Capítulo 6. Desarrollo del sistema

6.1 Análisis y diseño del sistema

Desde el punto de vista de la ingeniería del software, el navegador fue desarrollado utilizando prototipos bajo el modelo de espiral propuesto por Boehm, el cuál es un proceso evolutivo del software que utiliza la naturaleza interactiva de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial (análisis, diseño, código y pruebas) [PRES97], mediante la realización de una planeación, análisis, construcción y evaluación para cada prototipo realizado. El objetivo de este modelo es comenzar con prototipos sencillos y rápidos de realizar, implementando principalmente la interfaz del usuario y realizando cada vez prototipos más complejos reutilizando código, para finalmente, realizar los últimos prototipos sólo con optimizaciones.

Utilizando este modelo, se desarrollaron 22 prototipos del navegador, donde para cada uno de ellos se realizaron las siguientes actividades:

+*Planeación de adelantos*: cada nuevo prototipo tenía el objetivo de implementar una nueva técnica, corregir algún defecto o realizar una optimización del navegador, por lo que fue necesario para cada prototipo planear los cambios a realizar y las repercusiones que éstos tendrían.

+*Análisis*: para cada prototipo se realizó un análisis sobre los cambios en la estructura de datos (si era necesario) y las implicaciones que tenían esos cambios en el tiempo de respuesta del navegador.

+*Implementación de los cambios*: cambios al código para lograr los objetivos planeados.

+*Evaluación del prototipo*: se realizaron pruebas con los prototipos para verificar que los resultados fueran los esperados.

Es importante destacar en este proceso dos hechos:

1.- El desarrollo del sistema fue un proceso paulatino en el que algunos prototipos no lograron los objetivos deseados por lo que tuvieron que desecharse.

2.- Antes de comenzar a desarrollar cualquier prototipo se hizo un análisis muy cuidadoso de las necesidades del sistema para la elección del lenguaje a utilizar para la implementación del mismo. Se definió un subconjunto de lenguajes conocidos, los cuales fueron C y Pascal para MSDos, Visual Basic 4.0 y 6.0, Java y Visual C++. Para la elección del lenguaje adecuado se analizaron los siguientes elementos, descartando lenguajes en el siguiente orden.

+*Definición de colores en millones de colores*: lenguajes como C para MSDos y Pascal permiten sólo utilizar 256 colores al mismo tiempo en pantalla, por lo que estos dos fueron descartados.

+*Rutinas predefinidas para el dibujo de polígonos y líneas*: el lenguaje debe tener rutinas básicas para el dibujo de figuras. En este sentido Java cumple la condición, pero

aunque Visual C++ y Visual Basic están muy limitados en este sentido, pueden utilizar funciones de Windows, por lo que fueron aceptados.

+*Facilidad en el manejo de los controles gráficos para la interfaz del navegador*: Visual Basic presenta una arquitectura de controles muy sencilla de utilizar, tanto en diseño de interfaz como en el control de eventos; en Java es sencillo programar eventos aunque el diseño es más elaborado. Visual C++ sin embargo presenta una compleja estructura en ambos sentidos, por lo que fue descartado.

+*Cantidad de memoria a utilizar*: debido a que los escenarios virtuales pueden llegar a ser muy grandes (miles de polígonos) y éstos deben conservarse en memoria, el lenguaje debe tener las posibilidades de permitir utilizar mucha memoria así como el uso de pila y vectores grandes. Visual Basic 3.0 está limitado en este sentido, por lo que fue desechado.

+*Rapidez para desarrollar software y conocimiento del lenguaje:* finalmente, la última decisión para la elección del lenguaje fue el conocimiento subjetivo que se tenía de estos lenguajes, eligiendo el lenguaje Visual Basic 6.0.

Además del análisis del lenguaje, se hizo también un análisis muy cuidadoso de la estructura de datos a utilizar para representar los polígonos y demás elementos del navegador, de manera que conforme nuevos prototipos eran desarrollados, sólo era necesario aumentar elementos a estas representaciones, pero no hacer costosas modificaciones a las estructuras utilizadas.

6.2 Implementación del navegador

Como se vio en los capítulo dos y tres existen una serie de modelos, técnicas y algoritmos para la descripción, manipulación y presentación de objetos tridimensionales. Para la implementación del navegador, se utilizó la estructura de árboles BSP para el modelado de sólidos y apuntadores a una tabla de vértices para representar los polígonos. La justificación de la elección de estos modelados así como la descripción y explicación del funcionamiento del navegador y la interpretación de escenas son presentados en esta sección.

6.2.1 Estructura de datos

La estructura de árboles BSP elegida para la representación de sólidos en el navegador es la estructura normal de un árbol binario donde cada nodo del árbol tiene un apuntador a un polígono y dos apuntadores a otros nodos. Esta estructura puede definirse como:

```
type
   BSP_tree = record
      root : polygon;
      backChild, frontChild : BSP_tree;
   end;
```

donde *backChild* apunta al subárbol de los polígonos que se encuentran detrás del polígono *root*, mientras que *frontChild* al subárbol de los polígonos que están enfrente de él.

La elección de la estructura para representar polígonos requirió un estudio cuidadoso de los modelos presentados en la sección 2.1:

+*Enumeración explícita de vértices*: tiene la desventaja de que vértices compartidos por varios polígonos son almacenados varias veces en memoria y los cálculos para estos vértices son repetidos. Su ventaja es que sólo es necesario almacenar los vértices originales de cada polígono y al momento de pintar uno calcular sus nuevos valores almacenándolos temporalmente. Además, las rutinas para el pintado de polígonos utilizados son compatibles con esta representación de polígonos.

+*Apuntadores a una tabla de vértices*: al no almacenarse vértices repetidos se utiliza menos memoria y no hay repetición de cálculos, sin embargo, antes de pintar cualquier polígono todos los vértices deben ser transformados y almacenados. Además, antes de pintar un polígono es necesario generar una estructura temporal listando sus vértices (modelo anterior) para ser compatible con las rutinas de desplegado de polígonos.

+*Apuntadores a una tabla de aristas*: presenta las mismas ventajas y desventajas que el modelo anterior, pero hay un aumento en la memoria utilizada para guardar las aristas.

Debido a que tener apuntadores a aristas necesita más memoria y su beneficio sólo se presenta cuando se despliegan las aristas de los objetos(modelos de alambre) esta estructura fue descartada. Para elegir entre los primeros dos modelos presentados, que tienen la desventaja de repetición de cálculos y la generación de una estructura temporal respectivamente, se realizaron pruebas con varias escenas utilizando ambos modelos (ver

sección 7.1) obteniendo mejores resultados con el modelo de apuntadores a una tabla de vértices.

La estructura utilizada para representar un polígono fue la siguiente:

```
type
    polygon = record
       points:integer;      {numero de vertices del polígono}
       vértices[]:pointer;  {apuntadores a la tabla de vértices}
       virtual[]:bolean;    {indicador de aristas virtuales}
       normal:pointer;      {apuntador a la normal del polígono}
       centro:pointer;      {apuntador al centro del polígono}
       color:pointer;       {apuntador al color del polígono}
end;
```

Los elementos básicos para representar un polígono son el número de vértices que tiene, la lista de apuntadores a sus vértices y la lista de indicadores de aristas virtuales. La normal de cada polígono y su centro son calculados una sola vez y la estructura guarda un apuntador a éstos para ahorrar tiempo de cálculos. Aunque el centro de los polígonos no es un vértice puede ser insertado en la tabla de vértices. También se almacena un apuntador al color del polígono.

Esta estructura de datos es sumamente flexible ya que permite agregar elementos a la representación del polígono sin modificar esta estructura e impedir la repetición de vértices, normales y propiedades de color de los polígonos. Las variables de tipo *pointer* pueden representarse de dos maneras:

+*Como un apuntador a un registro*: bajo este esquema las tablas de vértices, normales y propiedades de color de los polígonos son implementadas como listas ligadas.

Cada registro puede ser tan complejo como se requiera y añadirlo o quitarle campos no afecta la estructura del polígono. Por ejemplo, un registro de la tabla de vértices puede contener los valores del vértice original (*x*, *y*, *z*), del vértice transformado (*x*', *y*', *z*') y del punto en pantalla que le corresponde (*x*, *y*). El registro de las propiedades de color puede contener los valores *r, g, b* del color del objeto, su transparencia, el índice de reflexión especular, etc.

+*Como una posición en un arreglo*: si las tablas son almacenadas como arreglos, es posible guardar la posición en el arreglo que le corresponde al elemento. Para poder añadir o quitar elementos por ejemplo, a las propiedades de color, es necesario representar cada campo de la estructura como un arreglo, de manera que la posición *i* en todos los arreglos se refiera al mismo elemento. Así, la tabla de vértices puede representarse con los arreglos verticesx[], verticesy[], verticesz[], etc, donde las coordenadas del vértice *i* son (verticesx[i], verticesy[i], verticesz[i]).

En este trabajo las tablas de vértices, normales y propiedades de color fueron representadas con arreglos con los siguientes campos:

+*Tabla de vértices*.- se almacenan las coordenadas *x, y, z* del punto original, las coordenadas transformadas (*x*',*y*',*z*'), su posición en pantalla (*x*, *y*) y la posición de su normal en la tabla de normales.

+*Tabla de normales*.- se guardan los valores originales de las normales (*nx*, *ny*, *nz*) y los valores de las normales transformadas (*nx*', *ny*', *nz*').

+*Tabla de propiedades de color*.- los valores *r, g, b* del color del polígonos son almacenados.

6.2.2 Interpretación de escenarios

Como ya se mencionó el navegador tiene la capacidad de interpretar escenas en formato NFF y VRML 1.0. Este proceso puede verse como un traductor, donde el lenguaje fuente es VRML o NFF y el lenguaje destino es la estructura explicada en la sección anterior. La manera de funcionar del traductor es dividiendo el texto en *tokens* (unidades mínimas de información), analizando la secuencia de los mismos y generando el lenguaje destino. Este proceso es dividido en cuatro partes:

+*Scanner*.- es el encargado de la lectura del archivo fuente y la obtención de *tokens*. En el caso del formato NFF los *tokens* son palabras o números, mientras que en el lenguaje VRML también lo son caracteres especiales como llaves y paréntesis.

+*Parser*.- recibe el *token* y analiza de qué tipo es, por ejemplo: número entero o decimal, palabra reservada, caracter especial, etc, asignándole un *id* que será reconocido por el analizador sintáctico.

+*Analizador sintáctico*.- decide si el tipo de *token* actual (*id*) es válido en base a la serie de *tokens* anteriores utilizando la gramática del lenguaje fuente.

+*Generador*.- se encarga de generar el lenguaje destino.

Este proceso puede ser explicado mediante el siguiente ejemplo: si el archivo fuente (en formato VRML) tiene el texto “Cube {}”, el scanner obtiene el token “Cube”. El *parser* lo analiza y le asigna un id, por ejemplo un entero. El analizador sintáctico acepta el tipo de *token*, ya que pertenece a un símbolo terminal de la gramática. Luego es leído el siguiente *token* (“{“) por el scanner, el *parser* lo identifica y le asigna un *id* y el analizador sintáctico

lo acepta. En este punto, si el *token* hubiera sido de otro tipo el analizador sintáctico lo hubiera rechazado y mandado un mensaje de error. El proceso continúa y cuando el analizador sintáctico recibe el *id* que le corresponde al caracter especial "}" manda llamar al generador para que genere un cubo según la estructura del navegador.

6.2.2.1 Generación de primitivas curvas tridimensionales

Durante la interpretación de un archivo externo el generador debe producir sólo polígonos, sin embargo tanto el formato de escenas NFF como VRML definen primitivas tridimensionales curvas como esferas, cilindros y conos. Estas primitivas son parametrizables, indicando su posición y radio, para el caso de la esfera y el centro superior e inferior de los cilindros junto con sus respectivos radios. Si el radio superior es 0, la primitiva es un cono.

Aunque este tipo de representaciones para primitivas es compacto y útil para usarse en archivos externos y debido a que éstas constan en su mayoría de superficies curvas, el generador necesita transformar esta representación a un modelo de mallas de polígonos (ver sección 2.1) mediante un conjunto de polígonos que representen la primitiva con cierta aproximación. Al proceso de transformar una superficie o cuerpo curvo a una aproximación de mallas de polígonos se le conoce como poligonalización. La aproximación es medida de acuerdo al número de particiones que se hacen sobre la superficie curva y también es conocido como nivel de poligonalización; mientras mayor sea el número de particiones y polígonos que representen la superficie curva mejor será la aproximación (Fig. 6.1)

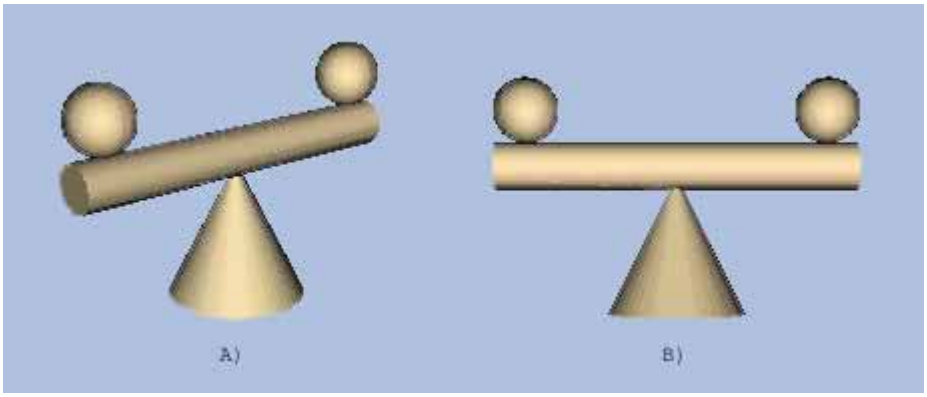

Fig. 6.1.-Primitivas generadas con un nivel de partición de 50. Las esferas están formadas por 4800 polígonos y los cilindros y conos por 52 y 51 polígonos respectivamente. Aunque la técnica de sombreado es Lambert, los polígonos son tan pequeños que no se notan y producen el efecto de ser superficies curvas.

Para generar las primitivas, se realizan cálculos asumiendo que éstas tienen su centro en el origen de coordenadas y una cierta orientación, por ejemplo, que el eje de conos y cilindros es el eje de coordenadas *y*. Una vez obtenido los puntos de la primitiva, éstos son trasladados y rotados para quedar en la posición y orientación deseados.

6.2.2.1.1 Generación de esferas

Para poligonalizar una esfera de radio *r* y centro (*x*, *y*, *z*) es necesario definir el número de particiones para dividir la esfera. Primero se genera la representación de una esfera de radio *r* con centro en (0,0,0) y luego todos los puntos son trasladados (*x*, *y*, *z*) unidades. Existen varias maneras de poligonalizar una esfera, a continuación se presenta el método empleado en la implementación del navegador.

Sea *n* el número de particiones para la esfera, una esfera es dividida en *n*-1 partes horizontalmente y en *n* partes verticalmente. Para explicar el procedimiento para generar una esfera se utiliza la figura 6.2 como referencia.

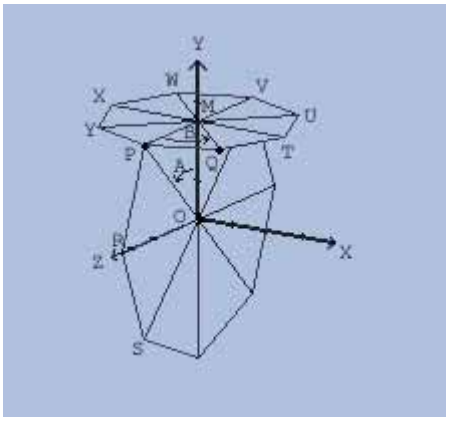

Fig. 6.2.-Elementos para la poligonalización de una esfera.
La esfera es dividida horizontalmente en los puntos *P*, *R* y *S*.
Para cada uno de estos puntos se realiza una división vertical (puntos *Q*, *T*, *U*, *V*, *W*, *X*, *Y*).

El primer paso es dividir horizontalmente la esfera en *n*-1 partes (puntos *P*, *R*, *S*), calculando para cada punto el ángulo *A*. Para cada una de estas particiones horizontales, se realiza una división vertical en *n* partes alrededor de toda la circunferencia (puntos *Q*, *T*, *U*, *V*, *W*, *X*, *Y*) calculando el ángulo *B*. Para cada uno de estos puntos se calcula su posición en el espacio.

Los puntos *P*, *R* y *S* son calculados de la siguiente manera: sus valores en *x* son 0, pues los puntos se encuentran sobre el plano *yz*. Ahora, sea *r* el radio de la esfera y *A* el ángulo entre el punto *P* y el punto *M*, en el eje *y*, y tomando en cuenta que el ángulo *PMO* es rectángulo, utilizando las propiedades de los triángulos rectángulos:

$$P_y = r\cos A$$
$$P_z = rsenA$$

De manera similar, los puntos de las particiones verticales son calculadas con las fórmulas:

$$Q_x = P_z senB$$
$$Q_y = P_y$$
$$Q_z = P_z \cos B$$

Finalmente, los puntos superior e inferior de la esfera tienen las coordenadas (0, *r*, 0) y (0, -*r*, 0) respectivamente. Para que la esfera se encuentre en la posición deseada (centro en *x*, *y*, *z*) los valores *x*, *y*, *z* deben ser sumados a los respectivos componentes de todos los puntos de la esfera; de esta manera, el puntos superior e inferior de la esfera se define por (*x*, *r*+*y* ,*z*) y (*x*, -*r*+*y*, *z*) respectivamente.

La figura 6.3 muestra las imágenes de esferas generadas utilizando diferentes niveles de particiones e indicando el número de polígono para cada esfera. El pseudocódigo para la generación de esferas es el siguiente (el método agregaPunto es utilizado para agregar un punto a la representación de la esfera):

```
procedure generaEsfera (r, x, y, z : real; n : integer)
var angyz, angxz: real;
var valorz : real;
var i, j : integer;
begin
    agregaPunto (x, r+y, z);
    for i = 1 To n - 1
    begin
        angyz = (i  * 180) / (n- 1);
        valorz = r * sen(angyz)
        for j = 1 to n
        begin
            angxz = (j * 360) / numpartesesf;
            agregaPunto ( valorz*sen(angxz)+x, r*cos(angyz)+y,
                          valorz*cos(angxz)+z );
        end;
    end;
    agregaPunto (x, -r+y, z);
end;
```

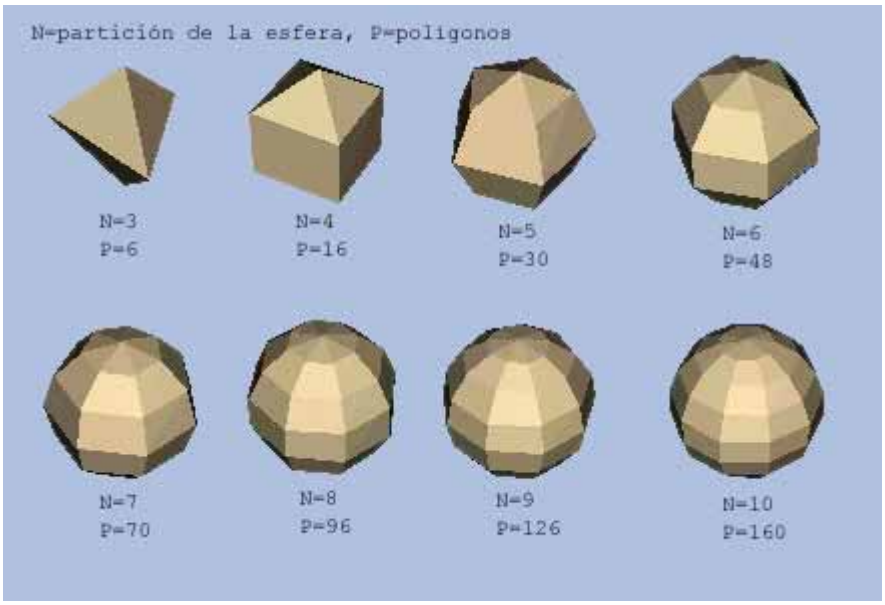

Fig. 6.3.-Esferas producidas con diferente nivel de partición (*N*). Para cada esfera generada se muestra el número de polígonos (*P*) que la componen. Se utilizó la iluminación de Lambert para pintarlas.

6.2.2.1.2 Generación de cilindros y conos

Los cilindros y conos son más sencillos de poligonalizar debido a que sólo existe un ángulo de rotación y no dos como en el caso de la esfera. Primero se genera un cilindro o cono del tamaño deseado con centro en el origen y teniendo como eje de rotación el eje *y*. De no ser esta la posición y orientación del cilindro o cono, los puntos deben ser trasladados y rotados.

Los cilindros y conos se representan por los puntos centrales y radios de sus caras superiores e inferiores. Para generar la primitiva con centro en origen y orientación sobre el eje *y* es necesario conocer la altura del cilindro o cono, la cuál debe ser calculada. Sean P_s y P_i los puntos centrales de la cara superior e inferior de la primitiva respectivamente, la altura es la distancia entre estos puntos, de la ecuación 4.14 se obtiene:

$$h = \sqrt{\left(Ps_x - Pi_x\right)^2 + \left(Ps_y - Pi_y\right)^2 + \left(Ps_z - Pi_z\right)^2}$$

El valor de *y* para la cara superior es *h*/2 y de la cara inferior –*h*/2.

Una vez obtenida la altura de la primitiva y tomando la figura 6.4 como referencia, la cara superior e inferior del cilindro (o la cara inferior del cono) es dividida en *n* partes, obteniendo los puntos *P* a *W*. El valor *y* es constante para estos puntos y los valores para *x*, *z* pueden ser calculados usando triángulos rectángulos. Sea *A* el ángulo entre el punto y el eje *z*, *r* el radio de una cara (ya sea superior o inferior), y *h* el valor *y* de esa cara, entonces:

$$P_x = r\cos A$$
$$P_y = h$$
$$P_z = rsenA$$

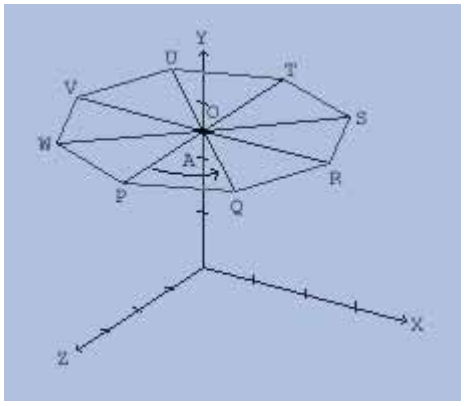

Fig. 6.4-Elementos para la poligonalización
de cilindros y conos. El cuerpo es dividido verticalmente
en n partes (puntos *P, Q, R, S, T, U, V, W*).

Una vez generada la primitiva es necesario trasladarla a la posición deseada y rotarla a la orientación especificada. Para resolver el problema de la posición es necesario

encontrar el centro de la primitiva (C_x, C_y, C_z), el cuál es obtenido con el promedio de los dos puntos dados P_s y P_i:

$$C_x = \frac{Ps_x + Pi_x}{2}$$
$$C_y = \frac{Ps_y + Pi_y}{2}$$
$$C_z = \frac{Ps_z + Pi_z}{2}$$

Una vez obtenidos estos valores, deben ser sumados a todos los puntos de la primitiva generada. Para lograr la orientación deseada, el eje del cilindro o cono debe corresponder al vector definido por P_s - P_i = (P_{sx}-P_{ix}, P_{sy}-P_{iy}, P_{sz}-P_{iz}). Los ángulos para rotar la primitiva de manera que quede con la orientación adecuada se pueden calcular con:

$$Angx = PI$$
$$Angy = -a\tan 2\left(C_z, C_x\right)$$
$$Angz = -a\tan 2\left(C_y, \sqrt{C_x^{\,2} + C_z^{\,2}}\right)$$

La función *atan2* es la función modificada de la función arco tangente (ver sección 4.1.8). La figura 6.5 muestra las imágenes de conos y cilindros generados utilizando diferentes niveles de partición. El siguiente pseudocódigo puede ser utilizado para la generación de un cilindro o cono:

```
procedure generaCilindro ( Rs, Psx, Psy, Psz, Ri, Pix, Piy, Piz : real,
                           n : integer)
var
   i : integer;
   ang, Cx, Cy, Cz : real;
   posysup, posyinf : real;
begin
    posysup = sqrt ( (Psx-Pix)^2 + (Psy-Piy)^2 + (Psz-Piz)^2) / 2
    posyinf   = -posysup
    for i = 1 To n
    begin
        ang = ( i * 360) / n
        if Rs<>0 then
             agregaPunto( Rs*cos(ang), posysup, Rs*sen(ang));
        agregaPunto( Ri*cos(ang), posyinf, Ri*sen(ang));
    end;
    Cx = (Psx-Pix) / 2;
    Cy = (Psy-Piy) / 2;
    Cz = (Psz-Piz) / 2;
    rotaCilindro ( PI, -atan2(Cz,Cx), -atan2(Cy, sqrt (Cx*Cx + Cz*Cz) );
    trasladaCilindro (Cx, Cy, Cz);
end;
```

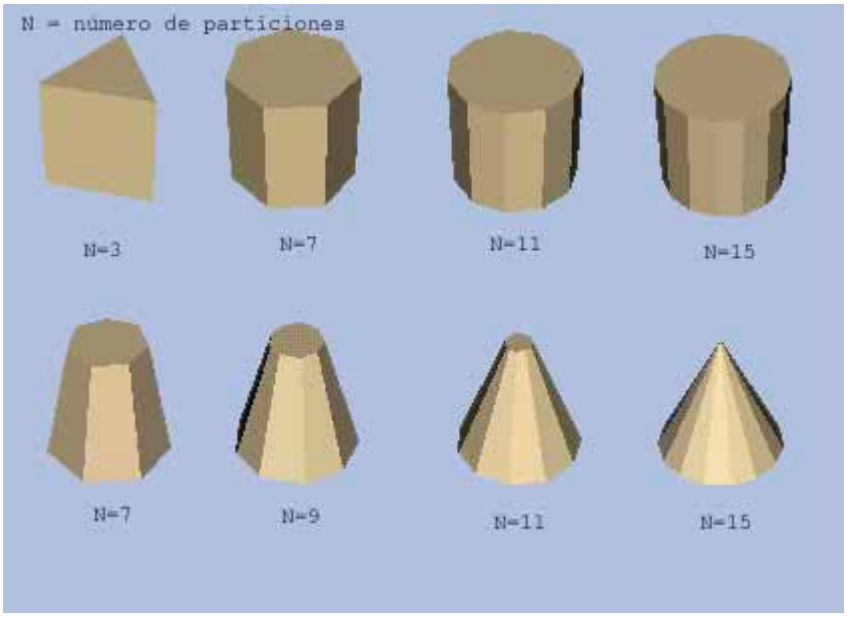

Fig. 6.5-Cilindros y conos generados con diferentes parametrizaciones y número de particiones. Se utilizó la iluminación de Lambert para pintarlas.

6.2.3 Funcionamiento del navegador

En esta sección se describe la manera en la que el navegador funciona, desde la interpretación de escenas hasta su navegación. El proceso completo del navegador puede dividirse en los siguientes pasos:

1.-*Interpretación de la escena*: un archivo en cualquier formato aceptado por el navegador es leído y traducido (ver sección 6.2.2), generándose una lista de polígonos.

2.-*Creación del árbol BSP*: en base a la lista de polígonos se genera la estructura del árbol BSP (sección 5.2).

3.- *Lectura de un comando de movimiento*: ya sea utilizando el teclado o el mouse se obtiene el movimiento deseado por el usuario (avanzar, rotar, etc.).

4.- *Transformación de vértices y normales*: en base al comando de movimiento los vértices y normales de los polígonos son transformados mediante la generación de la matriz de transformación (sección 4.2) que simula ese movimiento y la multiplicación de cada vértice y normal de los polígonos por esta matriz, almacenándose los puntos generados. Finalmente los vértices producidos son transformados a coordenadas de pantalla, utilizando el método de proyección ortogonal o perspectiva (sección 4.3) dependiendo de la opción elegida por el usuario.

5.-*Pintado del árbol BSP*: el último paso es el pintado del árbol BSP(sección 5.3) utilizando para cada polígono los vértices y normales transformados en el paso anterior. Cada vez que un polígono debe ser pintado se verifica la técnica de iluminación elegida por el usuario para pintar el polígono. Una vez terminado el desplegado del árbol BSP se

regresa al paso 3 o 1, dependiendo si se eligió hacer otro movimiento o abrir un nuevo escenario respectivamente.

6.2.3.1 Manejo de comandos de movimiento

El navegador implementado permite el uso del mouse o del teclado para informar al sistema el movimiento deseado mediante la siguiente tabla:

Movimiento	Rotación negativa / positiva en eje x	Rotación negativa / positiva en eje y	Rotación negativa / positiva en eje z
Teclado	u/I	h/k	i / y
Mouse	Botón izquierdo abajo y arriba	Botón izquierdo izquierda y derecha	Ambos botones derecha e izquierda
Movimiento	Traslación en y (subir y bajar)	Traslación en x (izquierda y derecha)	Traslación en z (acercarse y alejarse).
Teclado	e/d	s/f	w/r
Mouse	Botón derecho arriba y abajo	Botón derecho izquierda y derecha	Ambos botones arriba y abajo

El navegador almacena la posición *x*, *y*, *z* del centro actual del escenario y sus ángulos de rotación sobre los ejes *x*, *y* y *z* inicializándolos a cero. Cada comando de movimiento equivale a un incremento o decremento en estos valores, de manera que todos los movimientos son almacenados. Estos valores son utilizados para generar la matriz de transformación que genera la transformación deseada. El siguiente pseudocódigo muestra este proceso:

```
Comando:=obtenerComando();
if (comando= wTRASLADAR_ARRIBA) then
      ty:=ty+wINCREMENTO;
elseif (comando= wTRASLADAR_ABAJO) then
      ty:=ty-wINCREMENTO;
{…igual con la traslacion en x y z}
```

```
elseif (comando= wROTAR_X_POSITIVO) then
      angx:=angx+wINCREMENTO_ANG;
elseif (comando= wROTAR_X_NEGATIVO) then
      angx:=angx-wINCREMENTO_ANG;
{…igual con la rotación en y y z}
{generar la matriz de transformación}
crearMatrizIdentidad();
rotarMatrizEjeY(angy);
rotarMatrizEjeX(angx);
rotarMatrizEjeZ(angz);
trasladarMatriz(tx,ty,tz);
```

Los métodos para rotar la matriz de transformación y trasladarla (como *rotarMatrizEjeY* y *trasladarMatriz*) generan la matriz que produce el movimiento deseado (sección 4.2.2), y multiplican esta matriz por la original acumulando así las transformaciones en una sola matriz (ver sección 4.2.3).

6.2.3.2 Dibujo de polígonos

El navegador permite el desplegado de polígonos utilizando diferentes técnicas de iluminación y desplegado de aristas para poder ver la fragmentación producida en los polígonos durante la creación del árbol BSP. Los métodos implementados son:

+*Modelo de alambres*.- sólo las aristas originales de los polígonos son dibujadas sin tomar en cuanta el color o posición del polígono respecto a la luz, produciendo un desplegado muy rápido del escenario.

+*Modelo de alambres con fragmentación*.- se presentan tanto las aristas originales de los polígonos como las virtuales.

+*Aristas virtuales*.- se dibujan sólo las aristas virtuales para poder observar la fragmentación generada.

+*Aristas virtuales resaltadas*.- se dibujan todas las aristas de los polígonos, resaltando las virtuales (pintándolas más gruesas) y opacando las originales (usando líneas punteadas). Con este desplegado es más fácil observar la fragmentación comparándola con la escena original.

+*Líneas ocultas*: se dibuja la porción de las aristas originales que son visibles dado el punto de vista del observador.

+*Iluminación de Lambert*.- Los polígonos son dibujados mediante la técnica de sombreado de Lambert (sección 3.2.3.1) sin dibujar las aristas.

+*Iluminación de Lambert con aristas originales*.- además de dibujar los polígonos con la técnica de sombreado de Lambert se dibujan las aristas originales de los polígonos.

+*Iluminación de Lambert con aristas virtuales*.- igual que el método anterior pero también se dibujan las aristas virtuales. Este es el desplegado de polígonos que permite ver mejor la partición de polígonos.

+*Iluminación de Gouraud*.- se dibujan los polígonos utilizando la técnica de sombreado de Gouraud (ver sección 3.2.3.2).

+*Iluminación de Phong*.- se dibujan los polígonos utilizando la técnica de sombreado de Phong (ver sección 3.2.3.3).

Además de estos métodos para el desplegado de polígonos el navegador permite asignar al azar un color diferente a cada polígono, de manera que se facilite aún más la visualización de la fragmentación de polígonos. La figura 6.6 muestra una escena pintada con cada uno de los elementos mencionados.

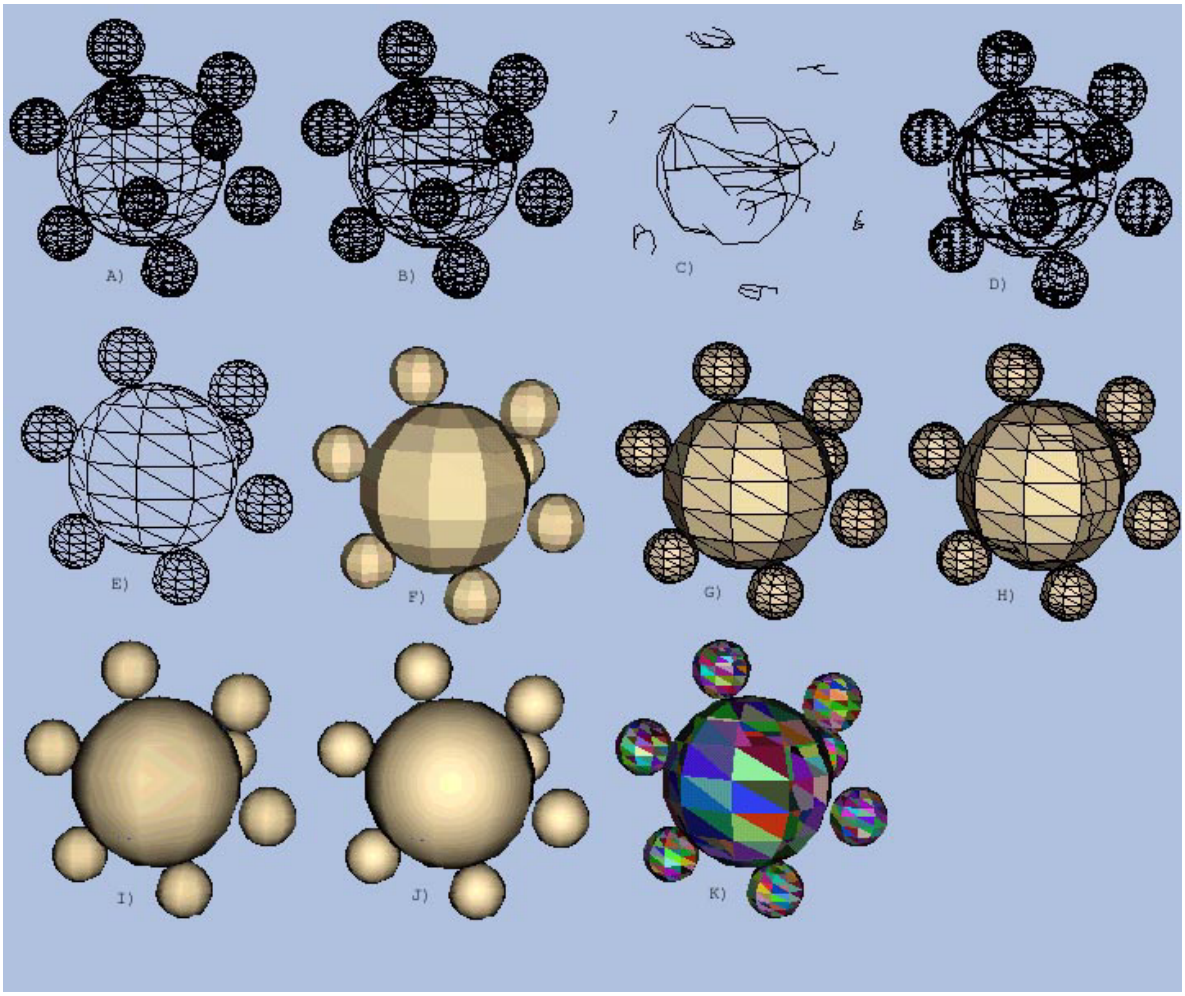

Fig. 6.6.- Escenas pintadas utilizando A)modelo de alambres, B)modelo de alambres con aristas virtuales, C)aristas virtuales, D)aristas virtuales resaltadas, E)líneas ocultas, F)iluminación de Lambert, G)iluminación de Lambert con aristas originales, H)iluminación de Lambert con aristas virtuales, I)iluminación de Gouraud, J)iluminación de Phong, K)iluminación de Lambert con cada polígono de diferente color.

Para pintar las aristas de los polígonos se toman en cuenta los indicadores de aristas virtuales del polígono, dependiendo del método a utilizar. Por ejemplo, para dibujar modelos de alambre, si el indicador de arista de un vértice es verdadero, la línea no es dibujada, sólo se mueve la pluma virtual a esa posición; si es falso, se dibuja la línea. El psesudocódigo que realiza esta función es el siguiente:

```
procedure dibujaPoligono(polígono: polygon) begin
   with polygon begin
      moveTo(vertices2Dx(.vertices(0)), vertices2Dy(.vertices(0)));
      for i=1 to .points-1 begin
         if (.virtual(i)) then
```

```
            moveTo(vertices2Dx(.vertices(i)), vertices2Dy(.vertices(i)));
         else
            lineTo(vertices2Dx(.vertices(i)), vertices2Dy(.vertices(i)));
      end;
   end;
end;
```

Para dibujar un polígono utilizando la técnica de Lambert es necesario calcular la intensidad de color de un punto del polígono (ver siguiente sección) y pintar el polígono entero de ese color. Aunque este punto puede ser cualquier vértice del polígono, se corre el riesgo de que dos polígonos adyacentes que comparten algún vértice, elijan el mismo vértice para calcular el vector *L* que va a la fuente de luz, de manera que ambos polígonos presentarán una intensidad de color muy parecida. Por otro lado, si eligen los vértices más alejados entre ellos, la diferencia de intensidad será muy notable (Fig. 6.7). Para evitar este problema es preferible elegir el centro del polígono y la normal del polígono para el cálculo de la intensidad de color de todo el polígono.

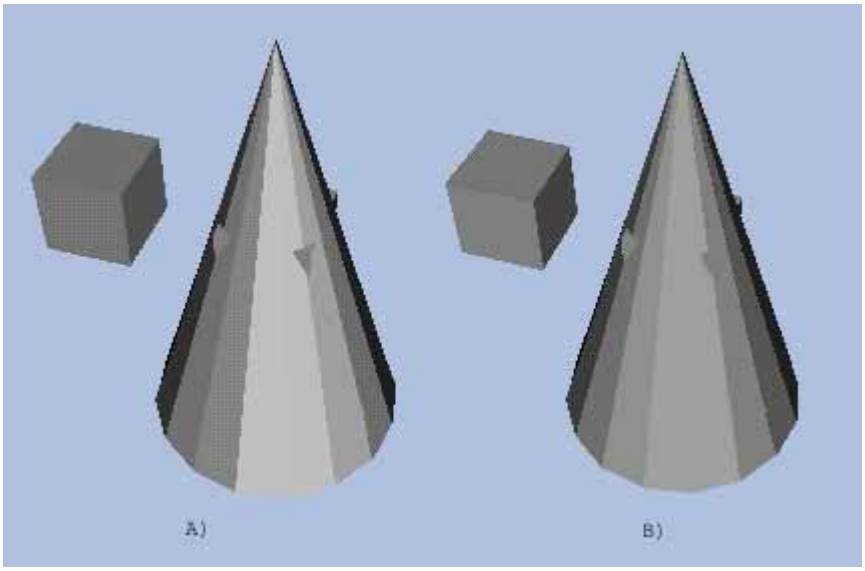

Fig. 6.7 .-Escenas iluminadas utilizando la técnica de Lambert y calculando la intensidad de color de cada polígono usando A)cualquier vértice del polígono y B)el centro del polígono. Nótese como la intensidad de color entre algunos polígonos de la figura A) varían notablemente.

Para dibujar polígonos utilizando la técnicas de sombreado de Gouraud y Phong (ver sección 3.2.3.2 y 3.2.3.3) es necesario implementar un procedimiento para rellenar polígonos (sección 6.2.3.2.2) de manera que para cada píxel se calcule su intensidad de color y se pinte en pantalla. Utilizando la técnica de Gouraud, la intensidad de color de los vértices es calculada utilizando las normales de los mismos y para los demás puntos del polígonos se interpolan trilinealmente las intensidades de color de los vértices. Con la técnica de Phong las normales de los vértices son interpoladas a través de todos los puntos del polígono. Utilizando estas normales, para cada punto del polígono su intensidad de color es calculada.

6.2.3.2.1 Intensidad de color de un punto

Dado un punto *P* la normal *N* para ese punto y los componentes RGB del color del polígono al que pertenece (Kd_r, Kd_g, Kd_b), el cálculo de la intensidad de color que le corresponde puede ser encontrado utilizando la ecuación 3.7 para cada uno de los componentes RGB (ver sección 3.2.2):

El término $cos\theta$ es el coseno del ángulo entre la normal *N* del polígono y el vector *L* que va del polígono hacia la luz (ver sección 3.2.3.1) y puede ser calculado como el producto punto de $N \cdot L$ (sección 4.1.2). El método para obtener el vector *L* varía dependiendo del tipo de cámara que se está utilizando:

+*Cámara ortogonal*.- todos los rayos de luz son paralelos entre sí y perpendiculares al plano de proyección. Debido a que el plano de proyección en el navegador es paralelo al

plano *XY,* el vector *L* siempre es (0,0,1), de manera que al realizar el producto punto de $N \cdot L$ se obtiene:

$$N_x \times 0 + N_y \times 0 + N_z \times 1 = N_z$$

y el cálculo del ángulo entre la normal del polígono y el vector de la luz puede reducirse a tomar el valor del componente *z* de la normal del polígono.

+*Cámara perspectiva*.- todos los rayos de luz surgen de un punto por lo que el vector *L* es diferente para la mayoría de los polígonos y debe ser calculado mediante la resta vectorial del punto donde se encuentra la fuente de luz P_L y el punto del polígono *P*. Así, el vector *L* es:

$$L = \left(P_{Lx} - P_x, P_{Ly} - P_y, P_{Lz} - P_z\right)$$

y el coseno del ángulo puede ser calculado con $N \cdot L$.

De manera similar, el término *cos Φ* equivale al ángulo entre el vector de reflexión *R* y el vector *V* que se dirige al observador. Ya que en la implementación del navegador la fuente de luz está en la misma posición que el observador, entonces

$$\cos\Phi = V \cdot R = L \cdot R = 2\left(N \cdot L\right)$$

El término *Kd* en cada fórmula (una para cada componente de color) equivale al valor del color del objeto; los demás términos I_a, I_p, *Ks* y *n* son valores del sistema.

Aunque utilizando la ecuación descrita se toman en cuenta la luz ambiental, reflexión difusa y especular y la atenuación de luz, el navegador permite calcular intensidades de color usando cualquier combinación de estos cuatro elementos. Esto se logra inicializando los componentes RGB del color final a cero y sumando cada uno de los elementos que deben tomarse en cuenta a estos valores. El siguiente pseudocódigo muestra la manera de hacerlo:

```
procedure calcularColor(puntox,puntoy,puntoz,anguloluz,r,g,b:double)
   colorR := 0;
   colorG := 0;
   colorB := 0;
   If luzambiental Then begin
      ColorR := colorR + r * luzambientalx;
      colorG := colorG + g * luzambientaly;
      colorB := colorB + b * luzambientalz;
   end;
   if atenuacionAtmosferica Then begin
      distancia = Sqr(allVerticesRotatedx(.centro) ^ 2 +
                  allVerticesRotatedy(.centro) ^ 2 +
                  (allVerticesRotatedz(.centro) - viewer(WZ)) ^ 2);
      atenuacion:= viewer[WZ] / distancia + 0.0001;
      if atenuacion > 1 Then
         atenuacion := 1;
   end;
   if luzdifusa Then begin
      colorR := colorR + r * anguloLuz * atenuación;
      colorG := colorG + g * anguloLuz * atenuación;
      colorB := colorB + b * anguloLuz * atenuación;
   end;
   if luzespecular Then begin
      anguloLuz := 2 * anguloLuz;
      colorR := colorR + atenuacion * 0.1 * (anguloLuz ^ 8);
      colorG := colorG + atenuacion * 0.1 * (anguloLuz ^ 8);
      colorB := colorB + atenuacion * 0.1 * (anguloLuz ^ 8);
   end;
end;
```

6.2.3.2.2 Relleno de polígonos

El relleno de polígonos general es complejo, por lo que el navegador está limitado a polígonos convexos. Un polígono es convexo sí y sólo sí para cualquier par de puntos en su interior el segmento de recta que los une se halla también dentro del polígono[BERG90], por lo tanto, un polígono convexo tiene a lo más 2 aristas en una línea de barrido horizontal (Fig. 6.8).

Un método muy sencillo para rellenar polígonos convexos consiste en almacenar en una tabla de líneas de barrido horizontales los valores máximos y mínimos de *x* en cada línea de barrido, procesando cada arista del polígono y comparando sus valores *x* con los almacenados en la tabla. Una vez llena la tabla el polígono es dibujado recorriendo la tabla y pintando los puntos desde la *x* mínima hasta la *x* máxima para cada valor de *y*.

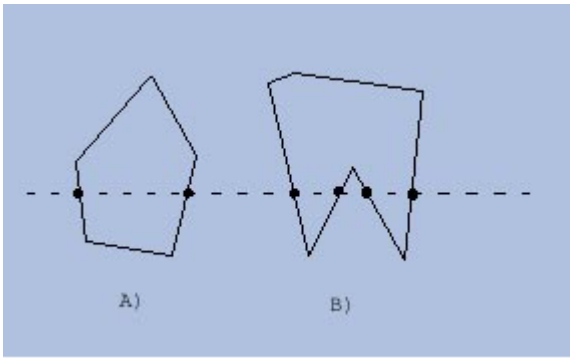

Fig. 6.8.- A)Polígono convexo, B)Polígono no convexo.

Para determinar los valores máximos y mínimos en la tabla es necesario analizar para cada arista del polígono el valor *x* correspondiente a cada valor *y*. Si la arista *P1P2* no es horizontal entonces el recíproco de su pendiente es:

$$\frac{1}{m} = \frac{P2_x - P1_x}{P2_y - P1_y}$$

Si $P_{1y} < P_{2y}$ se comienza con $y=P_{1y}$, donde el valor de x es P_{1x}, y se incrementa y en una unidad conforme se cambia a la siguiente línea de barrido. En la nueva línea de barrido $y= P_{1y}+1$ y $x=x'$, donde x' se calcula mediante:

$$m = \frac{P1_y + 1 - P1_y}{x' - P1_x} = \frac{1}{x' - P1_x}$$

y despejando x':

$$x' = P1_x + \frac{1}{m}$$

donde P_{1x} es substituido por x' para acumular los valores de x, pues si $1/m<=0.5$ la x nunca se incrementará. El pseudocódigo para calcular los valores de x de una arista y verificar si se introducen o no en la tabla de líneas de barrido se muestra a continuación:

```
procedure calcularlinea(x1,y1,x2,y2:double)
var
   m_inversa,x,y,ymin:double;
begin
   m_inversa := (x2-x1)/(y2-y1);
   {asegurar que y1<y2}
   if y1>y2 then begin
      swap(x1,x2);
      swap(y1,y2);
      swap(x1,x2);
   end;
   ymin:=y1;
   verificarPunto(x1,y1,ymin);
   verificarPunto(x2,y2,ymin);
   x:=x1;
   y:=y1+1;
   while y<y2 do begin
      x:=x+m_inversa;
      verificarPunto(round(x),y,ymin);
      y:=y+1;
   end;
end;
```

```
procedure verificarPunto(x,y,ymin:double) begin
   if x<tabla[y-ymin].xmin then
      tabla[y-ymin].xmin=x;
   if x>tabla[y].xmax then
      tabla[y-ymin].xmax=x;
end;
```

Una vez llena la tabla el polígono es dibujado línea por línea (o píxel por píxel) usando los valores máximos y mínimos de x en la tabla:

```
for y:=0 to ymax do begin
   with tabla[y] begin
      for x:=.xmin to .xmax do
         setPixel(x,y,color);
   end;
end;
```

Este procedimiento puede ser adaptado fácilmente para dibujar polígonos con las técnicas de sombreado de Gouraud o Phong, almacenando en la tabla de barrido además de los valores máximos y mínimos de *x*, los componentes de color o de las normales interpoladas respectivamente a lo largo de las aristas, y durante el relleno horizontal interpolar los valores correspondientes de *xmin* y *xmax* a lo largo de la línea horizontal.

6.2.3.2.3 Dibujo fuera de pantalla

El navegador dibuja el escenario completo cada vez que un comando de movimiento es introducido. La secuencia de estas imágenes generadas provocan el efecto de movimiento de una persona dentro del mundo virtual. Sin embargo, si los polígonos son dibujados directamente a la pantalla dos problemas surgen:

+Antes de comenzar a dibujar los polígonos de acuerdo al movimiento deseado es necesario borrar la imagen anterior, para impedir que zonas de ésta sean visibles en la nueva imagen. Este borrado y desplegado de imágenes constantemente genera un parpadeo en la animación que disminuye el efecto de movimiento.

+Si el escenario es muy complejo y está compuesto de muchos polígonos se verá como son dibujados paulatinamente los polígonos.

La solución a estos dos problemas relacionados con la animación es el dibujo fuera de pantalla, también conocido como *off-screen*. Esta técnica consiste en tener un buffer para la imagen en memoria. Cada vez que una imagen va a ser creada el buffer es inicializado al color de fondo, luego cada polígono es dibujado en memoria y cuando la imagen es terminada es copiada completa de la memoria a la pantalla. De esta forma, aunque la imagen tarde tiempo en ser generada, ésta será desplegada en pantalla hasta que esté completamente terminada; además, como no es necesario borrar la imagen anterior en pantalla, pues la nueva la cubrirá totalmente, el efecto del parpadeo en la animación no se verá.