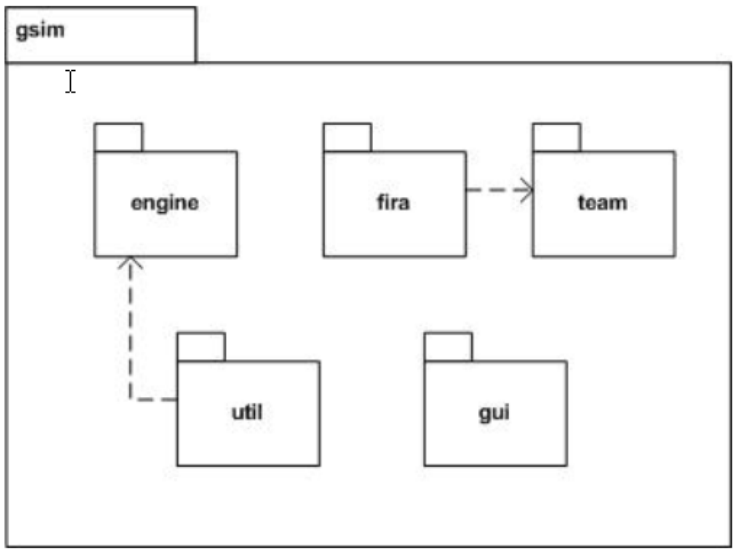

Figura 1: Paquetes del simulador GSim y sus dependencias.

Los paquetes gui y team no son necesarios para el funcionamiento de GSim, sino que fueron incluidos con la finalidad de probar el sistema.

El GSim fue diseñado para que los componentes involucrados en la simulación trabajen en forma concurrente. Estos componentes son cuatro: la física -o *Engine*-, los equipos izquierdo y derecho -o *LeftTeam* y *RightTeam*, respectivamente- y la salida, la cual a su vez puede estar compuesta por varios *listeners*; entre ellos, la interfaz gráfica -o *GUI*-. Un quinto componente -denominado *GSim*- regula la interacción entre estos (ver figura 2), interconectándolos y sincronizando el flujo de información.

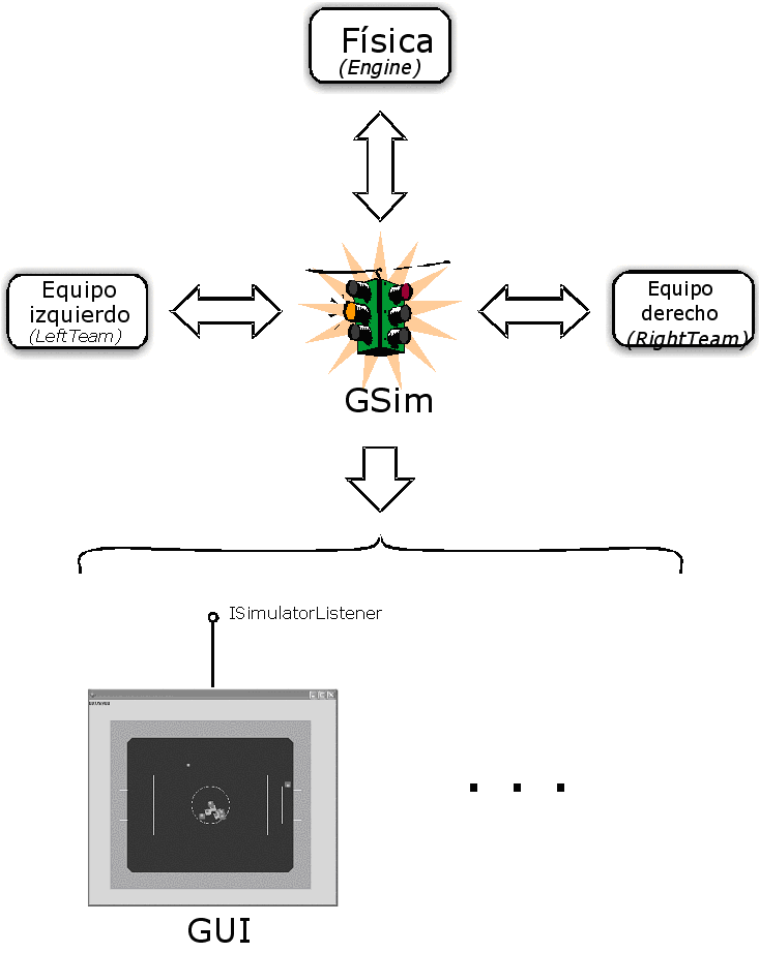

Figura 2: Interacción de los distintos componentes en el ciclo de simulación de GSim.

La interacción entre el componente *GSim* y los equipos izquierdo y derecho se realiza a través de la interfaz que utiliza el simulador de la FIRA (esta interfaz puede encontrarse en el Apéndice A). Esto permite portar las estrategias desde el GSim al simulador de la FIRA, y viceversa, de forma totalmente transparente. La concurrencia en la ejecución de las estrategias no sólo aprovecha mejor los tiempos de cálculo, sino que también permite explorar otras variantes de simulación, distintas al mecanismo proporcionado por el simulador de la FIRA. Una posibilidad sería, por ejemplo, que los equipos izquierdo y derecho se ejecuten en forma asincrónica, acercando la simulación a la forma en que actúan los robots en el mundo real.

Si bien el simulador GSim proporciona una *GUI* por defecto, la arquitectura permite que el usuario programador la reemplace, o incluso que implemente otros componentes que conviertan el estado del juego en una salida que pueda ser interpretada por un operador, o que pueda sea utilizada como insumo de otras herramientas. Los componentes que implementan la salida son denominados *listeners* del simulador. Estos *listeners* también se ejecutan de forma concurrente, lo cual permite mejorar la performance de la solución.

2.1.1. Bajo nivel

Visto en mayor profundidad, los componentes *Engine*, *LeftTeam*, *RightTeam* y los *listeners* utilizan colas de lectura bloqueante y escritura no bloqueante para intercambiar información con el componente *GSim*. En la figura 3, una flecha que entra en la cola representa una operación de escritura -operación *put*- mientras que una flecha saliente representa una operación de lectura -operación *get*-.

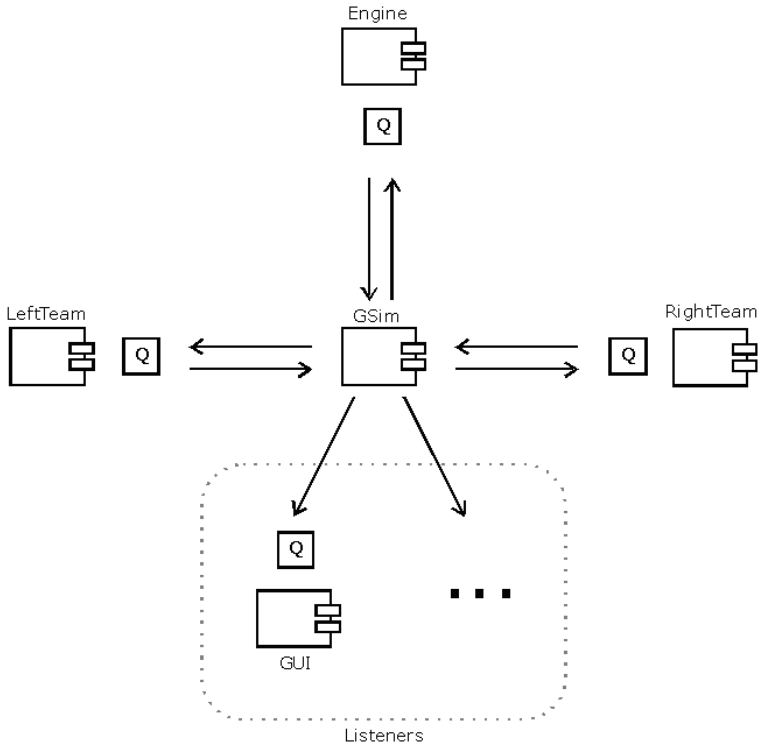

Figura 3: Intercambio de información entre los componentes del simulador utilizando colas de lectura bloqueante y escritura no bloqueante. Si bien la figura describe este intercambio en un alto nivel de abstracción, es importante destacar que en la implementación actual del simulador se utiliza una cola para cada operación de lectura/escritura. Por ejemplo, en la implementación actual, tanto los equipos izquierdo y derecho, como la *engine* física se comunican con el componente central a través de dos colas (una de lectura y otra de escritura), mientras que los *listeners* utilizan una única cola (de escritura).

Todas las colas poseen capacidad solamente para un elemento. Esto significa que si un elemento de la cola no fue consumido a tiempo, será sobrescrito sin ningún tipo de consideración. En el caso de la comunicación (unidireccional) entre el componente *GSim* y los *listeners*, podría ocurrir que algunos datos de salida se pierdan si el paso de simulación es más pequeño que el tiempo de procesamiento de los *listeners*.

Diagrama de clases

La figura 4 muestra un diagrama de clases simplificado del simulador GSim. A continuación se describen las distintas clases presentadas:

GSim: Es la clase encargada de efectuar la simulación, sincronizando los distintos hilos que participan en el proceso. La operación *init* recibe la configuración inicial de la simulación, *run* ejecuta la simulación propiamente dicha y *destroy* libera los recursos reservados por *init*. Como se verá más adelante (ver 2.2), la simulación efectuada por esta clase no espera a que cada estrategia termine de ejecutarse, a diferencia de la simulación de la FIRA. Esta clase puede ser extendida, permitiendo al programador modificar el mecanismo de simulación.

GSimEngine: Implementa el modelo físico (*engine* física).

GSimGUI: Se encarga de manejar la interfaz gráfica del simulador.

IStrategy: Homologa la interfaz que existe entre el simulador de la FIRA y las estrategias. Las estrategias de los equipos izquierdo y derecho deben extender esta interfaz. La operación *create* recibe el estado inicial del juego antes de comenzar la simulación, *strategy* es invocada periódicamente para calcular el nuevo estado de los robots y *destroy* permite liberar recursos eventualmente reservados por create.

ISimulatorListener: Interfaz que implementan los *listeners* que generan la salida del simulador como *GSimGUI*. Como muestra la figura 4, la clase *GSimEngine* también extiende de *ISimulatorListener*. Si bien esto podría hacer pensar que *GSimEngine* es un *listener* más, no hay que perder de vista que *GSimEngine* no sólo recibe el estado del juego sino que también lo modifica, mientras que los *listeners* que generan la salida únicamente reciben el estado del juego. La operación *tick* recibe y eventualmente modifica el estado actual del juego.

GSimThread: Es la clase base para los hilos que serán creados por el simulador. Entre ellos se encuentran los *listeners* generadores de salida (*GSimGUI*, por ejemplo), el modelo físico (*GSimEngine*) y las estrategias (*IStrategy*).

GSimThreadListener1: Esta clase está asociada con los *listeners* que generan la salida. Sus instancias son hilos que, en cada ejecución, consumen el estado del juego de la cola de entrada e invocan la operación *tick* del *listener* asociado, encargada de generar la salida. Es importante recalcar que si bien esta clase es responsable de consumir datos de la cola, no escribe datos en ella.

GSimThreadListener2: Es análoga a *GSimThreadListener1*, pero a diferencia de ésta, sí escribe en la cola el estado del juego. El estado es (potencialmente) modificado por la instancia de *ISimulatorListener* asociada. En este caso, la instancia asociada no es un *listener* generador de salida, sino que es la clase responsable de implementar el modelo físico.

GSimThreadTeam: Las instancias de esta clase son hilos asociados a las estrategias del juego. Encapsula la lógica de sincronización de la comunicación entre el simulador y la estrategia.

SyncQueue1: Representa una cola bloqueante con capacidad para un único elemento. Posee una operación *get* bloqueante y una operación *put* no bloqueante.

GSimConfig: Contiene los parámetros que serán pasados al simulador GSim antes de que comience la simulación.

GSimFIRA: Deriva de la clase *GSim* para sobrescribir la operación *run*, y así poder emular el ciclo de simulación de la FIRA.

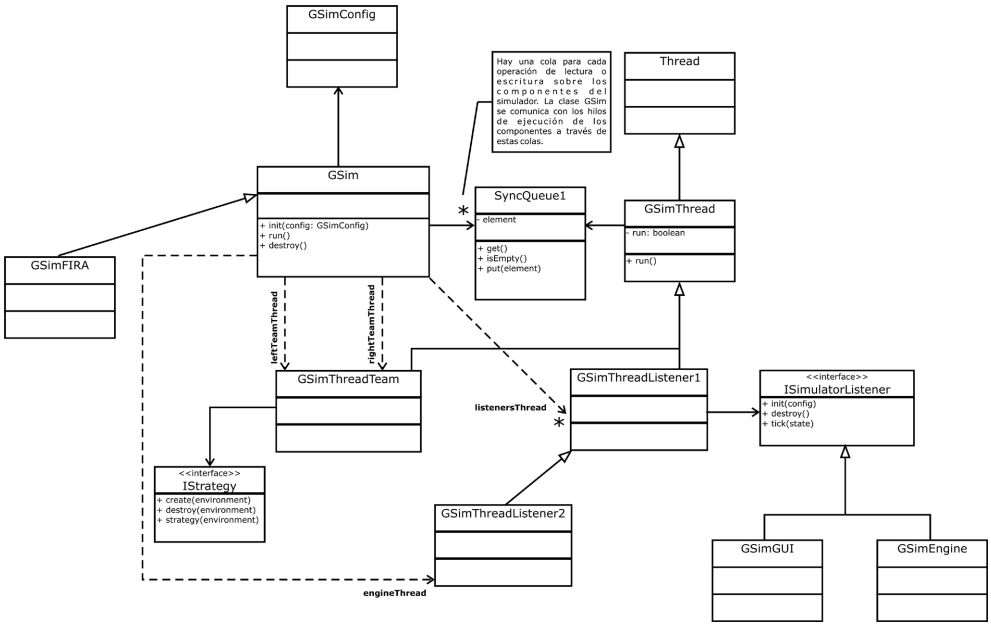

Figura 4: Diagrama de clases simplificado del GSim (extraído de [Cas03]).

Detalles

Como muestran con mayor detalle las figuras 5, 6 y 7, el componente *GSim* se comunica con los distintos componentes que participan de la simulación indirectamente, a través de las colas. Las verdaderas encargadas de interactuar directamente con estos componentes son las instancias de las clases derivadas de *GSimThread* (*GSimThreadTeam*, *GSimThreadListener1* y *GSimThreadListener2*).

Se definen dos colas, una de entrada y otra de salida, para cada componente que reciba datos de entrada y modifique estos datos o genere otros para retornárselos al simulador. Estos componentes son las estrategias de los equipos izquierdo y derecho, y la engine física. Para los listeners, que sólo reciben datos de entrada [1], se define una cola de entrada para cada uno.

La clase *GSimConfig* actúa como *Factory* de las estrategias de cada equipo y de la interfaz gráfica [2]. La engine física es creada directamente por la clase GSim.

En virtud de que la interfaz *IStrategy* homologa la interfaz utilizada por el simulador de la FIRA, las estrategias probadas en el simulador GSim pueden ser migradas de uno a otro con facilidad.

[1] Si bien los *listeners* generan una salida, ésta no se retorna al simulador, sino que se despliega en pantalla o se almacena en algún otro medio.

[2] Es importante recalcar que la clase *GSimConfig* sólo crea un único *listener*, el componente *GUI*. Sin embargo, la arquitectura permite que esta clase sea extendida (mejorada) para ser pasada como parámetro a la operación *init* de *GSim*.

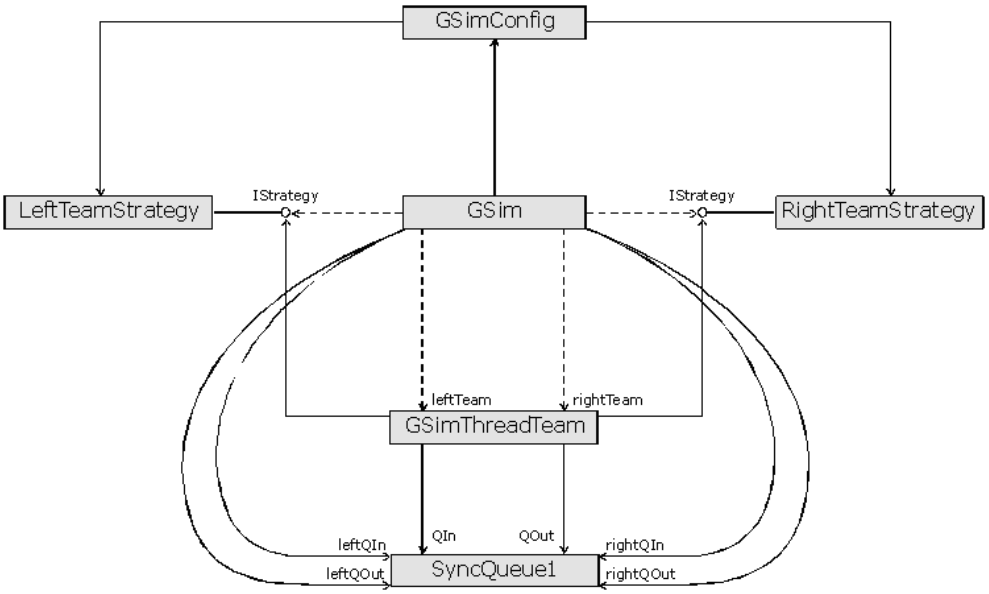

Figura 5: Clases involucradas con los componentes relativos a las estrategias de los equipos izquierdo y derecho

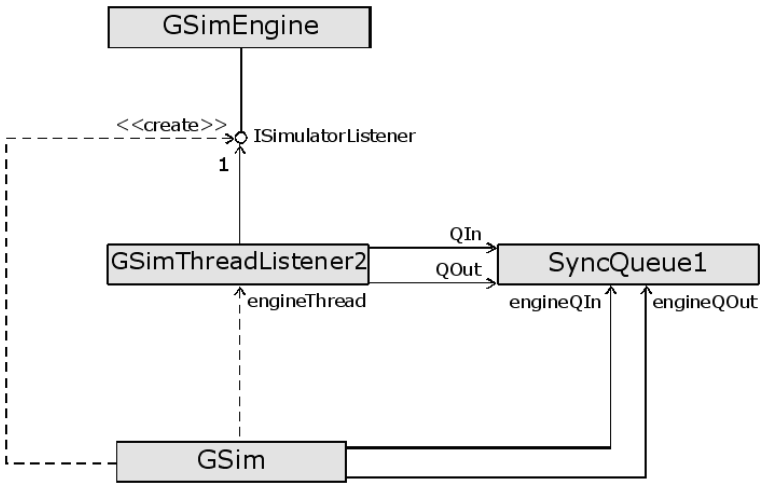

Figura 6: Clases involucradas con el componente que maneja la física de la simulación (*engine*)

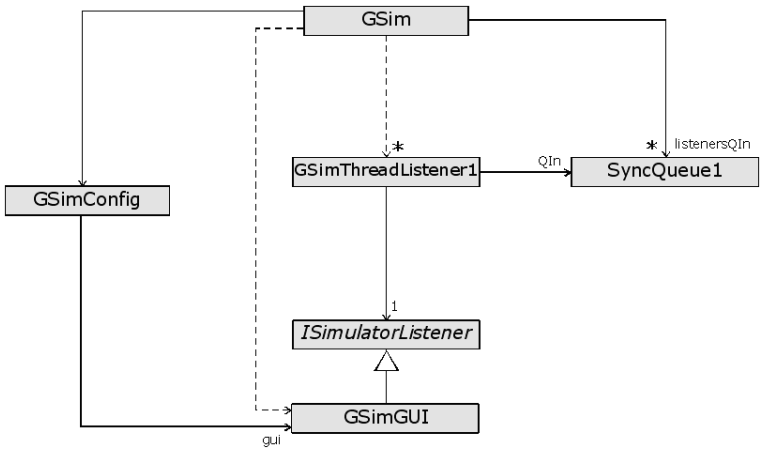

Figura 7: Clases involucradas con los componentes que generan la salida del simulador (*listeners*)

Intercambio entre GSim y la engine física y los listeners

La clase GSim intercambia el estado del juego con *GSimEngine* (*engine* física) y los *listeners* indirectamente a través de los hilos (instancias de *GSimThreadListener2* y *GSimThreadListener1*, respectivamente) y las colas asociadas (*SyncQueue1*). La figura 8 muestra las clases que modelan la información relacionada con el estado del juego, las cuales se describen a continuación:

GSimState: Las instancias de la clase *GSimState* guardan el estado de la pelota, de los propios robots y de las variables de juego (tiempo transcurrido -*currentGameTime*-, marcador de los equipos *leftScore* y *rightScore*-, quién posee la pelota *whosBall*- y estado del juego *gameState*-).

StateBall: Guarda la posición y la velocidad de la pelota.

StateRobot: Guarda la posición, rotación y velocidades de las ruedas de cada robot.

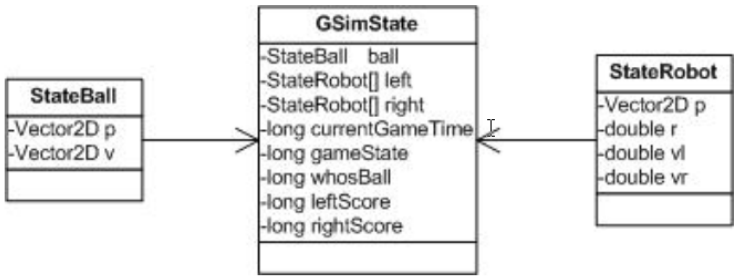

Figura 8: Datos intercambiados entre el módulo central del simulador (*GSim*) y la engine física (*GSimEngine*) y los listeners generadores de salida (por ejemplo, *GSimGUI*) (extraído de [Cas03]).

Es importante recalcar que al recibir el estado del juego *GSimEngine* sólo tiene en cuenta las velocidades de las ruedas de cada robot. A partir de esta información calcula el resto de los datos.

Intercambio entre *GSim* y los equipos

El intercambio entre *GSim* y las estrategias izquierda y derecha (interfaz *IStrategy*) se efectúa indirectamente a través de los hilos (instancias de *GSimThreadTeam*) y las colas asociadas. En la figura 9 se muestran las clases que modelan los datos intercambiados entre estos componentes.

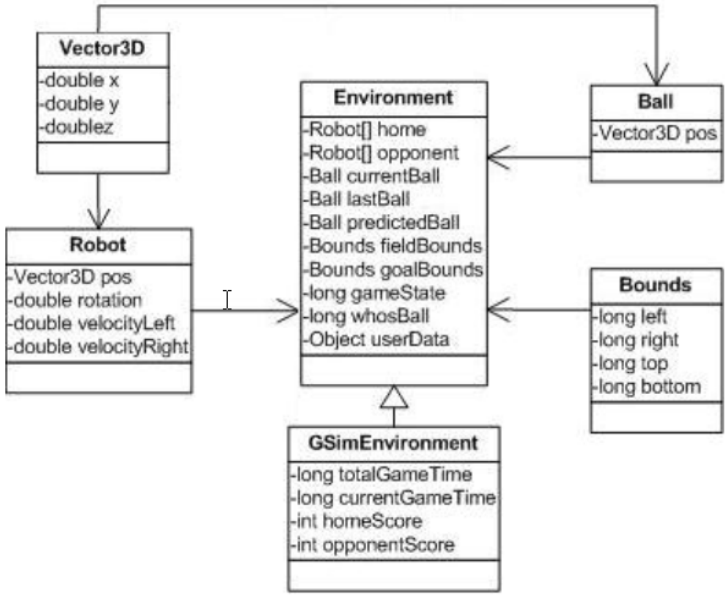

Figura 9: Datos intercambiados entre el módulo central del simulador (*GSim*) y las estrategias (que implementan la interfaz *IStrategy*) (extraído de [Cas03]).

Las clases *Environment*, *Robot*, *Ball*, *Bound* y *Vector3D* son definidas por el simulador de la FIRA para hacer posible la comunicación entre el simulador y la estrategia (ver [Referencia al archivo strategy.h]). La clase *Environment* es extendida por el simulador GSim con la finalidad de pasar los datos de tiempo y del marcador.

Cuando los datos van desde una estrategia a *GSim*, ésta tendrá en cuenta solamente las velocidades de las ruedas de los robots actualizados por la estrategia. Sin embargo, cuando los datos van desde *GSim* a una estrategia, ésta cuenta con todos los datos de *GSimEnvironment*.

2.2. Colaboración entre los componentes

2.2.1. Simulador GSim

A continuación se describe, en alto nivel, el ciclo de inicialización del módulo central del simulador (*GSim*) y demás componentes:

- Crear e inicializar el módulo *GSim*, a partir de la configuración (*GSimConfig*).
- Crear e inicializar el *estado del juego*[3] (*GSimState*) y el *ambiente* utilizado por las estrategias (*GSimEnvironment*).

[3] Nota: Para evitar ambigüedades, en esta subsección cuando se hace mención al *estado del juego* escrito en letra itálica, se está haciendo referencia estrictamente al concepto asociado a la clase *GSimState* (repositorio definido por

- Obtener la *GUI* desde la configuración. Concatenar la *GUI* en la lista de listeners.
- Crear e inicializar la *engine* física (*GSimEngine*).
- Obtener las estrategias izquierda y derecha desde la configuración, e inicializar las mismas.
- Inicializar los *listeners*.

Una vez inicializados los distintos componentes del simulador, da comienzo el ciclo principal donde se efectúa la simulación, descrito a continuación:

- Enviar el *estado del juego* (*GSimState*) a la *engine* física (*GSimEngine*) para actualizar las posiciones a partir de las velocidades de cada robot.
- Enviar el *ambiente* (*GSimEnvironment*) a ambas estrategias.
- Enviar el *estado del juego* a todos los *listeners*.
- Esperar un cierto intervalo de tiempo (retardo ocioso).
- Bloquear la ejecución hasta recibir el *estado del juego* desde la *engine* física, con las posiciones actualizadas.
- Recibir, sin bloquear la ejecución, el *ambiente* (sólo las velocidades de las ruedas) de ambas estrategias.
- Actualizar el *estado del juego* (sólo las velocidades de las ruedas) a partir del *ambiente* retornado por ambas estrategias.
- A partir del nuevo *estado del juego*, actualizar el *ambiente* (sólo las posiciones y las variables de juego).

El ciclo principal se repite hasta que se acabe el tiempo del juego. Luego de terminado el ciclo, deben liberarse los recursos.

Este ciclo comienza con el envío del *estado del juego* a la *engine* física, la cual sólo tendrá en cuenta las velocidades de cada robot para calcular las posiciones. Tanto el *estado del juego* inicial como el *ambiente* inicial se calculan en el ciclo de inicialización.

La recepción de resultados desde ambas estrategias es no-bloqueante. Según [Cas03], "lo que se quiere es que el simulador trabaje en tiempo real, es decir, que responda de la misma manera como si fuera una categoría real". De este documento se extrajo el diagrama de secuencia de la figura 10, correspondiente al ciclo de vida de los hilos que actúan sobre los distintos componentes del juego.

el simulador GSim que almacena el estado de la pelota, de los robots y de otras variables del juego, para comunicar el componente central con la *engine* física y los *listeners*) y no debe confundirse semánticamente con la noción ambigua del "estado del juego". Lo mismo ocurre con el *ambiente* (en letra itálica) que utilizan las estrategias, concepto ligado a la clase *GSimEnvironment* (extensión de la interfaz *Environment*, definida por el simulador de la FIRA, utilizada para comunicar el estado de la pelota, de los robots y de otras variables del juego entre el componente central y ambas estrategias).

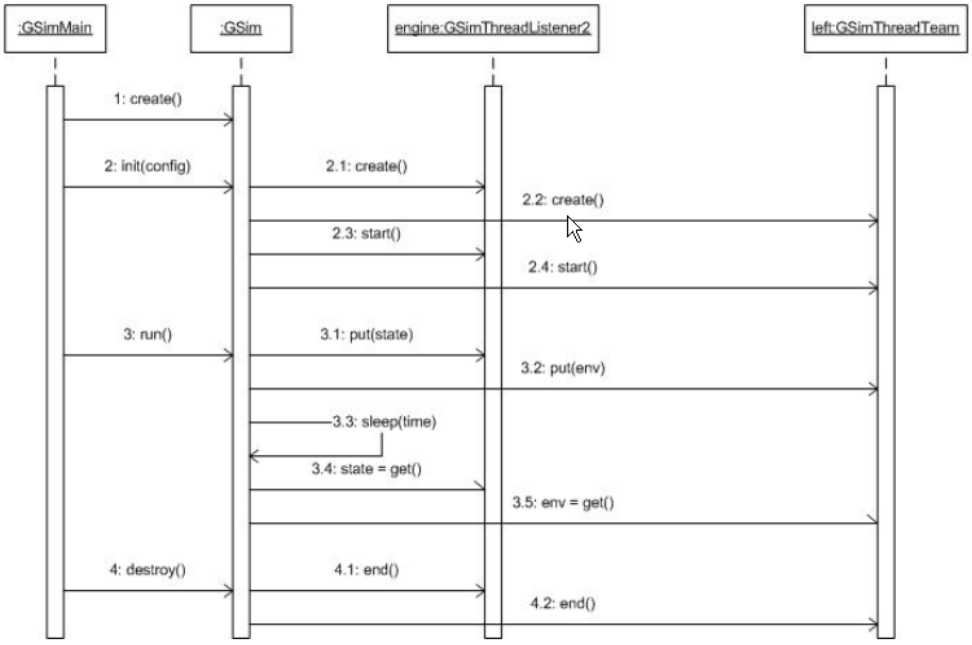

Figura 10: Diagrama de secuencia de la creación, intercomunicación y destrucción de los hilos de ejecución del simulador GSim.

Como se mencionó anteriormente, el hilo vinculado con la *engine* física es una instancia de la clase *GSimThreadListener2*, los hilos vinculados con los *listeners* son instancias de la clase *GSimThreadListener1*, mientras que los vinculados con las estrategias son instancias de *GSimThreadTeam*.

El penúltimo paso del ciclo tiene la finalidad de actualizar el *estado del juego* a partir del *ambiente*. En realidad, los únicos elementos que se actualizan del *estado del juego* son las velocidades de los robots. Luego, el paso siguiente actualiza el *ambiente* de ambas estrategias a partir del *estado del juego*. Aquí sólo se actualizan las posiciones recién calculadas, y las variables del juego (*gameState*, *whosBall*, *homeScore*, *opponentScore*, *currentGameTime* y *gameTime*).

En resumen, el ciclo de vida completo del simulador GSim está regido por los siguientes eventos:

- Creación de la instancia de la clase *GSim*.
- Invocación de la operación *init*, la cual recibe los parámetros de configuración.
- Invocación de la operación *run*, donde se da comienzo a la simulación.
- Invocación de la operación *destroy*, donde se liberan limpiamente los recursos.

Desfase entre los datos recibidos y enviados por las estrategias

Dentro del ciclo de simulación se produce un desfase en el *estado del juego* que reciben las estrategias y el *estado del juego* actual (actualizado por la *engine* física). Como muestra la figura 11, los datos de las velocidades recibidos en $t = T_1$ son los calculados -por las estrategias- para el *ambiente* anterior, en $t = T_0$.

Una buena estrategia debería tener en cuenta esta diferencia.

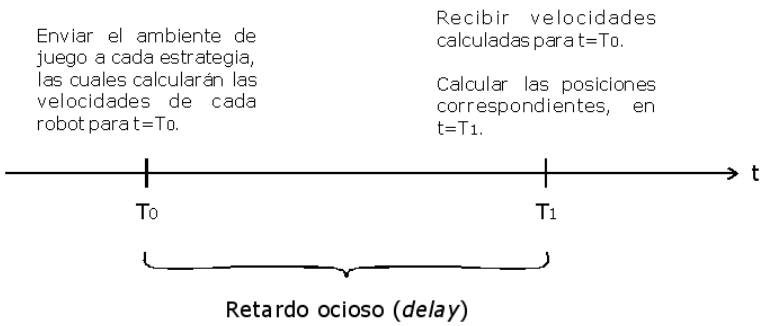

Figura 11: Desfase: las estrategias calculan velocidades para $t = T_0$, las cuales serán tenidas en cuenta recién en $t = T_1$.

2.2.2. Emulación del simulador de la FIRA

Uno de los objetivos iniciales de la construcción del simulador GSim era permitir probar estrategias, para luego migrarlas al simulador de la FIRA. Como se vio anteriormente, el módulo *GSim* no espera a que cada estrategia termine de calcular las velocidades, cosa que sí hace el simulador de la FIRA.

Con la finalidad de emular el comportamiento del simulador de la *Middle League SimuroSot*, se proporciona la clase *GSimFIRA*, que extiende de *GSim*.

GSimFIRA sólo modifica el ciclo principal donde se ejecuta la simulación. El esquema de ejecución pasa a ser el siguiente:

- Enviar el *ambiente* a ambas estrategias.
- Enviar el *estado del juego* a todos los *listeners*.
- Esperar un intervalo de tiempo (retardo ocioso).
- Bloquear la ejecución hasta recibir el *ambiente* (sólo las velocidades de las ruedas) de ambas estrategias.
- Actualizar el *estado del juego* (sólo las velocidades de las ruedas) a partir del *ambiente* retornado por ambas estrategias.
- Enviar el *estado del juego* a la *engine* física, para actualizar las posiciones de los robots.
- Bloquear la ejecución hasta recibir el *estado del juego* actualizado de la *engine* física.
- A partir del nuevo *estado del juego*, actualizar el *ambiente* (sólo las posiciones y las variables de juego).

Es importante recalcar que si bien el simulador espera a que ambas estrategias hayan terminado de calcular las velocidades, de ningún modo se ejecutan en forma secuencial. Como muestra el esquema, inicialmente se envía el *ambiente* a ambas estrategias, las cuales tienen toda la espera ociosa para realizar los cálculos, en forma concurrente.

La figura 12 muestra el diagrama de secuencia (extraido de [Cas03]) correspondiente al ciclo principal de los hilos que manejan los distintos componentes de la simulación.

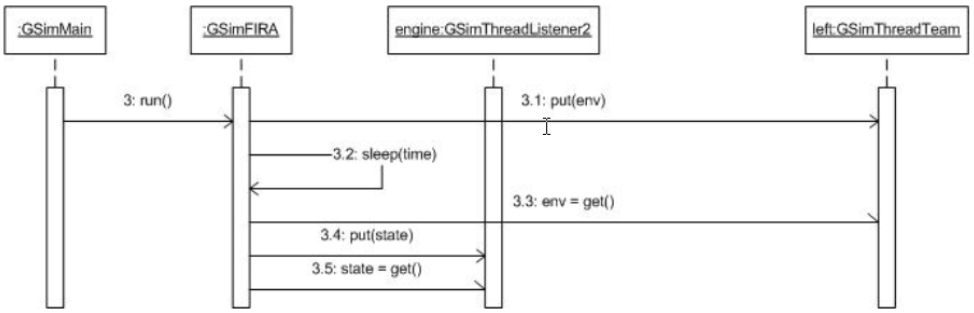

Figura 12: Diagrama de secuencia (sólo de la operación *run*) de la intercomunicación entre los hilos de ejecución del simulador para la clase *GSimFIRA*.

2.3. Modelo físico

Según [Cas03], el modelo de simulación del GSim no está basado estrictamente en leyes físicas, sino que es un híbrido que utiliza un modelo físico y un modelo geométrico. El modelo geométrico se aplica a la detección y resolución de colisiones, mientras que el modelo físico se aplica al resto.

2.3.1. Movimiento de los cuerpos

Robots

Para controlar el movimiento de los robots, los desarrolladores del simulador GSim consideraron que estos están constituidos por una carrocería de forma cúbica con dos ruedas independientes entre sí, controladas cada una por un motor y colocadas en lados opuestos del robot. Entonces, el movimiento del robot está determinado por la velocidad relativa de las dos ruedas.

Para un instante de tiempo t_i determinado, las siguientes magnitudes representan el estado del robot:

x_i: posición del centro del robot en el eje de las x.

y_i: posición del centro del robot en el eje de las y.

q_i: rotación del robot.

v_i: velocidad lineal del centro de masa.

ω_i: velocidad angular.

l_i: velocidad de la rueda izquierda.

r_i: velocidad de la rueda derecha.

Bajo las hipótesis de que las ruedas del robot no derrapan, y que éste se mueve sobre una superficie absolutamente plana, se tienen las siguientes ecuaciones:

$$v_i = \frac{l_{i-1} + r_{i-1}}{2} \tag{1}$$

$$\omega_i = \frac{r_{i-1} - l_{i-1}}{W} \tag{2}$$

$$x_i = x_{i-1} + v_{i-1} \cdot \triangle t \cdot cos\theta_{i-1} \tag{3}$$

$$y_i = y_{i-1} + v_{i-1} \cdot \triangle t \cdot sen\theta_{i-1} \tag{4}$$

$$\omega_i = \theta_{i-1} + \omega_{i-1} \cdot \triangle t \tag{5}$$

donde W es la distancia entre ambas ruedas, la cual se asume igual al largo del robot.

Para impedir que el robot cambie de velocidad instantáneamente, se restringe la aceleración máxima y mínima que se puede alcanzar utilizando la siguiente ecuación:

$$vel_i = vel_{i-1} + a \cdot \triangle t \tag{6}$$

donde $|a|$ debe ser menor que la aceleración máxima que puede alcanzarse. El valor de a se trunca a un valor máximo o mínimo en caso de que lo supere.

El intervalo en el cual pueden variar las velocidades de las ruedas del robot es $[-max_vel, max_vel]$ donde max_vel es un parámetro configurable del simulador.

Debido a que la restricción de las velocidades se realiza independientemente para cada rueda, podría llegar a suceder que -en caso de que la aceleración de cada rueda sobrepase los valores mínimos y máximos permitidos- ambas velocidades queden truncadas a un mismo valor. Si no se considera la velocidad relativa entre ambas ruedas, este truncamiento podría provocar que los robots se muevan en línea recta, cuando en realidad su intención es doblar.

Para solucionar este problema, luego de verificar la restricción de la aceleración se chequea que la relación de velocidades entre ambas ruedas se mantenga:

$\frac{l_i}{r_i} = \frac{lO_i}{rO_i}$ ó $\frac{r_i}{l_i} = \frac{rO_i}{lO_i}$, según corresponda,

donde l_i y r_i son las velocidades establecidas por la estrategia y lO_i y rO_i son estas mismas velocidades, pero luego de pasar por la restricción de la aceleración.

Pelota

Se considera a la pelota como si fuera una partícula y, por lo tanto, no contiene rotación ni velocidad angular dentro de sus variables de estado.

Las únicas variables de estado que se consideran son la posición de la pelota y la velocidad en cada instante de tiempo. Las ecuaciones que describen el movimiento son las siguientes:

$$vx_i = k \cdot vx_{i-1} \tag{7}$$

$$vy_i = k \cdot vy_{i-1} \tag{8}$$

$$px_i = px_i + vx_i \cdot \triangle t \tag{9}$$

$$py_i = py_i + vy_i \cdot \triangle t \tag{10}$$

donde la constante k es la constante de rozamiento de la pelota que cumple $0 \leq k \leq 1$.

2.3.2. Detección y resolución de colisiones

El simulador GSim distingue tres tipos de colisiones:

- Colisiones entre robots y paredes.
- Colisiones entre pelota y paredes.
- Colisiones entre robots y pelota.

No se tienen en cuenta las colisiones entre robots.

Las rutinas de detección y resolución de colisiones se aplican en cada paso de simulación, luego de calcular el siguiente estado de cada objeto.

Colisiones entre robots y paredes

Para detectar la colisión entre un robot y una pared se realizan las siguientes simplificaciones:

- Se considera la trayectoria del centro de masa del robot, la cual coincide con su centro geométrico. Dicha trayectoria es aproximada con un segmento de recta (los desarrolladores del GSim consideran que se trata de una buena aproximación, si el paso de simulación es pequeño). Los extremos de este segmento coinciden con las posiciones del robot antes y después de aplicar el paso de simulación.
- Cada pared es tratada como un segmento de recta, cuyos extremos son los puntos de intersección con las dos paredes adyacentes.

Como muestra la figura 13 este tipo de colisiones se detectan hallando la intersección entre ambos segmentos de recta. Si existe colisión, se considera el punto de intersección y el ángulo más pequeño, denominado *alfa*, entre las dos rectas involucradas. El valor de *alfa* pertenece al intervalo *[0,90]*.

La colisión se resuelve según el valor de *alfa*:

- Si *alfa* pertenece al intervalo *(45,90]* entonces el robot quedará enfrentado directamente a la pared. En otras palabras, se ajustará la orientación del robot para que el nuevo *alfa* sea igual a 90.
- Si *alfa* pertenece al intervalo *[0,45]*, entonces se cambiará la rotación del robot de manera que éste quede moviéndose en paralelo a la pared que acaba de chocar. En otras palabras, el nuevo *alfa* será igual a 0.

En ambos casos, la posición del robot se modifica para quedar junto a la pared y dentro de la cancha.

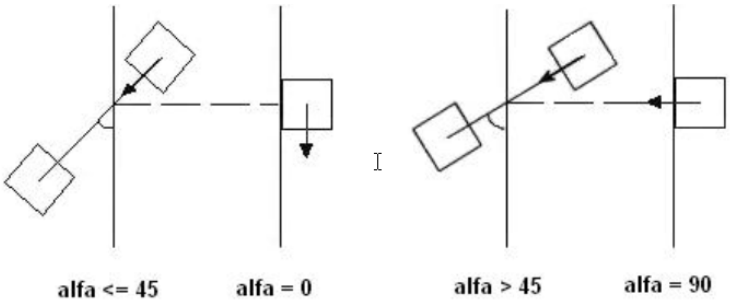

Figura 13: Colisión entre un robot y la pared izquierda de la cancha. La orientación del robot luego del choque depende del ángulo alfa con que éste colisiona con la pared. A la izquierda se presenta el antes y después del choque en caso de que . A la derecha se presenta el antes y después en el caso en que . (Extraído de [Cas03])

Las colisiones entre un robot y cualquiera de las ocho paredes limítrofes de la cancha son llevados, mediante una transformación *afín*, al caso canónico del choque entre un robot y la pared izquierda (ver figura 13).

La simplificación de considerar solamente el centro geométrico del robot provoca que la detección de colisiones no considere el volumen del mismo. Para solucionar esto, el mecanismo de detección de colisiones trabaja con paredes virtuales, que se encuentran a una distancia igual a la mitad del largo de un robot, de las paredes reales (ver figura 14).

Figura 14: Límites de la cancha en la detección de colisiones: el perímetro más grande representa las paredes reales de la cancha, mientras que el perímetro interior representa las paredes virtuales, con las cuales se interseca la trayectoria del centro del robot. La distancia entre ambos perímetros es igual a la mitad del largo de un robot. Análogamente, existe también un perímetro interno para la intersección con el centro de la pelota, pero la distancia entre éste y las paredes reales es igual al radio de la pelota.

Colisiones entre pelota y paredes

La detección de este tipo de colisiones es análogo al caso en que colisionan los robots con las paredes: se intersecan los segmentos de la trayectoria de la pelota (en un paso de simulación) y la

pared. Aquí también se consideran paredes virtuales, que están a una distancia igual al radio de la pelota, de las paredes reales.

La diferencia radica en cómo se resuelve la colisión, una vez detectada. Para esto, se considera que el choque es completamente elástico: no se pierde energía durante el mismo.

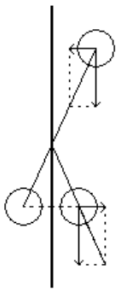

Figura 15: En la colisión entre la pelota y la pared izquierda de la cancha, luego del choque sólo cambian el sentido de la componente x de la velocidad y la coordenada y de la posición.

No se consideran los casos particulares de todas las paredes de la cancha, sino que mediante una transformación *afín*, todos son llevados al caso canónico en el cual chocan la pelota con la pared izquierda (ver figura 15).

Colisiones entre robots y pelota

La detección de la colisión entre un robot y la pelota se resuelve utilizando el mismo enfoque que la colisión entre la pelota y la pared, pero sustituyendo la pared por uno de los lados del robot. Además se tienen en cuenta las velocidades lineal y angular del robot.

Lo primero que se considera es la distancia entre el centro del robot y el centro de la pelota. Si esta distancia es menor que la mitad de la diagonal del robot, entonces puede que ambos objetos estén chocando.

Si esto último ocurre, entonces se calcula cuál de los cuatro lados del robot potencialmente colisionará con la pelota. Esto se logra determinando las cuatro distancias entre los vértices del robot y el centro de la pelota, y eligiendo los dos vértices más cercanos a éste. Dichos vértices determinan qué lado del robot estaría involucrado en la colisión.

Como muestra la figura 16, lo que se hace es hallar en cuál de las cuatro regiones del plano, definidas por las diagonales del robot, se encuentra el centro de la pelota.

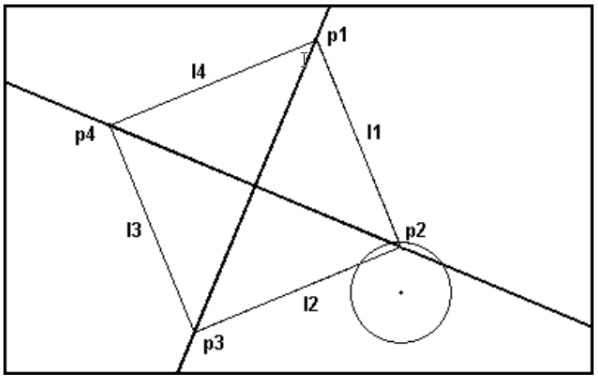

Figura 16: Las diagonales del robot dividen el plano en cuatro regiones. Los puntos *p1*, *p2*, *p3* y *p4* son los vértices del robot; *l1*, *l2*, *l3* y *l4* son sus lados. El punto *p1* es aquél que se encuentra arriba a la derecha cuando el robot tiene una rotación de 0. (Extraido de [Cas03])

Una vez hallado el lado del robot que estaría involucrado en un choque, se realiza una transformación *afín* de las coordenadas del robot y de la pelota, para llevarlos al caso canónico que se muestra en la figura 17.

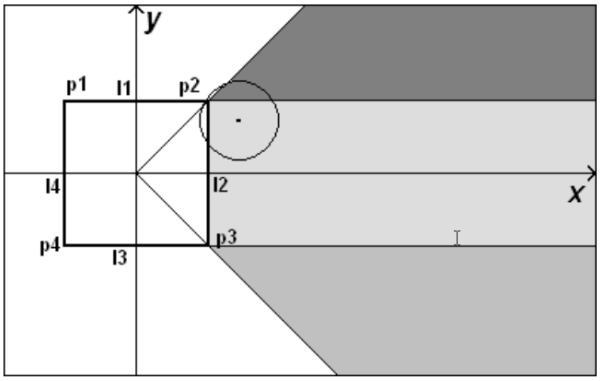

Figura 17: Caso canónico para detectar la colisión entre un robot y la pelota. Inmediatamente antes del choque, la pelota puede encontrarse en una de las tres zonas indicadas en distintas tonalidades de gris. (Extraido de [Cas03])

Hasta el momento sólo se ha comprobado que la pelota se encuentra al menos a distancia de choque con algún vértice del robot, y se ha determinado qué lado estaría involucrado en la colisión. Para detectar que se ha producido una colisión, se verifica que exista una interpenetración entre ambos objetos. Como muestra la figura 17, la pelota puede encontrarse en la zona media, superior o inferior.

Si la pelota se encuentra en la zona media, se detecta que ha ocurrido una colisión si la misma está interpenetrando el cuerpo del robot.

Si la pelota se encuentra en la zona superior o inferior, puede suceder que la misma se encuentre exactamente en la diagonal o no. Si está en la diagonal límite, entonces ha ocurrido una colisión. Si

no está en una diagonal, entonces se realiza la misma comprobación que para la zona media para detectar la colisión.

Si bien la resolución de choques en la zona media es similar al tipo de colisión entre la pelota y las paredes, en este caso se tiene en cuenta la velocidad lineal y angular del robot. Según sea positiva o negativa la velocidad lineal en x de la pelota, se aplica -respectivamente- una de las siguientes ecuaciones:

$$v_{ball_x} = \text{máx}(v_{ball_x}, v_{robot_x}) + \omega_{robot_x} \cdot r \tag{11}$$

$$v_{ball_x} = |(v_{ball_x}, v_{robot_x})| + \omega_{robot_x} \cdot r \tag{12}$$

donde r es la distancia entre el punto de choque y el centro del robot.

Si la pelota está en alguna de las diagonales, también se aplican las ecuaciones anteriores para la componente y de la velocidad, ya que ésta también cambia (ver Figura 18).

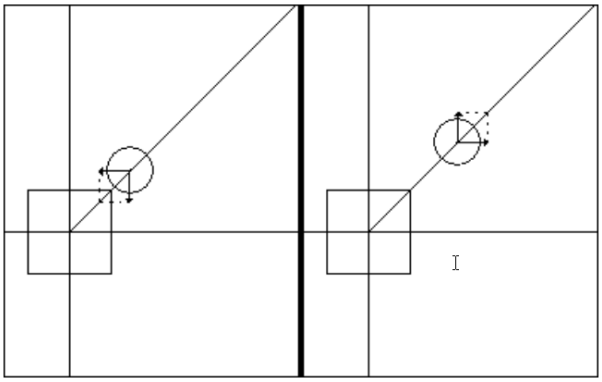

Figura 18: Colisión de uno de los vértices del robot con la pelota, cuando ésta se desplaza por la diagonal. (Extraido de [Cas03])

3. Análisis

3.1. Criterios de análisis

Se realizará un análisis del simulador GSim basado en los siguientes criterios:

- Flexibilización en la forma en que interactúan los distintos componentes.
- Regulación de la velocidad de la simulación.
- Capacidad de determinar el estado inicial de la simulación y el intervalo de ejecución.
- Aproximación al modelo de simulación de la FIRA.
- Capacidad de prescindir de la interfaz gráfica y de contar con registros (*logs*) completos que permitan reconstruir enteramente el juego.
- Contar con una API que permita automatizar el proceso de aprendizaje.
- Extensibilidad del simulador.

3.2. Flexibilidad de interacción

Resumen

En la simulación interactúan (por lo menos) tres componentes: el equipo izquierdo, el equipo derecho y el controlador de la simulación.

El controlador es quien administra la ejecución de los equipos, estableciendo en cada momento qué equipo deberá ejecutarse. Esta administración puede ser *expropiativa* o *no expropiativa*. En el primer caso, el controlador entrega el estado del juego a ambos equipos y da un plazo a ambas estrategias para que devuelvan la configuración de la jugada; vencido este plazo, los resultados proporcionados por una o ambas estrategias serán ignorados (o a lo sumo serán tenidos en cuenta en la próxima iteración, produciéndose un desfase). En el segundo caso, el controlador entrega el estado de juego a ambos equipos, pero espera a que cada equipo termine de ejecutar su estrategia y retornar la configuración.

Un ejemplo de administración -o simulación- *no expropiativa* es la forma en que funciona el simulador de la FIRA.

Debido a que en las competencias no simuladas ambos equipos operan al mismo tiempo, es vital desarrollar estrategias que minimicen el tiempo de ejecución. La simulación *expropiativa* puede ayudar a construir soluciones más flexibles, que faciliten la migración a otras ligas que involucren la interacción entre robots reales.

Simulador GSim

El simulador GSim permite trabajar de las dos formas. Si se utiliza el módulo *GSim*, la simulación será *expropiativa*; si se utiliza el módulo *GSimFIRA*, la simulación será *no expropiativa*, emulando así el simulador de la FIRA.

En caso de utilizarse el módulo *GSimFIRA*, ambas estrategias se ejecutan en forma concurrente. La diferencia radica en que el simulador espera a que cada estrategia entregue su resultado. Si ambas estrategias son *CPU-intensive*, entonces en el fondo el tiempo de ejecución será similar al que se obtendría en una ejecución secuencial, si se contara con un único procesador.

3.3. Velocidad de simulación

Resumen

Resultaría útil que el paso de simulación se redujera, cuanto mucho, al tiempo que demora cada equipo en ejecutar su estrategia, en el caso de una simulación *no expropiativa*, o a un tiempo mínimo tolerable, en el caso *expropiativo*.

En el simulador de la FIRA, por ejemplo, entre ejecución y ejecución existe un retardo de tiempo ocioso que intenta llevar la simulación a tiempos de escala humana. Si bien esto permite una mejor visualización del juego, entorpece el proceso de aprendizaje.

Una reducción del orden del paso de simulación permitiría que los mecanismos de aprendizaje adquieran experiencia en menos tiempo.

Simulador GSim

GSim permite configurar una serie de parámetros relacionados con la velocidad de simulación. Estos parámetros son *tick_interval*, *aceleration_rate*, *game_time*, *delay_time* y *steps_per_tick*. El cuadro 1 describe cada uno de ellos.

Claves del archivo de configuración	Atributos de la clase *GSimConfig*	Descripción
TICK_INTERVAL	*tick_interval*	Tiempo en milisegundos que quedará esperando el hilo principal del simulador para obtener datos tanto del motor de simulación como de los equipos. Por definición, este intercambio de datos se denomina *tick*.
ACELERATION_RATE	*aceleration_rate*	Relación entre el tiempo real y el tiempo simulado. Un valor de 1 significa tiempo real a la par con el tiempo de simulación (es decir, cada segundo de la realidad equivale a un segundo en la simulación). Un valor de 2 significa que cada segundo transcurrido en tiempo real equivale a dos segundos transcurridos en tiempo de simulación, mientras que un valor de 0.5 determina que por cada segundo transcurrido en la realidad hay medio segundo transcurrido en tiempo de simulación.
DELAY_TIME	*delay_time*	Tiempo en milisegundos que se le adicionará al tiempo indicado por el intervalo de *tick* para que el hilo principal se quede trancado esperando antes de pasar al siguiente *tick*.
STEPS_PER_TICK	*steps_per_tick*	Cantidad de pasos de simulación que ejecutará el motor de simulación en cada *tick*.

Cuadro 1: Parámetros que configuran la velocidad de simulación. (Extraido de [Cas03])

3.4. Estado inicial e intervalo de ejecución

Resumen

Si el simulador permitiera establecer el estado inicial -posición inicial de los jugadores y la pelota- y el segmento de tiempo de ejecución, se podría enfocar una técnica determinada de *Machine Learning* al aprendizaje de una jugada específica.

Esto no sólo reduciría el tiempo de adquisición de experiencia sino que, por ejemplo, tal vez ayude a que técnicas como reconocimiento de patrones converjan a soluciones útiles.

Simulador GSim

Así como se puede configurar la velocidad de simulación, GSim también permite configurar la duración total del juego y la posición y la velocidad inicial de la pelota y los robots. El cuadro 2 describe los parámetros involucrados.

Claves del archivo de configuración	Atributos de la clase *GSimConfig*	Descripción
GAME_TIME	*game_time*	Duración total del tiempo de simulación en milisegundos.
BALL_POS_X	*ball.p.x*	Posición inicial en *x*.
BALL_POS_Y	*ball.p.y*	Posición inicial en *y*.
BALL_VEL_X	*ball.v.x*	Velocidad inicial en *x*.
BALL_VEL_Y	*ball.v.y*	Velocidad inicial en *y*.
LEFT_PLAYER_POS_X_i	*left_players[i].p.x*	Posición inicial en *x*.
LEFT_PLAYER_POS_Y_i	*left_players[i].p.y*	Posición inicial en *y*.
LEFT_PLAYER_ROT_i	*left_players[i].r*	Rotación inicial en grados en el intervalo *[-180, 180]*. El ángulo 0 coincide con el eje *x* en el sentido positivo. Así, siguiendo un movimiento antihorario el ángulo será positivo, mientas que siguiendo un movimiento horario el ángulo será negativo.
LEFT_PLAYER_VEL_L_i	*left_players[i].vl*	Velocidad inicial de la rueda izquierda.
LEFT_PLAYER_VEL_R_i	*left_players[i].vr*	Velocidad inicial de la rueda derecha.
RIGHT_PLAYER_POS_X_i	*right_players[i].p.x*	Posición inicial en *y*.
RIGHT_PLAYER_POS_Y_i	*right_players[i].p.y*	Posición inicial en *x*.
RIGHT_PLAYER_ROT_i	*right_players[i].r*	Rotación inicial en grados en el intervalo [-180, 180].
RIGHT_PLAYER_VEL_L_i	*right_players[i].vl*	Velocidad inicial de la rueda izquierda.
RIGHT_PLAYER_VEL_R_i	*right_players[i].vr*	Velocidad inicial de la rueda derecha.

Cuadro 2: Parámetros que configuran la posición y la velocidad inicial de la pelota y los robots. (Extraído de [Cas03])

3.5. Aproximación al modelo de la FIRA

Resumen

Dado que uno de los objetivos es competir en la *Middle League SimuroSot*, no debe perderse de vista que todo el aprendizaje adquirido sobre la base del nuevo simulador, debe ser trasladado al simulador de la FIRA.

Esto propone desafíos interesantes, como intentar emular el mundo físico (prácticamente desconocido) que rige la simulación de la FIRA. Un aspecto importante es que el simulador estudiado cuente con parámetros que permitan calibrar la física utilizada.

Simulador GSim

La *engine* física del GSim resuelve geométricamente las colisiones entre la pelota y los robots, entre la pelota y las paredes y entre los robots y las paredes. Sin embargo, no detecta ni resuelve las colisiones entre robots. Esto representa una desventaja, ya que no se pueden correr simulaciones de juego completo que reflejen la realidad.

Respecto al estado del juego y al ambiente que utilizan las estrategias, si bien GSim homologa la interfaz que utiliza el simulador de la FIRA, deja sin implementar (está marcado como trabajo futuro) la variable de juego *gameState*. De acuerdo con la interfaz del simulador de la FIRA, la variable *gameState* puede tomar uno de cinco valores:

- 1 = Free ball
- 2 = Place kick (Kickoff)
- 3 = Penalty kick
- 4 = Free kick
- 5 = Goal kick

Un aspecto interesante del simulador GSim tiene que ver con que éste permite configurar algunos parámetros, para ajustarse -por ejemplo- al simulador de la FIRA. Estos parámetros son las dimensiones de la cancha, configuración de la pelota y configuración de los robots (ver cuadro 4).

Claves del archivo de configuración	Atributos de la clase GSimConfig	Descripción
FTOP	*ftop*	Coordenada y del límite norte de la cancha.
FBOT	*fbot*	Coordenada y del límite sur de la cancha.
FLEFTX	*fleftx*	Coordenada x del límite oeste de la cancha.
FRIGHTX	*frightx*	Coordenada x del límite este de la cancha.
GTOPY	*gtopy*	Coordenada y del límite norte de los arcos.
GBOTY	*gboty*	Coordenada y del límite sur de los arcos.
GLEFT	*gleft*	Coordenada x del límite oeste del arco izquierdo.
GRIGHT	*gright*	Coordenada x del límite este del arco derecho.
PENALTY_KICK_LEFT_X	*penalty_kick_left.x*	Coordenada x del punto penal del arco izquierdo.
PENALTY_KICK_LEFT_Y	*penalty_kick_left.y*	Coordenada y del punto penal del arco izquierdo.
PENALTY_KICK_RIGHT_X	*penalty_kick_left.x*	Coordenada x del punto penal del arco derecho.
PENALTY_KICK_RIGHT_Y	*penalty_kick_right.y*	Coordenada y del punto penal del arco derecho.
FOUL_KICK_LEFT_X	*foul_kick_left.x*	Coordenada x del punto de tiro libre del arco izquierdo.
FOUL_KICK_LEFT_Y	*foul_kick_left.y*	Coordenada y del punto de tiro libre del arco izquierdo.
FOUL_KICK_RIGHT_X	*foul_kick_right.x*	Coordenada x del punto de tiro libre del arco derecho.
FOUL_KICK_RIGHT_Y	*foul_kick_right.y*	Coordenada y del punto de tiro libre del arco derecho.
FREE_KICK_LEFT_TOP_X	*free_kick_left_top.x*	Coordenada x del punto de pique del cuadrante noroeste.
FREE_KICK_LEFT_TOP_Y	*free_kick_left_top.y*	Coordenada y del punto de pique del cuadrante noroeste.

Cuadro 4: Parámetros de configuración que permiten calibrar el simulador GSim. (Extraido de [Cas03])

Claves del archivo de configuración	Atributos de la clase GSimConfig	Descripción
FREE_KICK_LEFT_BOT_X	*free_kick_left_bot.x*	Coordenada x del punto de pique del cuadrante suroeste.
FREE_KICK_LEFT_BOT_Y	*free_kick_left_bot.y*	Coordenada y del punto de pique del cuadrante suroeste.
FREE_KICK_RIGHT_TOP_X	*free_kick_right_top.x*	Coordenada x del punto de pique del cuadrante noreste.
FREE_KICK_RIGHT_TOP_Y	*free_kick_right_top.y*	Coordenada y del punto de pique del cuadrante noreste.
FREE_KICK_RIGHT_BOT_X	*free_kick_right_bot.x*	Coordenada x del punto de pique del cuadrante sureste.
FREE_KICK_RIGHT_BOT_Y	*free_kick_right_bot.y*	Coordenada y del punto de pique del cuadrante sureste.
KICK_OFF_X	*kick_off.x*	Coordenada x del punto de saque.
KICK_OFF_Y	*kick_off.y*	Coordenada y del punto de saque.
CORNER_SIDE_LENGTH	*corner_side_length*	Largo del lado del triángulo isósceles del corner
CENTER_RADIUS	*center_radius*	Radio del círculo central
HEIGHT_BIG_AREA	*height_big_area*	Largo en el eje y del área grande
WIDTH_BIG_AREA	*width_big_area*	Largo en el eje x del área grande
HEIGHT_AREA	*height_area*	Largo en el eje y del área chica
WIDTH_AREA	*width_area*	Largo en el eje x del área chica
BALL_RAD	*ball_rad*	Radio de la pelota
BALL_FRICTION	*ball_friction*	Coeficiente de rozamiento de la pelota (ver 2.3).

Cuadro 6: (Continuación del cuadro 4) Parámetros de configuración que permiten calibrar el simulador GSim. (Extraido de [Cas03])

Claves del archivo de configuración	Atributos de la clase GSimConfig	Descripción
PLAYER_SIDE	*player_side*	Tamaño de lado de los robots. Los robots reales son de forma cúbica mientras que en el simulador (por ser éste 2D) son de forma cuadrada.
PLAYER_MAX_VEL	*player_max_vel*	Valor absoluto de la velocidad máxima alcanzada por las ruedas de cada robot. El intervalo en el cual los robots podrán variar las velocidades de sus ruedas independientemente estará dado por este parámetro y será *[-player_max_vel, player_max_vel]* (ver 2.3).
PLAYER_ACELERATION	*player_aceleration*	Valor absoluto de la aceleración máxima que podrá alcanzar cada robot (ver 2.3).
PLAYER_BREAK_RATE	*player_break_rate*	Valor absoluto de la desaceleración máxima que podrá alcanzar cada robot (ver 2.3).

Cuadro 8: (Continuación del cuadro 6) Parámetros de configuración que permiten calibrar el simulador GSim. (Extraido de [Cas03])

Como muestra la figura 6 (ver 2.1), actualmente la clase *GSim* es la responsable de crear a la *engine* física. En otras palabras, el simulador GSim no permite cambiar la *engine* sin modificar el código. Contrariamente, sí permite hacerlo para otros componentes, como ser la *GUI* o las estrategias de ambos equipos.

3.6. Interfaz gráfica y registro del juego

Resumen

La salida a una GUI puede entorpecer la ejecución. El simulador debería permitir prescindir de la misma. Asimismo, toda la salida debe poder ser recogida a posteriori para ser analizada potencialmente por cualquier algoritmo de aprendizaje *offline* como dato de entrada. Para esto, el registro generado debe ser lo suficientemente completo como para poder reconstruir enteramente una jugada.

Simulador GSim

El simulador permite configurar qué clase se encargará de manejar la *GUI* sin tener que modificar el código. Esto se logra actualizando la clave GUI_CLASS del archivo de configuración (esta clave está asociada al atributo *gui_class* de *GSimConfig*). Como todo *listener*, esta clase deberá

implementar la interfaz *ISimulatorListener*. Este mecanismo permite, por lo tanto, flexibilizar la forma en que se maneja la salida, pudiendo incluso anularse completamente.

La arquitectura del GSim permite disponer de varios *listeners* generadores de salida. Sin embargo, la implementación actual no permite trabajar con más *listeners* que la *GUI*. Por ejemplo, la clase *GSimConfig* no cuenta con mecanismos para levantar una lista de *listeners* del archivo de configuración. Esto impide contar con un componente que genere el registro y una *GUI* al mismo tiempo.

Una particularidad tanto de la arquitectura como de la implementación del GSim es el trato preferencial que se le da a la *GUI* frente a los potenciales *listeners* generadores de salida (ver figura 7). Sin embargo, desde un punto de vista dinámico, esto sólo ocurre durante la etapa de inicialización, donde la clase *GSimGUI* es agregada por la clase *GSim* a la lista de *listeners*. Posteriormente, es "tratada" como un *listener* más en cada ciclo de simulación. El diseño podría mejorarse quitándole a *GSimConfig* y *GSim* responsabilidades directas sobre *GSimGUI*, aumentando de esta forma la *mantenibilidad* del simulador al disminuír el acoplamiento entre los componentes.

Con respecto al registro del juego, en 2.1 se vio que el módulo principal del simulador no espera a que los *listeners* terminen de escribir la salida. Si a ésto se le suma el hecho de que las colas que reciben los mensajes sólo pueden almacenar un único elemento, podría ocurrir que parte de la salida se pierda. Este comportamiento es inaceptable si lo que se quiere es contar con un rastro de cada paso de la simulación.

3.7. API para automatizar aprendizaje

Resumen

Todas las funcionalidades mencionadas hasta el momento deben poder realizarse sin necesidad de que intervenga un operador. El simulador debería proporcionar una API que permita automatizar los procesos de aprendizaje.

Facciolo y Laurenzo [LF04] han sido categóricos en este punto, al describir el ambiente al que se enfrentaron en la búsqueda de una posible herramienta de análisis de estrategias de fútbol de robots *Middle League SimuroSot*, que utiliza reconocimiento de patrones: "el espantoso (*sic*) entorno requiere un operador constantemente".

Simulador GSim

El simulador GSim está implementado en Java y cuenta con una API extensible, que permite incluir las funcionalidades del simulador dentro de otro código. Además, proporciona facilidades para configurar los distintos componentes que trabajan en la simulación.

3.8. Extensibilidad

Resumen

Sería deseable que la arquitectura del simulador permita agregarle otros componentes, o mejorar los ya existentes.

Simulador GSim

Es posible extender la clase *GSim*, encargada de la simulación, habilitando la posibilidad de cambiar la *engine* física actual por una que utilice librerías físicas para manejar las colisiones, por ejemplo. Sin embargo, para hacerlo es necesario comprender los detalles de la implementación de *GSim*, debido a que la arquitectura no contempla la posibilidad de efectuar este cambio.

El diseño podría mejorarse aumentando las vías de extensión de la aplicación, por ejemplo, abriendo la posibilidad de que el usuario programador modifique la *engine* sin tener que cambiar la clase *GSim*.

4. Conclusiones

Es necesario contar con una alternativa al simulador proporcionado por la FIRA, debido a que éste no proporciona un ambiente apropiado para experimentar técnicas de *Machine Learning*. En la búsqueda de esta alternativa, el simulador GSim resulta ser un punto de partida interesante, ya que posee las siguientes fortalezas:

- Su diseño e implementación están orientados a objetos.
- Proporciona una interfaz del programador extensible y el hecho de estar desarrollado en Java facilita tanto el desarrollo como la comprensión del código existente.
- Está diseñado para soportar múltiples *listeners* que generen una salida a partir del estado del juego.
- Los distintos componentes responsables de la simulación trabajan concurrentemente, posibilitando el aumento de la velocidad de ejecución del juego.
- Permite ampliar el modelo de simulación de la *Middle League SimuroSot* de la FIRA, con miras a desarrollar estrategias que se adecúen a las ligas reales (no simuladas).
- Permite configurar la velocidad del simulador, así como también el estado inicial del juego.
- Permite calibrar algunos aspectos de la simulación, para ajustarlo al simulador de la FIRA.
- Las estrategias desarrolladas utilizando el GSim pueden migrarse al simulador de la FIRA con facilidad.

Por otro lado, se ha encontrado una serie de debilidades que ameritan las siguientes mejoras:

- Mejorar la *engine* física, para permitir la detección y resolución de colisiones entre robots.
- Ampliar las facilidades de configuración para permitir cambiar la *engine* física proporcionada por defecto.
- Retocar la arquitectura para ampliar las vías de extensión del simulador GSim.
- Extender el estado del juego para que utilice la variable *gameState*.
- Ampliar las facilidades de configuración para que se admitan varios *listeners*.
- Mejorar el diseño para que la *GUI* sea tratada como un *listener* más en la etapa de inicialización, con la finalidad de disminuir el acoplamiento entre los módulos.
- Permitir que el ciclo principal de simulación espere a que la salida haya sido escrita. Sería interesante además que esta característica sea configurable para cada *listener*.

Referencias

[Cas03] Alvaro Castroman, *Simulador gsim*, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay, 2003. 4, 9, 11, 12, 13, 15, 16, 19, 21, 22, 24, 25, 27, 28, 29

[LF04] Tomás Laurenzo and Gabrielle Facciolo, *Una herramienta de análisis de estrategias de fútbol de robots middle league simurosot.* 30

A. Interfaz del simulador de la FIRA

```
#ifndef Strategy_H
#define Strategy_H
// The following ifdef block is the standard way of creating macros
// which make exporting from a DLL simpler. All files within this DLL
// are compiled with the STRATEGY_EXPORTS symbol defined on the command
// line. this symbol should not be defined on any project that uses this
// DLL. This way any other project whose source files include this file
// see STRATEGY_API functions as being imported from a DLL, wheras this
// DLL sees symbols defined with this macro as being exported.
#ifdef STRATEGY_EXPORTS
#define STRATEGY_API __declspec(dllexport)
#else
#define STRATEGY_API __declspec(dllimport)
#endif
#include "string.h"
#include "stdio.h"
const long PLAYERS_PER_SIDE = 5;
// gameState
const long FREE_BALL = 1;
const long PLACE_KICK = 2;
const long PENALTY_KICK = 3;
const long FREE_KICK = 4;
const long GOAL_KICK = 5;
// whosBall
const long ANYONES_BALL = 0;
const long BLUE_BALL = 1;
const long YELLOW_BALL = 2;
// global variables -- Useful field positions ... maybe???
const double FTOP = 77.2392;
const double FBOT = 6.3730;
const double GTOPY = 49.6801;
const double GBOTY = 33.9320;
const double GRIGHT = 97.3632;
const double GLEFT = 2.8748;
const double FRIGHTX = 93.4259;
const double FLEFTX = 6.8118;
typedef struct
{
    double x, y, z;
} Vector3D;
typedef struct
{
    long left, right, top, bottom;
} Bounds;
typedef struct
{
    Vector3D pos;
    double rotation;
    double velocityLeft, velocityRight;
} Robot;
typedef struct
```

```
{
    Vector3D pos;
    double rotation;
} OpponentRobot;
typedef struct
{
    Vector3D pos;
} Ball;
typedef struct
{
    Robot home[PLAYERS_PER_SIDE];
    OpponentRobot opponent[PLAYERS_PER_SIDE];
    Ball currentBall, lastBall, predictedBall;
    Bounds fieldBounds, goalBounds;
    long gameState;
    long whosBall;
    void *userData;
} Environment;
typedef void (*MyStrategyProc)(Environment*);
/* MUST BE IMPLEMENTED */
// implement this function to allocate user data and assign
// to Environment->userData
extern "C" STRATEGY_API void Create ( Environment *env );
extern "C" STRATEGY_API void Strategy ( Environment *env );
// implement this function to free user data created in Create (Environment*)
extern "C" STRATEGY_API void Destroy ( Environment *env );
#endif // Strategy_H
```