Capítulo I.- Ambiente de desarrollo y bibliotecas gráficas

1.1.- Software de dominio público

1.1.1.- C++

El lenguaje en C++ es la más adecuada para la programación gráfica debido a su flexibilidad en el uso y gestión de la memoria y de procesos a bajo nivel de la computadora además de contar con compiladores muy eficientes, lo que agiliza el desarrollo de proyectos complejos. En este lenguaje, a pesar de su flexibilidad en cuanto al uso de los recursos de la computadora, la programación se vuelve compleja y más cuando se trata de gráficos porque no solo es el uso de la memoria principal (RAM) sino también de la memoria de la tarjeta de video que hace todas las transformaciones en vértices, texturas y al renderizar una escena programada. En esta tesis las transformaciones geométricas del modelo se realizan en la memoria principal esto fue porque el texturizado depende del campo de distancia almacenado en memoria principal, entonces en vez de ir acumulando transformaciones para que al final las ejecute el GPU de la tarjeta de video, nosotros programamos esas transformaciones y las ejecutamos por cada evento dentro del programa. Un ejemplo de la programación de una transformación geométrica es la siguiente:

```
void Modelos::Trasladar(float x, float y, float z)
{


    for(int i=0;i<=nVertices;i++)
    {
        VertexArray[i].x+=x;
        VertexArray[i].y+=y;
        VertexArray[i].z+=z;
    }

}
```

La transformación geométrica fue programada de esta forma porque es más rápido y más eficiente para el procesador hacer una suma y una asignación que hacer las respectivas sumas y multiplicaciones que conlleva la multiplicación de matrices. El programa debe hacer transformaciones permanentes al modelo para que se pueda texturizar, si se hacen las transformaciones usando OpenGL en si el modelo se guarda tal y como es cargado y el texturizado no es dinámico porque OpenGL guarda el modelo en memoria pero no lo modifica, solamente hace una multiplicación de matrices con la matriz de transformación acumulada (ver anexo A) al final, antes de dibujar en el render. En C++ se dispone de muchas facilidades para programar estructuras de datos (arreglos no homogéneos) lo cual facilita la programación y permite al programador una programación más limpia y ordenada.

```
typedef struct
      {
            float x;
            float y;
            float z;
      }Vertice;

typedef struct
      {
            int slices;
            int rows;
            int cols;
            float minimoEnCampo;
            float maximoEncampo;
      }PropiedadesCampo;

typedef enum{ERROR0,ERROR1,ERROR2,ERROR3,ERROR4,ERROR5};
typedef enum{MODELO_ROTAR,MODELO_TRASLADAR,MODELO_ESCALAR};
typedef
enum{TRASLADAR_X,TRASLADAR_Y,TRASLADAR_Z,ROTAR_X,ROTAR_Y,ROTAR_Z
,SIN_TRASLADAR,SIN_ROTAR};
```

Es muy importante notar que al tener modelos de las cabezas en mallados, es decir en una estructura de vértices, sólo es necesario transformar sus coordenadas. Los campos

de distancia (información volumétrica) se calcularán sobre voxelizaciones de dichos mallados y re-voxelizaciones, cada vez que sufran alguna transformación geométrica. Si se intentara transformar un volumen de voxeles, sería necesario realizar al mismo tiempo la interpolación de los valores de cada voxel. Al trabajar las transformaciones sólo sobre los mallados, no es necesario lo anterior.

Las desventajas de la programación en C++ es la dificultad técnica para programar, al ser un lenguaje de menor abstracción que otros lenguajes, la programación se vuelve muy compleja y por ende dificulta el mantenimiento del código. . Por otro lado, la programación orientada a objetos permite la reutilización de código desarrollado en distintos proyectos, lo cuál ocurre frecuentemente en el Laboratorio de Imágenes y Visualización, donde se desarrolló el presente proyecto.

1.1.2.- DevC++

Bloodshed **Dev-C++** [DEVC2007] es un entorno de desarrollo integrado (IDE por sus siglas en inglés) para programar en lenguaje C/C++. Usa el compilador MinGW que es una versión de GCC (GNU Compiler Collection). Dev-C++ puede además ser usado en combinación con Cygwin y cualquier compilador basado en GCC.

DevC++ fue muy utilizado en el laboratorio de CCADET entre 2003 y 2009, debido a su facilidad de manejo y de ser software libre. Los programas para obtener el campo de distancia y alineación se hicieron mediante este entorno de desarrollo y parte del código fue adaptado al software de la aplicación de esta tesis. Un ejemplo de este código es una función para reservar memoria:

```
float ***alloc_float_volMem(void)
{      short i,j,k;
       float ***v;  /* if we set static, it will use the same */

       /* alloc a ptr to a slices-list of ptrs */
       /* TO_ADD:  if global and not NULL, free before realloc */
```

```
	if((v=(float ***) calloc( slices, sizeof(float **))
)==NULL)
	{	printf("Error of memory 1 ");
		exit(1);
	}

	/* for each slice ptr, alloc a row-list of float ptrs INIT
to 0 */
	for(k=0; k<slices; k++)
	{	/* could have been an alloc_ima( cols, rows, sizeof(
*float) ) */
		if((v[k]=(float **) calloc( rows, sizeof(float *))
)==NULL)
		{	printf("Error of memory 2 ");
			exit(1);
		}

		/* for each row(=line) ptr-ptr, alloc a col-list of
floats */
		for(j=0; j<rows; j++)
		{	/* for each line ptr, alloc a col-list of
floats */
			if((v[k][j]=(float *) calloc( cols,
sizeof(float)) )==NULL)
			{	printf("Error of memory 3: volume TOO BIG
(%li bytes?)\n", cols*rows*slices);  /* Nc*Nl*Ns */
				exit(1);
			}
		};
	};
	/*** access: v[z][y][x]   k j i, slice k row j col i,  Ns
Nl Nc ***/
	return v;   /* caller sets dimens, if required */

}	/*** end  alloc_floa*/
```

1.1.3.- VRML

VRML (sigla del inglés Virtual Reality Modeling Language. "Lenguaje para Modelado de Realidad Virtual" [VRML2005])- formato de archivo normalizado que tiene como

objetivo la representación de escenas u objetos interactivos tridimensionales; diseñado particularmente para su empleo en la web.

El lenguaje VRML (*Virtual Reality Modeling Language*) posibilita la descripción de una escena compuesta por objetos 3D a partir de prototipos basados en formas geométricas básicas o de estructuras en las que se especifican los vertices y las aristas de cada polígono tridimensional y el color de su superficie. VRML permite también definir objetos 3D multimedia, a los cuales se puede asociar un enlace de manera que el usuario pueda acceder a una página web, imágenes, vídeos u otro fichero VRML de Internet cada vez que haga click en el componente gráfico en cuestión.

1.1.4.- Programación en OpenGL

OpenGL son librerías libres (o sea, de dominio público) a las que cualquier usuario puede tener acceso sin tener que comprar una licencia y las cuales sirven para la programación de gráficos en 3D. OpenGL es puramente estructurado y únicamente es para enviar mandos para dibujar gráficos, no soporta controladores de audio o interfaces de interacción con el usuario como lo es DirectX. Una referencia que nos resultó muy útil, en especial para los fundamentos matemáticos de graficos con OpenGL es [BUSS03].

La librería de OpenGL trabaja como una estructura de datos de pila donde las instrucciones de transformación y de dibujo se ejecutan en orden inverso es decir de la última instrucción del bloque glPushMatrix(); y glPopMatrix(); hasta la primera.

Ejemplo:

```
glPushMatrix();
      glScaled(.1,.1,.1);
      glTranslatef(xx,yy,zz);
      glTranslatef(-cx,-cy,-cz);
      glutSolitCube(D);
glPopMatrix();
```

glPushMatrix(); y glPopMatrix(); sirven como separadores para que otras transformaciones dentro del código no afecten a otras figuras geométricas. En el código anterior se ejecutaría de la siguiente manera:

1.- Se manda a dibujar el Cubo.
2.- Se traslada de su posición original a –cx,-cy,-cz
3.- Se traslada de igual forma xx, yy, zz.
4.- Se cambia el tamaño en un factor de .1 en ‘x’ en .1 en ‘y’ y en .1 en ‘z’

La unidad atómica para dibujar en OpenGL son los vértices que se mandan a dibujar con la instrucción `glVertex3f(x,y,z)` esta función es más que solo dibujar un punto en el espacio de 3D porque con ella se arman los triángulos o cuadrados para después formar un poligono más complejo o superficies de formas irregulares. Para dibujar un sencillo triángulo en OpenGL se debe poner entre 2 instrucciones que son glBegin(MODO); y glEnd(); donde MODO es de cómo se van a unir los vértices, esta unión puede ser desde que se unan los vertices en triángulo, cuadrado o conforme se vayan dibujando. La union entre vértices también le dan el carácter de un polígono solido, de malla o de puntos. Para mantener una congruencia se manda a dibujar triángulo por triángulo con sus respectivas uniones entre otros trángulos como se puede ver en la figura 1:

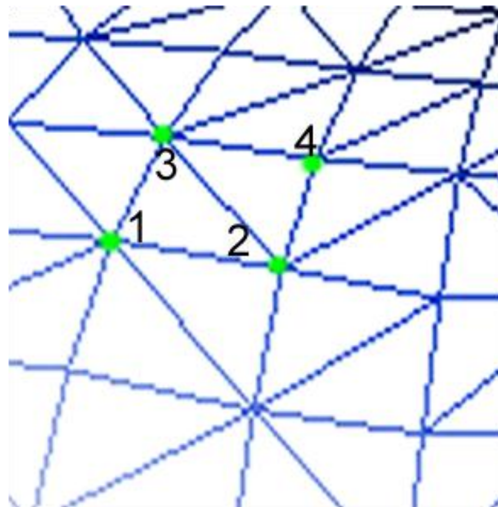

Figura 1: mallado por triángulos 1

En la figura anterior se puede ver que primero se manda a dibujar el vértice 1 y se une con el vértice 2 y 3 y el cuarto vértice se une con 3 y 2 y así se va armando toda la

figura. Entonces para figuras complejas se tienen 2 arreglos uno que es de los vértices y otro arreglo que es el de los índices de qué vértice se va a unir con cual otro.

Para poder mandar a dibujar se necesita configurar el puerto vista (Viewport) que es la ventana a través de la cual puede observar la escena. Para el viewport solo se necesita poner una posición inicial dentro del "render" y luego indicar los desplazamientos en las coordenadas x,y

```
glViewport(Vx,Vy,(GLsizei)(width),(GLsizei)(height));
```

Algo muy importante es el uso de cámaras en OpenGL. Una cámara en OpenGL simula una cámara real. Las cámaras también sufren cambios cuando se aplica una transformación 3D por ello se debe tener cuidado al poner alguna. Para expresar una cámara en OpenGL se necesitan de varios parámetros dentro de la función:

```
gluLookAt (Cx,Cy,Cz, Px, Py, Pz, Ux,Uy ,Uz);
```

Los tres primeros paramétros son las coordenadas de la posición de la cámara, los siguientes 3 son las coordenadas del punto a observar (punto de fuga) y por último se especifica un vector unitario "Up" que es el vector que orienta a la cámara con respecto a su propio sistema coordenado.

$$\bar{c} = c_x\hat{i} + c_y j + c_z k \qquad \textit{Vector de posición de la cámara}$$

$$\bar{p} = p_x\hat{i} + p_y j + p_z k \qquad \textit{Vector del punto del punto p}$$

$$\overline{Up} = u_x\bar{i} + u_y\bar{j} + u_z\bar{k} \leftrightarrow \left|\overline{Up}\right| = 1 \qquad \textit{Vector Up}$$

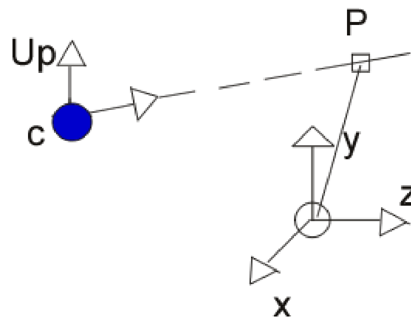

Figura 2: Representación geométrica de la cámara

La programación de luces en OpenGL requiere de 4 parámetros que 3 son para los tipos de luz (Ambiental, difusa, especular) y una posición de la fuente de luz. Los tipos de luces se construyen dentro del espacio de color RGB y que va de 0 a 1 en punto flotante esto para que la luz ambiental, difusa y especular se programe su comportamiento conforme al color.

```
GLfloat     light_specular1[]={1.0f,1.0f,1.0f,1.0f};
GLfloat     light_diffuse1[]={1.0f,1.0f,1.0f,1.0f};
GLfloat     light_ambient1[]={1.0f,1.0f,1.0f,1.0f};
GLfloat     light_position1[]={0.0f,0.0,-150.0f,1.0};

glLightfv(GL_LIGHT0,GL_POSITION,light_position1);
glLightfv(GL_LIGHT0,GL_AMBIENT ,light_ambient1);
glLightfv(GL_LIGHT0,GL_DIFFUSE ,light_diffuse1);
glLightfv(GL_LIGHT0,GL_SPECULAR,light_specular1);
glEnable (GL_LIGHT0);

glEnable(GL_LIGHTING);
```

Las luces se van agrupando deacuerdo a una numeración {GL_LIGHT0, GL_LIGHT1,…} se agregan los parámetros, se activa la luz y después se “encienden” las luces. Las luces son importantes para poder visualizar sombras, contornos y principalmente texturas.

El texturizado es un mapeo de los vertices a una textura. En OpenGL las texturas se mapean con la instrucción `glTexCoord2f(x,y);` x,y van de 0 a 1 en punto flotante. Se utilizan imágenes para texturizar y estas imágenes deben de tener un tamaño de multiplos de 2 si no, OpenGL no interpreta bien la imagen y produce efectos de deformación en la imagen. Lo que hace OpenGL para texturizar es tomar la imagen guardada en memoria y luego crea una tabla en punto flotante donde mapea la imagen a esa tabla en una razón de 1 entre la longitud de la imagen en 'x' y de 1 entre la longitud de la imagen en 'y' para que después se haga el mapeo con los vértices asignándolo a cada vértice un pixel de la imagen e interpolando la secuencia de pixeles con respecto al siguiente vértice con su asignación de textura. La textura de la figura 3 es una escala de color pensada para poder representar visualmente valores de distancia positivos (rojos) y negativos (azules) respecto al valor de referencia "0" en blanco. Servirá para cotejar objetos muy semejantes al superponerse en el espacio, o bien para desplegar un sólo objeto en cuya superficie se mapea la distancia al punto más cercano del segundo objeto (que no será visible, pero se encuentra en memoria).

Figura 3: Textura a mapear 256 x 256 pixeles

Ejemplo del texturizado de un poligono plano rectangular y que es mapeado simétricamente y rotado con respecto a la textura (véase figura: 4)

```
glBegin(GL_POLYGON);
      glTexCoord2f(.1,0);glVertex3f(-(width)/30,0,0);
      glTexCoord2f(.1,0);glVertex3f(-(width)/30,500,0);
      glTexCoord2f(.1,1);glVertex3f((width)/30,500,0);
      glTexCoord2f(.1,1);glVertex3f((width)/30,0,0);
glEnd();
```

Figura 4: texturizado con simetría horizontal

El siguiente ejemplo es de un texturizado asimétrico con respecto a la textura y rotado (véase figura 5)

```
glBegin(GL_POLYGON);
      glTexCoord2f(.1,0);glVertex3f(-(width)/30,0,0);
      glTexCoord2f(.1,0);glVertex3f(-(width)/30,500,0);
      glTexCoord2f(.1,.7);glVertex3f((width)/30,500,0);
      glTexCoord2f(.1,.7);glVertex3f((width)/30,0,0);
glEnd();
```

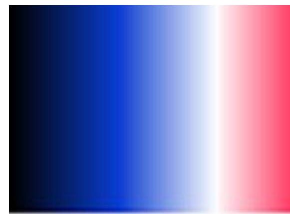

Figura 5: texturizado asimétrico horizontal

Ejemplo de un texturizando rotando en 180º la referencia de las texturas (véase figura 6):

```
glBegin(GL_POLYGON);
      glTexCoord2f(.1,1);glVertex3f(-(width)/30,0,0);
      glTexCoord2f(.1,1);glVertex3f(-(width)/30,500,0);
      glTexCoord2f(.1,0);glVertex3f((width)/30,500,0);
      glTexCoord2f(.1,0);glVertex3f((width)/30,0,0);
glEnd();
```

Figura 6: texturizado con una rotación de 180º

Existen varias formas de crear una ventana que contenga el render de nuestra aplicación una de ellas es usando las librerías de GLUT (OpenGL Utility ToolKit) que es una herramienta para crear la interface del render y la interacción de la escena con el teclado o mouse. GLUT es una buena herramienta para la portabilidad en distintos sistemas operativos. En esta tesis la aplicación de OpenGL se desarrolla en Windows y por ello se utilizará un ambiente de desarrollo como lo es Visual Studio 2005 con C++ y con ello crear la aplicación; se puede hacer comunmente de 2 formas; una es como una aplicación Win32 estandar y otra es como una aplicación MFC (Microsoft Foundation Class) . Aunque se comenzó el desarrollo de software en la plataforma con DevC++, la dificultad para programar las interfaces, no solamente en éste sino en otros proyectos de laboratorios del CCADET, nos llevó a buscar otras alternativas. La liberación de las licencias de Microsoft para académicos y estudientes ha permitido, desde fines de 2008 el uso de la plataforma Visual Studio. En el laboratorio de imágenes se están probando las plataformas de wxDevC++ y Visual Studio, la cuál seleccionamos para continuar el desarrollo, dado que portar los programas de DevC++ y librerías OpenGL resultó sencillo y rápido.

1.2.- Software comercial

1.2.1.- IDE Visual Studio: Aplicación OpenGL con MFC

MFC son unas librerias orientadas a objetos para la creación de ventanas en un entornos de Windows. Para nuestra aplicación de OpenGL con MFC se necesita crear una ventana de dialogo. Utilizando Visual Studio 2005 se crea un nuevo proyecto

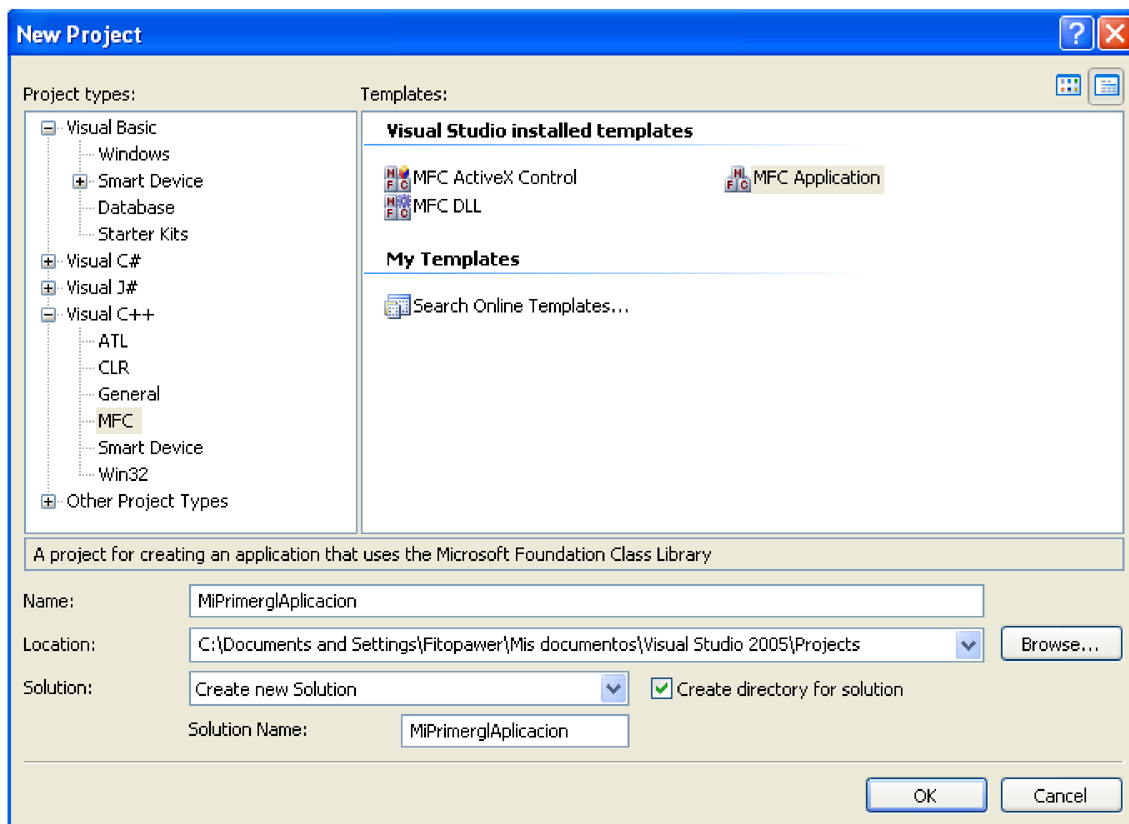

Figura 7: Ventana para crear un nuevo proyecto usando el framework de MFC

Se selecciona MFC y después se configura el tipo de aplicación. Para realizar nuestras las pruebas se escogió una ventana de Diálogo base y después se dejó que el MFC Wizard creara todas las clases y un único objeto que será la ventana principal. Obteniendo la ventana de diálogo de la Figura 8:

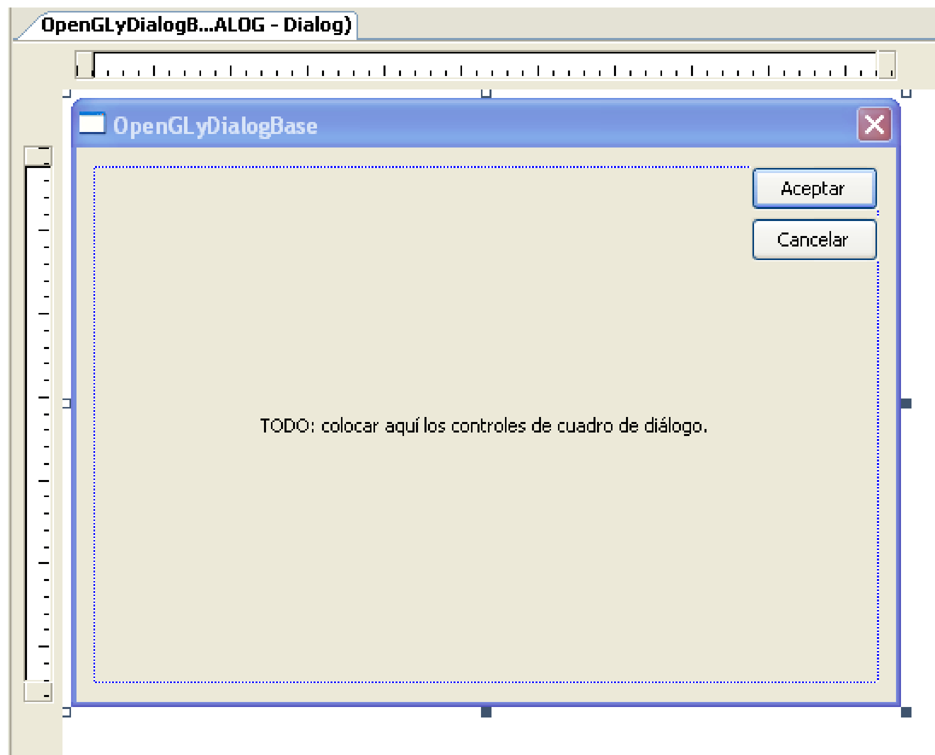

Figura 8: Ventana del tipo cuadro de diálogo base

Se agrega un tipo de control llamado Group Box en donde se va a renderizar la escena. Para lograr el renderizado se necesitan 3 funciones para la escena las cuales son las siguientes:

```
bool CrearGLWindow(HWND hWnd)
{
      GLuint       PixelFormat;
      hInstance = GetModuleHandle(NULL);
      hDrawWnd = hWnd;

      if(hDC=GetDC(hWnd),!hDC)
      {
            return FALSE;

      }

      pfd.nSize        =sizeof(PIXELFORMATDESCRIPTOR);
      pfd.nVersion       =1;
      pfd.dwFlags
      =PFD_DOUBLEBUFFER|PFD_DRAW_TO_WINDOW|PFD_SUPPORT_OPENGL;
      pfd.iPixelType           =PFD_TYPE_RGBA;
```

```
	pfd.cColorBits		=32;
	pfd.cRedBits		=0;
	pfd.cRedShift		=0;
	pfd.cGreenBits		=0;
	pfd.cGreenShift		=0;
	pfd.cBlueBits		=0;
	pfd.cBlueShift		=0;
	pfd.cAlphaBits		=0;
	pfd.cAlphaShift		=0;
	pfd.cAccumBits		=0;

	pfd.cAccumRedBits	=0;
	pfd.cAccumGreenBits	=0;
	pfd.cAccumBlueBits	=0;
	pfd.cAccumAlphaBits	=0;
	pfd.cDepthBits		=32;
	pfd.cStencilBits	=0;
	pfd.cAuxBuffers		=0;
	pfd.iLayerType		=PFD_MAIN_PLANE;
	pfd.bReserved		=0;
	pfd.dwLayerMask		=0;
	pfd.dwVisibleMask	=0;
	pfd.dwDamageMask	=0;

	if(PixelFormat=ChoosePixelFormat(hDC,&pfd),!PixelFormat)
	{
		MessageBox(NULL,"Unable to find the required pixel
format","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;

	}

	if(!SetPixelFormat(hDC,PixelFormat,&pfd))
	{
		//Failed
		MessageBox(hWnd,"Unable to set the required pixel
format","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;

	}
```

```
	if(hRC=wglCreateContext(hDC),!hDC)
	{
		//Failed
		MessageBox(NULL,"Unable to create Rendering
Context","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;
	}

	if(!wglMakeCurrent(hDC,hRC))
	{
		//Failed
		MessageBox(NULL,"Unable to bind Rendering with Device
Context","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;
	}
	RECT  WindowRect;

	GetClientRect(hWnd,&WindowRect);

	//Calculate drawing area
	width  = WindowRect.right - WindowRect.left;
	height = WindowRect.bottom - WindowRect.top;

	if (!InitGL())    // Initialize the OpenGL workspace
	{
		//Failed
		MessageBox(NULL,"Initialization
		Failed.","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;

	}



	return TRUE;
}
```

En la función de `CrearGLWindow(HWND hWnd)` es donde se va a inicializar y configurar todas las caracterizticas de la ventana donde se va a renderizar nuestra escena, esta se compone primero de obtener el manejador del control “Group Box” que es el parámetro

que entra a la función y donde dibujaremos, lo segundo es llenar una estructura del tipo PIXELFORMATDESCRIPTOR que es la configuración de el rasterizado para los pixeles, después se inicializa el “Group Box” con esa configuración utilizando su manejador para que por último se inicie los valores que tendrá por default la escena y esta función es `InitGL();`

```
int InitGL(void)
{
	/*Initialize the OpenGL workspace*/

	glLoadIdentity ();

	gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,10
0.0f;
	glMatrixMode (GL_MODELVIEW);

	glLoadIdentity ();
	//Enable features
	glEnable(GL_DEPTH_TEST);
	glEnable(GL_POLYGON_MODE);
	glEnable(GL_POLYGON_SMOOTH);
	glEnable(GL_POLYGON_STIPPLE);
	glDepthFunc(GL_LEQUAL);
	glClearColor(0.0f,0.0f,0.0f,0.0f);
	return TRUE;}
```

La tercera función indispensable para dibujar es una llamada `DibujarEscena(void)` la cuál es donde agregamos los vértices, polígonos, figuras, luces, trasnformaciones y todo aquello que queremos mandar a dibujar. Esta función es muy importante ya que por cada cambio que hagamos a un control de nuestra ventana de diálogo esta se debe llamar para ejecutar de nuevo las instrucciones de dibujo y se vea el cambio en el despliegue.

En la función:

```
BOOL CVisualizadorVersion20betaDlg::OnInitDialog()
```

Se llama a la función CrearGlWindow(..) y lo más importante para que funcione es que el parámetro que entra a la función es el manejador del “Group Box” y se obtiene con la función `GetDlgItem(RENDER)->m_hWnd` en la cual GetDlgItem toma el objeto “Group Box” utilizando si ID=RENDER (en este caso el ID del “Group Box” lo llamamos RENDER) y como GetDlgItem devuelve un apuntador al objeto entonces necesitamos que apunte a m_hWnd que es el manejador del objeto “Group Box”

```
CrearGLWindow(GetDlgItem(RENDER)->m_hWnd);
```

Ya para mandar a dibujar por primera vez nuestra escena llamamos a la función DibujarEscena(); en la función:

```
void CVisualizadorVersion20betaDlg::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // Contexto de dispositivo para
dibujo

		SendMessage(WM_ICONERASEBKGND,
reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

		// Centrar icono en el rectángulo de cliente
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Dibujar el icono
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CDialog::OnPaint();
	}
	DibujarEscena();
}
```

Con esto se dibuja por primera vez nuestra escena. Como queremos agregar controles a nuestra ventana y los valores que se obtengan sean enviados a nuestra escena, primero debemos agregar un control ya sea un botón, checkbox, scroll, etc… se arrastra el control desde el toolbox de Visual Studio

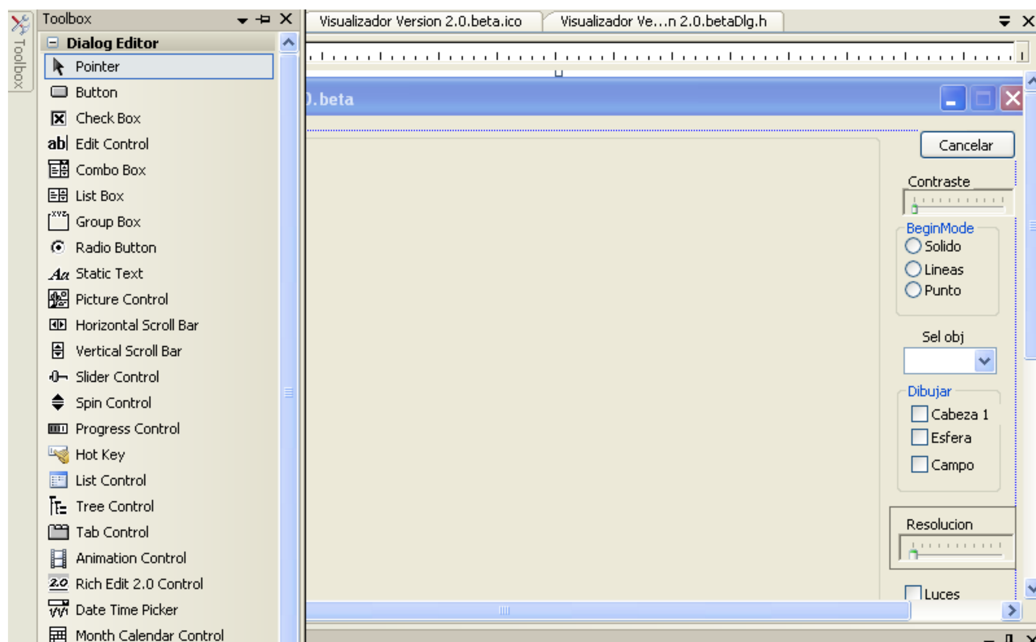

Figura 9: Ventana con una caja de grupo donde se renderizará la escena y con sus controles para transformar el modelo

Ya colocado el control con el botón derecho del mouse se de un click encima y se elije la opción “add variable” y se le da un nombre a la nueva variable, el tipo de variable lo da automáticamente el “Add Member Variable Wizard” así con esta variable podemos acceder a los datos del control.

Estos controles son muy útiles para poder controlar nuestra aplicación en OpenGL, cabe señalar que por cada cambio en el control se debe mandar a dibujar de nuevo con la función DibujarEscena(); por ejemplo si queremos hacer rotaciones o traslaciones con la posición del mouse y un click de alguno de sus botones el código quedaría de la siguiente forma:

```
void CVisualizadorVersion20betaDlg::OnMouseMove(UINT nFlags,
CPoint point)
{
```

```
        // Aqui se recibe los mensajes de la posición del MOUSE

        switch(nFlags){
              case MK_LBUTTON:
                    GetRotacion(point.x,point.y);
                    DibujarEscena();
                    break;
              case MK_RBUTTON:
                    GetTraslacion(point.x,point.y,0);
                    DibujarEscena();
                    break;
              case MK_MBUTTON:
                    GetTraslacion(0,0,point.y);
                    DibujarEscena();
                    break;
              default:GetBandera(false);
        }
        CDialog::OnMouseMove(nFlags, point);
  }
```

OnMouseMove es una función asociada a la clase que se ejecuta cuando existe un mensaje del sistema operativo sobre la ventana del tipo `ON_WM_MOUSEMOVE()` y esta función recibe 2 parámetros que uno es nFlag el cual es el que indica si se ha apretado un botón del mouse y el segundo es CPoint que es un objeto con 2 atributos enteros que son las coordenadas en pixeles de la posición del cursor del mouse.

Los mensajes en Windows son los eventos que suceden al interactuar el usuario con el sistema operativo. Estos eventos se van encolando deacuerdo a su aparición, los eventos van desde apretar un botón del teclado, dar un click encima de un botón, un cambio de posición del mouse etc…