CAPITULO 2

Transformaciones Geométricas y de Modelación en tres dimensiones.

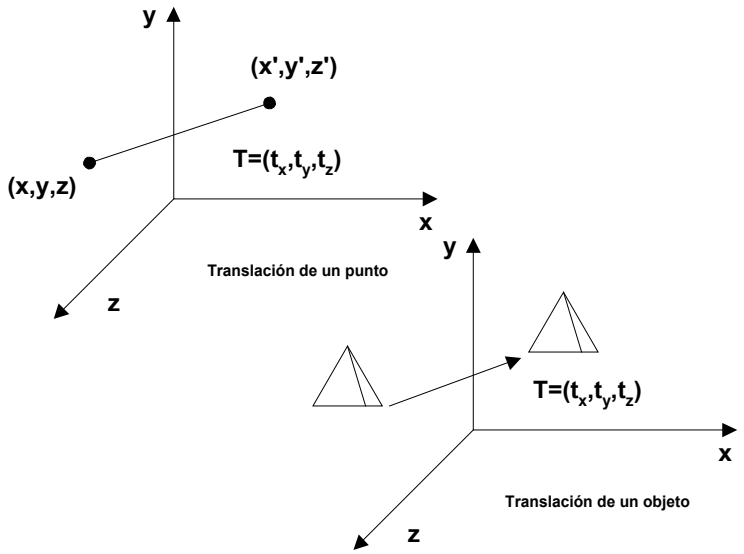
2.1 Traslación

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = T * P$$

Esto equivale a:

$$x' = x + t_x$$
$$y' = y + t_y$$
$$z' = z + t_z$$



Ejemplo Utilizando OpenGl

La traslación para el caso de tres dimensiones es similar al caso de dos dimensiones, esto es, por medio de la función glTranslate. Supongamos que deseamos dibujar una esfera en la posición (1,1,1). Para realizar esto podemos utilizar la función de dibujo de esfera de la librería GLU que viene con Opengl, su prototipo es el siguiente:

```
void gluSphere(GLUquadricObj *qobj,
GLdouble radius,
GLint slices,
GLint stacks
);
```

La esfera por defecto se dibuja en la posición (0,0,0), luego para dibujar ésta en la posición (1,1,1) debemos hacer uso de la traslación. El código de nuestro programa queda de la siguiente forma:

```
glPushMatrix();
  glTranslatef(1.0f,1.0f,1.0f);
  glColor3f(1.0f, 0.0f, 0.0f);
  gluQuadricNormals(Sphere, GLU_SMOOTH);
  gluSphere(BlueSphere, 1, 50, 50);
glPopMatrix();
```

Al ejecutar nuestro programa obtenemos la salida indicada en la figura 2.1. Se han efectuado algunas rotaciones para mejorar su apariencia 3D.

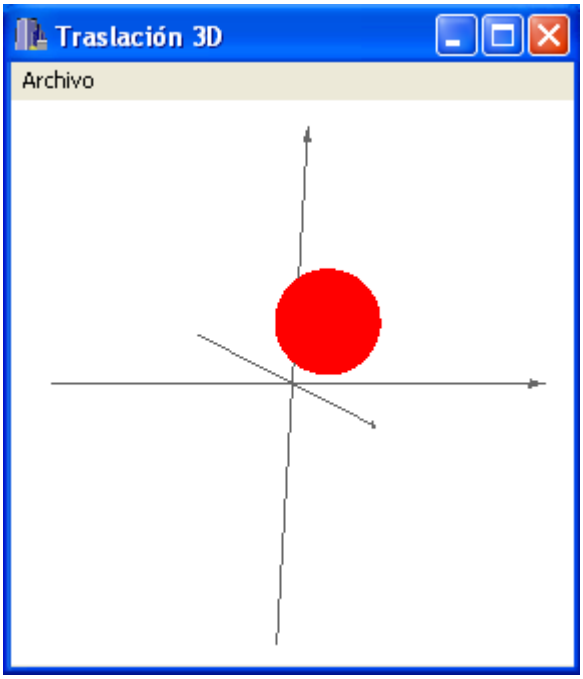


Figura 2.1: Esfera en (1,1,1)

Supongamos que además necesitamos dibujar un cilindro en la posición (2,2,2), nuestro código sería el siguiente:

```
glPushMatrix();
  glTranslatef(1.0f,1.0f,1.0f);
  glColor3f(1.0f, 0.0f, 0.0f);
  gluSphere(BlueSphere, 1, 50, 50);
glPopMatrix();
glPushMatrix();
  glTranslatef(2.0f,3.0f,2.0f);
  glColor3f(1.0f, 0.0f, 0.0f);
  gluCylinder(Cylinder,2,2,3,20,20);
glPopMatrix();
```

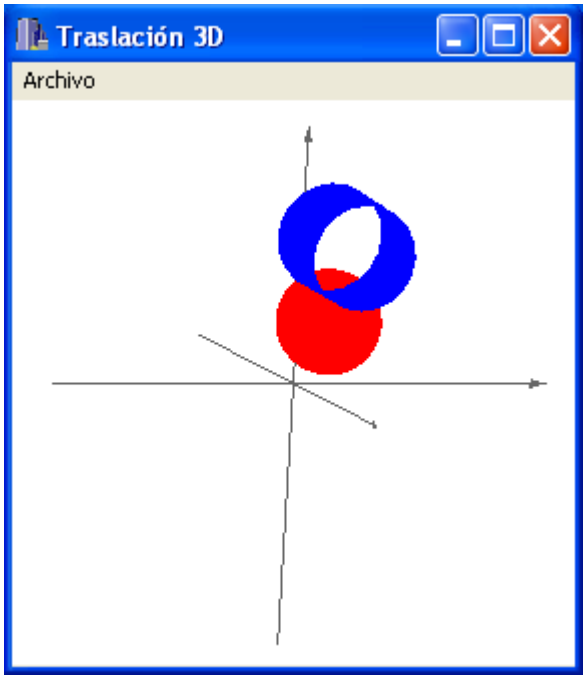


Figura 2.2: Esfera y Cilindro

La apariencia puede mejorarse aplicando opciones de iluminación y textura que veremos más adelante

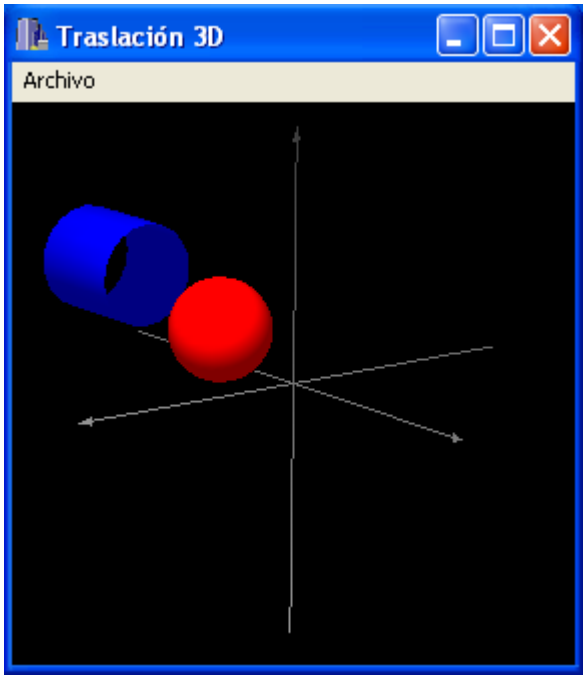
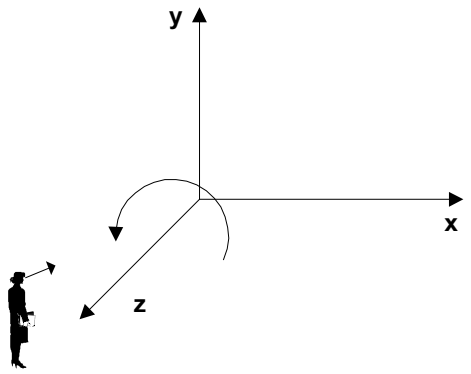


Figura 2.3: Esfera y Cilindro opción iluminación

2.2 Rotación

Para generar una transformación de rotación de un objeto debemos designar un eje de rotación (acerca del cual se rota el objeto) y la cantidad de rotación angular. A diferencia del caso bi-dimensional donde se trabaja en el plano *xy*, una rotación se puede especificar alrededor de cualquier línea en el espacio. Las más fáciles son aquellas paralelas a algún eje del sistema de coordenadas.

2.2.1 Rotación alrededor del eje z



Las ecuaciones son fácilmente derivadas del caso 2-dimensiones.

$$x' = x\cos\theta - ysen\theta$$
$$y' = xsen\theta + y\cos\theta$$
$$z' = z$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -sen\theta & 0 & 0 \\ sen\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_z(\theta) * P$$

las ecuaciones de transformación de Rotación para los otros dos ejes de coordenadas se obtienen con una simple permutación de los parámetros de coordenadas x, y, z como:

$$x \to y \to z \to x$$

2.2.2 Rotación alrededor del eje x

$$y' = y\cos\theta - zsen\theta$$
$$z' = ysen\theta + z\cos\theta$$
$$x' = x$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -sen\theta & 0 \\ 0 & sen\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_x(\theta) * P$$

2.2.3 Rotación alrededor del eje y

$$z' = z\cos\theta - xsen\theta$$
$$x' = zsen\theta + x\cos\theta$$
$$y' = y$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & sen\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sen\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_y(\theta) * P$$

Ejemplo Utilizando OpenGl

La rotación para el caso de tres dimensiones es similar al caso de dos dimensiones, esto es, por medio de la función glRotate. Supongamos que deseamos rotar la figura anterior en torno a un vector cualquiera, bastaría agregar glRotate para rotar la escena en torno al eje deseado. El siguiente código muestra como efectuar una animación, provocando en cada frame de la animación un rotación en torno al vector (0,0,1), el resultado es una rotación dinámica de nuestra escena.

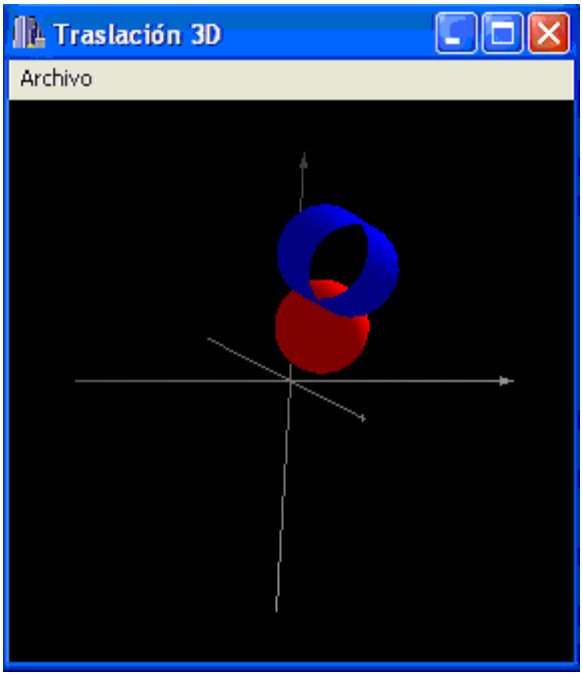


Figura 2.3: Esfera y Cilindro con Rotación alrededor de (0,0,1)

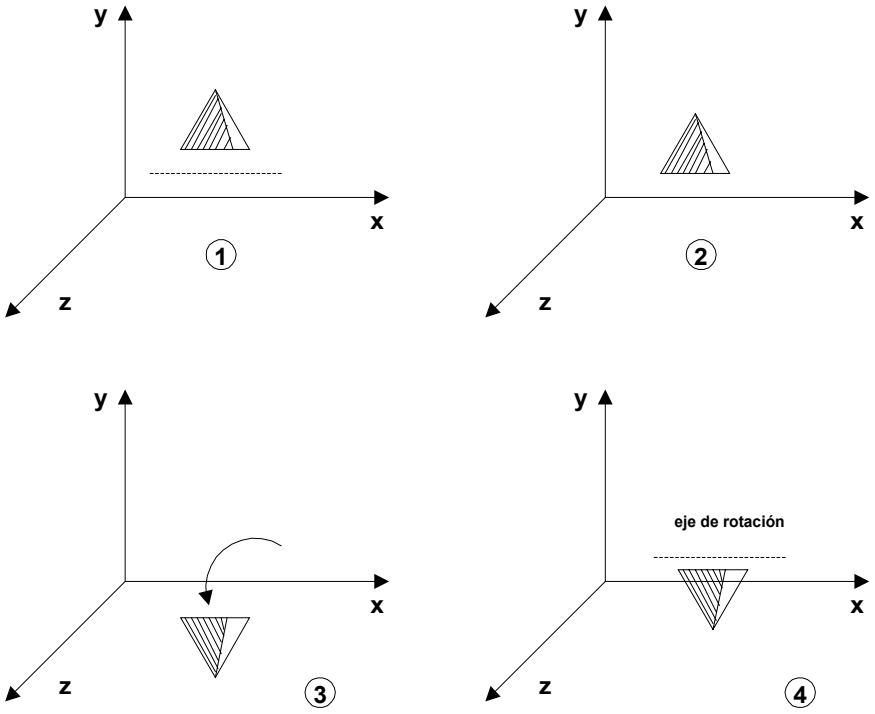
Las funciones más importantes de nuestra aplicación son la siguientes:

```
void __fastcall TForm1::OglWindowOglPaint(TObject *Sender)
{

      glPushMatrix();
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        iluminacion();
        ejes();
        glRotatef(rotacion,0,0,1);

//    Esfera
       glPushMatrix();
              glTranslatef(1.0f,1.0f,1.0f);
              glColor3f(1.0f, 0.0f, 0.0f);
              gluQuadricDrawStyle(Sphere, GLU_FILL);
              gluQuadricNormals(Sphere, GLU_SMOOTH);
              gluSphere(Sphere, 1, 50, 50);
      glPopMatrix();

//      Cilindro
      glPushMatrix();
              glColor3f(0.0f, 1.0f, 0.0f);
              glTranslatef(2.0f,2.0f,2.0f);
              glColor3f(0.0f, 0.0f, 1.0f);
              gluCylinder(Cylinder,1,1,2,20,20);
       glPopMatrix();
    glPopMatrix();
}


void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
  rotacion+=10.0;
  OglWindow->OglRepaint();
}
```

2.3 Rotaciones Generales en tres dimensiones

Caso1: eje es paralelo a un eje de coordenadas.

Trasladar el objeto de tal manera que el eje de rotación coincida con el eje de coordenadas.
Realizar la rotación alrededor del eje.
Trasladar el objeto de tal manera que el eje de rotación vuelva a su lugar de origen.

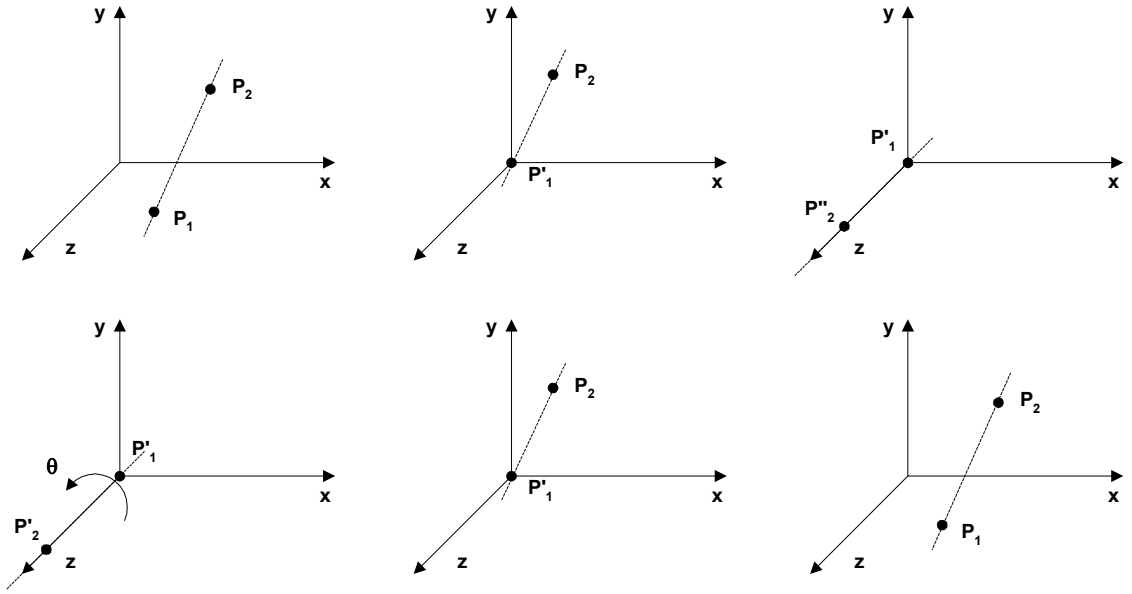


$$P' = T^{-1} * R_x(\theta) * T * P \quad (de\ cualquier\ \ punto\ de\ la\ \ figura)$$

Caso 2: Eje de rotación *no* es paralelo a algún eje de coordenadas.

Trasladar el objeto de tal manera que el eje de rotación pase por el origen del sistema de coordenadas.

Rotar el objeto de tal manera que el eje de rotación coincida con uno delos ejes de coordenadas.
Realizar la rotación alrededor del eje.
Aplicar rotación inversa de tal manera de devolver el eje de rotación a su orientación original.
Aplicar la traslación inversa de tal manera de devolver el eje de rotación a su posición original.

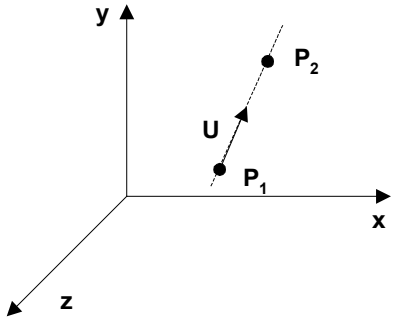


La coincidencia del eje de rotación puede hacerse con cualquiera de los tres ejes. Elegimos el eje z.

Se define un vector $V=P_2-P_1$, que son puntos del eje de rotación.

$$V = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

Se define entonces un vector unitario *U* como se indica.



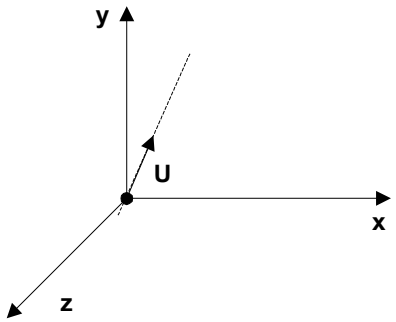
$$U = \frac{V}{|V|} = (a, b, c)$$

Donde:

$$a = \frac{x_2 - x_1}{|V|}; \qquad b = \frac{y_2 - y_1}{|V|}; \qquad c = \frac{z_2 - z_1}{|V|}$$

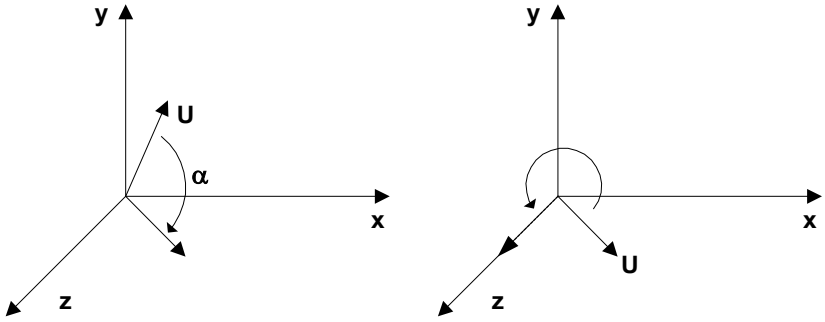
El primer paso en la transformación es efectuar un traslado al origen. Esto se obtiene con:

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



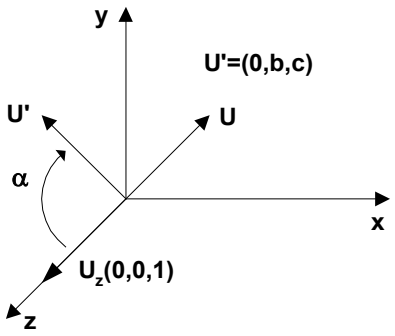
Ahora se necesita la transformación que ponga el eje de rotación en el eje z.

Para ello se rota *U* alrededor del eje x *y* se le coloca en el plano xz, y posteriormente se rota en torno al eje *y*, y se le hace coincidir con el eje z. Esto se ve en las siguientes figuras:



Necesitamos encontrar los valores de cosα y senα para la matriz $R_x(\alpha)$ para poner el eje en el plano xz.

Sea *U'* la proyección del vector *U* en el plano zy.



$$\cos\alpha = \frac{U'^{*}U_z}{|U'|*|U_z|} = \frac{c}{d}$$

$$donde \quad d = \sqrt{b^2 + c^2}$$

Igualmente el senα se puede determinar por el producto cruz de *U'* y U_z.

$$U'xU_z = U_x|U'||U_z|sen\alpha$$

ó

$$U'xU_z = U_x * b$$

Igualando ambas ecuaciones se tiene:

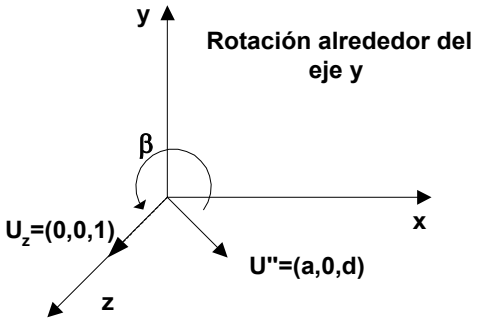
$$dsen\alpha = b$$

$$sen\alpha = \frac{b}{d}$$

Luego:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & \frac{-b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ahora se necesita $R_y(\beta)$ según la figura.



$$\cos\beta = \frac{U''*U_z}{|U''||U_z|} = d$$

$$donde \quad |U_z| = |U''| = 1$$

$$|U''| = \sqrt{a^2 + \left(\sqrt{b^2 + c^2}\right)^2} = \sqrt{a^2 + b^2 + c^2} = 1 \qquad por\ ser\ vector\ unitario$$

El producto cruz:

$$U'' x U_z = U_y |U''||U_z| sen\beta$$

ó

$$U'' x U_z = U_y * (-\alpha)$$

Se encuentra que:

$$sen\beta = -\alpha$$

Luego:

$$R_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y la rotación alrededor del eje z:

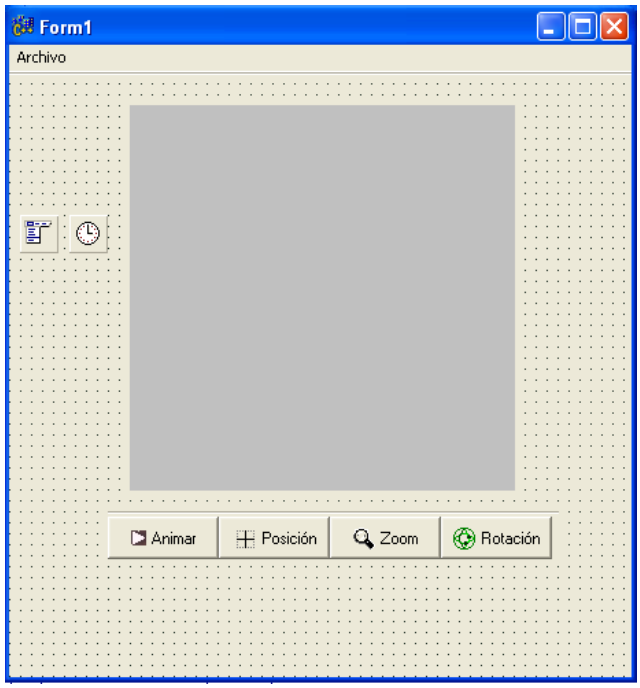
$$R_z(\theta) = \begin{bmatrix} \cos\theta & -sen\theta & 0 & 0 \\ sen\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finalmente la transformación total que debe realizar para rotar alrededor de un eje arbitrario es:

$$R(\theta) = T^{-1} * R_x{}^{-1}(\alpha) * R_y{}^{-1}(\beta) * R_z(\theta) * R_y(\beta) * R_x(\alpha) * T$$

Ejemplo Utilizando OpenGl

En los ejemplos anteriores hemos manejado las rotaciones y las traslaciones en forma independiente y sin aprovechar las herramientas que nos proporciona un ambiente de programación visual como Builder. En el siguiente ejemplo crearemos una aplicación que permita modificar los parámetros de traslación, rotación, zoom y posición de un par de objetos en la pantalla, estos parámetros serán modificados por controles windows. La siguiente figura muestra el diseño de nuestra aplicación.



En la figura se pueden apreciar los siguientes componentes:

1.- Menu: En este caso solo la opción Archivo y Salir, para una salida más elegante de nuestro programa
2.- Timer: Para generar nuestra animación
3.- Botón Animar: Para iniciar nuestra animación
4.- Botón Posición: Para posicionar nuestro dibujo
5.- Botón Zoom: Para controlar el acercamiento y alejamiento a nuestro dibujo.
6.- Botón Rotación: Para rotar la figura en algún ángulo deseado
7.- Componente panel Opengl: Aquí se "renderizara" nuestro dibujo.

El resultado final de nuestra aplicación se puede apreciar en la siguiente figura:

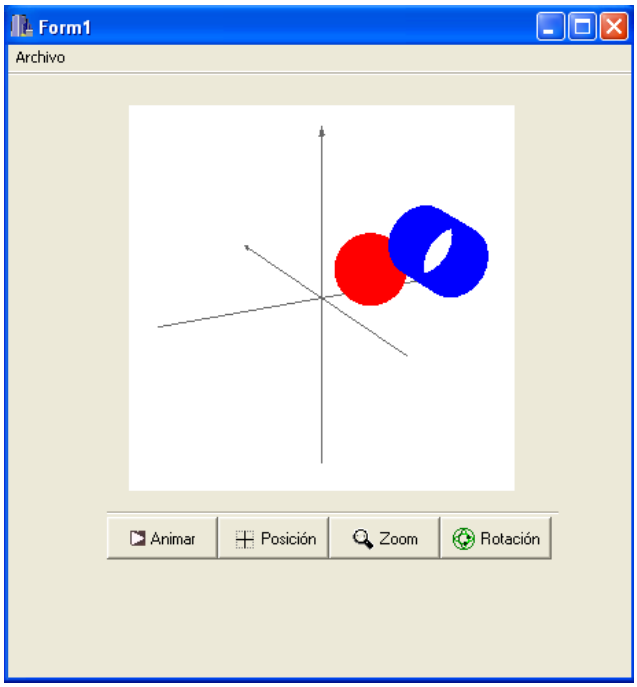


Figura 2.4: Aplicación con controles windows

El código principal de nuestro programa es el siguiente:

```
void __fastcall TForm1::OglWindowOglResize(TObject *Sender)
{
    Camera.OnSetViewport((GLshort)OglWindow->Width,
    (GLshort)OglWindow->Height);
    glViewport(0,0,OglWindow->Width,OglWindow->Height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    Camera.OglPerspectiveTransformation();
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
void __fastcall TForm1::OglWindowOglPaint(TObject *Sender)
{
    glPushMatrix(); // Apila identidad en tope de ModelView Stack
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    Camera.OglViewTransformation();
    glRotatef(angulo,1,1,0);
    //      ESFERA

      glPushMatrix();
            glTranslatef(1.0f,1.0f,1.0f);
            glColor3f(1.0f, 0.0f, 0.0f);
            gluQuadricDrawStyle(Sphere, GLU_FILL);
            gluQuadricNormals(Sphere, GLU_SMOOTH);
            gluSphere(Sphere, 1, 50, 50);
      glPopMatrix();
```

```
      //      CILINDRO

       glPushMatrix();
              glColor3f(0.0f, 1.0f, 0.0f);
              glTranslatef(2.0f,2.0f,2.0f);
              glColor3f(0.0f, 0.0f, 1.0f);
              gluCylinder(Cylinder,1,1,2,20,20);
       glPopMatrix();
       ejes();

glPopMatrix();
}
//---------------------------------------------------------------------------


void __fastcall TForm1::FormCreate(TObject *Sender)
{
  OglCamera::LoadCursors();
  Camera.OglWindow = OglWindow;
  Camera.SetTarget( 0.0, 0.0, 0.0);
  Camera.SceneRadius = 1.5;
  Camera.Twist = 0.0;
  Camera.ZoomMin = 0.1; Camera.ZoomMax = 5;
  Camera.Zoom  = 0.28;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::ZoomButtonClick(TObject *Sender)
{
  Camera.MoveMode = cmLinearInOut;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::OglWindowMouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
  Camera.OnMouseDown(Button, X, Y);
}
//---------------------------------------------------------------------------

void __fastcall TForm1::OglWindowMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
  Camera.OnMouseUp(Button);
}
//---------------------------------------------------------------------------

void __fastcall TForm1::OglWindowMouseMove(TObject *Sender,
      TShiftState Shift, int X, int Y)
{
  Camera.OnMouseMove(X,Y);
}
//---------------------------------------------------------------------------

void __fastcall TForm1::AnimeButtonClick(TObject *Sender)
{
```

```
  Timer1->Enabled = ! Timer1->Enabled;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::PanButtonClick(TObject *Sender)
{
  Camera.MoveMode = cmLinearPerpendicular;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::RotateButtonClick(TObject *Sender)
{
  Camera.MoveMode = cmArcRotate;
}
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------

void __fastcall TForm1::Salir1Click(TObject *Sender)
{
  exit(0);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
  angulo+=5.0;
                // Increase The Rotation Variable For The Triangle ( NEW )
  OglWindow->OglRepaint();
}
//---------------------------------------------------------------------------
```

Más adelante podemos agregar más efectos de iluminación y textura a nuestro ejemplo que le darán una mejor apariencia

2.4 Escalamiento

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = S * P$$

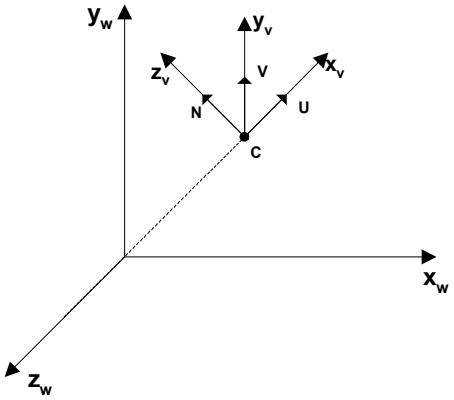
$$x' = x * S_x; \quad y' = y * S_y; \quad z' = z * S_z$$

Para escalar respecto a un punto fijo (x_f, y_f, z_f) se deben seguir los siguientes pasos:

1. Trasladar el punto fijo al origen.
2. Escalar el objeto relativo al origen de coordenadas.
3. Trasladar el punto fijo a su posición original.

$$T(x_f, y_f, z_f) * S(S_x, S_y, S_z) * T(-x_f, -y_f, -z_f) = \begin{bmatrix} S_x & 0 & 0 & (1-S_x)x_f \\ 0 & S_y & 0 & (1-S_y)y_f \\ 0 & 0 & S_z & (1-S_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.5 Determinación de un sistema de coordenadas para la visualización de un objeto en tres dimensiones.



Para el esquema anterior, se debe determinar *C, V, U* y *N* como vectores unitarios.

Se desea que un punto en coordenadas Globales se mapee a coordenadas de Vista. En general *N, U, V* se llama marco de referencia o Cámara Visual.

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = M * \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

$$P_v = M * P_w$$

Donde:

$$M = B * T$$

T permite trasladar el sistema *N, U* y *V* al origen del sistema de coordenadas globales.

B convierte cualquier vector del sistema de coordenadas globales al sistema de coordenadas *N, U, V* de cámara.

En particular debe convertir:

$$U \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad V \rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$N \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Donde *U, V* y *N* están expresados en coordenadas globales. Es decir, debemos tener:

$$B * \begin{bmatrix} U_x \\ U_y \\ U_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad B * \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$B * \begin{bmatrix} N_x \\ N_y \\ N_z \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Además, por ortonormalidad de los vectores *N, V, U* se debe cumplir que:

$$U_x{}^2 + U_y{}^2 + U_z{}^2 = 1$$
$$V_x{}^2 + V_y{}^2 + V_z{}^2 = 1$$
$$N_x{}^2 + N_y{}^2 + N_z{}^2 = 1$$

y también:

$$U_x * V_x + U_y * V_y + U_z * V_z = 0$$
$$U_x * N_x + U_y * N_y + U_z * N_z = 0$$
$$V_x * N_x + V_y * N_y + V_z * N_z = 0$$

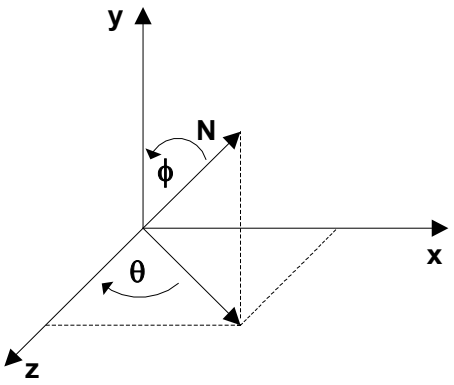
de aquí se deduce que:

$$B = \begin{bmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} U_x \\ U_y \\ U_z \\ 1 \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix} \begin{bmatrix} N_x \\ N_y \\ N_z \\ 1 \end{bmatrix}$$

Se debe ahora determinar los vectores *U, V* y *N* para tener el sistema completo.

2.6 Determinación de N

N puede expresarse en función de los ángulos ϕ y θ como sigue:



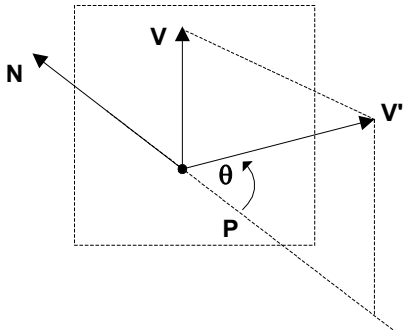
$$N_x = sen\phi * \cos\theta$$
$$N_y = sen\phi * sen\theta$$
$$N_z = \cos\phi$$

N apunta al origen del sistema de coordenadas Globales.

2.7 Determinación de V

La determinación de *V* no es directa. Debe ser normal a *N* que se ha elegido en forma arbitraria.

Elegimos un *V'* cualquiera que no necesariamente pueda ser normal a *N*.



De la figura se tiene:

$$V = V' - P$$
$$V' * N = |V'| \cos\theta = P \qquad es\ un\ módulo$$

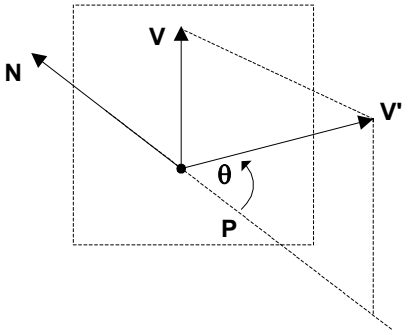
le damos la dirección de *N* multiplicando por *N*, es decir:

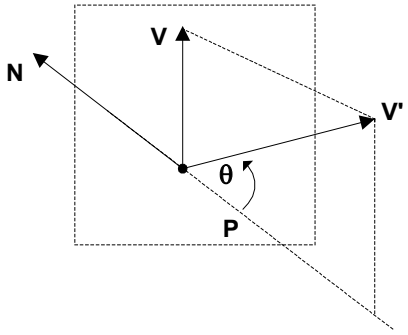
$$V = V' - (V' * N) * N$$

V debe ser unitario

2.8 Determinación de U

Una vez que se determina *N* y *V* es fácil encontrar *U*





Un método más general:

Dado vectores *N* y *V* se calcula

$$n = \frac{N}{|N|} = (n_1, n_2, n_3) \qquad vector\ normal\ unitario$$

$$u = \frac{VxN}{|VxN|} = (u_1, u_2, u_3)$$

y

$$v = nxu = (v_1, v_2, v_3)$$

Este método automáticamente ajusta la dirección de *V* de tal manera que *v* es perpendicular a *n*.

Luego:

$$B = \begin{bmatrix} u1 & u2 & u3 & 0 \\ v1 & v2 & v3 & 0 \\ n1 & n2 & n3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = R(matriz\ de\ rotación)$$

$$M_{wc,vc} = R * T$$

Esta transformación se aplica entonces a descripciones de coordenadas de objetos en la escena para transferirlo a marco de referencia o cámara.

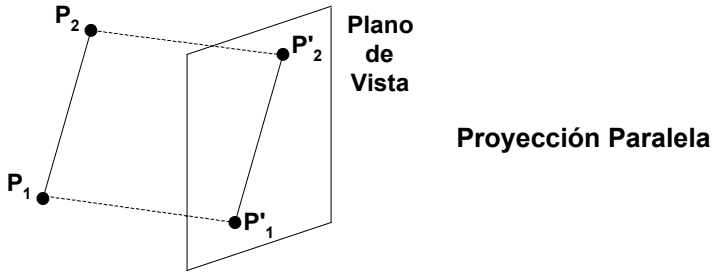
2.9 Proyecciones

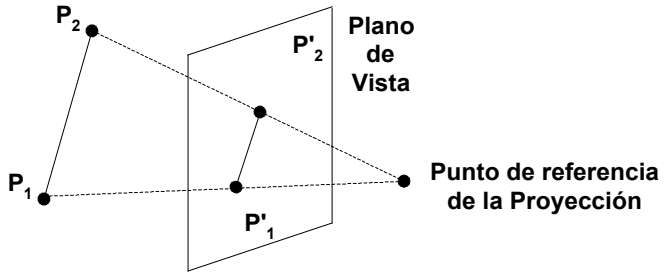
Una vez que los objetos en coordenadas globales se han convertido a coordenadas de cámara se pueden proyectar los objetos en tres dimensiones en un plano bi-dimensional.

Existen dos proyecciones básicas:

- Proyección Paralela.
- Proyección en Perspectiva.

Ambas se muestran en las siguientes figuras:

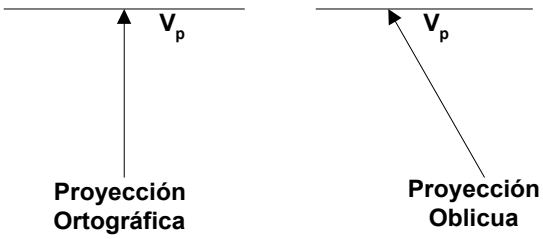




Proyección en Perspectiva

2.9.1 Proyecciones Paralelas

Cuando la proyección es perpendicular al plano de vista se tiene una proyección paralela ortográfica. En cualquier otro caso, la proyección es paralela oblicua.

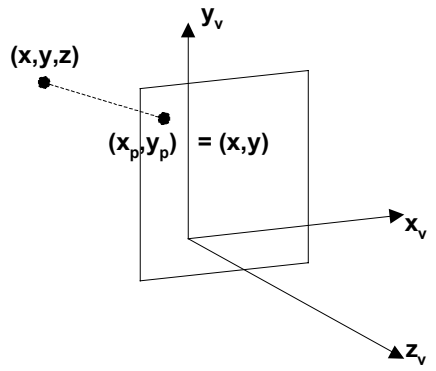


Las proyecciones ortográficas se usan más a menudo para generar el frente, codo y vistas de arriba y debajo de un objeto.

Las ecuaciones de transformación para una proyección paralela son directas.

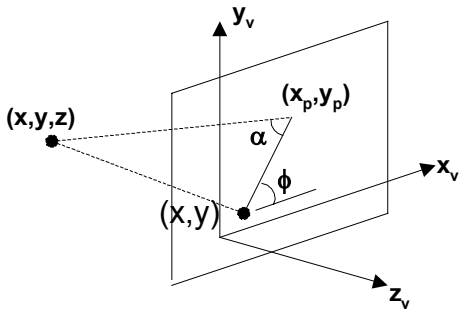
Si el plano de vista está en una posición z_{vp} a lo largo del eje z_v se tiene:

$$x_p = x \qquad y_p = y$$



Una proyección oblicua se obtiene proyectando puntos a lo largo de líneas paralelas que no son perpendiculares al plano de proyección.

En algunas aplicaciones se da un vector de proyección oblicua con dos ángulos α y ϕ como se muestra.



$$x_p = x + L\cos\phi$$
$$y_p = y + Lsen\phi$$

L depende del ángulo α y la coordenada z.

$$\tan\alpha = \frac{z}{L}$$

luego:

$$L = \frac{z}{\tan\alpha} = zL_1$$

Así,

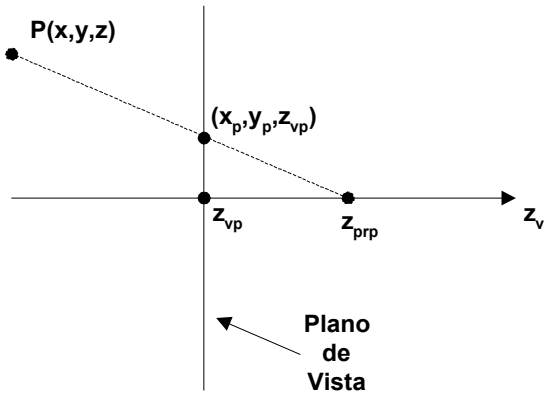
$$x_p = x + z(L_1\cos\phi)$$
$$y_p = y + z(L_1 sen\phi)$$

La matriz de transformación para producir una proyección paralela en el plano $x_v y_v$ se puede escribir como:

$$Mparalelo = \begin{bmatrix} 1 & 0 & L_1 \cos\phi & 0 \\ 0 & 1 & L_1 sen\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.9.2 Proyección en Perspectiva

Para obtener una proyección en perspectiva de un objeto tri-dimensional, se transforman los puntos a lo largo de líneas de proyección que se encuentran en un punto de referencia.



Se pueden escribir ecuaciones que describen las posiciones de coordenadas a lo largo de la línea de proyección en perspectiva en forma paramétrica:

$$x' = x - xu$$
$$y' = y - yu$$
$$z' = z - (z - z_{prp})u$$

u toma los valores 0 a 1 y la posición (x', y', z') representa un punto en cualquier lugar de la línea de proyecciones.

En el plano de vista $z' = z_{vp}$ y se puede entonces encontrar u:

$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

Sustituyendo este valor de u en x' e y' se obtiene las ecuaciones de transformación en perspectiva.

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = x\left(\frac{dp}{z - z_{prp}}\right)$$
$$yp = y\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = y\left(\frac{dp}{z - z_{prp}}\right)$$

Donde $dp = z_{prp} - z_{vp}$ es la distancia del plano de vista al punto de referencia de proyección.

Usando una representación de coordenadas homogéneas en tres dimensiones se puede escribir:

$$\begin{bmatrix} xh \\ yh \\ zh \\ h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{z_{vp}}{dp} & -z_{vp}\left(\frac{z_{prp}}{dp}\right) \\ 0 & 0 & \frac{1}{dp} & -\frac{z_{prp}}{dp} \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Donde

$$h = \frac{z - z_{prp}}{dp}$$

y

$$x_p = \frac{x_h}{h}, \qquad y_p = \frac{y_h}{h}$$

La coordenada z se retiene para problemas de visibilidad.

2.9.3 Casos especiales

Caso 1

El plano de vista es el plano *uv* de cámara, entonces z_{vp}=0, luego:

$$x_p = x\left(\frac{z_{prp}}{z - z_{prp}}\right) = x\left(\frac{1}{\frac{z}{z_{prp}} - 1}\right)$$

$$y_p = y\left(\frac{z_{prp}}{z - z_{prp}}\right) = y\left(\frac{1}{\frac{z}{z_{prp}} - 1}\right)$$

Caso 2

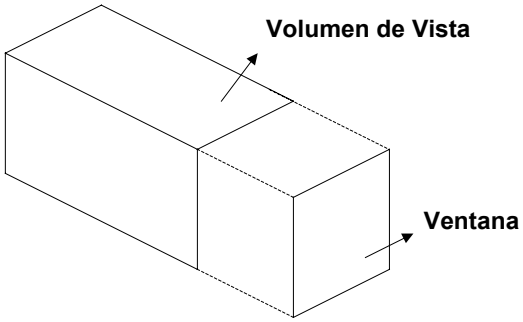
El punto de referencia de proyección está en el origen del sistema de coordenadas de vista.

Aquí z_{prp}=0

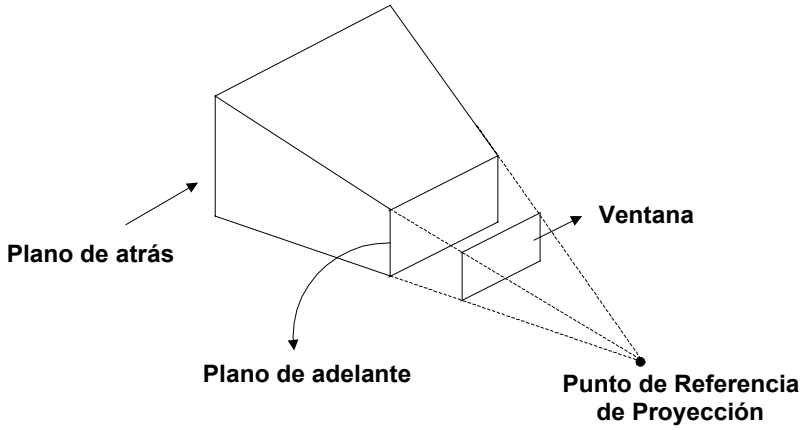
$$x_p = x\left(\frac{-z_{vp}}{z}\right) = x\left(\frac{-1}{\frac{z}{z_{vp}}}\right)$$

$$y_p = y\left(\frac{-z_{vp}}{z}\right) = y\left(\frac{-1}{\frac{z}{z_{vp}}}\right)$$

Volúmenes de vista



Proyección Paralela



Ejemplo Utilizando OpenGl

OpenGl soporta dos tipos de proyecciones y permite utilizar diferentes expresiones para definir la posición de la cámara y hacía donde mira.

La función gluLookAt() permite definir de forma especifica donde se situará, hacía donde mirara está, y cual va a ser el orden de los ejes de coordenadas. Esta sentencia tiene nueve argumentos que describen tres puntos, los valores de x0, y0, z0, representa el punto hacia donde mira la cámara virtual, este punto normalmente se identifica en el origen de coordenadas (0,0,0), ahora bien, podemos definir nosotros el punto que más propicio sea para nuestra escena.

Los siguientes tres argumentos representan el punto donde se situará la cámara de visualización, estas coordenadas no deben coincidir con el punto al cual miramos.

Las últimas tres coordenadas representan el vector de vista hacía arriba, es decir, indica cual será el vector cuya dirección será hacia arriba, "apuntará hacia la parte superior del monitor".

glOrtho() se utiliza para especificar una proyección ortográfica. Este tipo de proyección define un volumen de vista rectangular, concretamente define un paralelepípedo de tamaño infinito, este hecho nos lleva a definir una serie de planos de corte para detallar con exactitud el volumen de vista. Su formato es el siguiente:

glOrtho(xwmin, xwmax, ywmin, ywmax, pcerca, plejos);

Estos seis argumentos definen la ventana de visualización y los planos de corte tanto cercano como lejano. El objeto se visualizará entre los dos planos de recorte, en el caso que sobrepase estos planos se recortará, y si el objeto es tan grande que la ventana de visualización esta dentro de él, no se visualizará nada quedando la pantalla en negro.

glFrustum() se utiliza para definir una proyección perspectiva, se define de forma similar a la proyección ortográfica, pero con la diferencia que la proyección perspectiva define como volumen de vista una pirámide, en consecuencia el objeto a visualizar tiene un aspecto mucho más realista. Su formato es el siguiente:

glFrustum(xwmin, xwmax, ywmin, ywmax, pcerca,plejos);

Como vemos esta sentencia se define de forma similar a la utilizada para definir proyecciones paralelas, de igual forma que anteriormente definimos planos de corte para limitar el volumen de vista, que en este caso al ser una proyección perspectiva definirá un tronco piramidal.

gluPerpespective() es una alternativa a la función ***glFrustum***, la diferencia entre ambas está en la forma de definir la ventana de visualización. Si en la sentencia ***glFrustum*** definimos los dos vértices necesarios de la ventana, en la sentencia ***glPerpestive*** solamente definiremos el ángulo de apertura de la cámara y la relación entre el largo y ancho del plano cercano de corte. Su formato es el siguiente:

gluPerpective(apertura, aspect, pcerca, plejos);

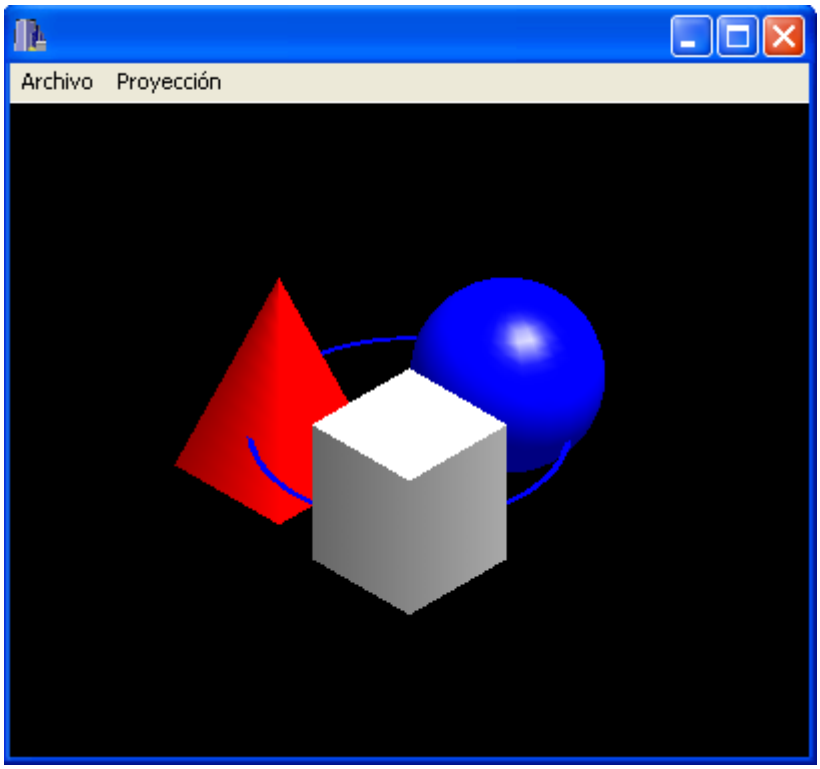
Apertura corresponde al ángulo de apertura de la cámara virtual, este ángulo puede tomar valores comprendidos entre 0º y 180º. El valor de *aspect, vendrá* dado por la relación entre el alto y ancho del plano de corte, por lo tanto aspect toma el valor de alto plano dividido entre largo plano.
Los valores de *pcerca* y *plejos* corresponden a los plano de corte cercano y lejano, de forma similar que en los casos anteriores.

La siguiente figura muestra un ejemplo con los diferentes parámetros para una proyección de una imagen ortográfica utilizando gluPerpective.

Para mayor información acerca de las proyecciones en OpenGL referirse al Anexo A.

El siguiente ejemplo muestra el dibujo de tres objetos donde se ha utilizado proyección paralela, en este ejemplo se ha definido el siguiente volumen de vista:

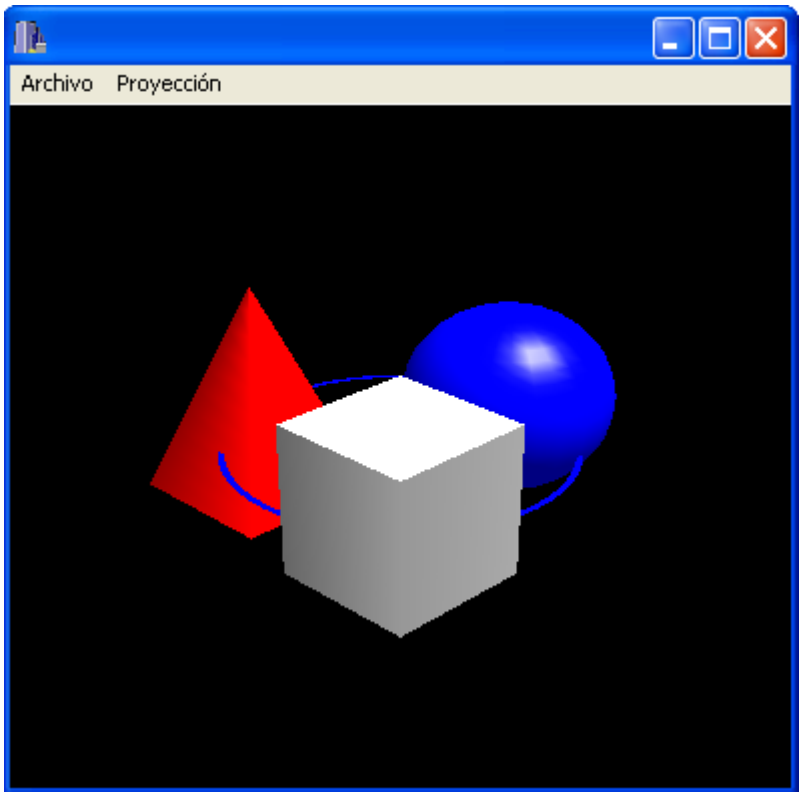
```
gluPerspective(30,1,1,-10);          // Proyección Paralela
```



El listado muestra la función que define el volumen de vista de nuestra aplicación:

```
void __fastcall TFormMain::FormResize(TObject *Sender)
{

  GLfloat nRange = 50.0;
  w = ClientWidth;                          // Obtiene ancho de la  Ventana
  h = ClientHeight;                         // Obtiene alto de la  Ventana
  if(h == 0)   h = 1;                       // Previene la división por cero
  glViewport(0, 0, w, h);           // Define la ventana de visualización
  glMatrixMode(GL_PROJECTION);      // Selecciona matriz de Proyección
  glLoadIdentity();
  gluPerspective(30,1,1,-10);               // Proyección Paralela
  glMatrixMode(GL_MODELVIEW); // Selecciona matriz de Proyección
  glLoadIdentity();
  gluLookAt(0,0,200,0,0,0,0,1,0);           // Ubicación del Observador
 }
```

La siguiente figura muestra los mismos objetos dibujados ahora en una proyección ortográfica,



Aquí hemos definido el siguiente volumen de vista

```
GLfloat nRange = 50.0;
 w = ClientWidth;
 h = ClientHeight;

 glOrtho(-nRange, nRange, -nRange*h/w, nRange*h/w,-nRange, nRange);
```

El listado muestra la función que define el volumen de vista de nuestra aplicación:

```
void __fastcall TFormMain::FormResize(TObject *Sender)
{

    GLfloat nRange = 50.0;
    w = ClientWidth;
    h = ClientHeight;
    if(h == 0) h = 1;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    If (w <= h)                          // Correción de apariencia
            glOrtho (-nRange, nRange, -nRange*h/w, nRange*h/w, -nRange,
            nRange);                     // Proyección Ortografica
    else
            glOrtho (-nRange*w/h, nRange*w/h, -nRange, nRange, -nRange,
            nRange);
    glMatrixMode(GL_MODELVIEW); // Selecciona matriz de modelo
    glLoadIdentity();
 }
```

El listado completa de la aplicación anterior es el siguiente:

```
__fastcall TFormMain::TFormMain(TComponent* Owner)
    : TForm(Owner)
{
    Application->OnIdle = IdleLoop;
    size = 50.0f;
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::IdleLoop(TObject*, bool& done)
{
    done = false;
    RenderGLScene();
    SwapBuffers(hdc);
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::FormCreate(TObject *Sender)
{
    hdc = GetDC(Handle);
    SetPixelFormatDescriptor();
    hrc = wglCreateContext(hdc);
    if(hrc == NULL)
      ShowMessage(":-)~ hrc == NULL");
    if(wglMakeCurrent(hdc, hrc) == false)
      ShowMessage("Could not MakeCurrent");
    w = ClientWidth;
    h = ClientHeight;
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    CreateInpriseCorporateLogo();
    SetupLighting();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::SetPixelFormatDescriptor()
{
    PIXELFORMATDESCRIPTOR pfd = {
      sizeof(PIXELFORMATDESCRIPTOR),
        1,
        PFD_DRAW_TO_WINDOW          |          PFD_SUPPORT_OPENGL          |
      PFD_DOUBLEBUFFER,
        PFD_TYPE_RGBA,
        24,
        0,0,0,0,0,0,
        0,0,
        0,0,0,0,0,
        32,
        0,
        0,
        PFD_MAIN_PLANE,
        0,
        0,0,
```

```
    };
    PixelFormat = ChoosePixelFormat(hdc, &pfd);
    SetPixelFormat(hdc, PixelFormat, &pfd);
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::FormResize(TObject *Sender)
{

    GLfloat nRange = 50.0;
    w = ClientWidth;
    h = ClientHeight;
    if(h == 0)
      h = 1;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (p==0){
    if (w <= h)
            glOrtho (-nRange, nRange, -nRange*h/w, nRange*h/w, -nRange,
      nRange);
    else
      glOrtho (-nRange*w/h, nRange*w/h, -nRange, nRange, -nRange,
      nRange);
            }
     if(p==1){
     gluPerspective(30,1,1,-10);
     }
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
     if(p==1){
        gluLookAt(0,0,200,0,0,0,0,1,0);
     }
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::RenderGLScene()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    DrawObjects();
    glFlush();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::DrawObjects()
{
    glBindTexture(GL_TEXTURE_2D, texture1);
    glPushMatrix();
    glColor3f(0.8f, 0.8f, 0.8f);
    glTranslatef(0.0f, 15.0f, 20.0f);
    glRotatef(90, 1.0, 0.0, 0.0);
    glCallList(startoflist);
    glPopMatrix();
    glPushMatrix();
    glDisable(GL_CULL_FACE);
    glColor3f(0.8f, 0.8f, 0.8f);
    glTranslatef(0.0f, 15.0f, 20.0f);
    glRotatef(-90, 1.0, 0.0, 0.0);
    glCallList(startoflist + 1);
    glEnable(GL_CULL_FACE);
```

```
    glPopMatrix();
    glPushMatrix();
    glColor3f(0.8f, 0.8f, 0.8f);
    glTranslatef(0.0f, -10.0f, 20.0f);
    glRotatef(90, 1.0, 0.0, 0.0);
    glCallList(startoflist + 2);
    glPopMatrix();
    glBindTexture(GL_TEXTURE_2D, texture2);
    glPushMatrix();
    glColor3f(1.0f, 0.0f, 0.0f);
    glTranslatef(-20.0f, -10.0f, -5.0);
    glRotatef(-90, 1.0, 0.0, 0.0);
    glCallList(startoflist + 3);
    glPopMatrix();
    glPushMatrix();
    glColor3f(1.0f, 0.0f, 0.0f);
    glTranslatef(-20.0f, -10.0f, -5.0);
    glRotatef(90, 1.0, 0.0, 0.0);
    glCallList(startoflist + 4);
    glPopMatrix();
    glPushMatrix();
    glBindTexture(GL_TEXTURE_2D, texture3);
    glColor3f(0.0f, 0.0f, 1.0f);
    glTranslatef(15.0, 5.0f, -7.5f);
    glRotatef(90, 1.0, 0.0, 0.0);
    glRotatef(90, 0.0, 0.0, 1.0);
    glCallList(startoflist + 5);
    glPopMatrix();
    glPushMatrix();
    glDisable(GL_CULL_FACE);
    glRotatef(-90, 1.0, 0.0, 0.0);
    glCallList(startoflist + 6);
    glEnable(GL_CULL_FACE);
    glPopMatrix();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::FormPaint(TObject *Sender)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glFlush();
    DrawObjects();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::FormDestroy(TObject *Sender)
{
    gluDeleteQuadric(GoldCube);
    gluDeleteQuadric(GoldCubeTop);
    gluDeleteQuadric(GoldCubeBottom);
    gluDeleteQuadric(RedPyramid);
    gluDeleteQuadric(RedPyramidBottom);
    gluDeleteQuadric(BlueSphere);
    gluDeleteQuadric(Ring);
    delete bitmap;
    wglMakeCurrent(NULL, NULL);
    wglDeleteContext(hrc);
}
//---------------------------------------------------------------------------
```

```
void __fastcall TFormMain::CreateInpriseCorporateLogo()
{
   startoflist = glGenLists(4);
   DrawGoldCube();
   DrawRedPyramid();
   DrawBlueSphere();
   DrawRing();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::DrawGoldCube()
{
   GoldCube = gluNewQuadric();
   GoldCubeTop = gluNewQuadric();
   GoldCubeBottom = gluNewQuadric();
   glNewList(startoflist, GL_COMPILE);
      gluCylinder(GoldCube, 15, 15, 25, 4, 10);
   glEndList();

   glNewList(startoflist + 1, GL_COMPILE);
      gluDisk(GoldCubeTop, 0, 15, 4, 1);
   glEndList();

   glNewList(startoflist + 2, GL_COMPILE);
      gluDisk(GoldCubeBottom, 0, 15, 4, 1);
   glEndList();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::DrawRedPyramid()
{
   RedPyramid = gluNewQuadric();
   RedPyramidBottom = gluNewQuadric();

   glNewList(startoflist + 3, GL_COMPILE);
      gluCylinder(RedPyramid, 16, 0, 35, 4, 10);
   glEndList();

   glNewList(startoflist + 4, GL_COMPILE);
      gluCylinder(RedPyramidBottom, 16, 0, 1, 4, 10);
   glEndList();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::DrawBlueSphere()
{
   BlueSphere = gluNewQuadric();

   glNewList(startoflist + 5, GL_COMPILE);
      gluSphere(BlueSphere, 15, 25, 25);
   glEndList();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::DrawRing()
{
   Ring = gluNewQuadric();
   glColor3f(1.0f, 1.0f, 1.0f);
   gluQuadricDrawStyle(Ring, GLU_FILL);
   gluQuadricNormals(Ring, GLU_SMOOTH);
   glNewList(startoflist + 6, GL_COMPILE);
```

```
        gluDisk(Ring, 24, 25, 50, 50);
    glEndList();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
    if(Key == VK_UP)
        glRotatef(-5, 1.0, 0.0, 0.0);
    if(Key == VK_DOWN)
        glRotatef(5, 1.0, 0.0, 0.0);
    if(Key == VK_LEFT)
        glRotatef(-5, 0.0, 1.0, 0.0);
    if(Key == VK_RIGHT)
        glRotatef(5, 0.0, 1.0, 0.0);
}
//---------------------------------------------------------------------------


void __fastcall TFormMain::Salir1Click(TObject *Sender)
{
  exit(0);
}
//---------------------------------------------------------------------------
 void __fastcall TFormMain::SetupLighting()
{
    GLfloat MaterialAmbient[] = {0.5, 0.5, 0.5, 1.0};
    GLfloat MaterialDiffuse[] = {1.0, 1.0, 1.0, 1.0};
    GLfloat MaterialSpecular[] = {1.0, 1.0, 1.0, 1.0};
    GLfloat MaterialShininess[] = {50.0};
    GLfloat AmbientLightPosition[] = {0.5, 1.0, 1.0, 0.0};
    GLfloat LightAmbient[] = {0.5, 0.5, 0.5, 1.0};

    glMaterialfv(GL_FRONT, GL_AMBIENT, MaterialAmbient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, MaterialDiffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, MaterialSpecular);
    glMaterialfv(GL_FRONT, GL_SHININESS, MaterialShininess);
    glLightfv(GL_LIGHT0, GL_POSITION, AmbientLightPosition);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, LightAmbient);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    glShadeModel(GL_SMOOTH);
}

void __fastcall TFormMain::O1Click(TObject *Sender)
{

p=0;
FormMain->Resize();
}
//---------------------------------------------------------------------------

void __fastcall TFormMain::P2Click(TObject *Sender)
{
```

```
p=1;
FormMain->Resize();
}
//---------------------------------------------------------------------------
```