universidad del país vasco — euskal herriko unibertsitatea

Facultad de Informática — **Informatika Fakultatea**

TITULACIÓN: Ingeniería Técnica en Informática de Sistemas

Interaccion corporal con el ordenador para niños con restricciones motoras

Alumno/a: D./Dña. Jose Antonio Fontan Inza

Director/a: D./Dña. Julio Abascal González

Proyecto Fin de Carrera, julio de 2012

Índice de contenido

RESUMEN

Para los niños el juego es la manera más natural de aprender. Mediante el juego los niños interactúan con su entorno. Cuando se trata de niños con graves restricciones motoras esta interacción se ve limitada. Es por esta razón por la que intentamos poner los medios existentes a su disposición.

Este proyecto muestra una interfaz persona-computador para manejar un robot Lego Mindstorm RCX en un entorno virtual proyectado sobre una superficie. En todo momento el sistema es capaz de determinar la posición del robot mediante un sensor Microsoft Kinect. Pretende ser el primer paso en la creación de una interfaz persona-computador para niños con restricciones motoras que ayude en su rehabilitación.

Palabras clave: Kinect, Lego, rehabilitación, restricciones, niños.

1.INTRODUCCIÓN

-Resumen

Se presenta un sistema de interacción persona-computador orientado a niños con restricciones motoras. Mediante el manejo de un robot Lego Mindstorms se permite la interacción con un escenario virtual proyectado sobre una superficie.

El sistema incluye un sensor Microsoft Kinect para capturar el entorno y obtener la situación del robot dentro de éste.

-Motivación

El juego se ha mostrado siempre como una parte importante en el proceso de aprendizaje de los niños. La manipulación de objetos proporciona una manera de desarrollar sus habilidades, ya sean cognitivas, motoras, sociales, etc. Este hecho cobra mayor importancia cuando se trata de niños con graves restricciones motoras, ya que su capacidad de interacción con el entorno se ve limitada.

Este proyecto pretende presentar una interfaz persona-computador que posibilite la manipulación de un robot Lego Mindstorms dentro de un entorno virtual con el que puede haber la posibilidad de interactuar.

En este proyecto se tocan diferentes áreas de la computación:

-La visión por computador, usada en la obtención de la posición del robot.

-La robótica, usada en el manejo del propio robot.

-El desarrollo de interfaces persona-computador para personas con restricciones motoras.

Este proyecto presenta dos objetivos principales:

Por un lado está el objetivo técnico que representa la realización de una aplicación que permita el manejo de un robot mediante el computador y que obtenga realimentación de su posición. Por otro lado, está el objetivo pedagógico/rehabilitador de la aplicación. Diversos estudios han demostrado que la utilización de robots como herramientas de ayuda en el proceso del aprendizaje es muy positiva [1].

2.OBJETIVOS

2.1.Técnicos

En este desarrollo definimos tres objetivos técnicos:

-Integración de Microsoft Kinect en una aplicación Windows y detección de objetos mediante ésta.

-Controlar mediante el ordenador un robot Lego Mindstorm RCX.

-Integrar los puntos anteriores y añadir una imagen virtual proyectada sobre la que se desplazará el robot.

El primero de los objetivos técnicos nos llevará a la elección del marco y herramientas de trabajo además de la elección de una biblioteca gráfica adecuada para ello.

El segundo objetivo nos llevará a la elección de la biblioteca de comunicaciones adecuada para manejar los robots Lego Mindstorm RCX.

Por último, el tercero nos planteará a la idoneidad o no del método utilizado tanto en la detección de objetos como en el manejo del robot.

2.2.Aplicación

El objetivo de la aplicación es el desarrollo de un interface persona-máquina, que nos permita manejar un robot Lego Mindstorm RCX y, obtener, en todo momento, realimentación de su posición en el entorno a través de la cámara del sensor Microsoft Kinect. Esta aplicación está pensada para ayudar a la rehabilitación de niños con graves restricciones motoras.

3.METODOLOGÍA DE TRABAJO

El desarrollo se ha dividido en tres objetivos o fases, que serán desarrolladas progresivamente, siendo las dos primeras independientes y, posteriormente, integradas en la tercera fase.

La metodología de trabajo elegida es la siguiente:

Tomando el objetivo de cada fase, se realizará el desarrollo y se probará. En base a esta prueba, se evaluarán los resultados obtenidos. Si estos son satisfactorios se procederá a pasar a la siguiente fase, en caso contrario, se volverá al desarrollo, buscando uno que nos permita obtener resultados satisfactorios. En caso de no encontrarlo, se procederá a analizar las razones de no hacerlo.

4.HERRAMIENTAS DE TRABAJO

Para el desarrollo de la aplicación se ha elegido como entorno de desarrollo Microsoft Visual Studio 2010, en gran parte debido a que el sensor Kinect, al ser un producto desarrollado por la propia Microsoft, su integración resulta más sencilla.

Como lenguaje de programación se ha elegido C#. Este es un lenguaje orientado a objetos desarrollado y estandarizado por Microsoft dentro de su plataforma .Net, aunque no se limita a generar programas para esa plataforma, ya que existen compiladores estándar que pueden utilizarse en otras plataformas.

La elección del lenguaje de programación (en detrimento de C++) condiciona la elección de las herramientas disponibles para la consecución de los objetivos. Ya que supone una limitación en cuanto a la elección de librerías disponibles tanto a nivel de gráficos, como a nivel de manejo de los Lego Mindstorm. Pero, en contrapunto, está la sencillez de la utilización de la memoria dinámica, en menor medida, el ser un lenguaje más moderno y, como razón principal la que pasamos a describir a continuación.

La aplicación se basa, principalmente, en una repetición del bucle:

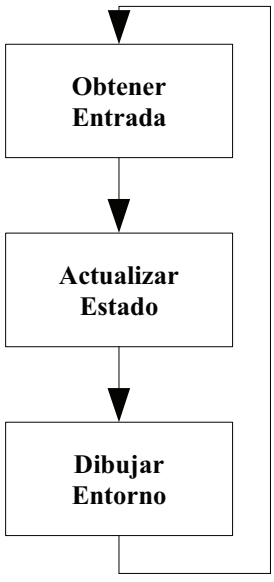


Figura 1: Esquema del bucle principal.

Este tipo de bucles está muy relacionado con el entorno de los videojuegos, por ello se ha elegido el marco de trabajo Microsoft XNA Framework para el desarrollo de la aplicación, ya que nos facilita la creación de este tipo de aplicaciones. No obstante, este marco de trabajo será integrado en una aplicación Windows Form estándar, para un manejo más sencillo.

La librería gráfica elegida es OpenCV. Es una librería muy extensa, que nos proporciona multitud de funciones para el manejo y manipulación de imágenes, visión artificial, etc. El hecho de que en origen esté programada para su uso en C++, nos obliga a utilizar un *wrapper* (unas dll envoltorio que nos permitirán utilizar la funcionalidad de las dll originales) para poder utilizarlo con C#. No son muchas las opciones disponibles, siendo las más interesantes, debido sobre todo a la opinión de los usuarios, EmguCV y OpenCVSharp. Debido a la sencillez en su puesta en marcha y a la semejanza con la original, hemos elegido esta última.

Por último, queda decidir la librería a utilizar para el manejo de los robots Lego Mindstorm RCX. Dado que Lego dejó de dar soporte a sus robots RCX hace algunos años, la oferta de librerías en C# que nos permitan al manejo de este tipo de robots es muy limitada. En este caso, la librería elegida es Aforge.Net que, además del manejo de robots Lego Mindstorm RCX, nos permite también el manejo de robots Lego Mindstorm NXT entre otros. Esto será útil en ampliaciones futuras al poder añadir otros tipos de robots sin dificultad. Aforge.Net proporciona otras librerías con diversos usos: tratamiento de imagen, detección de movimientos, algoritmos matemáticos, etc. Pero en nuestro caso sólamente utilizaremos la librería que concierne al manejo de robots.

5.DESCRIPCIÓN DEL SISTEMA

5.1.Primera Fase

En esta primera fase, los objetivos son los siguientes:

-Integrar XNA Framework en una aplicación Windows Forms estándar.
-Integrar la captura de imágenes del sensor Microsoft Kinect en la aplicación.
-Encontrar un método de detección de objetos adecuado a nuestros propósitos.

5.1.1.Hardware – Sensor Microsoft Kinect

-HISTORIA

El sensor Microsoft Kinect es un periférico originalmente creado para la consola de videojuegos de Microsoft, Xbox 360 y comercializado en noviembre de 2010 [2].

Fue presentado por primera vez en Junio de 2009 en la E3 (*Electronic Entertainment Expo*) bajo el nombre de "Proyecto Natal" . Posteriormente, antes de su muestra oficial en la E3 de 2010 Microsoft reveló que su nombre definitivo sería Kinect [4].

Debido al éxito del producto y a la aparición de SDK's (*Software Development Kit*, Kit de Desarrollo de *Software*) no oficiales, Microsoft liberó una versión beta de su propio Kinect SDK para que se pudiera desarrollar *software* para el dispositivo en Junio de 2011.

Posteriormente, en Marzo de 2012, se ha comercializado una versión exclusiva para el sistema operativo Microsoft Windows que mejora algunas de las limitaciones del sensor original. La distancia de detección ha disminuido (tan sólo 40 cm, en lugar de 1.2

m), se posibilita la detección de gestos faciales, se ha mejorado el reconocimiento de voz, etc. pero esta versión sólamente está disponible a nivel empresarial/institucional.

***Figura 2: Sensor Microsoft Kinect*. Fuente: www.madboxpc.com**

-CARACTERÍSTICAS TÉCNICAS

Las características técnicas del sensor son las siguientes [3]:

-una cámara RGB con una resolución máxima de 640x480 *pixels* a 30 fotogramas/segundo.

-un sensor de profundidad que nos proporciona una "imagen" de profundidad de resolución máxima de 640x480 *pixels* a 30 fotogramas/segundo.

-un emisor de IR para poder generar la "imagen" de profundidad

-un micrófono multiarray (4 micrófonos en total, 3 en el lado izquierdo y uno a la derecha, todos ellos situados en la parte inferior del sensor) que es capaz de separar las voces que hay delante del resto de sonidos del entorno para chatear o utilizar comandos de voz [5], captura sonido con 16 *bit* de profundidad y una frecuencia de muestreo de 16KHz.

-un motor situado entre la base y el sensor que permite el movimiento vertical de éste un total de 27º.

-rango de detección de los sensores de profundidad 1.2-3.5m.

-un sistema de seguimiento de esqueleto, capaz de seguir a 6 personas, incluyendo dos jugadores activos.

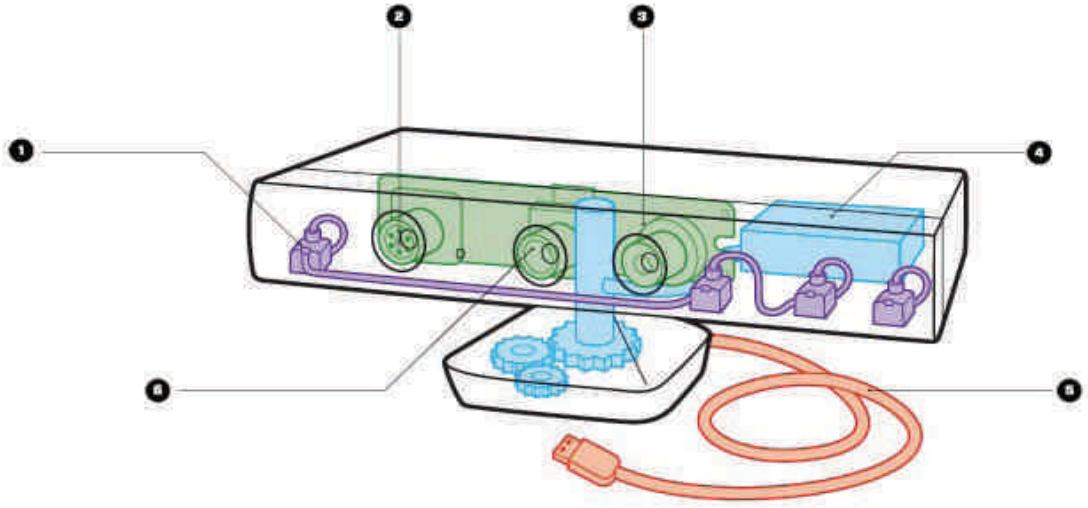


Figura 3: Esquema del sensor Microsoft Kinect [6]
Ilustración*: Kate Francis/Brown Bird Design*
1.Array de Micrófonos: Apuntan a la fuente del sonido mientras filtran el ruido de fondo
2.Emisor IR: Proyecta un patrón de luz IR en la estancia. Cuando la luz choca con una superficie, distorsiona el patrón, distorsión que es recibida por la cámara de profundidad
3.Cámara de Profundidad: Analiza los patrones de luz IR para construir un mapa 3-D de la estancia y de los objetos y personas allí situados
4.Motor de Inclinación: Ajusta automáticamente la orientación del sensor basándose en el objeto que está enfrente. Si el usuario es alto se inclinará hacia arriba, mientras que si es bajo lo hará hacia abajo
5.Cable USB: Transmite datos sin encriptar hacia la XBOX360 lo que hace que sea relativamente fácil utilizar el sensor Kinect con otros dispositivos
6.Cámara RGB: Captura imágenes de video. Esta información la usa el sensor Kinect para obtener detalles de los objetos o personas situados en la estancia

-SENSOR DE PROFUNDIDAD / EMISOR IR

Este es el corazón del sensor y la clave de su éxito. La configuración óptica del Kinect es lo que le permite rastrear movimientos en tiempo real. Fruto de un trabajo de investigación de más de 15 años, nos permite funciones que hasta hace poco tiempo sólamente estaban disponibles en dispositivos mucho más caros.

Básicamente cuenta con dos componentes: un proyector y una cámara VGA de InfraRojos. El primero de ellos proyecta un haz láser a través de todo el campo de juego, mientras la cámara recoge los rayos reflejados para poder separar los objetos que detecte en lo que se llama "campo de profundidad". Esencialmente son los *pixels* que el Kinect recibe como ruido IR designados con un diferente color en función de la distancia que los separa del sistema. De esta manera los objetos o personas más

cercanos aparecerán en colores brillantes (rojo, verde...) mientras que los objetos más lejanos lo harán en tono gris.

El *software* toma esta imagen y, pasándola por varios filtros, determina qué parte pertenece a una persona o no. El sistema sigue una guía básica de reglas para la distinción de las personas, del tipo "una persona tiene 2 brazos y dos piernas" o "una persona mide desde X cm hasta X cm". De esta manera se desechan los objetos que no cumplan las reglas establecidas.

Una vez ordenados los datos, se convierten las partes del cuerpo identificadas en un esqueleto con juntas móviles. Kinect tiene precargadas unas 200 poses comunes, por lo que puede rellenar los espacios que aparezcan si realizamos un movimiento que no permita a la cámara ver la totalidad de nuestro esqueleto. Uno de los problemas del sistema es que los dedos no se mapean individualmente lo que limita la aplicación del sistema en el ámbito de los videojuegos.

Todas estas operaciones son realizadas 30 veces por segundo, lo que nos garantiza una gran suavidad de funcionamiento [7].

5.1.2.Software

En esta primera fase debemos:

-integrar Microsoft XNA Framework en una aplicación Windows estándar
-integrar Microsoft Kinect en esa misma aplicación
-obtener un método de detección de objetos apropiado

5.1.2.1.Integrar Microsoft XNA Framework

Microsoft XNA (***X****NA's* ***N****ot* ***A****cronymed*, XNA no es un acrónimo) es un marco de trabajo desarrollado por Microsoft para facilitar el desarrollo de videojuegos. Una de sus características es eliminar o minimizar el código repetitivo (gestión del bucle del juego, etc). Se presentó en Marzo de 2004 en la *Game Developers Conference* en San José, California. Su última versión es la 4.0, liberada en Septiembre de 2010.

Se basa en la implementación de .NET Framework 2.0 en el caso del desarrollo de juegos para Windows, incluye un gran número de librerías de clases, siendo uno de sus objetivos la reutilización de código sea cual sea la plataforma destino.

Para la ejecución de este tipo de programas es necesario el *runtime*, específico para cada sistema, que nos permitirá poder ejecutar el *software* en cualquiera de los sistemas destino sin modificaciones. En nuestro caso, esto no es posible, ya que utilizaremos elementos propios de los sistemas Microsoft Windows [8].

Microsoft nos proporciona el entorno de desarrollo XNA Game Studio (utilizaremos la versión 4.0), que se integrará en Microsoft Visual Studio, permitiéndonos utilizar el marco de trabajo con facilidad.

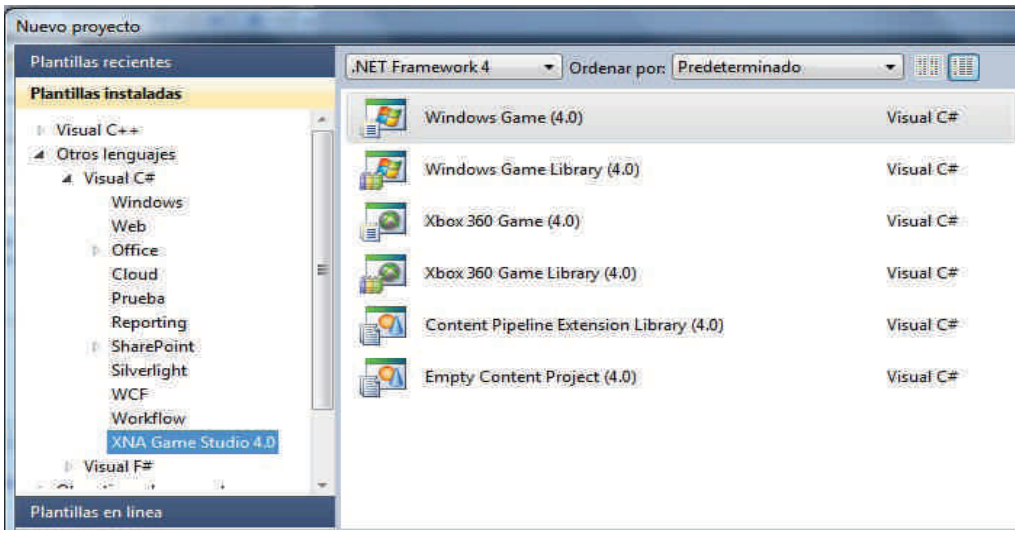


Figura 4: Creación de un proyecto con XNA Game Studio 4.0

Para este proyecto es necesaria la utilización de algunos controles estándar de Windows, sobre todo para la GUI (Interfaz Gráfica de Usuario). Dado que los proyectos de XNA Game Studio (XNA a partir de ahora) no los utilizan y que no permite especificar el manejador de la superficie sobre la que queremos pintar, debemos realizar algunas modificaciones para poder integrar ambos conceptos [9].

Al crear un proyecto XNA, se crean las clases principales para sostener el conjunto, tal y como observamos en el siguiente diagrama de clases:

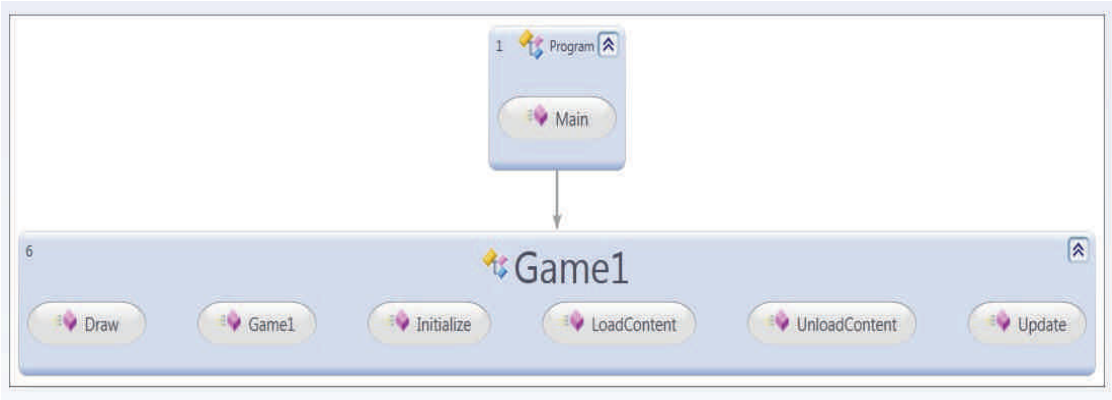


Figura 5: Diagrama de clases de un proyecto XNA vacío

La clase más importante de las dos que aparecen es Game1, que es la que se encarga de ejecutar los siguientes métodos y en este orden:

-*Initialize*: Aquí se realiza cualquier inicialización necesaria antes de ejecutarse. Se pueden cargar los elementos no gráficos que sean necesarios.

-*LoadContent*: Se usa para la carga de elementos gráficos que serán guardados en el *GameContent* (un repositorio de elementos que pueden llamarse en tiempo de ejecución y que serán cargados sólamente al inicio)

-*Update*: En este método estarán las operaciones necesarias para la actualización del "juego", tales como tratar las entradas, ejecutar la lógica, actualizar el entorno, etc.

-*Draw*: Este es el método que se llama para dibujar el resultado del método anterior.

Tanto *Update* como *Draw* se llaman una vez por *frame*, mientras que el resto se llaman una única vez, ya sea al principio de la ejecución o al terminar ésta.

-*UnloadContent*: Se llama al finalizar la ejecución y sirve para deshacerse de todo aquello que hemos cargado en *LoadContent*.

Como observamos, cada *frame* se irán ejecutando los métodos *Update-Draw* en este orden Este comportamiento se corresponde con el bucle que se ha mostrado en la Figura 1, integrando el tratamiento de las entradas en el método *Update*. Utilizar XNA evita tener que programar temporizadores para estas tareas, además de separar limpiamente las dos operaciones.

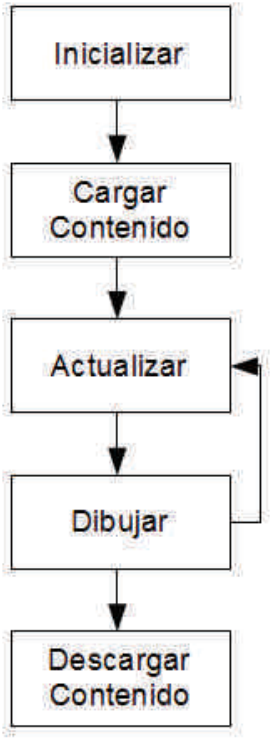


Figura 6: Ciclo de un programa creado con el Framework XNA Game Studio

Una vez creado el proyecto, el problema que se presenta es que XNA dibuja sobre su propia ventana, por lo que debemos cambiar la ventana de dibujo para que dibuje sobre la que nosotros le indiquemos. En nuestro caso un contenedor de tipo Panel dentro de una ventana estándar Windows. Para ello añadiremos al proyecto una ventana.

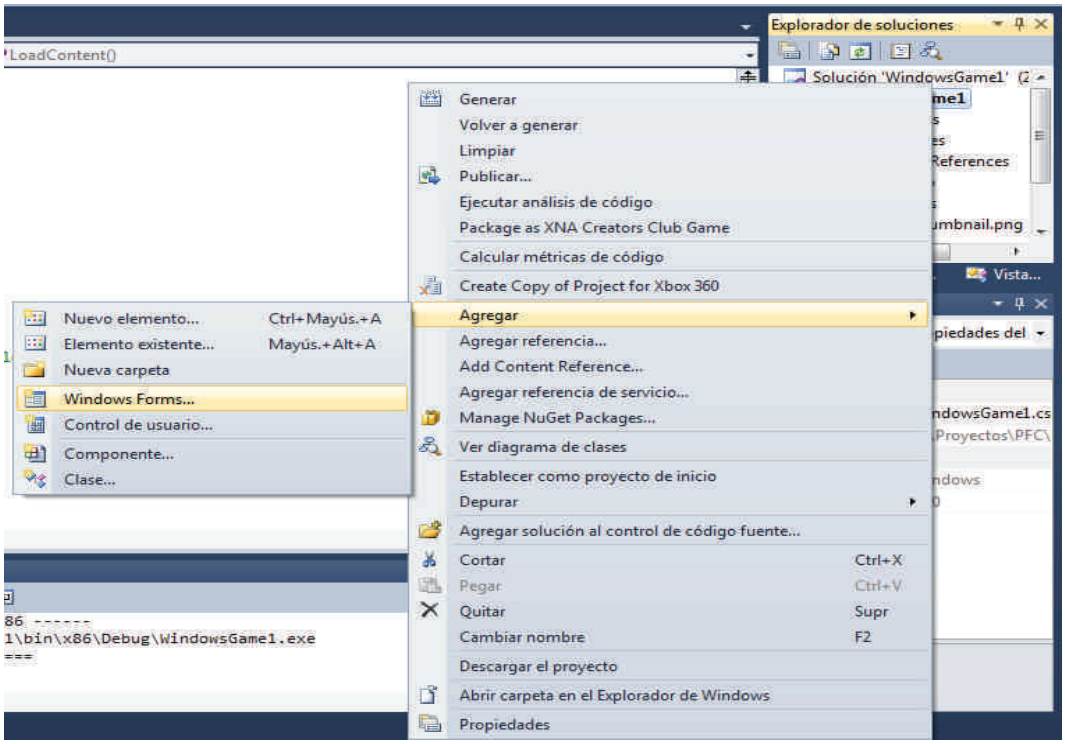


Figura 7: Añadir una ventana estándar al proyecto XNA

Una vez añadida la ventana, añadiremos un Panel que será sobre el que indicaremos a XNA que dibuje cada vez que ejecute el método Draw. Para ello deberemos realizar estas tres tareas:

-Cambiar el *handle* sobre el que dibujará XNA. Para ello en el constructor de la clase Game1 debemos suscribirnos al evento de preparación de parámetros del dispositivo.

```
graphics.PreparingDeviceSettings += new
EventHandler<PreparingDeviceSettingsEventArgs>(graphics_PreparingDeviceSe
ttings);
```

Y dentro del método cambiamos el manejador sobre el que dibujará el dispositivo gráfico.

```
void graphics_PreparingDeviceSettings(object sender,
PreparingDeviceSettingsEventArgs e)
{
    e.GraphicsDeviceInformation.PresentationParameters.DeviceWindowHandle
= formPrincipal.PanelXNA.Handle;
}
```

De esta manera cada vez que vaya a dibujar el dispositivo gráfico lo hará sobre el panel que hemos creado en nuestro formulario.

-Cuando la ventana por defecto reciba el foco hacerla invisible y cambiar la propiedad *TopMost* de nuestro formulario. De esta manera no estará por encima de la ventana estándar Windows que creemos.

En el constructor de la clase Game1 obtendremos el manejador de la ventana que XNA crea por defecto.

```
Form xnaWindow = (Form)Control.FromHandle((this.Window.Handle));
```

Una vez hecho esto nos suscribiremos al método GotFocus, que se llama cuando se produce un mensaje WM_SETFOCUS, es decir, cuando la ventana XNA es seleccionada y obtiene el foco. Dentro del método GotFocus haremos dos cosas, la primera es ocultar la ventana XNA (no necesitamos que se vea, ya que pintaremos sobre el panel) y la segunda es hacer que la propia ventana XNA no sea la que está en la posición superior, sino que sea nuestra ventana. Esto se hace porque queremos que sea nuestra ventana y no la XNA la que se vea en primera instancia.

```
xnaWindow.GotFocus += new EventHandler(xnaWindow_GotFocus);
void xnaWindow_GotFocus(object sender, EventArgs e)
{
   ((Form)sender).Visible = false;
   formPrincipal.TopMost = false;
}
```

-Adaptar el dispositivo y el *aspectratio* de la cámara cada vez que el tamaño del panel del formulario cambia, para no perder las proporciones (aunque en nuestro caso no es necesario ya que el tamaño del panel permanece constante).

Esto lo haremos suscribiéndonos al evento Panel_Resize, que se llama cada vez que el tamaño del panel cambia (en nuestro caso, esto no ocurre), y, dentro del método, cambiar las propiedades del dispositivo y el *aspectratio*.

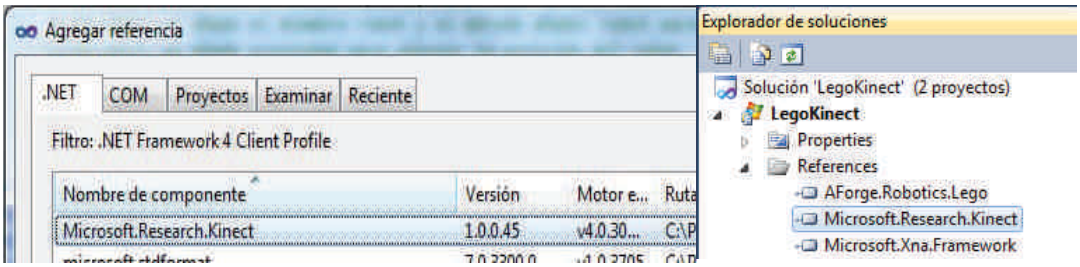
Una vez realizados estos tres pasos, ya tendremos integrado XNA en un formulario Windows estándar, pudiendo utilizar las ventajas que nos presenta XNA y los controles estándar de Windows.

5.1.2.2.Integrar el sensor Microsoft Kinect

Integrar el sensor Kinect en una aplicación de Windows es relativamente sencillo. Una vez instalado el SDK, debemos añadir a nuestra aplicación las referencias

necesarias para que podamos utilizar las librerías que nos proporciona Microsoft.

Una vez añadidas las referencias, creamos una clase que encapsulará las operaciones que deberá realizar el objeto Kinect. Para poder utilizar el sensor debemos inicializarlo previamente a su uso y, además, comprobar que realmente tenemos un sensor instalado en la máquina.



Figuras 8 y 9: Añadir la referencia a la librería de manejo de Microsoft Kinect

La comprobación de la existencia del sensor la haremos en el constructor de la clase. De esta manera, si no tenemos instalado ningún sensor Kinect, saldremos de la aplicación. El SDK de Microsoft nos proporciona la clase Runtime para poder realizar esta comprobación. Si el miembro estático Runtime.Kinects.Count es 0 no tenemos instalado ningún sensor en la máquina, por lo que actuamos en consecuencia saliendo de la aplicación. Esta clase nos sirve también para instanciar los sensores Kinect que tengamos instalados en la máquina, proporcionándonos los métodos necesarios para su configuración y manejo. Tendremos que crear un objeto de esta clase para poder acceder al manejo del sensor.

Una vez comprobada la existencia del sensor, procederemos a su inicialización. Esta consiste en inicializar el sensor con las opciones que necesitemos dependiendo de los tipos de datos que vamos a manejar (imagen en color, imagen de profundidad, imagen de profundidad e índice de jugardor y, por último, rastreo de esqueleto). En nuestro caso usaremos únicamente la imagen en color y la imagen de profundidad. Teniendo en cuenta que runtime es un objeto de la clase Runtime, haremos lo siguiente:

```
runtime.Initialize(RuntimeOptions.UseDepth | RuntimeOptions.UseColor );
```

Tras indicarle al sensor el tipo de imágenes que debe generar, debemos suscribir el objeto a los eventos que se generan cada vez que el sensor tiene una imagen del tipo que le hemos indicado disponible. Tal y como muestran las características técnicas del sensor, estos eventos serán llamados 30 veces por segundo, con lo que tendremos disponibles 30 imágenes de cada tipo por segundo.

```
runtime.VideoFrameReady += new  EventHandler
<ImageFrameReadyEventArgs>(VideoFrameListo);
runtime.DepthFrameReady += new EventHandler
<ImageFrameReadyEventArgs>(DepthFrameListo);
```

Como observamos, le estamos indicando los eventos que debe ejecutar cada vez que tiene disponible un *frame* de video o uno de profundidad. Dentro de los métodos VideoFrameListo y DepthFrameListo realizaremos las operaciones que estimemos oportunas para obtener las imágenes que necesitemos.

El paso final en la configuración es abrir los *streams* de video y profundidad indicándole el tipo de imagen que queremos, el número de *buffers* y la resolución de la imagen. Queremos que capture las imágenes de video a resolución máxima (640x480 *pixels*), mientras que las de profundidad las recogeremos a 320x240 *pixels*.

```
runtime.VideoStream.Open(ImageStreamType.Video, 2,
ImageResolution.Resolution640x480, ImageType.Color);
runtime.DepthStream.Open(ImageStreamType.Depth, 4,
ImageResolution.Resolution320x240, ImageType.Depth);
```

En los eventos VideoFrameListo y DepthFrameListo lo único que hacemos es copiar el *frame* recibido en el objeto de la clase Ckinect que le corresponda, video o profundidad.

En este punto hay que hacer una aclaración sobre el formato en el cual Kinect devuelve los *frames* obtenidos. La clase a la que pertenecen estos *frames* es ImageFrame, cuyos miembros vemos a continuación.

Figura 10: La clase ImageFrame y sus miembros

El miembro más interesante para nosotros es Image que pertenece a la clase PlanarImage y que contiene los pixels de la imagen consecutivos y sin comprimir, por lo que se puede acceder a ellos mediante la propiedad de PlanarImage denominada Bits como si fuera un *array*. Esto nos permitirá hacer copias de la imagen a otros formatos (como el IplImage de la librería OpenCV) que nos darán más flexibilidad a la hora de manejar las imágenes.

El *frame* de profundidad tiene otra serie de particularidades que hay que comentar. Se ha mencionado que la imagen de profundidad devuelve la distancia a la que los rayos lanzados rebotan contra algo. Esta información la recibimos en un objeto de tipo PlanarImage en el cual a cada pixel de la imagen de profundidad le corresponden dos *bytes*. El problema viene a la hora de interpretar esos dos *bytes*. El *byte* de menor peso es el primero de los dos y el de mayor peso el segundo. Los primeros 3 *bits* del *byte* de menor peso se corresponden con el índice del jugador (0 en nuestro caso ya que le hemos indicado a Kinect que sólamente queremos que genere la imagen de profundidad, sin información sobre el jugador). Para obtener la información correcta deberemos realizar algunas operaciones adicionales. Dado que a la información debemos acceder por *bytes*, para obtener el valor de la distancia en un pixel (x,y) concreto, debemos desplazar el primer *byte* del valor cinco *bits* a la izquierda (no 8 ya que los 3 primeros del segundo *byte* pertenecen al índice del jugador) y tres *bits* a la derecha el segundo *byte*, sumando ambos resultados.

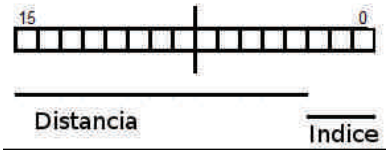


Figura 11: Obtención de la distancia en la imagen de profundidad (bytes ya ordenados correctamente)

Por lo que para obtener el valor de la distancia del pixel (X,Y) teniendo en cuenta que resultado es una matriz de enteros de 16 *bits* (de ahí viene el 2 de las multiplicaciones, 2 *bytes* por pixel) almacenada por filas y que profundidad es de tipo ImageFrame, la operación a realizar sería:

```
resultado[Y * profundidad.Image.Width + X] = (Int16)
(((profundidad.Image.Bits[Y * 2 * profundidad.Image.Width + X * 2]) >> 3)
|((profundidad.Image.Bits[Y * 2 * profundidad.Image.Width + X * 2 + 1])
<< 5));
```

Con lo que obtendríamos en resultado para cada pixel (X,Y) de la imagen la distancia a la que se encuentra el objeto contra el que ha chocado el rayo del emisor IR.

5.1.2.3.Detección de objetos

OpenCV (*Open source Computer Vision*) es una biblioteca libre orientada a la visión por computador desarrollada originalmente por Intel. Su primera versión data de 1999. Se distribuye bajo licencia BSD, y es gratuita tanto para usos académicos como comerciales. Existen versiones para Windows, Linux, Mac y Android. Tiene interfaces

para los lenguajes C++, C, Python y Java (sólo Android) [10]. Debido a que la aplicación se desarrolla en C#, no podemos usar directamente OpenCV, sino que usaremos un *wrapper*, OpenCVSharp [11]. Éste se encargará de realizar las llamadas pertinentes a las diferentes dll (*dynamic link library*-librería de enlace dinámico) que conforman OpenCV, ofreciéndonos toda su funcionalidad desde C#. Una vez añadidas las librerías de OpenCV al proyecto y las referencias a OpenCVSharp, podremos hacer uso de esta potente librería gráfica.

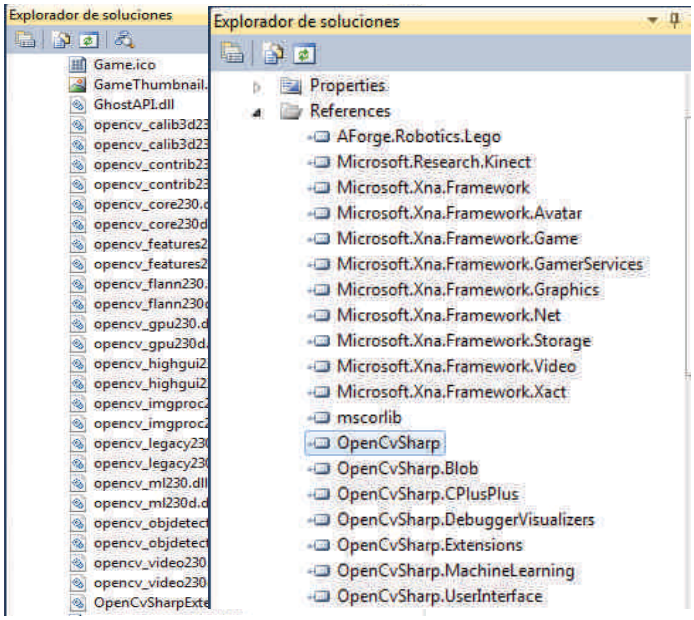


Figura 12: Librerías añadidas de OpenCV y referencias de OpenCVSharp

La detección y reconocimiento de objetos es un campo importante dentro de la visión por computador. Existen multitud de métodos para realizar esta tarea, y una gran cantidad de estudios y proyectos que profundizan y/o estudian nuevos métodos más eficientes. Resulta complicada la elección de uno de los métodos sin antes evaluar los resultados obtenidos, por lo que ha habido que considerar y/o probar varios métodos antes de decantarse por uno en particular. OpenCv nos ofrece distintos métodos para la detección de objetos, de los cuales se han considerado y/o probado tres:

1-Uso de rasgos Haar:

Se basa en el concepto de integral de imagen, en el cual la integral de una imagen en la posición (x,y) es la suma de los pixels situados a la izquierda y arriba de (x,y), incluidos éstos y en la utilización de clasificadores Haar, que son denominados rasgos. El sistema clasifica las imágenes en positivas y negativas en función de los resultados.

Paul Viola y Michael Jones realizaron al respecto varias publicaciones [12].

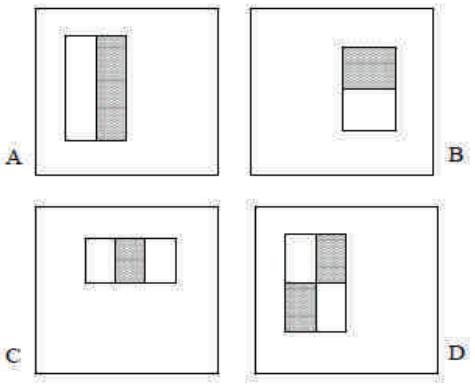


Figura 13: Ejemplo de rectángulos de rasgos. La suma de los pixels dentro del rectángulo blanco, se restan de la suma de los pixels del rectángulo gris. A y B muestran rasgos de dos rectángulos, C un rasgo de tres rectángulos y D uno de cuatro. Fuente: http://research.microsoft.com/enus/um/people/viola/Pubs/Detect/violaJones_CVPR2001.pdf

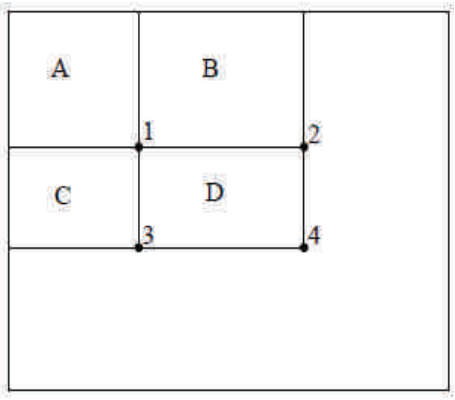


Figura 14: Integral de Imagen. La suma de los pixels del rectángulo D puede hacerse con cuatro referencias a un array. El valor de la integral de imagen en el punto 1 es la suma de todos los pixels dentro de A. El valor en 2 es la suma A+B. En 3 es A+C. En 4 es A+B+C+D. La suma de los pixels en D es 4+1-(2+3). Fuente: http://research.microsoft.com/enus/um/people/viola/Pubs/Detect/violaJones_CVPR2001.pdf

La detección de objetos con este sistema se realiza en dos fases. Una fase de aprendizaje en la cual le proporcionamos al computador imágenes donde está el objeto que queremos detectar (positivas), indicándole su situación. El computador procesará estas imágenes extrayendo los rasgos que caracterizan al objeto. Posteriormente, proporcionaremos imágenes donde no esté el objeto (negativas), para eliminar los rasgos que puedan producir "falsos positivos" (considerar el objeto algo que en realidad no lo es). Tras este primer paso, el siguiente es el testeo de los rasgos calculados para su comprobación [13].

Este sistema de detección se utiliza mucho en la detección de rostros en las fotografías. Tras la realización de una pequeña prueba, el sistema se rechazó debido a la carga de trabajo que supone (para que la extracción de rasgos sea buena, es necesario del orden de 1000 imágenes positivas y más de 2000 negativas, tomando su proceso del orden de hasta dos semanas). Además es sensible al escalado y a las rotaciones, como los objetos que nos interesa detectar pueden tener diferentes orientaciones, este sistema no es apropiado.

2-SURF (*Speeded Up Robust Features*)

SURF es un método de detección y descripción de puntos de interés que se usa en la detección de objetos presentado por Herbert Bay, Andreas Ess, Tinne Tuytelaars y Luc Van Gool en 2006. Es independiente de la escala y la rotación, por lo que, a priori, puede cumplir con nuestro objetivo [14].

Su funcionamiento a grandes rasgos es el siguiente, tomando como referencia una imagen del objeto que deseamos detectar, se seleccionan primero los "puntos de interés" de la imagen. Estos puntos suelen situarse en esquinas, intersecciones o *blobs* (conjunto de pixels adyacentes que tienen un mismo estado lógico, o dicho de otra forma, que no difieren de los de su alrededor [15]). La propiedad más interesante de un detector de puntos de interés es su repetibilidad. Repetibilidad quiere decir que un detector debería encontrar los mismos puntos de interés sea cual sea la perspectiva desde la que se ve el objeto (dentro de unos límites). Lo siguiente que se hace es definir un vector de rasgos (descriptor) que representa a los vecinos de cada uno de los puntos de interés. Este descriptor debe ser distintivo, y además, robusto frente al ruido, errores de detección y a las transformaciones geométricas y fotométricas. Finalmente, los vectores de descriptores se comparan entre dos imágenes diferentes, esta comparación se suele realizar por medio de distancias entre vectores (por ejemplo, distancia euclídea). La dimensión del vector incide directamente en el tiempo de cálculo, por lo que es recomendable que el vector sea de una dimensión baja.

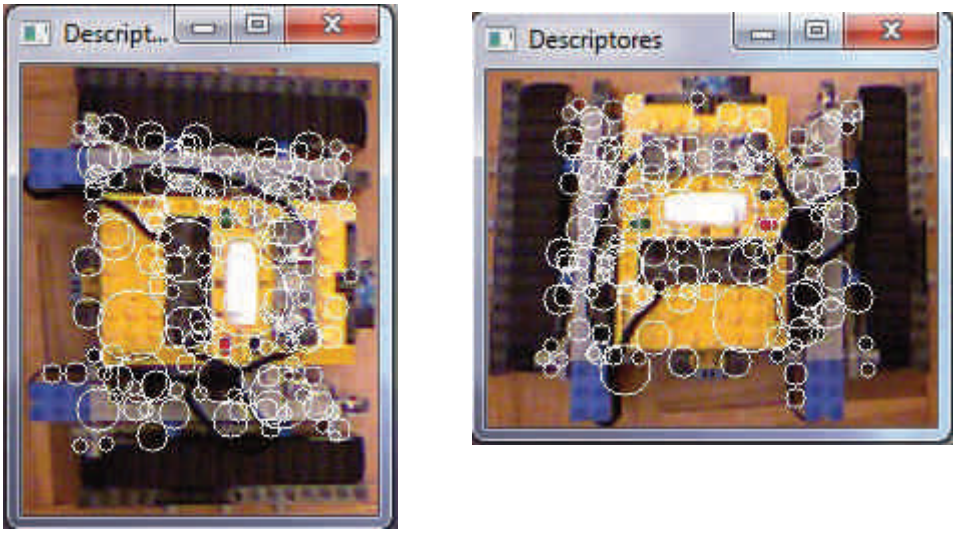


Figura 15: Repetibilidad. A pesar de girar 90º la imagen, los puntos de interés obtenidos son muy similares

OpenCVSharp nos ofrece, en uno de sus programas de ejemplo, una implementación de SURF, que, con los cambios pertinentes para adaptarla a la aplicación desarrollada ha sido probada para ver su rendimiento. Su mecánica sería la siguiente: tenemos una imagen del objeto que queremos detectar, de la cual calculamos y almacenamos sus descriptores. Una vez cada 15 *frames* obtenemos la imagen que capta el sensor Kinect y calculamos sus descriptores. Posteriormente tratamos de encontrar correspondencias entre los descriptores del objeto y los de la imagen. Si existen correspondencias, habrá que calcular la matriz de transformación que transforma los puntos de interés del objeto a los de la imagen.

Una vez obtenida la matriz sólamente hemos de transformar los puntos correspondientes a las cuatro esquinas del objeto original para obtener su posición dentro de la imagen obtenida por el sensor.

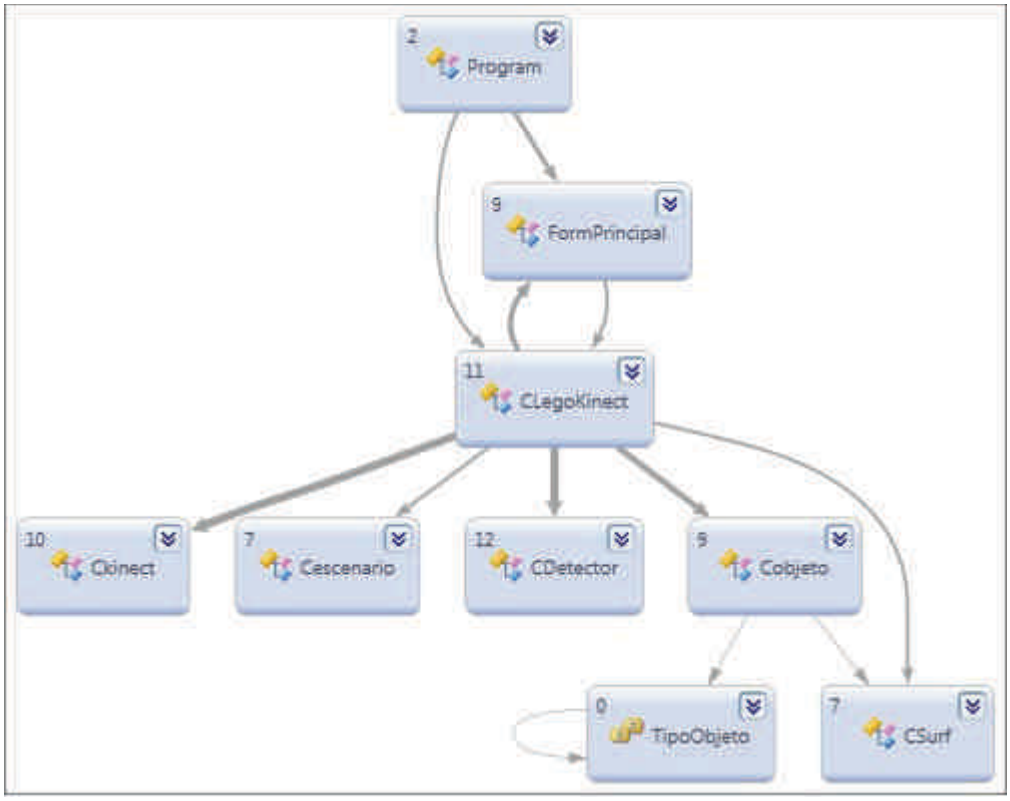


Figura 16: Diagrama de clases del proyecto utilizando SURF como método de detección

En el proyecto utilizamos el *framework* XNA. Esto hace que la clase que está por encima de todas es Program. Esta clase es la que se encarga de lanzar el marco y crear tanto el formulario principal como el objeto CLegoKinect. Este objeto de clase CLegoKinect será el encargado de gestionar la aplicación. Pasamos a explicar con detalle las clases más importantes que componen la aplicación en este punto.

-CLegoKinect: es la clase principal, contiene referencias a los elementos necesarios para el funcionamiento de la aplicación (sensor, objetos de la escena, etc). Se encarga también de recibir los eventos de la capa de presentación para distribuirlos a los objetos que corresponda.

-FormPrincipal: se corresponde con la capa de presentación, es el formulario utilizado como GUI.

-Ckinect: clase que encapsulará la inicialización y puesta en marcha de un sensor Kinect,. Podremos obtener las imágenes tanto RGB como de profundidad. En este momento, devolverá imágenes de tipo PlanarImage o Bitmap, más adelante esto cambiará por comodidad.

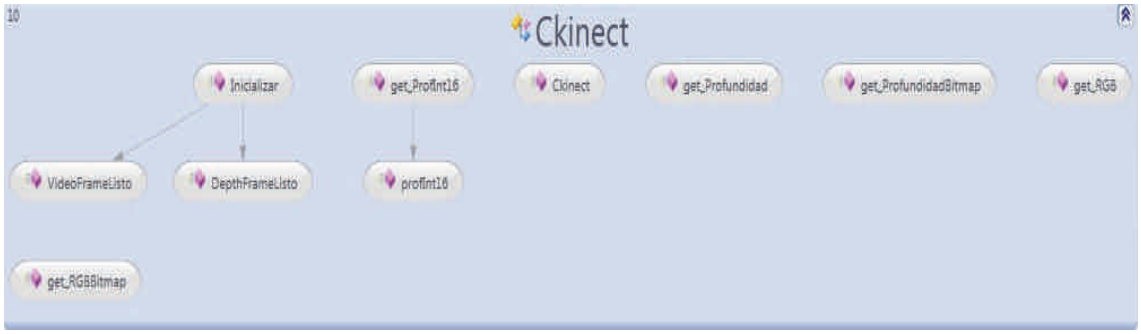


Figura 17: Clase Ckinect en detalle

Los métodos de la clase Ckinect son los siguientes:

-Inicializar que se encarga de inicializar el sensor y de suscribirse a los eventos que se produzcan al tener listo un *frame* de video o uno de profundidad.

-get_ProfInt16 nos devuelve la imagen de profundidad como una matriz de enteros de 16 *bits*, con las distancias ya calculadas.

-get_* y get_*Bitmap nos devuelven la imagen RGB o de profundidad en formato PlanarImage o Bitmap, según necesitemos.

-Cobjeto: clase que representa un objeto. Esta es la clase base sobre la que posteriormente se derivarán otras (como, por ejemplo, los robots). Contiene datos sobre el propio objeto (nombre, posición...) además de almacenar los descriptores y puntos clave obtenidos por SURF.

-Csurf: es la clase que contiene las operaciones necesarias para aplicar el método SURF de detección de objetos.

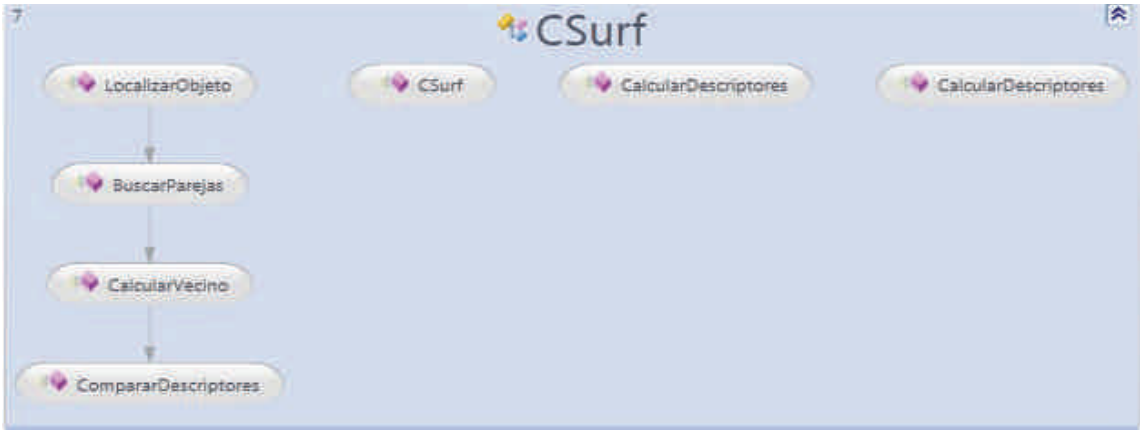


Figura 18: Clase CSurf en detalle

Observamos en la imagen que tenemos dos operaciones de CalcularDescriptores. Esto es debido a que los objetos que tengamos almacenados deberán, a su vez, proporcionar almacenamiento a la secuencia de datos que se obtiene de la operación. Esto no es necesario cuando tratamos la imagen obtenida en un determinado *frame*, debido a que no vamos a volver a acceder a esos datos en *frames* posteriores, si no que crearemos una secuencia nueva.

El método LocalizarObjeto devolverá las cuatro esquinas del objeto localizadas en la imagen, si hay alguna correspondencia o *null* en caso de que no se hayan encontrado. A la hora de calcular los vecinos aplica el algoritmo de clasificación *K-Nearest Neighbor* [16], que consiste en utilizar la distancia euclídea para clasificar los puntos. En nuestro caso, para cada descriptor del objeto se buscará el descriptor de la imagen que tenga el mismo valor de Laplaciano y que esté a menor distancia. Una vez realizado esto para todos los descriptores, hemos de encontrar una matriz de correspondencia entre los puntos del objeto y los obtenidos en la imagen. Para ello utilizaremos una función de OpenCVSharp llamada *FindHomografy* que a partir de dos matrices de puntos (A y B), trata de encontrar la matriz de transformación (C) que relaciona las dos matrices, es decir, la matriz C tal que AxC=B.

Este método tiene algunas desventajas, la principal de ellas es la cantidad de tiempo necesario para localizar el objeto en la imagen (unos 100ms de media en una imagen tomada por el sensor de 640x480 pixels). Bien es cierto que con valores mayores del umbral de detección (del orden de 1500), el tiempo de cálculo baja hasta valores asumibles para una aplicación en tiempo real, pero la eficacia en la detección disminuye drásticamente. Lo mismo ocurre si disminuimos el tamaño de la imagen a la mitad (320x240 pixels). El tiempo de cálculo desciende muchísimo pero también lo hace la detección. Se producen muchos falsos negativos (no detectar el objeto cuando en realidad sí debería hacerlo) y detecciones incorrectas (localizar mal el objeto). Teniendo en cuenta que además no realizamos el cálculo de los descriptores en todos los *frames* sino que sólamente una vez cada 15 *frames* (con lo que en caso de movimiento del robot podríamos perder su posición real) este método (por lo menos por sí solo) no es adecuado para esta aplicación en particular.

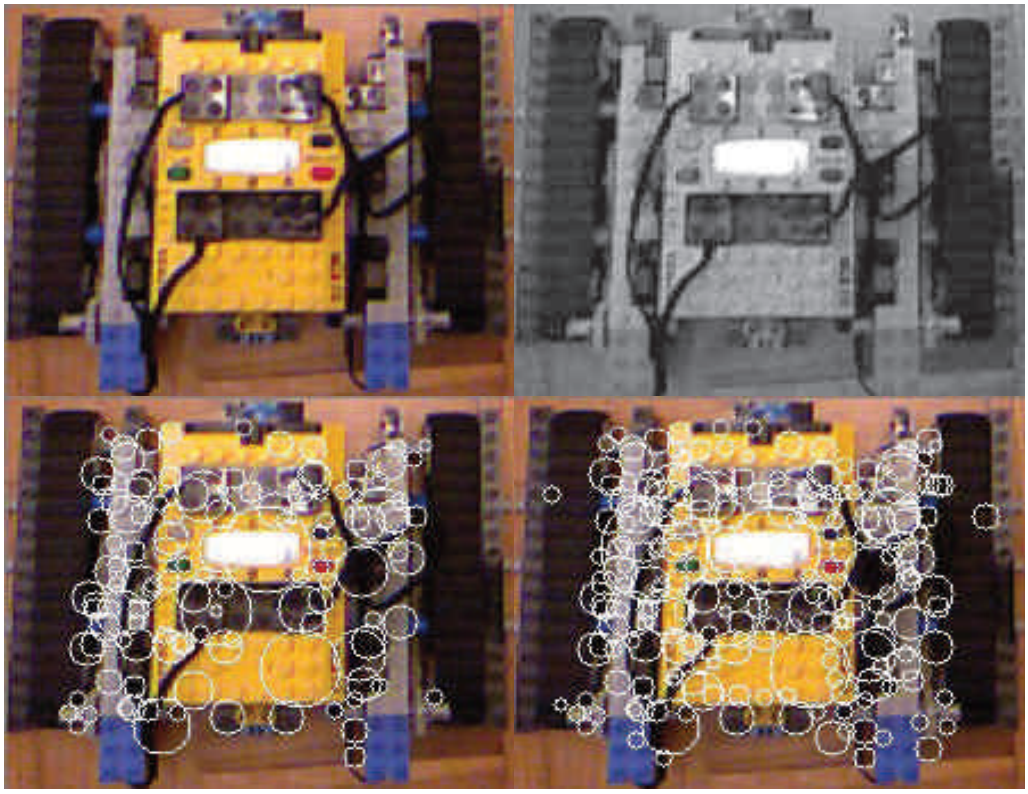


Figura 19: Cálculo de los puntos clave con SURF. La imagen superior izquierda es la original, el objeto a detectar, posteriormente se pasa a escala de grises (superior derecha). A partir de esta imagen se calculan los puntos de interés (inferiores). Están representados por circunferencias de tamaño función de su valor de gradiente o Laplaciano. Los puntos de interés tienen un valor mayor cuanto mayor es la variación del tono de gris con respecto a su alrededor. La imagen inferior izquierda representa los puntos con un valor umbral de 2200. La derecha representa los puntos con un umbral de 500, obsérvese el aumento de los puntos de interés.

3-*Color Tracking*

Este es el último método probado y el que finalmente se ha seleccionado, a pesar de sus limitaciones, que son varias. Básicamente consiste en aislar un color (o rango de colores) de la imagen para obtener su situación. Es obvio que esto trae consigo una serie de limitaciones, por ejemplo, que el color elegido para un objeto a detectar no deba aparecer en otros objetos que formen parte de la escena, ni en la propia imagen que forme la escena para evitar detecciones erróneas. Además es sensible a los cambios de

iluminación, por lo que las condiciones de uso serán limitadas, pero su tasa de *frames* por segundo es lo suficientemente alta como para poder usarse en un entorno productivo.

Para explicar cómo se pueden aislar los colores, primeramente se debe hablar del modelo de color HSV. Un modelo de color es un modelo matemático abstracto que permite representar los colores en forma numérica, utilizando típicamente tres o cuatro valores o componentes cromáticas [17]. Existen diferentes modelos de color en el mundo de las computadoras, como pueden ser RGB, CMYK o HSV [18], mientras que los dos primeros se dice que están orientados al *hardware*, el modelo de color HSV se dice que está orientado al usuario. El modelo RGB se basa en los colores aditivos y se usa en el trabajo con monitores, sus colores primarios son el rojo, verde y azul. El modelo CMYK se basa en los colores sustractivos y se usa en pintura, imprenta, etc. Sus colores primarios son el cyan, magenta, amarillo y negro. Ambos modelos están relacionados ya que los primarios de uno son secundarios del otro y viceversa. El modelo HSV, en cambio, define los colores en base a tres parámetros, matiz (*hue*), saturación y valor (brillo). El matiz se corresponde con el color, la saturación se refiere a la concentración del color: un color completamente saturado aparece puro. El valor (o brillo) indica el grado de claridad del color: un color con un valor alto es claro, mientras que con un valor bajo tiende a ser oscuro [19].

Para aislar un color convertiremos la imagen al modelo HSV. En este modelo, las variaciones de un color comparten una gama de matices (H) y un rango de valores de saturación y valor (S y V), siendo más sencillo su aislamiento que si utilizásemos modelos tales como el RGB, donde, para una variación de color las combinaciones son mayores. Opencvsharp nos ofrece la operación *InRangeS* que, a partir de una imagen en modelo HSV y dos valores que denotan los colores umbral máximo y mínimo, nos devuelve una imagen binaria dónde los *pixels* que corresponden a ese intervalo de colores aparecen en blanco y los que no, lo harán en negro.

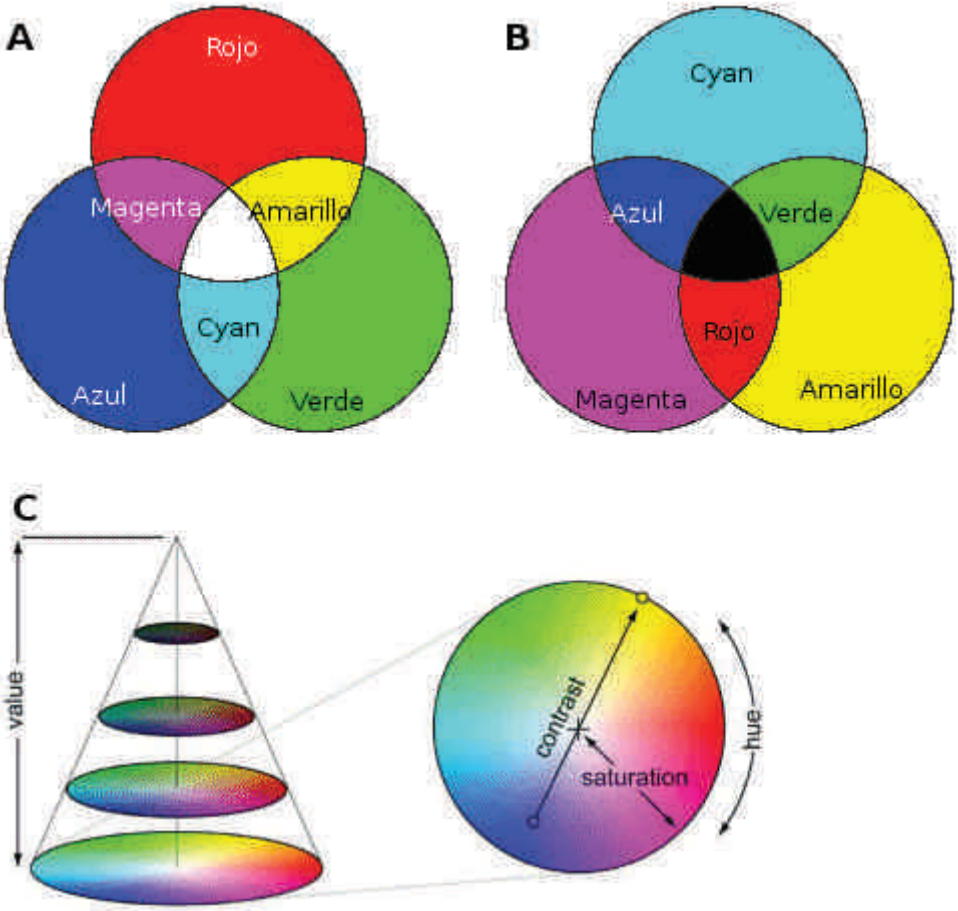


Figura 20: Varios modelos de color. A:RGB, B:CMYK, C:HSV

Pasando ya a la implementación, en esta fase el diagrama de dependencias de clases no varía (salvo que se podría eliminar la clase Csurf) , pero sí las operaciones contenidas en la clase Cdetector y el funcionamiento de la detección de objetos en el método *Update* de la clase CLegoKinect.

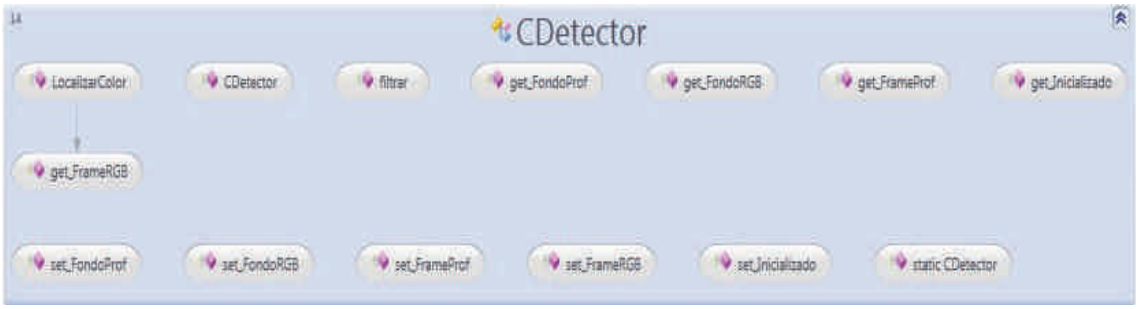


Figura 21: Métodos de la clase Cdetector utilizando Color Tracking

Aquí la clase Cdetector coge protagonismo ya que es la encargada de la detección de los colores (asociados normalmente a algún objeto) en la escena. Dicha operación la realiza con el método LocalizarColor que pasamos a explicar a continuación.

El método LocalizarColor toma como parámetro un color (en modelo HSV) y la imagen captada por el sensor y devuelve el punto donde está situado el centro de la masa de ese color. Este punto se correspondería (si se cumplen las restricciones en cuanto al color de los objetos) con la posición que ocupa el objeto que deseamos detectar. Para realizar esta operación se siguen una serie de pasos previos. Primero se debe convertir la imagen obtenida por el sensor al modelo HSV. Se calculan los umbrales máximo y mínimo del color (se usan márgenes fijos para los valores de H y S y un % para el V), para así obtener la imagen filtrada en escala de grises (aunque realmente es binaria, pixels en blanco si están en el rango o en negro si no lo están). El umbral se sitúa en ±5 para H, ±10 para S y un 40% para V. Esto es debido a las posibles variaciones de la tonalidad en función de la iluminación. Posteriormente se pasan un filtro de dilatación y otro de erosión a la imagen obtenida.

El filtro de dilatación dilata las zonas de *pixels* que tienen un tamaño mayor al indicado. El filtro de erosión filtro elimina los *pixels* "frontera", es decir, aquellos que están al borde del objeto o aquellos *pixels* sueltos (que son los que realmente no nos interesan porque introducen ruido a nuestra imagen). De esta manera, primero agrandamos las partes que aparecen *pixels* dentro de esos márgenes y después eliminamos *pixels* que no pertenecen al objeto que nosotros queremos detectar al estar sueltos o formar un área menor a una determinada [20]. Una vez pasado el filtro calculamos el baricentro del objeto sumando la posición de los *pixels* a 1 y dividiendo por el número de *pixels* encontrados. Si el número de *pixels* encontrados es menor a uno dado suponemos que el objeto realmente no se encuentra en el campo de visión del sensor y no tenemos en cuenta lo calculado. En cambio, si el número de *pixels* es mayor, devolvemos la posición calculada.

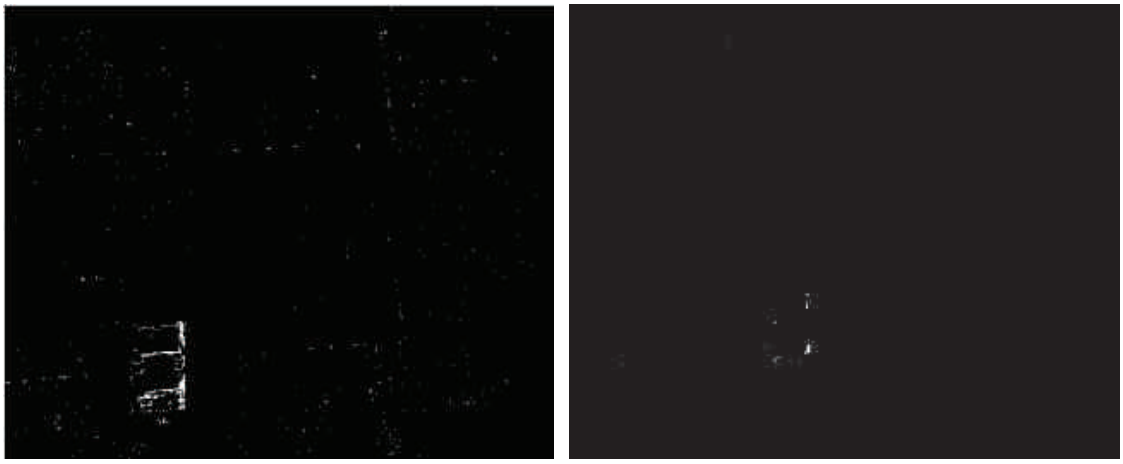


Figura 22: En estas dos imágenes se observa la diferencia entre la aplicación o no de la erosión, a la imagen de la izquierda no se le ha aplicado la erosión, mientras que a la de la derecha sí. Tras filtrar el color amarillo, se observa ruido en la imagen de la izquierda, que no aparece en la de la derecha (las condiciones lumínicas de ambas escenas son similares, ya que han sido tomadas con un intervalo de tiempo menor a un minuto). También se puede observar que la zona detectada como correcta disminuye en tamaño.

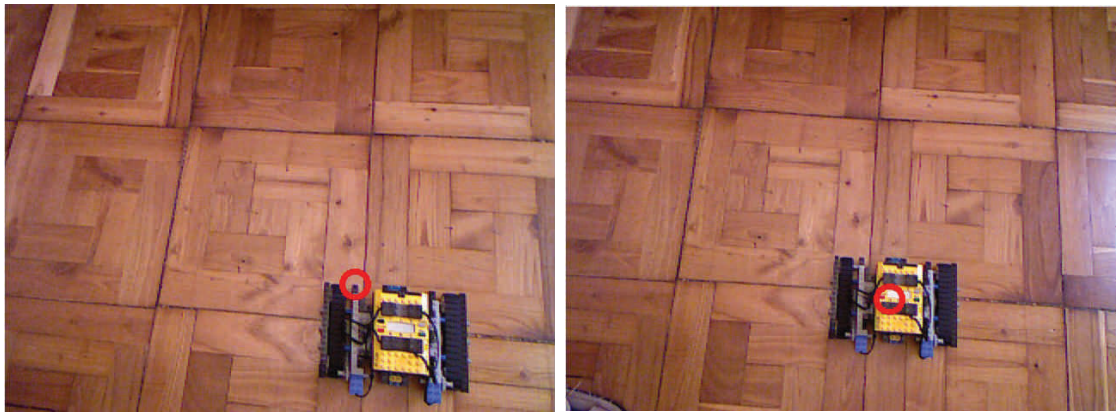


Figura 23: Como resultado de un filtrado sin erosión, en la imagen de la izquierda el centro del objeto se detecta muy desplazado debido al ruido, siendo poco preciso. En la derecha, el resultado del filtrado con erosión, aunque se han perdido puntos del objeto en el proceso, la detección es mucho más precisa que sin la aplicación de erosión.

5.1.3.Interfaz de Usuario

La interfaz de usuario en esta primera fase es muy simple tenemos el menú General que nos permite salir de la aplicación y el menú Detección desde donde podemos seleccionar el método de detección. Los métodos de detección a elegir son: *Color-Tracking* y SURF. Por defecto, al iniciar la aplicación el método seleccionado es

Color-Tracking.

Una vez inciada la aplicación, veremos dos paneles. En el panel de la derecha visualizaremos la escena captada. En este panel veremos una circunferencia de color rojo en el centro del objeto detectado si el método es *Color-Tracking*. Si el método de detección es SURF, el objeto detectado aparecerá delimitado por un cuadrilátero de color blanco.

Cuando tenemos seleccionado el método *Color-Tracking*, el panel de la izquierda mostrará la imagen captada por el sensor filtrada por el color del robot.

Si seleccionamos el método SURF, aparecerá un botón con el texto "Cargar Objeto". Si pulsamos el botón aparecerá un cuadro de diálogo para abrir la imagen del objeto que queremos detectar. Al abrir la imagen aparecerán varias ventanas donde podremos ver el proceso que sigue la imagen para obtener los puntos de interés. A partir de ese momento el *software* estará preparado para detectar el objeto cargado. Este proceso sólamente es obligatorio hacerlo la primera vez que seleccionamos SURF. Podemos sustituir el objeto a detectar cuantas veces deseemos pulsando el botón y cargando otra imagen. Como ayuda al sistema de detección SURF, cada vez que iniciamos el *software* se toma una instantánea del sensor y se guarda en el directorio de instalación con el nombre "escena.bmp". Con lo que podríamos hacer lo siguiente:

-Colocar el objeto a detectar en la escena
-Iniciar el software, se grabará la imagen "escena.bmp"
-Editar la imagen "escena.bmp" eliminando la parte de la imagen donde no aparece el objeto
-Seleccionar el método SURF
-Cargar la imagen editada como imagen del objeto a detectar

De esta manera se pueden realizar las pruebas necesarias con cualquier tipo de objeto , a partir de imágenes a la misma resolución que las que nos ofrece Kinect.

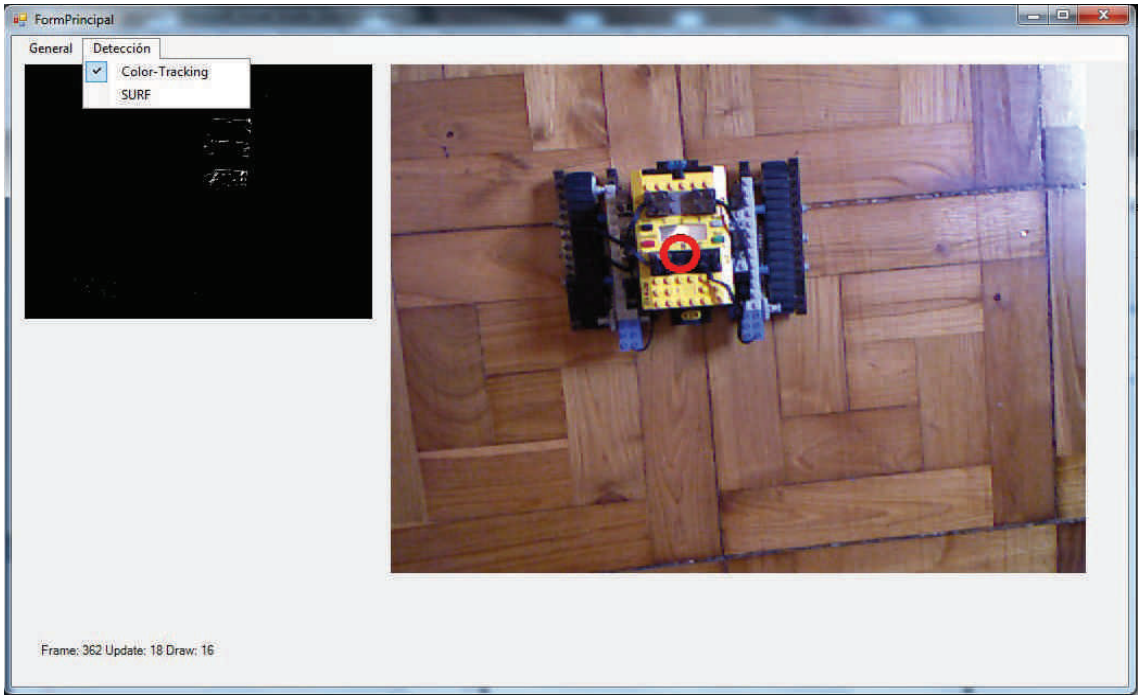


Figura 24: Ejemplo de detección de objeto mediante el método Color-Tracking.

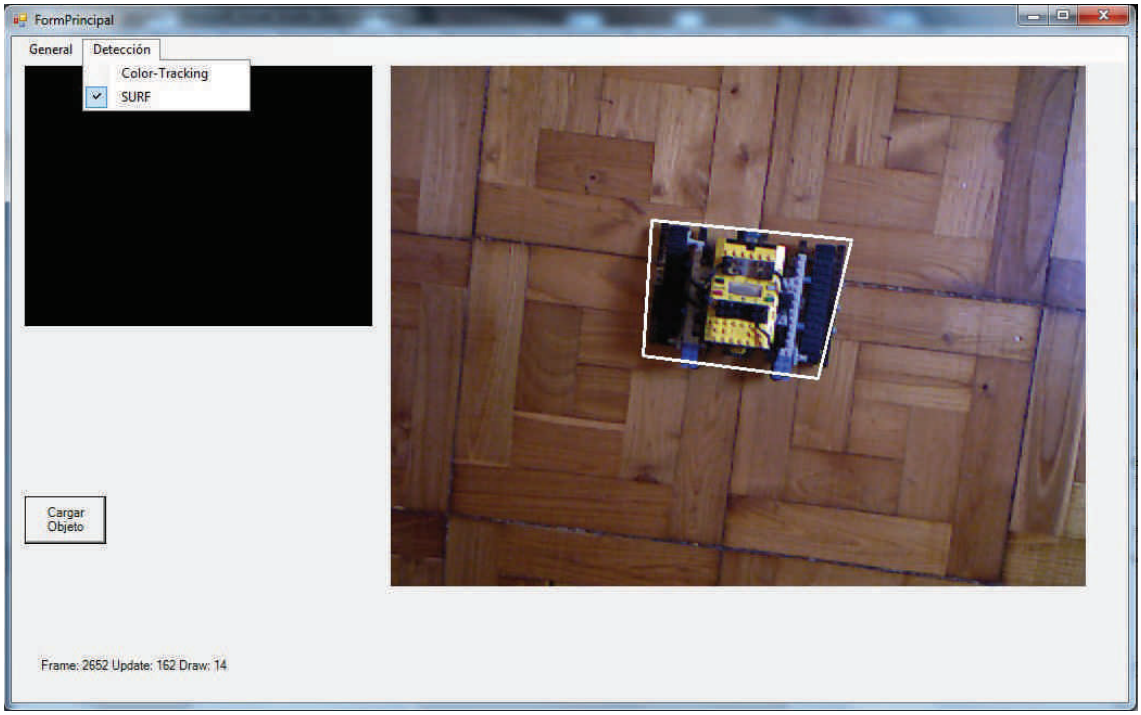


Figura 25: Ejemplo de detección de objeto mediante el método SURF.

En los ejemplos mostrados en las figuras se puede observar que se muestran en una etiqueta en la parte inferior algunos datos interesantes. En concreto el número de *frame* que se está mostrando y el tiempo que se emplea en ejecutar los métodos *Update* y *Draw*. Esto nos puede ayudar a la hora de valorar el rendimiento de ambos métodos.

5.2.Segunda Fase

En esta segunda fase el objetivo es controlar un robot Lego Mindstorm RCX mediante el ordenador, además de preparar la estructura de clases para la inclusión en un futuro de otros tipos de robots con cierta facilidad.

Para ello, se ha elegido como el marco de trabajo Aforge.Net para el manejo del robot. Aforge.Net es un marco de trabajo diseñado para desarrolladores e investigadores en campos como la visión artificial o la inteligencia artificial, y, una de sus aplicaciones está en el manejo de robots Lego Mindstorm. Posee una librería que nos proporciona manejo para robots Lego Mindstorm RCX y NXT, además de control para el robot Surveyor SRV1. Aforge.Net se publica bajo licencia LGPL v3 [21].

En esta fase nos hemos centrado sólamente en el manejo del robot Lego Mindstorm RCX. Sin embargo, la estructura de clases que se implementa permite fácilmente la inclusión de, por ejemplo, los robots del modelo NXT.

5.2.1.Hardware – Robots Lego Mindstorm RCX

-HISTORIA

Lego Mindstorms es un "juguete" de robótica creado por la empresa Lego. Posee elementos básicos de robótica, tales como la unión de piezas y la programación de acciones de forma interactiva. Se comercializó por primera vez en 1998.

La empresa llevaba bastante tiempo con la idea de conectar un computador a una de las piezas de una construcción Lego y que ésta pudiera ser programada. Pero la poca implantación de las computadoras en los hogares a principios de los años 90 y el alto coste de la tecnología retrasó el desarrollo de esta idea. Pasaron cinco años hasta que se dieron las condiciones para que se empezara el desarrollo de lo que terminaría siendo el bloque RCX, un bloque que posee un microcontrolador y que es el eje sobre el que gira el producto Mindstorms.

Lo que en un principio fue un acuerdo de colaboración entre Lego y el MIT, se convirtió en dos ideas diferentes en cuanto a diseño, dado que la elección de ambas en cuanto al público destino del producto eran completamente diferentes. Por un lado, Lego orientó el producto hacia niños de entre 10 y 14 años, mientras que el MIT lo orientó a la investigación del proceso de aprendizaje de los niños. Centrándonos en la decisión de la empresa Lego, esto hizo que el producto fuera diseñado de la forma que se conoce, robusto y asequible.

Figura 26: Bloque de control del robot Lego Mindstorm RCX

Una vez lanzado el producto, éste atrajo las miradas de un público con el que no se había contado, los aficionados a la robótica. Esto hizo que las ventas fueran mucho mayores de lo que se esperaban. Este hecho provocó que se creara una extensa comunidad de usuarios que ampliarion las posibilidades del producto, mediante la creación de entornos de programación, sistemas operativos y numerosas páginas web para intercambio de información.

En 2006 Lego sacaría al mercado el bloque NXT, una versión mejorada del RCX y actualmente va por la versión 2.0. [22]

Hoy día el bloque RCX está completamente descatalogado e incluso no se puede encontrar información sobre él en la propia página web de Lego, por lo que la ayuda e información hay que buscarla en la amplia comunidad de usuarios que todavía existe.

-CARACTERISTICAS TECNICAS

El bloque RCX cuenta con un microcontrolador Hitachi H8/3292 alimentado a 5V y con una frecuencia de funcionamiento aproximada de 16MHz. Tiene una ROM de 16KB y una RAM externa de 28KB [23]. Tiene además un convertidor A/D para la conversión de las entradas en datos manejables.

Posee 3 entradas y otras tantas salidas, las entradas pueden ser conectadas a sensores de estos tres tipos:

-Sensor rotacional. Mide la rotación de un eje, 16 puntos por vuelta
-Sensor de luz. Reacciona según los cambios de luz.
Valores entre 0(oscuro)-100
-Sensor de tacto. Reacciona al contacto entregando un valor binario.
1 apretado 0 no.

Estos sensores deben conectarse a los puertos de entrada 1,2 o 3 del RCX.

Las salidas se conectan a motores, que son los que generan el movimiento del robot, transforman la energía provista por las baterías del RCX (9V) en movimiento rotacional.

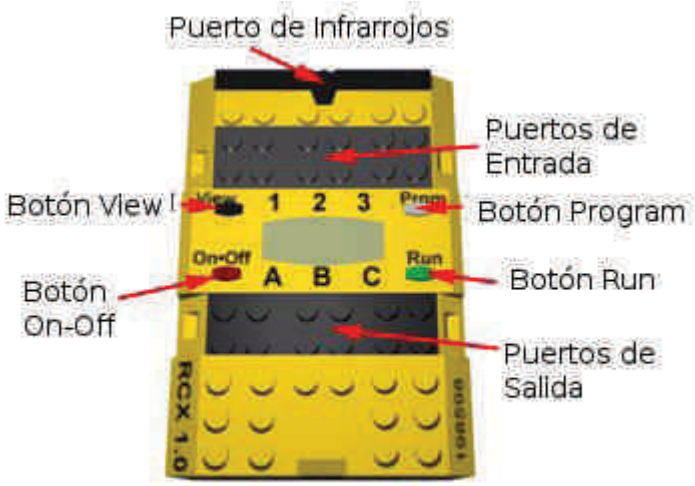


Figura 27: Partes del bloque RCX

En este proyecto se usarán dos entradas y dos salidas. Las dos entradas conectadas serán la 1 y la 3 y se conectarán a cada una un sensor de rotación, que irán acoplados a cada uno de los ejes de transmisión. El sensor 1 irá acoplado al eje izquierdo y el sensor 3 al eje derecho. Las dos salidas utilizadas serán la A y la C, cada una irá conectada a un motor (A eje izquierdo y C eje derecho). Para que el robot avance ambos motores deberán girar en sentido contrario con el motor A girando a izquierdas. Pero en cuanto a órdenes al bloque RCX ambos motores estarán avanzando, por lo que hay que comprobar la conexión y sentido de giro de los motores. Hay que recordar que dependiendo de cómo están conectados los motores pueden girar en un sentido o en el contrario con la misma orden. Al ser motores de corriente continua al variar la polaridad se cambia el sentido de giro. Para comprobar si las conexiones están realizadas correctamente se puede utilizar el *software* creado en esta fase. Una vez conectado al robot, elegimos movimiento libre. Si pulsando avanzar el robot retrocede, ambos motores están conectados al revés. Si gira hacia la izquierda, el motor C está conectado al revés. Y si gira hacia la derecha, será el motor A el que está conectado al revés.

La comunicación con el RCX se realiza mediante el puerto de infrarrojos que posee el bloque en su parte frontal, para ello es necesario conectar al computador un emisor-receptor de infrarrojos incluido en el kit del bloque RCX. Hay dos tipos de transmisores dependiendo de su conexión, los que se conectan al puerto serie estandar y, comercializados posteriormente, los que se conectan a un puerto USB. El transmisor establece un vínculo inalámbrico entre el computador y el bloque RCX. Una vez realizado el vínculo, se pueden transferir programas al bloque, mandar órdenes directas a los motores o leer directamente el estado de los sensores. Para establecer esta comunicación, el transmisor y el bloque RCX deben estar uno enfrente del otro a una distancia entre 10 y 15 cm, siendo posible la comunicación, si las condiciones de iluminación son óptimas a una distancia de hasta 30 m [24].

Figura 28: Transmisor IR por puerto serie

5.2.2.Software

-AFORGE.NET FRAMEWORK

Como se ha comentado en la introducción de este punto, se ha utilizado el marco de trabajo Aforge.Net para la comunicación y manejo del bloque RCX. Esto nos proporciona abstracción sobre cómo se comunica y las órdenes que hay que mandar realmente al bloque permitiendo centrarnos únicamente en su manejo a través del computador. Aforge.Net nos proporciona la librería Robotics, que va a ser la que vamos a utilizar en esta fase.

Pensada originalmente como una marco de trabajo para la visión por computador o la inteligencia artificial, Aforge.Net proporciona varios tipos de librerías que amplían su posible utilización como pueden ser las librerías orientadas a la robótica o al vídeo.

Precisamente la librería Robotics, nos permite la conexión, manejo, lectura o desconexión de/hacia un bloque RCX con cierta facilidad.

-MANEJO DEL ROBOT

Una vez instalado el marco de trabajo Aforge.Net, simplemente deberemos referenciar en nuestro proyecto a la librería Aforge.Robotics.Lego para poder utilizar la conectividad tanto con bloques RCX como con NXT (aunque estos últimos no han sido probados).

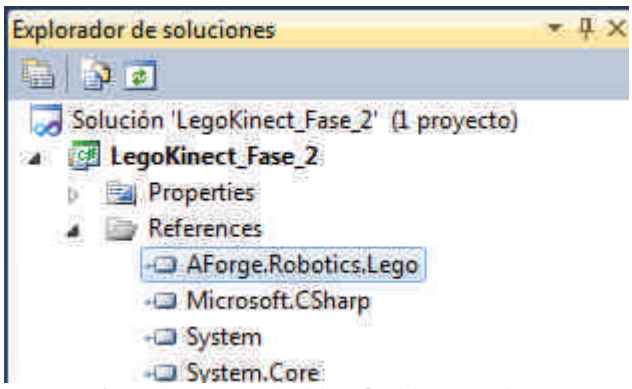


Figura 29: Referencia a la librería Aforge.Robotics.Lego

Antes de profundizar en el uso de la librería hemos de establecer una jerarquía de clases que nos permita poder extender el uso a diferentes robots en el futuro, para ello vamos a examinar el diagrama de clases resultante en esta fase.

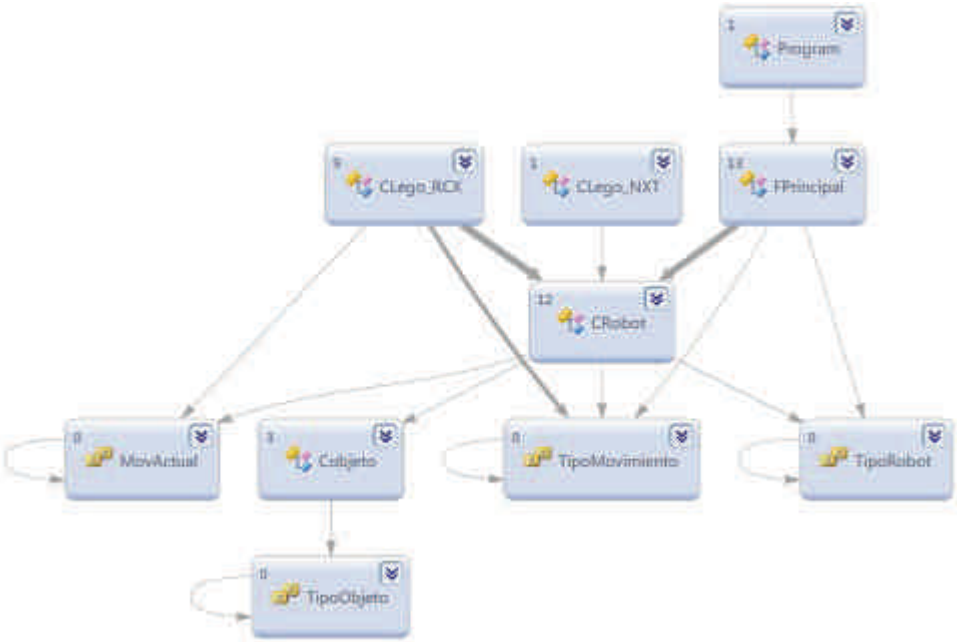


Figura 30: Diagrama de clases de la segunda fase

De la clase Cobjeto se ha hablado ya en el punto referente a la fase 1, pero es ahora cuando se ve realmente su utilidad. Además de definir los objetos tanto virtuales como reales que pueda haber en la escena, es la clase padre de Crobot, con lo que los robots heredarán las características propias de los objetos (nombre, posición, color, tipo de objeto, etc), además de añadir las suyas (movimiento que está realizando, velocidad, etc). A su vez, la clase Crobot sirve como base para desarrollar los diferentes tipos de robot que necesitemos. En este momento se encuentra implementado sólamente el tipo RCX, pero puede ser implementado con facilidad cualquier tipo de robot. Simplemente hay que definir la clase con el nombre C"tipoderobot", implementar las operaciones que vamos a explicar a continuación y añadir ese nombre "tipoderobot" al enumerado TipoRobot que observamos en el diagrama de clases. De esta manera, cuando vayamos a seleccionar un tipo de robot, el sistema nos mostrará los diferentes tipos definidos en TipoRobot y creará el objeto de la clase indicada.

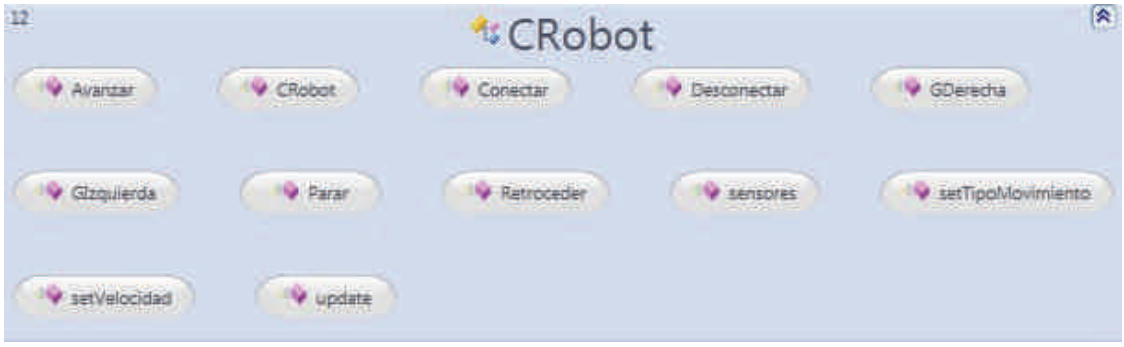


Figura 31: Métodos de la clase Crobot

Los métodos que componen la clase Crobot son los siguientes:

-Conectar: Se intenta conectar al robot

-Desconectar: Desconexión del robot

-Avanzar, Retroceder, Gderecha, Gizquierda, Parar : indican al robot el movimiento que debe realizar

-Sensores: Devuelve una cadena de caracteres con el valor leído en los sensores

-setVelocidad: Indica al robot la velocidad de movimiento

-setTipoMovimiento: Indica al robot el tipo de movimiento a realizar, en principio son dos, movimiento libre y temporizado. El movimiento libre consiste en que el movimiento permanece hasta que mandamos otro movimiento, el temporizado consiste en avanzar/retroceder durante un tiempo (por defecto 1s) y girar un número determinado de grados (por defecto aproximadamente 45º).

-*update*: Este método se usa para comprobar el movimiento que debe realizar el robot. Es el método que usarán el resto de clases para comunicarse con la clase Crobot. Recibe dos parámetros: el primero de ellos indica el movimiento actual a realizar, el segundo indica el tiempo transcurrido. El segundo parámetro sólo se utiliza en movimientos temporizados. El primer parámetro puede tener los siguientes significados: seguir en el estado actual (si su valor es NADA) o cambiar de estado. En los movimientos libres, sólo se recibe la orden de movimiento la primera vez, el resto del tiempo hasta que el robot reciba otra orden el primer parámetro será NADA. En los movimientos temporizados la orden se mantiene el tiempo necesario.

Estos son los métodos que debe reescribir cada clase de robot que queramos definir, haciéndose esto en función de las características del tipo de robot utilizado.

Dada la importancia del método *update* se va a explicar en profundidad. Primero es necesario detallar los tipos de movimientos que puede realizar el robot. Debido a las diferentes restricciones motoras que puedan tener los usuarios del software se ha decidido incluir dos tipos de movimientos al robot.

1.Movimiento Libre: consiste en que una vez que se le ha dado una orden al robot, éste continúa el movimiento hasta que se le da otra orden.

2.Movimiento Temporizado o Limitado: consiste en limitar el movimiento.Esta limitación puede ser en tiempo para los desplazamientos lineales o en grados para los rotacionales. Se ha limitado el desplazamiento lineal a 1s y las rotaciones a aproximadamente 45º.

Este es el pseudo-código del método *update* de la clase Crobot, recibe dos parámetros: movimiento y tiempo.

```
Si movimiento != nada
        Actualizar movimiento_actual;
        Dependiendo de movimiento
                Mandar orden al robot;
                Si el movimiento es lineal => Actualizar tiempo_inicial;
                Si el movimiento es rotacional => Actualizar giro_inicial;
Sino
        Si tipo_de_movimiento = temporizado
                Si el movimiento es lineal =>
                        Si ha transcurrido el tiempo =>
                                Parar;
                                Actualizar movimiento_actual;

                Si el movimiento es rotacional =>
                        Si ha girado los grados indicados =>
                                Parar;
                                Actualizar movimiento_actual;
Fin_si;
```

Las variables movimiento y movimiento_actual son del tipo enumerado MovActual, y pueden tener estos valores:

-Parado: Indica que el robot debe parar o está parado.
-Avanzar: Indica que el robot debe avanzar o está avanzando.
-Retroceder: Indica que el robot debe retorceder o está retrocediendo.
-Giro_Derecha: Indica que el robot debe girar a la derecha o está girando a la derecha.
-Giro_Izquierda: Indica que el robot debe girar a la izquierda o está girando a la izquierda.
-Nada: Indica al robot que no recibe nueva orden.

Tal y como se ve en el pseudo-código el funcionamiento es el siguiente:

Si recibimos una orden diferente a Nada, quiere decir que debemos cambiar el movimiento o iniciarlo de nuevo. Actualizamos el valor del movimiento que está realizando el robot. Comprobamos el movimiento recibido, activamos el movimiento en el robot e inicializamos el valor de la variable que nos limitará el movimiento, tiempo en el caso de Avanzar y Retroceder y grados en el caso de los giros.

Si recibimos la orden Nada, quiere decir que no hemos enviado una nueva orden al robot, por lo que deberíamos seguir en el estado que estaba con anterioridad. Esto ocurre siempre así en el caso del movimiento libre, pero en el caso del movimiento temporizado hay que realizar una comprobación. Debemos comprobar si ya ha concluido el movimiento que está realizando el robot, en caso afirmativo debemos parar el robot y cambiar el valor del movimiento actual a Parado. De esta manera, controlamos los movimientos que pueda realizar el robot.

En nuestro caso la clase se llama CLego_RCX (por lo que en TipoRobot se añade el valor Lego_RCX), contiene un objeto de la clase Aforge.Robotics.Lego.RCXBrick que encapsula las operaciones necesarias para la comunicación con el bloque RCX. Su implementación es muy sencilla. Para conectar, simplemente debemos comprobar que la operación Connect de la clase RCXBrick nos devuelve *true*, en caso contrario la conexión no se ha podido realizar. Para los movimientos, simplemente debemos indicar la velocidad de giro, la dirección en la que deben girar y encender los motores (no comprobamos si están parados o no, simplemente los ponemos en marcha, si ya lo estaban siguen estándolo).

```
        public override bool Avanzar()
        {
            try
            {
                // ponemos la potencia de los motores A y C la velocidad
indicada
                // los ponemos en modo avance y ponemos en marcha
                robot.SetMotorPower(RCXBrick.Motor.AC, velocidad);
                robot.SetMotorDirection(RCXBrick.Motor.AC, true);
                robot.SetMotorOn(RCXBrick.Motor.AC, true);
            }
            catch (Exception exc)
            {
                return false;
            }
            return true;
        }
```

Figura 32: Implementación del método Avanzar en la clase CLego_RCX

Como se puede observar, es muy sencilla la comunicación entre el computador y el bloque RCX. El método más complicado sería el de *Update*, ya que, en función del movimiento que estamos realizando (siempre y cuando el tipo de movimiento sea temporizado) debemos comprobar el tiempo transcurrido desde que hemos iniciado el movimiento o los grados girados desde que hemos empezado a girar. Para ello la clase Crobot tiene los atributos giroInicial y tiempoInicial, que indicarían tanto el tiempo como los grados indicados al inicio del movimiento, pudiendo, a partir de estos valores comprobar si el tiempo o los grados se han cumplido cuando se ejecuta el método *Update*.

5.2.3.Interfaz de Usuario

La interfaz de usuario es muy simple. Primeramente hemos de elegir en una lista desplegable el tipo de robot que vamos a manejar. Seguidamente intentará conectar con éste. Si lo consigue aparecerá una segunda lista desplegable donde podremos elegir el tipo de movimiento que queremos realizar. En un principio son dos movimientos, libre y temporizado. Si elegimos el movimiento libre nos aparecerá una cruceta de botones para controlar los cuatro movimientos posibles más el de parada. Si, en cambio, elegimos el movimiento temporizado, aparecerá un sólo botón que irá cambiando de apariencia en función del movimiento que se pueda realizar con el robot, siguiendo una secuencia cíclica que recorre todos los movimientos posibles. Hay que recordar que en el movimiento libre, si no enviamos otra orden al robot, éste seguirá realizando el movimiento designado.

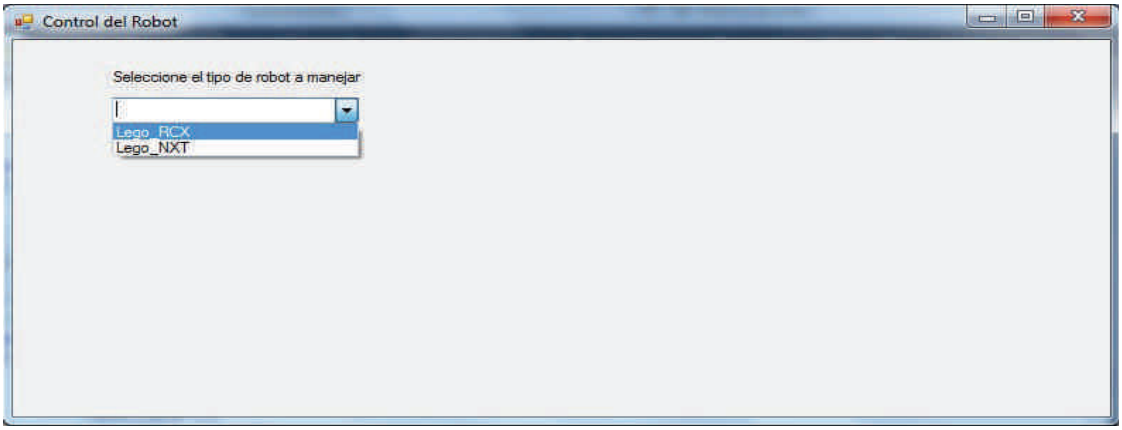


Figura 33: Paso 1.Selección del tipo de robot dentro de los disponibles

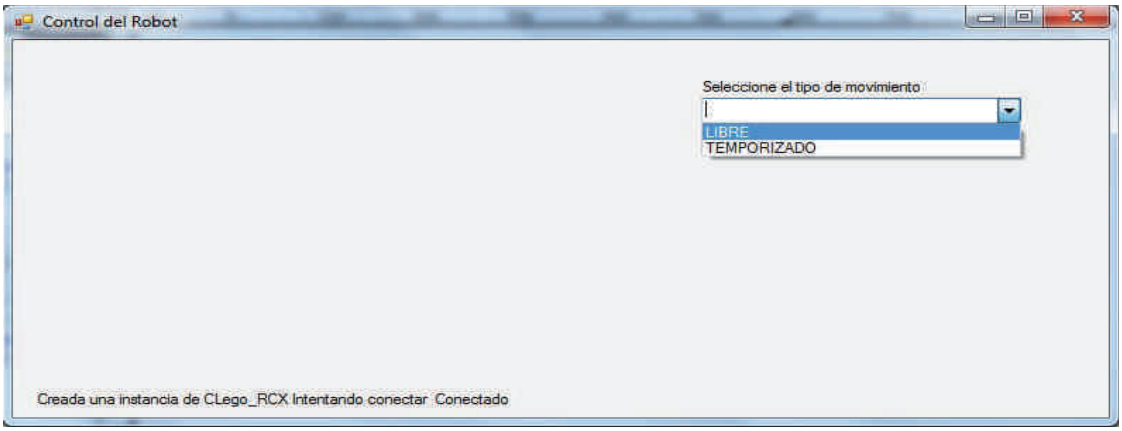


Figura 34: Paso 2. Tras informarnos que se ha conectado al robot nos permite seleccionar el tipo de movimiento

Figura 35: Paso 3. Tras seleccionar movimiento libre se nos muestra una cruceta de botones con los movimientos posibles

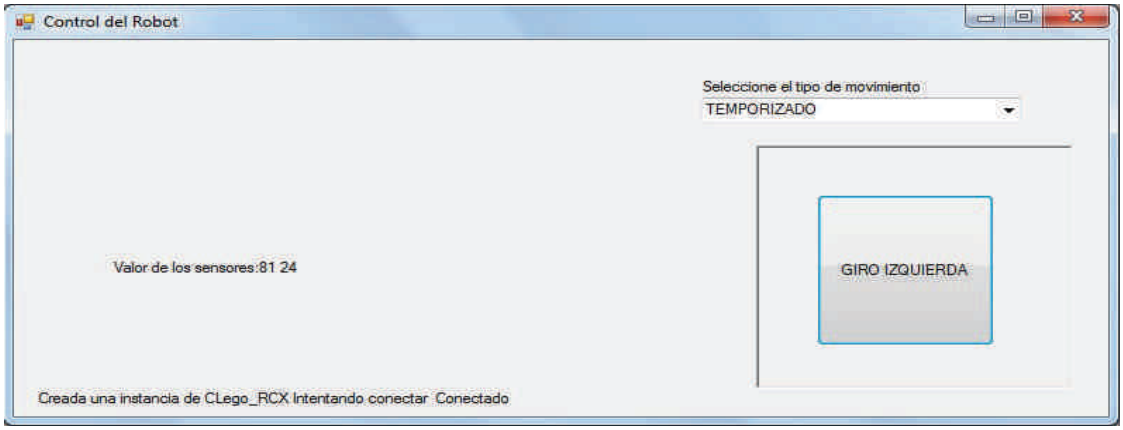


Figura 36: Paso 4. Tras seleccionar movimiento temporizado se nos muestra un botón que va variando cada segundo con los movimientos disponibles

Para salir de la aplicación simplemente hay que cerrar la ventana de Control del Robot.

5.3.Tercera Fase

En esta tercera fase, lo que se pretende es integrar las dos primeras fases en un mismo sistema, teniendo a la vez control sobre el robot y detección del mismo. Dada la naturaleza modular que se le ha querido dar desde un primer momento al sistema, esta fase no ha supuesto una mayor complicación a lo hora de implementarla. Además, se pretende probar el funcionamiento del sistema añadiendo un proyector que será el encargado de proyectar sobre el suelo la imagen del escenario. La idea es que el proyector y el sensor Kinect estén situados en altura y apuntando al suelo. Los ejes de captura y proyección deben ser perpendiculares a la superficie de proyección, para que la imagen proyectada y la captada no sufran variaciones en la orientación y haya que realizar un calibrado sensor-proyector en profundidad, sino que sirva uno más sencillo.

5.3.1.Hardware – Proyector Epson EMP-X5

El proyector utilizado para realizar las pruebas ha sido un Epson EMP-X5, cuyas características técnicas más útiles se detallan a continuación [25].

-Resolución: XGA 1024x768
-Color: 24 *bits*, 16.7 millones de colores
-Tamaño de pantalla proyectada: 75cm-7.5m
-Distancia de proyección: 0.84-10.42m

Debido a la no disponibilidad de un soporte adecuado, las pruebas se han realizado proyectando sobre una superficie vertical.

Figura 37: Proyector Epson EMP-X5. Fuente: http://www.techsmart.co.za

5.3.2.Software

Para la integración de las implementaciones de las dos fases anteriores los cambios realizados en las clases han sido mínimos. Las mayores modificaciones han tenido que realizarse en la interfaz de usuario, en los métodos *Update* y *Draw* de la clase CLegoKinect, en la clase Cescenario y en la clase CKinect. La interfaz de usuario se comentará en el punto correspondiente.

Para explicar la clase Cescenario, primero se debe explicar el concepto de escenario. Un escenario está formado por:

-una imagen que representa la escena
-un robot
-un conjunto de objetos reales o virtuales

Un escenario representa el entorno donde se va a desplazar e interactuar el robot. Debe mantener la estructura de datos necesaria para poder gestionar el conjunto de objetos. En este desarrollo se limita a ser el contenedor de la imagen proyectada y del robot que vamos a manejar. Se encarga además de mantener actualizada la posición del robot en cada *frame*.

El método más significativo de la clase es *update*. Este método toma dos parámetros, el primero es el movimiento a realizar por el robot que tiene asignado el escenario. El segundo es el tiempo transcurrido. *Update* se encarga:

-De actualizar el movimiento del robot, llamando al método *update* del objeto robot.
-De localizar la posición del robot, llamando al método localizarColor del objeto detector asignado. El color a localizar vendrá definido por el atributo color del objeto robot.

Figura 38: La clase Cescenario

En la clase CKinect se ha añadido un método de calibrado, este método se encarga de encontrar la correspondencia entre la imagen capturada por el sensor Kinect y la imagen proyectada por el proyector. Para ello se proyectará la imagen de un tablero de ajedrez de 4x4 casillas al seleccionar calibrar. Tras la espera de un tiempo de estabilización realizaremos una captura con el sensor Kinect. Mediante la operación de la librería OpenCVSharp *FindChessboardCorners*, obtendremos la posición de los vértices de los cuadrados intermedios en la captura realizada. Hallando la correspondencia entre los puntos hallados y la posición que ocupan en la imagen proyectada, se puede obtener la matriz de transformación. Esta matriz nos permitirá conocer la posición del robot en la imagen proyectada a partir de su posición en la imagen capturada. Para que esto funcione correctamente, proyector y sensor deben apuntar perpendicularmente a la superficie de proyección.

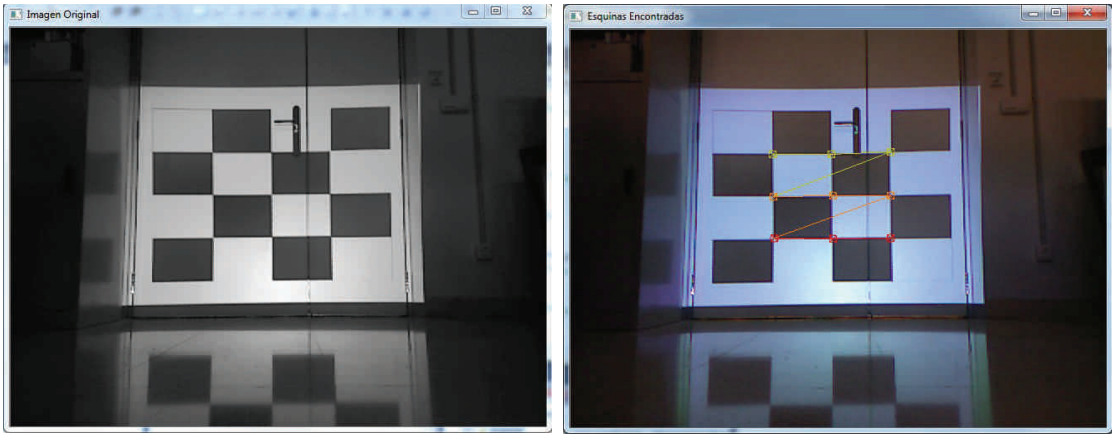


Figura 39: Calibrado. La imagen situada a la izquierda representa la imagen proyectada en escala de grises. La imagen situada a la derecha representa la imagen capturada con los vértices internos del tablero detectados.

Estos cambios implican que haya un control del estado en el que se encuentra la aplicación. Se ha añadido a la clase CLegoKinect un atributo de estado. El valor de este estado influye en la toma de decisiones en los métodos *Update* y *Draw.*

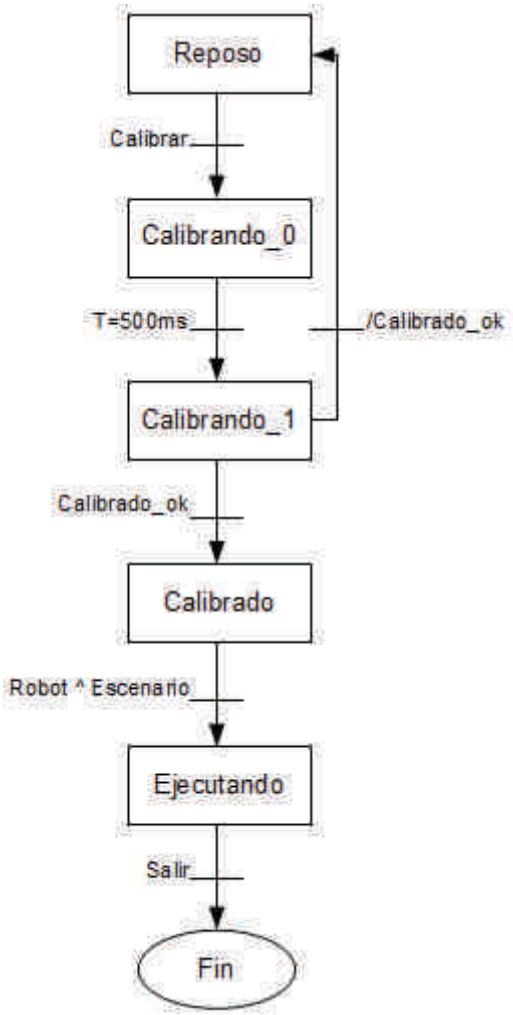


Figura 40. Diagrama de estados de la aplicación

Partiendo del estado de Reposo, una vez seleccionada la calibración pasamos al estado Calibrando_0. En este estado se proyecta la imagen de calibración y se espera un tiempo de estabilización para que el sensor obtenga la imagen correcta. Tras pasar este tiempo se pasa al estado Calibrando_1. En este estado se realiza la búsqueda de los puntos del tablero proyectado, si los puntos se encuentran correctamente pasaremos al estado Calibrado, si no se encuentran volvemos a Reposo. Desde el estado Calibrado se pasa al estado Ejecutando cuando se ha conectado con un robot y se ha inicializado el escenario. Si salimos del programa finalizaremos la ejecución.

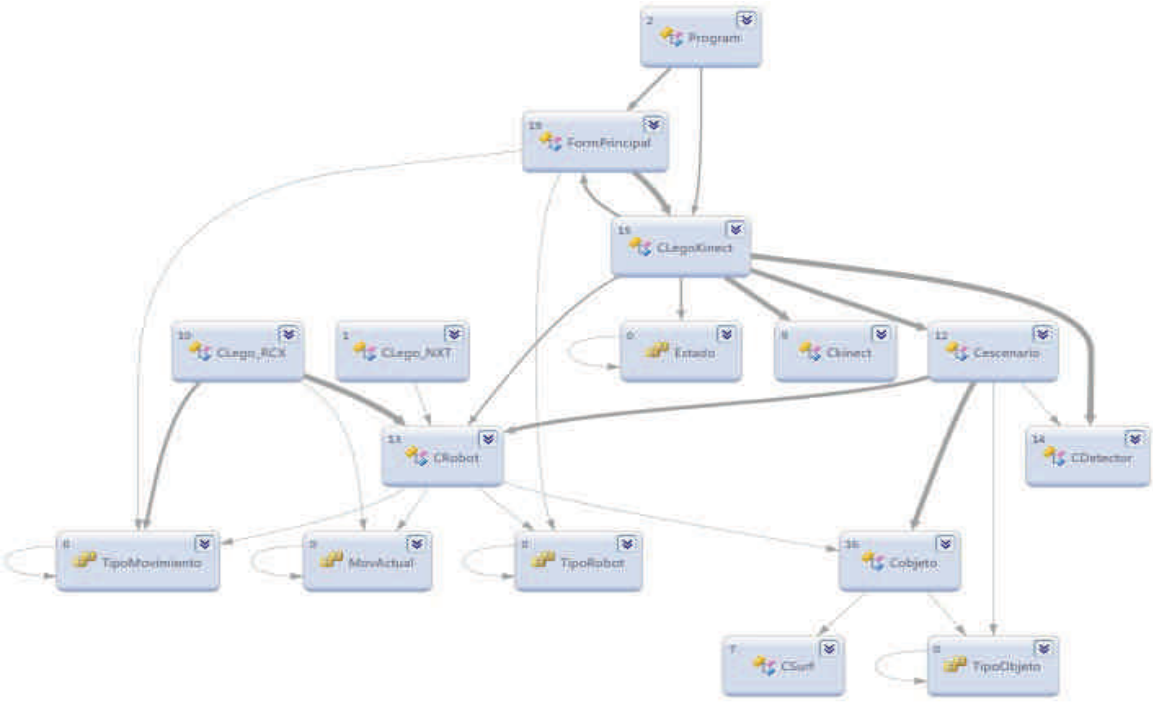


Figura 41. Diagrama de clases de la fase 3

En esta fase se han integrado la detección de objetos, los movimientos del robot y la proyección de imágenes. Es el momento de ver el funcionamiento global del sistema.

Partimos de la clase Program que es la encargada de crear las instancias del formulario que utilizaremos como GUI(Interfaz Gráfica de Usuario) y del objeto de la clase CLegoKinect que gobernará el sistema. El objeto LegoKinect prepara el marco XNA y crea las instancias de los objetos necesarios. Esto incluye la creación de un objeto de la clase Ckinect. Este objeto comprueba la existencia de un sensor Kinect, si éste no existe se procederá a salir de la aplicación. El objeto LegoKinect crea una instancia de un objeto detector y se la asigna al objeto escenario. De esta manera cuando estemos ejecutando, el escenario ya sabe qué instancia es la que se encarga de la detección de objetos.

Una vez que está preparado el marco XNA, tal y como aparece en la Figura 6, pasamos al método *Initialize*, es aquí donde llamaremos al método inicializar del objeto Kinect. Este método se encarga de la inicialización y subscripción del sensor a los eventos tal y como vimos en el punto **5.1.2.2**. Si hay algún error en la inicialización se mostrará el mensaje y se procederá a salir de la aplicación.

En *LoadContent* simplemente establecemos el valor del objeto que necesitamos para realizar las operaciones de dibujo con el marco XNA.

Ahora ya entramos en el bucle *Update-Draw* típico de XNA. En el método *Update* comprobaremos el estado de la aplicación. Hay que tener en cuenta que los cambios de estado de Reposo a Calibrando_0 y de Calibrado a Ejecutando los realizamos en base a la entrada recibida desde el GUI. En el caso de estar en el estado Calibrando_0 comprobaremos si ha transcurrido el tiempo de estabilización, si ha sido así pasaremos al estado Calibrando_1. En este estado comprobaremos si hemos realizado ya la calibración, ya que sólamente hay que realizarla una vez.

Si la calibración no se ha hecho, procedemos a llamar al método calibración del objeto Kinect. Si la calibración es correcta pasaremos al estado Calibrado, si no lo es volveremos al estado Reposo. Del estado Calibrado sólamente podemos pasar al estado Ejecutando si el escenario se inicializa y se consigue comunicación con el robot que queremos manejar. Si esto no ocurre así seguiremos en el estado Calibrado. En el estado Ejecutando actualizamos las imágenes del detector con las capturadas por el sensor y llamamos al método *update* del objeto Escenario. Este método, como se ha comentado anteriormente, se encarga de actualizar el estado del robot y de llamar al método LocalizarColor del objeto Detector que tiene asignado para poder obtener la posición del robot.

En cuanto al método *Draw*, no diferencia entre los estados Calibrando_0 o 1, en los dos manda mostrar la imagen de calibración en el proyector. En el caso de que no haya proyector no se realizará la calibración. En el estado de Reposo muestra la imagen capturada por el sensor. En el estado Ejecutando, si el escenario está inicializado:

-Imagen a Proyectar:

Si el escenario tiene una imagen asignada, muestra la imagen en la ventana destinada a ello. Esta imagen será la proyectada si el proyector está conectado. Si el escenario no tiene imagen asignada, se mostrará la captada por el sensor.

-Imagen mostrada en el GUI:

Si el detector está inicializado y tenemos una imagen correcta en el atributo que representa el *frame* en color del detector, obtenemos la posición del robot, la dibujamos en la imagen y mostramos la imagen. Hay que puntualizar que el escenario obtiene las imágenes tomadas por el sensor a través del objeto detector. El detector es el que se encarga de recoger las imágenes cuando las necesita.

5.3.3.Interfaz de Usuario

Una vez se inicia la aplicación, presenta una interfaz de usuario con tres menús, de los cuales dos están inhabilitados y un panel mostrando la imagen captada por el sensor.

El primero de los menús sirve para iniciar el calibrado de la cámara y el proyector. Una vez realizada la calibración se habilitarán los otros dos menús. Estos dos menús se denominan Robot y Escenario. El primero permite seleccionar el tipo de robot a manejar y el tipo de movimiento. El segundo permite cargar una imagen de escenario e inicializarlo. Se comprobará si hay seleccionado algún robot, en caso de no ser así se mostrará una caja de diálogo indicando el problema. En todo momento podemos inicilizar el escenario cuando deseemos.

Tras la inicialización el sistema ya está preparado para la detección del robot, pero nos falta indicar el tipo de movimiento del robot. Esto se puede hacer desde el menú Robot. Una vez indicado el tipo de movimiento se mostrará un panel con los botones correspondientes. El tipo de movimiento se puede cambiar en cualquier momento de la ejecución.

Para salir de la aplicación bastará con cerrar la ventana de ejecución.

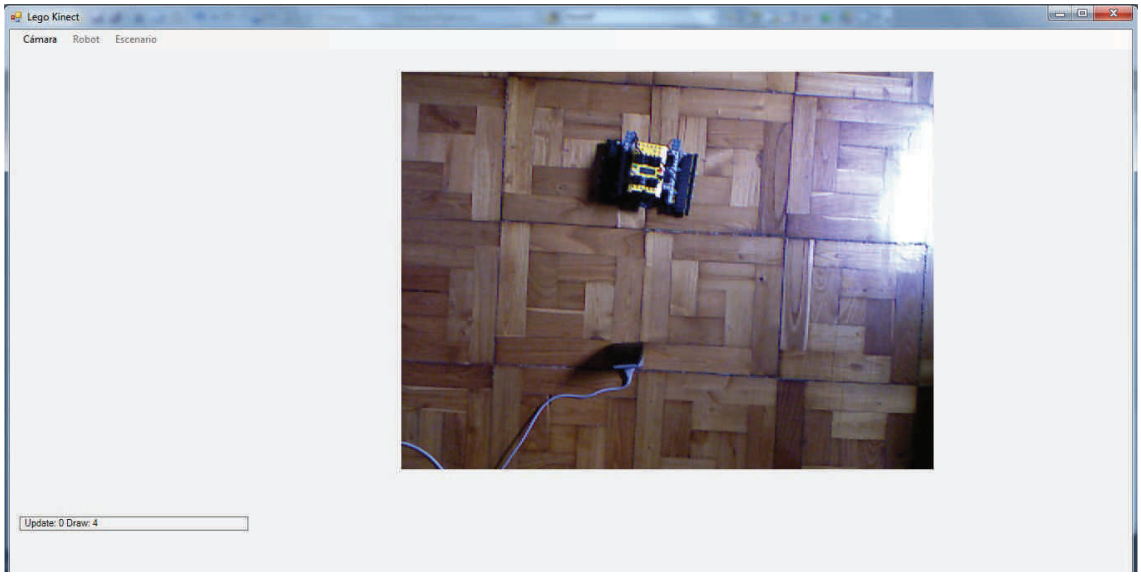


Figura 42. Interfaz inicial de la aplicación

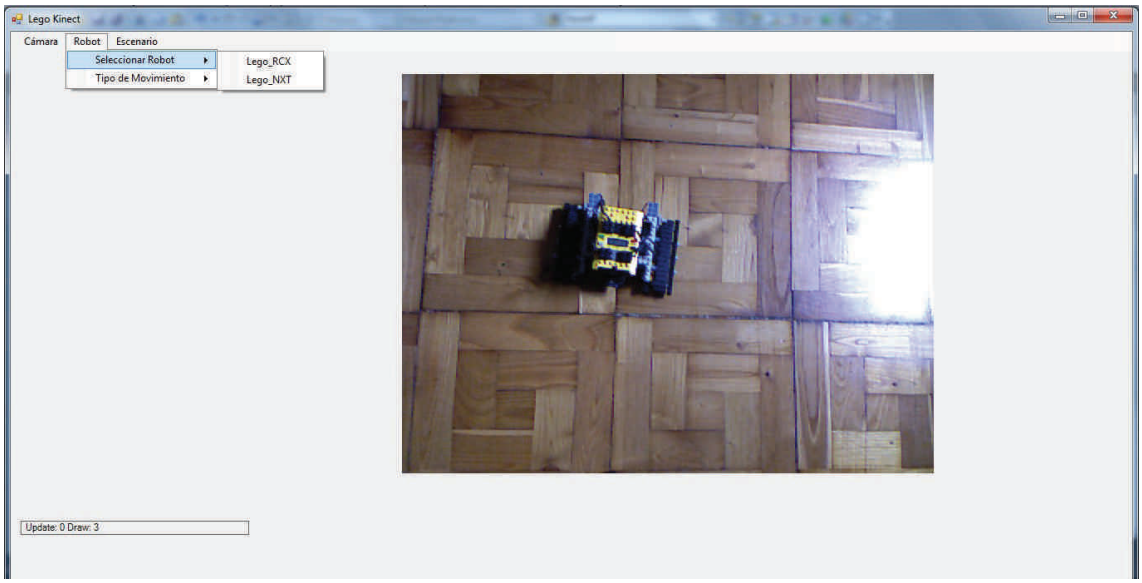


Figura 43. Tras calibrar la cámara seleccionamos el tipo de robot

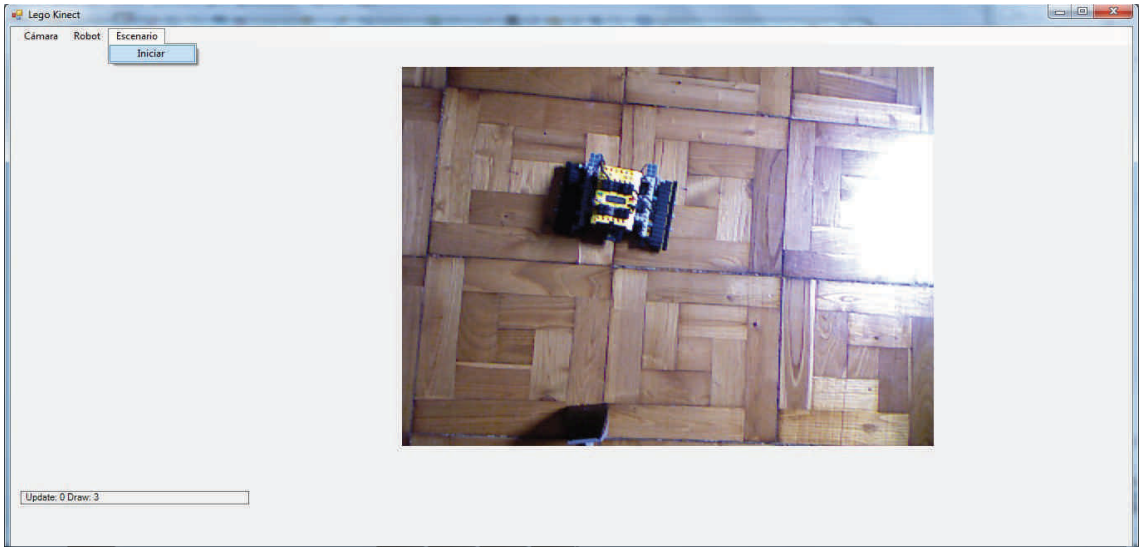


Figura 44. Inicializar el escenario

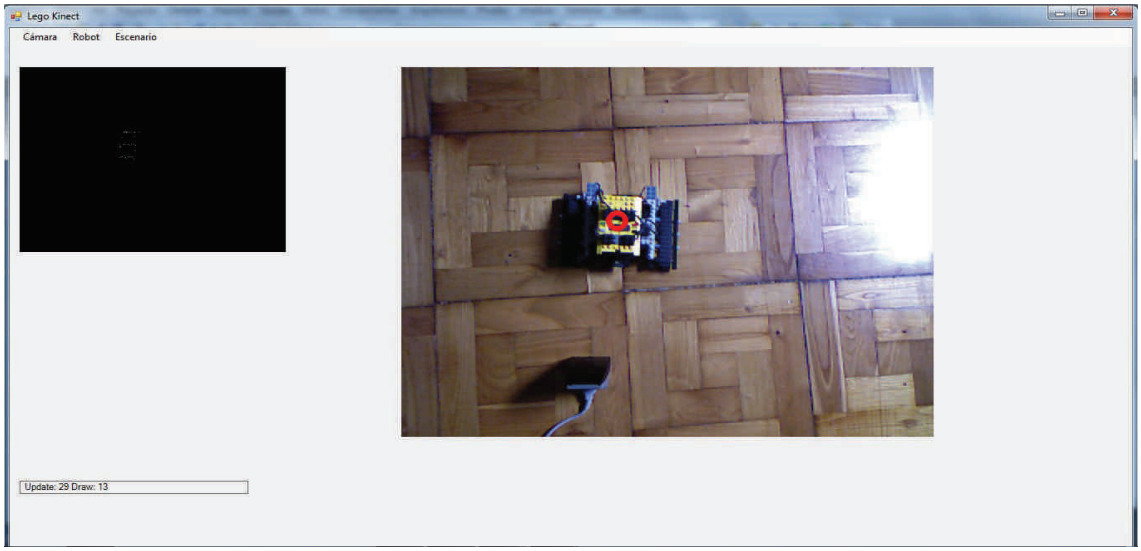


Figura 45. Una vez conectado al robot e inicializado el escenario comienza la detección del objeto

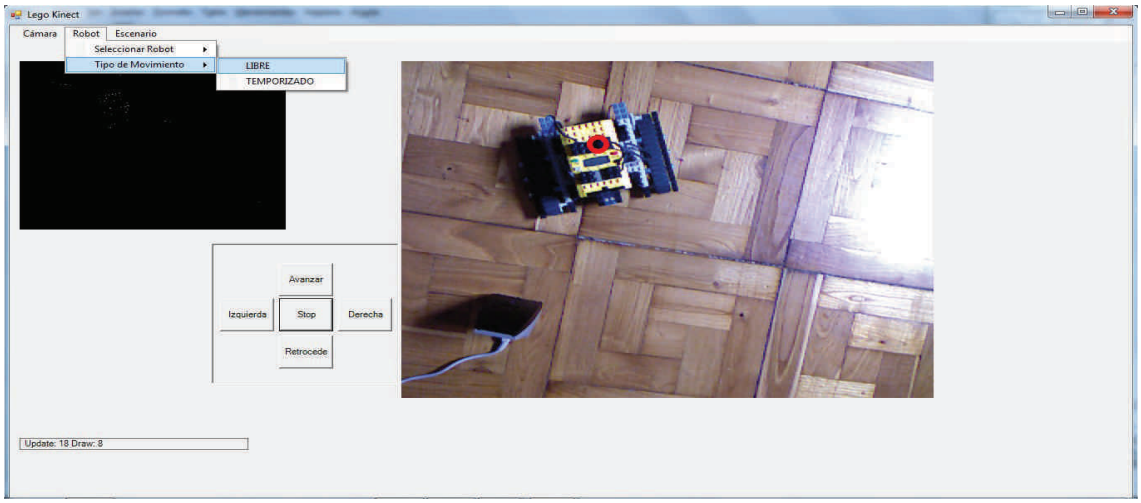


Figura 46. Interfaz de la aplicación tras seleccionar el tipo de movimiento

6.PRUEBAS, PROTOTIPOS Y LIMITACIONES

Cada una de las fases de desarrollo del proyecto se corresponde con las pruebas de funcionamiento realizadas. En la primera fase se prueban los métodos de detección *Color-Tracking* y SURF para decidir cuál de ellos es más apropiado para seguir con el desarrollo. En la segunda fase se prueba el funcionamiento de la librería de manejo del robot para comprobar si cumple las necesidades del proyecto. Por último, en la tercera fase se prueba el sistema junto con un proyector para comprobar si su rendimiento y funcionamiento es el deseado.

En la primera fase se ponen a prueba ambos sistemas de detección de objetos. Para comprobar su rendimiento en tiempo hay una etiqueta en la propia interfaz que nos indica el tiempo que consume en los métodos *Update* y *Draw*. En el caso de SURF sólamente se actualiza el *Update* cada vez que realiza el cálculo de descriptores de la imagen, una vez cada 15 *frames*.

Por una parte, SURF nos delimita el objeto cuando es detectado. Podríamos además detectar su orientación en función de los puntos de interés detectados. Por contra, su rendimiento es muy pobre. La localización del objeto es especialmente lenta. Entendemos por localización el cálculo de los descriptores de la escena y la búsqueda del objeto en ésta. Presenta demasiados falsos negativos y errores de localización. Incluso en ocasiones en los que la posición del objeto no varía lo detecta un *frame* correctamente y al siguiente no.

En las pruebas realizadas, para un umbral de 500 el tiempo medio de localización del objeto en 8 mediciones ha sido de 756ms. Para un umbral de 800 el tiempo medio ha sido de 629ms y para un umbral de 1500 de 265ms. En ninguno de estos casos se tiene en cuenta si la localización ha sido correcta o no. Son datos que constatan que este método no es adecuado para los fines que se buscan en este desarrollo. Hay que puntualizar que estos tiempos varían mucho en función del entorno en el que se han desarrollado las pruebas. Entornos con pocas variaciones de color producen mejores resultados (suelos lisos sin variaciones de tonalidad). Pero, en todo caso, estaríamos hablando de tiempos sobre los 170ms para un umbral de 700. Estos tiempos siguen siendo inasumibles para esta aplicación.

El método *Color-Tracking* presenta unos tiempos de localización de objetos más acordes a las necesidades de la aplicación. Estamos hablando del orden de 30ms para la ejecución del método *Update*. Sumado al tiempo del método *Draw* obtenemos del orden de 60-70ms para la ejecución de cada *frame*. Esto nos da del orden de 14-16 *frames* por segundo. No es una tasa muy alta pero tampoco se han hecho optimizaciones de código.

El mayor problema de este método es la sensibilidad a la iluminación que tiene. Diferentes ambientes lumínicos nos proporcionan diferentes resultados. Los mejores resultados se han obtenido estando el robot sobre superficies blancas (aunque no sean uniformes) e iluminación por medio de lámparas fluorescentes blancas. La luz que emiten este tipo de lámparas contaminan en menor medida los colores. De todas formas, este problema podría solucionarse añadiendo una configuración manual del color. De esta manera el usuario podría configurar en tiempo de ejecución el color del robot para adecuarlo a las características lumínicas del entorno.

A esta sensibilidad hay que añadirle el hecho de que el proyector añade luz a la escena y contamina los colores. Este hecho acentúa el problema. Además, hay que colocar el proyector y el sensor de manera que los ejes de proyección y captura sean paralelos. De esta manera, la calibración proyector-cámara es más sencilla. Ambos ejes deben ser perpendiculares a la superficie donde se proyecta la imagen, para que la correspondencia de puntos entre la superficie proyectada y la capturada sea lo más lineal posible.

Otro factor que influye en el funcionamiento del sistema es la distancia a la cual se coloca el sensor Kinect. Se ha comprobado que a distancias superiores a 2,25m la porción de imagen que ocupa el robot es muy pequeña. Por esta razón es muy difícil su detección. Es debido a que no se encuentran un número suficiente de *pixels* que sean identificados como el color a detectar. Las distancias óptimas serían entre 1,5m y 2m. No es aconsejable que esté demasiado cerca para poder captar la totalidad de la imagen proyectada en condiciones.

En cuanto al manejo del robot la mayor limitación que existe es debida al marco de trabajo elegido para la implementación y a la propia configuración del robot. La actualización de los movimientos del robot se realiza cada vez que se ejecuta el método *update* del robot. Este método es llamado a su vez por el método *update* del escenario, que recibe la llamada del método *Update* de la clase CLegoKinect. Esto hace que se ejecute la actualización cada, aproximadamente, 60ms. En movimiento libre no supone mayor problema, ya que no es un tiempo considerable desde que se emite la orden hasta que la recibe el robot. Pero cuando hablamos de movimientos temporizados el problema

aparece. Y aparece sobre todo en los giros. La medición en grados del giro no es exacta. No se ha encontrado una relación calculable exacta entre los valores del sensor de rotación y los grados girados por el robot, se ha hecho aproximandamente. Este hecho sumado a los intervalos que transcurren entre lecturas de los sensores hacen que haya variaciones en los giros. Además, sólamente podemos controlar un único robot.

Otro problema que encontramos es que el tiempo de adquisición de los datos de los sensores es bastante alto (del orden de 100ms). Esto supone retardos en la actualización de la imagen a mostrar, ya que el robot sigue moviéndose pero la imagen no se actualiza.

Hay que tener también en cuenta que la configuración del robot está limitada. Es una configuración cerrada que comprende el uso de dos motores, cada uno conectado a uno de los ejes de tracción y dos sensores de rotación.

7.CONCLUSIONES Y TRABAJO FUTURO

Este trabajo pretende sentar las bases del desarrollo de un sistema de ayuda a la rehabilitación de niños con restricciones motoras. Para ello se ha trabajado con diferentes métodos de detección de objetos buscando el más adecuado para la aplicación. Se han implementado modos de manejo del robot que se puedan asemejar a sus contrapartidas físicas existentes. Y se ha preparado una estructura de clases fácilmente ampliable para mejoras futuras.

A pesar de las limitaciones del sistema, puede ser totalmente válido para su aplicación introduciendo algunas mejoras que impliquen una detección más precisa y fiable y una tasa de *frames* por segundo superior. De esta manera, su aplicación en éste y otros campos será completamente viable y funcional.

No se ha realizado ninguna prueba en un entorno real, para ello hubiera sido necesario un soporte especial para sujetar tanto el proyector como el sensor.

En vista de las limitaciones que posee el sistema y de las pruebas realizadas, se pueden realizar diferentes mejoras y ampliaciones.

-Detectar la orientación del robot.

-Utilizar también la cámara de profundidad para la ayuda a la detección.

-Añadir la posibilidad de configurar los colores a detectar.

-Manejo del robot con mandos reales.

-Optimizar el rendimiento.

-Añadir manejo del robot por voz.

-Implementar una aplicación de autoría para la creación de diferentes escenarios de juego.

-Posibilidad de añadir objetivos.

-Añadir movimiento asistido.

-Implementación y prueba con otros robots.

-Posibilidad de control de más de un robot a la vez.

8.BIBLIOGRAFÍA

[1] M.M.K. Oishi et al., Design and Use of Assistive Technology: Social, Technical and Economic Chalenges, DOI 10.1007/978-1-4419703-2_4 ©Springer Science+Business Media, LLC 2010

[2]http://es.wikipedia.org/wiki/Kinect

[3]http://www.mydigitallife.info/kinect-sensor-for-xbox-features-and-prices/

[4]http://world.einnews.com/pr_news/56709028/microsoft-fully-unveils-kinect-for-xbox-360-controller-free-game-devicehttp://www.t3.com/features/exclusive-how-does-microsoft-xbox-kinect-work

[5]http://www.pensamientoscomputables.com/entrada/Kinect/microfono/multiarray/como-funciona/xbox-360

[6]http://www.wired.com/magazine/2011/06/mf_kinect/all/1

[7]http://www.t3.com/features/exclusive-how-does-microsoft-xbox-kinect-work

[8]http://es.wikipedia.org/wiki/XNA

[9]http://xnacommunity.codeplex.com/wikipage?title=XNAEditor&referringTitle=Home&ProjectName=xnacommunity

[10]http://code.opencv.org/projects/OpenCV/wiki/WikiStart

[11]http://code.google.com/p/opencvsharp/

[12] Paul Viola y Michael Jones, *Rapid Object Detection using a Boosted Cascade of Simple Features*, 2001

http://research.microsoft.com/en-us/um/people/viola/Pubs/Detect/violaJones_CVPR2001.pdf

[13]http://note.sonots.com/SciSoftware/haartraining.html

[14] Herbert Bay, Tinne Tuytelaars y Luc Van Gool, SURF:*Speeded Up Robust Features*, 2006

http://www.vision.ee.ethz.ch/~surf/eccv06.pdf

[15]http://www.ni.com/white-paper/3470/en

[16]http://www.scholarpedia.org/article/K-nearest_neighbor

[17]http://es.wikipedia.org/wiki/Modelo_de_color

[18]http://aprende.colorotate.org/color-models.html

[19]http://www.desarrolloweb.com/articulos/1483.php

[20]http://homepages.inf.ed.ac.uk/rbf/HIPR2/erode.htm

[21]http://www.gnu.org/licenses/lgpl.html

[22]http://es.wikipedia.org/wiki/Lego_Mindstorms

[23]http://www.lisha.ufsc.br/teaching/ish/processors/h8/overview.pdf

[24]http://cfievalladolid2.net/tecno/cyr_01/robotica/lego.htm

[25]http://www.epson.co.uk/Projectors/Epson-EMP-X5

9.ANEXOS

9.1.Contenido del CD

En el CD adjunto se proporcionan los archivos necesarios para la ejecución e instalación del software, así como los fuentes de las diferentes fases.

La estructura del CD es la siguiente:

-Instalación: en este directorio se encuentran los archivos necesarios para la instalación del *software*.

xnafx40_redist.msi: *runtime* redistribuible de XNA 4.0

KinectSDK-v1.0-beta2-x*.msi: SDK's de Kinect en versiones de 32 o 64 *bit*

-Programación: en este directorio se encuentran los archivos necesarios para poder seguir desarrollando el *software*.

en_XNA_Game_Studio_4.exe: XNA Game Studio 4.0, es necesario tener instalado Visual Studio con soporte para C#

KinectSDK-v1.0-beta2-x*.msi: SDK's de Kinect en versiones de 32 o 64 *bit*

AForge.NET Framework-2.2.3.exe: Marco de trabajo Aforge.Net

OpenCV-2.3.0-win-superpack.exe: Versión 2.3 de OpenCV

OpenCvSharp-2.3-x*-20111229.zip: Librerías de OpenCvSharp, el número de versión debe coincidir con la versión utilizada de OpenCV. En este caso la 2.3. Se proporciona tanto la versión de 32 como la de 64 *bit*.

-Ejecutables: en este directorio se encuentran los ejecutables y dll necesarias en las diferentes fases del proyecto.

LegoKinect-Fase 1: Ejecutable correspondientes a la fase 1, detección de objetos mediante *Color-Tracking* o SURF

LegoKinect-Fase 2: Ejecutable correspondientes a la fase 2, manejo de un robot Lego Mindstorm RCX

LegoKinect: Ejecutable correspondientes a la fase 3, desarrollo final

9.2.Instrucciones de instalación

En el directorio Instalación situado en el raíz del CD se encuentran los archivos necesarios para poder ejecutar el *software* en cualquier equipo. Los pasos a seguir son los siguientes:

1.Instalar el *runtime* de XNA.
2.Instalar la versión correspondiente (32 o 64 *bit*) del SDK de Kinect.
3.En principio ya se puede ejecutar el programa haciendo doble click sobre el ejecutable ya que todas las dll's necesarias están en el directorio del propio ejecutable.

Una aclaración sobre las dll que se encuentran añadidas al proyecto, las que comienzan por opencv_ y tbb corresponden a las librerías de OpenCV, si el nombre de la librería termina en d es la versión para *debugging*. La librería OpenCvSharpExtern.dll es la librería de OpenCvSharp necesaria para traducir las llamadas de OpenCvSharp a la librería que corresponda de OpenCV. Por último, las librerías GhostApi, PbkComm32 y PbkUsbPort son necesarias para la comunicación con el robot Lego Mindstorms RCX.

10. TABLA DE FIGURAS