4. DESARROLLO DEL PROCESO DE SIMULACIÓN VIRTUAL 3D.

Ha llegado el momento de explicar cómo se implementa la simulación virtual para conseguir un efecto visual en la pantalla del ordenador de forma que el operario pueda corroborar que la trayectoria y los movimientos que va a efectuar el robot real son satisfactorios.

En este apartado también explicaremos cómo se relacionan las dos aplicaciones 'Corte3D.exe' y 'Rx90.exe', ya que son dos proyectos distintos desarrollados por separado con Visual C++ que se comunican entre sí mediante ficheros de texto con los datos importantes del procesamiento realizado en el primero de ellos, tales como el nombre de la pieza a cargar, la posición de dicha pieza en el espacio de trabajo del robot real y el listado de puntos que conforman la trayectoria de corte.

Existe la posibilidad de utilizar el simulador del Rx-90 independientemente de la otra aplicación, cargar ficheros '.apt' y calcular también toda la trayectoria, pero no es ésa la finalidad del proyecto. El objetivo es poder efectuar cortes reales con total seguridad en la bondad de los resultados antes de ejecutar el proceso de corte.

4.1.INTRODUCCIÓN

4.1.1. Introducción a DirectX-3D.

Las aplicaciones en 3D se programaron exclusivamente para DOS durante muchos años, el motivo para no usar Windows era el pésimo rendimiento que éste ofrecía para manejar gráficos y sonido.

En 1995 Microsoft introdujo en el mercado DirectX, un conjunto de herramientas que permitía a los programadores manejar gráficos y sonidos con el mismo rendimiento que DOS, y con la ventaja de poder despreocuparse de los *drivers* del sistema (puesto que cada fabricante provee sus propios *drivers* a Windows).

Una aplicación hecha con DirectX funciona en cualquier PC aunque no se tenga el último modelo de tarjeta, puesto que DirectX automáticamente emula esas características mediante software de forma que al menos el programa funcione.

En este proyecto el diseño del robot no se hace directamente en DirectX, sino que se carga de un archivo ".x" sus elementos y de otro archivo (también ".x") la pieza que se modeló en CATIA.

El trabajo desarrollado en la simulación virtual del movimiento del manipulador se basa en la labor realizada por un compañero en un proyecto anterior [6], cuya finalidad era, mediante un sistema de comunicación entre el controlador CS7 del Rx-90 y el PC (aprovechado para la comunicación por el puerto serie que se implementó en [4]), ver en la pantalla el movimiento que en tiempo real estaba realizando el robot, esto es, leía las posiciones que iba tomando el manipulador, las procesaba y las representaba mediante un interfaz gráfico con una imagen virtual del robot.

En este caso, se utiliza la representación virtual con una mayor ambición al predecir el movimiento real del robot y permitir a un operario visualizar el corte industrial de una pieza antes de que éste se produzca en la realidad.

4.1.2. Diseño del robot manipulador en 3D Studio Max.

Como hemos comentado anteriormente, en este proyecto hemos partido de un diseño del Rx-90 realizado en [6]. Aun así, ha sido necesario corregir algunos problemas de escala entre los elementos del robot para ajustar las dimensiones a la

realidad e incorporar el diseño de la fresa neumática como efector final en lugar de la pinza.

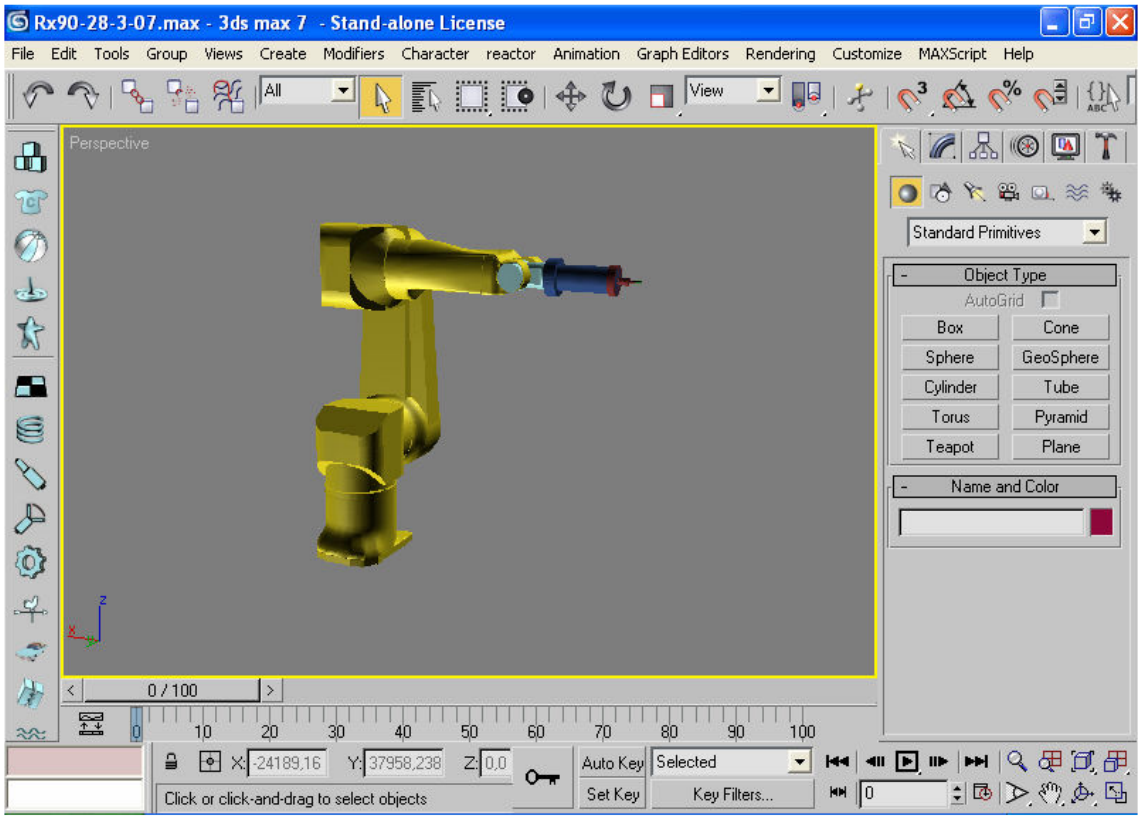

Figura 4.1. Nuevo diseño del Rx-90 con una fresa neumática como efector final..

El programa de diseño 3d Studio Max es una aplicación muy completa que además de las herramientas que facilitan el trabajo de modelaje permite crear animaciones en 3D. En un principio se observó la posibilidad de crear la simulación mediante el uso de este programa, ya que los resultados de renderizado son mucho mejores que los de DirectX, pero ello implicaba desarrollar un programa totalmente nuevo con un código de programación diferente: MaxScript.

Las razones por las que finalmente se optó por el uso de Visual C++ con las librerías de DirectX son las siguientes:

- Facilidad de aprendizaje. Visual C++ proviene de un lenguaje orientado a objetos de uso masivamente extendido (C++) y muy utilizado por el autor de este proyecto, frente a MaxScript que requeriría mucho tiempo previo para el aprendizaje.

- Facilidad para desarrollar rápidamente aplicaciones MFC con interfaz gráfica.
- Gran cantidad y variedad de librerías disponibles para su uso, lo que supone un ahorro de tiempo considerable.
- Extensa documentación disponible para su consulta.
- Posibilidad de reutilizar parte del código fuente escrito en proyectos anteriores tanto por ser este proyecto en sí continuación de [4] y aprovechar el trabajo se simulación gráfica con DirectX realizado en [6].

4.2. DESCRIPCIÓN GENERAL.

En este apartado vamos a explicar cómo se desarrolla el programa implementado para obtener finalmente la aplicación que satisfaga los requisitos exigidos al principio del proyecto.

El objetivo inicial en la fase de planificación del proyecto consistía en añadir a un sistema automatizado de corte tridimensional de piezas un interfaz que permitiese al usuario comprobar los detalles del corte en un entorno virtual antes de poner en funcionamiento el manipulador real.

Para ello disponíamos de dos proyectos anteriores:

- El primero de ellos realiza todo el trabajo previo de toma y procesamiento de datos y comunicación con el controlador del robot [4]. Se compone la trayectoria, se estudia su viabilidad y finalmente se envía al controlador para su ejecución.
- El segundo de ellos [6] fue un trabajo previo de simulación gráfica que también incluía comunicación con el controlador del robot a través del puerto serie, de hecho, este sistema de comunicación fue aprovechado y mejorado en [4]. Pero lo que a nosotros nos interesa de esta labor es el uso de librerías gráficas de DirectX y el diseño previo del Rx-90 en 3D Studio Max.

Es evidente que para el correcto funcionamiento de nuestra aplicación de simulación virtual necesitamos los cálculos previos de la aplicación 'Corte3D' (que ha sido modificada en este proyecto y cuyos cambios se explican extensamente en los capítulos anteriores), pero en este apartado nos vamos a centrar en la aplicación 'Rx90.exe'.

En la aplicación 'Corte3D' creamos una clase llamada **CDlgSimulacion** que abría un cuadro de diálogo para preguntar al operario si deseaba realizar la simulación, esta llamada se hace con la instrucción "system("Rx90.exe");" que ejecuta una aplicación siempre que ésta tenga su archivo ejecutable en la carpeta "SYSTEM32" de *Windows*.

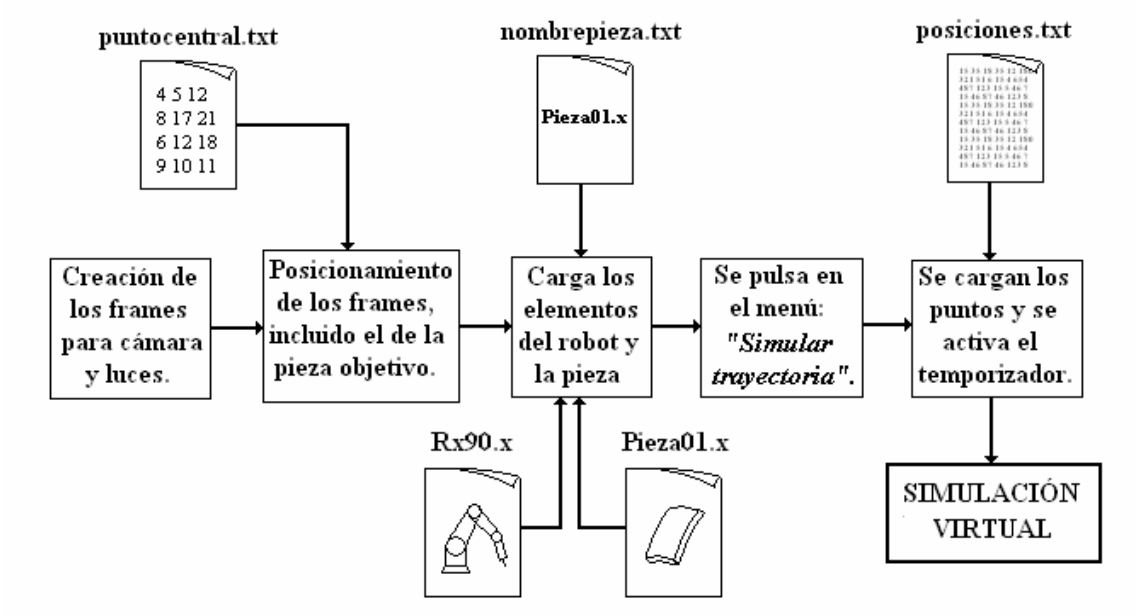

Figura 4. 2. Diagrama de la aplicación Rx90.exe para la simulación virtual.

4.2.1. Creación del interfaz de usuario.

Los primero que se hace es inicializar el sistema DirectX, pero no vamos a entrar en los detalles técnicos de inicialización de dispositivos y objetos necesarios para darle color, textura, luces y sombras a los distintos componentes de la escena.

Diremos a modo de resumen que se crean en primer lugar los distintos frames (componentes) para cámaras, luces y elementos del robot; se ubican en el espacio teniendo en cuenta que en DirectX el origen del sistema de referencia se encuentra en la esquina inferior izquierda de la escena y que sigue la regla de la mano izquierda (no derecha, como siempre), así que cuando queramos posicionar algún elemento hay que establecer los ejes de la siguiente manera:

{D3DVAL(eje -y), D3DVAL(eje +z), D3DVAL(eje +x)}.

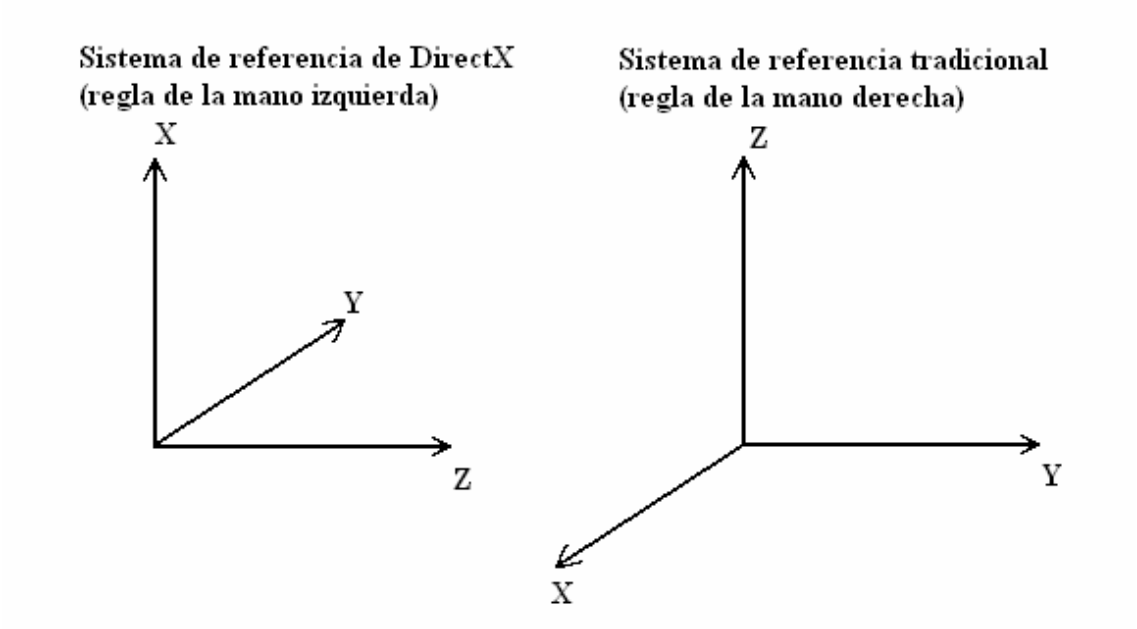

Figura 4. 3. Sistemas de referencia.

Una vez creados todos los frames de la escena (incluido el que soportará la pieza diseñada con programas CAD/CAM y cuyo corte es el objetivo de todo el proceso) hay que relacionarlos unos con otros. Para conseguir un efecto visual deseado, se relacionan los objetos de la forma 'padre-hijo'; así una pieza del robot, al moverse, arrastrará a todas las que dependan de ella (sus 'hijos') y heredarán su nueva posición.

Se llamará a continuación a una función que posicionará todos los componentes en un lugar concreto respecto del objeto 'padre'. El elemento 'padre' principal es el de la escena, llamado "lpScene", de él saldrán frames 'hijos' que soportarán la cámara, las luces, la base del robot y la pieza a cortar; de todos ellos el único que tienes objetos a su cargo es la base del robot. Para los frames de las luces, la cámara y los componentes del robot esta ubicación es fija al principio; el problema se plantea a la hora de localizar el punto donde pondremos la pieza seleccionada para el corte.

Hemos de recordar que en la aplicación 'Corte3D.exe' añadimos un método que a partir de los puntos de la trayectoria (ya compuestos respecto al sistema de referencias de la base del manipulador) obtenía el punto central de la pieza: **void CalculaPuntoCentral(int NPuntos).** Esta función escribía en un fichero de nombre "puntocentral.txt" las coordenadas de los tres puntos que se consiguieron por

aprendizaje del robot, para que el programa conociera la posición de la pieza matriz, y punto calculado en el método citado unas líneas más arriba..

Con estos datos, la nueva aplicación 'Rx90.exe' puede colocar la pieza (mejor dicho, el frame que soportará el objeto pieza, ya que todavía no hemos dicho qué elementos vamos a cargar en cada frame) en el sitio correcto y además inclinarla en el espacio según esté inclinado el *stock* o pieza matriz.

A continuación cargamos los objetos (componentes del robot y pieza objetivo) y los asociamos a su frame correspondiente y les damos el color y la textura deseada.

A parte de todo lo anterior, se han creado unos menús para que el usuario pueda mover el robot virtual como desee, cambiar el punto de vista para visualizar el robot desde otra posición de la cámara o iniciar la simulación de la trayectoria de corte.

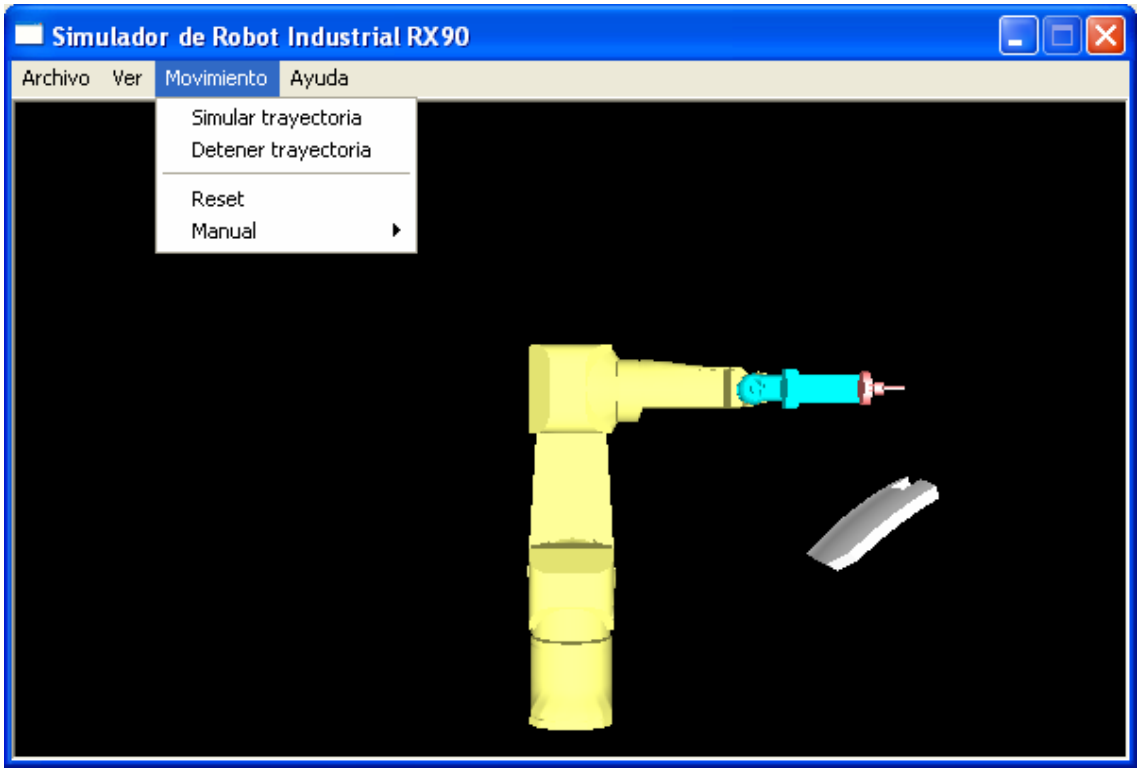

Figura 4. 4. Pantalla para la simulación virtual de trayectorias de corte.

4.2.2. Lectura de la trayectoria.

Una vez que el operario pulsa en el menú "*Simular trayectoria*" se procede a la lectura de las posiciones de la trayectoria de corte, calculadas previamente en la aplicación 'Corte3D', desde un fichero denominado "posiciones.txt".

En este fichero se guardaron todos los puntos en coordenadas articulares tras resolver el modelo cinemático inverso para comprobar su alcanzabilidad. Sería redundante partir de cero y recalcular todos estos datos cuando ya se han obtenido en un trabajo previo. Recordemos que los puntos que se guardaron en el fichero no son exactamente los que se enviarán al controlador del robot, y no sólo por el cambio de formato (el controlador entiende los puntos en coordenadas XYZeulerZYZ, es decir, coordenadas de la posición en cartesianas y de la orientación en los ángulos de Euler ZYZ; mientras que el simulador utiliza las variables articulares) sino porque se añaden puntos para realizar una maniobra de aproximación al punto inicial y otra de separación tras el último punto de la trayectoria de corte.

Además hemos de recordar que se hacía una comprobación de los posibles saltos angulares que pudiera realizar alguno de los ejes durante su funcionamiento, ya que el programa interno del controlador, el V_TRAJSIG, solucionaba esta dificultad de forma efectiva, pero el simulador no, al mover bruscamente una articulación desplazaba todos los elementos posteriores de forma que alteraba la trayectoria que seguía el extremo de la fresa en su función de corte. Para evitar ese efecto no deseado, introducíamos puntos intermedios de forma que el valor de las otras articulaciones se modificase levemente para que el efector final no se desviase del camino deseado.

Por tanto, la fase de captura de datos consiste en leer el fichero "posiciones.txt", contar el número de puntos que en él se guardan y crear el atributo '**m_pPuntoTrayectoria**' de la clase **CPuntoTrayectoria** con las dimensiones correctas y almacenar en él todas las coordenadas leídas desde el archivo de texto. Con esto tenemos los puntos de la trayectoria listos para ser utilizados.

4.2.3. Ejecución de la trayectoria

Al pulsar el botón del menú "*Simular trayectoria*" se realiza toda la captura y almacenamiento de datos descritos en el apartado *4.2.2* y a continuación se activa un temporizador que llamará a una rutina cada cierto tiempo (cada vez que transcurra el tiempo marcado en el temporizador). Esta función leerá una línea de datos del atributo '**m_pPuntoTrayectoria.m_sCoordtheta**' obteniendo los valores de los seis ángulos

que dan lugar a un punto de la trayectoria e irá (uno por uno) modificando ese ángulo en su articulación correspondiente del robot virtual.

El valor de temporizador es el que marcará la velocidad de movimiento del robot virtual, que se reducirá cuando llegue a zonas difíciles de la trayectoria determinadas por muchos puntos seguidos y muy juntos; y se acelerará en zonas donde los puntos estén más separados.

La ejecución de la trayectoria se puede detener en cualquier momento pulsando "*Detener trayectoria*", y empezar de nuevo con la misma trayectoria pulsando nuevamente "*Simular trayectoria*". Si lo que se desea es recuperar la posición inicial del robot se pulsa (en el mismo menú emergente donde se encuentran las opciones anteriores) "*Reset*".

Otra opción también permitida por esta aplicación y que resulta de un gran interés para la simulación de la trayectoria es el cambio de punto de vista. Con ello, podemos seguir la trayectoria y comprobar su fidelidad al contorno de la pieza posicionándonos (posicionando la cámara) en cualquier parte de la escena, visualizando el movimiento desde el lado del robot que queramos, e incluso desde detrás de la pieza.

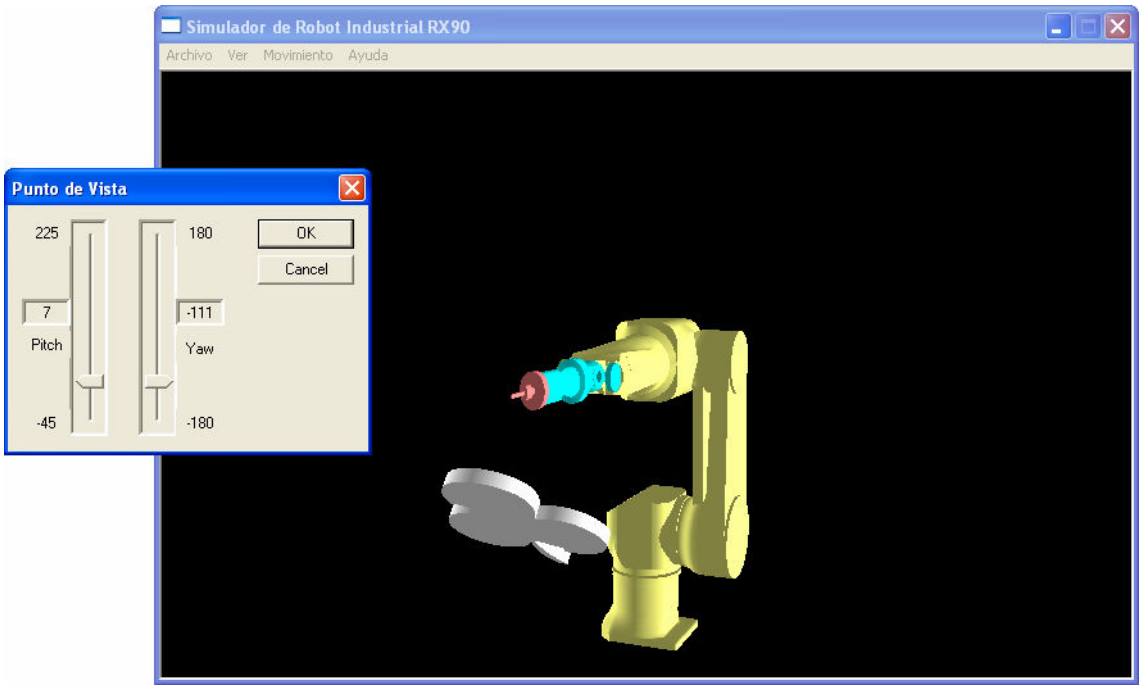

Figura 4. 5. Cambio del punto de vista de la escena..

4.2.4. Movimiento manual del robot.

Esta aplicación también le permite al usuario mover el robot virtual a su antojo para realizar las comprobaciones que necesite.

Dicho movimiento puede realizarse variando directamente las coordenadas articulares (*figura 4.6*) mediante un cuadro de diálogo con seis barras deslizadoras con topes según el rango de variación de cada eje (de forma que nunca se salga del espacio alcanzable por el manipulador), o bien, moviendo el efector final (la fresa en este caso) siguiendo la dirección de los ejes de referencia correspondientes al sistema ubicado en la base del robot mediante tres barras deslizadoras (*figura 4.7)*; en este caso, la orientación de la fresa no se ve modificada pero resulta mucho más fácil salirse del rango de alguna de las articulaciones, apareciendo en consecuencia un mensaje avisando de dicha circunstancia (entonces, tras pulsar '*Aceptar*', se devolvería al robot a la posición anterior al último movimiento en la barra deslizadora).

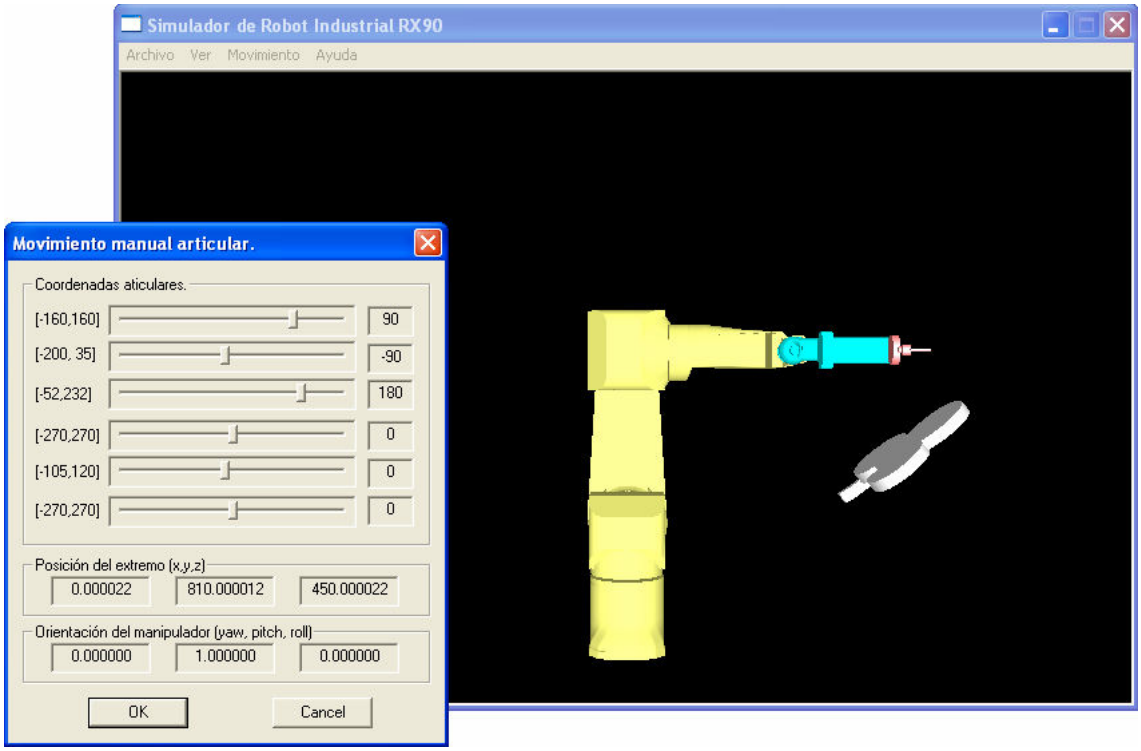

Figura 4. 6. Movimiento manual articular. Aplicación del modelo directo.

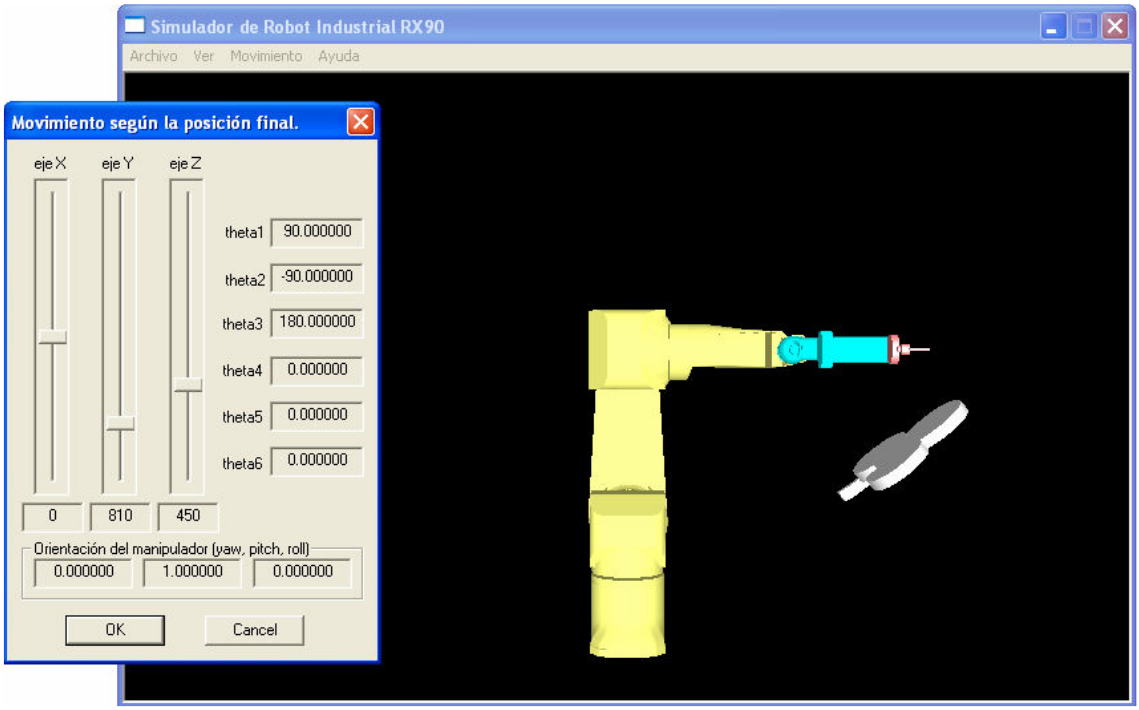

Figura 4. 7. Movimiento manual cartesiano. Aplicación del modelo inverso.

4.3. Descripción técnica de la aplicación de simulación virtual.

Al igual que hicimos con la aplicación 'Corte3D.exe' desarrollada en el apartado 3 de esta memoria hemos de explicar todo lo descrito en el punto anterior 4.2, adentrándonos en lo que son meramente algoritmos de código que implementan las diversas clases de la aplicación 'Rx90.exe'.

Seguiremos el mismo esquema de exposición y presentaremos una lista de las clases y su funcionalidad. A continuación desarrollaremos en profundidad cada clase por separado.

4.3.1. Diagrama de clases.

Clase	Ficheros	Funcionalidades
CRx90App	Rx90.h Rx90.cpp	Clase principal de la aplicación MFC.
CRx90Doc	Rx90Doc.h Rx90Doc.cpp	Clase documento implementada directamente por AppWizard al crear un proyecto MFC.
CRx90View	Rx90View.h Rx90View.cpp	Clase vista creada por AppWizard para visualizar los documentos de la clase CRx90Doc.
CMainFrame	MainFrm.h MainFrm.cpp	Clase principal de configuración. Se inicializa el sistema DirectX, se crea un objeto DirectDraw constituyendo el frame que conformará la escena y se crean los frames que formarán la imagen virtual del robot. Es en esta clase donde se carga el diseño previo del Rx90 y de la pieza objetivo, asignándole cada componente a un frame y relacionando unos con otros mediante el calificativo de 'padre-hijo'.

		Se le da color y textura a cada elemento. El robot se moverá rotando cada elemento según se indique y arrastrando a los objetos 'hijos'. La imagen se renderiza cada vez que se actualiza alguna variable de posición.
CRobot	Robot.h Robot.cpp	Clase utilizada para guardar la estructura con los datos de posicionamiento actual de las articulaciones del robot.
CDlgCamara	DlgCamara.h DlgCamara.cpp	Esta clase permite cambiar el punto de vista de la escena modificando la posición de la cámara. El movimiento de la cámara se efectúa cambiando los valores del cuadro de diálogo asociado 'pitch' y 'yaw', el primero para variar el ángulo horizontal y el segundo para el ángulo vertical.
CDlgCoordArt	DlgCoordArt.h DlgCoordArt.cpp	Clase asociada al movimiento manual del robot utilizando el modelo directo, es decir, se modifican sus coordenadas articulares para modificar su posición.
CDlgCoordCart	DlgCoordCart.h DlgCoordCart.cpp	Clase asociada al movimiento manual del robot utilizando, en este caso, el modelo inverso, es decir, que lo que cambiamos es la posición del extremo de la fresa moviéndola siguiendo la dirección de los ejes coordenados ubicados en la base del robot.
CCaptura	Captura.h	Esta clase se usa para obtener el listado de

	Captura.cpp	puntos que forman la trayectoria de corte del manipulador. En el caso de que esta aplicación sea llamada desde la aplicación 'Corte3D', los puntos se leerán desde un fichero de texto y se añadirán otros para que el robot se desplace suavemente hasta la posición inicial.
CPuntoTrayectoria	PuntoTrayectoria.h PuntoTrayectoria.cpp	Clase muy similar a la clase CPuntoTrayectoria de la aplicación 'Corte3D'; contiene las estructuras y operaciones necesarias para trabajar con los puntos de las trayectorias: permite almacenar los puntos en diversas formas (XYZvector, XYZeulerZYZ, theta(i), matriz de transformación homogénea) y realizar conversiones entre ellos y ejecutar las operaciones necesarias para cada formato.

Tabla 4. 1. Diagrama de clases de la aplicación 'Rx-90.exe'

4.3.2. CRx90App.

Esta es la clase principal de la aplicación de simulación virtual, es generada automáticamente por el programa de desarrollo Visual C++ al crear un proyecto MFC.

En esta clase se declaran los atributos siguientes, que serán utilizados para operar con los puntos de las trayectorias:

- **CMainFrame *TheFrame:** Puntero a una variable de la clase **CMainFrame**, cuya función es la de guardar los elementos de la escena. Con esta variable podemos acceder a los elementos del robot, conocer su posición actual y modificarla.

- **CRobot RX90Robot:** Es una variable de tipo **CRobot**, que permite leer y escribir sobre la estructura Rx90Pos, que la que almacena los valores de cada una de las variables articulares del robot virtual.
- **CCaptura* m_pPuntosTrayectoria:** Puntero a un array de objetos de la clase **CCaptura**, que contendrá la información en coordenadas articulares de la trayectoria. La clase **CCaptura** será la encargada de leer desde un archivo de texto dichos puntos.
- **CPuntoTrayectoria* m_pPuntoTrayectoria:** Este puntero es exactamente igual que el anterior pero apunta a un array de la clase **CPuntoTrayectoria**. Con dicha clase podremos hacer todo el procesamiento necesario para obtener las coordenadas de la trayectoria a partir de un archivo '.apt' sin necesidad de pasar por la aplicación 'Corte3D'.
- **CPuntoTrayectoria m_MundoReal:** Instancia de la clase **CPuntoTrayectoria** que contiene la información geométrica sobre el sistema de referencias ligado a la pieza matriz en la realidad (entorno real). Se utiliza para obtener la transformación que relaciona el entorno real con el entorno virtual (diseño) y, por tanto, determinar la localización y orientación iniciales de la herramienta de corte antes de ejecutar la trayectoria en el entorno real.
- **CPuntoTrayectoria m_MundoVirtual:** Instancia de la clase **CPuntoTrayectoria** que contiene la información geométrica sobre el sistema de referencias ligado a la pieza matriz en el diseño (entorno virtual). Se utiliza para obtener la transformación que relaciona el entorno real con el entorno virtual y, por tanto, determinar la localización y orientación iniciales de la herramienta de corte antes de ejecutar la trayectoria en el entorno real.

Las funciones de esta clase son las básicas que se crean para una aplicación MFC del tipo SDI:

- **CRx90App():** Constructor estándar.

- **InitInstance():** Es el método de inicialización de la aplicación. En este caso, aparte de la inicialización de los controladores propios de una MFC, se crea el atributo **TheFrame** de la clase **CMainFrame**, creando con él la ventana para DirectX.
- **OnAppAbout():** Método que abre el diálogo "Acerca de …".
- **OnIdle(LONG lCount):** Método que renderiza la imagen cuando se produce un cambio en el atributo **TheFrame**, esto es, cuando
  - "TheFrame->bInitialized == TRUE"

4.3.3. CMainFrame.

Esta clase se ocupa de configurar toda la pantalla DirectX para darle el efecto virtual deseado, crea los frames necesarios para la escena, la cámara, las luces, los componentes del robot y la pieza objetivo, los posiciona, los relaciona unos con otros, les da color y textura y finalmente los mueve según lo requiera el usuario.

Además maneja los menús para que el operario pueda interactuar con el robot virtual.

Los atributos de esta clase son muchos y nos tendríamos que extender mucho para abarcarlos a todos, así que aquí comentaremos los más importantes a continuación:

- **LPDIRECT3DRMFRAME lpObject_frame[elementos]:** Vector de sistemas de coordenadas. Los frames son los cuadros donde se pueden cargar objetos (en nuestro caso se cargarán desde un archivo ".x"), así pues creamos un vector con un número determinado por los componentes que conforman el robot más uno, que será la pieza objetivo.
- **LPDIRECT3DRMFRAME lpWorld_frame:** Se crea el frame principal de la escena, que conllevará que todos los demás sean descendientes de él y sus posiciones estarán referidas directamente (en el caso de sus 'hijos') o indirectamente ('hijos' de sus 'hijos', etc…) a él.

- **CRobot* m_pRobot:** Puntero a un atributo de la clase CRobot que nos permite acceder a la estructura que guarda los datos internos del Rx-90 virtual, sus variables articulares.
- **BOOL move:** Variable de tipo booleano que indica cuando el robot está en movimiento y cuando está detenido.
- **int Pant:** Atributo de tipo entero que almacena el valor previo que tenía el ángulo de rotación de la cámara denominado 'pitch'. Este valor es importante para el caso en el que después de hacer cambios en el cuadro de diálogo "*Punto de vista*" se pulse "*Cancelar*", y haya que volver a los valores anteriores.
- **int Yant:** Atributo de tipo entero que almacena el valor previo que tenía el ángulo de rotación de la cámara denominado 'yaw'. Este valor es importante para el caso en el que después de hacer cambios en el cuadro de diálogo "*Punto de vista*" se pulse "*Cancelar*", y haya que volver a los valores anteriores.
- **int theta_ant[6]:** Array de seis variables que almacenan los valores de las coordenadas articulares previos a las modificaciones realizadas en el cuadro de diálogo "*Movimiento manual articular*". Valores que tendrán que imponerse en el caso de que el usuario pulse "*Cancelar*".
- **Vector Punto[3]:** Array de tres elementos de tipo vector que almacenan los tres puntos que se leyeron directamente de la posición del Rx-90 real al principio del proceso de generación de trayectorias. Estos puntos se guardaron en un fichero de texto desde la aplicación 'Corte3D' y serán ahora leídos y guardados en este array.
- **CCaptura* pCaptura:** Puntero a un atributo de la clase **CCaptura**. Se utilizará cuando se pulse en el menú superior emergente "*Simular trayectoria*", para acceder a la función encargada de leer la trayectoria de corte desde fichero.

- **CDlgCamara *pDlgCamara:** Puntero a la clase **CDlgCamara**, utilizada para cargar el cuadro de diálogo encargado de modificar el punto de vista de la escena.
- **CDlgCoordArt *pDlgCoordArt:** Puntero a la clase **CDlCoordArt**, cuya función es llamar al cuadro de diálogo para el movimiento manual a través de las coordenadas articulares del robot.
- **CDlgCoordCart *pDlgCoordCart:** Al igual que los anteriores, es un puntero a la clase **CDlgCoordCart** que carga el diálogo que modifica la posición del robot siguiendo la dirección de los ejes coordenados del sistema localizado en la base del manipulador.

Seguidamente mostraremos los métodos implementados en esta clase:

- **CMainFrame():** Constructor estándar de la clase. En ella se inicializan punteros y variables relacionadas con DirectX.
- **D3DInit():** Función que inicializa el sistema DirectX, enumera los dispositivos y crea un objeto DirectDraw, un objeto IDirect3D y se crean los frames de la escena y de la cámara, posicionándolos y orientándolos. En definitiva, se crea la escena.
- **BOOL MyScene:** Función principal en la composición final de la escena, llama a los métodos encargados de crear los frames, las luces, los objetos 'mesh' y las texturas. Relacionando además los frames con los objetos 'mesh'. Esta función llama a: **MakeMyFrames**, **MakeMyLights**, **SetMyPositions**, **MakeMyMesh**, **MakeMyWrap** y **AddMyTexture** que se explicarán a continuación. Los parámetros que se le pasan son parámetros inicializados en el método **D3DInit()**, vemos entre ellos el frame principal de la escena y el de la cámara. Es una función de tipo booleano, por lo que devuelve TRUE si no ha habido fallos y FALSE si sí los ha habido. Los parámetros que se le pasan son los siguientes:
  - **LPDIRECT3DRMDEVICE dev.**
  - **LPDIRECT3DRMVIEWPORT view.**

- **LPDIRECT3DRMFRAME lpScene.**
- **LPDIRECT3DRMFRAME lpCamera**

- **Void MakeMyFrames:** Crea todos los frames que se necesitan en la escena aparte del principal y el de la cámara, que se le pasan como parámetros. Se crean dos frames para iluminación, se guarda en la variable que apunta al frame 'padre' la dirección de la variable 'lpScene', ya que éste será el 'padre' de la escena; se le asignan como 'hijos' el frame de la base del robot y el de la pieza objetivo. A partir de ahí se van creando los frames de los demás elementos del robot modificando el 'padre' en cada ocasión para que cada componente descienda directamente del que le precede en el robot. Los parámetros que se le pasan son:
  - **LPDIRECT3DRMFRAME lpScene.**
  - **LPDIRECT3DRMFRAME lpCamera.**
  - **LPDIRECT3DRMFRAME * lplpLightFrame1.**
  - **LPDIRECT3DRMFRAME * lplpLightFrame2.**
  - **LPDIRECT3DRMFRAME * lplpWorld_frame.**
  - **LPDIRECT3DRMFRAME * lplpObject_frame.**
- **void MakeMyLights:** Función que crea las luces de la escena, las posiciona y las orienta. Los parámetros que recibe son:
  - **LPDIRECT3DRMFRAME lpScene.**
  - **LPDIRECT3DRMFRAME lpCamera.**
  - **LPDIRECT3DRMFRAME lpLightFrame1.**
  - **LPDIRECT3DRMFRAME lpLightFrame2.**
  - **LPDIRECT3DRMLIGHT * lplpLight1.**
  - **LPDIRECT3DRMLIGHT * lplpLight2.**
  - **LPDIRECT3DRMLIGHT * lplpLight3).**

- **void SetMyPositions:** Establece las posiciones de las luces, de la cámara, de la pieza objetivo y de los elementos que componen el robot. En el caso de la pieza diseñada para su corte los datos de la posición y la orientación son más complejos de determinar, ya que dependen de dónde y cómo se encuentre el *stock* o pieza matriz real. Ya se explicará más adelante la función que lee dichos datos desde un fichero de texto. Los parámetros que se le pasan a esta función se listan a continuación.
    - **LPDIRECT3DRMFRAME lpScene.**
    - **LPDIRECT3DRMFRAME lpCamera.**
    - **LPDIRECT3DRMFRAME lpLightFrame1.**
    - **LPDIRECT3DRMFRAME lpLightFrame2.**
    - **LPDIRECT3DRMFRAME lpWorld_frame.**
    - **LPDIRECT3DRMFRAME * lplpObject_frame).**
- **void MakeMyMesh:** Crea la malla en la estructura de la escena. Carga los objetos que se visualizarán en la pantalla, los correspondientes a los elementos del robot los carga del fichero "Rx90.x" y la pieza objetivo la carga de un fichero cuyo nombre es el nombre de la pieza tal y como se guardó en el programa de diseño CAD/CAM (recordemos que se usó CATIA en este caso): "nombre de la pieza.x". Estos objetos los escalaremos y les daremos color. Seguidamente se muestran los dos parámetros que se le pasan a la función:
    - **LPDIRECT3DRMMESHBUILDER * lplpbase_builder.**
    - **LPDIRECT3DRMMESHBUILDER * lplpObject_builder).**
- **void MakeMyWrap:** Este método crea una especie de envoltorio en torno a un objeto con el fin de determinar la forma de aplicarle textura. En nuestro caso se aplica el envoltorio a la malla de cada elemento del robot creada con **MakeMyMesh**.
    - **LPDIRECT3DRMMESHBUILDER base_builder.**

- **LPDIRECT3DRMWRAP * lpWrap.**

- **void AddMyTexture:** La textura son imágenes que se pueden aplicar a los objetos, esta función añade la textura a cada objeto de la escena en el envoltorio creado en la función anterior **MakeMyWrap**.
  - **LPDIRECT3DRMMESHBUILDER lpbase_builder.**
  - **LPDIRECT3DRMTEXTURE * lplpTex.**
- **void SetPosition:** Esta función consigue el movimiento relativo entre los componentes del manipulador virtual. En el archivo "Rx90.x" vienen definidos los elementos y los ejes de referencia de cada uno de ellos. A partir de estos ejes se define el nuevo eje de giro. Dado el eje y el ángulo de giro, parámetros que recibe esta función, se mueve el elemento correspondiente y sus descendientes. Para esta función se ha declarado la clase **CRobot**, simplemente se utiliza para crear una estructura con la posición actual del robot virtual. Los parámetros de esta función son, como ya se ha dicho unas líneas más arriba:
  - **double pos.**
  - **int eje.**
- **void SetCamera:** Permite cambiar el punto de vista desde el que podemos observar la evolución del movimiento del manipulador virtual. Esta función es llamada cuando se usa el cuadro de diálogo "*Punto de Vista*", y los parámetros que se le pasan son:
  - **double pitch.**
  - **double yaw.**
- **void OnTimer:** Esta función salta cada vez que expira el tiempo programado en el temporizador. A su vez llama a la rutina **where** encargada de leer un punto de la trayectoria almacenado en el atributo **m_pPuntosTrayectoria** en forma de variables articulares. Si se han leído ya todos los puntos se llama a la función que detiene el temporizador. El

único parámetro que recibe esta función es un valor que indica qué temporizador es el que ha expirado, es decir, puede haber tareas ejecutándose que necesiten un número de ciclos de reloj diferente, actuándose según corresponda en la rutina **OnTimer.**

  - **UINT nIDEvent.**

- **Void OnVerPuntoDeVista():** Esta función abre el cuadro de diálogo que permite modificar la posición de la cámara y ver la escena desde otro punto de vista. Durante su ejecución se detiene el temporizador. No se le pasa ningún parámetro.
- **void OnMovimientoStart():** Si el robot está detenido, llama a la función encargada de cargar los datos de la trayectoria en el atributo **m_pPuntoTrayectoria** y poner el marcha el temporizador 1 programándolo en nuestro caso a 400 ms (*SetTimer(1,400,NULL);*). Esta rutina no recibe parámetros de ningún tipo.
- **void OnMovimientoStop():** Este método detiene el temporizador y, por tanto, el movimiento.
- **void OnMovimientoReset():** Esta función detiene el temporizador y devuelve el robot a su posición original.
- **void CamaraStart():** Si se ha detenido el movimiento del robot con la apertura del cuadro de diálogo para cambiar el punto de vista, esta función vuelve a poner en marcha el temporizador (y con él el movimiento del robot) al cerrar dicho diálogo.
- **void OnMovimientoManualArticular():** Este método detiene el temporizador y hace una llamada al cuadro de diálogo que mueve el robot manualmente modificando el valor de sus variables articulares.
- **void OnMovimientoManualCartesiana():** Esta función abre también un cuadro de diálogo para mover manualmente el robot, pero ahora modificando la posición del extremo de la fresa según la dirección de los

ejes coordenados situados en la base del manipulador, es decir, aplicando el modelo inverso.

- **void LeeNombrePieza(char Nombre):** Cuando se ejecuta la aplicación después de hacer todo el procesamiento de la trayectoria con el programa 'Corte3D', no conocemos qué pieza ha de cargarse, puesto que fue esta última aplicación la que le preguntó al operario qué archivo ".apt" deseaba abrir. Lo que hicimos entonces fue guardar el nombre de dicho archivo en otro en formato texto, es decir, guardamos en "nombrepieza.txt" el nombre de la pieza que se cargó. Esta función abre este fichero de texto y le añade la terminación ".x", que es el formato que entiende DirectX. Finalmente tendremos que cargar en **MakeMyMesh** la pieza "nombrepieza" del archivo "nombrepieza.x". En el anexo *A.1.4* se explica cómo se exporta la pieza diseñada en CATIA al formato ".x".

- **Vector LeePuntoCentral():** Esta función abre el fichero de texto "puntocentral.txt" para leer las coordenadas del punto donde se ha de situar la pieza objetivo y de los tres puntos iniciales que indicaban la posición del *stock* matriz. Estas coordenadas recordamos que se calcularon en la aplicación 'Corte3D' con los puntos de la trayectoria ya compuestos respecto al sistema de referencias global situado en la base del robot. El resultado leído desde el archivo se devuelve con el tipo *Vector*.

- **Vector InclinacionPieza():** Para ubicar la pieza objetivo no sólo necesitamos conocer el punto donde colocarla, obtenido leyendo desde fichero con la rutina anterior, además hemos de conocer la inclinación en los tres ejes del espacio de la pieza matriz, ya que ésta puede haberse colocado, en principio, de cualquier forma. Para orientar el *stock* en el espacio necesitamos los tres puntos iniciales del robot que se consiguieron por el método de "enseñanza" (moviendo el robot real manualmente hasta los puntos). Estos puntos se leyeron también desde fichero con la función anterior **LeePuntoCentral()** y se guardaron en un atributo de esta clase llamado **Punto[3]** que es una array de tres elementos de tipo vector. Desde el

punto de vista geométrico, tres puntos en el espacio son suficientes para orientar un plano, y aunque la pieza objetivo no sea plana (puede tener tres dimensiones) la orientación es exactamente la misma. Esta función devuelve una variable de tipo vector con los tres ángulos de inclinación "*yaw*", "*pitch*" y "*roll*".

4.3.4. CRobot.

Esta clase es muy simple y sirve únicamente para acceder a la estructura que almacena los valores actuales de las variables articulares de los ejes del robot virtual e inicializarlos a cero; que no significa que inicialmente todos los ángulos valgan cero, ya que dichos valores hay que escalarlos para que correspondan exactamente con los ángulos del robot real. Esto ya se explicará cuando comentemos con detalle la clase ***CDlgCoordArt***.

El único atributo declarado en esta clase es el siguiente:

- **Rx90Pos Posicionact**: Es una variable del tipo Rx90Pos, que a su vez es un tipo cuya estructura se declara en el archivo de cabecera "*Estructura.h*", que se constituye con dos arrays de tipo *double* y seis elementos: *Artic* y *Cart*. Así pues, la variable **Posicionact** almacena los valores actuales de los ejes articulares en el campo *Artic*.

Por otro lado, el único método que conforma esta clase se muestra a continuación:

- **CRobot()**: Rutina cuya finalidad es la de inicializar los valores almacenados en el campo *Artic* de la variable **Posicionact** a cero.

4.3.5. CDlgCamara.

Como ya se explicó anteriormente, esta clase permite modificar la posición de la cámara para ver la escena desde otros puntos de vista. El cambio se produce moviendo dos barras deslizadoras que aparecen en el cuadro de diálogo de "*Punto de vista*", cada barra representa un ángulo de rotación de la cámara, uno para el sentido vertical y otro para el horizontal.

Los atributos reseñables de esta clase son:

- **int nYaw:** Esta variable se utiliza para guardar el dato de la posición de la barra deslizadora que produce el cambio en el sentido de giro horizontal.
- **int nPitch:** Variable de tipo int que almacena el valor del *scroll* (barra deslizadora) que modifica el ángulo de giro en el sentido vertical.
- **int YawAnt:** Atributo de tipo int que almacena el valor previo del ángulo horizontal antes de realizar las modificaciones pertinentes. Se usa este valor para el caso de pulsar "*Cancelar*" tras cambiar la posición de la cámara.
- **int PitchAnt:** Variable que, al igual que la anterior, almacena el valor previo del ángulo vertical, para el caso en que se pulse "*Cancelar*" después de hacer las modificaciones necesarias.
- **CMainFrame *m_pFrame:** Puntero a una variable de la clase **CMainFrame** que permite manejar los objetos de la escena desde la clase actual. En este caso el frame que modificamos es el que soporta la cámara.

Una vez vistos los atributos pasamos a listar y comentar los métodos que conforman esta clase:

- **BOOL OnInitDialog()**: Ésta es la rutina que salta cuando se pulsa en el menú "*Ver*" y a continuación "*Punto de vista*". Establece los rangos de variación de las barras deslizadoras y almacena los valores actuales de dichas barras para el caso en el que se modifiquen y se pulse finalmente "*Cancelar*".
- **void OnCancel()**: Si se pulsa "*Cancelar*", se imponen en las barras deslizadoras el valor de las variables que guardaban los valores previos (antes de los cambios), y a continuación se coloca la cámara también en la posición guardada. Si antes de iniciar el cuadro de diálogo el robot virtual estaba en movimiento, se vuelve a poner en marcha llamando a la función **CamaraStart(),**
- **void OnOK()**: Al pulsar "*Ok*" en el cuadro de diálogo se guardan los ángulos en los que ha quedado definitivamente posicionada la cámara en las variables de la clase **CMainFrame,** para que pueda ser leído cuando se

vuelva a llamar a la función **OnInitDialog** de esta clase. Nuevamente, si el robot virtual estaba en movimiento antes de abrir el diálogo hay que volver a ponerlo en marcha.

- **void OnVScroll**: Esta función permite leer los cambios que se producen en las barras deslizadoras que modifican los ángulos de posicionamiento de la cámara. Su funcionamiento es el siguiente, cada vez que se produce un cambio en una de las barras salta esta rutina que lee el nuevo valor, y a continuación se impone ese valor a la posición de la cámara llamando a la función **SetCamara** de la clase **CMainFrame.**

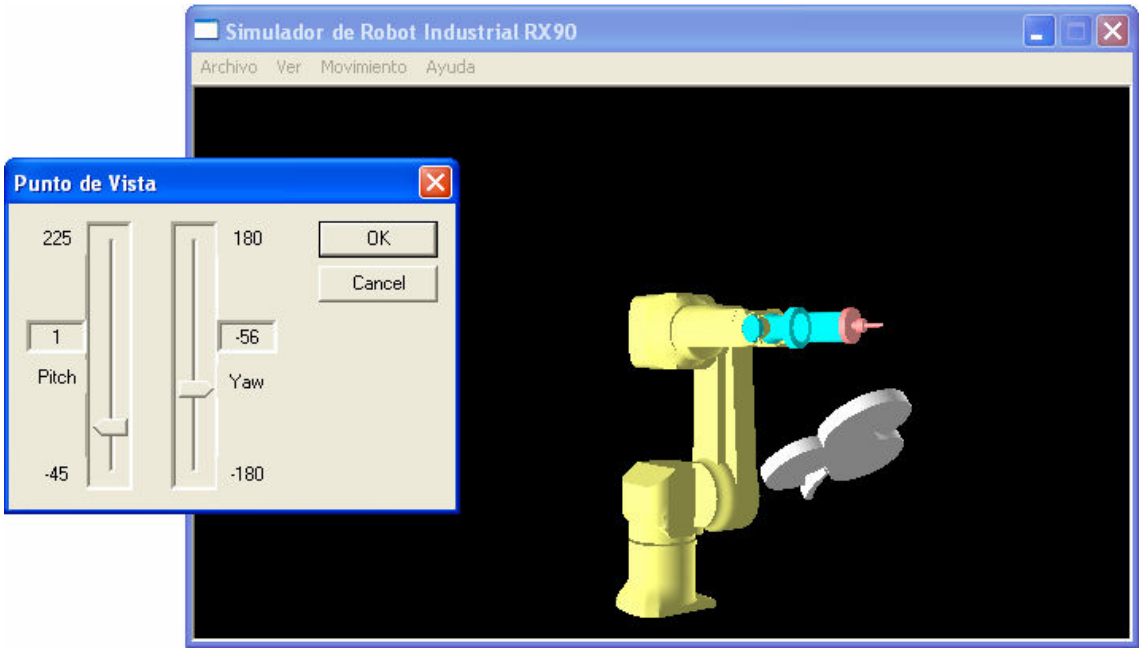

Figura 4. 8. Cuadro de diálogo para el cambio de Punto de Vista.

4.3.6. CDlgCoordArt.

Una de las opciones de las que no disponía el proyecto primitivo de simulación es la de mover el robot virtual al antojo del usuario. Con esta clase y la siguiente, se pueden modificar las variables angulares de los distintos ejes que constituyen el manipulador; ésta es una aplicación del modelo cinemático directo del Rx-90.

Para mover cada articulación se disponen seis barras deslizadoras cuyos rangos de variación se corresponden con las limitaciones mecánicas de cada eje en el robot real, como se puede ver en la *figura 4.6*.

Seguidamente mostramos las variables propias de esta clase:

- **CMainFrame *m_pFrame**: Puntero a una variable de la clase **CMainFrame** que permite manejar los objetos de la escena desde la clase actual. En este caso los frames que modificamos son los que soportan a los elementos del robot.
- **int th[6]**: Array de seis elementos de tipo entero que se utiliza para almacenar las posiciones que se obtienen de las barras deslizadoras. Estos valores leídos están en grados, y hay, además, que escalarlos para que correspondan exactamente con los del robot real. Estos valores de escala se muestran en la *tabla 4.2*.
- **m_artic1-6**: Son seis variables de control asociadas a cada una de las seis barras deslizadoras que modifican las variables articulares del robot.

Eje	Valor leído del *scroll* en grados.	Valor escalado de los ejes en radianes.
1	θ_1	$\theta_1 \frac{\pi}{180} - \frac{\pi}{2}$
2	θ_2	$\left(\theta_2 \frac{\pi}{180} + \frac{\pi}{2}\right)\frac{1}{2}$
3	θ_3	$\left(\theta_3 \frac{\pi}{180} - \pi\right)\frac{1}{3}$
4	θ_4	$\left(\theta_4 \frac{\pi}{180}\right)\frac{1}{4}$
5	θ_5	$\left(\theta_5 \frac{\pi}{180}\right)\frac{1}{5}$
6	θ_6	$\left(\theta_6 \frac{\pi}{180}\right)\frac{1}{6}$

Tabla 4. 2. Escalado de los ángulos de las variables articulares.

Por otro lado, las funciones que hacen posibles mover el robot según los ángulos de sus ejes son:

- **BOOL OnInitDialog**(): Es la rutina que salta cuando se pulsa dentro del menú "*Movimiento*" la opción "*Manual articular*". Primeramente establece los rangos de variación de las seis barras deslizadoras, que como ya se ha comentado anteriormente, coinciden con las limitaciones mecánicas del manipulador real. Lee la posición actual del robot (antes de que se modifique nada) e impone inicialmente dichos valores a las barras, además de guardar estos datos en un array de la clase **CMainFrame**.

- **Void HScroll:** Esta función es llamada cada vez que se produce un cambio en los *scrolls*, en tal caso se lee el nuevo valor de las seis barras, se escalan y se llama a la función de la clase **CMainFrame** que permite mover el robot virtual en la escena. Además se calcula el modelo directo para mostrar también en el cuadro de diálogo los datos de la posición del extremo de la fresa en coordenadas cartesianas de posición y de orientación.

- **Void ModeloCinematicoDirecto:** Función que recibe como parámetros cinco variables enteras correspondientes a los valores de los cinco primeros ejes (la sexta coordenada articular no afecta al valor de la posición y de la orientación). Aplica las expresiones (2.14) y (2.15) de la posición y la orientación obtenidas en el apartado 2.1.6. del presente documento, y que volvemos a mostrar aquí a modo de recordatorio.

$$\begin{aligned} a_x &= (c_1c_{23}c_4 - s_1s_4)s_5 + c_1s_{23}c_5 \\ a_y &= (s_1c_{23}c_4 + c_1s_4)s_5 + s_1s_{23}c_5 \\ a_z &= -s_{23}c_4s_5 + c_{23}c_5 \end{aligned} \qquad (4.\ 1)$$

$$\begin{aligned} p_x &= ((c_1c_{23}c_4 - s_1s_4)s_5 + c_1s_{23}c_5)d_6 + c_1s_{23}d_4 + c_1c_2a_2 \\ p_y &= ((s_1c_{23}c_4 + c_1s_4)s_5 + s_1s_{23}c_5)d_6 + s_1s_{23}d_4 + s_1c_2a_2 \\ p_z &= -(s_{23}c_4s_5 - c_{23}c_5)d_6 + c_{23}d_4 - s_2a_2 \end{aligned} \qquad (4.\ 2)$$

Donde el vector *A* representa la orientación de la fresa, *P* la posición del extremo, $s_i = \sin\theta_i$, $c_i = \cos\theta_i$, $s_{ij} = \sin(\theta_i + \theta_j)$, $c_{ij} = \cos(\theta_i + \theta_j)$ y a_2,

d_4 y d_6 son los parámetros de Denavit-Hartenberg aplicados en este manipulador.

- **void OnCancel**(): Función que restaura los valores de los ejes articulares previos a todas las modificaciones que se han realizado desde que se abrió el cuadro de diálogo de "*Movimiento manual articular*".
- **void OnOK**(): Método que almacena los valores actuales de las variables articulares en un array de seis elementos de la clase **CMainFrame**.

4.3.7. CDlgCoordCart.

Al igual que la clase anterior, permite al usuario de la aplicación realizar operaciones con el manipulador virtual moviendo el extremo de la fresa en la dirección de los ejes *X, Y* y *Z*, sin modificar su orientación.

En esta clase se implementa un cuadro de diálogo que aporta como datos la posición y la orientación del extremo en coordenadas cartesianas y las seis variables articulares. La ventana muestra a su vez tres barras deslizadoras verticales asociadas a los tres ejes de referencia, pudiendo así modificar las coordenadas de la posición en el espacio mientras que la aplicación resuelve el modelo cinemático inverso para calcular las nuevas variables articulares y mover el robot virtual con dichos valores. Los detalles se pueden observar en la *figura 4.7.*

Los atributos de esta clase son:

- **m_pFrame**: Puntero a la clase **CMainFrame** que permite el acceso desde la clase actual a los objetos que conforman el robot y moverlos según correspondan.
- **m_posicx, m_posicy, m_posicz**: Son tres variables de control para las barras deslizadoras asociadas a los ejes coordenados.

Por otro lado, las funciones que hacen posible este movimiento según los ejes cartesianos son:

- **BOOL OnInitDialog**(): Este método es llamado al pulsar en el menú "*Movimiento*" y a continuación "*Manual cartesiano*". Establece los rangos

de variación de las posibles coordenadas cartesianas que pueda alcanzar el extremo de la fresa, pero como eso depende de la orientación con la que se pretenda alcanzar, en un principio se pone la suma de las longitudes de los elementos que constituyen el brazo robótico. Se leen los valores que en el momento actual poseen los ejes articulados, se escalan para trabajar con ellos según la *tabla 4.2*, y se guardan estos datos por si finalmente se cierra el cuadro de diálogo pulsando "*Cancelar*". Seguidamente se llama a la función que calcula el modelo directo para conocer la ubicación en el sentido de la posición y la orientación de la fresa en coordenadas cartesianas.

- **void ModeloCinematicoDirecto**: Esta función resuelve el problema directo para determinar la posición actual del extremo del robot. Una vez obtenida las coordenadas *X*, *Y* y *Z* haciendo uso de las expresiones (4.1) y (4.2) se imponen estos valores en las barras deslizadoras como posición inicial. Esta rutina recibe como parámetros las cinco primeras variables articulares (la sexta y última no afecta al cálculo de la posición).

- **void OnVScroll**: Esta rutina salta cuando se produce una modificación en las barras deslizadoras del cuadro de diálogo "*Movimiento manual cartesiano*". Se lee la nueva posición con las coordenadas *X*, *Y* y *Z*, se calcula el modelo cinemático inverso para obtener las coordenadas articulares llamando a la función **ModeloCinematicoInverso** de esta clase. Se comprueba si dichas variables pertenecen al rango permitido por las limitaciones mecánicas de cada eje del robot real; si efectivamente se produce una salida de rango aparecerá un mensaje comunicándoselo al usuario y devolverá al robot su posición original (posición que tenía cuando se inició el cuadro de diálogo). Si por el contrario todo es correcto, se impondrán los datos calculados a cada eje del robot virtual.

- **void ModeloCinematicoInverso**: Función que recibe como parámetros las coordenadas cartesianas *X*, *Y* y *Z*, como variables de tipo entero, y calcula las coordenadas articulares correspondientes a los seis ejes del robot virtual. Lo primero que hace es calcular la orientación actual de la fresa ya que no debe

modificarse; para ello lee de la clase **CMainFrame** las variables articulares y obtiene el vector orientación con las expresiones mostradas en (4.1). A continuación resuelve las ecuaciones dadas en el capítulo 2 de esta memoria, concretamente la expresión (2.19) para θ_1, (2.26) para θ_2, (2.33) y (2.34) para θ_3, (2.39) para θ_4 y finalmente (2.41) para θ_5, recordemos que como la fresa es simétrica respecto del eje Z del sistema de referencia solidario al extremo del manipulador, la variable θ_6 no sirve para nada (al contrario que en el caso de la garra).

- **void OnCancel():** Si después de modificar la posición del robot se pulsa "*Cancelar*", el robot volverá a la posición previa a la apertura del cuadro de diálogo. Se imponen los valores almacenados en **OnInitDialog** dos veces, porque la imagen visualizada en la escena no cambia hasta la posición impuesta correctamente (si el cambio es muy brusco); imponiendo dichas posiciones dos veces nos aseguramos que el robot virtual realmente muestre la posición deseada.

- **Void OnOk():** Esta función cierra el cuadro de diálogo de "*Movimiento manual cartesiano*" validando las modificaciones realizadas en su posición.

4.3.8. CCaptura.

Esta clase se utiliza para obtener los datos de la trayectoria de corte que va a seguir el robot virtual. Además vemos que se repiten funciones ya desarrolladas en la aplicación 'Corte3D', tales como **AbrirYLeerFicheroAPT** y **ProcesarContenidoFicheroAPT**, que tendrán que usarse en el caso de que la aplicación que tratamos en este apartado de simulación virtual no sea llamada a través de 'Corte3D', y haya que preguntar al operario el fichero ".apt" que desea abrir y procesar.

Los atributos más reseñables de esta clase los exponemos a continuación:

- **int func:** si se han realizado las operaciones de captura y procesamiento de datos correctamente se pone esta variable a 1, y en la clase **CMainFrame** se

podrá poner en marcha el temporizador que iniciará el movimiento del robot virtual.

- **int iteración:** Este atributo de tipo entero crece a medida que vamos representando posiciones de la trayectoria del robot virtual y detiene el temporizador cuando su valor alcanza el número total de puntos de la trayectoria.
- **CMainFrame *m_pFrame:** Puntero a la clase **CMainFrame** que permite acceder desde la clase actual a los frames que soportan los objetos de la escena.
- **Vector m_vPuntos:** Variable de tipo Vector, es decir, con tres campos reales (uno para la coordenada *X*, otro para la *Y* y finalmente otro para *Z*), que se utilizará para almacenar puntos del espacio de trabajo.
- **int TotalPuntosAPT**: Variable de tipo entero que indica el numero de puntos que posee la trayectoria que finalmente recorrerá el robot virtual.

Las funciones importantes de esta clase son las tres primeras que vamos a explicar a continuación, las otras tres son repetición de funciones ya explicadas al comentar la clase **CCorte3DApp** de la aplicación 'Corte3D'.

- **void Capture()**: Esta rutina abre el fichero de texto "posiciones.txt" donde se guardaron las coordenadas articulares de los puntos de la trayectoria procesados en la aplicación 'Corte3D'. Llama a la función **AproximacionPuntoInicial** para mover el robot de la posición actual a la primera localización de la trayectoria. En el caso de que haya habido problemas al leer el archivo de texto, significa que la aplicación no ha sido llamada desde 'Corte3D', sino que se ejecuta de forma independiente, por tanto no se habrá cargado ninguna pieza para el corte en la escena pero sí se pueden seguir trayectorias al pulsar "*Simular trayectoria*", habrá que hacer uso de las funciones que leen el fichero APT y lo procesan.

- **void AproximacionPuntoInicial**(): Función que añade a la trayectoria una serie de puntos al principio para que el robot virtual se aproxime suavemente desde la posición actual al primer punto.
- **void where**(): Esta rutina es llamada cada vez que salta el temporizador de la clase **CMainFrame** y lee un punto de la trayectoria en coordenadas articulares, lo escala como se explicó en la *tabla 4.2*, y se lo impone a los ejes del robot virtual llamando a la función **SetPosition** de la clase **CMainFrame**.
- **BOOL AbrirYLeerFicheroAPT(CString& strContFicheroAPT)**: Esta función ya se comentó perfectamente en el apartado 3.4.2 del presente documento.
- **BOOL ProcesarContenidoFicheroAPT(CString& strContFicheroAPT)**: Función ya explicada en el apartado 3.4.2.
- **void CalculaPuntoCentral(int NPuntos)**: Función ya explicada en el apartado 3.4.2.

4.3.9. CPuntoTrayectoria.

Debido a la posible necesidad de que la aplicación que nos ocupa en este apartado tenga que procesar los puntos leídos de la trayectoria y operar con ellos es distintos formatos, hemos copiado esta clase de la aplicación 'Corte3D' con las modificaciones que se le realizaron.

Explicar nuevamente esta clase, con sus atributos y sus métodos sería redundar en lo que ya se comentó en el apartado 3.4.4 de esta memoria.