TEMA 3: SISTEMAS DE VISUALIZACIÓN EN TIEMPO REAL. VISUALIZACIÓN POR HARDWARE

3.1 INTRODUCCIÓN

Se entiende por *sistema informático en tiempo real* aquel que tiene ciertas restricciones de comportamiento en el tiempo. Algunas restricciones bastante comunes son que el sistema responda a una señal externa en un tiempo lo suficientemente corto o que produzca una frecuencia de salida mínima[1]. En particular, un sistema de síntesis de gráficos 3D en tiempo real, será un sistema que pueda redibujar la escena en un tiempo suficientemente corto, lo que permitirá cierta sensación de continuidad visual y dotar de un grado de interactividad adecuado a nuestra aplicación.

Para conseguir esta disminución del coste de la representación gráfica se utilizan librerías de funciones que implementan técnicas especiales de visualización, y también existe la posibilidad, dadas las características de los procesos que efectúan, de utilizar hardware especial para acelerar los cálculos.

Las características que se pueden destacar en este tipo de sistemas son:

- Trabajan con primitivas poligonales. Los objetos 3D siempre serán representados como superficies para proceder a su visualización y, a su vez, las funciones de las librerías trabajaran internamente con objetos poligonales que descompondrán en triángulos.
- Para permitir mayor grado de realismo se incorpora el pegado de texturas de color, normalmente bidimensionales.
- Se intenta que el mayor número posible de cálculos se resuelva mediante *operaciones lineales*, reducibles a sumas y multiplicaciones, y fácilmente implementables por hardware.
- Se incluye algún tipo de método para eliminar superficies ocultas. Las dos alternativas más comunes son:
  - Utilizar BSP-Trees para ordenar los polígonos por distancias y aplicar el algoritmo del pintor comenzando la visualización por los más lejanos.
  - Utilizar una memoria especial llamada *Z-buffer* que para cada pixel guarda la distancia a la que está el punto correspondiente del objeto tridimensional cercano de los dibujados en esa dirección hasta el momento.
- En estos métodos se utiliza, además de la memoria de pantalla (frame buffer) convencional, memoria adicional para permitir ciertas operaciones superponer diferentes tipos de información sobre cada pixel. El número de bits de información que se puede almacenar para cada pixel (número de *bitplanes*) constituye una característica fundamental.

Veamos los componentes más comunes de esta memoria de pantalla.

a) Memoria de pantalla normal (memoria de color):

Guarda el color de cada punto. Para gráficos por ordenador se suelen tener 24 bits de color (trueColor: 8 bits por componente R, G, B). Otros sistemas utilizan 32 bits, 4 bytes (3 de R, G, B y uno para el factor de transparencia "componente o canal *alfa"*). Para hacer gráficos interactivos esta memoria de color se suele multiplicar por dos, constituyendo lo que se llama un sistema de *doble buffer*. El doble buffer se utiliza para evitar la sensación de parpadeo (*flickering*) que se produce al redibujar la escena mientras la estamos observando. Para ello el sistema gráfico dibuja en un segundo buffer (el buffer trasero o *backbuffer*) mientras el usuario percibe la imagen del buffer delantero (*frontbuffer*). Cuando se acaba de dibujar la nueva imagen se conmutan (*swap*) las dos memorias, y el usuario pasa a ver de repente en el frontbuffer la nueva imagen acabada, comenzándose a dibujar la siguiente imagen en el backbuffer. De esta manera vemos siempre imágenes terminadas, y nunca el resultado intermedio.

En ocasiones se colocan otros planos de color adicionales que se pueden dibujar de forma independiente, los planos de superposición. Existen dos tipos de planos de superposición: aquellos que se

[1] Ver Tema 6: Simulación en Tiempo Real.Concepto de Tiempo Real.

van a dibujar siempre por encima de los planos de color normales (memoria de *overlay*) y los que se van a dibujar siempre por debajo (memoria de *underlay*). El overlay se puede usar para dibujar partes de la escena próximas que no cambian frecuentemente (por ejemplo, la cabina del piloto en un simulador de vuelo), mientras que el underlay se utiliza para dibujar el fondo (por ejemplo, las montañas lejanas del mismo simulador de vuelo). La librería debe disponer de algún sistema para distinguir cuál de los planos de color debe ser el que se utilice para dibujar cada pixel particular. Esto puede realizarse definiendo una máscara en los pixels, o bien utilizando colores especiales en los planos de underlay y overlay.

b) Memoria de acumulación:

Se utiliza también para almacenar color, con el propósito de acumular imágenes sucesivas. Con estas memorias se pueden realizar efectos curiosos como el 'motion bluring' que consiste en un desenfoque producido por el movimiento del objeto, que se puede conseguir sumando con diferentes pesos varias imágenes del objeto mientras se mueve. La memoria de acumulación se puede también utilizar para eliminar los efectos de dentado (*aliasing*) en la imagen, suavizando los bordes de los objetos y los detalles de las texturas. Para ello se hace la media entre varias imágenes que se diferencian por ligeras variaciones del punto de vista.

c) Memoria de profundidad (Z-buffer):

Como hemos dicho antes, la existencia de esta memoria es una característica particular de los sistemas de visualización en tiempo real. En esta memoria se suelen utilizar 16 o 32 bits por pixel para guardar información relativa a la distancia o profundidad de cada pixel. Cuando se comienza a dibujar la escena toda la memoria del z-buffer se borra, inicializando su valor a la máxima distancia posible. A partir de entonces cada vez que se dibuja un triángulo se comprueba, para cada pixel que cubre su proyección, si la distancia al observador es menor que la almacenada en ese punto del z-buffer. Si lo es, entonces el pixel se cambia al color del objeto y su valor de profundidad se almacena como nuevo valor en el z-buffer.

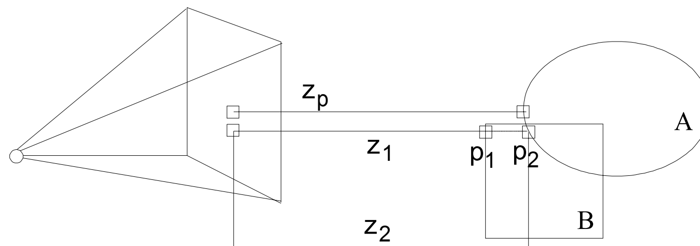
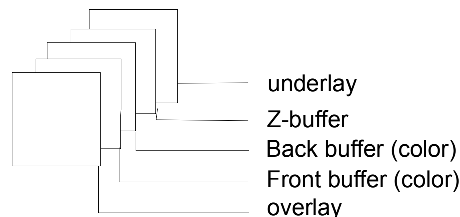


Figura 3.1.C.: Funcionamiento del z-buffer.

A se dibuja después que B, pero por ser $z_2 > z_1$, el color de p_1 no se borra para cambiarlo por p_2.

Al final tenemos una distribución de memoria del siguiente tipo:



3.2. LA PIPELINE GRÁFICA

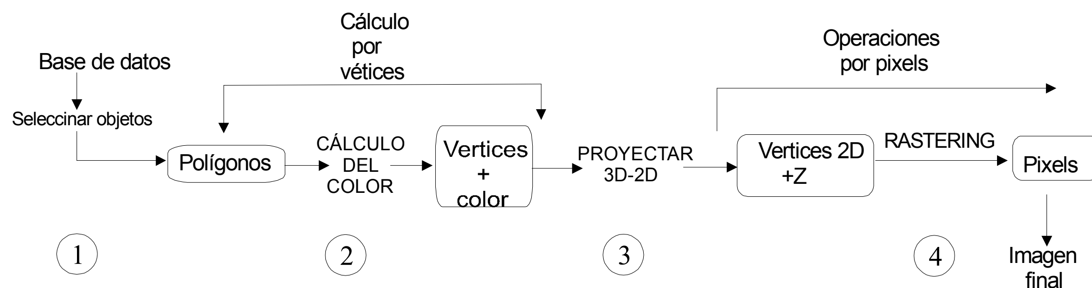
El proceso computacional que lleva desde la descripción de los objetos en la escena hasta la imagen final se produce, en los sistemas en tiempo real, mediante una serie de fases consecutivas y conectadas entre sí (la salida de datos de una fase constituye la entrada de la siguiente), que se denomina pipeline ('oleoducto') gráfica.

Debemos distinguir los datos propios de la escena, que van a circular por este sistema de procesamiento secuencial, del conjunto de variables, accesibles al programador, que van a determinar el procesamiento al que se ven sometidos los datos. Este conjunto de variables se denomina *contexto gráfico*. El contexto gráfico incluye variables cuyo valor puede ser cambiado por el programa a lo largo del proceso. Entre ellas hay opciones que controlan la activación o desactivación de ciertas funciones (por ejemplo, el uso del doble buffer o de la comparación del z-buffer) o controlan el modo en que algunas operaciones son realizadas (por ejemplo, variantes en el método de iluminación) y también incluyen datos que son utilizados por la pipeline, como las matrices de proyección y transformación, posición del observador, color actual, material actual, fuentes de luz, texturas, etc.

Recordemos que uno de los objetivos de los sistemas de visualización en tiempo real es descomponer el proceso de visualización en fases simples que tengan un coste mínimo y puedan ser implementadas por hardware. Según cuál sea el equipo disponible, parte de la pipeline será ejecutada por el hardware específico y parte tendrá que ejecutarse por la CPU del sistema. Esta idea permite que una misma aplicación pueda funcionar en equipos con diferente tipo de aceleración gráfica (o con ninguna). En cualquier caso, la parte ejecutada en la CPU comenzará siempre por el principio de la pipeline, y a partir de cierto punto los datos serán transferidos al hardware gráfico que continuará el proceso hasta el final.

La idea fundamental del procesado en tiempo real es que todos los objetos deben ser descompuestos en polígonos. Estos polígonos serán descompuestos a su vez en triángulos. Cada triángulo será proyectado sobre la ventana bidimensional y rellenado con los colores adecuados para reflejar los efectos de la iluminación, texturas, etc. Una vez se han generado los triángulos, en la pipeline existen dos partes claramente diferenciadas: una primera etapa operaciones realizadas sobre cada uno de los vértices, y después de que éstos se proyecten sobre la ventana, entonces comienza una segunda fase de cálculos realizados para cada pixel cubierto por los triángulos.

El siguiente sería un esquema del proceso en pipeline que comienza por la Base de Datos de la escena:



Las fases principales son:

1. - Seleccionar los objetos que deben dibujarse en cada fotograma
    - Convertir esos objetos a primitivas poligonales.
    - Producir a partir de ellas listas de vértices.

2. - Calcular el color para cada vértice según un método de sombreado local.

3. - Proyección 3D→2D de cada vértice, que además nos proporciona su profundidad
    - Recorte o *clipping* del triángulo si alguna parte de él cae fuera de la ventana.

4. - Rellenado de los triángulos (*rastering*). Para ello se realizan varias fases de procesamiento por pixel

- Comprobar si cada nuevo pixel es visible o no (comprobación de profundidad).
- Interpolación lineal del color para el nuevo pixel (método de Gouraud).
- Si existe una textura definida o transparencia, efectuar la modificación de color correspondiente.

Vamos a estudiar ahora con más detalle algunas de estas fases.

3.2.1. SELECCIÓN DE OBJETOS Y CONVERSIÓN A POLÍGONOS.

En gráficos interactivos toda representación de los objetos 3D que no sea poligonal se debe previamente transformar en una representación de superficie poligonal para poder hacer uso de la pipeline gráfica.

En esta fase de selección se intenta optimizar la calidad y cantidad de objetos que se van a utilizar. La selección de objetos se basa en dos técnicas principales:

- Selección del detalle. Un mismo objeto puede estar representado con más de un nivel de detalle, y por tanto, podemos escoger el que sea más adecuado (según su calidad y coste) en cada momento.
- Selección por visibilidad. Si queremos disminuir el coste de computación en la pipeline, podemos dejar de enviar los objetos que estén:
  - Ocultos por otros objetos.
  - Fuera del campo de visión.

Veremos más en detalle cómo estas técnicas pueden implementarse haciendo uso de estructuras de datos especiales[2].

La conversión a polígonos se puede efectuar por la propia aplicación, a criterio del programador, o utilizar funciones de la librería gráficas. Algunas librerías permiten definir objetos con representaciones no basadas en polígonos, pero incluyen funciones para efectuar la conversión. Por, ejemplo OpenGL incluye la definición de superficies paramétricas tipo NURBS utilizando funciones específicas (glNurbsSurface( )).

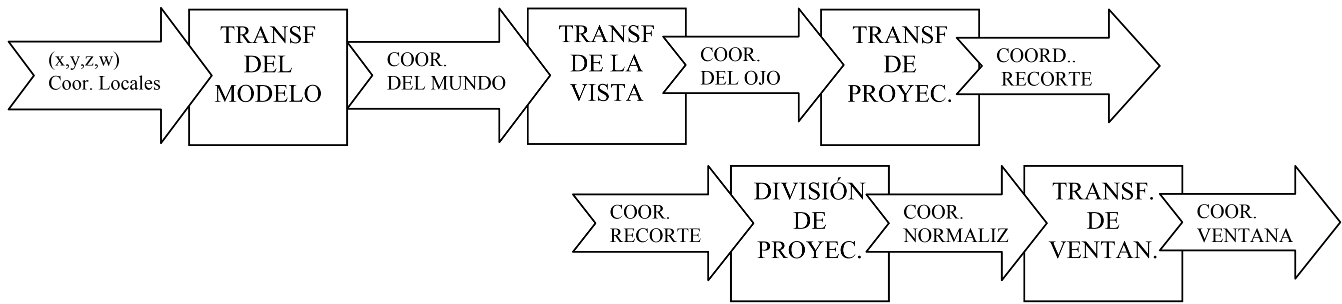
Al final de este proceso tendremos una lista de polígonos descritos por vértices. Cada vértice debe contener como mínimo la información acerca de su posición en el espacio, pero también datos que permitan calcular posteriormente su color. Si el programa no proporciona para cada vértice un color previamente calculado, deberá indicarse el valor del vector normal a la superficie en ese punto para que la pipeline calcule el color mediante un modelo de iluminación.

En el caso de que se desee pegar una textura sobre el objeto[3], deben proporcionarse los datos para que la pipeline calcule automáticamente las coordenadas de textura, o bien dar explícitamente los valores de estas coordenadas.

PIPELINA DE TRANSFORMACIONES SOBRE VERTICES

Una vez los objetos han sido seleccionados y se han descompuesto en vértices, estos sufre una proceso de transformación hasta convertirlos en pixels (nos centraremos en esas transformaciones y obviaremos la iluminación local aunque es un proceso que se realiza en coordenadas del ojo). Las transformaciones se pueden organizar como muestra las siguiente estructura de tubería:

[2] Ver Tema 6: Simulación en Tiempo Real. Estructuras Jerárquicas.
[3] Ver Tema 7: Técnicas Avanzadas de Modelado Geométrico. Texturas

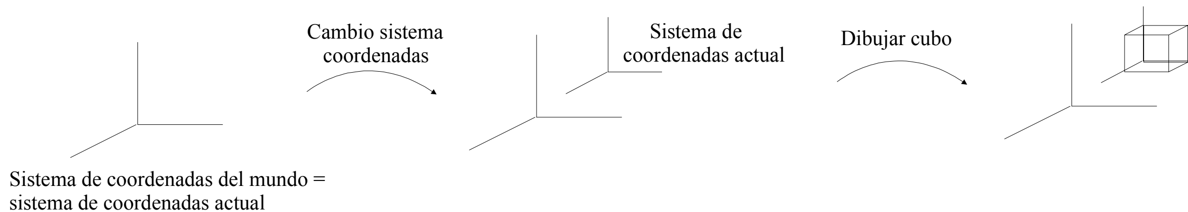


Esto pude verse como el proceso de hacer una foto: las tranformaciones del modelo consisten en posicionar el objeto en el punto de la escena que nos interesa, las transformaciones de la vista consisten en colocar la camara con la dirección adecuada. Las transformación de proyección son similares al proceso de enfocar la lente de la camara la división de proyección es el proceso de impregnación de la pelicula fotografica y las transformaciones de ventana se prodrian ver como el proceso de revelado donde seleccionamos el tamaño y proporciones del papel sobre el cual revelamos la foto.

3.2.2. TRANSFORMACIÓN DE OBJETOS O MODELADO.

Es muy común que dispongamos de una especificación del objeto en un cierto sistema de coordenadas (*sistema de coordenadas del objeto* o sistema *local*). Por ejemplo, hemos modelado una casa en la que las coordenadas de los vértices y la orientación de los vectores normales viene definida respecto a un sistema ortogonal situado en el centro de la casa. Sin embargo, los diferentes objetos situados en una escena comparten un sistema de referencia absoluto (*sistema de coordenadas del mundo*, de la escena, o sistema *global*). Por tanto necesitamos un mecanismo para transformar la posición, orientación y escala de los objetos de forma que podamos combinarlos en una escena común con otros, mover el objeto o efectuar diferentes copias del mismo en diferentes posiciones y tamaños.

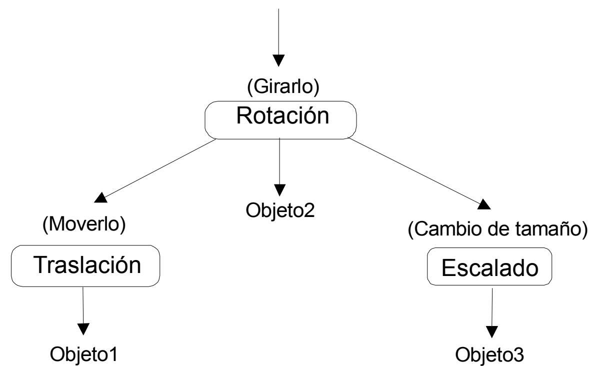
La forma en que estas transformaciones de modelado se efectúan en las librerías gráficas suele ser a través de un sistema de coordenadas actual que se almacena en el contexto gráfico. Este estado actual del sistema de coordenadas se representa por medio de una matriz que expresa la transformación que en cada instante lo liga con el sistema absoluto del mundo. Supongamos que tenemos una función que dibuja un cubo cuyos vértices han sido definidos respecto a con un sistema de coordenadas situado en su centro. ¿Cómo podemos dibujar el cubo en una posición cualquiera de la escena? Suponiendo que el sistema de coordenadas actual de la escena está inicialmente en el origen del sistema del mundo, tendríamos que trasladarlo al punto donde queremos situar el cubo, y entonces llamar a la función que dibuja el cubo.



Estas operaciones de transformación son siempre matriciales, pueden descomponerse en sumas y multiplicaciones, y por tanto caen dentro de la categoría de operaciones lineales que podemos incluir de forma eficiente en el procesado gráfico de los vértices que definen a los objetos.

Para visualizar cómo se combinan diferentes transformaciones de modelado podemos utilizar estructuras árboreas como ésta; en las que se especifica el orden de las operaciones y a qué objetos afecta (a todos aquellos bajo el nodo que indica la transformación).

Notar que debemos especificar el ordenen las operaciones.



Hay librerías que incluyen una estructura de datos en la que pueden definirse directamente estas estructuras jerárquicas de transformaciones. En otras librerías no existe una estructura de datos explícita, y las operaciones y su alcance se deben indicar mediante instrucciones de la librería en el código del programa. Por ejemplo esto sucede en OpenGl, donde el ejemplo anterior podría ser programado como se muestra en el anexo 9.

Las coordenadas utilizadas internamente por muchas librerías gráficas para representar la posición de los vértices tienen cuatro componentes. La cuarta componente se denomina coordenada homogénea, y tiene por defecto un valor unidad. Una vez se han efectuado todas las transformaciones matriciales, tanto las de proyección como las de modelado, la librería gráfica interpretará que las coordenadas 3D de cada punto son las tres primeras coordenadas, dividida cada una de ellas por el valor *w* de la coordenada homogénea.

El uso de estas coordenadas homogéneas tiene tres razones principales:

- Poder representar objetos infinitos, asignando a alguno de sus vértices un valor w = 0.
- Poder efectuar la proyección de perspectiva con una operación matricial (lo veremos luego).
- Poder representar la transformación de traslación (que en principio consiste en sumar un vector) como una multiplicación matricial.

Coord. Homogéneas $\rightarrow$ $\vec{X} = \left\{ \frac{x}{\omega}, \frac{y}{\omega}, \frac{z}{\omega} \right\}$.

Nota :

Para representar un punto en el infinito pondremos $\omega = 0$.

- **Traslación** $x' = T \cdot x$

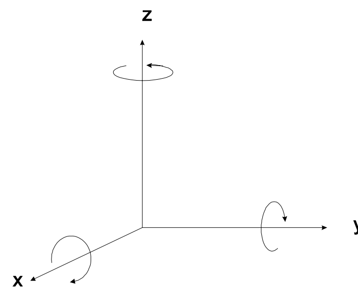
$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} x' \\ y' \\ z' \\ \omega' \end{pmatrix} = T(t_x, t_y, t_z) \cdot \begin{pmatrix} x \\ y \\ z \\ \omega \end{pmatrix} = \begin{pmatrix} x + t_x \cdot \omega \\ y + t_y \cdot \omega \\ z + t_z \cdot \omega \\ \omega \end{pmatrix}$$

Matriz de translación

$$\left.\begin{array}{l} x_{3D} = x/\omega \\ y_{3D} = y/\omega \\ z_{3D} = z/\omega \end{array}\right\} \Rightarrow \left\{\begin{array}{l} x'_{3D} = x'/\omega' = x'/\omega = \dfrac{x + t_x \cdot \omega}{\omega} = \dfrac{x}{\omega} + t_x = x_{3D} + t_x \\ y'_{3D} = y'/\omega' = y'/\omega = \dfrac{y + t_y \cdot \omega}{\omega} = \dfrac{y}{\omega} + t_y = y_{3D} + t_y \\ z'_{3D} = z'/\omega' = z'/\omega = \dfrac{z + t_z \cdot \omega}{\omega} = \dfrac{z}{\omega} + t_z = z_{3D} + t_z \end{array}\right.$$

- **Rotación**

Normalmente se ofrecen funciones para realizar una rotación alrededor de cualquiera de los tres ejes o alrededor de un eje arbitrario, pero teniendo siempre en cuenta cierto criterio de signo para interpretar el sentido de giro.



$$M_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Matriz de rotación alrededor del eje *x*

$$M_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Matriz de rotación alrededor del eje *y*

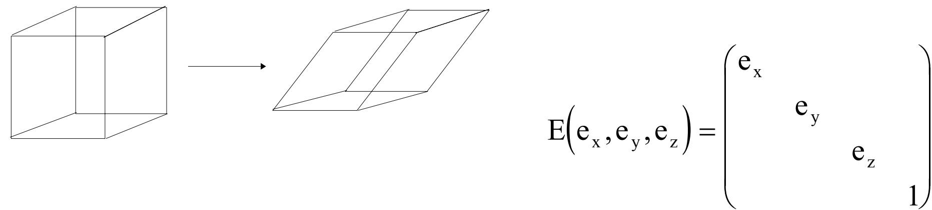
$$M_z(\theta) = \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Matriz de rotación alrededor del eje *z*

Para efectuar giros más complejos podemos combinar estas rotaciones básicas de forma secuencial.

Las rotaciones no afectan solamente a la posición del objeto, sino también a los vectores normales, de manera que la iluminación calculada posteriormente corresponda a la orientación real del objeto en la escena.

- **Escalado**

Mantiene la forma básica del objeto, sólo altera sus proporciones. Las anteriores transformaciones sólo varían la posición pero no sus proporciones. Para cambiar la escala del objeto en cada uno de los tres ejes, con sendos factores de escala (e_x , e_y , e_z) , utilizaríamos la matriz:



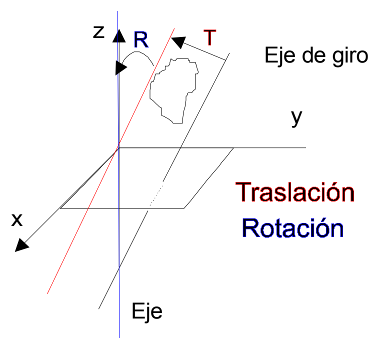
$$E(e_x, e_y, e_z) = \begin{pmatrix} e_x & & & \\ & e_y & & \\ & & e_z & \\ & & & 1 \end{pmatrix}$$

- **Composición de transformaciones**

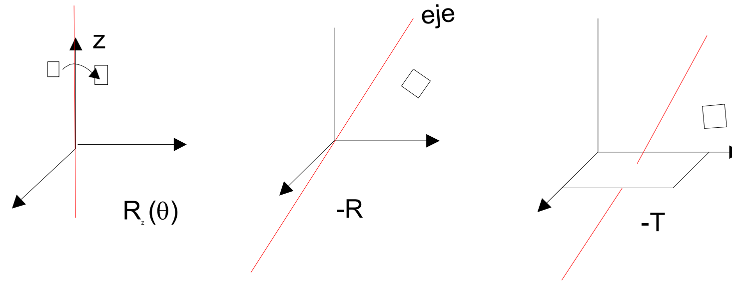
Podremos también concatenar varias transformaciones sobre un objeto, ya que el sistema de referencia local resultará afectado consecutivamente por todas las operaciones que vayamos haciendo dentro del alcance adecuado. Debemos tener en cuenta la forma en que estas transformaciones se acumulan. Por ejemplo, si efectuamos una rotación y luego una traslación, el resultado no es el mismo que si hacemos las operaciones al revés. Si giramos respecto al eje X, la orientación de los ejes Y, Z habrá cambiado, y eso afectará a todas las traslaciones ,rotaciones y escalados posteriores, que se efectuarán utilizando los nuevos ejes.

Un ejemplo sencillo de composición es el siguiente: deseamos hacer rotar un objeto, pero no respecto al origen de coordenadas actual, sino respecto a otro punto. Para conseguirlo tendremos que trasladar el origen de coordenadas al punto que deseamos como centro de la rotación, efectuar la rotación y luego devolver el origen de coordenadas a su posición inicial, para lo cual deberíamos haberlo apilado previamente.

Otro ejemplo, un poco más complejo, es rotar un objeto respecto a un eje arbitrario. Para ello efectuaríamos una traslación T que haga pasar el eje de giro por el origen de coordenadas, luego una rotación (rotaciones) R que hagan coincidir el eje con uno de los ortogonales x, y, z. Entonces podríamos usar una función de rotación (por ejemplo R_z) y luego tendríamos que deshacer las transformaciones R y T.



Para volver a colocar al objeto:

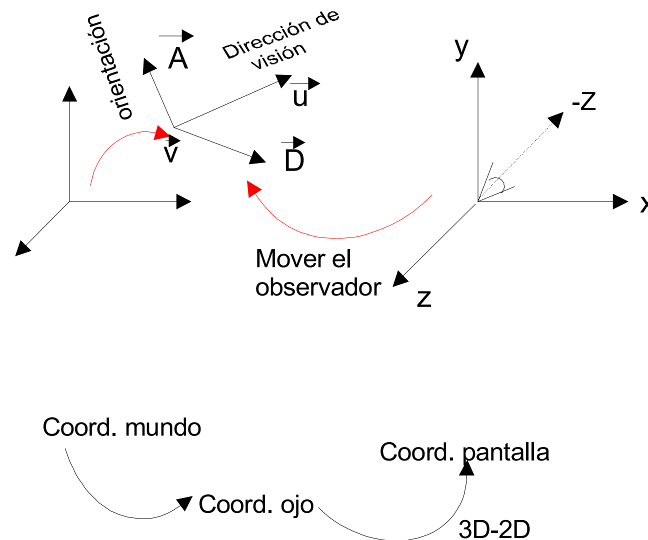


3.2.3. ASIGNACIÓN DE COLOR A LOS VÉRTICES.

Una vez tenemos los vértices situados y orientados en el sistema de coordenadas del mundo podemos aplicar de forma adecuada el modelo de iluminación para asignarles un color, si es que éste no ha sido asignado explícitamente en el programa y el subsistema de iluminación está activado. Como necesitamos efectuar un cálculo lo más eficiente posible, todas las librerías para gráficos en tiempo real utilizan un modelo de iluminación basado en el sombreado local. Este tipo de modelo utilizará los valores de la posición y el vector normal de los vértices junto a la especificación del material y las fuentes de luz, que se encontrará en el contexto gráfico. Si se desea restringir este cálculo a los vértices visibles, pueden efectuarse primero la fase de proyección y recorte que veremos a continuación.

3.2.4. PROYECCIÓN 3D-2D.

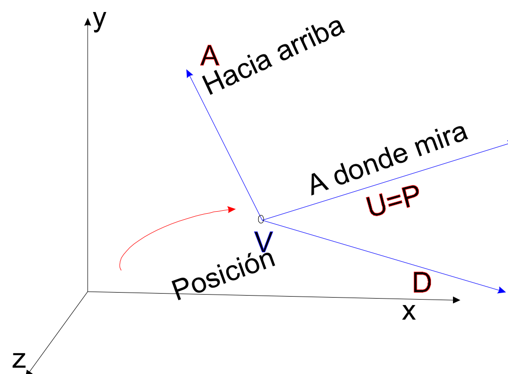
Esta transformación es el paso fundamental para la visualización de escenas tridimensionales, ya que es aquí donde cada vértice de cada triángulo pasa de estar situado en el espacio 3D a hacerlo en coordenadas referidas a la ventana de visualización o *viewport*. Esta transformación se va a realizar en dos pasos: primero efectuaremos la *transformación de visión*, pasaremos del sistema de coordenadas del mundo al *sistema de coordenadas del ojo u observador*, definido por la posición y orientación del observador o cámara, y en segundo lugar efectuaremos la *transformación de perspectiva*, o proyección propiamente dicha, que pasará de coordenadas del observador a *coordenadas de ventana*.



3.2.4.1. Transformación de Visión

Para describir la posición y orientación el observador se utilizan generalmente tres vectores que especificarán los nueve grados de libertad posibles (tres de posición y tres de orientación). Una posible combinación es:

- Definir la posición del punto de vista mediante un vector V.
- Definir un vector unitario con la dirección hacia dónde mira: P
- Definir la dirección vertical para el observador por medio de un tercer vector A



Para hacer coincidir el sistema del mundo con el del observador tendremos que trasladar toda la escena un mismo vector -V, de manera que ahora el punto de vista estará situado en el origen del mundo. A continuación tenemos que efectuar una rotación de manera que ambos ejes se alineen. La matriz total

de la transformación de visión será el producto de las matrices correspondientes a ambas transformaciones: M=RT.

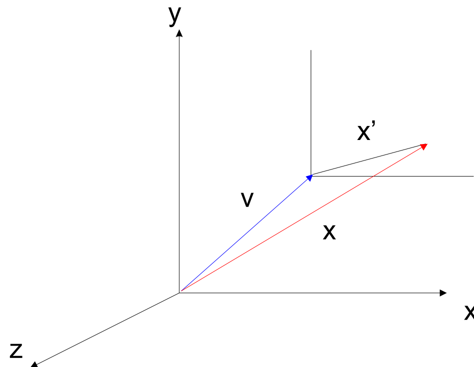
- **Rotación**

Si consideramos los vectores anteriores, y definimos U como el producto vectorial P x A tendremos la matriz:

$$R = \begin{pmatrix} d_x & d_y & d_z & 0 \\ u_x & u_y & u_z & 0 \\ a_x & a_y & a_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Una vez tenemos los vértices de la escena expresados en coordenadas del sistema observador, éstas se pasan a coordenadas de ventana mediante una proyección 3D-2D. Si deseamos una proyección *ortoédrica o paralela*, bastará utilizar dos de las coordenadas del sistema observador como coordenadas de ventana (con las correcciones de escala adecuadas), pero este tipo de proyección no hace disminuir el tamaño de los objetos con la distancia. Para realizar una proyección que simule este efecto se usa el sistema de la *perspectiva lineal o cónica*.

- **Traslación**



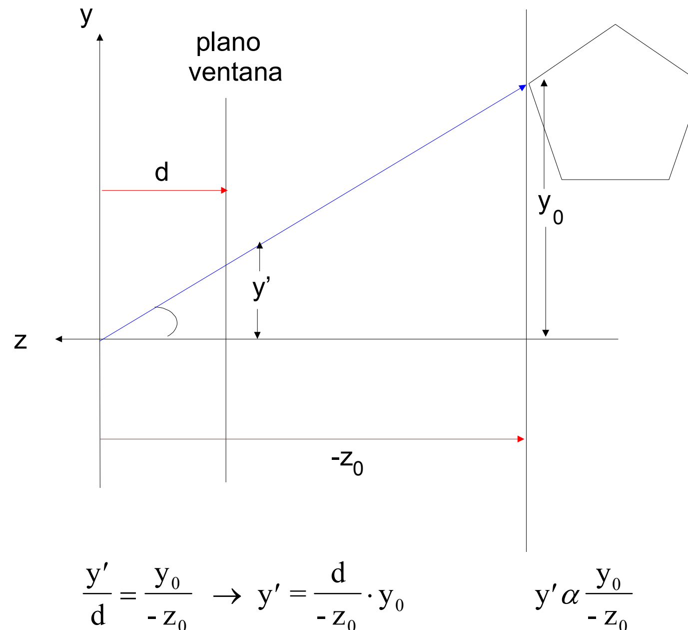
$$T\left(-v_x, -v_y, -v_z\right) = \begin{pmatrix} 1 & & & -v_x \\ & 1 & & -v_y \\ & & 1 & -v_z \\ & & & 1 \end{pmatrix}$$

3.2.4.2. Perspectiva Lineal

La perspectiva lineal es una aproximación (que es utilizada también en el método de trazado de rayos [ENLACE]) conocida desde antiguo para la representación plana de escenas tridimensionales. Consiste en trazar desde los objetos líneas rectas que convergen en un único punto (la posición del observador o cámara). Cuando estas líneas intersectan un plano (que representaría el lienzo, la ventana o la retina del observador) se forma allí la imagen proyectada de la escena se trata de una operación que debemos realizar con todos los vértices. En este tipo de proyección se cumple que cuanto más lejos esté un objeto, más pequeña será su imagen en el plano de proyección.

Al considerar la perspectiva de esta forma estamos realizando dos aproximaciones: la primera es suponer que la distancia a la que se encuentran los objetos (que influye en la disminución de su tamaño aparente) se puede aproximar por su distancia en el eje z. Esta aproximación será buena para ángulos pequeños, pero fallará si aumenta el campo de visión, ya que entonces la diferencia entre la distancia y la componente z será evidente.

La segunda aproximación consiste en considerar que la superficie de proyección es plana, cuando nuestra visión se basa en el uso de una superficie curva. De nuevo sucede que las diferencias no son demasiado grandes si consideramos ángulos de visión pequeños.



$$\frac{y'}{d} = \frac{y_0}{-z_0} \rightarrow y' = \frac{d}{-z_0} \cdot y_0 \qquad y' \alpha \frac{y_0}{-z_0}$$

La perspectiva lineal cumple una propiedad fundamental: la proyección de un triángulo 3D es un triángulo 2D formado uniendo las proyecciones de los vértices, de manera que para cada triángulo de la escena obtendremos su correspondiente triángulo bidimensional sobre el plano de proyección. Para poder realizar de manera eficiente esta computación tenemos que buscar de nuevo una representación lineal, para lo que utilizaremos otra vez una matriz en coordenadas homogéneas.

De la figura podemos deducir fácilmente la fórmula de la perspectiva lineal, que básicamente consiste en dividir las dos coordenadas paralelas al plano de proyección (x,y) por la distancia de cada punto al observador.

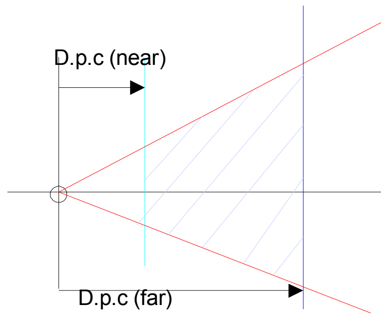
3.2.4.3. Implementación de la Perspectiva Lineal en la Pipeline Gráfica

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ x \\ 1 \end{pmatrix} \qquad \begin{array}{l} x' = x \\ y' = \alpha \cdot y \\ z' = z \\ w' = -z \end{array} \qquad \begin{array}{l} x'_{3D} = x'/w' = x/-z \\ y'_{3D} = y'/w' = \alpha . y/-z \end{array}$$

$\alpha \leftarrow$ representa el aspecto, la proporción anchura/altura de la ventana en pixels.

Notemos que este sistema aprovecha el significado de la coordenada homogénea para realizar una división encubierta con la ayuda de la matriz de transformación, de forma que al final reproducimos el efecto de hacer el tamaño de los objetos inversamente proporcional a su distancia el eje z.

En las librerías gráficas se proporciona además información para recortar el tamaño del espacio de proyección. Por un lado se suele especificar el **ángulo de perspectiva o del campo de visión** (en OpenGL se da el ángulo de visión vertical *fovy* (*field of view* en y)). Por otro lado, como veremos inmediatamente, también el espacio piramidal de proyección (el *frustrum*) se limita según la coordenada z.



3.2.4.4. Utilización de los Planos Cercano y Lejano

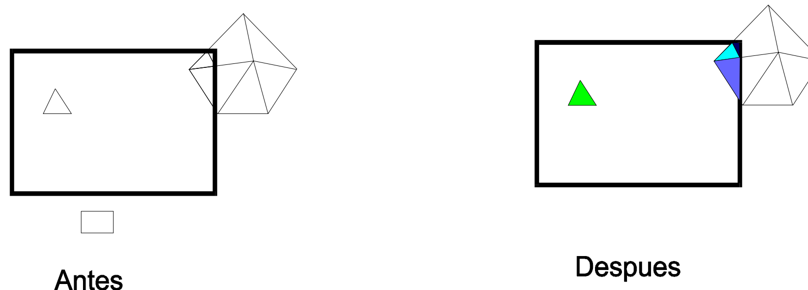
En las librerías para gráficos en tiempo real el programa debe añadir a los parámetros de perspectiva que ya hemos visto la posición de dos planos de recorte, *el plano cercano* y el *plano lejano*, que delimitan un tronco de pirámide donde se sitúan los objetos que serán efectivamente visibles en la proyección. Los objetos que salgan fuera de estos planos no se verán, y los que los intersecten serán recortados, dejando visible solamente la parte interior.

La utilidad de estos planos es doble: por una parte permiten restringir la parte de la escena que se desea ver en función de la distancia. En ciertos casos esto puede permitir que nos centremos en una parte de la escena que nos interesa. Sin embargo, la principal razón para utilizar estos planos es calibrar adecuadamente los valores de profundidad del z-buffer, de la memoria de distancia. En efecto, los valores almacenados en el z-buffer no son valores absolutos de distancia, sino valores relativos a la posición entre los dos planos de recorte, de manera que si estos están muy alejados entre sí, una diferencia de una unidad en la profundidad almacenada en el z-buffer representará una gran distancia en coordenadas del mundo, y ello puede provocar problemas a la hora de efectuar la ocultación de superficies no visibles, ya que dos planos situados a similar distancia pueden confundirse a la hora de comprobar cuál es visible. Por ello es conveniente situar los planos de recorte tan cercanos entre sí como sea posible, lo que aumentará la capacidad de precisión en el test que utiliza el z-buffer.

PIPELINE DE TRANSFORMACIONES EN OpenGL.
Ver Capitulo 3 de "OpenGL Programming Guide" de Addison-Wesley.
Ver Ejemplo Teteras.

3.2.5. RECORTE (CLIPPING).

Una vez llegada a esta fase tenemos ya los vértices de los triángulos proyectados sobre un plano, pero nada impide que sus coordenadas caigan fuera de la ventana. Como la siguiente fase del procesado en tiempo real va a consistir en el rellenado de los triángulos, nos conviene limitar éstos a la parte que este dentro de la ventana, ahorrándonos la interpolación de pixels que van a quedar fuera del área de visualización. Es por ello por lo que se realiza una fase previa de recorte, en la cual se comprueba qué triángulos de la lista de visualización se encuentran dentro de la ventana y cuáles fuera, de manera que:

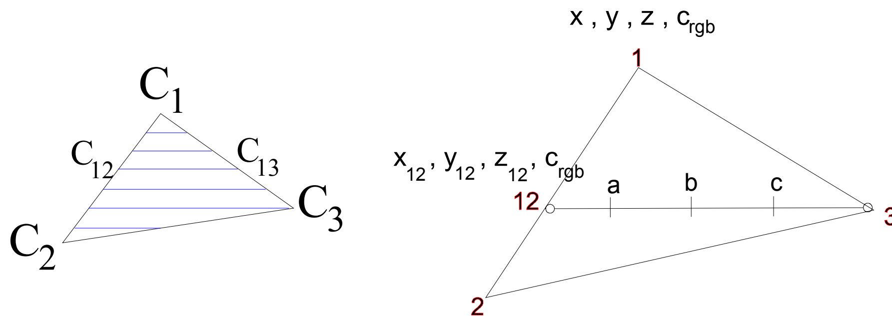


- Si están completamente fuera de la ventana, entonces los desechamos.
- Si están completamente dentro se siguen procesando sin alterarlos.

Para triángulos que tienen una parte dentro y otra parte fuera:
- Se recorta y elimina de la lista de visualización lo que queda fuera.
- Se retriangula la parte que queda dentro de la ventana, calculando por interpolación la posición, el color y otras propiedades de los nuevos vértices que pudieran aparecer.

3.2.6. RELLENO DE LOS TRIÁNGULOS (RASTERING).

Se trata de la última fase, en ocasiones la más costosa, del proceso, por lo que es la primera que se suele integrar en el hardware gráfico. En esta etapa se trata de asignar colores a los pixels correspondientes al interior de cada triángulo proyectado que cae dentro del área de visualización. Los colores asignados deben calcularse por el método de Gouraud, interpolando linealmente entre los colores de los tres vértices. Otras variables como la coordenada de profundidad z o las coordenadas de textura sufren el mismo proceso de interpolación lineal, fácilmente reproducible por hardware.



El algoritmo que generalmente se utiliza para rellenar la mitad de un triángulo como el de la figura es el siguiente:

$\Rightarrow$ hallar x_{12}, z_{12}, c_{12}

$\Rightarrow$ hallar $\Delta x_a, \Delta x_b, \Delta z_a, \Delta z_b, \Delta c_a, \Delta c_b$

$\Rightarrow$ inicialización: $x_a \leftarrow x_1; x_b \leftarrow x_1; z_a \leftarrow z_1, z_b \leftarrow z_1, c_a \leftarrow c_1, c_b \leftarrow c_1$

$\Rightarrow$ desde $y \leftarrow y_1$ hasta y_3

$x_a \leftarrow x_a + \Delta x_a;\ x_b \leftarrow x_b + \Delta x_b;$

$z_a \leftarrow z_a + \Delta z_a\ ;\ z_b \leftarrow z_b + \Delta z_b;$

$c_a \leftarrow c_a + \Delta c_a\ ;\ c_b \leftarrow c_b + \Delta c_b$

hallar $\Delta c_c, \Delta z_c$ || $c_c \leftarrow c_a\ ;\ z_c \leftarrow z_a$

desde $x \leftarrow x_a$ hasta x_b

$c_c \leftarrow c_c + \Delta c_c\ ;\ z_c \leftarrow z_c + \Delta z_c$

si (NO activado test-profundidad ó $z_c < z(x, y)$

$c(x, y) \leftarrow f(c_c, z_c, \ldots)$

$z(x, y) \leftarrow z_c$

Obsérvese que el resultado final es la actualización del color *c(x,y)* y el valor de profundidad en el z-buffer *z(x,y)* para los puntos del área de visualización que superan el test de visibilidad basado en el z-buffer. El color final de estos pixels actualizados, representado aquí por una función $f(c_c, z_c, \ldots)$, dependerá del tipo de operaciones por pixel que estén definidas (transparencia, texturación, etc.)

3.2.6.1. Consideraciones sobre el Coste Computacional

Como comentamos, las operaciones de la pipeline pueden dividirse más o menos en dos partes: las realizadas por vértice y las realizadas por pixel en la fase de rellenado. Podemos decir que el coste computacional es en ambos casos lineal, aunque puede variar para vértices o pixels particulares (dependiendo de ciertos factores como el número de fuentes de luz presentes, el tipo de operaciones de rellenado, etc.). Por ello resulta evidente que habrá que controlar estas variables a la hora de intentar optimizar el coste del proceso de visualización[4].

Otro problema que puede aparecer en el proceso gráfico en tiempo real se debe a la propia estructura de la pipeline, en la que los datos necesarios deben transmitirse de una fase a otra. Esta transmisión en algunos casos puede ser más lenta que el propio procesamiento de los datos, especialmente cuando se utiliza el bus del sistema.

En general la fase de procesamiento o transmisión de datos más lenta restringirá la velocidad global de la pipeline. Cuando una etapa es claramente menos eficiente que las demás se habla de la existencia de un *cuello de botella*, que hay que intentar solucionar para aprovechar la capacidad de otros módulos de la pipeline.
Todos estos factores de coste se describirán con más detalle en el tema 6.

3.2.6.2. Modo Retenido e Inmediato: La Lista de Visualización

Ya hemos comentado que después de la fase de selección de objetos y su conversión a triángulos se crea una lista con los polígonos que deseamos dibujar en el fotograma. Esta *lista de visualización* o *display list* debe contener toda la información necesaria para el procesamiento que efectuará el resto de la pipeline.

Las librerías gráficas pueden funcionar en dos modos diferentes (algunas admiten ambos modos, otras solamente uno). En el llamado *modo inmediato*, la librería generará una lista de visualización diferente para cada fotograma, repitiendo de nuevo el proceso de selección y poligonización. En el modo retenido, la librería guarda la lista de visualización generada para reutilizarla en los siguientes fotogramas, ya que si no hay grandes alteraciones en la escena, basta con cambiar el punto de vista o efectuar transformaciones sobre ciertos objetos para tener los datos correspondientes a las sucesivas imágenes.

La distinción entre estos dos modos se hace importante en ciertas arquitecturas de hardware para la visualización. Hay procesadores gráficos que permiten almacenar la lista de visualización en una memoria propia, evitando el coste de enviar cada fotograma los datos desde la memoria principal. Sin embargo este sistema resulta muy ineficiente cuando hay que efectuar frecuentes actualizaciones masivas de la lista de visualización, debido a que la escena cambia mucho de un fotograma a otro (muchos vértices aparecen, desaparecen, o se mueven).

[4] Ver Tema 6: Simulación en Tiempo Real.