Capítulo IV. Implementación

4.1. Conexión con la IR Tower

Cuando se diseña el comportamiento de un robot, por lo general se requiere que este sea independiente de cualquier otra información aparte de la obtenida por medio de sensores. Cargarle las instrucciones de lo que debe hacer es suficiente para que el robot realice cada una de las tareas que se planearon para él.

La memoria limitada del RCX siempre será un obstáculo, ya que es muy pequeña. No permite descargarle una gran cantidad de instrucciones o métodos que requieran un amplio uso de la memoria para calcular diferentes valores. La mejor solución a esto es hacer que una computadora realice estos cálculos y después los envíe al RCX como un simple valor.

Para este proyecto se requirió llegar un poco más allá de un comportamiento autónomo de los robots. Se necesitaba una comunicación infrarroja entre cada uno de los robots y un sistema funcionando en una computadora, con la finalidad de enviar comandos a un robot que no necesite ningún tipo de procesamiento complejo.

La parte central de toda esta investigación consiste en lograr esta conectividad PC – RCX, obteniendo el mayor alcance posible del envío de información, además de obtener una respuesta en un tiempo considerablemente corto. En el API de leJOS existen diversas clases que le permiten a la IR Tower enviar o recibir datos y todas se encuentran en el paquete josx.rcxcomm.

4.1.1. RCXAbstractPort

Fue escrita por Brian Bagnall y Lawrie Griffiths [WEB21]. Implementa el control de flujos de entrada y salida de datos. Usa un manejador de paquetes para enviarlos y recibirlos.

Al ser una clase abstracta, cada versión específica del RCXAbstractPort debe sobrescribir el constructor con el manejador de paquetes que requiera. Sus subclases directas son: RCXPort, RCXF7Port, RCXLNPPort y RCXLNPAddressingPort.

4.1.2. RCXPort

Fue escrita por Brian Bagnall y Lawrie Griffiths [WEB21]. Utiliza un sistema de comunicaciones de bajo nivel. Asegura la transmisión de todos los paquetes y tiene la funcionalidad de terminar la comunicación cuando la IR Tower esté fuera de rango, y continuarla cuando regrese a una posición donde la señal sí pueda ser recibida. Esta clase no soporta el envío de mensajes diseccionados, en su lugar los envía a todos los dispositivos receptores dentro de su alcance de transmisión.

4.1.3. RCXF7Port

Fue escrita por Brian Bagnall y Lawrie Griffiths [WEB21]. Utiliza un sistema de comunicaciones serial, la cual usa rutinas específicas del RCX sólo soportadas por los *opcodes* del LEGO *firmware*. El *opcode* F7, que se usa para definir un mensaje, es utilizado para permitir una comunicación en ambos sentidos.

Este protocolo no es muy recomendado para usarse con los flujos de datos de Java. Los *bytes* tienden a perderse o a ser enviados en más de una ocasión. Puede usarse para intentar una posible interacción con programas que utilicen el *firmware* de LEGO, como los hechos en NQC.

4.1.4. RCXLNPPort

Esta subclase usa el Protocolo de Red de LegOS (LNP). Esto permite la comunicación con programas hechos con LegOS, tanto para el RCX como para la PC. Asegura que los paquetes no están corruptos, pero no asegura que lleguen correctamente al ser enviados, los

paquetes pueden perderse en el camino. Tampoco soporta el direccionamiento de los paquetes [WEB21].

4.1.5. RCXLNPAddressingPort

Es básicamente lo mismo que RCXLNPPort, pero esta soporta un direccionamiento. Las direcciones consisten en un identificador del *host* de los 4 *bits* más significativos y de un número de puerto de los 4 *bits* menos significativos. Una clara desventaja de esta clase es que sólo permite conexiones punto a punto, lo cual limita la comunicación de una computadora a un RCX, o bien, entre dos RCX [WEB21].

4.1.6. RCXBean

Se encuentra en el paquete josx.rcxcomm.RCXBean. Esta clase es de un nivel más alto que maneja toda la comunicación con el RCX. Fue escrita por Brian Bagnall [BAGNALL, 2002]. Basa su funcionamiento en utilizar clases RCXPort, InputStream y OutputStream. Cuenta con los siguientes métodos públicos:

- *public String getComPort( )* – Regresa una cadena representativo del nombre del puerto de comunicaciones usado por el RCXBean.
- *public void setComPort(String value) throws IOException* – Cambia el puerto de comunicaciones que será usado, por ejemplo COM1 ó USB.
- *public void sendInt(int v) throws IOException* – Envía un entero como cuatro *bytes*.
- *public void send(byte b) throws IOException* – Envía un solo *byte*.
- *public void send(byte[ ] b) throws IOException* – Envía un arreglo de *bytes*.
- *public byte receive( ) throws IOException* – Recibe un solo *byte*, este método espera hasta que un *byte* sea recibido.

- *public int receiveInt( ) throws IOException* – Recibe un entero, el cual se conforma de cuatro *bytes*.
- *public byte[ ] receive(int n) throws IOException* – Recibe un arreglo de *bytes*, n es la cantidad de *bytes* que va a esperar y también el tamaño del arreglo.
- *public synchronized void lock(Object o) throws IOException* – Crea un candado en el RCXBean. El RCXBean no puede ser bloqueado por otro objeto antes de que sea liberado por medio del método *free( )*. Si este método es llamado mientras el RCXBean ya está siendo bloqueado, entonces se lanzará una IOException. Es importante resaltar que este método no evita que otros *thread*s usen el RCXBean.
- *public synchronized void free(Object o)* – Hace este RCXBean disponible para otros *thread*s.
- *public void close( )* – Cierra este RCXBean.

4.1.7. RCXRemote

Se encuentra en el paquete josx.rcxcomm.RCXRemote. Esta es una clase que crea un RCXPort para establecer la comunicación. Tiene dos variables públicas que permiten un acceso remoto: DataInputStream in y DataOutputStream out. Cuenta con sólo dos métodos estáticos, *start( )* y *stop( )*; el método *start( )* inicializa el RCXPort y le da una funcionalidad de *listener* que siempre está a la espera del envío o recepción de datos [WEB21].

4.1.8. Tower

Después de experimentar con las clases anteriores se notó que cada una tenía alguna deficiencia para ser usada individualmente en el proyecto. Aunque el rango de alcance era muy corto en todos los casos, la clase RCXLNPAddressingPort tenía algo similar a lo que se buscaba, que era enviar instrucciones a un RCX en particular.

Al analizar con un poco de más profundidad esta clase se obtuvo la respuesta: la clase Tower. Es una interfaz de bajo nivel usada para realizar una conexión con la IR Tower. Sus métodos permiten enviar y recibir mensajes al RCX [WEB21]:

- *protected final native int open(String p)* – Abre la conexión con la IR Tower, recibiendo como parámetro una cadena representativa de puerto de comunicaciones USB.
- *protected final native int close( )* – Termina la comunicación establecida con la IR Tower.
- *protected final native int send(byte b[ ], int n)* – Método que envía un arreglo de *bytes* a través de la IR Tower al RCX. Recibe como parámetros el arreglo de *bytes* que va a ser enviado y el tamaño del arreglo.

Aunque esta clase incluye más métodos, estos tres son los que lograron la comunicación deseada, se obtuvo un alcance mucho más largo de la transmisión, y el envío de *bytes* permite enviar la información necesaria.

4.2. Protocolo de Comunicación Infrarroja

Como ya se había mencionado anteriormente, cada mensaje enviado por la IR Tower consta de tres partes:

0x55 0xff 0x00 D1 ~D1 D2 ~D2 C ~C

1) Encabezado: 0x55 0xff 0x00
2) Mensaje: D1 D2
3) Suma de Comprobación: C ~C, donde C = D1 + D2.

Ya que leJOS automáticamente inserta en cada mensaje el encabezado y la suma de comprobación, sólo nos queda modificar libremente D1 y D2.

Para aprovechar al máximo estos dos parámetros se decidió utilizar D1 como el número de identificación de cada robot. D2 será utilizado para indicar qué acción debe realizar el robot:

- ACTION_FORWARD = 1

  Avanza un segundo.

- ACTION_BACKWARD = 2

  Retrocede un segundo.

- ACTION_TURN_LEFT = 3

  Gira tomando como eje la llanta izquierda.

- ACTION_TURN_RIGHT = 4

  Gira tomando como eje la llanta derecha.

- ACTION_ROTATE_LEFT = 5

  Gira a la izquierda tomando como eje el centro del robot.

- ACTION_ROTATE_RIGHT = 6

  Gira a la derecha tomando como eje el centro del robot.

- ACTION_SHOOT = 7

  Toma el balón con las pinzas y realiza un giro aproximado de 270° antes de abrirlas y soltarlo, todo con la finalidad de darle el impulso necesario en dirección a la portería. Es importante aclarar que los robots tiran por su lado derecho.

- ACTION_OPEN_CLAMPS = 8

  Abre la pinzas.

- ACTION_CLOSE_CLAMPS = 9

Cierra las pinzas.

- ACTION_AUTONOMOUS_MODE = 10

  Devuelve el mando al comportamiento autónomo del robot. Este comando es llamado automáticamente después de realizar un tiro a la portería, pero también puede llamarse en cualquier otro momento.

- ACTION_STOP = 11

  Detiene los motores.

La recepción infrarroja de los robots está ejecutándose en todo momento en un *thread* separado del programa principal. Se diseñó de esta forma porque se vio la necesidad de que el entrenador quiera mandar instrucciones aisladas a los robots sin tener que esperar a que tomen el balón para poder enviarlas (ver Apéndices C y D para más detalles).

4.3. Comportamiento

Para este proyecto se diseñó un comportamiento basado en eventos, cada señal recibida del ambiente por medio de los sensores activaría una reacción. Para lograr esto se sacaría el máximo provecho de los *thread*s con los que cuenta leJOS.

El modelo de control de leJOS permite un encapsulamiento de cada comportamiento planeado, admitiendo una fácil reestructuración de los comportamientos al poder eliminarlos o agregar nuevos sin repercusiones negativas al resto del diseño.

El API de Behavior de leJOS, ubicada en el paquete josx.robotics, cuenta con la interfaz Behavior y la clase Arbitrator. La interfaz Behavior es la que nos permite implementar cada uno de los comportamientos que hemos definido, y la clase Arbitrator es la que regula qué comportamiento es el que está activo en un momento dado.

4.3.1. josx.robotics.Behavior

- *public boolean takeControl( )* – Regresa un valor *boolean* para indicar si el comportamiento debe tomar el control del robot o no.
- *public void action( )* – Representación de lo que el robot debe hacer cuando este comportamiento está activo. Una observación importante es que el contenido de este método no debe tener un ciclo infinito.
- *public void suppress( )* – Este método indica lo que el robot debe hacer cuando este comportamiento requiera ser detenido.

Esta interfaz debe ser implementada por cada una de las clases que sean diseñadas para ser comportamientos. Cada clase debe sobrescribir estos métodos, colocando en su interior la programación que sea conveniente en cada caso.

4.3.2. josx.robotics.Arbitrator

- *public Arbitrator(Behavior [ ] behaviors)* – El constructor de la clase crea un objeto que regula la activación de cada uno de los comportamientos. Las prioridades de los comportamientos se basa en el índice que ocupan dentro del arreglo de Behavior, mientras más alto sea el índice más alta será la prioridad, así que en este caso sí afecta directamente cómo se ordene el arreglo antes de ser enviado al constructor.
- *public void start( )* – Inicia el sistema de arbitraje de comportamientos. Aquí es donde el Arbitrator llama el método *action( )* del comportamiento que debe activarse. Cuando otro comportamiento cumple con la condición descrita en su método *takeControl( )*, el Arbitrator activa el método *supress( )* del comportamiento actual antes de llamar al *action( )* del nuevo comportamiento. Si dos comportamientos cumplen su condición de

activación al mismo tiempo, el Arbitrator elegirá el comportamiento que tenga la más alta prioridad para que tome el control del robot.

4.3.3. Arquitectura Subsumption

La arquitectura Subsumption es un paradigma de programación de robots creada en la década de 1980 por Rodney Brooks. Se basa en la idea de que varios comportamientos pueden estar corriendo simultáneamente. Los valores recibidos desde los sensores determinan cuál comportamiento es el que controlará al robot en un momento dado. Los comportamientos de mayor prioridad tomarán el control del robot, reemplazando los comportamientos de prioridades más bajas [KNUDSEN, 1999].

La figura 4.1 muestra los 4 niveles de comportamientos utilizados en esta investigación, utilizando el rectángulo verde para indicar lo que activa la condición necesaria para que se ejecute el comportamiento, el blanco muestra la acción que se ejecutará, el símbolo S dentro del círculo rojo permite visualizar cómo un comportamiento substituye a otro de menor prioridad, y finalmente el azul muestra el recurso por el cual los comportamientos compiten.

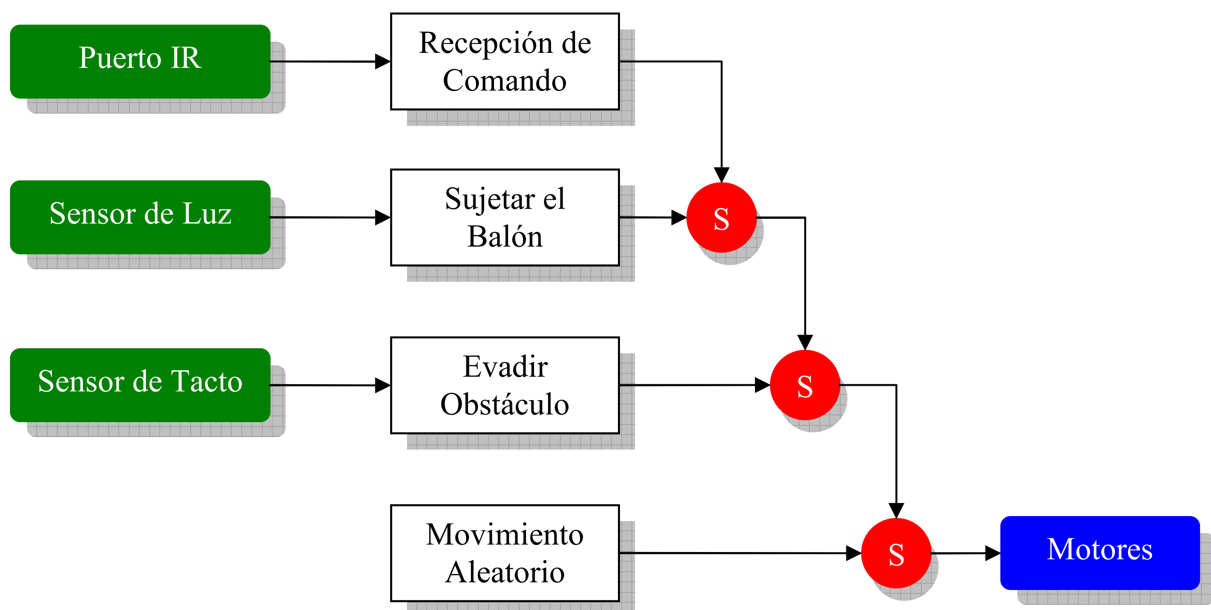


Figura 4.1 Arquitectura Subsumption usada [Elaboración Propia].

4.3.4. Puerto IR

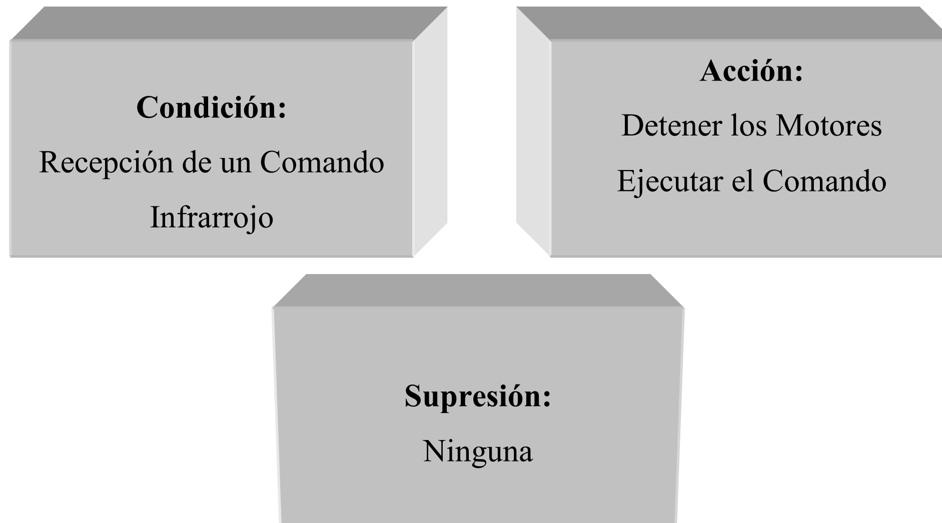


Figura 4.2 Comportamiento del puerto infrarrojo del RCX [Elaboración Propia].

Es el comportamiento con más alta prioridad, no puede ser interrumpido hasta que el entrenador lo decida.

4.3.5. Sensor de Luz

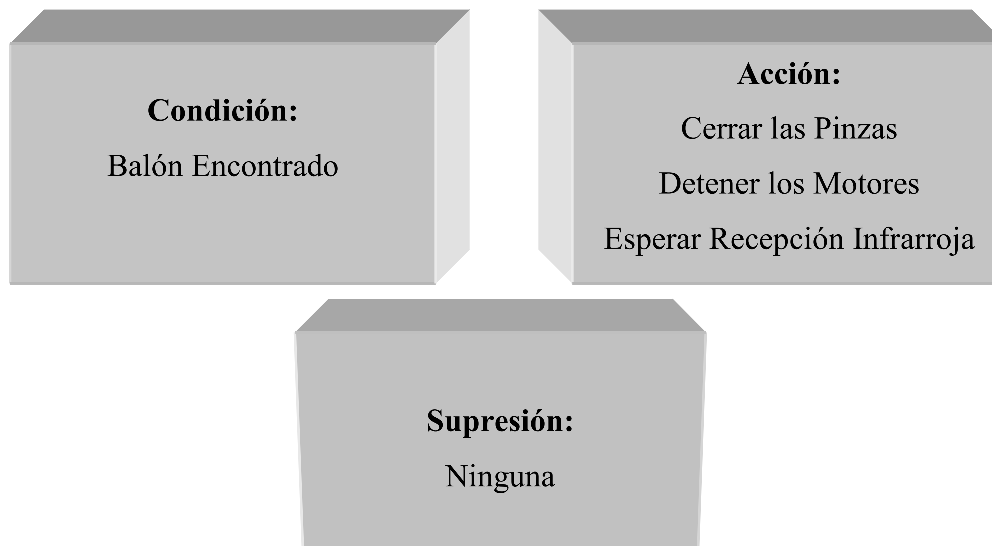


Figura 4.3 Comportamiento del sensor de luz [Elaboración Propia].

Es el comportamiento autónomo con la prioridad más alta, puede suprimir cualquier otro que esté activo, excepto la transmisión infrarroja.

4.3.6. Sensor de Tacto del Puerto 1

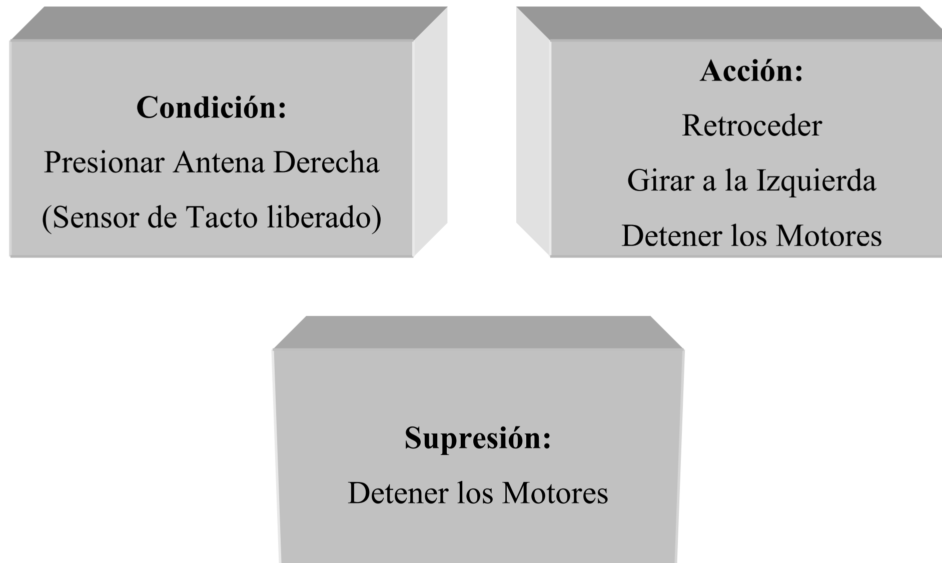


Figura 4.4 Comportamiento del sensor de tacto conectado al puerto de entrada 1 [Elaboración Propia].

Los dos sensores de tacto tienen la misma prioridad, por lo que uno puedo suprimir al otro o al movimiento inicial del robot.

4.3.7. Sensor de Tacto del Puerto 3

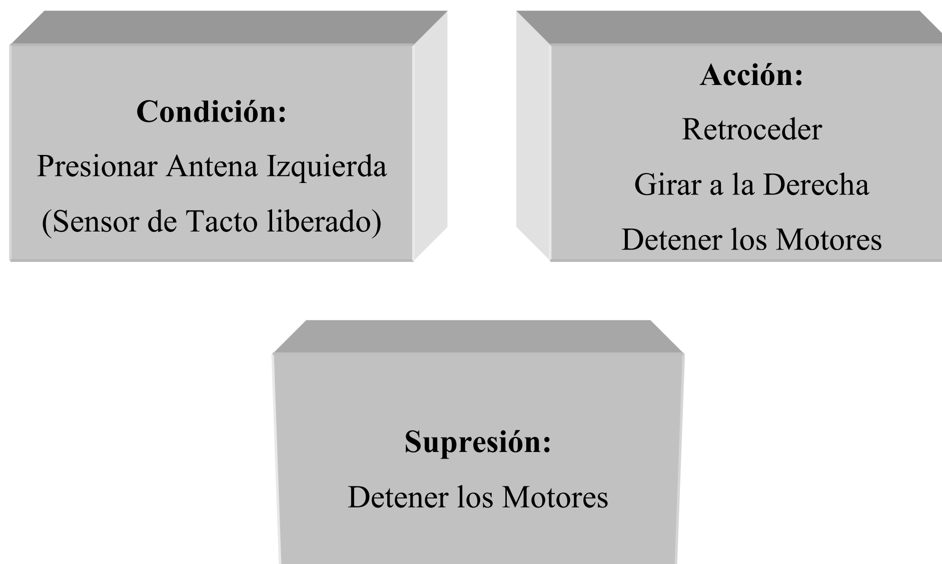


Figura 4.5 Comportamiento del sensor de tacto conectado al puerto de entrada 1 [Elaboración Propia].

4.3.8. Movimiento Aleatorio

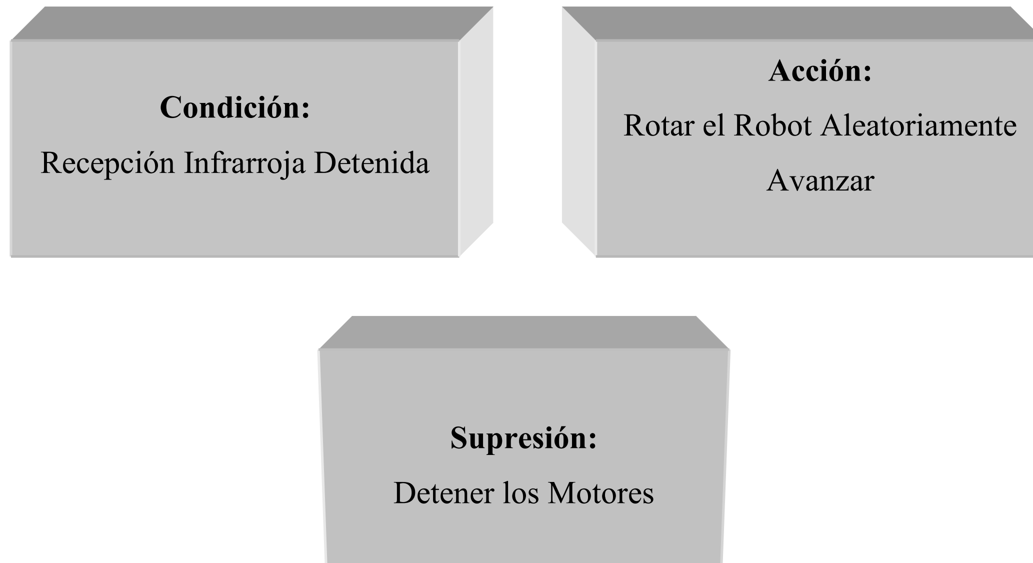


Figura 4.6 Comportamiento del movimiento aleatorio inicial [Elaboración Propia].

Se comprobó que al disminuir los patrones de comportamiento repetitivos se reducía la posibilidad de que el robot quedara atrapado en un ciclo infinito, como el que pudiera darse al chocar contra una esquina de la cancha.

4.4. Interfaz Gráfica

4.4.1. Los Patrones de Diseño MVC y Observador

Los llamados patrones de diseño son sólo formas convenientes que facilitan la reutilización de código orientado a objetos entre diferentes proyectos o programadores, son soluciones recurrentes a problemas de diseño que se ven una y otra vez [COOPER, 1998].

Un patrón está conformado de cuatro partes principales [GAMMA, 1995]:

- Nombre: Describe al diseño en muy pocas palabras, ayuda a identificarlo entre los demás patrones y puede dar una idea de su comportamiento.

- Problema: Contiene las reglas de cuándo se debe aplicar el patrón.

- Solución: No es referente a un diseño en particular, ya puede ser implementada como una plantilla a cualquier problema que cumpla con las reglas de aplicación del patrón.

- Consecuencias: Relacionadas al impacto directo que tuvo el patrón sobre las características del sistema, como lo son su flexibilidad, portabilidad y extensión.

El patrón Modelo-Vista-Controlador es mejor conocido como Model-View-Controller ó MVC. Cada una de las palabras de su nombre indica uno de los componentes que lo conforman [COOPER, 1998]:

- Modelo: Tienen el único acceso directo permitido a los datos y realiza las operaciones necesarias sobre ellos.
- Vista: Componente visual de estos datos, lo que el usuario ve. Cada sistema puede tener tantas vistas diferentes como necesite.
- Controlador: Sirve de intermediario entre el modelo y la vista.

El patrón observador define una dependencia uno a muchos entre objetos de tal forma que cuando uno cambia, todos sus dependientes son notificados y actualizados automáticamente [FREEMAN, 2004].

Todo el sistema está basado en estos dos patrones, ya que es más eficiente a la hora de agregar nuevas vistas. Cada una de las modalidades del sistema es una vista diferente que trabaja con el mismo modelo. Si se requieren nuevas funcionalidades en el sistema, estas son agregadas en el modelo sin tener que modificar cada una de las vistas.

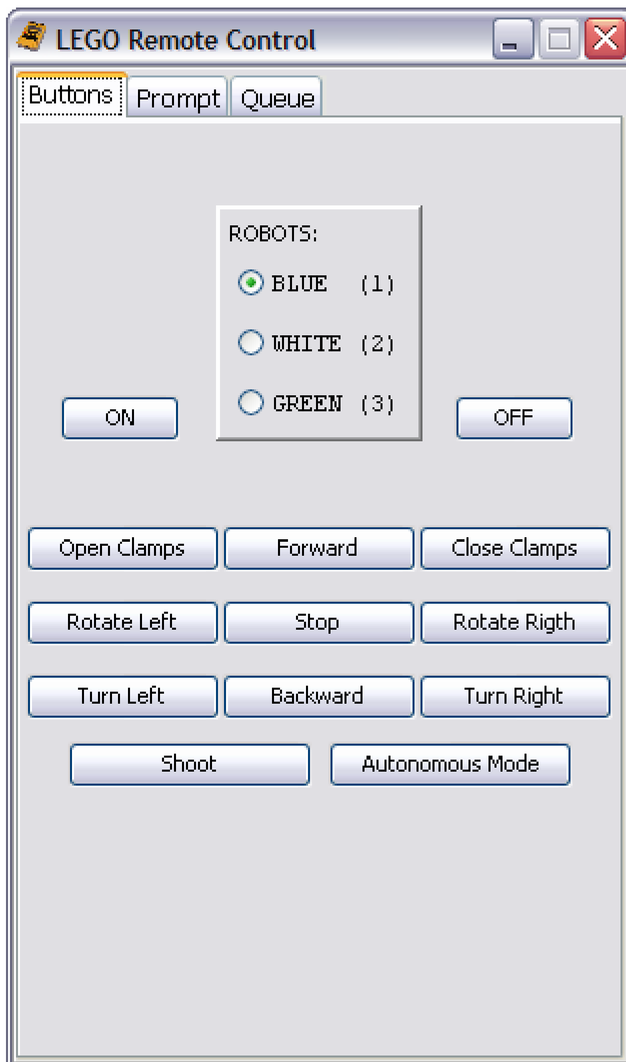
El modelo es el encargado de realizar la conexión con la IR Tower y de manejar el uso de la cola de envío de mensajes. La interfaz gráfica primaria tiene la función servir como controlador y hacer las llamadas a cada método que se necesite.

Cuando se realiza un cambio en una de las vistas, las demás son inmediatamente notificadas de dicho cambio gracias a que cada una de estas es un observador y el modelo está siempre siendo observado. En este sistema dicha propiedad se aplicó al envío de mensajes, ya que si un comando es enviado desde una de las vistas, las demás también lo mostrarán.

4.4.1.1. Modalidad con Botones

Esta área de la interfaz gráfica es la más sencilla de utilizar. Está compuesta de dos secciones:

- Selección de robots
- Botones de ejecución inmediata de comandos.



La selección de robots le dice al sistema a cuál de los tres debe enviar el comando que se elige en la sección de botones. Cada comando se inserta a la cola de salida, donde espera su turno para ser enviado.

Los botones ON y OFF les dicen a los 3 robots que deben iniciar su comportamiento autónomo o detenerse respectivamente, esto es especialmente útil a la hora de iniciar y finalizar una prueba.

Imagen 4.1 Interfaz gráfica de la modalidad de botones [Elaboración Propia].

4.4.1.2. Modalidad de Línea de Comandos

El uso de la línea de comandos es una funcionalidad básica del programa, ya que permite el envío de mensajes al robot elegido sin tener que presionar botones. La característica más interesante es que muestra la lista de los comandos enviados durante todo el tiempo que el sistema estuvo activo. Su uso es sencillo, ya que sólo debe escribirse el comando para la acción deseada, seguido del número de identificación del robot al que le será enviado.

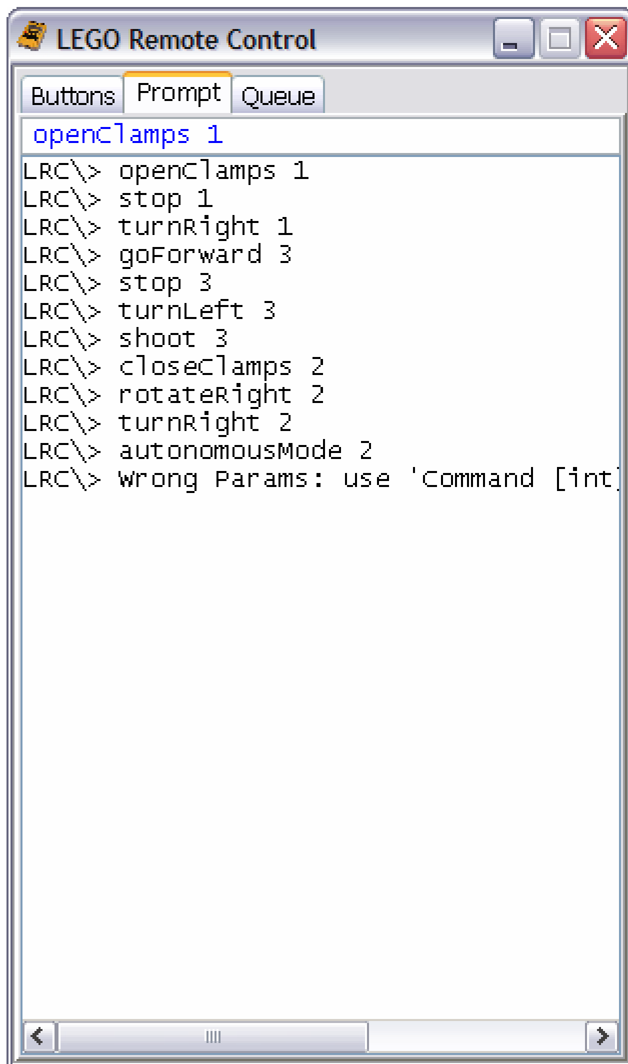


Imagen 4.2 Interfaz gráfica de la modalidad de línea de comandos [Elaboración Propia].

4.4.1.3. Modalidad de Manejo de Cola de Envío de Mensajes

Esta es la modalidad más avanzada con la que cuenta el programa. Está compuesta de tres secciones:

- Cola de Envío de Mensajes

  Tiene una capacidad de almacenamiento de 10 mensajes, los cuales se van enviando 1 cada segundo. La interfaz presenta una animación continua de la cola, a menos que la opción de Pausa sea presionada. Es importante mencionar que la opción de pausar es respetada aunque se cambie de vista.

  Los comandos pueden ser insertados en los lugares vacíos de la cola, o también pueden colocarse en el lugar que otro comando ocupaba. Al suceder esto los comandos ascienden un lugar en la cola de espera, debido a esta propiedad un comando se eliminará si está ubicado en la décima posición de la cola cuando otro comando se inserte.

- Selección del Robot

  Sólo permite seleccionar un robot a la vez para poder enviarle un comando. El robot 1 aparece seleccionado por defecto, y muestra una lista de comandos en color azul. Cuando se eligen los robots 2 y 3, el color de los comandos cambiará a blanco y verde respectivamente.

  Esta discriminación por colores se mantendrá aún dentro de la cola de envío de mensajes para poder diferenciarlos.

- Comandos Disponibles

Cada comando tiene la propiedad de que puede ser arrastrado hacia la cola de espera, e insertado en el lugar que se quiera. Para esto se utilizan eventos de arrastrar y soltar (*drag & drop*) cada comando, los cuales no pueden ser soltados en lugares no permitidos para evitar errores.

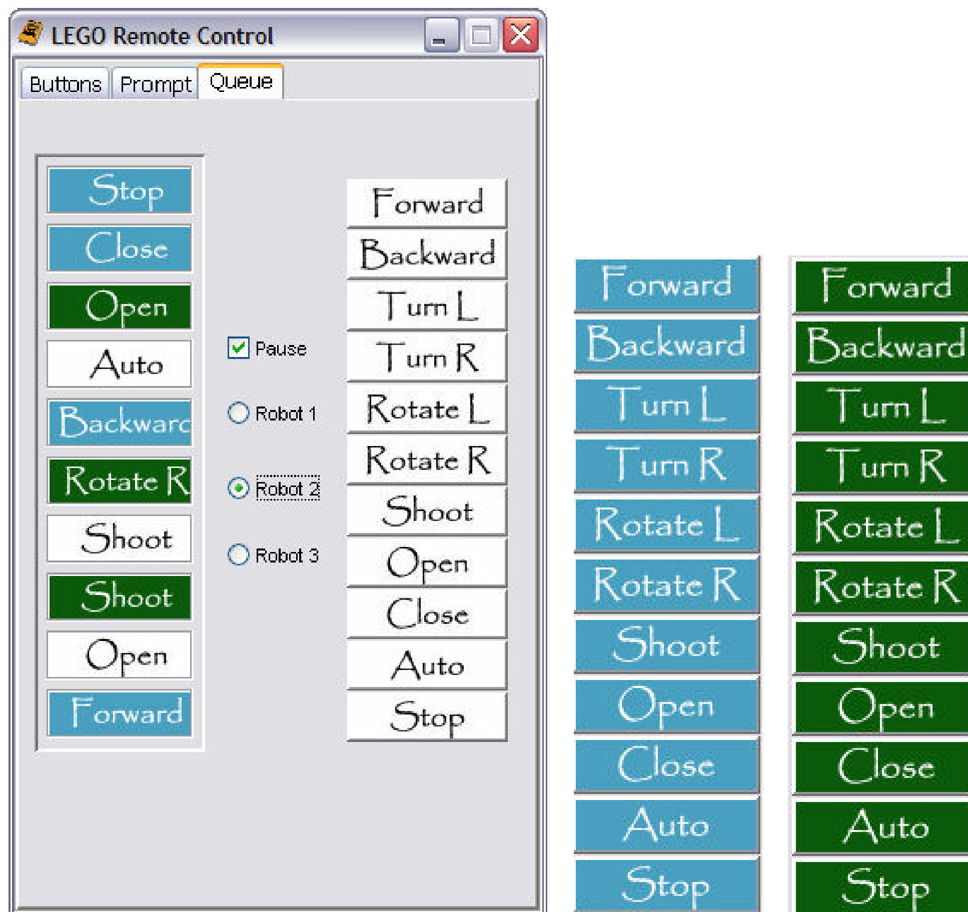


Imagen 4.3 Interfaz gráfica de la modalidad de cola de envío de mensajes [Elaboración Propia].

4.4.2. LEGO Cam

La segunda zona de la interfaz gráfica está dedicada a obtener una imagen continua de la LEGO Cam, esto con la finalidad de observar la totalidad de la cancha y a todos los robots al mismo tiempo. La imagen se ajustó a su máxima capacidad de captura de 640x480 píxeles. Y está posicionada a una altura aproximada de 2 metros para poder abarcar la cancha.

Como puede observarse en la imagen, cada robot tiene un color distintivo que le es útil al entrenador para saber a qué robot le está enviando órdenes.

Imagen 4.4 Interfaz gráfica de la conexión con la LEGO Cam [Elaboración Propia].

4.5. Consideraciones en la Construcción de los Robots

Un problema muy grande que existe al construir robots móviles es hacer que avancen derecho y esto se debe a tres motivos principales [WEB22]:

1) Cuando se mueven hacia adelante los motores en verdad están funcionando en direcciones opuestas al estar colocados uno frente a otro. Mientras el robot se intenta mover hacia delante uno de los motores va en dirección de las manecillas del reloj y el otro va en contra. Se observa que uno de los motores se mueve con más eficiencia que el otro, lo que dará como resultado una trayectoria curva.
2) Las llantas pueden tener pequeñas variaciones en su tamaño. Aunque los motores se movieran a la misma velocidad, una variación en el tamaño, por más pequeña que sea, causará una desviación en la ruta.
3) La superficie en la que el robot se mueve puede no ser completamente plana. Si una llanta pasa sobre una deformación, provocará un pequeño giro que cambiará la trayectoria inicial.

Existen diversas soluciones para este problema, pero no se usaron por diferentes razones:

- Un solo motor mueve ambas llantas por medio del diferencial
    - Las llantas delanteras deben ser movidas por un sistema de dirección, el cual ocupa otro motor. El colocar las llantas delanteras exactamente derechas y mantenerlas así durante el recorrido se convierte en un inconveniente. Esto, a su vez, se soluciona utilizando un sensor de rotación en lugar de un motor, pero el *kit* de LEGO MINDSTORMS no lo incluye.
- Utilizar un sistema de doble diferencial, uno conectado a cada motor.
    - El *kit* de LEGO MINDSTORMS sólo incluye un diferencial.

- Usar una fuente luminosa y un sensor de luz como sistema de orientación.
    - El robot ya utiliza los 3 puertos de entrada con el que cuenta el RCX: dos para los sensores de tacto y uno para el sensor de luz que debe encontrar la pelota.
    - El *kit* de LEGO MINDSTORMS sólo incluye un sensor de luz.
    - No puede reutilizarse el sensor de luz cambiando su función de encontrar la pelota una vez que ya se tenga presionada por las pinzas. El mejor posicionamiento encontrado para el sensor fue en un lugar muy bajo, y queda completamente cubierto por la pelota al ser hallada.

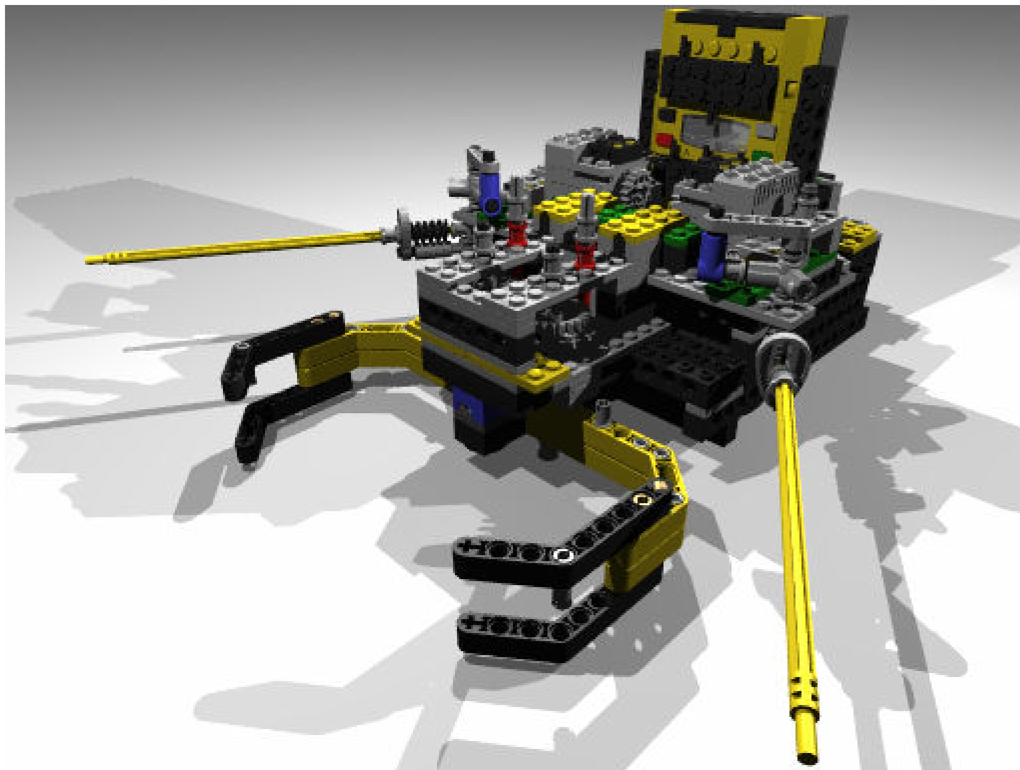


Imagen 4.5 Diseño final del robot [Elaboración Propia].

Para más información del proceso de construcción, consultar el Apéndice A.

4.6. Equipo Adicional

4.6.1. Medidas de la Cancha

Imagen 4.6 Medidas de la cancha [Elaboración Propia].

4.6.2. Medidas de las Porterías

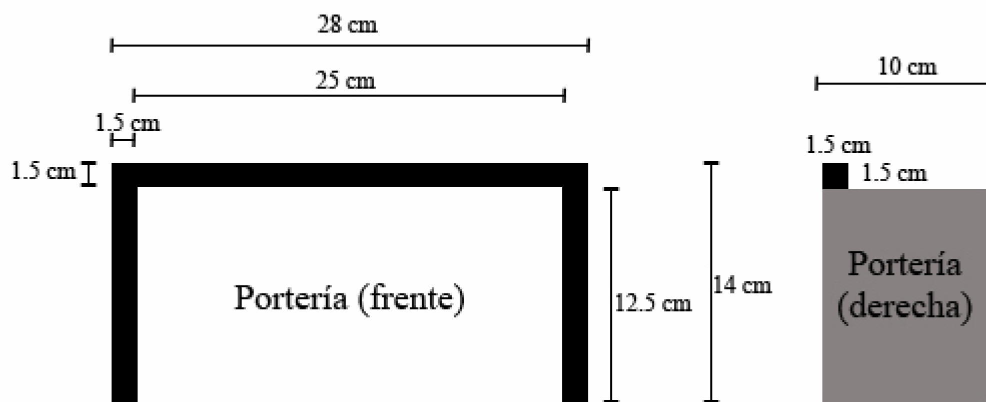


Imagen 4.7 Medidas de la portería [Elaboración Propia].

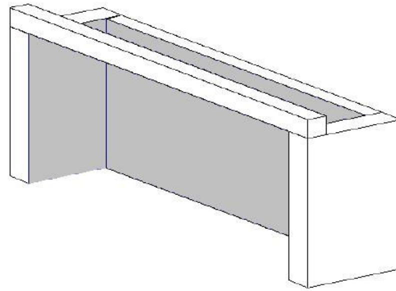
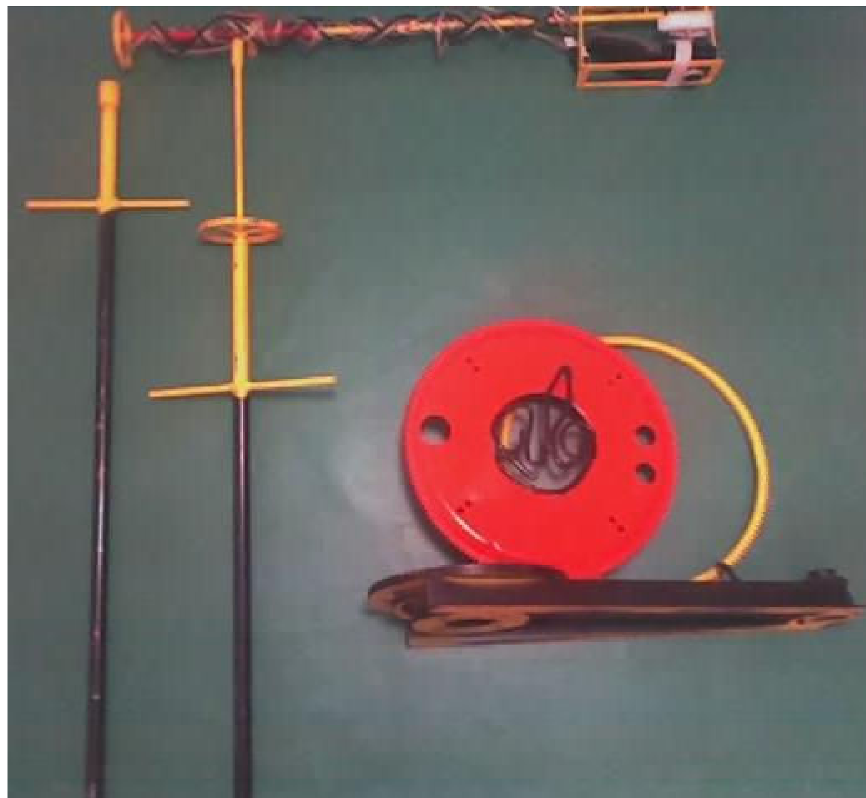


Imagen 4.8 Modelo tridimensional de la portería [Elaboración Propia].

Las paredes que rodean la cancha tendrán 12.5 cm. de altura, al igual que los postes de las porterías.

4.6.3. Medidas del Poste

Aquí se montan la LEGO Cam y la IR Tower. La altura es ajustable, pero debe alcanzar los 2 m. al estar ensamblado, todo con la finalidad de lograr una imagen completa de la cancha desde la LEGO Cam. El contrapeso debe ser suficiente para que el poste no se caiga.



4.7. Pruebas

Se realizaron dos tipos de pruebas diferentes, tanto para encontrar fallas en el sistema como para comprobar la exactitud de los robots.

4.7.1. Encontrar el Balón

Se colocaron los 3 robots en la cancha y se midió el tiempo que transcurrió antes de que uno de ellos encontrara el balón.

No. de Prueba	Tiempo (min.:seg.)	Observaciones
1	01:56.96	Se colocó el balón en su posición original.
2	00:48.64	
3	01:21.53	
4	02:49.43	Se colocó el balón en su posición original.
5	00:40.50	
6	Detenida al superar 5 min.	
7	Detenida al superar 5 min.	
8	1:15.64	
9	Detenida al superar 5 min.	
10	3:25.32	Se colocó el balón en su posición original.

Tabla 4.1 Tiempos que se tardan los robots en encontrar el balón [Elaboración Propia].

Resultados:

- El balón frecuentemente es arrojado a ubicaciones en las que es difícil de encontrar, como lo son las esquinas, o inclusive dentro de las porterías, por lo que se debe inicializar su posición.

- Varias veces durante la prueba los robots tuvieron que recibir instrucciones para que pudieran liberar a otro robot, o inclusive al mismo balón atrapado entre las pinzas sin ser visto.
- Los robots, al tener una vista muy limitada, en ocasiones tienen el balón dentro de sus pinzas, pero no logran verlo.
- El balón debe ser encontrado de frente para que el robot pueda verlo.
- El robot ve el balón a una distancia aproximada de 2 cm.
- Al unir las dos observaciones anteriores se da el caso en que el robot golpea el balón de frente al no poder frenar a tiempo.
- Al sólo distinguir intensidades de luz y no colores, si dos robots se encuentran de frente existe la posibilidad de que uno confunda al otro con el balón.
- Las pruebas que se alargaban demasiado fueron suspendidas.
- Se recomienda que cuando se observe por la cámara que un robot tiene el balón en sus pinzas pero no lo ha visto, se le dé la instrucción de cerrar las pinzas y no esperar más tiempo a que lo encuentre.

4.7.2. Serie de Tiros Penales

Se realizaron dos rondas, una en cada portería para comparar ambos lados de la cancha.

Portería 1	
No. de Tiro	**Anotación**
1	Sí
2	Sí
3	No
4	No
5	Sí

Portería 1	
No. de Tiro	**Anotación**
6	Sí
7	No
8	No
9	Sí
10	No

Tabla 4.2 Anotaciones en dos series de tiros penales [Elaboración Propia].

Portería 2	
No. de Tiro	Anotación
1	Sí
2	Sí
3	Sí
4	No
5	No

Portería 2	
No. de Tiro	Anotación
6	Sí
7	Sí
8	No
9	Sí
10	No

Tabla 4.3 Anotaciones en dos series de tiros penales [Elaboración Propia].

Resultados:

- Para medir el porcentaje de efectividad de los robots al tirar, se decidió realizar una serie de penales. De esta manera no existen obstáculos entre el robot y la portería, permitiendo al balón entrar directamente.
- Por las medidas de la cancha, y por el espacio libre que necesita el robot para tirar correctamente, al balón y el robot son colocados un poco antes de la media cancha para realizar el tiro penal.
- Debido a que los robots son diestros, en el momento de colocarlo para realizar el tiro penal, es importante colocarlo un poco a la derecha.
- Recordando que la batería afecta directamente el movimiento de los robots, y que su voltaje va disminuyendo gradualmente con el uso, la puntería de los tiros baja y se hace necesaria una nueva calibración.
- La fricción causada por arrastrar el balón al realizar un tiro también es un factor, al igual que la cancha, ya que cualquier obstáculo, como el polvo o un desperfecto del piso, puede causar un desvío en la trayectoria del balón.