Transformaciones en OpenGL

Aquí se explica cómo se utilizan las transformaciones en *OpenGL*. Se incluyen solamente algunas nociones básicas con la única intención de entender la práctica. 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 en varias etapas, primero para armar los objetos complejos a partir de piezas simples, luego para posicionar los objetos, las luces y la cámara en la escena y más adelante, para generar una imagen de la escena.

Cada vértice, posición, normal, dirección de luz, plano, etc., se representa mediante un vector columna de 4 dimensiones y cada transformación se representa mediante una matriz de 4x4. La transformación se realiza multiplicando la matriz actualmente activa por los vectores que van llegando. Para combinar varias transformaciones, se premultiplican las matrices. Las transformaciones tienen las mismas propiedades que esas operaciones, en particular la falta de conmutatividad.

Recordemos que *OpenGL* funciona como máquina de estados: lo que se lee en el programa se aplica de ahí en adelante; y lo mismo sucede con las matrices de transformación. Por esto, esas transformaciones, en el programa, se definen en el orden inverso al que se aplican a un dado vector. Un punto o vector termina siendo multiplicado por el producto de todas las matrices que están definidas antes de enviar ese vector al *pipeline* gráfico.

1. Puntos y Vectores 4D

La cuarta dimensión no es más que un truco, que permite usar multiplicaciones matriz-vector para proyectar en perspectiva y trasladar; con sólo tres dimensiones esas dos transformaciones no serían tan sencillas. La GPU es una máquina especializada en hacer productos matriz-vector 4D y manejar bloques de píxeles. Las cuatro dimensiones se denominan **coordenadas homogéneas**.

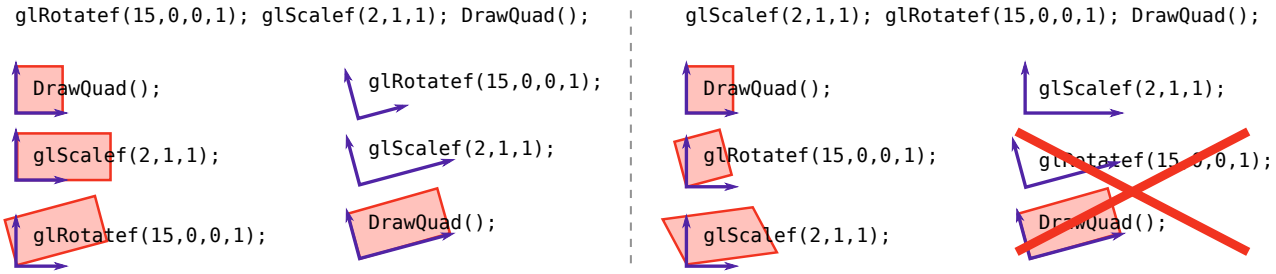
Los puntos y vectores *son* 3D, pero *se representan* en 4D. Un punto o vértice en $\{x, y, z\}$ tiene coordenadas homogéneas $\{wx, wy, wz, w\}$, con cualquier $w \neq 0$, se define normalmente como $\{x, y, z, 1\}$, pero al ser multiplicado por algunas matrices adquiere un $w \neq 1$ y su nueva posición será $\{wx, wy, wz, w\}$, que representa un punto en $\{x, y, z\}$. Los vectores o direcciones se definen de entrada con w nulo, diremos que $\{\Delta x, \Delta y, \Delta z, 0\}$ es un vector o dirección, o *un punto en el infinito* que es una forma común, en CG, de denominar a una dirección. Por ejemplo: la posición del sol en el infinito, es en realidad la dirección de los rayos de luz, iluminan la escena desde una dirección constante y común a todos los objetos.

2. Transformaciones y Espacios

Supongamos que hay una rutina `DrawQuad()` , que dibuja un cuadrado entre $\{0,0\}$ y $\{1,1\}$, alineado con los ejes. Veamos el siguiente código:

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

En la teoría general justificaremos dos formas válidas y equivalentes de entender las transformaciones. De una forma ($\leftarrow$): dibujo un cuadrado entre $\{0,0\}$ y $\{1,1\}$, luego lo estiro al doble horizontalmente y lo giro 30^o. De la otra forma ($\rightarrow$): giro 30^o los ejes del sistema de coordenadas, luego estiro el nuevo eje x al doble, para dibujar ahí un rectángulo entre $\{0,0\}$ y $\{1,1\}$ *de ese sistema transformado*. El resultado es el mismo en ambos casos, pero el proceso se describe en sentido inverso. La segunda forma de ver las transformaciones permite programar y leer el programa en forma natural; pero si hay rotación y escala no uniforme, se requiere una verificación cruzada, pues la más fácil de interpretar es la primera forma. Se puede ver el problema cambiando el orden de la rotación y el escalado en el código anterior.



Normalmente los objetos vienen construidos en un sistema CAD o similar, o se ensamblan a partir de primitivas, como nuestro simple cuadrado unitario. Los objetos se construyen con su propio sistema de coordenadas, normalmente centrado o en un borde. Ese sistema de coordenadas suele denominarse *sistema del objeto* o se dice que las coordenadas de los vértices están en el **espacio del objeto**.

La escena se arma en un espacio arbitrario, definido por el usuario: el **espacio del modelo**. Los objetos se ubican allí mediante transformaciones (matrices) **model–view**: cada objeto es trasladado, orientado y escalado para ubicarlo en la escena, con una misma matriz para todos sus vértices del objeto. En ese mismo espacio se definen las luces y se ubica la cámara o el ojo, tanto su posición como su orientación.

Mediante una llamada a `gluLookAt()` se define una transformación particular que transforma todo vértice y todo vector de la escena en el espacio del modelo hacia el **espacio visual**, con origen en el ojo, x hacia la derecha de la vista, y hacia arriba (hacia la cabeza, que puede estar inclinada) y z pinchando el ojo; en este espacio se pueden ubicar todos los objetos o luces no están fijas, sino que se mueven solidarios con la cámara (un arma o una linterna que se mueve con el personaje, una luz tipo flash, etc.).

Luego se aplica la **projection matrix**, que es una transformación que define el mecanismo de proyección (perspectiva u ortogonal) a la vez que se especifica la fracción del espacio que será visible, el volumen visual o **clipping volume**, descartando lo que quede fuera de esa región.

Finalmente se mapea el volumen visual al **viewport** o **espacio de la ventana gráfica (window–space)**, un *prisma* (¡todavía 3D!) cuyo frente se adapta al tamaño de la imagen, definido por el ancho y alto del *viewport*, pero mantiene la profundidad z para saber quién tapa visualmente a quién.

3. Selección y Modificación de Matrices

OpenGL mantiene matrices homogéneas (4x4) para realizar las transformaciones. Hay cuatro matrices que manipula el programador: una para mapear las coordenadas del modelo al sistema visual (*modelview*), otra para definir la porción del espacio visible y mapearlo en un cubo canónico (*projection*), la tercera para alterar las coordenadas de textura (*texture*), y la cuarta transforma el color (*color*). Las dos últimas raramente se utilizan; las dos primeras alteran las coordenadas de los vértices y se explican en detalle más adelante.

Ahora analizaremos cómo se modifican y cómo se opera con ellas. Primero hay que definir la matriz activa con una de las siguientes llamadas a la función `glMatrixMode`:

```
glMatrixMode(GL_MODELVIEW);
glMatrixMode(GL_PROJECTION);
glMatrixMode(GL_TEXTURE);
glMatrixMode(GL_COLOR);
```

las operaciones subsiguientes se aplicarán sobre la matriz seleccionada, hasta que se seleccione otra. En cada caso, la matriz activa, con los valores que tuviese al momento, se multiplica o se reemplaza por la que aparezca en el código subsiguiente y se va actualizando.

3.1. El *stack* de matrices

Para cada una de las matrices OpenGL provee un *stack* o pila; que permite guardar (*push*) una matriz para recuperarla (*pop*) posteriormente:

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

La llamada a *push* guarda una copia de la matriz actual en la cima del *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* (pila, *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.

3.2. Carga directa de matrices

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

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

carga el conjunto de 16 `float`s cuyo puntero es `m` y lo pone en lugar de la matriz actual, la matriz que estaba activa no se guarda, se pierde. Esta llamada es para introducir alguna matriz definida o calculada por software. En cambio,

```
glLoadIdentity();
```

es una llamada del mismo tipo, pero especial: reemplaza la matriz actual por la identidad. Es la llamada habitual para empezar el proceso; así como para contar se empieza poniendo en cero, para multiplicar se empieza con la unidad, de otro modo los cambios se acumularían sobre lo que estaba.

OpenGL usa las matrices en `float` (los `double`s 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]`, `m[6]` es el elemento de la segunda columna (1) y tercera fila (2). Las columnas `m+0`$=\mathbf{e}_x$, `m+4`$=\mathbf{e}_y$ y `m+8`$=\mathbf{e}_z$ son los nuevos vectores base del sistema de coordenadas (normalmente con sus cuartas componentes nulas: `m[3]=m[7]=m[11]=0`), mientras que `m+12`$=$ **o** es el nuevo origen de coordenadas o la traslación (normalmente `m[15]=1`).

3.3. Multiplicación de matrices

La combinación de transformaciones 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 $a * m$.

Para las operaciones habituales como traslación, rotación y escalado, en una base ortonormal, *OpenGL* hace la matriz y el producto (composición); sólo hay que usar alguna de las siguientes llamadas:

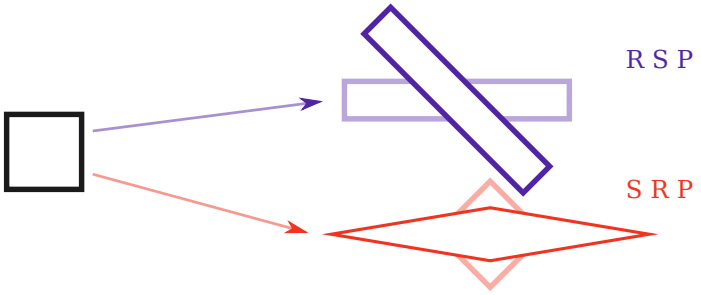
```
glTranslatef(tx, ty, tz);
glRotatef(ang, rx, ry, rz);
glScalef(sx, sy, sz);
```

donde los parámetros son obvios excepto para la rotación, cuyo ángulo es `ang`, en grados sexagesimales, alrededor de una recta *por el origen* definida por el vector $r = \{\texttt{rx},\texttt{ry},\texttt{rz}\}$ 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 desde el eje al origen, rotar y volver a la posición original.

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: $a = a * m$ o, en forma detallada: $a[i][j] = \sum_k (a[k][j] * m[i][k])$. El producto es el correcto, pero los índices $[columna][fila]$ están al revés de lo usual y que se aprendió

en álgebra: $a_{ji} = \sum_k a_{jk} m_{ki}$, donde el primer subíndice es la fila y el segundo la columna. Abajo a la izquierda se muestra la composición para un punto P, transformado por las matrices A a C, a la derecha se utilizan, como ejemplo, una rotación R y un escalado no–uniforme S.

$\hat{P}$ = C(B(A(P))) = Z(P)
Z = C B A
C B A = C (B A) = (C B) A
C B A ≠ B C A ≠ A B C



4. Resumen

Resumiendo, hay tres operaciones básicas que alteran la matriz actual o activa:

- Reemplazo por restauración (`glPushMatrix`, `glPopMatrix`).
- Reemplazo por carga (`glLoadIdentity`, `glLoadMatrix`).
- Composición por multiplicación (`glMultMatrix`, `glRotate`, `glTranslate`, `glScale`).

Todas estas operaciones las hace OpenGL, en la placa gráfica. Pero si se van a utilizar *shaders*, conviene balancear el uso de la CPU y la GPU para que no se repitan cálculos innecesarios para muchas primitivas, vértices (muchos más) o fragmentos (muchísimos más aun); cuando todos utilizan las mismas matrices, conviene hacer los productos de matrices en la CPU y pasar solamente el resultado a la GPU. Hay bibliotecas como OpenGL Mathematics (GLM), que son especialmente útiles para esas tareas.

5. Dispositivo de salida y ventana

La ventana gráfica, para la imagen final en la pantalla, se define al inicializar el subprograma gráfico. Desde *GLUT* (no lo maneja *OpenGL*) se pide una ventana gráfica al sistema operativo:

```
glutInitWindowSize(Wp,Hp);
glutInitWindowPosition(Xp,Yp);
```

los parámetros son números enteros que representan la posición y el tamaño de la ventana inicial del programa, medidas en píxeles en el sistema del dispositivo (**device coordinates**) donde se vaya a renderizar (monitor).

Hay que recordar que para glut, *para el sistema operativo y para las imágenes el origen es el borde superior izquierdo* $\{0, 0\}$ en *upper-left*; x hacia la derecha; y hacia abajo.

Esta primera llamada, que mapea la ventana del programa en el dispositivo de salida, es la última transformación que sufre la escena; pero no es de *OpenGL* sino del SO. Esta transformación se define únicamente en la inicialización (notar el `glutInit...`). Luego, el sistema operativo permitirá mover y cambiar el tamaño de la ventana; los cambios de tamaño generan interrupciones y eventos (*resize*) que disparan *callbacks* apropiados de glut (o cualquier otra GUI) que el programa deberá atender.

La penúltima transformación ya sí es de *OpenGL*; adapta 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 la posición y tamaño del rectángulo en píxeles. Pero, *para* OpenGL*, el origen de la ventana del programa es el píxel inferior izquierdo* (*lower-left*).

Si `glViewport` no está presente, se considera que es toda la ventana del programa `(0,0,Wp,Hp)`.

Puede haber más de un *viewport* en un mismo programa y 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 el *callback* que maneja el *resize*. El programador decide como cambiar las ventanas de dibujo o información cuando el usuario altera la ventana del programa.

Las coordenadas (*números reales*) dentro de cada *viewport* se denominan **window coordinates** y tienen su origen en el píxel *lower-left* del *viewport*. Si bien lo que se visualiza es una imagen (2D), este es un sistema 3D izquierdo, con z entre cero (plano *near*) y uno (plano *far*).

6. Matriz de Proyección

Después de definir el *viewport* se define la matriz de proyección, que pese a su nombre 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):

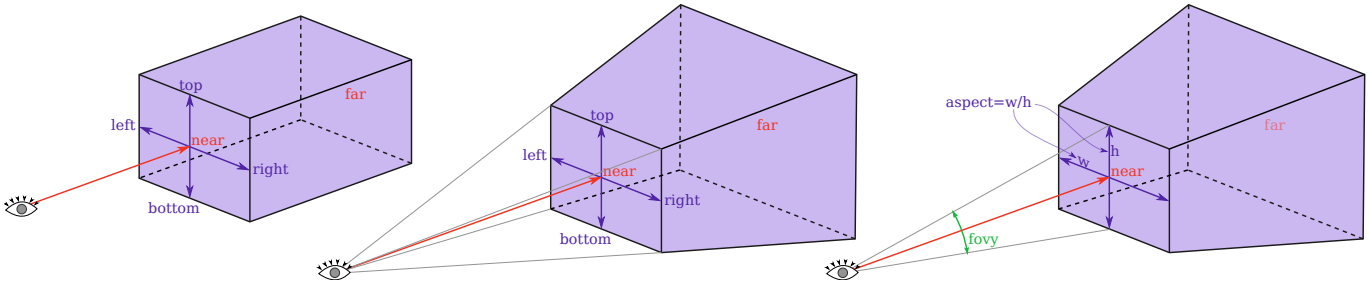
```
glOrtho(left, right, bottom, top, near, far);
```

- 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 seis planos de recorte o **clipping planes** (*clipping* = poda, recorte), que limitan el volumen visual o **clipping volume**. 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* (positivos, aun cuando el z es negativo).

El campo visual es un paralelepípedo para la proyección ortogonal y una pirámide truncada o *frustum* en latín ~ base de estatua) en el caso de la proyección perspectiva.



Todo lo que esté por fuera de ese recinto no se visualiza (*culling* = descarte) y los objetos que lo atraviesan serán recortados (*clipping* = poda), solo se verá la parte interior.

La rutina que define los planos, en realidad multiplica la matriz de proyección activa (normalmente la identidad) por una que transforma esos seis planos en los límites de un cubo entre -1 y $+1$. Podemos decir que la matriz de proyección 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), en una ortogonal no es necesario.

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 partes 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*, pero necesariamente centrado, es una función de la biblioteca *GLU*:

```
gluPerspective(fovy, aspect, near, far);
```

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

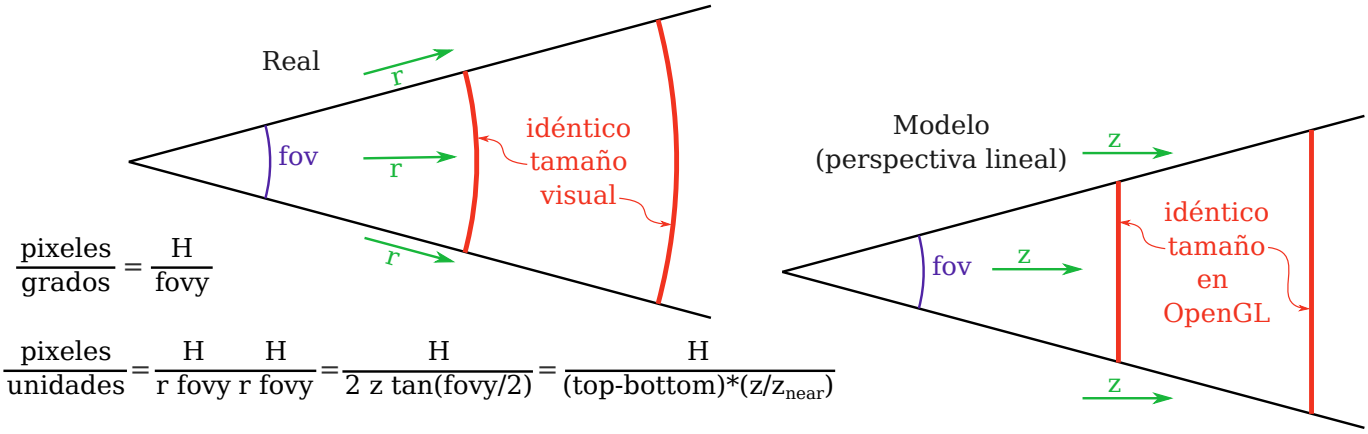
Definido el modelo de cámara, *OpenGL* realiza los recortes y descartes en estas clip-coordinates 4D y luego, para los vértices interiores, se realiza la división por la coordenada w (mal llamada *división perspectiva*). El volumen visual queda así mapeado en un cubo $[-1,1]^3$ (**NDC** o **Normalized Device Coordinates**) conservando la profundidad, para poder realizar el ocultamiento de líneas al rasterizar. En el *NDC*, la coordenada z vale -1 en el plano *near* y $+1$ en el plano *far*, por lo que es un sistema izquierdo.

7. Escala visual

El volumen visible fue definido en unidades espaciales del modelo y, dado que se cuenta con las dimensiones en pixeles del *viewport*, entonces ya está determinada la escala visual en píxeles/unidad.

La proyección ortogonal es un mapeo lineal, incluyendo a z; por lo tanto, la escala $H/(top-bottom)$ y la precisión numérica de z (32 bits para repartir entre z_{near} y z_{far}) son independientes de la profundidad.

En perspectiva, aparecen algunos detalles que le son propios: 1) *OpenGL* aproxima la perspectiva real por un modelo lineal, donde la escala depende de z y no de la distancia r al ojo. 2) Al mapear el *frustum* en un cubo, los objetos alejados se comprimen más que los cercanos y se ven más chicos. 3) Si se analizara la matriz de transformación perspectiva, se vería que la coordenada z no se mapea linealmente, sino como $a + b/z$ (a y b dependen de z_{near} y z_{far}); de modo que el *depth−buffer*, que se utiliza para determinar la oclusión visual, es más preciso cerca del ojo.



Las ecuaciones de arriba muestran la escala en función de z. Normalmente, el programador define una ventana visual de modo que quepan los objetos que quiere mostrar y de ellos obtiene el z para definir la escala visual inicial: cuantos píxeles quiere que ocupen los objetos de interés.

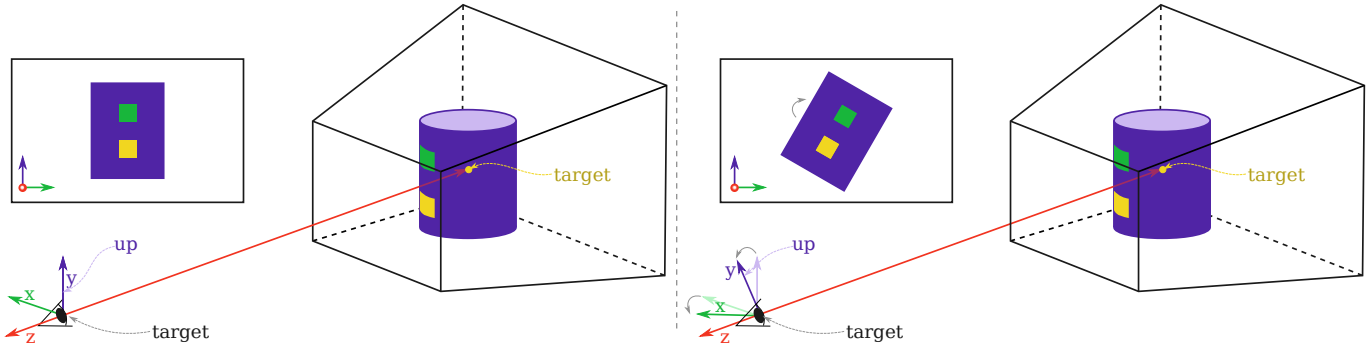
8. Composición de la escena – Matriz model-view

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 de cada objeto (por ej.: una rueda se dibuja centrada en el origen y circular en x,y), pero luego son ubicados y orientados en el espacio del modelo y finalmente se transforman todas las coordenadas al espacio visual de la cámara.

Se comienza con la llamada de definición del estado inicial para 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 ubicar los objetos que se mueven junto con la vista; por ej.: el arma de un jugador en primera persona o la luz de una lámpara que se mueve con la cámara (un flash).



La matriz de proyección definió el espacio visible; ahora hay que ubicarlo en la escena. Hay dos formas de hacerlo: una es ubicar la cámara (el sistema de coordenadas del ojo) en el espacio del modelo, con una sentencia `gluLookAt()` de *GLU* y la otra es ubicar el espacio del modelo en el sistema del ojo (actual) mediante traslaciones y rotaciones. La primera es mucho más intuitiva:

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

los parámetros son coordenadas en el sistema del modelo, la posición del ojo o cámara (*eye*), el punto al que se mira (*center* o *target*) que no es punto definido sino un vector; 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 (imagen) o la rotación de la cámara alrededor del eje *eye-target*.

Esos datos parecen construir el sistema de referencia visual, con origen en el ojo y los tres versores base, pero en realidad es al revés: aquí se construye el espacio del modelo y pasa a estar activo a partir de ahora. A partir de aquí se posicionan las luces *fijas al piso* y la mayoría de los objetos de la escena.

El *stack* de matrices permite armar modelos complejos utilizando partes pre ensambladas. Cuando se pre ensambla una pieza en una rutina, se *cargan* (se leen o definen los vértices) y se mueven algunos objetos (probablemente pre ensamblados en otra rutina). Ejemplo, auto:

```
Posicionarse en el sistema global, donde se ubicará un auto
Dibujar auto: {
    Push (guarda matriz auto)
    Mover el sistema de coordenadas al centro de rueda delantera izquierda
    Dibujar rueda: {
        Push (guarda matriz rueda di)
        Mover el sistema de coordenadas al centro del bulón 1
        Dibujar bulon 1: {...}
        Pop (vuelve matriz rueda di)
        ... (bulones 2 y 3)
```

```
            Push (guarda matriz rueda di)
            Mover el sistema de coordenadas al centro del bulón 4
            Dibujar bulon 4: {...}
            Pop (vuelve matriz rueda di)
            Dibujar llanta
            Dibujar cubierta
        }
        Pop (vuelve matriz auto)
        ...
        Push (guarda 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 sus piezas que podrían estar en rutinas sueltas. Por eso es *indispensable que antes de salir, cada rutina de dibujo* deje las cosas como estaban*; ya sean las matrices o el resto del estado (*push *y* pop attributes*) que se modifiquen dentro de la rutina*.

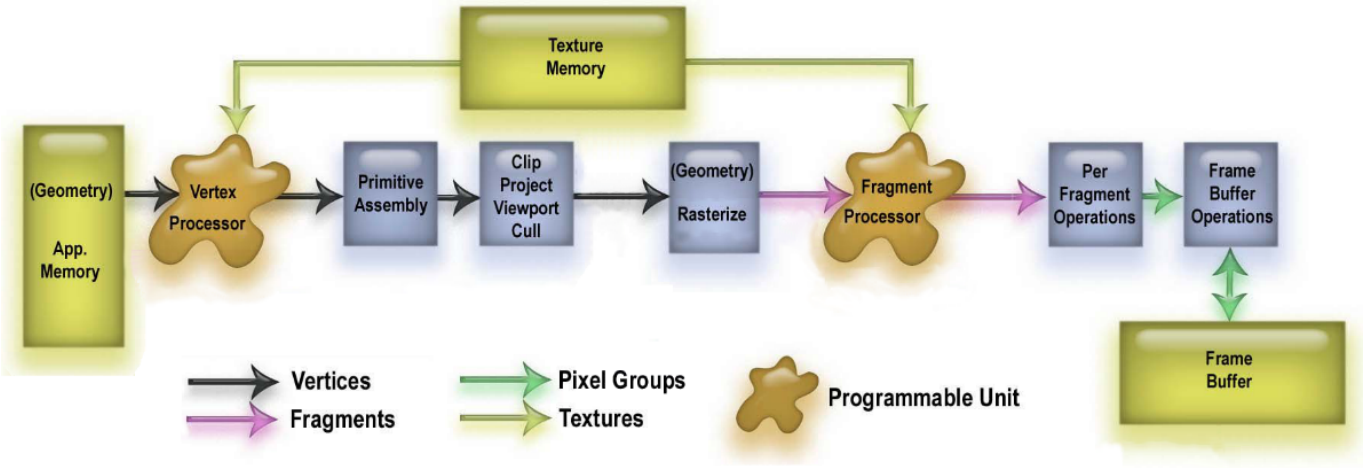
El modelado o dibujo 3D en sí, suele realizarse en programas de CAD o cualquier otro software especializado, para luego dejar que *OpenGL* posicione o mueva los objetos (*B-rep*) leídos de un archivo de intercambio. También puede realizarse directamente en *OpenGL*, pero con muchísimas limitaciones, componiendo primitivas básicas con algunas piezas simples de alto nivel provistas por glu o glut.

El *stack* de matrices es vital para realizar movimientos jerárquicos, como por ejemplo un brazo que gira con una mano que se mueve y en ella dedos que también se mueven, el brazo arrastra la mano y esta a los dedos, pero todos tienen movimientos individuales definidos por el programa. Un ejemplo más sencillo es la rueda que gira respecto del auto en movimiento.

Por cuestiones de performance, es siempre preferible deshacer las acciones que usar los *stacks* de matrices o atributos. En nuestras simples aplicaciones de ejemplo no hay problemas.

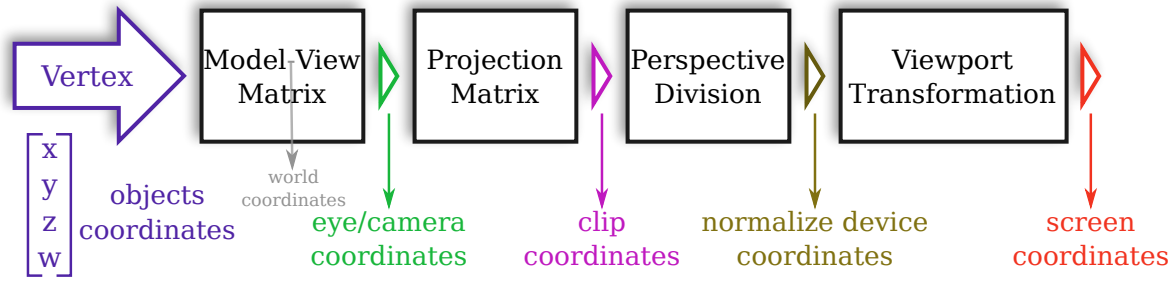
9. Transformaciones en el Pipeline

Toda la escena (modelo, luces y cámara) está definida en un espacio arbitrario, el del modelo, con posiciones y direcciones dadas en coordenadas homogéneas (4D). La matriz *model-view* (model→view) es la primera transformación que se aplica a los vértices que llegan a la GPU (1), para pasarlos al sistema visual con las *eye coordinates*: el eje x hacia la derecha, y hacia arriba y z hacia atrás del ojo. Es decir que al principio del *pipleline* los vértices ya tienen coordenadas en el espacio del ojo.



En la etapa indicada en (2) se realizan, varias transformaciones sucesivas. Primero actúa la matriz de proyección y las coordenadas pasan al *clip space* con *clip coordinates*. Es aquí donde se recortan (*clip*) las primitivas que atraviesan alguno de los seis *clipping planes* y se descarta (*culling*) todo lo que quede fuera del rango. Este espacio es aún 4D, los vértices que sobreviven son aquellos que tienen coordenadas homogéneas $\{wx, wy, wz, w\}$ donde x, y y z están entre -1 y $+1$. La coordenada z cambia de dirección (el espacio tiene orientación izquierda) y no varía linealmente con la coordenada z visual, pero es monótona ($a + b/z$), de modo que sirve para saber que fragmento tapa visualmente a otro.

Luego se procede a la (mal llamada) división perspectiva: se dividen las coordenadas por w con lo cual los vértices quedan 3D y dentro de un cubo canónico $[-1, 1]^3$, cuyas coordenadas son las *normalized evice coordinates* (NDC).



Conociendo el tamaño de la ventana gráfica (`glViewport`) se acomodan los límites al ancho y alto enteros del *viewport*. Estamos ahora en el *window–space*. Los valores de x e y son coordenadas de píxel para los buffers y el valor de z para el *depth−buffer*, varía ahora entre 0 en el plano *near* y 1 en el *far*.

Para todo lo que sigue en el *pipeline*, cada fragmento ya tiene asignada una posición x, y invariante y que correspondiente a un píxel del *viewport*; pero el *fragment shader* (3), si estuviese programado, podría alterarle la coordenada *z* para lograr algún efecto especial.