Transformaciones en OpenGL

Aquí se explican las transformaciones en OpenGL. Se incluyen algunas nociones básicas con la única intención de entender la práctica y de qué estamos hablando. En otro documento se detallan las bases teóricas de las transformaciones en general.

Una transformación es la acción de asignar un punto/vector transformado a cada punto/vector original del espacio. Por ejemplo: el punto movido o girado de determinada manera. Se utilizan, por ejemplo, para armar objetos complejos a partir de piezas simples, para posicionar los objetos, las luces y la cámara en la escena y también para proyectar la escena sobre el plano de la imagen.

En la teoría más general justificaremos las dos formas válidas y equivalentes de entender las transformaciones, una consiste en pensarlas como una serie de movimientos y deformaciones que sufren los objetos modelados y la otra considera que es el sistema de coordenadas el que se deforma y se mueve para acomodar los objetos. De una forma dibujo un cuadrado entre {0,0} y {1,1}, luego lo estiro al doble horizontalmente (2x1) y lo giro 30º; de la otra forma giro 30º los ejes del sistema de coordenadas, luego estiro el eje x (2x1) para, finalmente, dibujar ahí un rectángulo girado entre {0,0} y {1,1}. El resultado es el mismo en ambos casos, pero el proceso se describe en sentido inverso.

Debemos recordar que OpenGL funciona como máquina de estados: lo primero que se lee en el programa se aplica a todo lo que sigue, por lo tanto es lo último que se hace sobre los objetos. En una máquina de estados, la segunda forma de ver las transformaciones permite programar y leer el programa en forma natural, progresiva; pero casi siempre se requiere una verificación cruzada entre las dos interpretaciones. Suponiendo que hay una rutina que dibuja rectángulos, las operaciones anteriores en OpenGL se verían así:

```
glRotatef(30,0,0,1); glScalef(2,1,1); Draw_Rectangle(0,0,1,1);
```

y pueden entenderse como operaciones sucesivas sobre el sistema de coordenadas.

En OpenGL se utilizan muchas transformaciones. A grandes rasgos: las primitivas y partes simples se ubican para formar piezas más complejas, que a su vez se colocan en la escena, en el **espacio del modelo**, que es un espacio arbitrario, definido por el usuario para armar la escena. En ese mismo espacio se definen las luces y la cámara, su posición y orientación. Luego se define el mecanismo de proyección (perspectiva u ortogonal) a la vez que se especifica la fracción del espacio que será visible, descartando lo que quede fuera de ese volumen. Aquí el espacio (las coordenadas) es el **espacio visual**, con **x** hacia la derecha de la vista, **y** hacia arriba y **z** pinchando el ojo; en este espacio se pueden ubicar todos los objetos o luces que se mueven solidarios con la cámara. Luego se mapea el volumen visual en un prisma cuyo frente se adapta al tamaño de la imagen pero mantiene la profundidad (**z**) para rasterizar usando el *depth−buffer*. Por último se aplasta y se ubica la imagen en la ventana del programa. Esas transformaciones, en el programa, se definen en orden inverso.

La secuencia de transformaciones en OpenGL se realiza mediante la premultiplicación de una matriz de 4x4 que codifica la transformación, por cada vector de cuatro dimensiones que representa un punto, una dirección o un plano. Las transformaciones tienen las mismas propiedades que esas operaciones, en particular la falta de conmutatividad. La cuarta dimensión es un truco que permite usar la misma operación de multiplicación para proyectar en perspectiva y trasladar; con sólo tres dimensiones esas transformaciones no serían tan sencillas. La placa gráfica es una máquina especializada en hacer esas operaciones además de manejar bloques de píxeles. La cuarta dimensión se llama aquí "coordenada homogénea": a un punto de coordenadas $\{\mathbf{x},\mathbf{y},\mathbf{z}\}$ le corresponde el vector homogéneo $\{\mathbf{wx},\mathbf{wy},\mathbf{wz},\mathbf{w}\}$, con cualquier $\mathbf{w} \neq 0$. En el caso en que **w** sea cero, diremos que se trata de un vector o una dirección o "un punto en el infinito" que es una forma común, en la jerga de CG, de denominar a una dirección en el espacio, por ejemplo: la posición del sol en el infinito, para iluminar una escena.

Los detalles de los distintos espacios o sistemas de coordenadas los analizaremos en el orden en que aparecen al leer el programa.

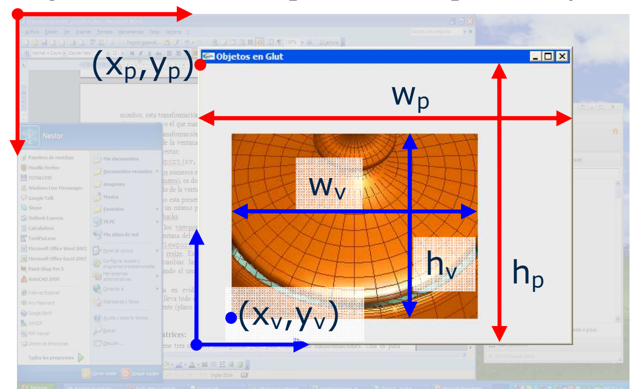
Dispositivo de salida y ventana:

La ventana en el dispositivo de salida (monitor, impresora, archivo) se define al inicializar el subprograma gráfico. Si se trata de un monitor, glut pide una ventana gráfica al sistema operativo:

```
glutInitWindowSize(wp, hp);
glutInitWindowPosition(xp, yp);
```

los parámetros son números enteros que representan posición y tamaño de la ventana, medidas en píxeles en el sistema de coordenadas del dispositivo (*device coordinates*) o imagen donde se vaya a renderizar. Hay que recordar que tanto para glut, como para el sistema operativo y para las imágenes el origen es el píxel de arriba y a la izquierda {0,0} → *upper−left*.

Esta primera llamada, que mapea la ventana del programa en el dispositivo de salida, es la última transformación que sufre la escena. Siendo una salida al monitor, esta transformación se define únicamente en la inicialización (notar el glutInit…) luego es el sistema operativo el que maneja el movimiento y cambio de tamaño de la ventana.



La penúltima transformación es la que aplasta el espacio visible en un *viewport* o ventana de dibujo, y lo ubica dentro de la ventana del programa:

```
glViewport(xv, yv, wv, hv);
```

los números enteros definen el rectángulo en píxeles. Para OpenGL, el origen de la ventana del programa es el píxel inferior izquierdo.

Las coordenadas dentro del *viewport* se denominan *window coordinates* y también tienen su origen en el píxel *lower−left*. Aquí se dibujaran los objetos al rasterizar; por lo cual es un sistema 3D, con **z** entre cero (plano *near*) y uno (plano *far*). También se dibujan aquí los recuadros de selección, indicadores y, en el caso de las GUI, los cuadros de diálogo, texto, *sliders*, etc. es decir todos los elementos del plano de la imagen, con **z** = 0 y medidas en píxeles.

Si `glViewport` no esta presente, se considera que es toda la ventana del programa. Puede haber más de un *viewport* en un mismo programa; cada uno debe tener sus propios *callbacks*.

El tamaño de los *viewports* cambia junto con el tamaño de la ventana del programa, por lo tanto las llamadas a `glViewport` solo aparecen en la rutina que maneja el *resize*. El programador decide como cambiar las ventanas de dibujo o información cuando el usuario altera la ventana del programa.

Matrices y *Stacks*:

Las anteriores transformaciones eran mapeos sencillos, a partir de aquí OpenGL mantiene matrices homogéneas (4x4) para realizar las transformaciones que siguen. Hay tres matrices, una para mapear el modelo al sistema visual; otra para definir la porción del espacio visible y mapearlo en un cubo canónico y la tercera para la aplicación de texturas, que cumple funciones similares y con métodos similares, pero que no trataremos aquí. Las dos primeras se explican en detalle más adelante, pero aquí analizaremos como se *backup*ean y restauran y como se opera con ellas.

Hay que definir la matriz activa con una llamada a la función:

```
glMatrixMode([GL_MODELVIEW, GL_PROJECTION o GL_TEXTURE]);
```

las operaciones subsiguientes se aplicarán sobre la matriz seleccionada. En cada caso, la matriz elegida, con los valores que tuviese de antes, se multiplica o se reemplaza por la que aparece en la operación y se obtiene la matriz actual. Hay tres operaciones básicas que alteran la matriz:

- Reemplazo por pop.
- Reemplazo por carga.
- Composición por multiplicación.

OpenGL provee un *stack* o pila para cada una de las tres matrices. La pila permite guardar (*push*) un estado para recuperarlo (*pop*) posteriormente:

```
glPushMatrix();      glPopMatrix();
```

La llamada a *push* guarda una copia de la matriz actual en el *stack*; la matriz guardada es igual a la que sigue siendo activa; la altura del *stack* se incrementa. La llamada al *pop* reemplaza la matriz actual por la última que se puso en el *stack* ("lifo" o *last−in, first−out*), la altura de la pila decrece una unidad y la matriz reemplazada se pierde. Funciona como un proceso de *backup/restore*.

La altura máxima de cada pila está relacionada con el uso; *model−view* se utiliza mucho y tiene una profundidad de al menos treinta y dos matrices; las pilas de proyección y texturas, tienen al menos dos.

Se puede reemplazar la matriz actual por una matriz guardada en memoria. La función:

```
glLoadMatrixf(GLfloat *m);
```

carga el conjunto de 16 *floats* apuntados por `m` y lo pone en lugar de la matriz actual, la matriz que estaba no se guarda, se pierde. Esta llamada es para introducir alguna matriz especial calculada por software. En cambio,

```
glLoadIdentity();
```

es una llamada especial, que carga la matriz identidad y es la llamada habitual para empezar el proceso, el punto de partida.

La combinación de transformaciones (transformación de transformación) se realiza con operaciones automáticas de multiplicación. Las operaciones estándar se realizan con matrices generadas en forma automática por OpenGL, pero el programador puede usar una matriz guardada en un array:

```
glMultMatrixf(const GLfloat *m);
```

reemplaza la matriz actual `a`, por el producto `a*m`.

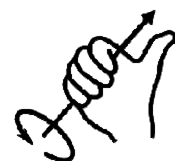
OpenGL usa las matrices en *float* (los *doubles* se aceptan pero son convertidos) y están ordenadas por columnas, es decir que están transpuestas respecto al estándar de C:

- Si `m` se define como `float[16]`, el segundo elemento: `m[1]` es el de la columna 0 y fila 1.
- Si se define como `float[4][4]`, `m[0][1]` es el elemento en la columna 0 y fila 1, `m[2]` es un puntero a la columna 2 (veremos en la teoría que es el nuevo vector base para las coordenadas **z**).

Para las operaciones habituales como traslación, rotación y escalado, OpenGL hace la matriz y el producto; sólo hay que usar alguna de las siguientes llamadas:

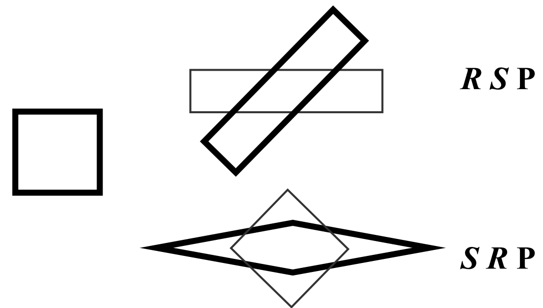
```
glTranslatef(tx, ty, tz);
glRotatef(ang, rx, ry, rz);
glScalef(sx, sy, sz);
```

donde los parámetros son obvios excepto para la rotación de ángulo `ang`, en grados sexagesimales, alrededor de una recta por el origen definida por el vector `r` = {`rx,ry,rz`}, el sentido de la rotación está dado por el signo del ángulo y la regla de la mano derecha sobre el vector. Para rotar alrededor de un eje que no pasa por el origen, habrá que trasladar el origen al eje, rotar y volver el origen a la posición inicial.



En todos los casos de composición, la matriz actual `a` se reemplaza por una que consiste en el producto de `a` por la matriz que entra `m`: `a*=m` o `a[i][j]=`Σ_k`(a[k][j]*m[i][k])`. Como corresponde, la matriz `a` provee la fila `j` y `m` provee la columna `i`. El producto es el correcto, pero los índices [columna][fila] están al revés de lo usual y que se aprendió en álgebra: $a_{ji} = \Sigma_k\, a_{jk}\, m_{ki}$.

$$\underline{P} = \ldots\ C(B(A(P))) = Z(P)$$
$$Z = C\,B\,A$$
$$C\,B\,A = C(B\,A) = (C\,B)\,A \quad (\neq C\,A\,B)$$



Es por esta sucesión reemplazos por productos que cuando se quiere "partir de cero" se carga la matriz identidad, en caso contrario los cambios se acumulan. El programador decide como hacer las cosas.

Todas estas operaciones las hace OpenGL, normalmente en la placa gráfica.

Matriz de Proyección:

Pese al nombre, la matriz de proyección no proyecta, pero sí define como se realizará la proyección. En primer lugar se llama a:

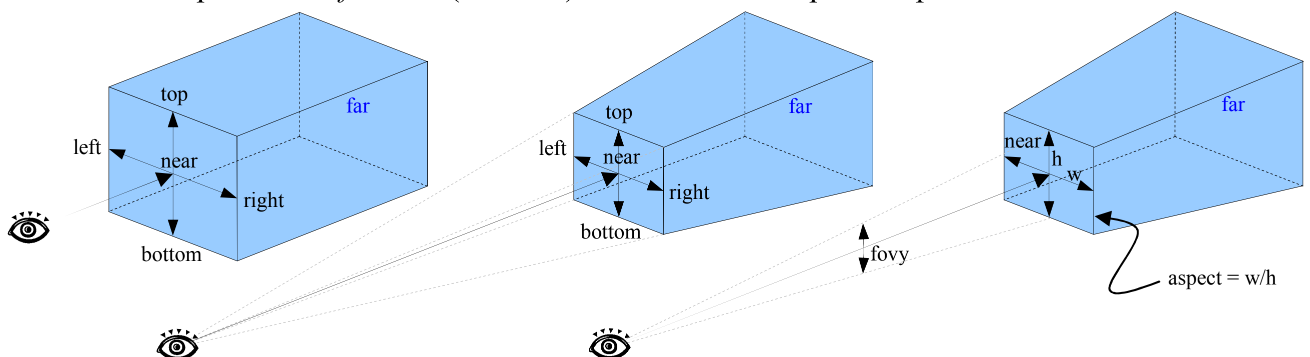
```
glMatrixMode(GL_PROJECTION); glLoadIdentity();
```

OpenGL ofrece dos modos de proyección o "modelos de cámara": ortogonal y perspectiva central, cuyas matrices se generan automáticamente a partir de algunos datos relevantes:

- Proyección ortogonal (paralelepípedo):
  ```
  glOrtho(left, right, bottom, top, near, far);
  ```
- Proyección perspectiva (*frustum* o tronco de pirámide):
  ```
  glFrustum(left, right, bottom, top, near, far);
  ```

Estos datos, en *float,* se definen en el sistema de coordenadas visual, de la cámara o del observador. En ambos casos se especifican las coordenadas visuales de los planos de recorte o *clipping−planes* que limitan el volumen a visualizar. En éste modelo, la cámara no ve desde cero hasta el infinito, sólo será visible lo que quede entre un plano cercano al ojo, a distancia `near` y otro alejado, a distancia `far`.

Para la proyección ortogonal, el campo visual es un paralelepípedo, se definen las posiciones de los seis *clipping−planes* que recortan el espacio. En el caso de la proyección perspectiva, el campo visual es un tronco de pirámide o *frustum* (en Latín) definido también por seis planos.



Todo lo que esté por fuera de ese recinto no se visualiza (*culling* = descarte) y los objetos que lo atraviesan serán recortados (*clipping* = poda, recorte), solo se verá la parte interior. Podemos decir que la matriz de proyección sólo define los seis planos principales de recorte o *clipping* de la escena.

Los parámetros `near y far son distancias positivas` de los planos al ojo. En perspectiva central, los planos *near* y *far* deben estar delante del ojo (`far` > `near` > 0).

Los otros parámetros son coordenadas **x** o **y** de los límites visuales, que normalmente se centran alrededor de cero (el ojo). En vista ortogonal da lo mismo; pero descentrar la vista en perspectiva, muestra en el monitor una porción de lo que se vería mirando para otro lado; normalmente el

programador pretende que el usuario sea el observador con la vista centrada en el centro del *viewport*; aunque un caso razonable para usar *viewports* excéntricos sería para armar una escena por pares en un *array* de monitores. Dado que las bases del frustum tienen distinto tamaño, los parámetros que definen los planos laterales están fijados, por convención, en el plano *near*. La distancia del ojo al plano *near* es entonces la que define el ángulo de apertura del campo visual (*field of view* o fov).

Otra opción para definir el frustum (centrado) de la perspectiva es una función de la biblioteca GLU:

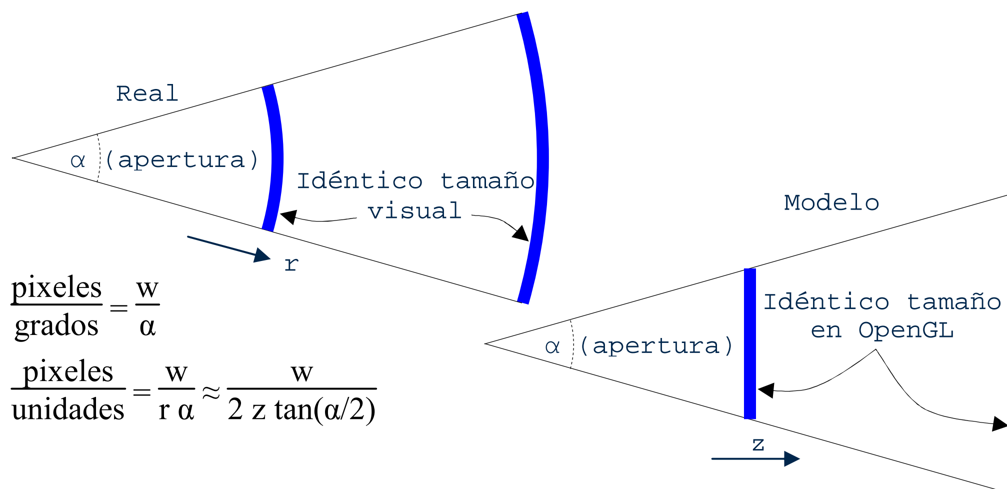
```
gluPerspective(fovy, aspect, zNear, zFar);
```

donde `fovy` es el ángulo de visión vertical en grados y `aspect` el cociente ancho/alto.

Luego de definir el modelo de cámara, se realiza, para cada vértice, la división por la coordenada homogénea **w** (división perspectiva) y el volumen de visualización tridimensional o *clip−space*, se mapea internamente en un cubo $[-1,1]^3$ (NDC o *Normalized Device Coordinates*), conservando la profundidad para poder realizar el ocultamiento de líneas al rasterizar.

El volumen visible esta definido en unidades espaciales, pero aun no esta ubicado en la escena. Aún así, dado que el cubo canónico será aplastado y mapeado en el *viewport*, definido en pixeles, entonces ya está determinada la escala visual en píxeles/unidad.

Aquí apareen detalles importantes de la perspectiva: 1) Al mapear el *frustum* en un cubo, los objetos alejados se comprimen mas que los cercanos (perspectiva lineal) y la escala depende de z. 2) Cuando se analiza la matriz de transformación perspectiva, se puede ver que la coordenada **z** no se mapea linealmente, pero 1/**z** sí; de modo que el *depth−buffer* es más preciso (menos *z−fighting*) cerca del ojo.



$$\frac{\text{pixeles}}{\text{grados}} = \frac{w}{\alpha}$$

$$\frac{\text{pixeles}}{\text{unidades}} = \frac{w}{r\,\alpha} \approx \frac{w}{2\,z\,\tan(\alpha/2)}$$

En el modelo ortogonal **z** sí se mapea linealmente y por lo tanto con precisión independiente de la profundidad. En ambos casos, en el NDC, **z** vale -1 en el plano *near* y 1 en el plano *far*, por lo que el sistema NDC es izquierdo (es más importante recordar que en sistema visual estaban entre cero y uno).

Composición de la escena:

Todos los objetos se dibujan y posicionan en un sistema de coordenadas arbitrario que se suele denominar *world coordinate system* o sistema de coordenadas global o espacio del modelo, este sistema es arbitrario, del usuario o del programador. Los objetos individuales se suelen definir en un sistema propio del objeto pero luego son ubicados en el espacio (intermediario) del modelo y finalmente todas las coordenadas son transformadas al espacio visual de la cámara.

Se comienza con la llamada de definición del estado inicial de la matriz *model-view*:

```
glMatrixMode(GL_MODELVIEW); glLoadIdentity();
```

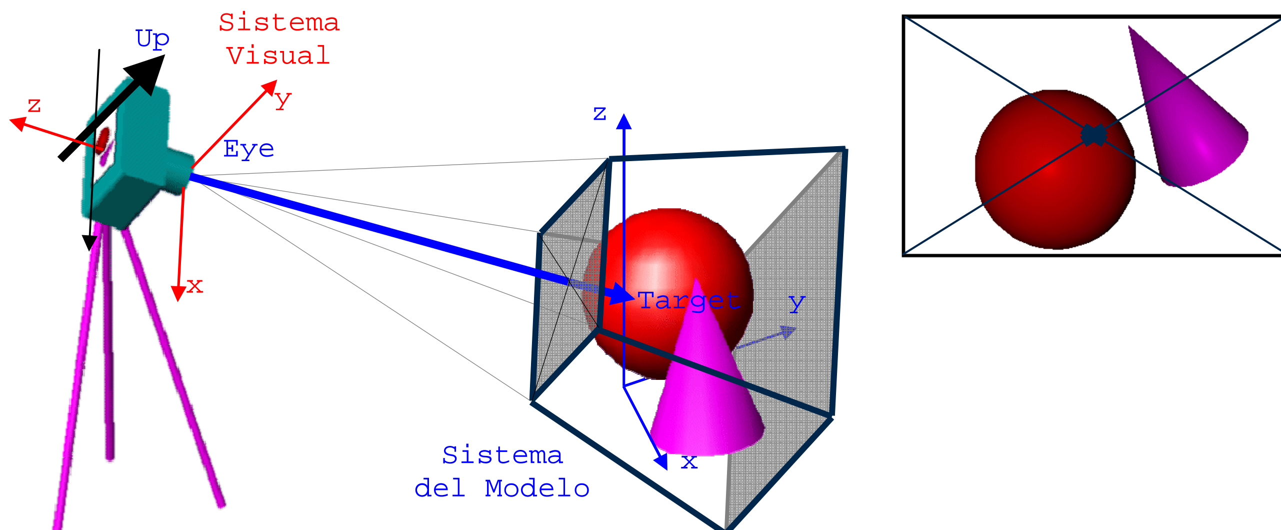
la identidad nos coloca, a partir de aquí, en el sistema de coordenadas visual o de la cámara, que tiene el eje **x** positivo hacia la derecha, **y** hacia arriba y **z** hacia atrás del ojo. En este espacio se pueden definir las posiciones de los objetos que se mueven junto con la vista, por ejemplo el arma de un jugador en primera persona o la luz de una lámpara que se mueve con la cámara (un flash).

Ya se definió, mediante la matriz de proyección, el tipo de cámara y el espacio visible; ahora hay que ubicarlos en la escena, hay que asignarle posición y orientación. Hay dos formas de hacerlo: una es ubicar la cámara (el sistema de coordenadas del ojo) en el espacio del modelo, con `gluLookAt()` y la

otra es ubicar el espacio del modelo en el sistema del ojo con traslaciones y rotaciones. La primera es mucho más intuitiva:

```
gluLookAt(eyex,eyey,eyez,centerx,centery,centerz,upx,upy,upz)
```

los parámetros son las coordenadas del ojo o cámara (*eye*), el punto al que se mira (*center o target*) que puede ser un punto cualquiera, sin contenido material y, finalmente, un vector *up* que indica la dirección de la cabeza o parte superior de la cámara, para fijar la inclinación de la vista o la rotación de la cámara alrededor del eje ojo-centro.



A partir de aquí, estamos en el sistema de coordenadas global o espacio del modelo. Aquí suelen posicionarse las luces y los objetos de la escena.

El *stack* de matrices permite armar modelos complejos utilizando partes preensambladas. Cuando se preensambla una pieza en una rutina, se cargan y se mueven algunos objetos (probablemente preensamblados en otra rutina). Ejemplo, auto:

```
Posicionarse en el sistema global, donde se ubicará el auto
Dibujar auto: {
  Push (matriz auto)
  Mover el sistema de coordenadas al centro de rueda delantera izquierda
  Dibujar rueda: {
    Push (matriz rueda di)
    Posicionar bulón 1
    Dibujar bulon 1: {Push…….Pop}
    Pop (matriz rueda di)
    ……
    ……
    Push (matriz rueda di)
    Posicionar bulón 4
    Dibujar bulon 4: {….}
    Pop (matriz rueda di)
    Dibujar cubierta
    Dibujar llanta
  }
  Pop (matriz auto)
  ….
  ….
  Mover al centro de rueda trasera izquierda
  Dibujar rueda: {….}
  Dibujar otras partes del auto....
  Pop (matriz auto)
}
```

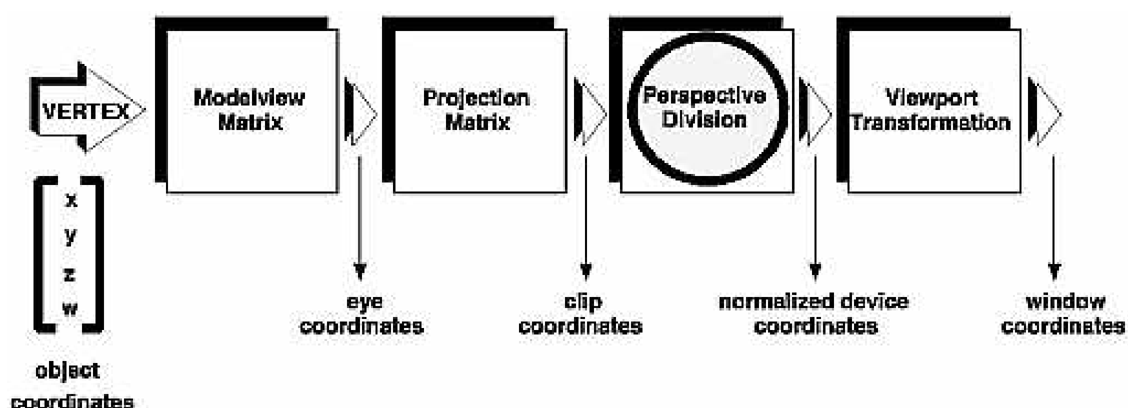
La rutina para dibujar el auto se puede llamar tantas veces como se quiera, armando una rutina `dibujar_auto()` y definiendo las transformaciones necesarias para llevar cada auto a la posición que le corresponde en la escena. Lo mismo sucede con las piezas. Por eso es indispensable que cada

rutina de dibujo "deje las cosas como estaban" antes de salir, ya sean las matrices o el resto del estado (*push* y *pop attributes*) que vaya a modificarse dentro de la rutina.

El modelado en sí suele realizarse en programas de CAD o cualquier otro software especializado, para luego dejar que OpenGL posicione las primitivas leídas. También puede realizarse directamente en OpenGL, pero con muchísimas limitaciones, componiendo primitivas básicas con algunas piezas de alto nivel provistas por GLU o GLUT.

Pipeline:

La secuencia de operaciones es la siguiente:



Toda la escena: modelo, luces y cámara, se transforma mediante la matriz *model-view* (*model→view*) hacia el sistema visual.

La escena, vista desde el ojo, es transformada luego mediante la matriz de proyección, que recorta (*clip*) el espacio visible, pero conservando el w del espacio proyectivo (4D).

Luego se procede a la (¿mal llamada?) división perspectiva, que simplemente divide las coordenadas por **w** y "normaliza" a un cubo $[-1,1]^3$ conservando un **z**, que puede ser no-lineal, pero es monótono, y sirve para hacer ocultamiento. Esta transformación se realiza siempre en la misma forma y por eso no es necesaria la definición explícita por parte del usuario.

Luego se utiliza la definición del `glViewport` (y `glDepthRange`, pero no se usa) para acomodar los valores a un dispositivo de salida específico. Los valores de **x** e **y** irán al dispositivo de salida y el de **z** al *depth−buffer*.

Finalmente se realizan las operaciones "por píxel", ocultando, rasterizando y rellenando.

En glut_base

Si se lee el programa se puede observar la secuencia de sucesos explicada:

1) El tamaño de la ventana se define en el `main` y cambia junto con el viewport en el `resize`.
2) Se define el campo visual cargando la matriz de proyección y la proyección adecuada.
3) Luego se inicializa la matriz *modelview* y se cargan las luces y la cámara (o viceversa).
4) Finalmente se dibujan los objetos.

En el programa se mantiene el origen del espacio del modelo en el centro del *viewport*, pero podría hacerse otro mapeo cualquiera de las coordenadas del modelo a las visuales.

La cámara siempre mira al origen desde una esfera de radio 5 constante. La posición se define mediante latitud y longitud que varían al mover el *mouse* o en la rotación animada del `Idle_cb()`. En la rutina `calc_eye()` se traducen esos valores en coordenadas cartesianas y se calcula el vector *u,* que se mantiene perpendicular a los paralelos y al vector camara-origen.

El *zoom* es esencialmente bidimensional, solo agranda o achica la imagen, la escala se aplica a **x** e **y**, pero no a **z**. Eso se nota pues afecta a `w0` y `h0`, que definen los *clipping planes* centrados; pero no afecta a `znear` y `zfar`, que se mantienen en 2 y 8. Estos números se entienden así: la cámara siempre está a distancia 5 del origen, por lo tanto se ve lo que esté a menos de 3 unidades del origen hacia el ojo o hacia el otro lado. Todos los objetos fueron escalados para caber en una esfera de radio 3.

Estas técnicas simples hacen que el programa sea simple pero no es 3D real, si se quisiera hacer un *pan*eo o movimiento horizontal de la cámara, sin que quede en evidencia el *zoom* bidimensional, habría que hacer mantenimiento de `znear` y `zfar`, para que los objetos permanezcan visibles pero con los planos bien ajustados. Eso haría una verdadera GUI de manipulación en 3D.

Como puede observarse, `znear` y `zfar` son positivos. En primer lugar, todas las ubicaciones de los clipping planes se dan en el sistema del ojo, pero aún así, deberían tener coordenadas **z** negativas. Hay que entenderlos como distancias y no como coordenadas, es muy importante no confundirse en ello. Si bien en proyección ortogonal se admiten negativos, en perspectiva ambos deben ser positivos.

Trabajos de aplicación:

En glut_base:

1) Comentar la llamada a `glViewport` y analizar el resultado.
   Cambiar `glViewport`: `(100,50,w,h)`, `(100,50,w-100,h-50)`, `(20,10,w-40,h-20)`.
   *Zoom*ear mucho para ver el viewport lleno y des*zoom*ear mucho para ver donde esta el centro.
2) Cambiar el `znear` y el `zfar` a 4.3 y 5.7 respectivamente.
3) Analizar los parámetros en la definición de la proyección perspectiva y ortogonal.
4) Donde se inicializa la matriz modelview cambiar:

```
glMatrixMode(GL_MODELVIEW); glLoadIdentity();
glTranslatef(1,2,3);
float mat[16];
glGetFloatv(GL_MODELVIEW_MATRIX,mat);
for (int j=0;j<15;j++) cout << mat[j] << ",";
cout << mat[15] << endl;
glLoadIdentity();
```

5) Cambiar gluLookAt(…) por lo siguiente:

```
float
d=sqrt(eye[0]*eye[0]+eye[1]*eye[1]+eye[2]*eye[2]),
  ac=atan2(eye[1]/d,eye[2]/d)/atan(1.0f)*45;
glTranslatef(0,0,-d);
glRotatef(ac,1,0,0);
```

6) Analizar el parámetro que cambia de luz posicional a direccional.
7) Analizar la ubicación de la luz fija o con la cámara.
8) Analizar las dos interpretaciones de la rutina que dibuja el cubo a partir de copias del cuadrado.